PYTHON SCRIPTS FOR ABAQUS

LEARN BY EXAMPLE

Gautam Puri

Dedicated to Mom

First Edition 2011

Printed in the United States of America

Book website: www.abaquspython.com

Contents

Preface

If you're reading this, you've probably decided to write a Python script to run an FEA analysis in Abaqus. But you're not sure where to begin, you've never written a working script for Abaqus, and you've never worked with the programming language Python either. The good news is you've found the right book to deal with the situation. Through the course of this text you're going to learn the basics of writing scripts for Abaqus and understand the working of the Abaqus Scripting Interface. At the same time you're going to learn what you need to know of the Python programming language itself. You're going to receive the stable foundation you need so that you spend more time focusing on your research and less time debugging code.

The aim of this book is not to teach you every single built-in scripting method offered by Abaqus. There are literally hundreds of these, and chances are you will only use a few of them for your own simulations. We'll focus on these, and put you in a position where you can branch out on your own. For the record all the keywords and methods of the Abaqus Scripting Interface are listed in the Abaqus Scripting Reference Manual. The documentation also consists of a manual called the Abaqus Scripting User's Manual which provides helpful advice on different scripting topics. You could potentially learn to write Abaqus scripts in Python from the documentation itself, as many people (such as me) have had to do in the past. But as a beginner you will likely find yourself overwhelmed by the sheer quantity of information provided there. You will spend a lot of time making avoidable mistakes and discovering for yourself, after hours (or days or months) of trial and error, the correct method to accomplish a given task through a script. This book gives you the guidance you need to start writing complex scripts right off the bat. Once you've read through all the pages you will have the knowledge and the confidence to write your own scripts for finite element simulations in Abaqus, and will then be able to refer to the Abaqus documentation for more information specific to your research task.

Why write scripts?

If you plan to learn scripting in Abaqus chances are you already know why it is useful and intend to use it to accomplish some task for your analyses. But for the sake of completeness (and for those of you who are reading because your professor/boss forced you to), a few uses shall be mentioned.

Let's assume you regularly use a few materials in all your simulations. Every time you start a new simulation in the GUI mode (Abaqus/CAE) you need to open up the materials editor and enter in material properties such as the Density, Young's Modulus, and Poisson's Ratio and so on for each of these materials. You could instead put all of these materials in a script. Then all you would need to do is go to *File > Run Script...* and your material database would be populated with these materials in a couple of seconds. Basically you would be using the script to perform a repetitive task to save time. That is the one use of a script, to perform the same task the same way multiple times with minimal effort. We will in fact look at this example of creating materials with a script in the first chapter.

A more complex use of a script is if you have a certain part on which you plan to apply loads and boundary conditions, and you wish to change the loads, constraints, or the geometry of the part itself and rerun the simulation numerous times to optimize the design. Let's assume for example you apply a load on a horizontal cantilevered beam and you want to know how much the beam bends as you increase its length. One way to do this would be to recreate the beam part 7 or 8 times. If your simulation has complex parameters you might have to apply sections, loads and constraints to it every time. A more sophisticated and efficient way to accomplish the same task is to write a script with the length of the beam assigned to a variable. You could then change the value of this variable and rerun the script in a loop as many times as you need to. The script would redraw the beam to the new length and apply the loads and BCs in the correct regions (accounting for the change in location of loads and BCs with the geometry). While this may sound like too much work for a simple beam simulation, if you have a more complex part with multiple dimensions that are all related to each other then remodeling it several times will prove to be very time consuming and a script will be the wise choice.

An added advantage of a script is that you have your entire simulation setup saved in the form of a small readable text file only a few kilobytes in size. You can then email this text file to your coworker and all he would need to do is run this script in Abaqus. It would redraw the part, apply the materials, loads, boundary conditions, create the steps, and even create and run the job if programmed to do so. This also has the advantage of readability. If a coworker takes over your project, he does not need to navigate through the model tree to figure out how you created the complex geometry of your part file, or what points and edges you applied each load or boundary condition on. He only needs to open up the script file and it's all clearly spelled out. And you can put comments all over the script to explain why you did what you did. It keeps things compact and easy to follow.

What you need...

This book assumes that you have some previous experience with running simulations in Abaqus in the GUI (Abaqus/CAE). This means you know how to set up a basic simulation, create parts, enter material properties, assign sections, apply forces and boundary conditions, create interactions, mesh parts and run jobs by using the toolbars or menus in Abaqus/CAE. When we start learning to write scripts you will essentially be performing all of these same procedures, except in the form of Python code.

However you do not need to be an expert at these tasks. For every example we work on, we first look at the procedure to be carried out in the Abaqus/CAE. This procedure has been spelled out in the text, and is also demonstrated as silent video screencasts where you can watch me perform the analysis step by step. This is to ensure that you know how to perform the task in the GUI itself, before trying to write a script. These screencasts have been posted on the book website www.abaquspython.com (and hosted on YouTube) where I've found they are also being used by beginners trying to teach themselves Abaqus. Following the creation of these videos, I was employed by Dassault Systèmes Simulia Corp. to create an Abaqus tutorial series on their new 'SIMULIA Learning Community'. I have recorded audio narration with detailed explanation over all of these, and other newer tutorials as well. These are currently displayed (free) at www.simulia.com/learning. If you wish to brush up on your Abaqus skills you may watch these. Refer to the book website for up-to-date information and links.

The book assumes that you have some basic knowledge of programming. This includes understanding concepts like variables, loops (for, while) and if-then statements. You are all set if you have experience with languages such as C, C++, Java, VB, BASIC etc. Or you might have picked up these concepts from programmed engineering software such as MATLAB or Mathematica.

In order to run the example scripts on your own computer you will need to have Abaqus installed on it. Abaqus is the flagship product of SIMULIA, a brand of Dassault

Systèmes. If you have Abaqus (research or commercial editions) installed on the computers at your workplace you can probably learn and practice on those. However not everyone has access to such facilities, and even if you do you might prefer to have Abaqus on your personal computer so you can fiddle around with it at home. The good news is that the folks at SIMULIA have generously agreed to provide readers of this book with Abaqus Student Edition version 6.10 (or latest available) for free. It can be downloaded off the book website. This version of Abaqus can be installed on your personal computer and used for as long as you need to learn the software. There are a few minor restrictions on the student edition, such as a limitation on the number of nodes (which means we will not be able to create fine meshes), but for the most part these will not hinder the learning experience. For our purposes Abaqus SE is identical to the research and commercial editions. The only difference that will affect us is the lack of replay files but I'll explain what those are and how to use them so you won't have any trouble using them on a commercial version. Abaqus SE version 6.9 and version 6.10 were used to develop and test all the examples in this book. The Abaqus Scripting Interface in future versions of Abaqus should not change significantly so feel free to use the latest version available to you when you read this.

How this book is arranged...

The first one-third of this book is introductory in nature and is meant to whet your appetite, build up a foundation, and send you in the right direction. You will learn the basics of Python, and get a feel for scripting. You'll also learn essential stuff like how to run a script from the command line and what a replay file is.

The second part of the book helps you 'Learn by Example'. It walks you through a few scripting examples which accomplish the same task as the silent screencasts on the book website but using only Python scripts. Effort has been taken to ensure each example/script touches on different aspects of using Abaqus. All of these scripts create a model from start to finish, including geometry creation, material and section assignments, assembling, assigning loads, boundary conditions and constraints, meshing, running a job, and post processing. These scripts can later be used by you as a reference when writing your own scripts, and the code is easily reusable for your own projects. Aside from demonstrating how to set up a model through a script, the later chapters also demonstrate how to run optimization and parametric studies placing your scripts inside loops and varying parameters. You also get an in-depth look into extracting information from output databases, and job monitoring.

The last part of the book deals with GUI Customization – modifying the Abaqus/CAE interface for process automation and creating vertical applications. It is assumed that you have no previous knowledge of GUI programming in general, and none at all with the Abaqus GUI Toolkit. GUI Customization is a topic usually of interest only to large companies looking to create vertical applications that perform repetitive tasks while prompting the user for input and at the same time hiding unnecessary and complex features of the Abaqus interface. Chances are most readers will not be interested in GUI Customization but it has been included for the sake of completeness and because there is no other learning resource available on this topic.

Acknowledgements

I would like to thank my mother for giving me the opportunity to pursue my studies at a great expense to herself. This book is dedicated to her. I would also like to thank my father and my grandmother for their love, support and encouragement.

I'd like to thank my high school Physics teacher, Santosh Nimkar, for turning a subject I hated into one I love. The ability to understand and predict real world phenomena using mathematics eventually led me toward engineering.

I'd like to extend a special thank you to Rene Sprunger, business development manager at SIMULIA (Dassault Systèmes Simulia Corporation) for his support and encouragement, without which this book might never have materialized. I'd also like to thank all the professionals at SIMULIA for developing the powerful realistic simulation software Abaqus, and for creating the remarkable Abaqus Scripting Interface to enhance it.

PART 1 – GETTING STARTED

The chapters in Part 1 are introductory in nature. They help you understand how Python scripting fits into the Abaqus workflow, and explain to you the benefits and limitations of a script. You will learn the syntax of the Python programming language, which is a prerequisite for writing Abaqus scripts. You will also learn how to run a script, both from within Abaqus/CAE and from the command line. We'll introduce you to replay files and macros, and help you decide on a code editor.

It is strongly recommended that you read all of these chapters, and do so in the order presented. This will enhance your understanding of the scripting process, and ensure you are on the right track before moving on to the examples of Part 2.

1

A Taste of Scripting

1.1 Introduction

The aim of this chapter is to give you a feel for scripting in Abaqus. It will show you the bigger picture and introduce you to idea of how a script can replace actions you would otherwise perform in graphical user interface (GUI) Abaqus/CAE. It will also demonstrate to you the ability of Python scripts to perform just about any task you can perform manually in the GUI.

1.2 Using a script to define materials

When running simulations specific to your field of study you may find yourself reusing the same set of materials on a regular basis. For instance, if you analyze and simulate mostly products made by your own company, and these contain a number of steel components, you will need to define the material steel and along with its properties using the materials editor every time you begin a new simulation. One way to save yourself the trouble of defining material properties every time is to write a script that will accomplish this task. The Example 1.1 demonstrates this process.

Example 2.1 – Defining materials and properties

Let's assume you often use Titanium, AISI 1005 Steel and Gold in your product. The density, Young's Modulus and Poisson's Ratio of each of these materials is listed the following tables.

Properties of Titanium

Property	Metric	English
Density	4.50 g/cc	0.163 lb/in^3
Modulus of Elasticity	116 GPa	16800 ksi
Poisson's Ratio	0.34	0.34

Properties of AISI 1005 Steel

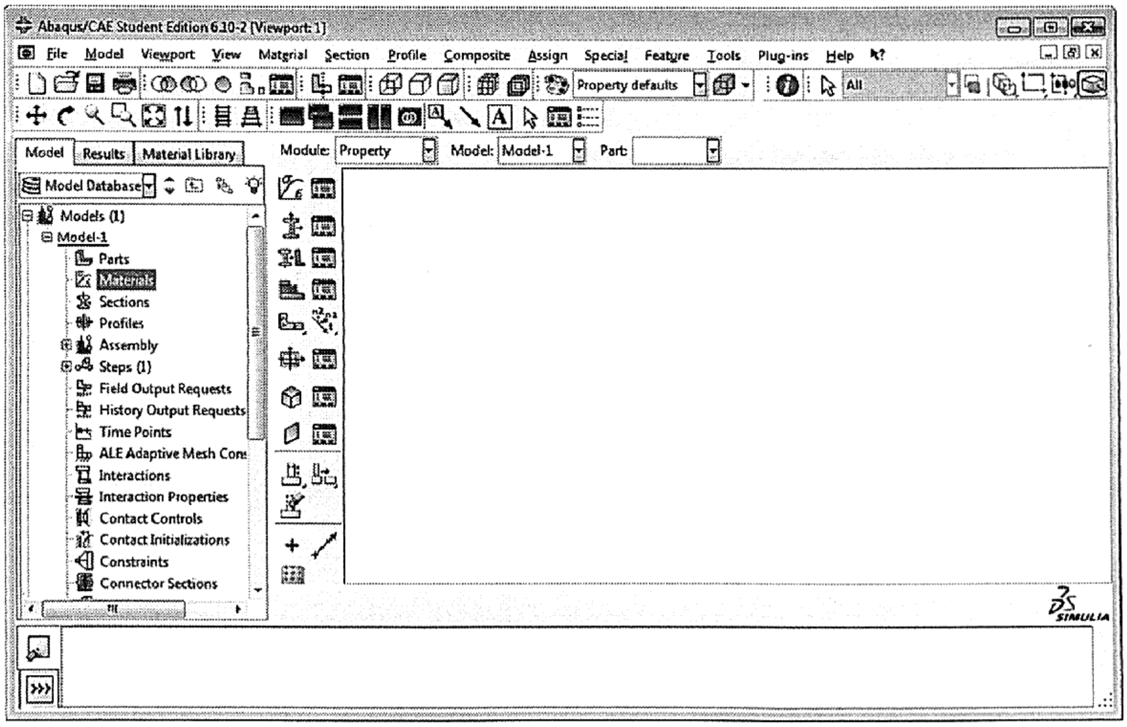
Property	Metric	English
Density	7.872 g/cc	0.2844 lb/in^3
Modulus of Elasticity	200 GPa	29000 ksi
Poisson's Ratio	0.29	0.29

Properties of Gold

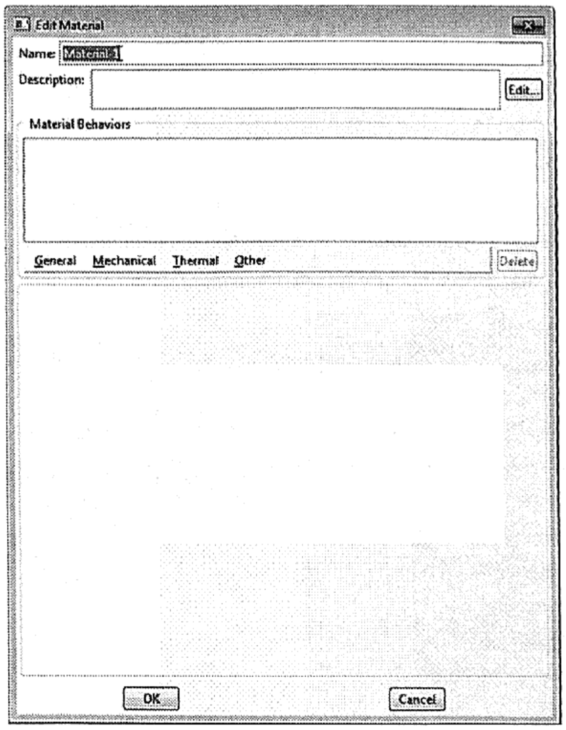
Property	Metric	English
Density	19.32 g/cc	0.6980 lb/in^3
Modulus of Elasticity	77.2 GPa	11200 ksi
Poisson's Ratio	0.42	0.42

Let's run through how you would usually define these materials in Abaqus CAE.

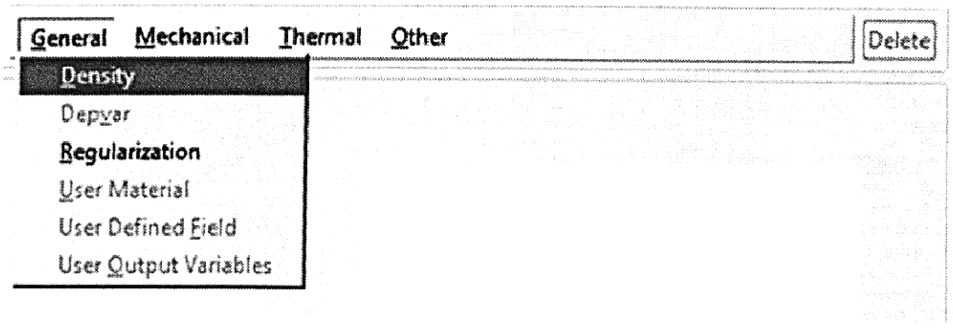
1. Startup Abaqus/CAE
2. If you aren't already in a new file click **File > New Model Database > With Standard/Explicit Model**
3. You see the model tree in the left pane with a default model called **Model-1**. There is no '+' sign next to the Materials item indicating that it is empty.



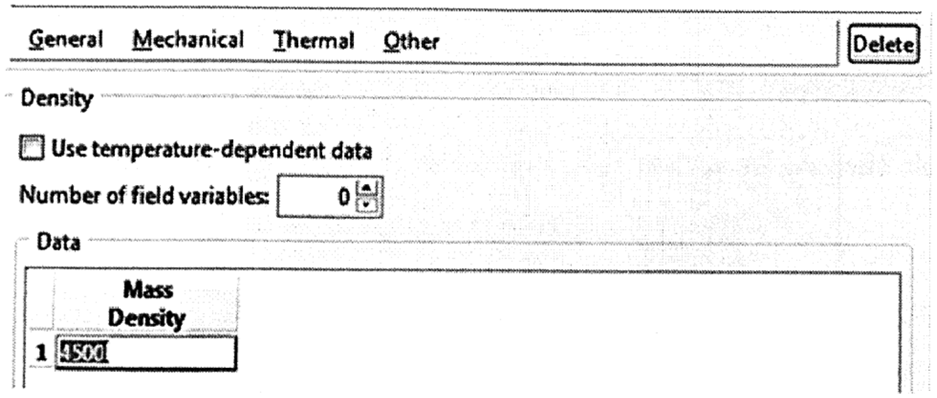
4. Double click the **Materials** item. You see the **Edit material** dialog box.



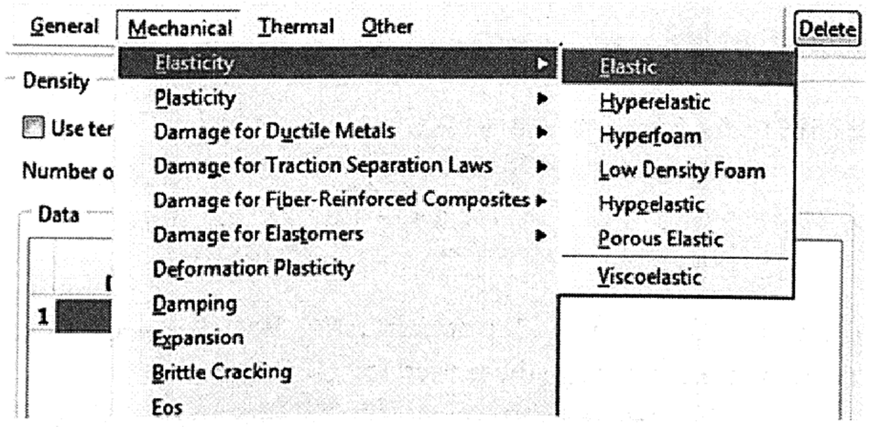
5. Name the material **Titanium**
6. Click **General > Density**.



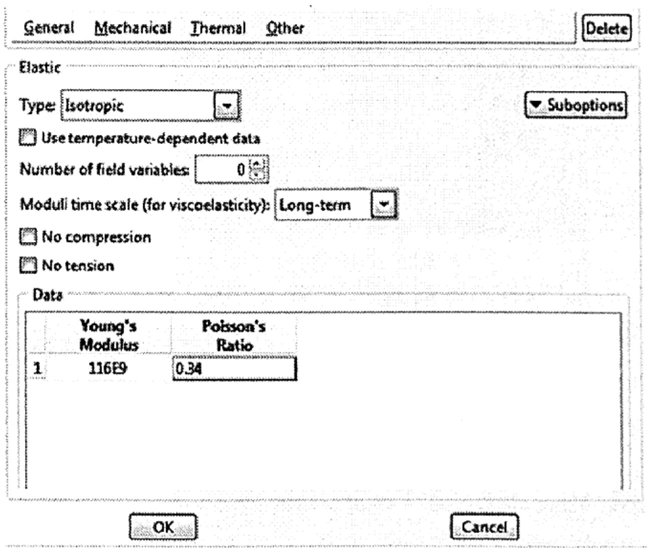
7. Let's use SI units with MKS (m, kg, s). We write the density of 4.50 g/cc as 4500 kg/m^3. Type this in as shown in the figure.



8. Then click **Mechanical > Elasticity > Elastic**



9. Type in the modulus of elasticity and Poisson's ratio. The Young's modulus of 116 GPa needs to be written as **116E9** Pa (or 116E9 N/m^2) to keep the units consistent. The Poisson's ratio of **0.34** remains unchanged.



Elastic

Type: Isotropic

☐ Use temperature-dependent data

Number of field variables: 0

Moduli time scale (for viscoelasticity): Long-term

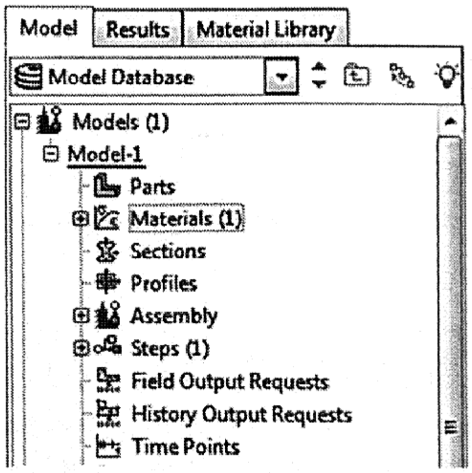
☐ No compression

☐ No tension

Data

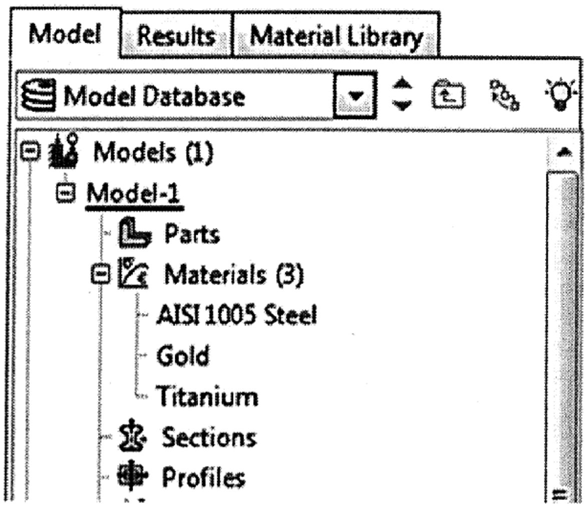
	Young's Modulus	Poisson's Ratio
1	116E9	0.34

10. Click **OK**. The material is created and the model tree on the left indicates the presence of 1 material with the number **1** in parenthesis. Clicking the '+' symbol next to it reveals the name of the material **Titanium**, and double clicking it will reopen the **Edit material** window.



11. Repeat the process for the other 2 materials, **AISI 1005 Steel** and **Gold**. Remember to keep the units consistent with those used for **Titanium.**

12. When you're done the model tree should appear as it does in the figure with the 3 materials displayed.



That wasn't too hard. You defined 3 materials and you can now use these for the rest of your analysis. The problem is that you will need to define these materials in this manner all over again whenever you open a new file in Abaqus CAE to start a new study on your products. This is a tedious process, particularly if you have a lot of materials and you define a large number of their properties. Aside from consuming time there is also the chance of typing in a number wrong and introducing an error into your simulations, which will later be very hard to spot.

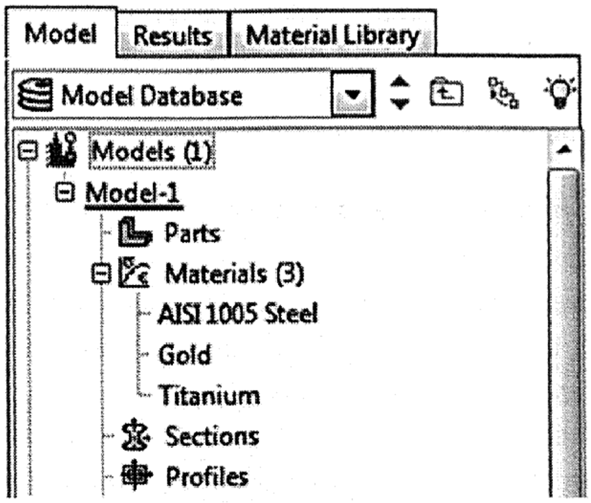
One way to fix this situation is to add your materials to the materials library. Then you could import the materials every time you created a new Abaqus file. Another way to do this would be in the form of a script. You type out the script once and place it in a file with the extension .py and every time you need these materials you go to **File > Run Script...**

Let's put a script together. Start by opening up a simple text editor. My personal favorite is Notepad++. It is free and it has got a clean interface. It also displays line numbers next to your code (making it easier to spot debugging errors) and can color code your script by auto-detecting Python from the file extension. On the other hand you may wish to use one of the Python editors from Python.org such as PythonWin. The idea is to create a simple text file, and then save it with a .py extension.

Open a new document in Notepad. Type in the following statements:

```
mdb.models['Model-1'].Material('Titanium')
mdb.models['Model-1'].materials['Titanium'].Density(table=((4500, ), ))
mdb.models['Model-1'].materials['Titanium'].Elastic(table=((200E9, 0.3), ))

mdb.models['Model-1'].Material('AISI 1005 Steel')
mdb.models['Model-1'].materials['AISI 1005 Steel'].Density(table=((7872, ), ))
mdb.models['Model-1'].materials['AISI 1005 Steel'].Elastic(table=((200E9, 0.29), ))

mdb.models['Model-1'].Material('Gold')
mdb.models['Model-1'].materials['Gold'].Density(table=((19320, ), ))
mdb.models['Model-1'].materials['Gold'].Elastic(table=((77.2E9, 0.42), ))
```

Save the file as 'ch1ex1.py'. Now open a new file in Abaqus CAE using **File > New**. Click on **File > Run Script...** The script will run, probably so fast you won't notice anything at first. But if you look closely at the **Materials** item in the model tree you will see the number 3 in parenthesis next to it indicating there are 3 defined materials. If you click the '+' sign you will see our 3 materials.



In fact if you double click on any of the materials, the **Edit Material** window will open showing you that the density and elastic material behaviors have been defined.

The script file has performed all the actions you usually execute manually in the GUI. It's created the 3 materials in turn and defined their densities, moduli of elasticity and Poisson's ratios. You could open a new Abaqus/CAE model and repeat the process of running the script and it would take about a second to create all 3 materials again.

If by chance you tried to decipher the script you just typed you may be a little lost. You see the words 'density' and 'elastic' as well as the names of materials buried within the code, so you can get a general idea of what the script is doing. But the rest of the syntax isn't too clear just yet. Don't worry, we'll get into the details in subsequent chapters.

1.3 To script or not to script..

Is writing a script better than simply storing the materials in the materials library? Well for one, it allows you to view all the materials and their properties in a text file rather than browsing through the materials in the GUI and opening multiple windows to view each property. Secondly you can make two or three script files, one for each type of simulation your routinely perform, and importing all the required materials will be as easy as **File > Run Script**. On the other hand if you store the materials in a material library you will need to search through it and pick out the materials you wish to use for that simulation each time.

At the end of the day it is a judgment call, and for an application as simple as this either method works just fine. But the purpose of this Example 1.1 was to demonstrate the power of scripting, and give you a feel for what is possible. Once you've read through the rest of the book and are good at scripting, you can make your own decision about whether a simulation should be performed with the help of a script or not.

1.4 Running a complete analysis through a script

You've seen how a script can accomplish a simple task such as defining material properties. A script however is not limited to performing single actions, you can in fact run your entire analysis using a script without having to open up Abaqus/CAE and see the GUI at all. This means you have the ability to create parts, apply material properties, assign sections, apply loads and constraints, define sets and surfaces, define interactions and constraints, mesh and run the simulations, and also process the results, all through a script. In the next example you will write a script that can do all of these things.

Example 2.2 – Loaded cantilever beam

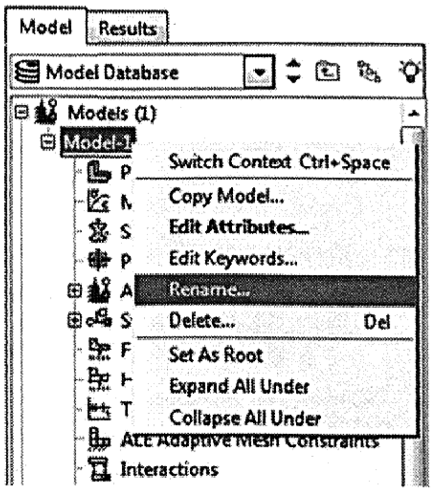
Just as in the previous example, we will once again begin with demonstrating the process in Abaqus/CAE and then perform the same tasks with a script. We're going to create a simple cantilever beam 5 meters long with a square cross section of side 0.2 m made of AISI 1005 Steel. Being a cantilever this beam will be clamped at one end. That means that it can neither translate along the X, Y or Z axes, nor can it rotate about them at that fixed end. This is also known as an encastre condition. A pressure load of 10 Pa will cause the beam to bend downwards with the maximum deflection experienced the free end.

Field output and history output data will be collected. Field output data provides information on the state of the overall system during the load step, such as the stresses and strains. Instead of using the defaults, we will instruct Abaqus to track the stress components and invariants, total strain components, plastic strain magnitude, translations and rotations, reaction forces and moments, and concentrated forces and moments. History output data provides information on the state of a smaller section such as a node at frequent intervals. For this we will allow Abaqus to track the default variables for history output.

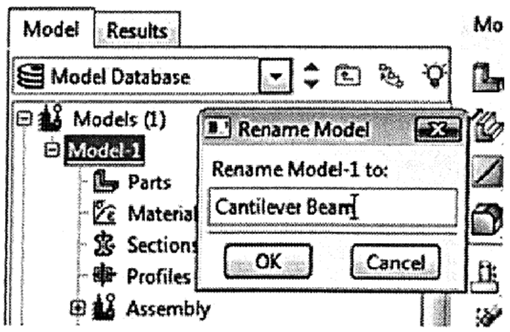
We will mesh the beam using an 8-node linear brick, reduced integration element (C3D8R) with a mesh size of 0.2. We will create a job, submit it, and inspect the results.

Let's start by performing these tasks in the GUI mode using Abaqus CAE.

1. Startup Abaqus/CAE
2. If you aren't already in a new file click **File > New**
3. In the Model Database panel right click **Model-1** and choose **Rename....**

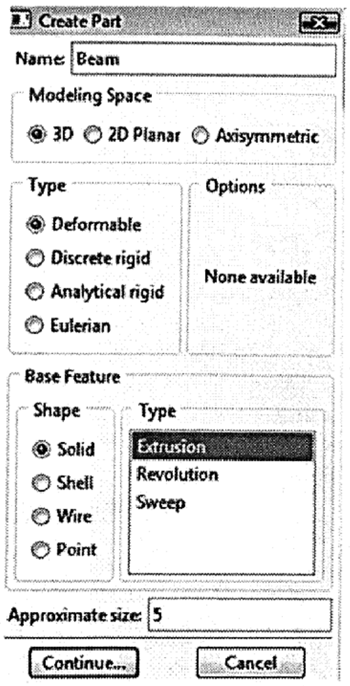


4. Type in **Cantilever Beam**. **Model-1** will change to **Cantilever Beam** in the tree.



Rename Model-1 to: Cantilever Beam

5. Double click on the **Parts** item. The **Create Part** dialog is displayed. Name the part **Beam**. In the **Modeling Space** section, choose **3D**. For the **Type** choose **Deformable**. For **Base Feature** choose **Solid** as the shape and **Extrusion** as the type. Set the **Approximate Size** to **5**. Press **Continue..**



Name: Beam

Modeling Space: [✓ 3D]

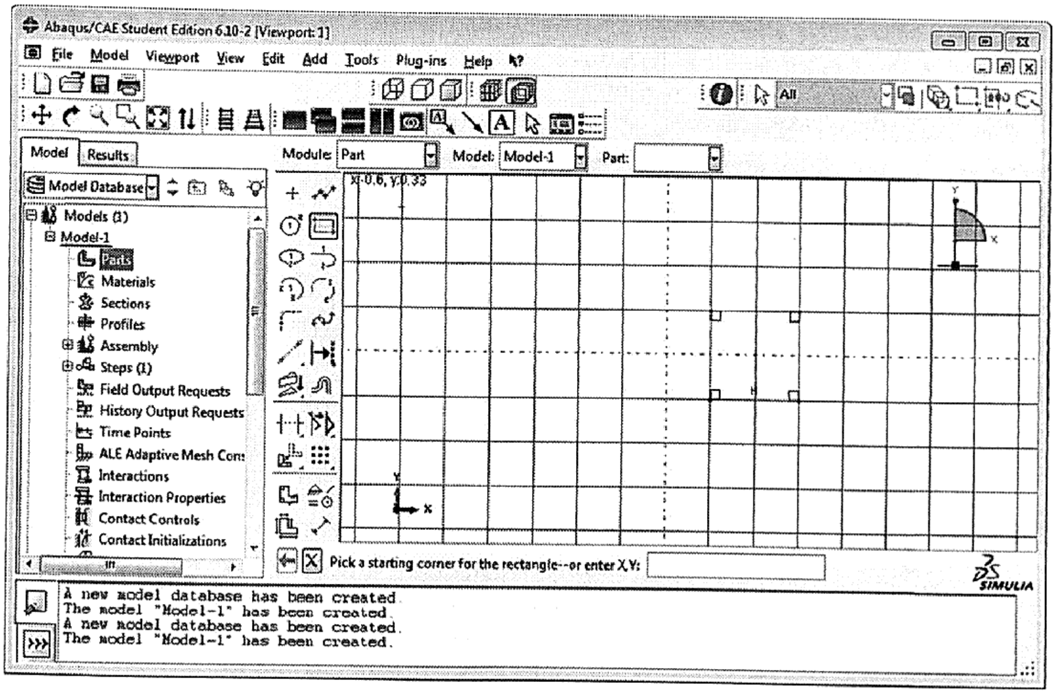
Type: [✓ Deformable]

Base Feature Shape: [✓ Solid]

Base Feature Type: [✓ Extrusion]

Approximate size: 5

6. You find yourself in the **Sketcher** window. Select the rectangle tool from the toolbar. For the first point click on (0.1, 0.1). For the second point click on (0.3, -0.1). A rectangle is drawn with these two points as the vertices.



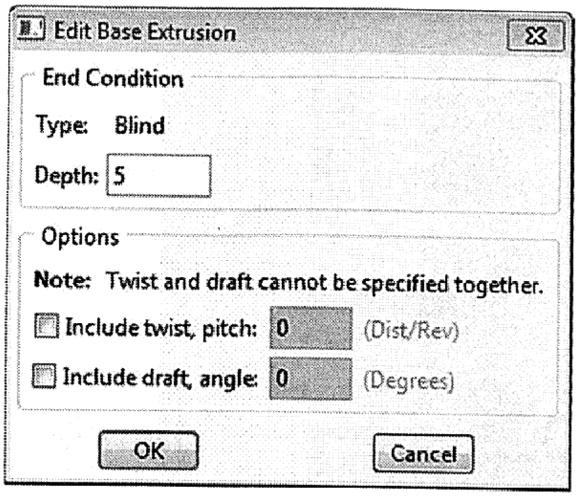
Module: Part Model: Model-1 Part:

X:-0.6, Y:0.33

Pick a starting corner for the rectangle--or enter X,Y:

```
A new model database has been created.
The model "Model-1" has been created.
A new model database has been created.
The model "Model-1" has been created.
```

7. Click the red **X** button at the bottom of the window indicating **End procedure** and then click **Done.**
8. In the **Edit Base Extrusion** window set **Depth** to **5.**



End Condition

Type: Blind

Depth: 5

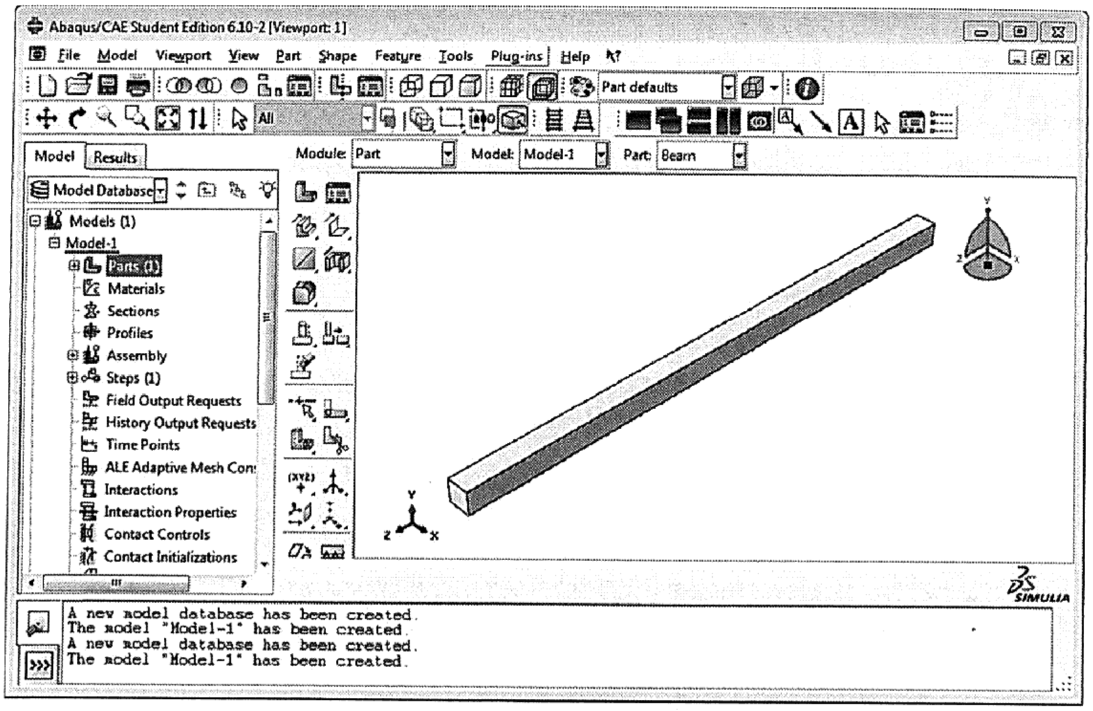
Options

Note: Twist and draft cannot be specified together.

☐ Include twist, pitch: 0 (Dist/Rev)

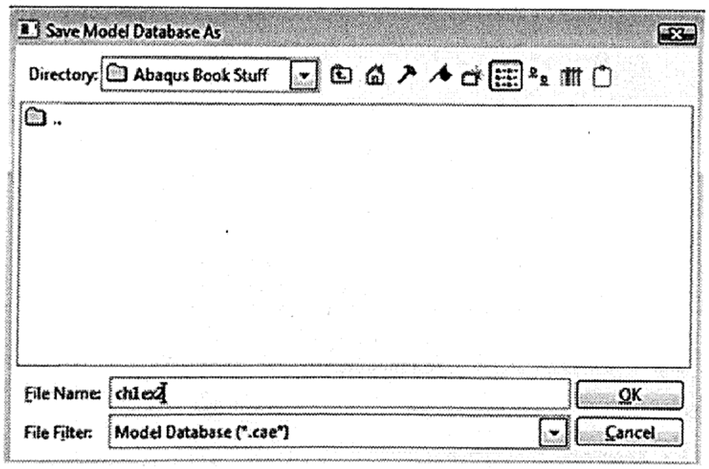
☐ Include draft, angle: 0 (Degrees)

9. Click **OK.** You will see a 3D rendering of the part **Beam** you just made. The **Parts** item in the model tree now has a sub-item called **Beam.**



```
A new model database has been created.
The model "Model-1" has been created.
A new model database has been created.
The model "Model-1" has been created.
```

10. Now would be a good time to save your file. Choose **File** > **Save**. Select the directory you save your files in and name this file 'cantilever_beam.cae' (or something more creative if you prefer)

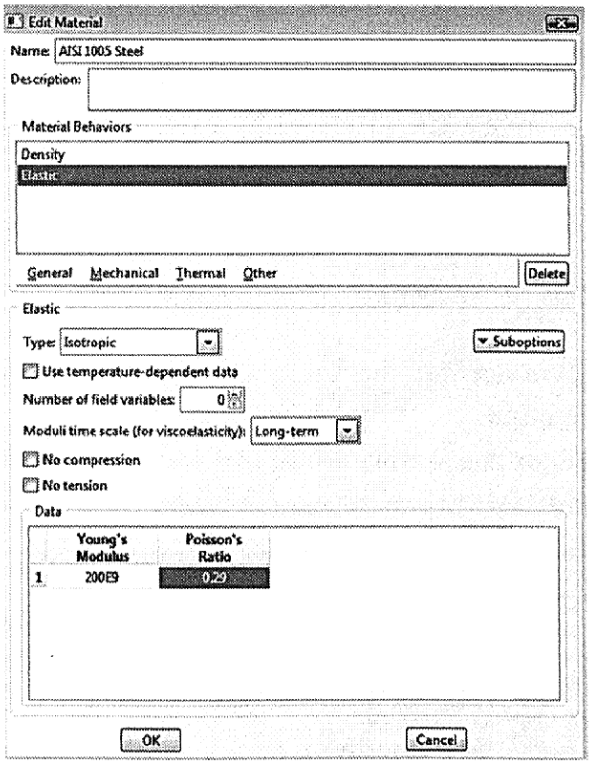


Directory: Abaqus Book Stuff

File Name: ch1ex2

File Filter: Model Database (*.cae*)

11. Double click the **Materials** item in the model tree. Name it **AISI 1005 Steel**. Set **General** > **Density** to **7872** kg/m^3. Set **Mechanical** > **Elasticity** > **Elastic** to a Young's Modulus of **200E9** N/m^2 and a Poisson's Ratio of **0.29**.


Edit Material

Name: AISI 1005 Steel

Description:

Material Behaviors

Density

Elastic

Elastic

Type: Isotropic

Use temperature-dependent data

Number of field variables: 0

Moduli time scale (for viscoelasticity): Long-term

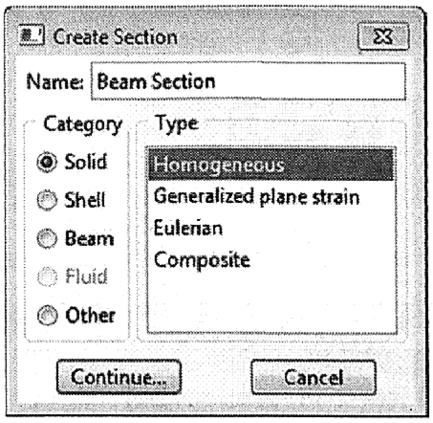
No compression

No tension

Data

	Young's Modulus	Poisson's Ratio
1	200E9	0.29

12. Click **OK**. The material is added to the model tree.
13. Double click on the **Sections** item. The **Create Section** window is displayed. Name it **Beam Section**. Set the **Category** to **Solid** and the **Type** to **Homogeneous** if this isn't already the default.

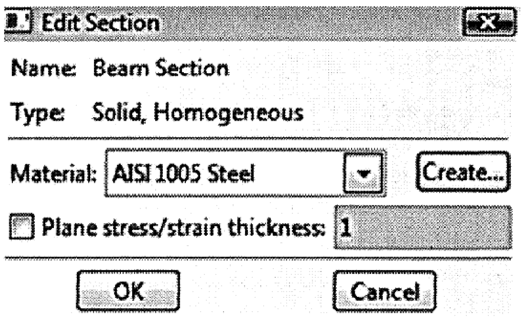

Create Section

Name: Beam Section

Category: [✓ Solid] Shell Beam Fluid Other

Type: [✓ Homogeneous] Generalized plane strain Eulerian Composite

14. Click **Continue**. The **Edit Section** window is displayed with the **Name** set to **Beam Section** and **Type** set to **Solid, Homogeneous**. Under the **Material** drop down menu choose **AISI 1005 Steel** which is the material you created a moment ago.


Edit Section

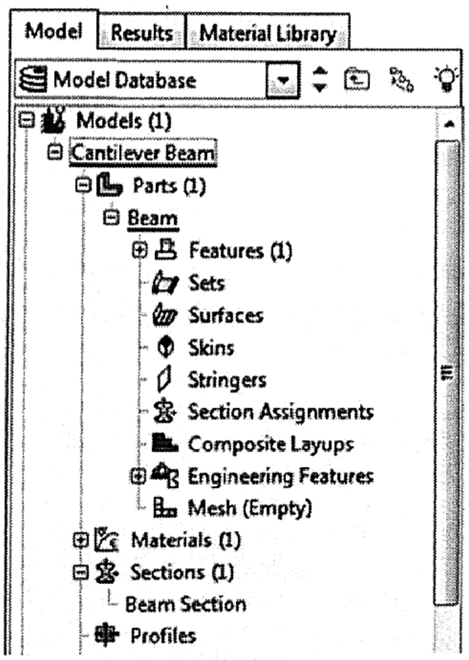
Name: Beam Section

Type: Solid, Homogeneous

Material: AISI 1005 Steel

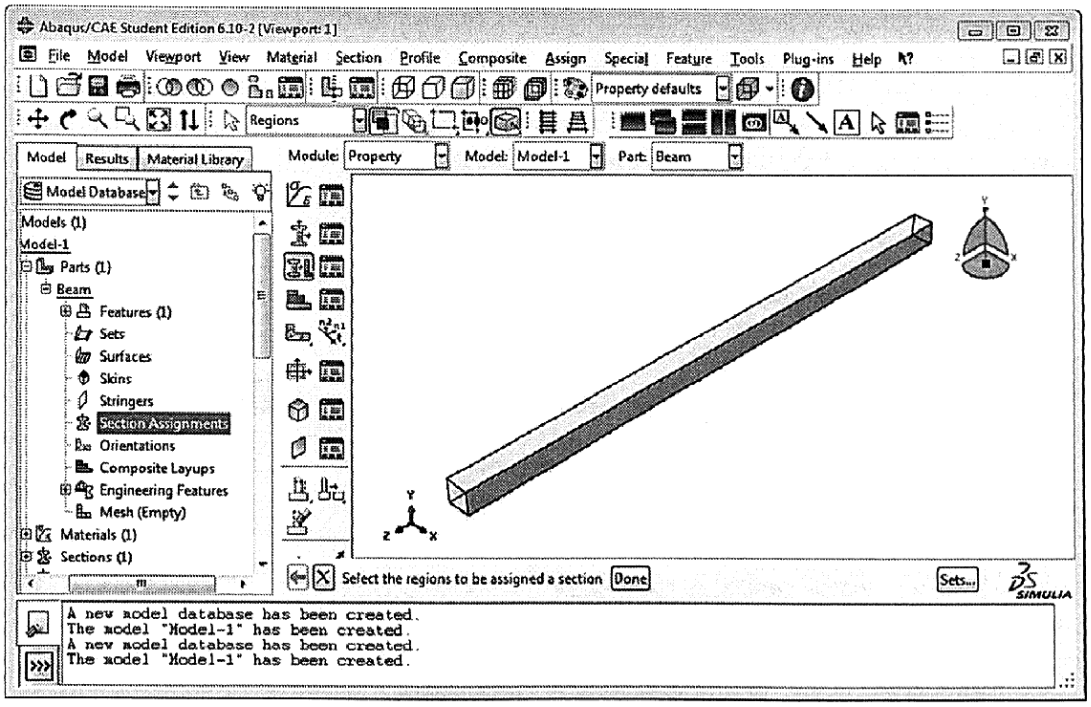
Plane stress/strain thickness: 1

15. Click **OK**. You will notice that the **Sections** item in the model tree now has a sub-item called **Beam Section.**
16. Next we need to assign this section to the part **Beam**. Expand the **Parts (1)** item by clicking the + symbol next to it to reveal the **Beam** item. Expand that too to reveal a number of sub-items such as **Features**, **Sets**, **Surfaces** and so on.



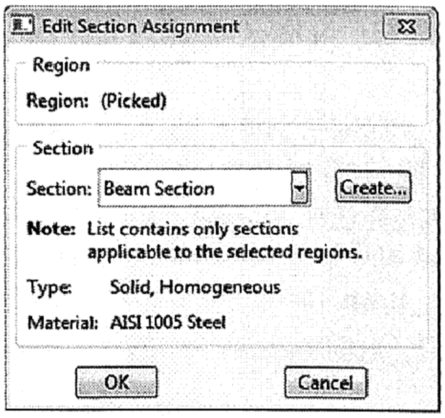
```
Models (1)
  Cantilever Beam
    Parts (1)
      Beam
        Features (1)
        Sets
        Surfaces
        Skins
        Stringers
        Section Assignments
        Composite Layups
        Engineering Features
        Mesh (Empty)
    Materials (1)
    Sections (1)
      Beam Section
    Profiles
```

17. Double click the sub-item **Section Assignments**. You will see the hint **Select the regions to be assigned a section** below the viewport. Hover your mouse over the beam in the viewport and when all its edges light up click to select it.


Abaqus/CAE Student Edition 6.10-2 [Viewport: 1]
File Model Viewport View Material Section Profile Composite Assign Special Feature Tools Plug-ins Help
Property defaults
Regions
Model Results Material Library
Module: Property Model: Model-1 Part: Beam
Model Database
Models (1)
Model-1
Parts (1)
Beam
Features (1)
Sets
Surfaces
Skins
Stringers
Section Assignments
Orientations
Composite Layups
Engineering Features
Mesh (Empty)
Materials (1)
Sections (1)
Select the regions to be assigned a section Done
Sets...
SIMULIA

```
A new model database has been created.
The model "Model-1" has been created.
A new model database has been created.
The model "Model-1" has been created.
```

18. Click **Done**. You see the **Edit Section Assignment** window. Set the **Section** to **Beam Section** which is the section you created in steps 13-15.



Region: (Picked)

Section: Beam Section

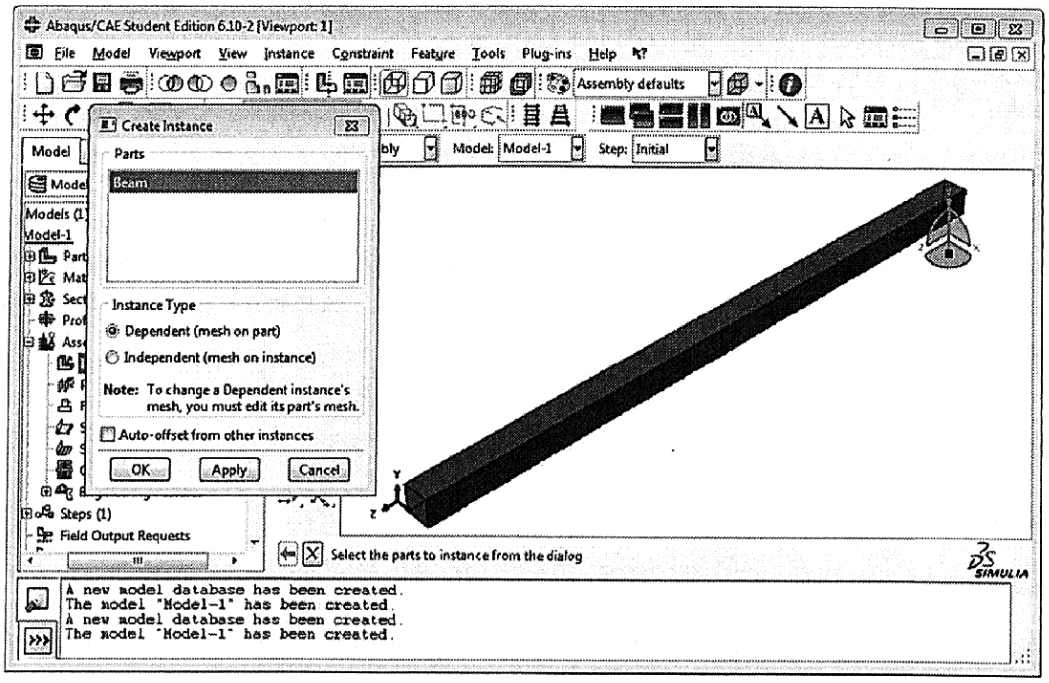
Note: List contains only sections applicable to the selected regions.

Type: Solid, Homogeneous

Material: AISI 1005 Steel

19. Click **OK**. The **Section Assignments** item now has 1 sub-item **Beam Section (Solid, Homogeneous).** The part in the viewport changes color (to green on my system) indicating it has been assigned a section.
20. Let's import the part into an assembly. Click the + symbol next to the **Assembly** item in the model tree and double-click the **Instances** sub-item. You see the **Create**

Instance window. For **Parts, Beam** is the only option available and it is selected by default. For the **Instance Type** choose **Dependent (mesh on part).**


Abaqus/CAE Student Edition 6.10-2 [Viewport: 1]
File Model Viewport View Instance Constraint Feature Tools Plug-ins Help
Assembly defaults
Model: Model-1
Step: Initial
Models (1)
Model-1
Steps (1)
Field Output Requests
OK
Apply
Cancel
SIMULIA

Create Instance

Parts: Beam

Instance Type

(● Dependent (mesh on part)) ○ Independent (mesh on instance)

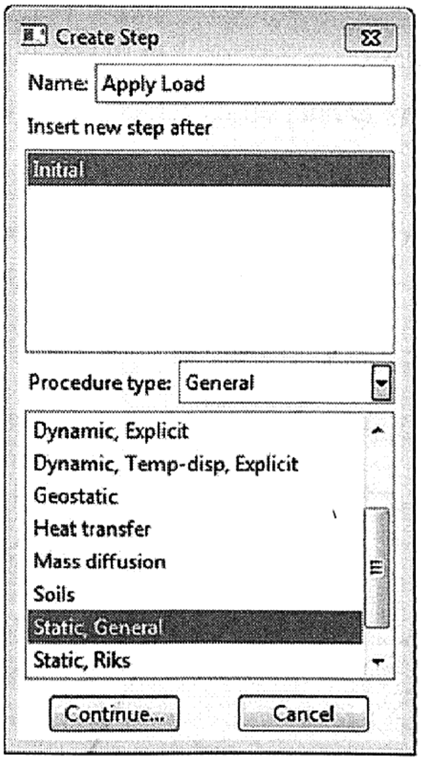
Note: To change a Dependent instance's mesh, you must edit its part's mesh.

☐ Auto-offset from other instances

Select the parts to instance from the dialog

```
A new model database has been created.
The model "Model-1" has been created.
A new model database has been created.
The model "Model-1" has been created.
```

21. Click **OK**. The **Instances** sub-item of the **Assembly** item now has a sub-item of its own called **Beam-1**. You can right-click on it and choose **Rename....** Change the name to **Beam Instance**.
22. Next we create a step in which to apply the load. Notice that the **Steps** item in the model tree already has the **Initial** step. Double-click the **Steps** item. The **Create Step** window is displayed. Name the step **Apply Load**. For **Insert new step after** the only option is **Initial** and it is selected by default. Set the **Procedure type** to **General** from the drop down menu. In the list scroll down till you see **Static, General** and select it.


Create Step

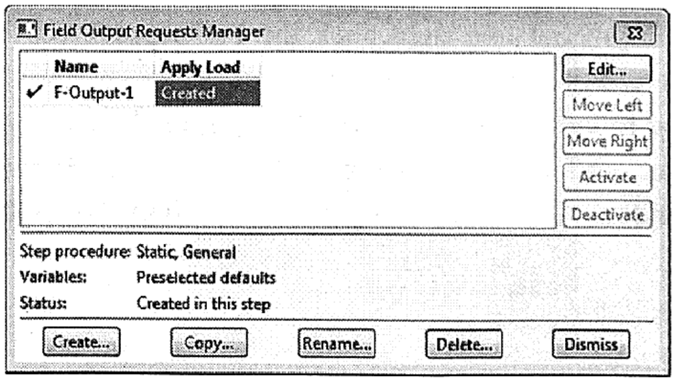
Name: Apply Load

Insert new step after: Initial

Procedure type: General

Static, General

23. Click **Continue...**. You see the **Edit Step** window. For the description type in **Load is applied during this step**. Leave everything else set to the defaults.
24. Click **OK**. You'll notice that the **Steps** item in the Model Database now has 2 steps, **Initial** and **Apply Load**.
25. Let's now create the field output requests. Right click ~~the~~ **Field Output Requests** item in the model tree and choose **Manager**. You see the **Field Output Requests Manager** window with an output request **F-Output-1** created in the **Apply Load** step.

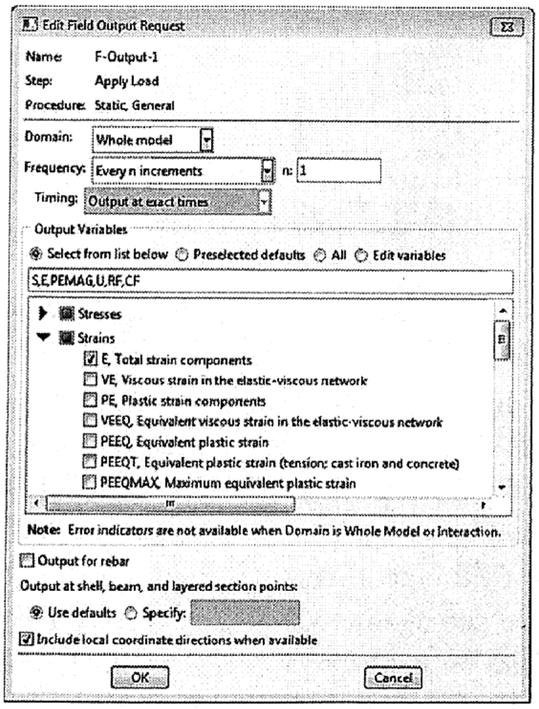

Field Output Requests Manager

Name	Apply Load
✓ F-Output-1	Created

Step procedure: Static, General
Variables: Preselected defaults
Status: Created in this step

Click the **Edit** button. You notice a number of output variables selected by default. On top of the list of available output variables you see a comma separated listing of the ones selected which by default reads **CDISP, CF, CSTRESS, LE, PE, PEEQ, PEMAG, RF, S, U,.**

26. From the **Strains** remove **PE, Plastic strain components**, **PEEQ, Equivalent plastic strain** and **LE, Logarithmic strain components**. Add **E, Total strain components**. Remove **Contact** entirely. The variables you are left with are displayed above as **S,E,PEMAG,U,RF,CF**


Edit Field Output Request

Name: F-Output-1
Step: Apply Load
Procedure: Static, General

Domain: Whole model
Frequency: Every n increments n: 1
Timing: Output at exact times

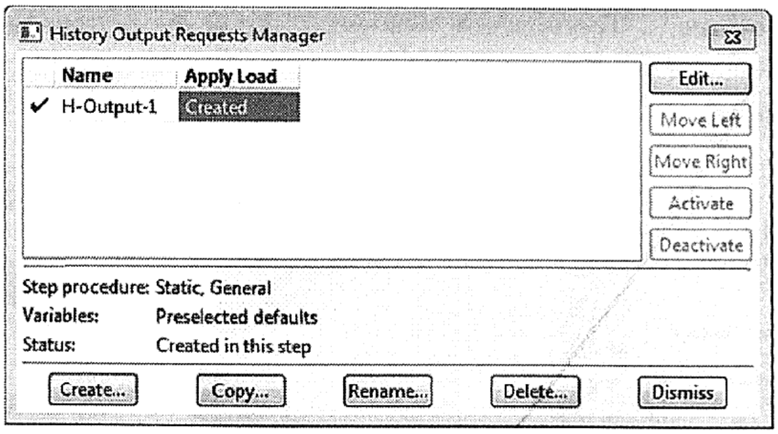
Output Variables
[✓ Select from list below] Preselected defaults All Edit variables
S,E,PEMAG,U,RF,CF

- Stresses
- Strains
  - [✓] E, Total strain components
  - [ ] VE, Viscous strain in the elastic-viscous network
  - [ ] PE, Plastic strain components
  - [ ] VEEQ, Equivalent viscous strain in the elastic-viscous network
  - [ ] PEEQ, Equivalent plastic strain
  - [ ] PEEQT, Equivalent plastic strain (tension: cast iron and concrete)
  - [ ] PEEQMAX, Maximum equivalent plastic strain

Note: Error indicators are not available when Domain is Whole Model or Interaction.

[ ] Output for rebar
Output at shell, beam, and layered section points:
[✓ Use defaults] Specify:
[✓] Include local coordinate directions when available

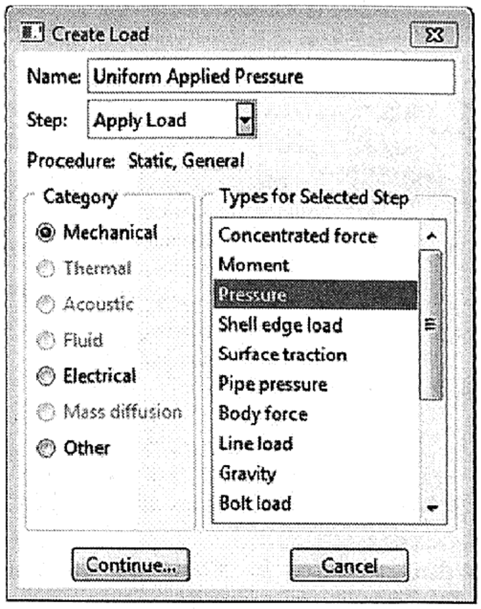
27. Click **OK**. Then click **Dismiss...** to close the **Field Output Request Manager** window. In the model tree right click the **F-Output-1** sub-item of **Field Output Requests** and rename it **Selected Field Outputs**.
28. Let's move on to history output requests. Right click **History Output Requests** in the model tree and choose **Manager**. You see the **History Output Requests Manager** window. It is very similar to the **Field Output Requests Manager** window.



Name	Apply Load
✓ H-Output-1	Created

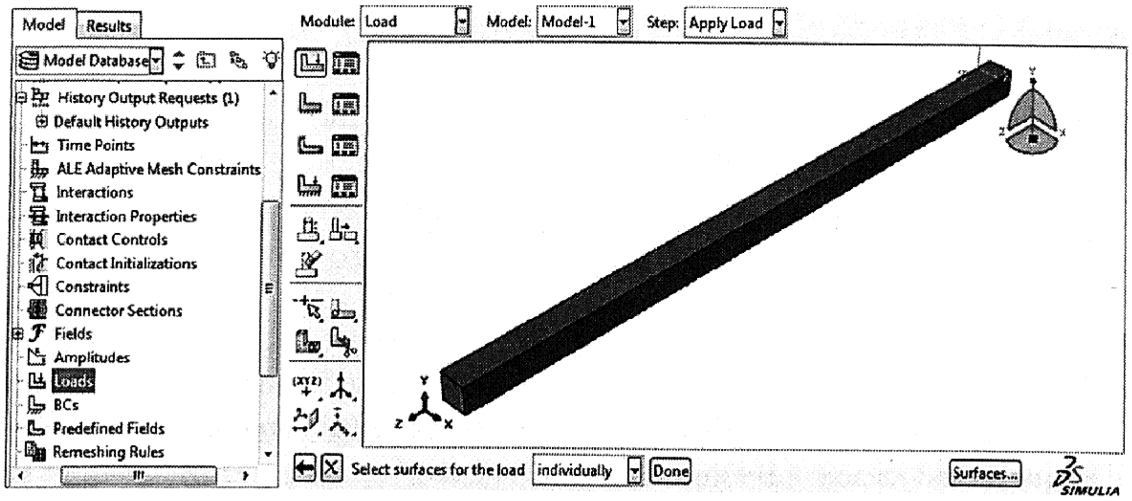
Step procedure: Static, General
Variables: Preselected defaults
Status: Created in this step

29. If you click **Edit** you can see the variables selected by default. We're going to leave the default variables selected so **Cancel** out of the **Edit History Output Requests** window. In the model tree right click the **H-Output-1** sub-item of **History Output Requests** and rename it **Default History Outputs**.
30. It's time to apply loads to the beam. In the model tree double click the **Loads** item. You see the **Create Load** window. Name the load **Uniform Applied Pressure**. For the step select **Apply Load**. Under **Category** choose **Mechanical**. And from the **Types for Selected Step** list choose **Pressure**.

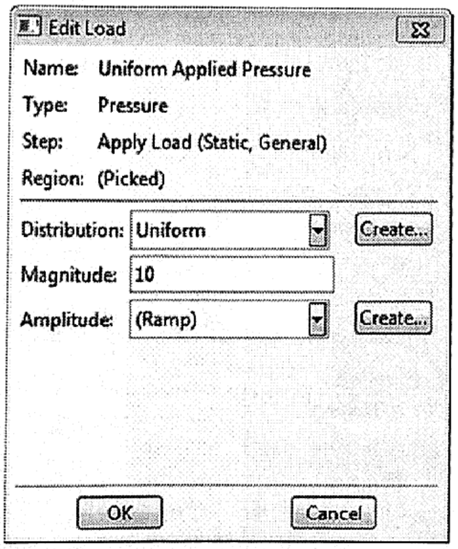


Name: Uniform Applied Pressure
Step: Apply Load
Procedure: Static, General
Category: [✓ Mechanical]
Types for Selected Step: [✓ Pressure]

31. Click **Continue....** The viewport displays a hint at the bottom **Select surfaces for the load**. Hover your mouse over the top surface of the beam till its edges light up. Click to select.



32. Click **Done**. You see the **Edit Load** window. For **Distribution** choose **Uniform** from the drop down list. For **Magnitude** enter a value of **10** Pa (just type in 10 without units).


Edit Load

Name: Uniform Applied Pressure

Type: Pressure

Step: Apply Load (Static, General)

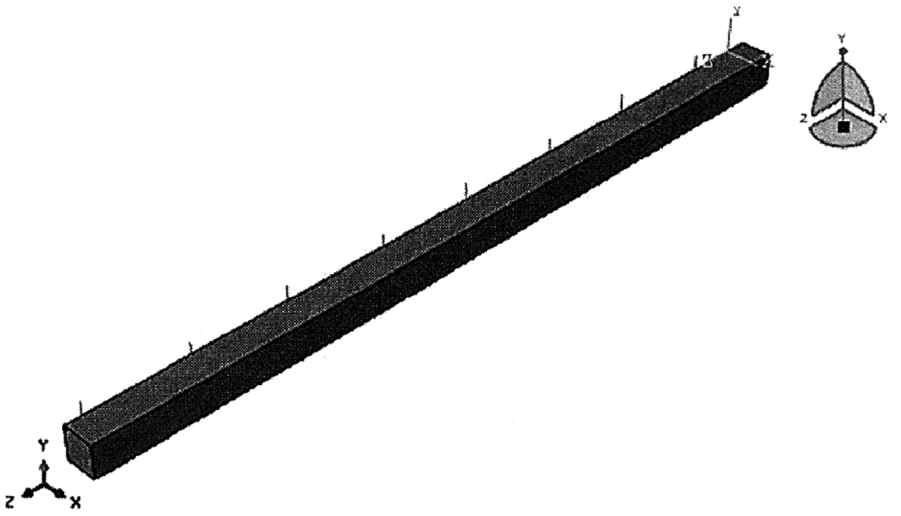
Region: (Picked)

Distribution: Uniform

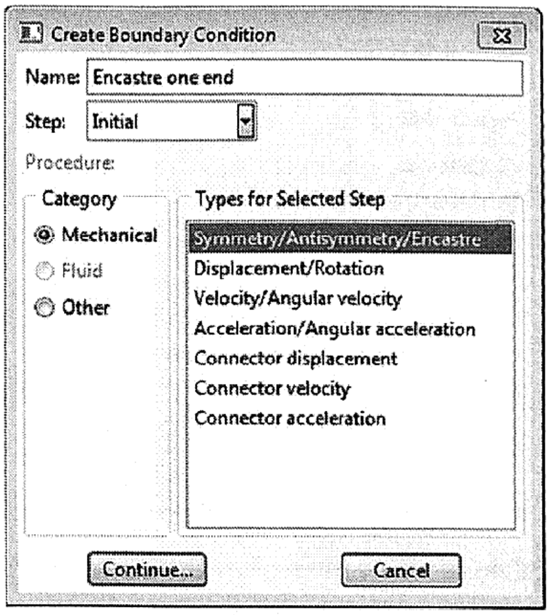
Magnitude: 10

Amplitude: (Ramp)

33. Click **OK**. The viewport updates to show the pressure being applied on the top surface with the arrows representing the direction. Also the **Loads** item in the Model Database tree now has a sub-item called **Uniform Applied Pressure**.



34. The next step is to apply the boundary conditions or constraints. Double click on the **BCs** item in the Model Database tree. You see the **Create Boundary Condition** window. Name it **Encastre one end**. Change **Step** to **Initial**. Under **Category** choose **Mechanical**. From the available options for **Types for Selected Step** choose **Symmetry/Antisymmetry/Encastre**.


Create Boundary Condition

Name: Encastre one end

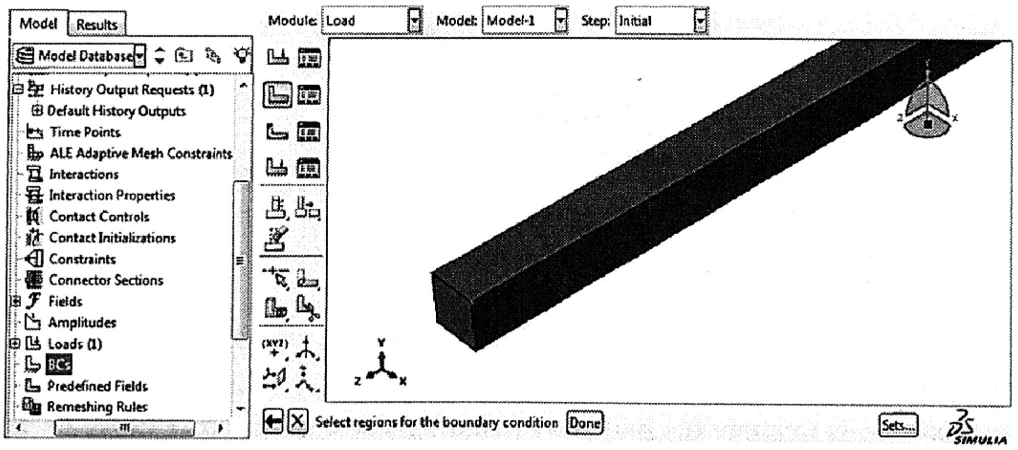
Step: Initial

Procedure:

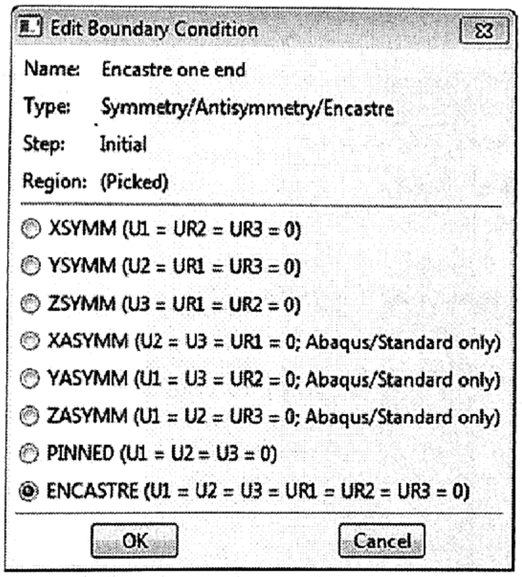
Category: [✓ Mechanical] Fluid Other

Types for Selected Step:
- [✓ Symmetry/Antisymmetry/Encastre]
- Displacement/Rotation
- Velocity/Angular velocity
- Acceleration/Angular acceleration
- Connector displacement
- Connector velocity
- Connector acceleration

35. Click **Continue….** The viewport displays a hint at the bottom **Select regions for the boundary condition**. Hover your mouse over the surface at one end of the beam till its edges light up. Click to select it.



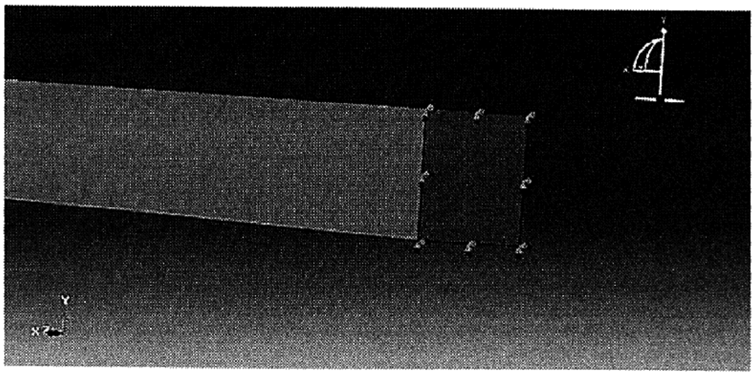
36. Click **Done**. You see the **Edit Boundary Condition** window. Choose **ENCASTRE (U1 = U2 = U3 = UR1 = UR2 = UR3 =0)**. This will clamp the beam at this end.


Edit Boundary Condition

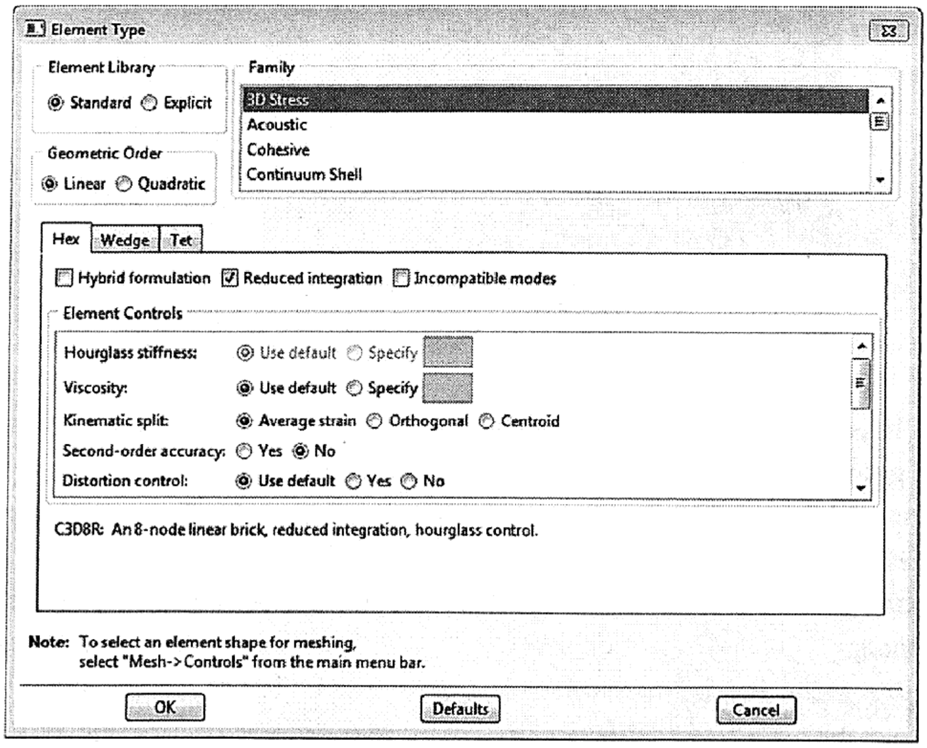
Name: Encastre one end
Type: Symmetry/Antisymmetry/Encastre
Step: Initial
Region: (Picked)

- XSYMM (U1 = UR2 = UR3 = 0)
- YSYMM (U2 = UR1 = UR3 = 0)
- ZSYMM (U3 = UR1 = UR2 = 0)
- XASYMM (U2 = U3 = UR1 = 0; Abaqus/Standard only)
- YASYMM (U1 = U3 = UR2 = 0; Abaqus/Standard only)
- ZASYMM (U1 = U2 = UR3 = 0; Abaqus/Standard only)
- PINNED (U1 = U2 = U3 = 0)
- [✓ ENCASTRE (U1 = U2 = U3 = UR1 = UR2 = UR3 = 0)]

37. The viewport will update to show the end of the beam being clamped. Also the **BCs** item now has a sub-item called **Encastre one end.**



38. If you haven't been saving your work all along now would be a good time to do it. We're going to mesh the part and then run the simulation.
39. In the model tree expand the **Parts** item again, and then the **Beam** sub-item. You see the **Mesh (Empty)** sub-item at the bottom. Double-click it. You are now in mesh mode and you notice the toolbar next to the viewport changes to provide you with mesh tools.
40. Using the menu bar go to **Mesh > Element Type**. The **Element Type** window is displayed. For **Element Library** choose **Standard**, for **Geometric Order** choose **Linear**, and for **Family** choose **3D Stress** from the list. Leave everything else at the defaults. You will notice the description **C3D8R: An 8-node linear brick, reduced integration, hourglass control** near the bottom of the window.



Element Type

Element Library: [✓ Standard] Explicit

Geometric Order: [✓ Linear] Quadratic

Family: 3D Stress, Acoustic, Cohesive, Continuum Shell

Hex | Wedge | Tet

Hybrid formulation [✓ Reduced integration] Incompatible modes

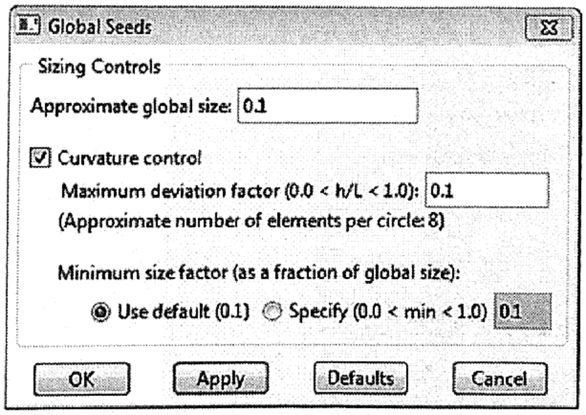
Element Controls

Hourglass stiffness:	[✓ Use default] Specify
Viscosity:	[✓ Use default] Specify
Kinematic split:	[✓ Average strain] Orthogonal Centroid
Second-order accuracy:	Yes [✓ No]
Distortion control:	[✓ Use default] Yes No

C3D8R: An 8-node linear brick, reduced integration, hourglass control.

Note: To select an element shape for meshing,
select "Mesh->Controls" from the main menu bar.

41. Click **OK**.
42. Then use the menu bar to navigate to **Seed** > Instances **~~Part~~**. The **Global Seeds** window is displayed. Change the **Approximate global size** to **0.2**, which is the width of our beam. Set the **Maximum deviation factor** to **0.1**.



Global Seeds

Sizing Controls

Approximate global size: 0.1

[✓ Curvature control]

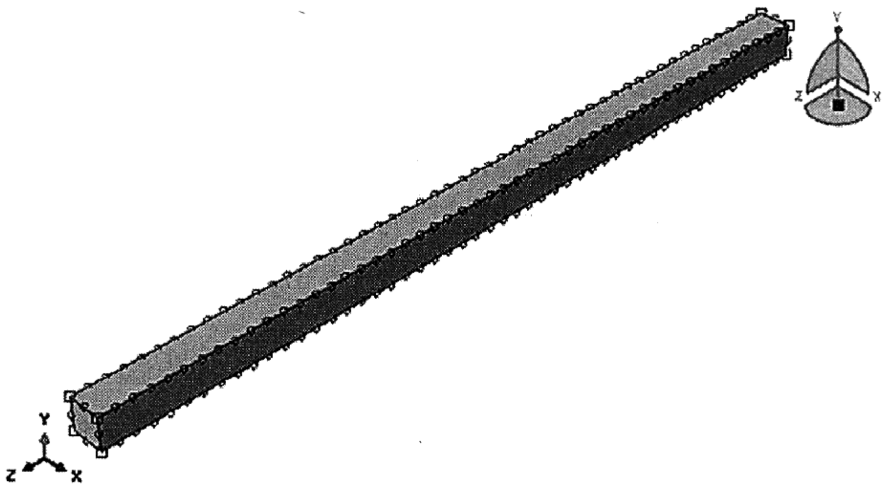
Maximum deviation factor (0.0 < h/L < 1.0): 0.1

(Approximate number of elements per circle: 8)

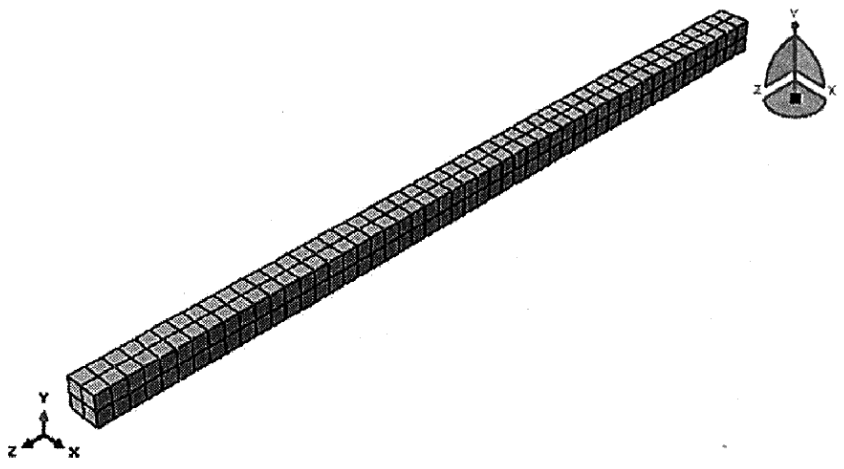
Minimum size factor (as a fraction of global size):

[✓ Use default (0.1)] Specify (0.0 < min < 1.0) 0.1

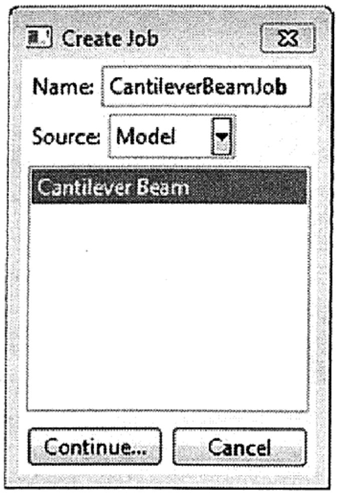
43. The beam in the viewport updates to show where the nodes have been applied.



44. Then from the menu bar go to **Mesh** > ~~**Part.**~~ Instances You see the question **OK to mesh the part?** at the bottom of the viewport window. Click on **Yes**. The part is meshed. The **Mesh** item in the model tree no longer has the words **(Empty)** next to it.



45. Now it is time to create the analysis job.
46. All the way at the bottom of the model tree you see **Analysis** with the sub-item **Jobs**. Double-click on it. The **Create Job** window is displayed. Name it **CantileverBeamJob.** Notice that there are no spaces in the name. Putting a space in the job name can cause problems because Abaqus uses the job name as part of the name of some of the output files such as the output database (.odb) file. **Source** is set to **Model** and the only model you can select from the list is **Cantilever Beam.**

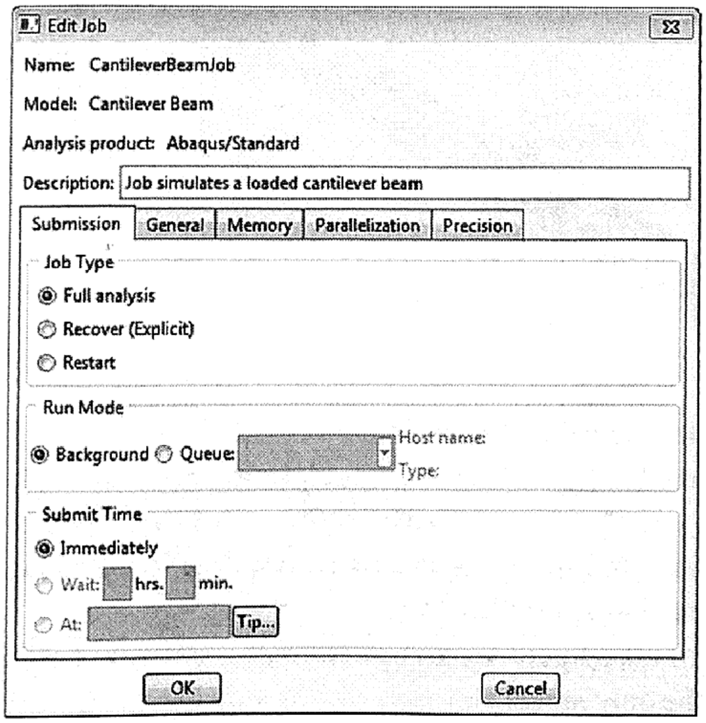


Name: CantileverBeamJob

Source: Model

Cantilever Beam

47. Click **Continue....** You see the **Edit Job** window. In the **Description** textbox type in **Job simulates a loaded cantilever beam**. Set the **Job Type** to **Full Analysis.** Leave the other settings to default. Notice that in the **Memory** tab there is an option for **Memory allocation units**. On my system the option selected is **Percent of physical memory**, and for the **Maximum preprocessor and analysis memory** my system defaults to **50%**. You might wish to play with these numbers if your computer has insufficient resources.



Name: CantileverBeamJob

Model: Cantilever Beam

Analysis product: Abaqus/Standard

Description: Job simulates a loaded cantilever beam

Job Type

[✓ Full analysis] Recover (Explicit) Restart

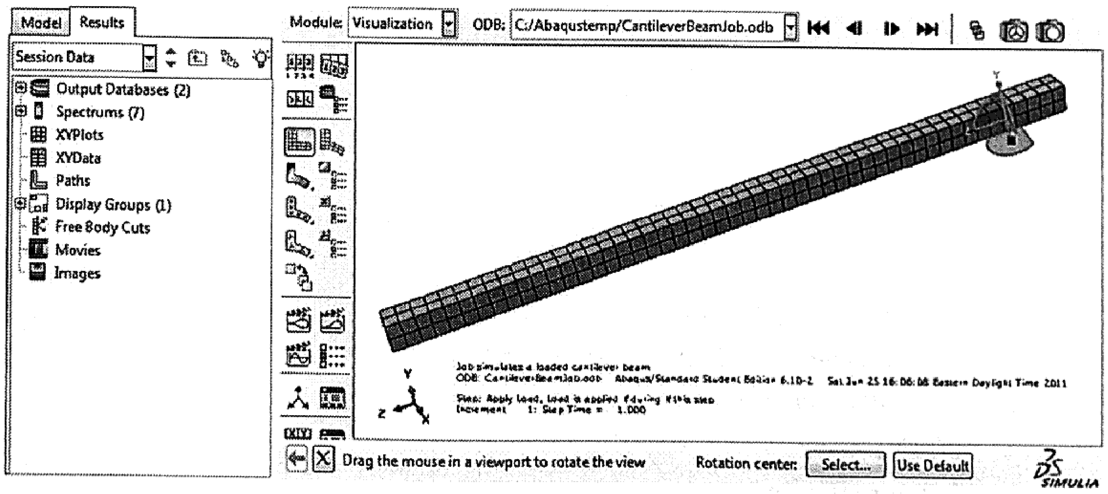
Run Mode

[✓ Background] Queue: Host name: Type:

Submit Time

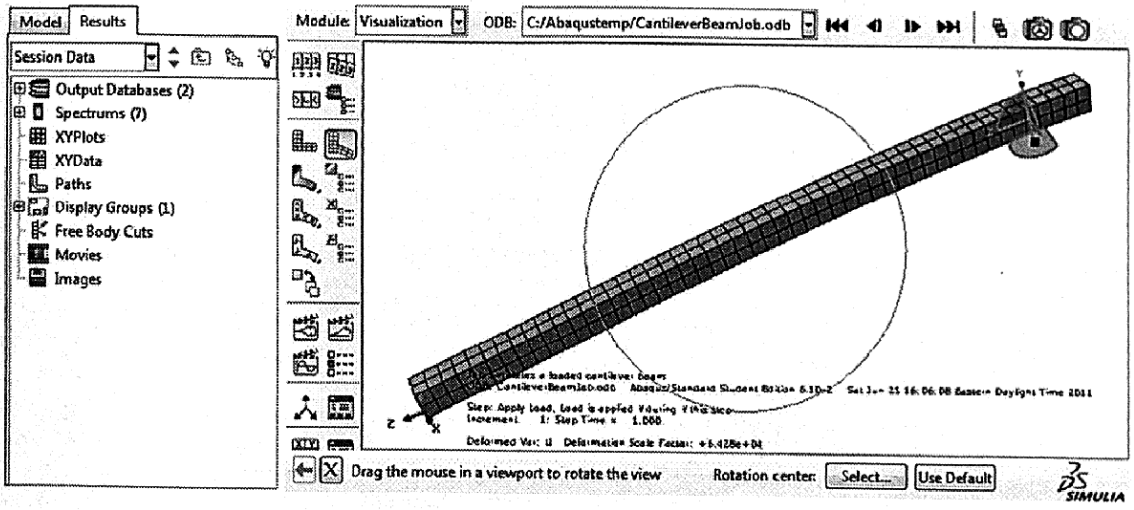
[✓ Immediately] Wait: hrs. min. At:

48. Notice that the **Jobs** item in the model tree now has **CantileverBeamJob** listed (you might have to hit the '+' symbol to see it). Right-click on it and choose **Submit**.
49. The job starts running. You see the words **(Submitted)** appear next to its name in parentheses, then a few seconds later you see **(Running)** and when the simulation is complete you see **(Completed)**.
50. Right click on **CantileverBeamJob (Completed)** and choose **Results**. You see the undeformed shape.



```
Job simulates a loaded cantilever beam
ODB: CantileverBeamJob.odb    Abaqus/Standard Student Edition 6.10-2    Sat Jun 25 16:06:08 Eastern Daylight Time 2011

Step: Apply load, load is applied during this step
Increment     1: Step Time =    1.000
```

51. Click the **Plot Deformed Shape** button in the toolbar to the left of the viewport. You will see your deformed beam. Of course the deformation has been exaggerated by Abaqus. You can change that if you wish by going to **Options > Common...** if you wish.



```
...tes a loaded cantilever beam
...CantileverBeamJob.odb    Abaqus/Standard Student Edition 6.10-2    Sat Jun 25 16:06:08 Eastern Daylight Time 2011

Step: Apply load, load is applied during this step
Increment     1: Step Time =    1.000

Deformed Var: U   Deformation Scale Factor: +6.428e+04
```

You have created and run a complete simulation in Abaqus/CAE. It was a very basic setup, but it covered all the essentials from creating a part and assigning sections and material properties to applying loads and constraints and meshing. Now we're going to watch a script perform all the same actions that we just did.

Open up a text editor such as Notepad++ and type in the following script.

```
# ********************************************************************************
# Cantilever Beam bending under the action of a uniform pressure load

# ********************************************************************************

from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# -------------------------------------------------------------------------
# Create the model
mdb.models.changeKey(fromName='Model-1', toName='Cantilever Beam')
beamModel = mdb.models['Cantilever Beam']

# -------------------------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the beam cross section using rectangle tool
beamProfileSketch = beamModel.ConstrainedSketch(name='Beam CS Profile',
                                                sheetSize=5)
beamProfileSketch.rectangle(point1=(0.1,0.1), point2=(0.3,-0.1))

# b) Create a 3D deformable part named "Beam" by extruding the sketch
beamPart=beamModel.Part(name='Beam', dimensionality=THREE_D,
                                   type=DEFORMABLE_BODY)
beamPart.BaseSolidExtrude(sketch=beamProfileSketch, depth=5)

# -------------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs
# modulus and poissons ratio
beamMaterial = beamModel.Material(name='AISI 1005 Steel')
beamMaterial.Density(table=((7872, ),       ))
beamMaterial.Elastic(table=((200E9, 0.29), ))
```

```
# -----------------------------------------------------------------------------
# Create solid section and assign the beam to it

import section

# Create a section to assign to the beam
beamSection = beamModel.HomogeneousSolidSection(name='Beam Section',
                                                material='AISI 1005 Steel')

# Assign the beam to this section
beam_region = (beamPart.cells,)
beamPart.SectionAssignment(region=beam_region, sectionName='Beam Section')

# -----------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
beamAssembly = beamModel.rootAssembly
beamInstance = beamAssembly.Instance(name='Beam Instance', part=beamPart,
                                                      dependent=ON)

# -----------------------------------------------------------------------------
# Create the step

import step

# Create a static general step
beamModel.StaticStep(name='Apply Load', previous='Initial',
                     description='Load is applied during this step')

# -----------------------------------------------------------------------------
# Create the field output request

# Change the name of field output request 'F-Output-1' to 'Selected Field Outputs'
beamModel.fieldOutputRequests.changeKey(fromName='F-Output-1',
                                        toName='Selected Field Outputs')

# Since F-Output-1 is applied at the 'Apply Load' step by default, 'Selected Field
# Outputs' will be too
# We only need to set the required variables
beamModel.fieldOutputRequests['Selected Field Outputs'].setValues(variables=('S',
                                                'E', 'PEMAG', 'U', 'RF', 'CF'))

# -----------------------------------------------------------------------------
# Create the history output request

# We try a slightly different method from that used in field output request
# Create a new history output request called 'Default History Outputs' and assign
# both the step and the variables
```

```
beamModel.HistoryOutputRequest(name='Default History Outputs',
                               createStepName='Apply Load', variables=PRESELECT)

# Now delete the original history output request 'H-Output-1'
del beamModel.historyOutputRequests['H-Output-1']

# -------------------------------------------------------------------------
# Apply pressure load to top surface

# First we need to locate and select the top surface
# We place a point somewhere on the top surface based on our knowledge of the
# geometry
top_face_pt_x = 0.2
top_face_pt_y = 0.1
top_face_pt_z = 2.5
top_face_pt = (top_face_pt_x,top_face_pt_y,top_face_pt_z)

# The face on which that point lies is the face we are looking for
top_face = beamInstance.faces.findAt((top_face_pt,))

# We extract the region of the face choosing which direction its normal points in
top_face_region=regionToolset.Region(side1Faces=top_face)

# Apply the pressure load on this region in the 'Apply Load' step
beamModel.Pressure(name='Uniform Applied Pressure', createStepName='Apply Load',
                   region=top_face_region, distributionType=UNIFORM,
                   magnitude=10, amplitude=UNSET)

# -------------------------------------------------------------------------
# Apply encastre (fixed) boundary condition to one end to make it cantilever

# First we need to locate and select the top surface
# We place a point somewhere on the top surface based on our knowledge of the
# geometry
fixed_end_face_pt_x = 0.2
fixed_end_face_pt_y = 0
fixed_end_face_pt_z = 0
fixed_end_face_pt = (fixed_end_face_pt_x,fixed_end_face_pt_y,fixed_end_face_pt_z)

# The face on which that point lies is the face we are looking for
fixed_end_face = beamInstance.faces.findAt((fixed_end_face_pt,))

# We extract the region of the face choosing which direction its normal points in
fixed_end_face_region=regionToolset.Region(faces=fixed_end_face)

beamModel.EncastreBC(name='Encaster one end', createStepName='Initial',
                                             region=fixed_end_face_region)

# -------------------------------------------------------------------------
# Create the mesh

import mesh
```

```
# First we need to locate and select a point inside the solid
# We place a point somewhere inside it based on our knowledge of the geometry
beam_inside_xcoord=0.2
beam_inside_ycoord=0
beam_inside_zcoord=2.5

elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN, secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)


beamCells=beamPart.cells
selectedBeamCells=beamCells.findAt((beam_inside_xcoord,beam_inside_ycoord,
                                                      beam_inside_zcoord),)
beamMeshRegion=(selectedBeamCells,)
beamPart.setElementType(regions=beamMeshRegion, elemTypes=(elemType1,))

beamPart.seedPart(size=0.1, deviationFactor=0.1)

beamPart.generateMesh()

# -----------------------------------------------------------------------
# Create and run the job

import job

# Create the job
mdb.Job(name='CantileverBeamJob', model='Cantilever Beam', type=ANALYSIS,
        explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE,
        description='Job simulates a loaded cantilever beam',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)

# Run the job
mdb.jobs['CantileverBeamJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['CantileverBeamJob'].waitForCompletion()

# End of run job
# -----------------------------------------------------------------------
# Post processing

import visualization

beam_viewport = session.Viewport(name='Beam Results Viewport')
beam_Odb_Path = 'CantileverBeamJob.odb'
an_odb_object = session.openOdb(name=beam_Odb_Path)
beam_viewport.setValues(displayedObject=an_odb_object)
beam_viewport.odbDisplay.display.setValues(plotState=(DEFORMED, ))
```

Typing out the above code might be a real pain and you'll likely mistype a few variable names or make other syntax errors creating a lot of bugs. It might be a better idea just to use the source code provided with the book – cantilever_beam.py.

Open a new Abaqus model. Then go to **File > Run Script**. The script will recreate everything you did manually in Abaqus/CAE. It will also create and submit the job so you will probably notice the analysis running for a few seconds after you run the script. You can then right click on the 'CantileverBeamJob' item in the model tree and choose **Results** to see the output. It will be identical to what you got when performing the simulation in the GUI.

1.5 Conclusion

In the example we did not use the script to accomplish anything that could not be done in Abaqus/CAE. In fact we first performed the procedure in Abaqus/CAE before writing our script. But I wanted to drive home an important point: You can do just about anything in a script that you can do in the GUI. Once you're able to script a basic simulation, you'll be able to move on to more complex tasks that would really only be feasible with a script such as making automated decisions when creating the simulation or performing repetitive actions within the study.

As for the script from this example, we're going to take a closer at it in Chapter 4. Before we can do this you're going to have to learn a little Python syntax in Chapter 3. But first let's take a look at the different ways of running a script in Chapter 2.

2

Running a Script

2.1 Introduction

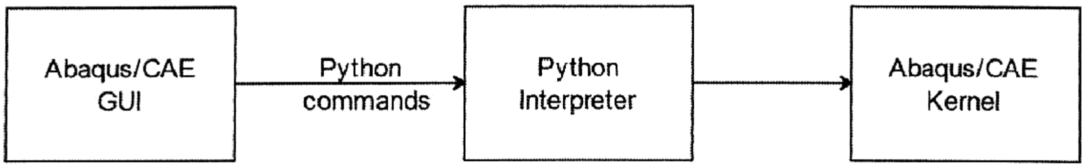
This chapter will help you understand how Python scripting fits into Abaqus, and also point out some of the different ways a script can be run. While you may choose to use only one of the methods available, it is handy to know your options.

2.2 How Python fits in

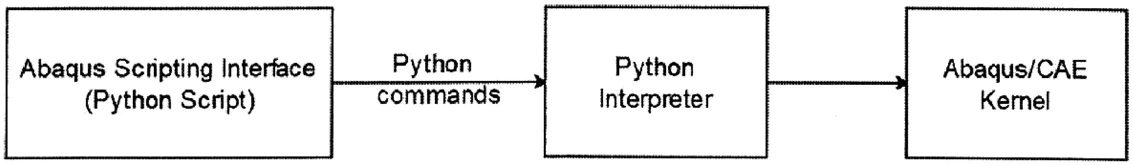
A few years ago Abaqus existed purely as a finite element solver. It had no preprocessor or postprocessor. You created text based input files (.inp), submitted them to the solver using the command line, and got text based output files. Today it has a preprocessor which generates the input file for you – Abaqus/CAE (CAE stands for Complete Abaqus Environment), and a postprocessor that helps you visualize the results from the output database – Abaqus/Viewer. When you use Abaqus/CAE, the actions you perform in the GUI (graphical user interface) generate commands in Python, and these Python commands are interpreted by the Python Interpreter and sent to the Abaqus/CAE kernel which executes them. For example when you create a new material in Abaqus/CAE, you type in a material name and specify a number of material behaviors in the 'Edit Material' dialog box using the available menus and options. When you click OK after this, Abaqus/CAE generates a command or a number of commands based on what you have entered and sends it to the kernel. They may look something like:

```
beamMaterial = beamModel.Material(name='AISI 1005 Steel')
beamMaterial.Density(table=((7872, ),       ))
beamMaterial.Elastic(table=((200E9, 0.29), ))
```

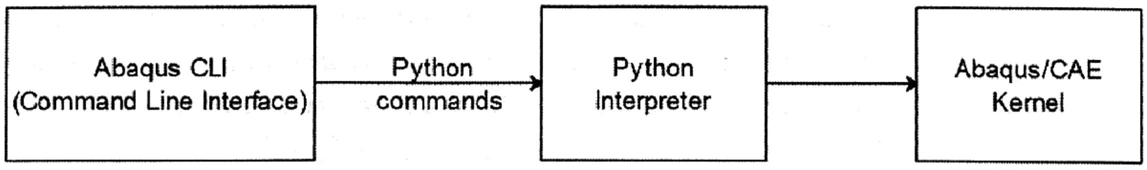
In short, the Abaqus/CAE GUI is the easy-to-use interface between you, the user, and the kernel, and the GUI and kernel communicate using Python commands.



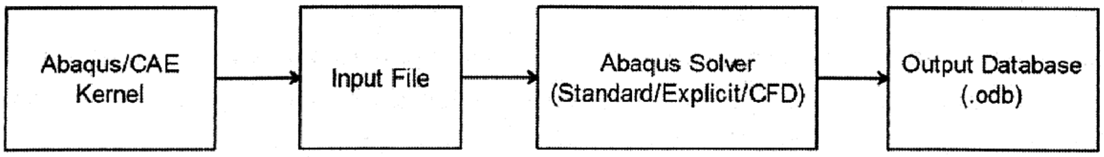
The Abaqus Scripting Interface is an alternative to using the Abaqus/CAE GUI. It allows you to write a Python script in a .py file and submit that to the Abaqus/CAE Kernel.



A third option is to type scripts into the kernel command line interface (CLI) at the bottom of the Abaqus/CAE window.



The Abaqus/CAE kernel understands the model and creates an input file that can be submitted to the solver. The solver accepts this input file, runs the simulation, and writes its output to an output database (.odb) file.



2.3 Running a script within Abaqus/CAE

You have the option of running a script from within Abaqus/CAE using the **File > Run Script...** menu option. You can do this if your script irrespective of whether your script only performs a single task or runs the entire simulation.

2.3.1 Running a script in GUI to execute a single or multiple tasks

If you have a script that performs a single independent task or multiple tasks assuming some amount of model setup has already been completed or will be performed later, you need to run that script in Abaqus/CAE. For instance, in Example 1.1 of Chapter 1, we wrote a script which only creates materials. On its own this script cannot run a simulation, it does not create a part, assembly, steps, loads and boundary conditions and so on. However it can be run within Abaqus/CAE to accomplish a specific task. When we ran the script using **File > Run Script...** you noticed the model tree get populated with new materials. You could then continue working on the model after that.

Such scripts will not run as standalone from the command line, or at least they won't accomplish anything.

2.3.2 Running a script in GUI to execute an entire simulation

If you have a script that can run the entire simulation, from creating the part and materials to applying loads and constraints to meshing and running the job, one way to run it is through the GUI using *File > Run...* This was demonstrated in Example 1.2 of Chapter 1. However such a script can also be run directly from the command line.

2.4 Running a script from the command line

In order to run a script from the command line, the Abaqus executable must be in your system path.

Path

The path is a list of directories which the command interpreter searches for an executable file that matches the command you have given it. It is one of the environment variables on a Windows machine.

The directory you need to add to your path is the "Commands" directory of your Abaqus installation. By default Abaqus Student Edition v6.10 installs itself to directory "C:\SIMULIA\Abaqus". It likely did the same on your computer unless you chose to install it to a different location during the installation procedure. One of the sub-directories of "C:\SIMULIA\Abaqus" is "Commands", so its location is "C:\SIMULIA\Abaqus\Commands". This location needs to be added to the system path.

Check if Abaqus is already in the path

The first thing to do is to check if this location has already been added to your path as part of the installation. You can do this by opening a command prompt. To access the command prompt in Windows Vista or Windows 7, click the Start button at the lower left corner of your screen, and in the 'Start search' bar that appears right above it type 'cmd' (without the quotes) and hit enter. In Windows XP you click the Start button, click 'Run', and type in 'cmd' and click OK. You will see your blinking cursor. Type the word 'path' (without the quotes). You wil'l see a list of directories separated by semicolons that are in the system path. If Abaqus has been add to the path, you will see "C:\SIMULIA\Abaqus\Commands" (or wherever your Commands folder is) listed among the directories. If not, you need to add it manually to the path.

Add Abaqus to the Path

Adding a directory to the path differs slightly for each version of Windows. There are many resources on the internet that instruct you on how to add a variable to the path and a quick Google search will reveal these. As an example, this is how you add Abaqus to the path in Windows Vista and Windows 7.

1. Right click **My Computer** and choose **Properties**
2. Click **Advanced System Settings** in the menu on the left.
3. In the System Properties window that opens, go to the **Advanced** tab. At the bottom of the window you see a button labeled **Environment Variables...** Click it.
4. The environment variables window opens. In the **System variables** list, scroll down till you see the **Path** variable. Click it, then click the **Edit** button. You see the **Edit System Variable** window.
5. The variable name shall be left at its default of **Path**. The variable value needs to be modified. It contains a number of directories, each separated by a semi colon. It may look something like **C:\Windows\System32\;C:\Windows\;C:\Windows\System32\Wbem**. At the end of it, add another semi colon, and then type in **C:\SIMULIA\Abaqus\Commands**. So it should now look something like **C:\Windows\System32\;C:\Windows\;C:\Windows\System32\Wbem;C:\SIMULIA\Abaqus\Commands**. Click **OK** to close the window, and click **OK** to close the **Environment Variables** window.

6. Now if you go back to the command prompt and type **path**, you see the path has been updated to include Abaqus

Running the script from the command line

Now that Abaqus is in the system path, you can run your scripts from the command line.

First you navigate to the folder containing your script files using DOS commands such as **cd** (change directory) command. For example, when you start the command prompt, if your cursor looks something like **C:\Users\Gautam>,** and your script is located in the folder **C:\Users\Gautam \Desktop\Abaqus Book**, then type in

```
cd C:\Users\Gautam \Desktop\Abaqus Book
```

and press Enter. Your cursor will now change to **C:\Users\Gautam\Desktop\Abaqus Book>**

You are now in a position to run the script with or without the Abaqus/CAE GUI being displayed.

2.4.1 Run the script from the command line without the GUI

Type the command to run the script without the Abaqus/CAE GUI. The exact command varies depending on the version of Abaqus.

In the commercial version of Abaqus you would type

```
abaqus cae noGUI= "cantilever_beam.py"
```

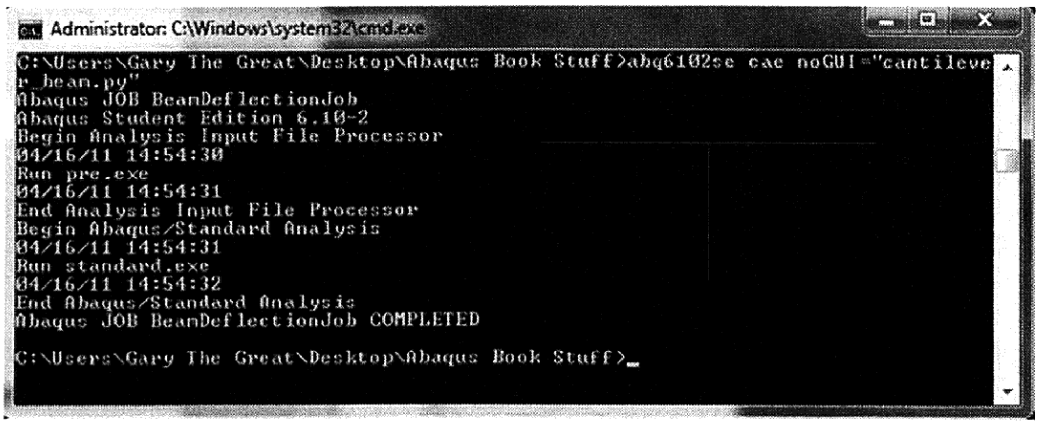
In the student edition (SE) version 6.9-2 you would type

```
abq692se cae noGUI="cantilever_beam.py"
```

In the student edition (SE) version 6.10-2 you would type

```
abq6102se cae noGUI="cantilever_beam.py"
```

Notice the difference in the first word of all these statements. If you are not using either of these versions the command you use will be different as well. To figure out exactly what it is, go to the 'Commands' folder in the installation directory and look for a file with the extension '.bat'. In the commercial version of Abaqus this file is called 'abaqus.bat', hence in the commercial version you use the command 'abaqus cae noGUI="cantilever_beam.py". In Abaqus 6.10-2 student edition, the file is called 'abq6102se.bat' which is why the command 'abq6102se cae noGUI="cantilever_beam.py" has been used. Depending on the name of your file, change the first word in the statement.



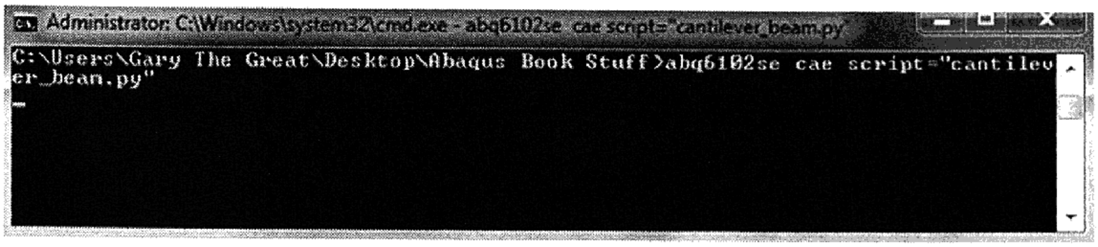
```
C:\Users\Gary The Great\Desktop\Abaqus Book Stuff>abq6102se cae noGUI="cantileve
r_beam.py"
Abaqus JOB BeamDeflectionJob
Abaqus Student Edition 6.10-2
Begin Analysis Input File Processor
04/16/11 14:54:30
Run pre.exe
04/16/11 14:54:31
End Analysis Input File Processor
Begin Abaqus/Standard Analysis
04/16/11 14:54:31
Run standard.exe
04/16/11 14:54:32
End Abaqus/Standard Analysis
Abaqus JOB BeamDeflectionJob COMPLETED

C:\Users\Gary The Great\Desktop\Abaqus Book Stuff>_
```

When you run your scripts in this manner, you will not see the GUI at all. While the script is running, you will notice that the cursor is busy and you cannot type in any other commands at the prompt. This is because we have used the built in method waitForCompletion() in the script which prevents the user from executing other DOS commands while the simulation is running. We will take a look at this statement again a little later, just be aware that if we did not include the waitForCompletion() command in the script, the prompt would continue to remain active even while the simulation is being run. And if you find yourself running batch files, or linking your simulations with optimization software such as ISight or ModelCenter, this knowledge will come in handy.

2.4.2 Run the script from the command line with the GUI

If on the other hand you wish to have the GUI displayed replace the word 'noGUI' with 'script'. So in student edition version 6.10-2 you would type

```
abq6102se cae script="cantilever_beam.py"
```


Administrator: C:\Windows\system32\cmd.exe - abq6102se cae script="cantilever_beam.py"

```
C:\Users\Gary The Great\Desktop\Abaqus Book Stuff>abq6102se cae script="cantilev
er_beam.py"
```

When you run your scripts in this manner, Abaqus/CAE will open up and the script is run within it. In addition the cursor will remain busy (as seen in the figure), and will only be released once you close that instance of Abaqus/CAE.

2.5 Running a script from the command line interface (CLI)

The kernel command line interface is the area below the viewport in Abaqus/CAE. Chances are the message area is currently displayed here. If you click the box with '>>>' on it you will be able to type in commands. We will use this to test a few different Python commands in the next chapter. For now I wish to make you aware that it is possible to run a script from here using the execfile() command.

Type in

```
Execfile('cantilever_beam.py')
```

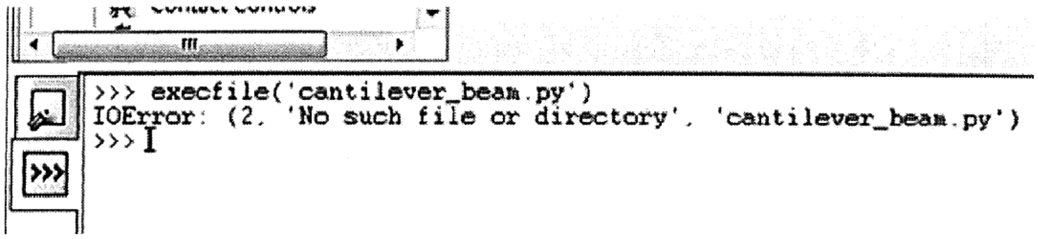
The file you've passed as an argument to execfile() needs to be present in the current work directory for Abaqus, otherwise you need to spell out the full path such as:

```
Execfile('C:\Users\Gautam\Desktop\Book\cantilever_beam.py')
```

By default the work directory is C:\Temp although you can change it using **File > Set Work Directory..**

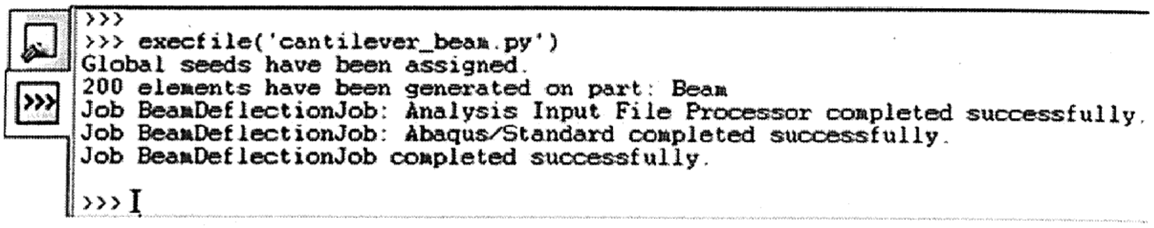
If the file is not in the current work directory and you did not specify the full path, Abaqus will not find the script and will display an IOError.

```
IOError: (2, 'No such file or directory', 'cantilever_beam.py')
```



```
>>> execfile('cantilever_beam.py')
IOError: (2, 'No such file or directory', 'cantilever_beam.py')
>>>
```

If the file is present in the work directory, or you specify the full path, the script executes successfully.



```
>>>
>>> execfile('cantilever_beam.py')
Global seeds have been assigned.
200 elements have been generated on part: Beam
Job BeamDeflectionJob: Analysis Input File Processor completed successfully.
Job BeamDeflectionJob: Abaqus/Standard completed successfully.
Job BeamDeflectionJob completed successfully.
>>>
```

2.6 Conclusion

This chapter has presented to you some of the various ways in which scripts can be run. You may choose the appropriate method based on the task at hand, or feel free to go with personal preference.

3

Python 101

3.1 Introduction

In the cantilever beam example of Chapter 1, we began by creating the entire model in Abaqus/CAE. We then opened up a new file and ran a script which accomplished the exact same task. How exactly did the script work and what did all those code statements mean? Before we can start to analyze this, it is necessary to learn some basic Python syntax. If you have any programming experience at all, this chapter should be a breeze.

3.2 Statements

Python is written in the form of code statements as are other languages. However you do not need to put a semi-colon at the end of each statement. What the Python interpreter looks for are carriage returns (that's when you press the ENTER key on the keyboard). As long as you hit ENTER after each statement so that the next one is on a new line, the Python interpreter can tell where one statement ends and the other begins.

In addition statements within a code block need to be indented, such as statements inside a FOR loop. In languages such as C++ you use curly braces to signal the beginning and end of blocks of code whereas in Python you indent the code. Python is very serious about this, if you don't indent code which is nested inside of something else (such as statements in a function definition or a loop) you will receive a lot of error messages.

Within a statement you can decide how much whitespace you wish to leave. So a=b+c can be written as a = b + c (notice the spaces between each character)

3.3 Variables and assignment statements

In some programming languages such as C++ and Java, variables are strongly typed. This means that you don't just name a variable; you also declare a type for the variable. So for example if you were to create an integer variable 'x' in C++ and assign it a value of 5, your code would look something like the following:

```
int x;
x=5;
```

However Python is not strongly typed. This means you don't state what type of data the variable holds, you simply give it a name. It could be an integer, a float or a String, but you wouldn't tell Python, it would figure it out on its own. So if you were to create an integer variable x in Python and assign it a value of 5 you would simply write:

```
x=5
```

In addition Python doesn't mind if you try to do things like multiplying a whole number with a float. Some languages object to this type of mixing and require an explicit cast. Python is also able to recognize String variables, and concatenates them if you add them. So a statement like

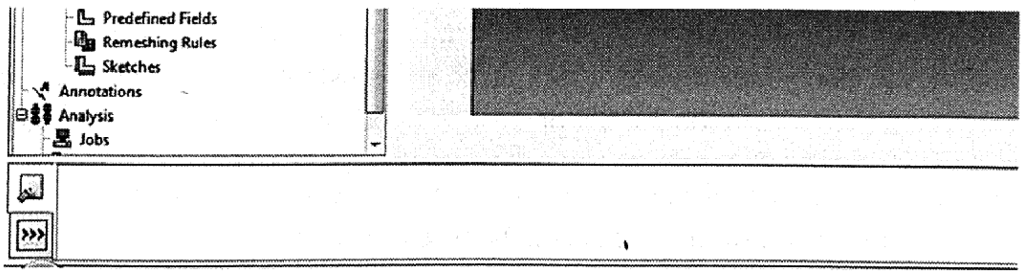
```
greeting = 'h' + 'ello'
```

stores the value 'hello' in the variable 'greeting'.

Let's work through an example to understand some of these concepts.

Example 4.1 - Variables

Open up Abaqus CAE. In the lower half of the window below the viewport you see the message area. If you look to the left of the message area you see two tabs, one for "Message area" and the other for "Kernal Command Line Interface".

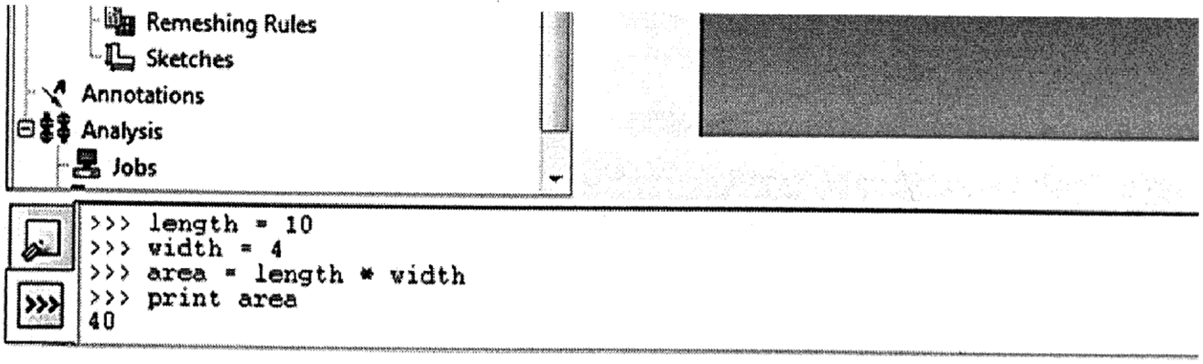


Click the second one. You see the kernel command prompt which is a ">>>" symbol.

Type the following lines, hitting the ENTER key on your keyboard after each.

```
>>> length = 10
>>> width = 4
>>> area = length * width
>>> print area
```

The number 40 is displayed. Since we set length to 10 and width to 4, the area being the product of the two is 40. The print statement displays the value stored in the area variable. The following image displays what you should see on your own screen.



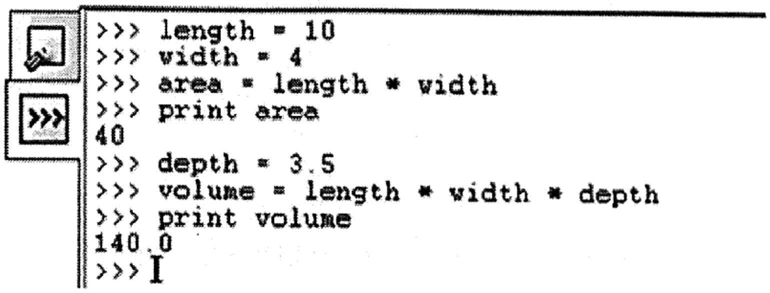
```
>>> length = 10
>>> width = 4
>>> area = length * width
>>> print area
40
```

So you see the Python interpreter realized that the variables 'length' and 'width' store integers without you needing to specify what type of variables they are. In addition when assigning their product to the variable 'area', it decided for itself that 'area' was also an integer.

What if you had combined integers and floats? Add on the following statements:

```
>>> depth = 3.5
>>> volume = length * width * height
>>> print volume
```

The output is 140.0 . Note the ".0" at the end. Since your height variable was a float (number with decimal point in layman terms), the volume variable is also a float, even though two of its factors 'length' and 'width' are integers.

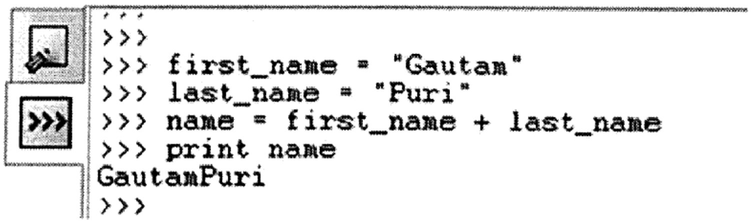


```
>>> length = 10
>>> width = 4
>>> area = length * width
>>> print area
40
>>> depth = 3.5
>>> volume = length * width * depth
>>> print volume
140.0
>>> 
```

Let's experiment with Strings. Add the following lines

```
>>> first_name = "Gautam"
>>> last_name = "Puri"
>>> name = first_name + last_name
```

The output is "GautamPuri". Notice that we did not tell Python that 'first_name' and 'last_name' are String variables, it figured it out on its own. Also when we added them together, Python concatenated them together.



```
>>>
>>> first_name = "Gautam"
>>> last_name = "Puri"
>>> name = first_name + last_name
>>> print name
GautamPuri
>>>
```

As you can tell from this example, not having to define variable types makes it a lot less painful to type code in Python than in a language such as C++. This also saves a lot of heartache when dealing with instances of classes so that you don't have to define each variable as being an object of a class. If you don't know what classes, instances and objects are, you will find out in the section on "Classes" a few pages down the line. But first let's talk about lists and dictionaries.

3.4 Lists

Arrays are a common collection data type in just about every high level programming language so I expect you've dealt with them before and know why they're useful. You aren't required to use them to write Abaqus scripts, but chances are you will want to store information in similar collections in your scripts. Let's explore a collection type in Python known as a List.

In a list you store multiple elements or data values and can refer to them with the name of the list variable followed by an index in square brackets []. The lowest index is 0. Note that you can store all kinds of data types, such as integers, floats, Strings, all in the same list. This is different from languages such as C, C++ and Java where all array members must be of the same data type. Lists have many built-in functions, some of which are:

- len() – returns the number of elements in the list
- append(x) – adds x to the end of the list making it the last element

- remove(y) – removes the first occurrence of y in the list
- pop(i) – removes the element at index [i] in the list, also returns it as the return value

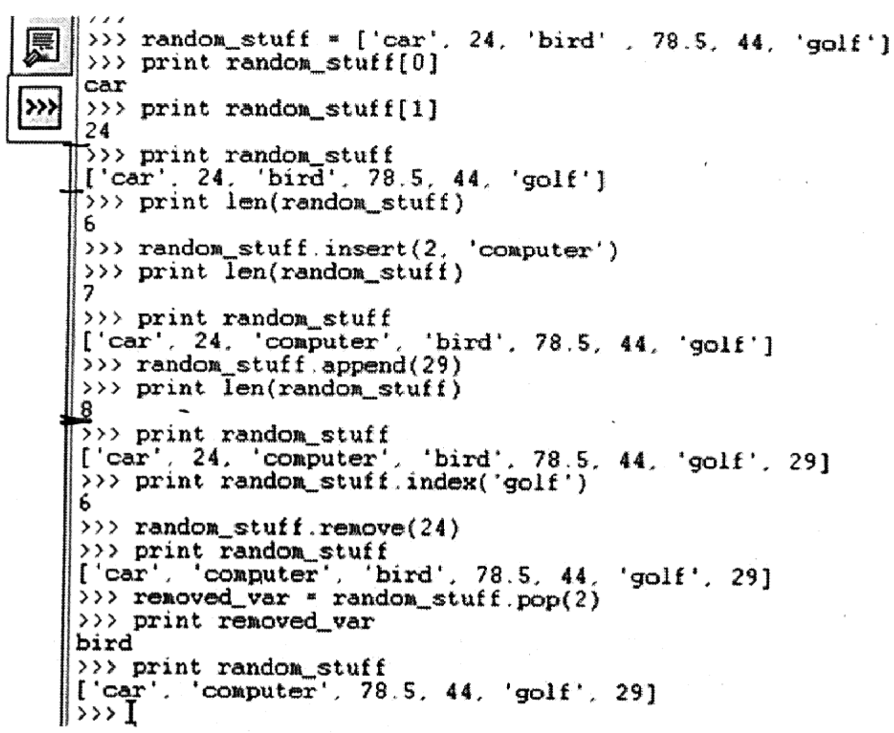
Let's work through an example.

Example 4.2 - Lists

In the 'Kernel Command Line Interface' tab of the lower panel of the window, type in the following statements hitting ENTER after each.

```
>>>random_stuff = ['car', 24, 'bird' , 78.5, 44, 'golf']
>>> print random_stuff[0]
>>> print random_stuff[1]
>>> print random_stuff
>>> print len(random_stuff)

>>> random_stuff.insert(2, 'computer')
>>> print len(random_stuff)
>>> print random_stuff
>>> random_stuff.append(29)
>>> print len(random_stuff)
>>> print random_stuff
>>> random_stuff.remove(24)
>>> print random_stuff

>>> removed_var = random_stuff.pop(2)
>>> print removed_var
>>> print random_stuff
```

Your output will be as displayed the following figure. Note that the lowest index is 0, not 1, which is why random_stuff[0] refers to the first element 'car'. The len() function returns the number of elements in the list. The append() function adds on whatever is passed to it as an argument to the end of the list. The remove() function removes the element that matches the argument you pass it. And the pop() function removes the element at the index position you pass it as an argument.



```
>>> random_stuff = ['car', 24, 'bird' , 78.5, 44, 'golf']
>>> print random_stuff[0]
car
>>> print random_stuff[1]
24
>>> print random_stuff
['car', 24, 'bird', 78.5, 44, 'golf']
>>> print len(random_stuff)
6
>>> random_stuff.insert(2, 'computer')
>>> print len(random_stuff)
7
>>> print random_stuff
['car', 24, 'computer', 'bird', 78.5, 44, 'golf']
>>> random_stuff.append(29)
>>> print len(random_stuff)
8
>>> print random_stuff
['car', 24, 'computer', 'bird', 78.5, 44, 'golf', 29]
>>> print random_stuff.index('golf')
6
>>> random_stuff.remove(24)
>>> print random_stuff
['car', 'computer', 'bird', 78.5, 44, 'golf', 29]
>>> removed_var = random_stuff.pop(2)
>>> print removed_var
bird
>>> print random_stuff
['car', 'computer', 78.5, 44, 'golf', 29]
>>>
```

3.5 Dictionaries

Dictionaries are a collection type, just as lists are, but with a slightly different feel and syntax. You do not really need to create your own dictionaries in order to write scripts in Abaqus, you can accomplish most tasks with a list, but you never know when you might prefer to use a dictionary. More importantly though, Abaqus stores a number of its own constructs in the form of dictionaries, and you will be accessing these regularly, hence knowing what dictionaries are will give you a better understanding of scripting.

Dictionaries are sets of key:value pairs. To access a value, you use the key for that value. This is analogous to using an index position to access the data within a list. The difference is that keeping track of the key to access a value may be easier in a certain situation than remembering the index location of a value in a list. Since there are no index positions, dictionaries are unordered.

To remove a key:value pair, you use the del command. To remove all the key:value pairs, you use the clear command.

Aside: If you've worked with the programming language PERL, dictionaries are very similar to the hash collections. If you're coming from a Java environment, dictionaries are similar to the Hashtable class.

An example should make things clear.

Example 4.3 – Dictionaries

In the 'Kernel Command Line Interface', type in the following statements hitting ENTER after each. You will see an output after each print statement.

```
>>>names_and_ages = {'John':23, 'Rahul':15, 'Lisa':55}
>>> print names_and_ages['John']
>>> print names_and_ages['Rahul']
>>> print names_and_ages
>>> del names_and_ages['John']
>>> print names_and_ages
>>> names_and_ages.clear()
>>> print names_and_ages
```

Here names_and_ages is your dictionary variable. In it you store 3 keys, 'John', 'Rahul' and 'Lisa'. You store their ages as the values. This way if you wish to access Lisa's age, you would write names_and_ages['Lisa'].

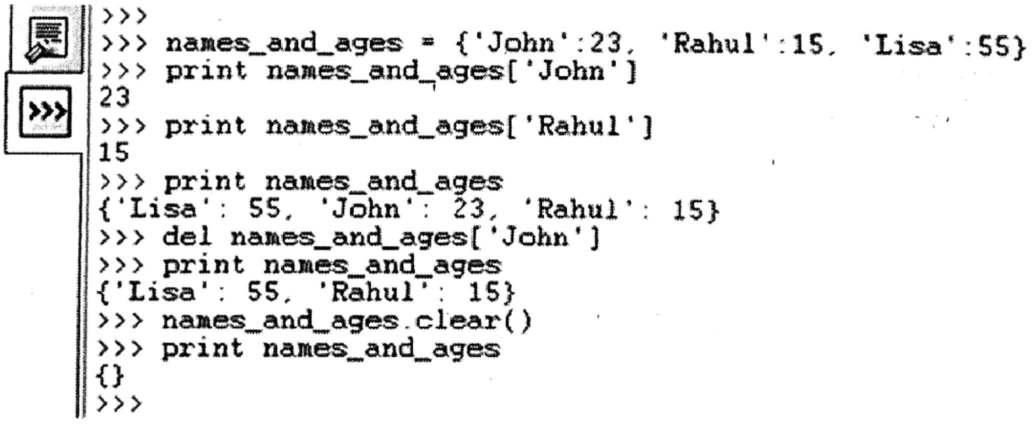
The del command removes the key:value pair 'John':23, leaving only Rahul and Lisa. The clear command removes all the key value pairs leaving you with an empty dictionary {}.

Note that since the dictionary is unordered, the first statement could instead have been written as

```
>>> names_and_ages = {'Rahul':15, 'Lisa':55, 'John':23}
```

and it would have made no difference since your values (ages) are still bound to the correct keys (names).

The following image displays what you should see.



```
>>>
>>> names_and_ages = {'John':23, 'Rahul':15, 'Lisa':55}
>>> print names_and_ages['John']
23
>>> print names_and_ages['Rahul']
15
>>> print names_and_ages
{'Lisa': 55, 'John': 23, 'Rahul': 15}
>>> del names_and_ages['John']
>>> print names_and_ages
{'Lisa': 55, 'Rahul': 15}
>>> names_and_ages.clear()
>>> print names_and_ages
{}
>>>
```

So how does Abaqus use dictionaries?

You're probably wondering when you would actually use dictionaries. You will be using them all the time, and already did so more than once in the cantilever beam example of Chapter 1 (Example 1.2), except you didn't know it at the time. Here's a block of code from the example.

```
# ----------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Cantilever Beam')
beamModel = mdb.models['Cantilever Beam']

# -----------------------------------------------------
```

Look closely at the statement

```
beamModel = mdb.models['Cantilever Beam']
```

Here you see the model database 'mdb' has a property called 'models'. This property is a dictionary object containing a key:value pair for each model you create. The model name itself is the 'key', and the value is an instance of the model object.

You know that the syntax to access a value in a dictionary is *dictionary_name['key']*. So when you want the script to refer to the cantilever beam model you say *models['Cantilever Beam']*.

To be a little more precise, models in not exactly a dictionary object but a subclass of a dictionary object. What does that mean? Well, to put it simply, it means that the programmers at Abaqus created a new class that has the same properties and methods as dictionary, but also has one or more new properties and methods that they defined. For example the changeKey() method that changes the name of the key from 'Model-1' to 'Cantilever Beam' is not native to Python dictionaries, it has been created by programmers at Abaqus. You don't have to worry about how they did it unless you are a computer science buff, in which case google 'subclassing in Python'. As far as a user/scripter is concerned, the 'models' object works similar to a dictionary object with a few enhancements. Also in Abaqus these enhanced dictionaries are referred to as 'repositories'. You will hear me use this word a lot when we start dissecting scripts.

Let's look at another block of code from Example 1.2.

```
# -------------------------------------------------------
# Create the history output request

# we try a slightly different method from that used in field output request
# create a new history output request called 'Default History Outputs' and assign
both the step and the variables
beamModel.HistoryOutputRequest(name='Default History Outputs', createStepName='Apply
Load', variables=PRESELECT)

# now delete the original history output request 'H-Output-1'
del beamModel.historyOutputRequests['H-Output-1']

# -------------------------------------------------------
```

Look closely at the statement

```
del beamModel.historyOutputRequests['H-Output-1']
```

Notice that your model beamModel has a dictionary or 'repository' (subclass of a dictionary) called historyOutputRequests. One of the key:value pairs has a key 'H-Output-1', and is referred to as *historyOutputRequests['H-Output-1']*. In the Abaqus Scripting Interface you will often find aspects of your model stored in repositories. For the record, in this statement the 'H-Output-1' key:value pair in the repository is being deleted using the del command.

3.6 Tuples

We've covered lists and dictionaries so far. Tuples are a third type of collection tool in Python. They are similar to lists, except once a tuple is created it cannot be edited. You cannot add on elements: commands such as append() do not work on it. And you cannot delete elements either using the del command. To access the elements of a tuple you use the same index notation as you do for lists. Let's work an example.

Example 4.4 – Tuples

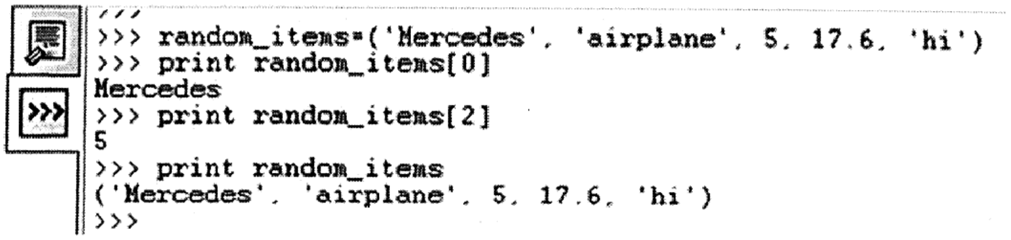
In the 'Kernel Command Line Interface' type in the following statements hitting ENTER after each. You will see an output after each print statement.

```
>>>random_items = ('Mercedes', 'airplane', 5, 17.6, 'hi')
>>> print random_items[0]
>>> print random_items[2]
>>> print random_items
```

A tuple called random_items is created which contains integers, floats and Strings. Note that you use arcs for parentheses, whereas if random_items was a list you would instead use square brackets as:

```
random_items = ['Mercedes', 'airplane', 5, 17.6, 'hi']
```

The following image displays the outputs of the above statements:



```
>>> random_items=('Mercedes', 'airplane', 5, 17.6, 'hi')
>>> print random_items[0]
Mercedes
>>> print random_items[2]
5
>>> print random_items
('Mercedes', 'airplane', 5, 17.6, 'hi')
>>>
```

So how does Abaqus use tuples?

Well just as in the case of dictionaries, you have actually already used tuples in the cantilever beam example of Chapter 1, you just didn't know it. Let's look at a block of code from that example.

```
# ---------------------------------------------------------
# Apply pressure load to top surface

# First we need to locate and select the top surface
# We place a point somewhere on the top surface based on our knowledge of the
geometry
top_face_pt_x = 0.2
top_face_pt_y = 0.1
top_face_pt_z = 2.5
```

```
top_face_pt = (top_face_pt_x,top_face_pt_y,top_face_pt_z)
```

Here you create 3 variables top_face_pt_x, top_face_pt_y and top_face_pt_z, and then store them in a tuple called top_face_pt. At this point you do not need to understand why the above statements were used, we will examine these statements in subsequent chapters. I only wish to point out what tuples are and the fact that you will be using them regularly when writing scripts.

3.7 Classes, Objects and Instances

When you run scripts in Abaqus you invariably use built-in methods provided by Abaqus to perform certain tasks. All of these built-in methods are stored in containers called classes. You often create an "instance" of a class and then access the built-in methods which belong to the class or assign properties using it. So it's important for you to have an understanding of how this all works.

Python is an object oriented language. If you've programmed in C++ or Java you know what object oriented programming (OOP) is all about and can breeze through this section. On the other hand if you're used to procedural languages such as C or MATLAB you've probably never worked with objects before and the concept will be a little hard to grasp at first. (Actually MATLAB v2008 and above supports OOP but it's not a feature known by the majority of its users).

For the uninitiated, a class is a sort of container. You define properties (variables) and methods (functions) for this class, and the class itself becomes a sort of data type, just like integer and String are data types. When you create a variable whose data type is the class you've defined, you end up creating what is called an object or an instance of the class. The best way to understand this is through an example.

Example 4.5 – 'Person' class

In the following example, assume we have a class called 'Person'. This class has some properties, such as 'weight', 'height', 'hair' color and so on. This class also has some methods such as 'exercise()' and 'dyeHair()' which cause the person to lose weight or change hair color.

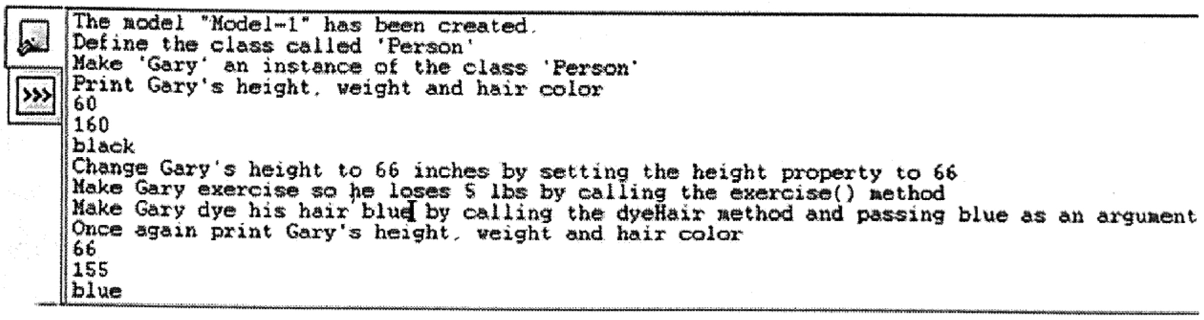
Once we have this basic framework of properties and methods (called the class definition), we can assign an actual person to this class. We can say Gary is a 'Person'. This means Gary has properties such as height, weight and hair color. We can set Gary's height by using a statement such as Gary.height = 68 inches. We can also make Gary exercise by saying Gary.exercise() which would cause Gary.weight to reduce. Gary is "an object of type Person" or "an instance of the Person class".

Open up notepad and type out the following script

```
print "Define the class called 'Person'"

class Person:
        height = 60
        weight = 160
        hair_color = "black"

        def exercise(self):
                self.weight = self.weight - 5

        def dyeHair(self, new_hair_color):
                self.hair_color = new_hair_color

print "Make 'Gary' an instance of the class 'Person'"
Gary = Person()

print "Print Gary's height, weight and hair color"
print Gary.height
print Gary.weight
print Gary.hair_color

print "Change Gary's height to 66 inches by setting the height property to 66"
Gary.height = 66

print "Make Gary exercise so he loses 5 lbs by calling the exercise() method"
Gary.exercise()

print "Make Gary dye his hair blue by calling the dyeHair method and passing blue as
an argument"
Gary.dyeHair('blue')

print "Once again print Gary's height, weight and hair color"
print Gary.height
print Gary.weight
print Gary.hair_color
```

Open a new file in Abaqus CAE (**File > New Model Database > With Standard/Explicit Model**). In the lower half of the window, make sure you are in the "Message Area" tab, not the "Kernel Command Line Interface" tab. The print statements in our script will display here in the "message area" that's why you want it to be visible.

Run the script you just typed out (**File > Run Script...**). Your output will be as displayed in the following figure.



```
The model "Model-1" has been created.
Define the class called 'Person'
Make 'Gary' an instance of the class 'Person'
Print Gary's height, weight and hair color
60
160
black
Change Gary's height to 66 inches by setting the height property to 66
Make Gary exercise so he loses 5 lbs by calling the exercise() method
Make Gary dye his hair blue by calling the dyeHair method and passing blue as an argument
Once again print Gary's height, weight and hair color
66
155
blue
```

Let's analyze the script in detail. The first statement is

```
print "Define the class called 'Person'"
```

This basically prints "Define the class called 'Person'" in the message window using the 'print' command. Hence that is the first message displayed. The following statements define the class:

```
class Person:
    height = 60
    weight = 160
    hair_color = "black"

    def exercise(self):
        self.weight = self.weight - 5

    def dyeHair(self, new_hair_color):
        self.hair_color = new_hair_color
```

A class named 'Person' has been created. It has been given the properties (variables) 'height', 'width' and 'hair_color', which have been assigned initial values of 60 inches, 160 lbs, and the color black.

In addition two methods (functions) have been defined, 'exercise()' and 'dyeHair()'. The 'exercise()' method causes the weight of the person to decrease by 5 lbs. The 'dyeHair()' function causes 'hair_color' to change to whatever color is passed to that function as the argument 'new_hair_color'.

What's with the word 'self'? In Python, every method in a class receives 'self' as the first argument, that's a rule. The word 'self' refers to the instance of the class which will be created later. In our case this will be 'Gary'. When we create 'Gary' as an instance of the 'Person' class, *self.weight* translates to *Gary.weight* and *self.hair_color* translates to *Gary.hair_color*. In object oriented languages like C++ and Java you do not pass self as an argument, this is a feature unique to the Pythons syntax and might even be a little annoying at first.

```
print "Make 'Gary' an instance of the class 'Person'"
Gary = Person()
```

These statements define Gary as an instance of the Person class, and also print a comment to the message area indicating this fact.

```
print "Print Gary's height, weight and hair color"
print Gary.height
print Gary.weight
print Gary.hair_color
```

We then display Gary's height, weight and hair_color which are currently default values. Notice how we refer to each variable with the instance name followed by a dot "." symbol followed by the variable name. The format is *InstanceName.PropertyName*.

These statements make the following lines appear on the screen:

```
Print Gary's height, weight and hair color"
60
160
black
```

```
print "Change Gary's height to 66 inches by setting the height property to 66"
Gary.height = 66
```

We now change Gary's height to 66 inches by using an assignment statement on the 'Gary.height' property. We print a comment regarding this to the message area.

```
print "Make Gary exercise so he loses 5 lbs by calling the exercise() method"
Gary.exercise()
```

These lines call the exercise function and display a comment in the message area. Notice that you use the format *InstanceName.MethodName()*. Although we don't appear to pass any arguments to the function (there's nothing in the parenthesis), internally the Python interpreter is passing the instance 'Gary' as an argument. This is why in the function definition we had the word 'self' listed as an argument. Why does the interpreter pass 'Gary' as an argument? Because you could potentially define a number of instances of the Person class in addition to Gary, such as 'Tom', 'Jill', 'Mr. T', and they will all have the same 'exercise()' method. So then if you were to call *Tom.exercise()*, it would be Tom's weight that would reduce while Gary's would remain unaffected.

If you look once again at the definition of the 'exercise()' method in the Person class, you'll notice that it decreases the weight of the instance by 5 lbs. So Gary's weight should now be 155 lbs, down 5 lbs from before.

```
print "Make Gary dye his hair blue by calling the dyeHair method and passing blue as
an argument”
Gary.dyeHair('blue')
```

These lines call the 'dyeHair()' function and display a comment in the message area. The difference you notice between the 'exercise()' and 'dyeHair()' functions is that you pass a hard argument to 'dieHair()' telling it exactly what color you wish to dye the individuals hair. Internally an argument of 'self' is also passed.

Take another look at the definition of the 'dyeHair()' method in the 'Person' class. You'll notice that the variable being passed as an argument is assigned to the 'hair_color'. So Gary's hair color should now have changed from black to blue.

```
print "Once again print Gary's height, weight and hair color"
print Gary.height
print Gary.weight
print Gary.hair_color
```

We print out Gary's height, weight and hair color again to notice the changes. The 'Gary.height' statement was used to reset his height to 66 inches, the 'exercise()' method was used to reduce his weight to 155 lbs, and the 'dyeHair('blue')' method should have changed his hair color to blue. These print statements display the property values in the message area. The output is what you expect:

```
Once again print Gary's height, weight and hair color
66
155
blue
```

Hopefully this example has made the concept of classes and instances clear to you. There's a lot more to OOP than this, we've only touched the surface, but that's because you only need a basic understanding of OOP to write Abaqus scripts. In none of our examples will you actually define a new class of your own.

So why learn about classes, objects and instances?

Now that you've understood classes you're probably wondering why I bothered telling you about them. Well you won't need to define your own classes, but understanding what they are will help you understand the Abaqus Scripting Interface. If you look back at the cantilever beam script (Example 1.2) of Chapter 1, you'll notice many spots where instances of a class are created and properties/methods of the class are accessed using the dot operator '.'

For instance, look at the following chunk of code of Example 1.2

```
# ----------------------------------------------------
# Create the part

import part

# a) Sketch the beam cross section using rectangle tool
beamProfileSketch = beamModel.ConstrainedSketch(name='Beam CS Profile', sheetSize=5)
beamProfileSketch.rectangle(point1=(0.1,0.1), point2=(0.3,-0.1))
```

Consider the statement

```
beamProfileSketch = beamModel.ConstrainedSketch(…some parameters…)
```

It creates a new instance of the 'ConstrainedSketch()' class called 'beamProfileSketch'. This is similar to our example where we created an instance called 'Gary' of the 'Person' class using the statement *Gary = Person()*.

Consider the next statement:

```
beamProfileSketch.rectangle()
```

It calls the 'rectangle()' method of the 'ConstrainedSketch()' class. This is similar to our example where we called the 'exercise()' method of our instance 'Gary' using 'Gary.exercise()'.

So you see a basic understanding of classes and OOP helps you understand the script better. All of Abaqus's scripting libraries have been programmed in the form of classes with properties and methods. So when you want to create something like a sketch, you need to create an instance of the 'ConstrainedSketch()' object. And when you wish to use the built in functions such as 'rectangle()', you need to reference them from an instance of a class that defines the method. All this makes a lot more sense now that you understand OOP.

Aside: If you were looking closely at the code statements we just discussed, you might have noticed that there was a slight difference in syntax between the cantilever beam example and the 'person class' example. It is a little more complicated; notice we do not say *beamProfileSketch = ConstrainedSketch()*, instead we say *beamProfileSketch = beamModel.ConstrainedSketch()*. So what is going on here? Well, 'beamModel' is an instance of the 'models' class. And the 'models' class has a member called 'sketches' which is a repository of 'ConstrainedSketch()' objects. And it's one of those 'ConstrainedSketch()' classes that you are making an instance of. Hence you refer to it with *beamModel.ConstrainedSketch()*. I guess one way of thinking of this is that you have classes nested within classes.

If this sounds too confusing don't worry about it. As I said I've only provided you with a basic overview of the OOP concept, enough to gain a basic understanding and get the job done. If you want to better understand how Abaqus classes have been programmed by SIMULIA then you might wish to learn more about the ins and outs of OOP with a book on programming or some internet resources. But if all you're trying to do is write scripts for your Abaqus simulations you don't need to spend time asking questions like why you create a constrained sketch using *models.ConstrainedSketch()* as opposed to just *ConstrainedSketch()*. The fact is you do, that's how Abaqus's libraries have been coded, and it's ok to just accept it. Just as you accept that the print command always prints to the message area even though you don't know how Abaqus does it.

If for the sake of personal satisfaction you wish to dig deeper into OOP, one way to learn it is the way most of us did – by learning the standard fare object oriented language like C++ or Java. Then again with the popularity and support

Python enjoys today, you could instead read a book or an online tutorial on full-fledged Python programming that covers OOP in detail.

Abstraction in OOP

One final concept I wish to point out about OOP is the concept of abstraction. In our example with the 'Person' class, you noticed that when we wanted 'Gary' to lose 5 lbs we used the statement *Gary.exercise()*. And then the 'exercise()' method was defined in such a way that it decreased *Gary.weight* by 5. Why did we bother calling *Gary.exercise()* when we could instead have written *Gary.weight = Gary.weight – 5* ?

Well it's because it was the better, or safer, way to do it. We defined an exercise method to do the job, hence it made sense to use it. In the programming world often a class is written by one programmer and stored in a library and the instance of the class is created and used by another programmer. If there was no method defined in the class to perform a task such as losing weight, the second programmer would have to read through the code of the class and figure out which variable to change. If on the other hand the programmer who made the original class defined a method that does what the second programmer needs, then programmer number 2 does not have to go through all the code of the class, all he needs to know is what functions/methods are available to him. In fact it is possible for the coder of the original class to make some of the variables private, meaning that they can only be changed by calling a method. This is known as "abstraction", or basically hiding the internal workings of the class.

We're not going to go into further details of how this works, mostly because knowing that probably wouldn't help you write better scripts. The programmers at SIMULIA who wrote the classes and libraries for the Abaqus Scripting Interface decided what variables are public and which ones are only accessible through a built in function. As a user of the software you only need to know when you can access a variable or when to use the function. The way to find out is by looking up the members and methods of the class you are working with in the Abaqus documentation. Whatever you can access will be listed there. All this will become clearer with examples, and quite frankly you won't even be thinking about abstraction and how it works, you'll just write code the way you know it works in Abaqus.

3.8 What's next?

In this chapter you learned :

- how to define variables and write code statements,
- how to create collection types – lists, dictionaries, and tuples,
- object oriented programming (OOP) concepts – classes, instances, data abstraction

You also referred to code snippets from the cantilever beam example from Chapter 1 to see the syntax in action.

You now understand some of the Python syntax behind much of Example 1.2. However you still don't understand the Abaqus specific commands and methods that were used. In the next chapter we're going to take a closer look at the cantilever beam example and try to make sense of it all.

4

The Basics of Scripting – Cantilever Beam Example

4.1 Introduction

Now that you have the required understanding of Python syntax, we can plunge into scripting. Every script you write will perform a different task and no two scripts will be alike. However they all follow the same basic methodology. The best way to understand this is to go through the cantilever beam script in detail.

4.2 A basic script

Since we already have the cantilever beam example from Chapter 2 we shall work our way through it, statement by statement. Not only will you understand exactly what is going on in the script, you will also learn some of the most important methods that you will likely use in every script you write.

Example 4.1 – Cantilever Beam

For your convenience a copy of the code from Chapter 2 has been listed here.

```
# ********************************************************************************
# Cantilever Beam bending under the action of a uniform pressure load

# Created for the book "Python Scripts for Abaqus - Learn by Example"
# Author: Gautam Puri
# ********************************************************************************

from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)
```

```
# -----------------------------------------------------------------------
# Create the model
mdb.models.changeKey(fromName='Model-1', toName='Cantilever Beam')
beamModel = mdb.models['Cantilever Beam']

# -----------------------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the beam cross section using rectangle tool
beamProfileSketch = beamModel.ConstrainedSketch(name='Beam CS Profile',
                                                sheetSize=5)
beamProfileSketch.rectangle(point1=(0.1,0.1), point2=(0.3,-0.1))

# b) Create a 3D deformable part named "Beam" by extruding the sketch
beamPart=beamModel.Part(name='Beam', dimensionality=THREE_D,
                                     type=DEFORMABLE_BODY)
beamPart.BaseSolidExtrude(sketch=beamProfileSketch, depth=5)

# -----------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs
# modulus and poissons ratio
beamMaterial = beamModel.Material(name='AISI 1005 Steel')
beamMaterial.Density(table=((7872, ),       ))
beamMaterial.Elastic(table=((200E9, 0.29), ))

# -----------------------------------------------------------------------
# Create solid section and assign the beam to it

import section

# Create a section to assign to the beam
beamSection = beamModel.HomogeneousSolidSection(name='Beam Section',
                                                material='AISI 1005 Steel')

# Assign the beam to this section
beam_region = (beamPart.cells,)
beamPart.SectionAssignment(region=beam_region, sectionName='Beam Section')

# ------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
```

```
beamAssembly = beamModel.rootAssembly
beamInstance = beamAssembly.Instance(name='Beam Instance', part=beamPart,
                                                             dependent=ON)

# -------------------------------------------------------------------------
# Create the step

import step

# Create a static general step
beamModel.StaticStep(name='Apply Load', previous='Initial',
                     description='Load is applied during this step')

# -------------------------------------------------------------------------
# Create the field output request

# Change the name of field output request 'F-Output-1' to 'Selected Field Outputs'
beamModel.fieldOutputRequests.changeKey(fromName='F-Output-1',
                                        toName='Selected Field Outputs')

# Since F-Output-1 is applied at the 'Apply Load' step by default, 'Selected Field
# Outputs' will be too
# We only need to set the required variables
beamModel.fieldOutputRequests['Selected Field Outputs'].setValues(variables=('S',
                                                  'E', 'PEMAG', 'U', 'RF', 'CF'))

# -------------------------------------------------------------------------
# Create the history output request

# We try a slightly different method from that used in field output request
# Create a new history output request called 'Default History Outputs' and assign
# both the step and the variables
beamModel.HistoryOutputRequest(name='Default History Outputs',
                               createStepName='Apply Load', variables=PRESELECT)

# Now delete the original history output request 'H-Output-1'
del beamModel.historyOutputRequests['H-Output-1']

# -------------------------------------------------------------------------
# Apply pressure load to top surface

# First we need to locate and select the top surface
# We place a point somewhere on the top surface based on our knowledge of the
# geometry
top_face_pt_x = 0.2
top_face_pt_y = 0.1
top_face_pt_z = 2.5
top_face_pt = (top_face_pt_x,top_face_pt_y,top_face_pt_z)

# The face on which that point lies is the face we are looking for
top_face = beamInstance.faces.findAt((top_face_pt,))
```

```
# We extract the region of the face choosing which direction its normal points in
top_face_region=regionToolset.Region(side1Faces=top_face)

# Apply the pressure load on this region in the 'Apply Load' step
beamModel.Pressure(name='Uniform Applied Pressure', createStepName='Apply Load',
                   region=top_face_region, distributionType=UNIFORM,
                   magnitude=10, amplitude=UNSET)

# -----------------------------------------------------------------------
# Apply encastre (fixed) boundary condition to one end to make it cantilever

# First we need to locate and select the top surface
# We place a point somewhere on the top surface based on our knowledge of the
# geometry
fixed_end_face_pt_x = 0.2
fixed_end_face_pt_y = 0
fixed_end_face_pt_z = 0
fixed_end_face_pt = (fixed_end_face_pt_x,fixed_end_face_pt_y,fixed_end_face_pt_z)

# The face on which that point lies is the face we are looking for
fixed_end_face = beamInstance.faces.findAt((fixed_end_face_pt,))

# We extract the region of the face choosing which direction its normal points in
fixed_end_face_region=regionToolset.Region(faces=fixed_end_face)

beamModel.EncastreBC(name='Encaster one end', createStepName='Initial',
                                              region=fixed_end_face_region)

# -----------------------------------------------------------------------
# Create the mesh

import mesh

# First we need to locate and select a point inside the solid
# We place a point somewhere inside it based on our knowledge of the geometry
beam_inside_xcoord=0.2
beam_inside_ycoord=0
beam_inside_zcoord=2.5

elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN, secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)


beamCells=beamPart.cells
selectedBeamCells=beamCells.findAt((beam_inside_xcoord,beam_inside_ycoord,
                                                        beam_inside_zcoord),)
beamMeshRegion=(selectedBeamCells,)
beamPart.setElementType(regions=beamMeshRegion, elemTypes=(elemType1,))

beamPart.seedPart(size=0.1, deviationFactor=0.1)
beamPart.generateMesh()

# -----------------------------------------------------------------------
# Create and run the job

import job

# Create the job
mdb.Job(name='CantileverBeamJob', model='Cantilever Beam', type=ANALYSIS,
        explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE,
        description='Job simulates a loaded cantilever beam',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)

# Run the job
mdb.jobs['CantileverBeamJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['CantileverBeamJob'].waitForCompletion()

# End of run job
# -----------------------------------------------------------------------
# Post processing

import visualization

beam_viewport = session.Viewport(name='Beam Results Viewport')
beam_Odb_Path = 'CantileverBeamJob.odb'
an_odb_object = session.openOdb(name=beam_Odb_Path)
beam_viewport.setValues(displayedObject=an_odb_object)
beam_viewport.odbDisplay.display.setValues(plotState=(DEFORMED, ))
```

4.3 Breaking down the script

The script executes from top to bottom in Python. I have included comments all over the script to explain what's going on. Lines that start with the hash (#) symbol are treated as comments by the interpreter. Make it a point to comment your code so you know what it means when you look at it after a few months or another member of your team has to continue what you started.

Observe the layout of the script. I have divided it into blocks or chunks of code clearly demarcated by:

```
#-------------------------------------------------------
# comment describing the block of code
```

Try reading these comments. You will realize that the script follows these steps:

1. Initialization (import required modules)
2. Create the model
3. Create the part
4. Define the materials
5. Create solid sections and make section assignments
6. Create an assembly
7. Create steps
8. Create and define field output requests
9. Create and define history output requests
10. Apply loads
11. Apply boundary conditions
12. Meshing
13. Create and run the job
14. Post processing

Let's explore each code chunk one at a time.

4.3.1 Initialization (import required modules)

The code block dealing with this step is listed below:

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)
```

We begin the script using a couple of 'from-import' statement.

The first import statement:

```
from abaqus import *
```

imports the **abaqus** module and creates references to all public objects defined by that module. Thus it makes the basic Abaqus objects accessible to the script. One of the things it provides access to is a default model database which is referred to by the variable **mdb**. You use this variable **mdb** in the next block of code which is the 'create the model' block. You need to insert this import statement in every Abaqus script you write.

The second import statement:

```
from abaqusConstants import *
```

is for making the symbolic constants defined by the Abaqus Scripting Interface available to the script. What are symbolic constants? They are variables with a constant value (hence the term constant) that have been given a name that makes more sense to a user (hence the term symbolic) but have some meaning to Abaqus. Internally they might be integer or float variables. But for the sake of clarity of code they are displayed as a word in the English language. Since they are constants they cannot be modified

We use symbolic constants in the script. Look at the relevant lines in the script where the part is created. Notice the statement:

```
beamPart=beamModel.Part(name='Beam', dimensionality=THREE_D,
                                     type=DEFORMABLE_BODY)
```

Both **THREE_D** and **DEFORMABLE_BODY** are symbolic constants defined in the **abaqusConstants** module. So if we did not import this module into our script we would get an error as the interpreter would not recognize these symbolic constants. So place this import statement in every script you write.

The third import statement:

```
        import regionToolset
```

imports the **regionToolset** module so you can access its methods through the script. If you look at the 'create the loads' block, you will notice the statement:

```
top_face_region=regionToolset.Region(side1Faces=top_Plate)
```

We are using the **Region()** method defined in the **regionToolset** module. Hence the module needs to be imported otherwise you will receive an error. I tend to place this import statement in every script I write, whether or not the **Region()** method is used, just to be on the safe side.

Basically every script should have these 3 import statements placed in it at the top. You may not always need them, but by including them you spend less time thinking about whether or not you need them and more time writing useful code.

The fourth statement:

```
session.viewports['Viewport:1'].setValues(displayedObject=None)
```

blanks out the viewport. The viewport is the window in the Abaqus/CAE that displays the part you are working on. It allows Abaqus to display information to you visually. The viewport object is the object where the information about the viewport is stored such as what to display and how to do so.

The default name for the viewport is 'Viewport:1'. This is not only the name displayed to the user, it is the key for that viewport in the **viewports** dictionary/repository. Hence we refer to the viewport with the **viewports['Viewport:1']** notation. The method **setValues()** is a method of the **viewport** object that can be used to modify the viewport. It accepts two parameters, the **displayedObject** which defines what is displayed, and the **displayMode** which defines the layers (more about that later). When we set the **displayedObject** to **None**, that causes an empty viewport to be displayed.

4.3.2 Create the model

The following block creates the model

```
# ---------------------------------------------------------------------
# Create the model
mdb.models.changeKey(fromName='Model-1', toName='Cantilever Beam')
beamModel = mdb.models['Cantilever Beam']
```

As stated before, the variable **mdb** provides access to a default model database. This variable is made available to the script thanks to the

```
from abaqus import *
```

import statement we used earlier, hence you don't define it yourself.

The default model in Abaqus is always named 'Model-1', which is why when you open a new file you always see 'Model-1' in the model database tree on the left in the GUI.

The first statement:

```
mdb.models.changeKey(fromName='Model-1', toName='Cantilever Beam')
```

changes the name of the model from the default of 'Model-1' to 'Cantilever Beam'. **changeKey()** is a method of models which is in the model database, hence we refer to it using **mdb.models.changeKey()**.

If you recall from Chapter 3, the **models** repository is a subclass of a dictionary object which keeps track of model objects. As explained before, a subclass means that it has the same properties and methods of the dictionary object along with a few more properties and methods, such as **changeKey()**, that developers at SIMULIA decided to add in. The model name 'Model-1' is the key, while the value is a **model** object. The **changeKey()** method which is not native to Python essentially allows us to change the key to 'Cantilever Beam' while referring to the same model object.

The second statement:

```
beamModel = mdb.models['Cantilever Beam']
```

assigns our model to the **beamModel** variable. This is so that in future we do not have to keep referring to it as **mdb.models['Cantilever Beam']** but can instead just call it **beamModel**. Look at the 'create the part' block and notice the statement

```
beamProfileSketch = beamModel.ConstrainedSketch(name='Beam CS Profile',
                                                sheetSize=5)
```

Don't worry about what it means just yet, I only want to point out that if we did not define the variable **beamModel**, then the same statement would have to be written as:

```
beamProfileSketch = mdb.models['Cantilever Beam'].
                          ConstrainedSketch (name='Beam CS Profile, sheetSize=5)
```

which is a little bit longer. This type of syntax will get longer as we refer to properties and objects nested further down.

Of course you could choose to write things the long way, or you could do it my way.

4.3.3 Create the part

The following block of code creates the part

```
# -------------------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the beam cross section using rectangle tool
beamProfileSketch = beamModel.ConstrainedSketch(name='Beam CS Profile',
                                                sheetSize=5)
```

```
beamProfileSketch.rectangle(point1=(0.1,0.1), point2=(0.3,-0.1))

# b) Create a 3D deformable part named "Beam" by extruding the sketch
beamPart=beamModel.Part(name='Beam', dimensionality=THREE_D,
                                     type=DEFORMABLE_BODY)
beamPart.BaseSolidExtrude(sketch=beamProfileSketch, depth=5)
```

The first two statements

```
import sketch
import part
```

import the **sketch** and **part** modules into the script, thus providing access to the objects related to sketches and parts. As such you shouldn't be able to create a sketch or a part without these import statements but honestly if you leave them out in most cases Abaqus figures out what you are trying to do and appears to import these modules automatically without complaining. It is however recommended that you stay in the habit of including them because it's good programming practice and because you never know if an older or newer version of Abaqus will throw an error.

The statement

```
beamProfileSketch = beamModel.ConstrainedSketch(name='Beam CS Profile',
                                                 sheetSize=5)
```

creates a constrained sketch object by calling the ConstrainedSketch() method of the Model object. The sketch module defines ConstrainedSketch objects. The first argument is the **name** you wish to give the sketch, we're calling it 'Beam CS Profile'. This is used as the repository key given to our **ConstrainedSketch** object, just as 'Cantilever Beam' is the key for our model object. The second argument is the default **sheetsize**, which is a property you defined when manually performing the cantilever beam simulation in Abaqus/CAE. It sets the approximate size of the sheet, and therefore the grid you see when you are in the sketcher. Of course when you're working in a script the sheetsize isn't really important, that only helps you see things better when working in the GUI, but it's a required paramenter to the **ConstrainedSketch()** method hence you must give it a value. Note that the statement can be written without the words 'name' and 'sheetSize' as:

```
beamProfileSketch = beamModel.ConstrainedSketch('Beam CS Profile', 5)
```

It means the same thing to the interpreter; it just isn't as clear to someone reading your script. Also you'll have to make sure the arguments are passed in the correct order as is required by the method as stated in the documentation.

The statement

```
beamProfileSketch.rectangle(point1=(0.1,0.1), point2=(0.3,-0.1))
```

uses the **rectangle()** method of the **ConstrainedSketch** object to draw a rectangle on the sketch plane. The two parameters are the coordinates of the top left and bottom right corners of the rectangle. Note that the statement can also be written without the words **point1** and **point2** as:

```
beamProfileSketch.rectangle((0.1,0.1), (0.3,-0.1))
```

The statement

```
beamPart=beamModel.Part(name='Beam', dimensionality=THREE_D,
                                     type=DEFORMABLE_BODY)
```

uses the **Part()** method to create a **Part** object and place it in the parts repository. The first parameter 'Beam' is its **name** and its key in the repository. The second parameter, **dimensionality**, is set to a symbolic constant **THREE_D** which defines it to be a 3D part. It is defined to be of the **type** deformable body using the **DEFORMABLE_BODY** symbolic constant. In subsequent chapters you will define different parameters in place of these depending on the simulation. The created part instance is stored in the **beamPart** variable. If you haven't already guessed, the statement can also be written without the words **name**, **dimensionality**, and **type** as

```
beamPart=beamModel.Part('Beam', THREE_D, DEFORMABLE_BODY)
```

The statement

```
beamPart.BaseSolidExtrude(sketch=beamProfileSketch, depth=5)
```

creates a **Feature** object by calling the **BaseSolidExtrude()** method. What is a **Feature** object? Well, Abaqus is a feature based modeling system. The **Feature** object contains the parameters specified by the user, as well as the modifications made to the model by Abaqus based on those parameters. The **Feature** object is defined in the **Part** module hence you do not use an 'import feature' statement. The **BaseSolidExtrude()** method is used to create extrusions. The first parameter passed to it is our **ConstrainedSketch** object **beamProfileSketch**. Note that this must be a closed profile. The second parameter is the **depth** to which we wish to extrude our profile sketch. The statement can be written without the keywords **sketch** and **depth** as:

```
beamPart.BaseSolidExtrude(beamProfileSketch, 5)
```

4.3.4 Define the materials

The following block creates the material

```
# -------------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs
# modulus and poissons ratio
beamMaterial = beamModel.Material(name='AISI 1005 Steel')
beamMaterial.Density(table=((7872, ), ))
beamMaterial.Elastic(table=((200E9, 0.29), ))
```

```
import material
```

This statement imports the **material** module into the script providing access to objects and methods related to materials.

```
beamMaterial = beamModel.Material(name='AISI 1005 Steel')
```

This statement creates a **Material** object using the **Material()** method and places it in the **materials** repository. The parameter passed to the **Material()** method is the name given to the material, and the key used to refer to it in the **materials** repository. The **Material** object is assigned to the variable **beamMaterial**.

```
beamMaterial.Density(table=((7872, ), ))
```

This statement creates a **Density** object which specifies the density of the material by using the **Density()** method. The **Density** object is defined in the **material** module, hence you do not use an 'import density' statement. The argument passed to the **Density** method is supposed to be a table. Why a table? Well you might have a density that depends on temperature. In which case you would have a table in the form *((density1, temperature1), (density2,temperature2), (density3,temperature3))* and so on…

In our case we have one density which is not temperature dependent, but we must use the same format. So we can't say *table=7872*, we need to write *table=((7872,),)* where we leave a space after the first comma for *temperature1* (or rather the lack of it), and a space after the second comma for *(denstiy2, temperature2)*.This probably looks a little strange, and you will often generate a lot of syntax errors typing the wrong number of commas or parenthesis, so be aware of that. For the record, we can leave out the word 'table', but all the parentheses and commas in the statement will remain as they are:

```
beamMaterial.Density(((7872, ), ))
```

The statement:

```
beamMaterial.Elastic(table=((200E9, 0.29), ))
```

creates an **Elastic** object which specifies the elasticity of the material by using the **Elastic()** method. The **Elastic** object is defined in the **material** module, hence you do not use an *import elastic* statement. The argument passed to the **Elastic()** method must be a table just like the argument to the **Density()** method. The table must be of the form *((YM1, PR1), (YM2, PR2), (YM3, PR3))* and so on where *YM* is Young's modulus and *PR* is Poisson's ratio. For our material we have only one Young's modulus and one Poisson's ratio so we write *table=((200E9, 0.29),)* leaving a second comma there to indicate the spot for (YM2, PR2). The statement can be written without the keyword 'table' as:

```
beamMaterial.Elastic(((200E9, 0.29), ))
```

4.3.5 Create solid sections and make section assignments

The following code block creates the sections and makes assignments

```
# ----------------------------------------------------------------------
# Create solid section and assign the beam to it

import section

# Create a section to assign to the beam
beamSection = beamModel.HomogeneousSolidSection(name='Beam Section',
                                                material='AISI 1005 Steel')

# Assign the beam to this section
beam_region = (beamPart.cells,)
beamPart.SectionAssignment(region=beam_region, sectionName='Beam Section')
```

```
import section
```

This statement imports the **section** module making its properties and methods accessible to the script.

```
beamSection = beamModel.HomogeneousSolidSection(name='Beam Section',
                                                material='AISI 1005 Steel')
```

This statement creates a **HomogeneousSolidSection** object using the **HomogeneousSolidSection()** method. These are defined in the section module. The first parameter given to the method is **name**, which is used as the repository key. The second parameter is **material**, which has been defined in the 'define the materials' code block. Note that this **material** parameter must be a String, it cannot be a **Material** object. That means we cannot say *material=beamMaterial* even though we had defined the **beamMaterial** variable to point to our beam material, because **beamMaterial** is a **Material** object. 'AISI1005 Steel' on the other hand is a String, and it is the key assigned to that material in the **materials** repository.

The statement

```
beam_region = (beamPart.cells,)
```

is used to find the **cells** of the beam. The **cell** object defines the volumetric regions of a geometry. **Part** objects have cells. **beamPart.cells** refers to the Cell object that contains the information about the cells of the beam.

Notice however that there is a comma after beamPart.cells. This is because we are trying to create a variable which is a **Region** object. A **Region** object is a type of object on which you can apply an attribute. You can use a **Region** object to define the geometry for a section assignment, or a load, or a boundary condition, or a mesh, basically it forms a link between the geometry and the applied attribute. A **Region** object can be a sequence of **Cell** objects. In fact it can be a sequence of quite a few other objects, including **Node** objects, **Vertex** objects, **Edge** objects and **Face** objects. In our script we are assigning a **Cell** object to it. But since it needs to be a sequence of **Cell** objects, not just one **Cell** object that we are providing, we stick the comma at the end to make it a sequence. We then assign it to the variable **beam_region**.

Why exactly are we creating a **Region** object? Because we need it for the next statement where we use the **SectionAssignment()** method.

```
beamPart.SectionAssignment(region=beam_region, sectionName='Beam Section')
```

This statement creates a **SectionAssignment** object, which is an object that is used to assign sections to a part, an assembly or an instance. This is done using the **SectionAssignment()** method. Its first parameter is a **region,** in this case the region is the entire part. We have already created a region in the previous statement called **beam_region** using all the cells of the part, and we now this region as our first parameter. The second argument is the **name** we wish to give the section, which is also the key it will be assigned in the **sections** repository. This argument must be a String, therefore we cannot use the variable **beamSection** which is a **Section** object, but rather its name/key. The statement can be written without the keywords **region** and **sectionName** as:

```
beamPart.SectionAssignment(beam_region, 'Beam Section')
```

4.3.6 Create an assembly

The following block creates the assembly.

```
# -------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
beamAssembly = beamModel.rootAssembly
beamInstance = beamAssembly.Instance(name='Beam Instance', part=beamPart,
                                                            dependent=ON)
```

```
import assembly
```

This statement imports the **assembly** module giving the script access to its methods and properties.

```
beamAssembly = beamModel.rootAssembly
```

This statement assigns the **rootAssembly** to the variable **beamAssembly**. The **rootAssembly** is an **Assembly** object. It is a member of the **Model** object. However you do not need to create it using any method, it is created by default when the **Model** object is created. Hence we simply refer to it without creating it.

The statement

```
beamInstance = beamAssembly.Instance(name='Beam Instance', part=beamPart,
                                                            dependent=ON)
```

creates an **PartInstance** object which is the instance of a part in the assembly. To do this it uses the **Instance()** method which creates a **PartInstance** object and places it in the **instances** repository. It has 2 mandatory parameters, followed by optional ones. The first mandatory parameter required by the **Instance()** method is a **name** for the instance which will be used as its key in the **instances** repository. The second mandatory paramenter is a **Part** object. Note that this is not a String that is the name/key for the part, therefore we cannot use the name 'Beam' which we gave to our part instance. Instead we use the variable that identifies it, **beamPart**. The third parameter used here is an optional one which decides whether the part instance is dependent or independent. By default it is set to **OFF**. The statement can be written without the keywords **name**, **part** and **dependent** as

```
beamInstance = beamAssembly.Instance(name='Beam Instance', part=beamPart,
                                     dependent=ON)
```

4.3.7 Create steps

The following block creates the steps.

```
# ------------------------------------------------------------------------
# Create the step

import step

# Create a static general step
beamModel.StaticStep(name='Apply Load', previous='Initial',
                     description='Load is applied during this step')
```

```
import step
```

The statement imports the **step** module so the script can access its methods and properties.

```
beamModel.StaticStep(name='Apply Load', previous='Initial',
                     description='Load is applied during this step')
```

The statement creates an analysis step by creating a **StaticStep** object using the **StaticStep()** method. The **StaticStep** object is derived from the **AnalysisStep** object which in turn is derived from the **Step()** object. The **StaticStep** object is used specifically for a static load step. Other types of **Step** objects are used for other kinds of loading steps which you will encounter in subsequent examples.

The **StaticStep()** method has 2 mandatory arguments, and a few optional ones in addition. The first mandatory parameter is the **name** you wish to give the analysis step, which is also the key in the repository. The second mandatory parameter is the **previous** step. This must be a String, not a **step** object. We want our loading step to occur after the default 'Initial' step, hence we use its key 'Initial' as the parameter. Among the optional arguments is **description** which is exactly what its name suggests. This doesn't have any bearing on the simulation of course, it is for the benefit of the next person who goes through your simulation. The statement can be written without the **name, previous** and **description** keywords as:

```
beamModel.StaticStep('Apply Load', 'Initial', 'Load is applied during this step')
```

4.3.8 Create and define field output requests

The following code block creates the field output requests.

```
# -----------------------------------------------------------------------
# Create the field output request

# Change the name of field output request 'F-Output-1' to 'Selected Field Outputs'
beamModel.fieldOutputRequests.changeKey(fromName='F-Output-1',
                                        toName='Selected Field Outputs')

# Since F-Output-1 is applied at the 'Apply Load' step by default, 'Selected Field
# Outputs' will be too
# We only need to set the required variables
beamModel.fieldOutputRequests['Selected Field Outputs'].setValues(variables=('S',
                                                   'E', 'PEMAG', 'U', 'RF', 'CF'))
```

```
beamModel.fieldOutputRequests.changeKey(fromName='F-Output-1',
                                        toName='Selected Field Outputs')
```

The statement changes the name of the preexisting **FieldOutputRequest** object. The model has a **FieldOutputRequest** object created by default whose name is 'F-Output-1'. This is also its key in the **fieldOutputRequests** repository. We use the **changeKey()** method to change its name from 'F-Output-1' to 'Selected Field Outputs' by providing these two Strings as arguments. The **FieldOutputRequest** object is defined in the step module hence you do not use an *import fieldOutputRequest* statement.

```
beamModel.fieldOutputRequests['Selected Field Outputs'].setValues(variables=('S',
                                                   'E', 'PEMAG', 'U', 'RF', 'CF'))
```

This statement tells Abaqus what properties you wish to include in your field output request. Notice how we use the key to access our **FieldOutputRequest** object using the *fieldOutputRequests['Selected Field Outputs']* notation. We use the **setValues()** method to instruct Abaqus which variables are desired by passing these as a sequence of Strings. If we wish to leave these at the defaults, we can use the symbolic constant **PRESELECT** in the form *variables=PRESELECT* and rewrite the statement as

```
beamModel.fieldOutputRequests['Selected Field Outputs'].
                                                  setValues(variables=PRESELECT)
```

4.3.9 Create and define history output requests

The following block defines the history output requests:

```
# ----------------------------------------------------------------------
# Create the history output request

# We try a slightly different method from that used in field output request
# Create a new history output request called 'Default History Outputs' and assign
# both the step and the variables
beamModel.HistoryOutputRequest(name='Default History Outputs',
                               createStepName='Apply Load', variables=PRESELECT)

# Now delete the original history output request 'H-Output-1'
del beamModel.historyOutputRequests['H-Output-1']
```

The history output request block could have been written in a manner similar to the field output request i.e., rename H-Output-1 using the **changeKey()** method. This is how you might have expected it to look:

```
#------------------------------------------------------------
#Create the history output request

# change the name of history output request 'H-Output-1' to 'Selected History
# Outputs'
beamModel.historyOutputRequests.changeKey(fromName='H-Output-1',
                                          toName='Selected  History Outputs')
# since H-Output-1 is applied at the ‘Apply Load’ step by default, ‘Selected
# History Outputs’ will be too
# we leave the variables at defaults
beamModel.historyOutputRequests['Selected History Outputs'].
                                                 setValues(variables=PRESELECT)
```

This is absolutely correct. However we've used a different method just to demonstrate another way to do this. Instead of renaming the history output, we create a new one and delete the default one.

```
beamModel.HistoryOutputRequest(name='Default History Outputs',
                               createStepName='Apply Load', variables=PRESELECT)
```

This statement creates a new **HistoryOutputRequest** object using the **HistoryOutputRequest** method. The first argument is the **name** of the object, which is its key in the **historyOutputRequests** repository. The second argument, **createStepName**, is the name of the step in which it is created, which is the String 'Apply Load'. The third argument specifies the **variables** you desire, which in this case has been set to defaults using the **PRESELECT** symbolic constant.

```
del beamModel.historyOutputRequests['H-Output-1']
```

This statement then deletes the preexisting **HistoryOutputRequest** object named 'H-Output-1' which is created by default.

Which of the two methods is better? It's a personal choice, feel free to use the one you prefer.

4.3.10 Apply loads

The following block applies the loads:

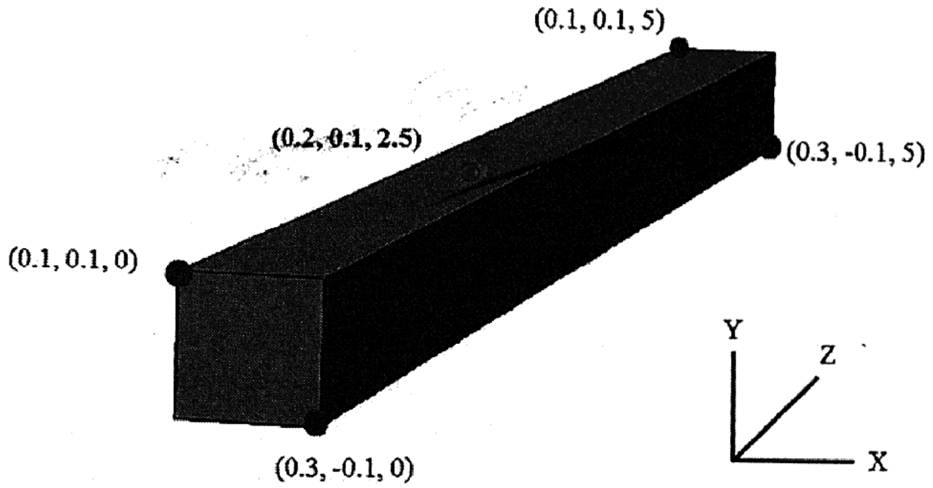
```
# -----------------------------------------------------------------------
# Apply pressure load to top surface

# First we need to locate and select the top surface
# We place a point somewhere on the top surface based on our knowledge of the
# geometry
top_face_pt_x = 0.2
top_face_pt_y = 0.1
top_face_pt_z = 2.5
top_face_pt = (top_face_pt_x,top_face_pt_y,top_face_pt_z)

# The face on which that point lies is the face we are looking for
top_face = beamInstance.faces.findAt((top_face_pt,))

# We extract the region of the face choosing which direction its normal points in
top_face_region=regionToolset.Region(side1Faces=top_face)

# Apply the pressure load on this region in the 'Apply Load' step
beamModel.Pressure(name='Uniform Applied Pressure', createStepName='Apply Load',
                   region=top_face_region, distributionType=UNIFORM,
                   magnitude=10, amplitude=UNSET)
```

```
top_face_pt_x = 0.2
```

```
top_face_pt_y = 0.1
top_face_pt_z = 2.5
```

These statements assign the X, Y and Z coordinates of a point on the top surface of the beam to variables. In fact they are the exact center of the top surface of the beam. Why do we need these coordinates? Well in order to select a surface in a script, you tell Abaqus the coordinates of a point that is on that surface, and using that information Abaqus figures out which surface you are talking about.

How would you know where the center of the top surface of the beam is? By trying to visualize the geometry you've created. In the part creation step we explicitly set the coordinates of the rectangle while sketching out the profile. This profile was then extruded. So you basically know exactly where the beam is in space. The top left corner of the rectangle in the sketch was at X = 0.1, Y = 0.1. Since the sketch was made on the XY plane, Z = 0. The bottom right corner of the rectangle in the sketch was X = 0.3, Y = -0.1, and again Z = 0. The sketch was then extruded 5 m, therefore the extrusion was from Z = 0 to Z = 5. Try to visualize the beam in your head or scribble it on a piece of paper. Your rough sketch might look something like the following figure.



Once you visualize it in this manner, it is pretty easy to figure out the coordinates of the center of the top surface are (0.2, 0.1, 2.5).

```
top_face_pt = (top_face_pt_x,top_face_pt_y,top_face_pt_z)
```

This statement assigns the X, Y and Z coordinates of the point to a variable called top_face_pt. From the syntax you should realize that this is a tuple which you learnt about in Chapter 3. To refresh your memory, they're like lists, except that you cannot change the elements once you create them. Also you define them using semi-circular parentheses as opposed to square brackets.

```
top_face = beamInstance.faces.findAt((top_face_pt,))
```

This statement uses the **findAt()** method to find any face that is at that point or at a distance of less than 1E-6 from it. For the record, the point should not be shared by more than one face because then the method will return whichever face it first encounters and there is no way of knowing which one it will be. As an argument we pass the coordinates of the point, and the method returns a **Face** object. Notice that we have multiple parenthesis and we put a comma after **top_face_pt**. You might have expected to write **findAt(top_face_pt)**. However **findAt()** can accept a sequence of points and normals as arguments and return a sequence of face objects, in which case you are required to use the syntax *findAt(((x1,y1,z1),),((x2,y2,z2),), ((x3,y3,z3),))*. In our case we wish to supply just one point, but we need to follow syntax requirements. Hence we can write **findAt(((top_face_pt_x, top_face_pt_y, top_face_pt_z),))**. Since we have already defined **top_face_pt = (top_face_pt_x, top_face_pt_y, top_face_pt_z)** we instead write **findAt((top_face_pt,))**. Notice this has only 2 parentheses as opposed to 3.

```
top_face_region=regionToolset.Region(side1Faces=top_face)
```

This statement creates a region using the **Face** object we have just created. It uses the **Region()** method to create a **Region** object. To complicate things the **Region()** method works a little differently depending on the kind of arguments you give it. It can accept a number of arguments, such as faces, edges, vertices and so on. If you do not use the *name* argument it will use these to create a set-like region (this is done in the next section when applying constraints). On the other hand if you use the *name* argument it will create a surface-like region. We want a surface-like region so that we can apply the pressure load on it. The *name* argument also specifies the direction we want the normal to the region to point. When the face we pass as a parameter is set equal to **side1Faces** which is the *name* argument, the normal of the region is the same as the normal of the face. If instead we had written **side2Faces=top_face**, the normal to our region would have pointed into the beam rather than out of it. The **Region** object is defined in the **regionToolset** module, which is why we used the *import regionToolset* statement in the initialization section of our script.

```
beamModel.Pressure(name='Uniform Applied Pressure', createStepName='Apply Load',
                   region=top_face_region, distributionType=UNIFORM,
                   magnitude=10, amplitude=UNSET)
```

This statement applies the pressure load to the top surface. It uses the **Pressure()** method to create a **Pressure** object. The **Pressure()** method requires you to specify the **name** or repository key, for which we pass the String 'Uniform Applied Pressure'. It requires the name or key of the step, **createStepName**. We pass the String 'Apply Load' which is the name we gave our step in the 'create step' code block. We must tell the method which **region** to apply the pressure to by giving it a **Region** object. This is why we created the **Region** object **top_face_region** in the previous statement. The pressure magnitude is also required, which is pretty self-explanatory. The **distributionType** and **amplitude** parameters are optional. We set the distribution type to uniform using the symbolic constant **UNIFORM** to tell Abaqus our pressure is uniformly distributed. **UNIFORM** is the default value so we did not need to specifically set it. For the **amplitude**, you would provide an **amplitude** object if you had created one. In our case we do not have any varying amplitude, and have not created an **amplitude** object, hence we use the symbolic constant **UNSET** to indicate this. If we did not include the amplitude parameter, this would have been set to UNSET by default. Hence our statement could have been written without **distributionType** and **amplitude** as:

```
beamModel.Pressure(name='Uniform Applied Pressure', createStepName='Apply Load',
                   region=top_face_region, magnitude=10)
```

4.3.11 Apply constraints/boundary conditions

The following block applies the constraints:

```
# -------------------------------------------------------------------------
# Apply encastre (fixed) boundary condition to one end to make it cantilever

# First we need to locate and select the top surface
# We place a point somewhere on the top surface based on our knowledge of the
# geometry
fixed_end_face_pt_x = 0.2
fixed_end_face_pt_y = 0
fixed_end_face_pt_z = 0
fixed_end_face_pt = (fixed_end_face_pt_x,fixed_end_face_pt_y,fixed_end_face_pt_z)

# The face on which that point lies is the face we are looking for
```

```
fixed_end_face = beamInstance.faces.findAt((fixed_end_face_pt,))

# We extract the region of the face choosing which direction its normal points in
fixed_end_face_region=regionToolset.Region(faces=fixed_end_face)

beamModel.EncastreBC(name='Encaster one end', createStepName='Initial',
                                              region=fixed_end_face_region)
```

```
fixed_end_face_pt_x = 0.2
fixed_end_face_pt_y = 0
fixed_end_face_pt_z = 0
```

These statements assign the X, Y and Z coordinates of a point on the surface of the fixed end of the beam to variables. This point is the exact center of the clamped surface of the beam. We are going to use this point to find the surface in a manner similar to the one used in the 'apply loads' step. Once we tell Abaqus the coordinates of the point, it will be able to figure out which surface we are talking about.

How did we get these coordinates? Once again visualize the geometry you've created. In the part creation step we explicitly set the coordinates of the rectangle while sketching out the profile. This profile was then extruded. So you know the beams location in space. The top left corner of the rectangle in the sketch was at X = 0.1, Y = 0.1. Since the sketch was made on the XY plane, Z = 0. The bottom right corner of the rectangle in the sketch was X = 0.3, Y = -0.1, and again Z = 0. Since our point is at the center of this face, X=(0.3-0.1)/2 = 0.2, Y=(0.1-(-0.1))/2 = 0, and Z of course is 0.

```
fixed_end_face_pt = (fixed_end_face_pt_x,fixed_end_face_pt_y,fixed_end_face_pt_z)
```

This statement assigns the X, Y and Z coordinates of the point to a variable called **fixed_end_face_pt**, using the same method used in the 'apply load' code block. Once again from the syntax you realize that this is a tuple.

```
fixed_end_face = beamInstance.faces.findAt((fixed_end_face_pt,))
```

This statement uses the **findAt()** method to find any face that is at that point or at a distance of less than 1E-6 from it. As was described in the previous section, the method **findAt()** returns the **Face** object on which the point is located.

```
fixed_end_face_region=regionToolset.Region(faces=fixed_end_face)
```

This statement selects a region using the **Face** object we have just created. It uses the **Region()** method to create a **Region** object. This is the same method used to create a region in the previous section ('apply load' section) so you already know how it works.

The one difference in the syntax from the 'apply load' code block is that the name of the parameter **side1faces** has been replaced with **faces**. Remember the **Region()** method returns a set-like region or a surface-like region depending on the parameters as explained in the 'apply load' section. In this case we apply the encastre constraint on the returned **Region** object. Since encastre is performed on the set of points and not just the surface, we need **Region()** to return a set-like region. Hence we use faces instead of side1Faces. In fact if you were to replace this statement with

```
fixed_end_face_region=regionToolset.Region(side1Faces=fixed_end_face)
```

you would get the following error: "TypeError: region, found Region, expecting Set".

```
beamModel.EncastreBC(name='Encaster one end', createStepName='Initial',
                                               region=fixed_end_face_region)
```

This statement applies the encastre constraint to the end of the beam. It uses the **EncastreBC()** method to create a **TypeBC** object, which is an object that stores data on several types of predefined boundary conditions commonly used in stress/displacement analysis. This **TypeBC** object is derived from the **BoundaryCondition** object. The **EncastreBC()** method requires you to specify the name or repository key for the boundary condition, for which we pass the String 'Encastre one end'. It requires the name or key of the step. We pass the String 'Initial' which is the name that was given by default to the first step. We must tell the method which region to apply the pressure to by giving it a set-like Region object. This is why we created the **Region** object **fixed_end_face_region** in the previous statement.

4.3.12 Mesh

The following block creates the mesh:

```
# -----------------------------------------------------------------------
# Create the mesh

import mesh

# First we need to locate and select a point inside the solid
# We place a point somewhere inside it based on our knowledge of the geometry
```

```
beam_inside_xcoord=0.2
beam_inside_ycoord=0
beam_inside_zcoord=2.5

elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN, secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)


beamCells=beamPart.cells
selectedBeamCells=beamCells.findAt((beam_inside_xcoord,beam_inside_ycoord,
                                                       beam_inside_zcoord),)
beamMeshRegion=(selectedBeamCells,)
beamPart.setElementType(regions=beamMeshRegion, elemTypes=(elemType1,))

beamPart.seedPart(size=0.1, deviationFactor=0.1)

beamPart.generateMesh()
```

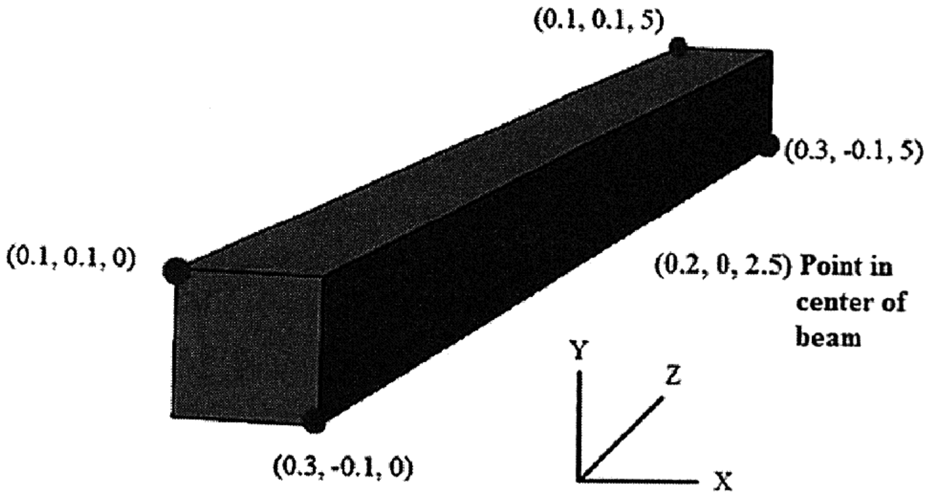
```
import mesh
```

This statement makes the methods and attributes of the mesh module available to our script.

```
beam_inside_xcoord=0.2
beam_inside_ycoord=0
beam_inside_zcoord=2.5
```

These statements assign the X, Y and Z coordinates of a point inside the beam to variables. In fact they are the exact center of the beam. Why do we need these coordinates? Well, in order to mesh the beam we need to assign the cells in the interior of the beam to the mesh. To identify these we need to first find a point inside the beam.

The center of the beam is once again found using basic geometry. You can visualize it mentally or sketch it. This is displayed in the following figure. Once you've sketched it out its pretty easy to see that the coordinates of the center of the beam are (0.2, 0, 2.5).



The next statement

```
elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN, secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)
```

creates an **ElementType** object using the **ElemType()** method. The **ElementType** object can later be used as an argument to the **setElementType()** method to set the element type of the part mesh, and this is done a few statements later. The object and the method are defined in the mesh module which is why we use the *import mesh* statement. The only required argument for the **ElemType()** method is **elemCode** which is a SymbolicConstant in Abaqus that specifies the element code. These are the same codes you see in Abaqus/CAE in the **Element Type** window when you go to **Mesh > Element Type...** The other parameters are optional. The **elemLibrary** parameter specifies which element library to use. The options are **STANDARD** and **EXPLICIT**, both of which are symbolic constants. The default value is **STANDARD**, and since this is what we wanted anyway we could have just left the argument out. The **kinematicSplit** parameter refers to kinematic split control. It accepts a few SymbolicConstants, which are defined in the documentation. The one used here is **AVERAGE_STRAIN** which is the default hence it could have been left out altogether. The **secondOrderAccuracy** parameter specifies, as its name suggests, whether second order accuracy is used. It accepts the SymbolicConstants **ON** and **OFF**, of which the latter is the default value (hence we could have left it out and it would have defaulted to **OFF**). The **hourglassControl** accepts a few SymbolicConstants as values. The reduced integration numerical procedure sometimes faces a problem where there is no straining at the integration points. This results in a phenomenon called "hourglassing". Hourglass control associates a small stiffness with these zero-energy modes, to counter the issue. We set it to **DEFAULT**. Since the default value is **ENHANCED**, we could have set it to this instead for the same result. Or we could have just left it out and Abaqus would have assigned the default to it anyway. The last optional argument used is **distortionControl**. This can be used to prevent excessive deformation where the element volumes become negative. We have set it to **DEFAULT**. Since the default is **OFF** we could have set it to **OFF** instead. Or we could have left it out altogether and it would default to **OFF** on its own. Hence the statement could have been written as:

```
elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN, secondOrderAccuracy=OFF,
                          hourglassControl=ENHANCED)
```

In fact since we accepted all the defaults for the optional arguments we could have written:

```
elemType1 = mesh.ElemType(elemCode=C3D8R)
```

The statement

```
beamCells=beamPart.cells
```

assigns the cells of the part to a variable called **beamCells**. You saw the **cells** object used in the 'create section' code block, where **beamPart.cells** were required to create the region of the section assignment. If you recall, the **cell** object defines the volumetric regions of a geometry. **Part** objects have cells. **beamPart.cells** refers to the **Cell** object that contains the information about the cells of the beam.

Notice however that this time there is no comma after **beamPart.cells**. This is because we only want a **Cell** object to use in the next statement, not a sequence of Cell objects (which would give us a **Region** object).

```
selectedBeamCells=beamCells.findAt((beam_inside_xcoord,beam_inside_ycoord,
                                    beam_inside_zcoord),)
```

This statement uses the **findAt()** method to select the internal cells of the beam. You've seen the **findAt()** method in the 'apply load' and 'apply constraint' sections, the syntax is the same. The resulting variable **selectedBeamCells** is a Region object.

```
beamMeshRegion=(selectedBeamCells,)
```

This statement converts the **Region** object into a sequence of regions because a comma has been placed at the end of **selectedBeamCells**. You can then use this sequence of regions in the **setElementType** command.

```
beamPart.setElementType(regions=beamMeshRegion, elemTypes=(elemType1,))
```

This statement uses the **setElementType()** method to set the element type of the mesh. It requires you to provide the regions to mesh as a sequence of **Region** objects as one of the parameters. We have this stored in the **beamMeshRegion** variable. The second required parameter is a sequence of **ElementType** objects which tell Abaqus what element types to use on the regions we wish to mesh.

Note that we could have left out the statement

```
beamMeshRegion =(selectedBeamCells,)
```

entirely, and then rewritten the last statement as

```
beamPart.setElementType(regions=(selectedBeamCells,), elemTypes=(elemType1,))
```

So which way is better? Either works fine, just remember there's more than one way to do things and often readability of the code can be an important factor in deciding which one to go with.

```
beamPart.seedPart(size=0.1, deviationFactor=0.1)
```

This statement assigns the global seeds to the part. The **size** is a required argument, which defines the global size for the edges. The **deviationFactor** is an optional argument which is the ratio of the chordal deviation to the element length. The **seedPart()** method is defined in the **mesh** module which we imported.

```
beamPart.generateMesh()
```

This statement uses the **generateMesh()** method to generate the mesh on our part. The **generateMesh()** method is defined in the mesh module which we have imported.

4.3.13 Create and run the job

The following block runs the job

```
# -----------------------------------------------------------------------
# Create and run the job

import job

# Create the job
mdb.Job(name='CantileverBeamJob', model='Cantilever Beam', type=ANALYSIS,
        explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE,
        description='Job simulates a loaded cantilever beam',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)

# Run the job
mdb.jobs['CantileverBeamJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['CantileverBeamJob'].waitForCompletion()

# End of run job
```

```
import job
```

This statement imports the job module allowing the script to access its methods.

```
mdb.Job(name='CantileverBeamJob', model='Cantilever Beam', type=ANALYSIS,
        explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE,
        description='Job simulates a loaded cantilever beam',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)
```

This statement creates the job. The **Job()** method creates a **Job** object for the model. It is a method of the **ModelJob** object which is derived from the **Job** object. It has two required arguments. The first is **name** which is the repository key. We have called it ''CantileverBeamJob'. Note that there are no spaces in the name, this is required. The second argument is the name/key of the model, which in our case is the String 'Cantilever Beam'. There are a number of optional arguments. One of them is **description**, in which you can provide a String description of the job. The other optional arguments are defined in the documentation. Some of these such as **memory** may be worth altering if your computer has trouble handling the simulations. The **Job()** method is defined in the **job** module which is why we imported it with the statement *import job*.

```
mdb.jobs['CantileverBeamJob'].submit(consistencyChecking=OFF)
```

This statement submits the job for analysis. The **submit()** method is a method of the **Job** object. It has no required arguments although it does have a few optional ones. Here we have used the **consistencyChecking** argument which performs a consistency check for the simulation setup. The **submit()** method is defined in the **job** module which has been imported.

```
mdb.jobs['CantileverBeamJob'].waitForCompletion()
```

This statement makes sure the control lies with the script till the analysis finishes executing. The **waitForCompletion()** method is a method of the **Job** object. **waitForCompletion()** is defined in the **job** module which has been imported. Why is that important? It is particularly useful when running scripts from the command line because the blinking cursor prompt gets disabled (busy) until the script finishes running. You might be running your scripts through an optimization tool such as ISight or ModelCenter, and often the only way for an external software to know that one analysis job is completed is to have Abaqus return control only after it has finished running.

4.3.14 Post processing

The following code performs some post processing tasks:

```
# -----------------------------------------------------------------------
# Post processing

import visualization

beam_viewport = session.Viewport(name='Beam Results Viewport')
beam_Odb_Path = 'CantileverBeamJob.odb'
an_odb_object = session.openOdb(name=beam_Odb_Path)
beam_viewport.setValues(displayedObject=an_odb_object)
beam_viewport.odbDisplay.display.setValues(plotState=(DEFORMED, ))
```

```
import visualization
```

This statement imports the **visualization** module. This allows the script to access methods that replicate the functionality of the visualization module of Abaqus/CAE.

```
beam_viewport = session.Viewport(name='Beam Results Viewport')
```

This statement uses the **Viewport()** method to create a **Viewport** object. The only required argument is **name** which is a String specifying the repository key. In this case we name it 'Beam Results Viewport'.

```
beam_Odb_Path = 'CantileverBeamJob.odb'
```

This statement assigns the name of the output database file to a variable for later use.

```
an_odb_object = session.openOdb(name=beam_Odb_Path)
```

This statement creates an **Odb** object by opening the output database whose path is provided as an argument, and assigns it to the variable **an_odb_object**. Note that we have not provided a complete path, only the file name, hence it will search for the file in the default Abaqus working directory. You may provide an absolute path if you are working with an output database file saved elsewhere on the hard drive.

```
beam_viewport.setValues(displayedObject=an_odb_object)
```

The statement uses the **setValues()** method to set the display to the selected output database. If you recall, this same method was used in the 'initialization block' (Section 4.3.1) of the script with **displayedObject=none** to blank the viewport. Just so you know, the above statement could have been written instead as

```
session.viewports['Beam Results Viewport']
                                    .setValues(displayedObject=an_odb_object)
```

The statement

```
beam_viewport.odbDisplay.display.setValues(plotState=(DEFORMED, ))
```

This statement changes the viewport display to the deformed beam by using the **setValues()** method and setting the plot state to the symbolic constant **DEFORMED**. For the record, the above statement could also have been written as

```
session.viewports['Beam Results Viewport'].odbDisplay
                                    .display.setValues(plotState=(DEFORMED, ))
```

4.4 What's Next?

In this chapter you worked through all the steps in the creation and setup of a finite element simulation in Abaqus using a Python script. Not only did you see the bigger picture, but you also examined individual statements and learnt of a number of new objects and methods that you will regularly encounter when scripting. In subsequent chapters we are going to look at many more examples, each of which we will perform tasks that weren't demonstrated in this one. But first, let's learn a little more Python syntax.

5

Python 102

5.1 Introduction

In Python 101, we covered many aspects of Python syntax. We spent a great deal of time understanding important concepts such as lists and tuples, and object oriented programming. That knowledge helped you understand the cantilever beam script. The example did not however use any conditional statements or any iterative loops. If...else... statements and for-loops are usually a major element in any sort of program you write, and you will need to use them in more complicated Python scripts as well. We'll cover them in this chapter.

This book assumes that you are familiar with at least one programming language, whether it be a full-fledged language like C++ or Java, or an engineering tool such as MATLAB. Hence the concepts of conditional statements and loops should not be new to you. This chapter aims only to familiarize you with the syntax of these constructs in Python.

5.1.1 If... elif ... else statements

The if-statement in Python is very similar to that used in other programming languages. It tests if a certain condition is true. If it is then it executes a statement or block of code.

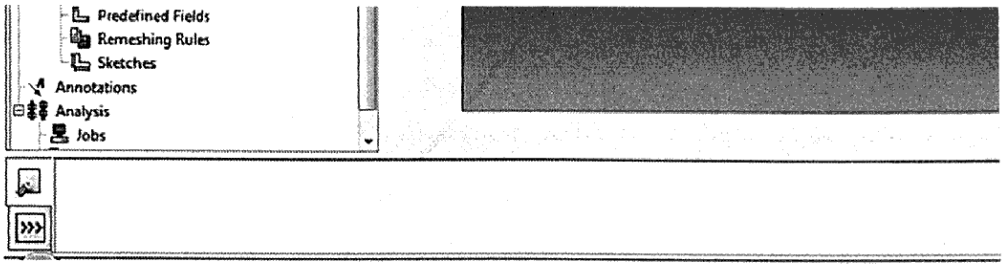
If it is not true, Python checks to see if an else-if or else block is present. Else-if is written as **elif** in Python. **Elif** tests another condition whereas **else** does not test for any condition.

The syntax is a little different in Python. You do not put the **if** and **else** blocks of code within curly braces as you do in many other languages. In Python you indent the block instead. Also the colon ':' symbol is used. To indent the block is analogous to using braces in other languages, if you don't do it you will get an error. The syntax looks something like this.

```
if a_certain_condition_is_true :
    do this
    and this
elif another_condition_is_true:
    do this
    and this
else:
    do this
```

Example

Open up Abaqus CAE. In the lower half of the window you see the message area. If you look to the left of the message area you see two tabs, one for **Message area** and the other for **Kernal Command Line Interface**.

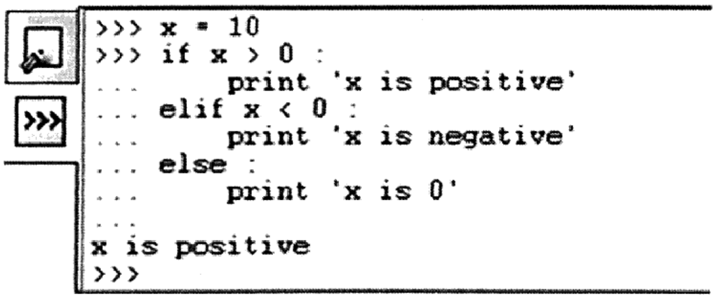


Click the second one. You see the kernel command prompt which is a ">>>" symbol.

Type the following lines, hitting the ENTER key on your keyboard after each.

```
X = 10
if x > 0 :
        print 'x is positive'
elif x < 0:
        print 'x is negative'
else :
        print 'x is 0'
```

Here is what you see


```
>>> x = 10
>>> if x > 0 :
...     print 'x is positive'
... elif x < 0 :
...     print 'x is negative'
... else :
...     print 'x is 0'
...
x is positive
>>>
```

5.1.2 For loops

The **for** loop in Python is conceptually similar to that in other languages – it provides the ability to loop or iterate over a certain set of data. However its implementation is a little different in Python.

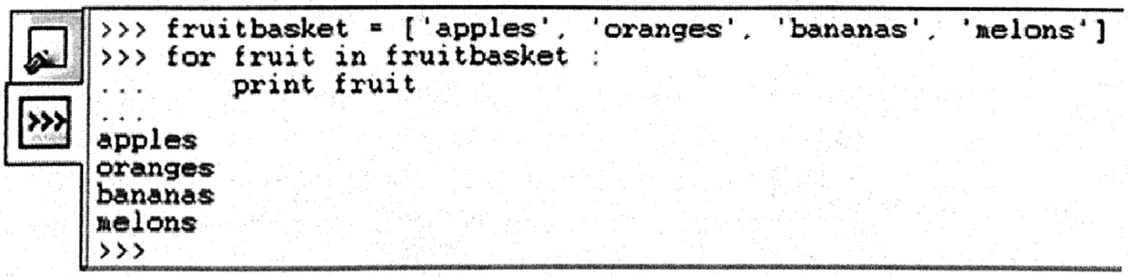
In C, C++, Java or MATLAB, you find yourself iterating either a fixed number of times by incrementing a variable every loop till it reaches a certain value, or until a condition is satisfied. In Python on the other hand, you create a sequence (a list or a string), and the for loop iterates over the items in that list (or characters in a string).

Example

Type the following statements in the Abaqus kernel command interface prompt

```
fruitbasket = ['apples', 'oranges', 'bananas', 'melons']
for fruit in fruitbasket :
        print fruit
```

Here is what you see:


```
>>> fruitbasket = ['apples', 'oranges', 'bananas', 'melons']
>>> for fruit in fruitbasket :
...     print fruit
...
apples
oranges
bananas
melons
>>>
```

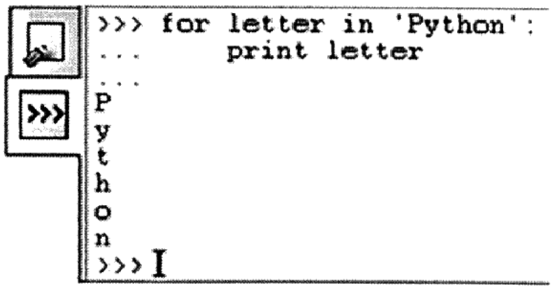
In the above example, fruitbasket is a list consisting of a sequence of strings. With each iteration, the **for** loop takes an element (in this case a string) out of the list and assigns it to the variable fruit. The print statement then prints it out on screen. Basically our **for** loop iterates 4 times.

Example

Type the following in the Abaqus kernel command interface prompt

```
for letter  in 'Python' :
       print letter
```

Here is what you see:



```
>>> for letter in 'Python':
...      print letter
...
P
y
t
h
o
n
>>> 
```

In the above example, 'Python' is a string, essentially a sequence of characters. With each iteration, the **for** loop takes an element (in this case a character) out of the String and assigns it to the variable letter. The print statement then prints it out on screen. So this for loop iterates 6 times.

This type of **for** loop is great for iterating through the elements of a list and performing an action on each one. Abaqus stores its repository keys in lists, hence it is easy to iterate through them using a **for** loop. This will be demonstrated in Chapter 8 while performing a dynamic, explicit truss analysis.

5.1.3 range() function

Sometimes you may wish to use a **for** loop to iterate a certain number of times, rather than loop through each element of a preexisting list. However the **for** loop can only operate on a sequence. A workaround is to generate a list for the task using the **range()** function.

The **range()** function generates a list which consists of arithmetic progressions. It can take one, two or three arguments. If one argument is provided, a list is generated starting at 0, and ending at one integer less than the argument provided. It will naturally have the same number of elements as the value of the integer argument.

`range(5)` returns [0, 1, 2, 3, 4]

If two arguments are provided, the first one is treated as the beginning of the list, and the end of the list is one less than the second argument.

`range(5,9)` returns [5, 6, 7, 8]

If three arguments are provided, the first one is treated as the beginning of the list, and the end of the list is one less than the second one. However all elements in the list must be multiples of the third argument.

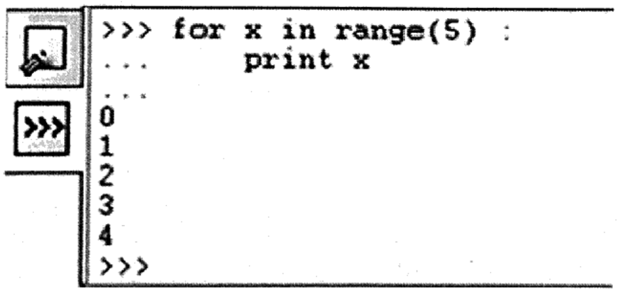
`range(2, 10, 3)` returns [3, 6, 9]

Using the **range()** function, you can specify a for loop to iterate a certain number of times.

Example

```
for x in range(5) :
        print x
```

Here is what you see:



```
>>> for x in range(5) :
...         print x
...
0
1
2
3
4
>>>
```

The above for loop iterates 5 times. The range(5) statement returns a list [0, 1, 2, 3, 4] and the for loop iterates for each element (integer) in this list, assigning it to the variable x. The print statement prints this variable to the screen.

5.1.4 While-loops

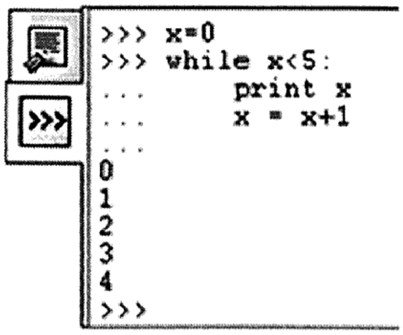
The **while** loop executes as long as a certain condition or expression returns true. It is similar to the **while** loop in other languages. The syntax is

```
while condition:
        do this
        and this
```

Example

```
x = 0
while x<5:
        print x
        x = x+1
```

Here is what you see



```
>>> x=0
>>> while x<5:
...         print x
...         x = x+1
...
0
1
2
3
4
>>>
```

When the **while** loop is first encountered, x = 0, and the x < 5 condition is satisfied and the loop is executed. In each iteration the value of x is incremented by 1. When x = 5, the x<5 condition is no longer satisfied and control breaks out of the loop.

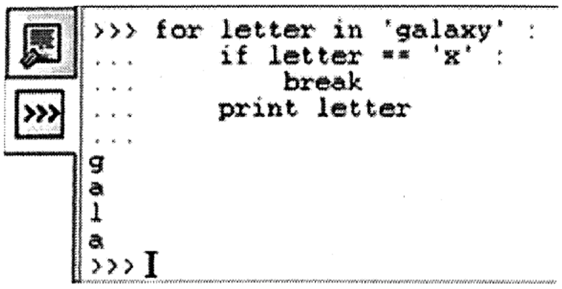
5.1.5 break and continue statements

The **break** statement allows program control to break out of a **for** loop or a **while** loop.

Example

```
for letter in 'galaxy' :
        if letter == 'x' :
                break
        print letter
```

Here is what you see:



```
>>> for letter in 'galaxy' :
...       if letter == 'x' :
...           break
...       print letter
...
g
a
l
a
>>>
```

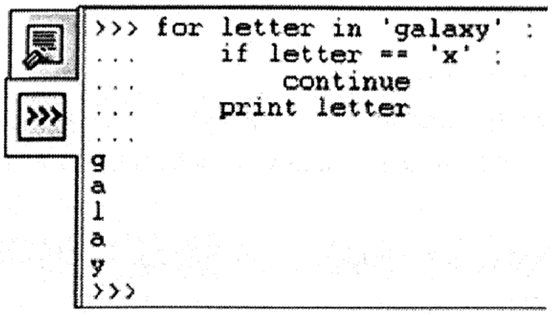
Each of the letters in the word galaxy are printed out turn by turn until the letter ‘x’ is reached. Since the **if** condition returns true, the **break** statement is encountered, and the program breaks out of the loop.

The **continue** statement on the other hand ends the current iteration without executing the remaining statements and begins the next iteration

Example

```
for letter in ‘galaxy’ :
      if letter == ‘x’ :
            continue
      print letter
```

Here is what you see:



```
>>> for letter in 'galaxy' :
...       if letter == 'x' :
...           continue
...       print letter
...
g
a
l
a
y
>>>
```

Once again, each of the letters in the word galaxy are printed out turn by turn until the letter x is reached. Since the if-condition returns true, the **continue** statement is encountered. The current iteration is terminated before the print statement is executed, and the next iteration begins.

5.2 What's Next?

You now possess enough basic knowledge of Python syntax to proceed with scripting for Abaqus. The Python documentation, as well as a number of tutorials, are available at www.python.org if you wish to study the language further.

Before we start working with more examples, let's introduce you to some other important topics such as macros and replay files. Please proceed to the next chapter.

6

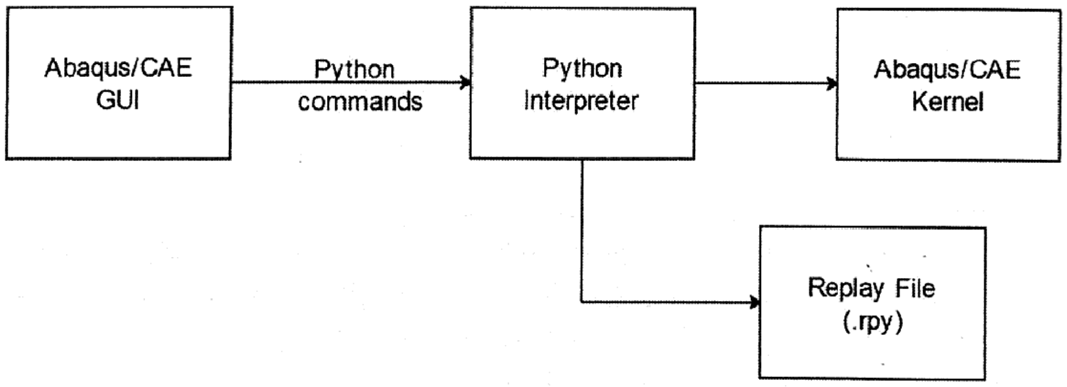
Replay files, Macros and IDEs

6.1 Introduction

The Abaqus Scripting Interface consists of thousands of commands and attributes separated into various Abaqus modules. It would be impossible for you to memorize all of these. Fortunately there is an easier way – replay files. In this chapter we'll talk about how you can use these. We'll also look at Macros, a feature provided by Abaqus, that makes it easy to create simple scripts without requiring any actual coding. And we'll get you hooked up with a good text editor to type your scripts through the rest of the book.

6.2 Replay Files

In Chapter 2, Section 2.2 (page 33), we talked about how Python fits into the bigger scheme of things. To summarize, when the user performs actions in the GUI (Abaqus/CAE), Python commands are generated which pass through the interpreter and are sent to the kernel. Fortunately for us, Abaqus keeps a record of these commands in the form of a replay file with the extension '.rpy'.



The replay file is written in the current work directory. The work directory is C:\Temp by default, and you can change it using **File > Set Work Directory..**

The easiest way to look up the necessary commands is to perform an action in Abaqus/CAE and then open up this replay file. If it is currently in use Abaqus may not let you open it; in this case right click on it and choose copy to create a copy of it in Windows Explorer that you can open.

NOTE: Abaqus Student Edition (current version at time of writing is 6.10-2) does not write replay files. This is one of its limitations. You need to be using the commercial or research editions of Abaqus for replay files to be written to the working directory. However if all you have is the student version, you can achieve the same thing with Macros. We will speak about these shortly. However I recommend you read the next section since everything with replay scripts applies to macros as well.

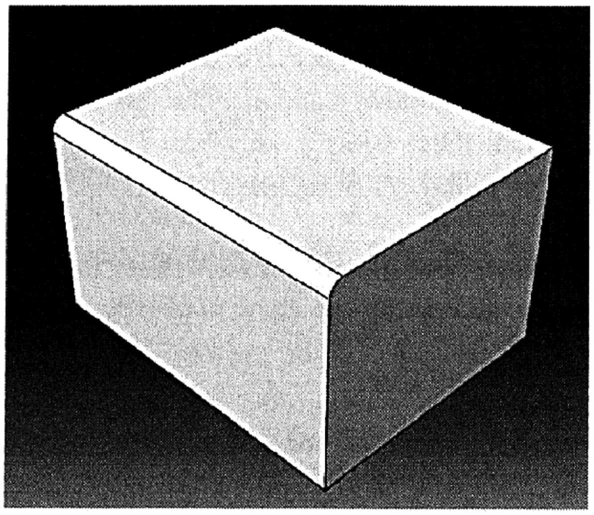
6.3 Example - Compare replay with a well written script

You will find that sometimes the replay file alone is exactly what you need for creating a script with minimal effort. For example if you open up a new model in Abaqus/CAE, do a bunch of stuff, create parts, materials etc, you could copy all the statements from the replay file and save them in a .py file and use this in future to get back to the same point starting from a new model. It would be sort of like saving the .cae, except python scripts take up a lot less space and you can email them to people as text.

However if you are looking to work with the script, modify it, add iterative methods, or parametrize it, the form of the script in the replay file will most likely not be ideal. I'll demonstrate this with an example.

a. Start up Abaqus/CAE. If Abaqus is already open close it and reopen it as you start out with a blank replay file when you start a new Abaqus session.
b. Right click on **Model-1** in the model tree and choose **Rename**. Name it **Block Model**.
c. Double click on **Parts** in the model tree. You see the **Create Part** window.
d. Set the **Name** to **Block**, **modeling space** to **3D**, **type** to **Deformable**,**base feature shape** to **Solid**,**base feature type** to **Extrusion** and **approximate size** to **200**. Click **Continue**. You see the sketcher.
e. Choose the **Create Lines: Rectangle tool.** Click on the origin of the graph and then click anywhere in the top right quadrant to complete the rectangle.

f. Use the **Add Dimension** tool to give it a width of **25** and a height of **15**.
g. Click the red X to close the **Add Dimension** tool and then **Done** to exit the sketcher. You see the **Edit Base Extrusion** dialog box
h. Give the extrusion a **depth** of **20**. Click **OK**. You see the block in the viewport.
i. Choose the **Create Round or Fillet** tool. Click on the top left edge of the block to select it and choose **Done**
j. Give it a radius of **1**.
k. Click the red **X** to exit the **Create Round or Fillet** tool.



Now look in the Abaqus work directory which is C:\Temp by default or whatever you've set it to be. Open it in a text editor such as WordPad which comes with windows. (Notepad will not be good to view the replay file as a lot of the carriage returns are removed).

Here is what you will see (FYI I have modified the information in the top 3 lines):

```
# Abaqus/CAE Release 6.10-1 replay file
# Internal Version: xxxxxxxxxxxxxxxxx
# Run by xxxxxx on Sat MonthDayxx:xx:xx 2011
#

# from driverUtils import executeOnCaeGraphicsStartup
# executeOnCaeGraphicsStartup()
#: Executing "onCaeGraphicsStartup()" in the site directory ...
from abaqus import *
from abaqusConstants import *
session.Viewport(name='Viewport: 1', origin=(0.0, 0.0), width=411.136439800262,
    height=212.019445240498)
```

```
session.viewports['Viewport: 1'].makeCurrent()
session.viewports['Viewport: 1'].maximize()
from caeModules import *
from driverUtils import executeOnCaeStartup
executeOnCaeStartup()
session.viewports['Viewport: 1'].partDisplay.geometryOptions.setValues(
    referenceRepresentation=ON)
mdb.models.changeKey(fromName='Model-1', toName='Block Model')
session.viewports['Viewport: 1'].setValues(displayedObject=None)
s = mdb.models['Block Model'].ConstrainedSketch(name='__profile__',
    sheetSize=200.0)
g, v, d, c = s.geometry, s.vertices, s.dimensions, s.constraints
s.setPrimaryObject(option=STANDALONE)
s.rectangle(point1=(0.0, 0.0), point2=(22.5, 12.5))
s.ObliqueDimension(vertex1=v[3], vertex2=v[0], textPoint=(6.54132556915283,
    -6.48623704910278), value=25.0)
s.ObliqueDimension(vertex1=v[0], vertex2=v[1], textPoint=(-8.33698463439941,
    4.81651592254639), value=15.0)
p = mdb.models['Block Model'].Part(name='Part-1', dimensionality=THREE_D,
    type=DEFORMABLE_BODY)
p = mdb.models['Block Model'].parts['Part-1']
p.BaseSolidExtrude(sketch=s, depth=20.0)
s.unsetPrimaryObject()
p = mdb.models['Block Model'].parts['Part-1']
session.viewports['Viewport: 1'].setValues(displayedObject=p)
del mdb.models['Block Model'].sketches['__profile__']
p = mdb.models['Block Model'].parts['Part-1']
e = p.edges
p.Round(radius=1.0, edgeList=(e[4], ))
```

As you can see, Abaqus has been recording everything you did in CAE in the replay file from the moment the software started up.

You see some statements that you would normally include in all scripts such as

```
from abaqus import *
from abaqusConstants import *
```

But you would be unlikely to write statements such as

```
session.Viewport(name='Viewport: 1', origin=(0.0, 0.0), width=411.136439800262,
    height=212.019445240498)
session.viewports['Viewport: 1'].makeCurrent()
session.viewports['Viewport: 1'].maximize()
from caeModules import *
from driverUtils import executeOnCaeStartup
executeOnCaeStartup()
```

in your script since you probably don't want your script to change the size of the viewport that it is run in, nor are you likely to want to run a startup script.

The remaining statements are the meat of the script. They rename the model, draw the sketch and create the part, and fillet it. However they are written in a very literal sense. For example, the **ObliqueDimensions()** command is used to dimension the edges of the rectangle. When you are using a script you are more likely to enter in the exact coordinates in the **rectangle()** method as **point1** and **point2** as we did in the cantilever beam example.

In addition the statements dealing with the edge round

```
e = p.edges
p.Round(radius=1.0, edgeList=(e[4], ))
```

appear to assign all the edges of the block to a variable 'e', and then Abaqus refers to the desired edge as e[4] which makes sense to it internally as it stores each of the **Edge** objects in a certain order; but this does not make any sense to a human.

Here is what this same script would look like if I wrote it.

```
# ********************************************************************************
# Create a block with a rounded edge

# Created for the book "Python Scripts for Abaqus - Learn by Example"
# Author: Gautam Puri
# ********************************************************************************

from abaqus import *
from abaqusConstants import *

# ------------------------------------------------
# Create the model (or more accurately, rename the existing one)

mdb.models.changeKey(fromName='Model-1', toName='Block Model')
blockModel = mdb.models['Block Model']

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# ---------------------------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the block cross section using the rectangle tool
```

```
blockProfileSketch = blockModel.ConstrainedSketch(name='Block CS Profile',
                                                    sheetSize=200)
blockProfileSketch.rectangle(point1=(0.0,0.0), point2=(25.0,15.0))

# b) Create a 3D deformable part named "Block" by extruding the sketch
blockPart=blockModel.Part(name='Block', dimensionality=THREE_D,
type=DEFORMABLE_BODY)
blockPart.BaseSolidExtrude(sketch=blockProfileSketch, depth=20)

# -------------------------------------------------------------------
# Round the edge

edge_for_round = blockPart.edges.findAt((12.5, 15.0, 20.0), )
blockPart.Round(radius=1.0, edgeList=(edge_for_round, ))
```

The first thing you notice is how much more readable this script is. Secondly (and more importantly), we do not refer to internal edge or vertex lists. The statements for rounding the edge are

```
edge_for_round = blockPart.edges.findAt((12.5, 15.0, 20.0), )
blockPart.Round(radius=1.0, edgeList=(edge_for_round, ))
```

The **findAt()** method refers to coordinates that we can visualize by scribbling the block on a piece of paper. If you decided you wanted to round another edge in a second iteration of the analysis, you could change the coordinates right here and rerun the script. The replay file script on the other hand cannot be modified, since you wouldn't know what to change **e[4]** to since we do not know the sequence of Abaqus's internal edge list.

So you see that the replay file is useful only if you want to exactly replay what was done in Abaqus. However it requires some work to modify it for any other use. As it gets longer it will require too many major changes to be worth the effort.

However having a replay file helps you write your own script. You can see that the major methods used were the same in the replay script and the one I wrote. These include **changeKey()**, **ConstrainedSketch()**, **rectangle()**, **BaseSolidExtrude()** and **Round()**. By performing a task in Abaqus/CAE and looking at the replay file we very quickly know the names of the methods that need to be used and what arguments they require. While it is easy to remember a name like **Round()**, you are unlikely to remember the names of the thousands of other methods available through the Abaqus Scripting Interface. The replay file will tell you at a glance the names of the methods you need, and you can then look these up in the Abaqus Scripting Reference Manual to understand and use them.

Note also that my code is very similar to that used in the Cantilever Beam example. I have infact copied and pasted that code here, and modified it using some help from the replay file. The fastest way to write Python scripts is to reuse code where possible, modify it suitably, and find out what new methods are required by performing the required task in Abaqus/CAE and reading the replay file. The only place you can't really do this is when dealing with output databases, but we'll get to ODB object model interrogation (after a few hundred pages) and teach you what you need to know then.

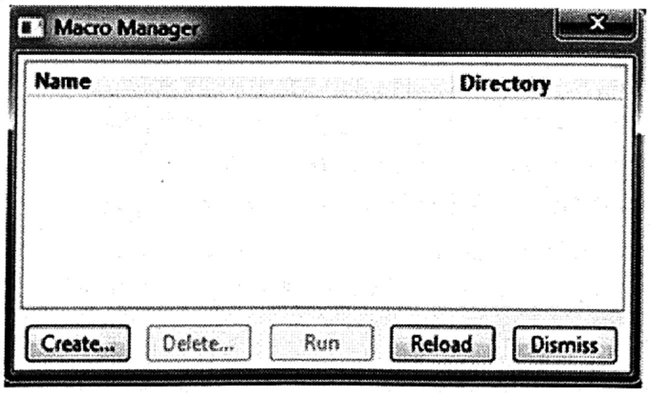
6.4 Macros

Macros are similar to replay files. The difference between them is that the replay file starts at the beginning of your Abaqus session and is continuously updated until you close Abaqus/CAE. In addition it can only be saved by making a copy of the .rpy file in Windows Explorer otherwise it will get overwritten during your next session. Macros on the other hand allow you to define at what point the replay data should start getting logged, and when it should stop. In addition you can give the replay data a name and call it later from within Abaqus. The statements in it will be the same as those in the .rpy file, except you won't have to search through hundreds of lines of other replay statements to find the few you need.

Macros are stored in a file called 'abaqusMacros.py'. Abaqus stores each macro within a function with the name you assign to the macro.

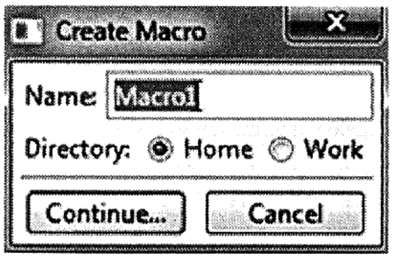
Let's demonstrate this:

Start Abaqus/CAE (or open a new model in Abaqus/CAE). Go to **File > Macro Manager**.

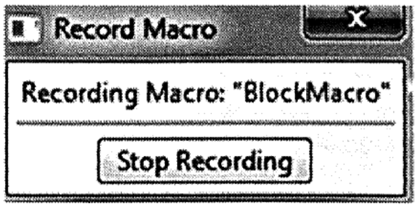


You see the **Macro Manager** dialog box as shown in the figure.

Click on **Create.** You see the **Create Macro** dialog box.



Type in a **name** for the macro such as **BlockMacro**. It needs to be one word as you cannot have a space in a macro name. This is because the name of the macro will be the name of the function in the abaqusMacros.py file and function names cannot have spaces. Change the directory to **Work** so that the macro is saved in the Abaqus work directory. Click **Continue.**



Abaqus begins recording the macro.

Repeat all the steps described in the previous section to rename the model, create the part 'Block' and round the edge. Then click **Stop Recording.**

You see BlockMacro appear in the list in the Macro Manager. As you create more macros they will appear here.

Open 'abaqusMacros.py' in the work directory. Here's what the contents will look like:

```
# Do not delete the following import lines
from abaqus import *
from abaqusConstants import *
import __main__

def BlockMacro():
    import section
    import regionToolset
    import displayGroupMdbToolset as dgm
    import part
    import material
    import assembly
    import step
    import interaction
    import load
    import mesh
    import job
    import sketch
    import visualization
    import xyPlot
    import displayGroupOdbToolset as dgo
    import connectorBehavior
    mdb.models.changeKey(fromName='Model-1', toName='Block Model')
    session.viewports['Viewport: 1'].setValues(displayedObject=None)
    s1 = mdb.models['Block Model'].ConstrainedSketch(name='__profile__',
        sheetSize=200.0)
    g, v, d, c = s1.geometry, s1.vertices, s1.dimensions, s1.constraints
    s1.setPrimaryObject(option=STANDALONE)
    s1.rectangle(point1=(0.0, 0.0), point2=(22.5, 13.75))
    s1.ObliqueDimension(vertex1=v[3], vertex2=v[0], textPoint=(16.4174423217773,
        -4.17431116104126), value=25.0)
    s1.ObliqueDimension(vertex1=v[0], vertex2=v[1], textPoint=(-5.90002059936523,
        7.25688123703003), value=15.0)
    p = mdb.models['Block Model'].Part(name='Block', dimensionality=THREE_D,
        type=DEFORMABLE_BODY)
    p = mdb.models['Block Model'].parts['Block']
    p.BaseSolidExtrude(sketch=s1, depth=20.0)
    s1.unsetPrimaryObject()
    p = mdb.models['Block Model'].parts['Block']
    session.viewports['Viewport: 1'].setValues(displayedObject=p)
    del mdb.models['Block Model'].sketches['__profile__']
    p = mdb.models['Block Model'].parts['Block']
    e1 = p.edges
    p.Round(radius=1.0, edgeList=(e1[4], ))
```

You notice that the name of our macro 'BlockMacro' is the name of the function (indicated by the **def** keyword). In addition there are a number of **import** statements to import all modules that might be required by almost any script. Other than that the statements are the same as the ones in the replay file. Essentially what Abaqus has done is given you the statements of the replay file that were written while the macro was recording.

You can run an existing macro from the **Macro Manager** by choosing it from the list and clicking **Run**. In our case this will only work in a new model because we rename 'Model-1' to 'Block Model'. (If no 'Model-1' is present then you will get an error.) If you'd used the macro to do something like create a material, you could then run the macro inside any instance of Abaqus and it would create that material for you again.

You can see how macros help you perform a repetitive task without actually writing a single Python statement yourself. The added advantage is that users of Abaqus Student Edition can use this in place of the replay file which they do not have access to. In fact even if you're using the Research or Commercial editions of Abaqus, you may prefer to create a macro of a task you are trying to script in order to see which commands Abaqus/CAE uses as opposed to reading the replay file which will include everything from the moment your Abaqus session began.

6.5 IDEs and Text Editors

Python scripts are basically text files with a .py extension. This means you can write them in the most basic of text editors – Notepad – which ships with every version of Windows. However you are unlikely to enjoy this experience too much, especially since Python code needs to be indented. In addition notepad displays everything in one font color, including things like comments, function names and import statements. This makes everything harder to read, and also harder to debug. You might enjoy scripting with something a little more sophisticated.

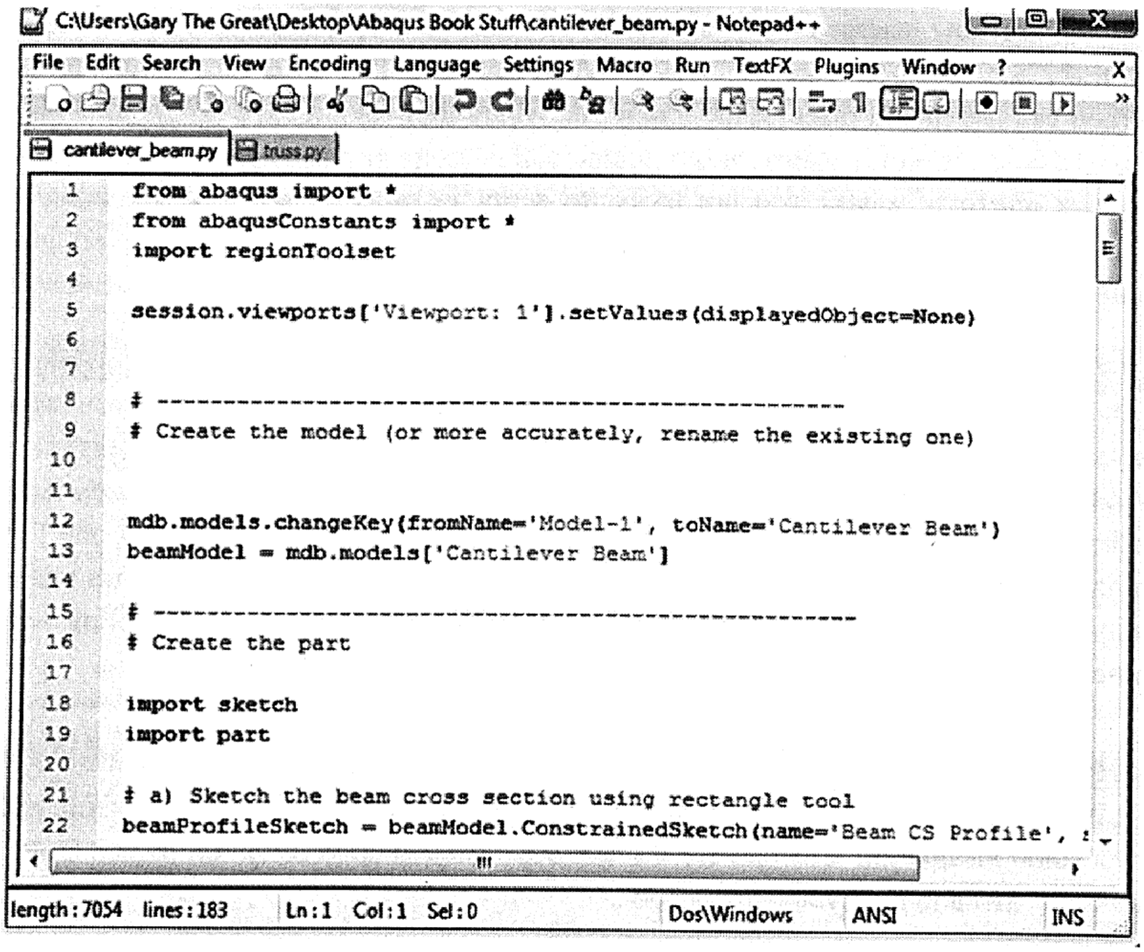
6.5.1 IDLE

IDLE is an IDE (integrated development environment) that is installed by default with any Python installation. Chances are it is already installed on your system if you look in the 'Start' menu in the Python application.

If you were programming in pure Python you could run your scripts directly from IDLE. However since you will be writing scripts for Abaqus, they would need to be run from within Abaqus/CAE (**File > Run Script**) or from the command line. You will essentially use IDLE as a text editor.

6.5.2 Notepad ++

Notepad++ is a free code editor. It is like an enhanced version of Notepad that is great for writing code. It has syntax highlighting and also displays line numbers next to statements which helps with debugging code. In addition you can have multiple files open in multiple tabs and switch between them easily. It supports a number of popular languages, including Python, and will choose the appropriate language and coloring based on the file extension.


C:\Users\Gary The Great\Desktop\Abaqus Book Stuff\cantilever_beam.py - Notepad++
File Edit Search View Encoding Language Settings Macro Run TextFX Plugins Window ?
cantilever_beam.py
truss.py
length : 7054 lines : 183 Ln : 1 Col : 1 Sel : 0 Dos\Windows ANSI INS

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)


# -----------------------------------------------------------
# Create the model (or more accurately, rename the existing one)


mdb.models.changeKey(fromName='Model-1', toName='Cantilever Beam')
beamModel = mdb.models['Cantilever Beam']

# -----------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the beam cross section using rectangle tool
beamProfileSketch = beamModel.ConstrainedSketch(name='Beam CS Profile', s
```

All of the scripts for this book were written in Notepad++, it is my personal favorite. The website for Notepad++ (at the time of publication) is http://notepad-plus-plus.org/

6.5.3 Abaqus PDE

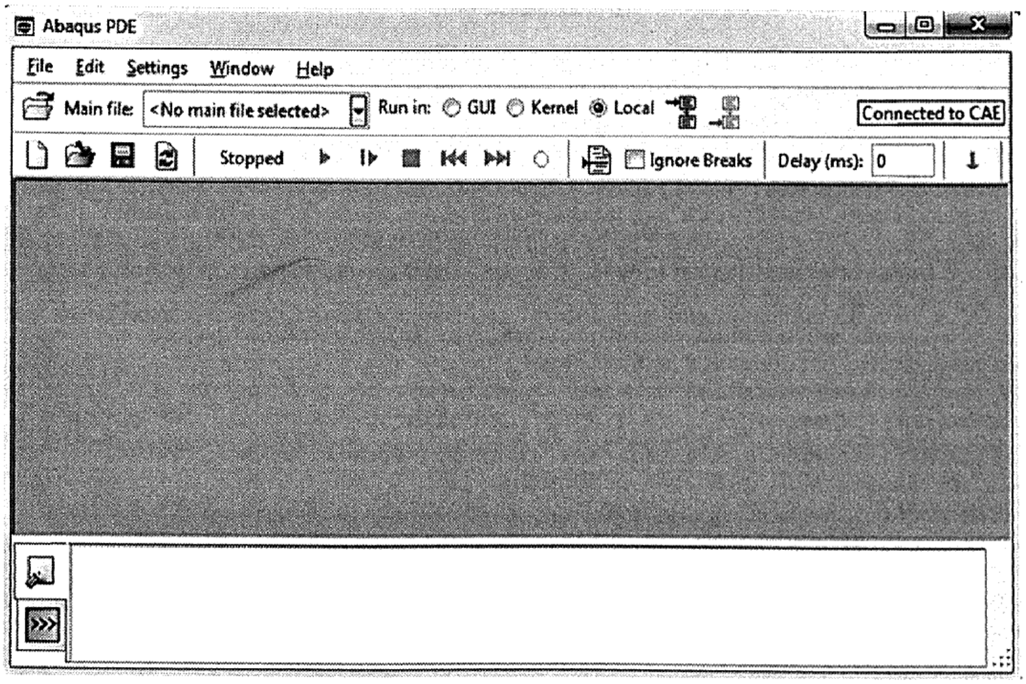
Abaqus Python Development Environment (PDE) is an application that comes bundled with Abaqus. It allows you to create and edit scripts, run then, and offers debugging features.

You can start Abaqus PDE from within Abaqus/CAE by going to **File > Abaqus PDE…** Alternatively you can start it by going to the system command prompt and typing (in Abaqus Student Edition version 6.10-2)

```
abq6102se -pde
```

You will need to change the 'abq6102se' to the command required to run your version of Abaqus (refer to Chapter 2 for details).

If you start Abaqus PDE from within Abaqus/CAE, it will be connected to CAE, as indicated by the words "Connected to CAE" displayed in the top left of the Abaqus PDE window (see figure). This means you will be using your Abaqus license tokens. If you run it from the command line however, Abaqus PDE will not be connected to CAE.



Abaqus PDE gives you the option to run the script in 3 modes – 'GUI', 'Kernel' and 'Local' in the toolbar (see figure). You choose the correct one depending on whether the scripts should run in Abaqus/CAE GUI, the Abaqus/CAE kernel or locally. By default .guiLog scripts run in GUI, and .py scripts run in the kernel.

What are .guiLog scripts? These are similar to macros, in the sense that you can perform some tasks in the GUI and a Python script will be written recording this. However .guiLog scripts describe the activity of the user in the GUI, which buttons were clicked and so on, whereas .py scripts record the Python commands called. So for example, when you close a dialog box, a .guiLog script records the fact that you clicked on a certain button, whereas a .py script records which function was called depending on the options you checked off in the dialog box.

This may be better understood with a demonstration. Open a new file in Abaqus PDE (**File > New Model Database > With Standard/Explicit Model**). Click the **Start Recording** button in the toolbar which appears as a red circle. Repeat all the steps from the previous section to rename the model, create a block and round an edge. Then click the **Stop Program** button represented by the solid square.

```
from abaqusTester import *
import abaqusGui
selectTreeListItem('Model Tree', ('Model Database','Models','Model-1'), 0)
showTreeListContextMenu('Model Tree')
selectMenuItem('Model Tree Menu + Rename')
setTextFieldValue('Rename Model + Rename To', 'Block Model')
pressButton('Rename Model + Ok')
selectTreeListItem('Model Tree', ('Model Database','Models','Block Model','Parts'),
0)
doubleClickTreeListItem('Model Tree', ('Model Database','Models','Block
Model','Parts'), 0)
setTextFieldValue('prtG_PartCreateDB + Create', 'Block Part')
pressButton('prtG_PartCreateDB + Continue')
pressButton('Sketcher GeomToolbox + Rectangle')
clickInViewport('Viewport: 1', (0.256754, -0.321101), 0.728166, LEFT_BUTTON)
clickInViewport('Viewport: 1', (27.216, 17.1468), 0.728166, LEFT_BUTTON)
pressButton('Sketcher ConsToolbox + Add Dimension')
clickInViewport('Viewport: 1', (5.00671, -0.0642202), 0.728166, LEFT_BUTTON)
clickInViewport('Viewport: 1', (8.21614, -8.15596), 0.728166, LEFT_BUTTON)
commitTextFieldValue('skcK_DimensionProcedure + New Dimension', '25')
clickInViewport('Viewport: 1', (-0.513509, 4.55963), 0.728166, LEFT_BUTTON)
clickInViewport('Viewport: 1', (-6.54723, 4.55963), 0.728166, LEFT_BUTTON)
commitTextFieldValue('skcK_DimensionProcedure + New Dimension', '15')
pressButton('Procedure + Cancel')
pressButton('prtK_NewPartProc + Done')
pressButton('prtG_ExtrudeFeatureDB + Ok')
pressFlyoutItem('Create Blend Flyout + Round/Fillet')
clickInViewport('Viewport: 1', (-0.112969, 0.0541739), 0.0044191, LEFT_BUTTON)
pressButton('prtK_BlendRoundProc + Done')
commitTextFieldValue('prtK_BlendRoundProc + Radius', '1.0')
pressButton('Procedure + Cancel')
```

You will notice that as you were working in the GUI, the .guiLog was storing a log of everything you did in the GUI. It is evident that this log is of a different nature compared to a script. It records information such as which button you clicked, where in the viewport you clicked, and even trivial things like clicking the 'cancel procedure' red X.

Let's see how this guiLog can be used. Create a new model in Abaqus by going to **File > New Model Database > With Standard/Explicit Model**. Leave the .guiLog file open in Abaqus PDE

Click the 'Play' button represented by the solid triangle. You will see that each of the lines in the .guiLog is highlighted one by one. At the same time, in the Abaqus/CAE window, you see the corresponding task being performed. It is almost like you are watching the person who created the guiLog at work except that you do not see their mouse cursor moving about. You may find it useful to pass a .guiLog file along to coworkers to demonstrate how you performed a task in the GUI.

At the bottom of the Abaqus PDE window, you see a message area and a command line interface similar to the one you see in Abaqus/CAE. The difference is that this is a GUI Command Line Interface whereas the one in Abaqus/CAE is a Kernel Command Line Interface. You will understand the difference between the two when we cover GUI customization in the last few chapters of the book. For now just know that a GUI API can be called from here, so you could for instance check the functionality of a dialog box.

Abaqus PDE has a number of debugging features. You can use the '**Set/Clear Breakpoint at cursor location**' tool to set a breakpoint at any statement (does not include comments or empty lines) and the statements before that point will be executed. You can then choose to contine after a breakpoint if you wish.

You can access the Abaqus PDE debugger using **Window > Debugger**. The debugger is displayed between the Abaqus PDE main window and the message area. You can display the watch list by clicking on 'show watch'. This allows you to watch the value of variables as the script executes. To add a variable to the watch list right click on it in the main window and select **Add Watch: (variable name).** This could be very useful for debugging purposes. Then again in Python it is quite common to debug code using 'Print' statements so go with your preference.

6.5.4 Other options

A free IDE popular in the Python world is PythonWin. Some individuals prefer this to IDLE. Another popular text editor is TextPad, which is quite similar to Notepad++. However this is not currently free but I believe you can try a fully functional evaluation version. A Google search will reveal many more options.

6.6 What's Next?

You will be relying heavily on replay files or macros when writing scripts, and you now understand how these work. Hopefully you've also decided on an IDE or text editor to use for subsequent examples.

You now have a basic knowledge of the Python programming language and an understanding of how to write scripts for Abaqus. You also know about replay files and macros. It is time to proceed to Part 2 of this book.

PART 2 – LEARN BY EXAMPLE

We shall now begin scripting in earnest. Every chapter in Part 2 is made up of one example. Each example introduces new topics and concepts. The first few examples/chapters create simple single run simulations. Subsequent chapters delve into topics of optimization, parameterization, output database processing and job monitoring.

For each example, the steps to perform the study in Abaqus/CAE are described. This is to ensure that you know how to run the simulation in the GUI before you script it. Instead of reading the procedure you may watch the videos on the book website. Following the CAE procedure is the corresponding script, and line-by-line explanation.

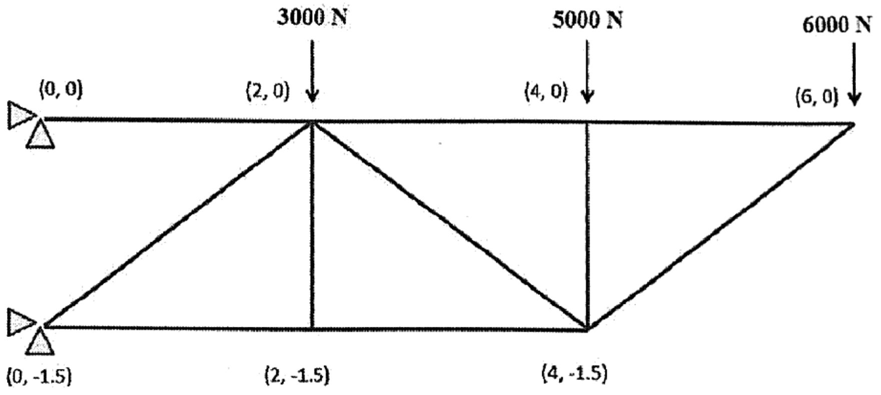
You don't necessarily need to read all of these chapters. However each of them demonstrates different tasks and if something is repeated the previous occurrence will be referenced. It might help to skim through each example and form a general idea of what each script does, so that you know where to find reusable code when writing your own scripts.

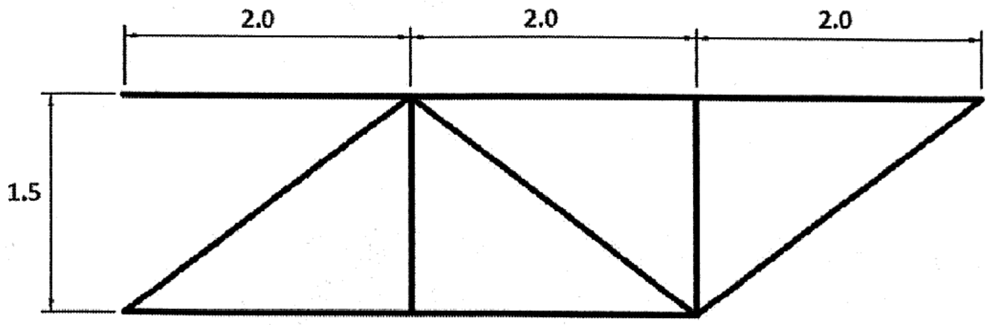
7

Static Analysis of a Loaded Truss

7.1 Introduction

In this chapter we will write a script to perform a static analysis on a truss. The problem is displayed in the figure. One end of the truss is fixed to a wall while the other end is free. Concentrated forces of 3000 N, 5000 N and 6000 N are applied to the nodes of the truss in the –Y direction.





(Dimensions are in meters)

In this example the following tasks will be demonstrated first using Abaqus/CAE, and then using a Python script.

- Create a part
- Assign materials
- Assign sections
- Create an Assembly
- Create a static, general step
- Request field outputs
- Assign loads
- Assign boundary conditions
- Create a mesh
- Create and submit a job
- Plot overlaid deformed and undeformed results and display node numbers on plot
- Plot field outputs

The new topics covered are:

- Model / Preprocessing
    - Work in 2D
    - Create wire features
    - Create sections of type 'truss' and specify cross sectional areas
    - Use truss elements (with pin joints)
    - Use concentrated force loads
- Results / Post-processing
    - Allow multiple plot states (both deformed and undeformed plots overlaid)
    - Use Common Plot Options -> Show Node Labels
    - Display field output as color contours

7.2 Procedure in GUI

You can perform the simulation in Abaqus/CAE by following the steps listed below. You can either read through these, or watch the video demonstrating the process on the book website.

1. Rename **Model-1** to **Truss Structure**
   a. Right-click on Model-1 in Model Database
   b. Choose **Rename..**
   c. Change name to **Truss Structure**
2. Create the part
   a. Double-click on **Parts** in Model Database. **Create Part** window is displayed.
   b. Set **Name** to **Truss**
   c. Set **Modeling Space** to **2D Planar**
   d. Set **Type** to **Deformable**
   e. Set **Base Feature** to **Wire**
   f. Set **Approximate Size to 10**
   g. Click **OK**. You will enter Sketcher mode.
3. Sketch the truss
   a. Use the **Create Lines:Connected**tool to draw the profile of the truss
   b. Split the lines using the **Split** tool
   c. Use **Add Constraints > Equal Length** tool to set the lengths of the required truss elements to be equal
   d. Use the **Add Dimension** tool to set the length of the horizontal elements to 2 m and the length of the vertical elements to 1.5 m.
   e. Click **Done** to exit the sketcher.
4. Create the material
   a. Double-click on **Materials** in the Model Database. **Edit Material** window is displayed
   b. Set **Name** to **AISI 1005 Steel**
   c. Select **General > Density**. Set **Mass Density** to **7872** (which is 7.872 g/cc)
   d. Select **Mechanical > Elasticity > Elastic**. Set **Young's Modulus** to **200E9** (which is 200 GPa) and **Poisson's Ratio** to **0.29**.
5. Assign sections
   a. Double-click on **Sections** in the Model Database. **Create Section** window is displayed
   b. Set **Name** to **Truss Section**
   c. Set **Category** to **Beam**
   d. Set **Type** to **Truss**
   e. Click **Continue...** The **Edit Section** window is displayed.
   f. In the **Basic** tab, set **Material** to **the AISI 1005 Steel** which was defined in the create material step.

   g. Set **Cross-sectional Area** to **3.14E-4**
   h. Click **OK.**
6. Assign the section to the truss
   a. Expand the **Parts** container in the Model Database. Expand the part **Truss.**
   b. Double-click on **Section Assignments**
   c. You see the message **Select the regions to be assigned a section** displayed below the viewport
   d. Click and drag with the mouse to select the entire truss.
   e. Click **Done.** The **Edit Section Assignment** window is displayed.
   f. Set **Section** to **Truss Section.**
   g. Click **OK.**
   h. Click **Done.**
7. Create the Assembly
   a. Double-click on **Assembly** in the Model Database. The viewport changes to the **Assembly Module.**
   b. Expand the **Assembly** container.
   c. Double-click on **Instances.** The **Create Instance** window is displayed.
   d. Set **Parts** to **Truss**
   e. Set **Instance Type** to **Dependent (mesh on part)**
   f. Click **OK.**
8. Create Steps
   a. Double-click on **Steps** in the Model Database. The **Create Step** window is displayed.
   b. Set **Name** to **Loading Step**
   c. Set **Insert New Step After** to **Initial**
   d. Set **Procedure Type** to **General > Static, General**
   e. Click **Continue..** The **Edit Step** window is displayed
   f. In the **Basic** tab, set **Description** to **Loads are applied to the truss in this step.**
   g. Click **OK.**
9. Request Field Outputs
   a. Expand **the Field Output Requests** container in the Model Database.
   b. Right-click on **F-Output-1** and choose Rename…
   c. Change the name to **Selected Field Outputs**
   d. Double-click on **Selected Field Outputs** in the Model Database. The **Edit Field Output Request** window is displayed.

  e. Select the desired variables by checking them off in the **Output Variables** list. The variables we want are **S (stress components and invariants), U (translations and rotations), RF (reaction forces and moments)**, and **CF (concentrated forces and moments).** Uncheck the rest. You will notice that the text box above the output variable list displays **S,U,RF,CF**
  f. Click **OK**.

10. Assign Loads
    a. Double-click on **Loads** in the Model Database. The **Create Load** window is displayed
    b. Set **Name** to **Force1**
    c. Set **Step** to **Loading Step**
    d. Set **Category** to **Mechanical**
    e. Set **Type for Selected Step** to **Concentrated Force**
    f. Click **Continue...**
    g. You see the message **Select points for the load displayed below the viewport**
    h. Select the upper left node by clicking on it
    i. Click **Done**. The **Edit Load** window is displayed
    j. Set **CF2** to **-3000** to apply a 3000 N force in downward (negative Y) direction
    k. Click **OK**
    l. You will see the force displayed with an arrow in the viewport on the selected node
    m. Repeat steps a-l two more times, once each for the upper middle and upper right node. Name the forces **Force2** and **Force3**, and set them to **-5000** and -**6000** respectively.
11. Apply boundary conditions
    a. Double-click on **BCs** in the Model Database. The **Create Boundary Condition** window is displayed
    b. Set **Name** to **Pin1**
    c. Set **Step** to **Initial**
    d. Set **Category** to **Mechanical**
    e. Set **Types for Selected Step** to **Displacement/Rotation**
    f. Click **Continue...**
    g. You see the message **Select regions for the boundary condition** displayed below the viewport

   h. Select the two nodes on the extreme left. You can press the "Shift" key on your keyboard to select both at the same time.
   i. Click **Done**. The **Edit Boundary Condition** window is displayed.
   j. Check off **U1** and **U2**. This will create a pin joint which does not allow translation but permits rotation.
   k. Click **OK**.

12. Create the mesh
   a. Expand the **Parts** container in the Model Database.
   b. Expand **Truss**
   c. Double-click on **Mesh (Empty)**. The viewport window changes to the **Mesh module** and the tools in the toolbar are now meshing tools.
   d. Using the menu bar click on **Mesh > Element Type ...**
   e. You see the message **Select the regions to be assigned element types** displayed below the viewport
   f. Click and drag using your mouse to select the entire truss.
   g. Click **Done**. The **Element Type** window is displayed.
   h. Set **Element Library** to **Standard**
   i. Set **Geometric Order** to **Linear**
   j. Set **Family** to **Truss**
   k. You will notice the message **T2D2: A 2-node linear 2-D truss**
   l. Click **OK**
   m. Click **Done**
   n. Using the menu bar lick on **Seed > Edge by Number**
   o. You see the message **Select the regions to be assigned local seeds** displayed below the viewport
   p. Click and drag using your mouse to select the entire truss
   q. Click **Done**.
   r. You see the prompt **Number of elements along the edges** displayed below the viewport.
   s. Set it to **1** and press the "Enter" key on your keyboard
   t. Click **Done**
   u. Using the menu bar click on **Mesh > Part**
   v. You see the prompt **OK to mesh the part?** displayed below the viewport
   w. Click **Yes**

13. Create and submit the job

a. Double-click on **Jobs** in the Model Database. The **Create Job** window is displayed
b. Set **Name** to **TrussAnalysisJob**
c. Set **Source** to **Model**
d. Select **Truss Structure** (it is the only option displayed)
e. Click **Continue..** The **Edit Job** window is displayed
f. Set **Description** to **Analysis of truss under concentrated loads**
g. Set **Job Type** to **Full Analysis**.
h. Leave all other options at defaults
i. Click **OK**
j. Expand the**Jobs** container in the Model Database
k. Right-click on **TrussAnalysisJob** and choose **Submit**. This will run the simulation. You will see the following messages in the message window:
   The job input file "TrussAnalysisJob.inp" has been submitted for analysis.
   Job TrussAnalysisJob: Analysis Input File Processor completed successfully
   Job TrussAnalysisJob: Abaqus/Standard completed successfully
   Job TrussAnalysisJob completed successfully

14. Plot results deformed and undeformed
   a. Right-click on **TrussAnalysisJob (Completed)** in the Model Database. Choose **Results**.The viewport changes to the **Visualization** module.
   b. In the toolbar click the **Plot Undeformed Shape** tool. The truss is displayed in its undeformed state.
   c. In the toolbar click the **Plot Deformed Shape** tool. The truss is displayed in its deformed state.
   d. In the toolbar click the **Allow Multiple Plot States** tool. Then click the **Plot Undeformed Shape** tool. Both undeformed and deformed shapes are now visible superimposed on one another.
   e. Click again on the **Allow Multiple Plot States** tool to disallow this feature. Click on **Plot Deformed Shape** to have the deformed state displayed once again in the viewport.
   f. In the toolbar click the **Common Options** tool. The **Common Plot Options** window is displayed.
   g. In the **Labels** tab check **Show node labels**
   h. Click **OK**. The nodes are now numbered on the truss in the viewport.

15. Plot Field Outputs
    a. Using the menu bar click on **Result > Field Output...** The **Field Output** window is displayed.
    b. In the **Output Variable** list select **U** which has the description **Spatial displacement at nodes.** In the **Invariant** list **Magnitude** is displayed. In the **Components** list **U1** and **U2** are displayed
    c. In the **Invariant** list select **Magnitude**. Click **Apply**. You might see the **Select Plot State** window with the message **The field output variable has been set, but it will not affect the current Display Group instance unless a different plot state is selected below.** For the **Plot state** select **Contour** and click **OK**.
    d. Click **OK** to close the **Field Output** window. You notice in the viewport a color contour has been applied on the truss with a legend indicating the **U** magnitude.
    e. Once again, using the menu bar click on **Result > Field Output...** The **Field Output** window is displayed.
    f. In the **Output Variable** list select **U** which has the description **Spatial displacement at nodes**.
    g. In the **Component** list select **U1**.
    h. Click **OK**. The visualization updates to display **U1** which is displacement in the X direction.

7.3 Python Script

The following Python script replicates the above procedure for the static analysis of the truss. You can find it in the source code accompanying the book in **truss.py**. You can run it by opening a new model in Abaqus/CAE **(File > New Model database > With Standard/Explicit Model)** and running it with **File > Run Script…**

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# -------------------------------------------------------------------
# Create the model


mdb.models.changeKey(fromName='Model-1', toName='Truss Structure')
trussModel = mdb.models['Truss Structure']
```

```
# ----------------------------------------------------------------------
# Create the part

import sketch
import part

trussSketch = trussModel.ConstrainedSketch(name='2D Truss Sketch', sheetSize=10.0)
trussSketch.Line(point1=(0, 0), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, 0))
trussSketch.Line(point1=(4, 0), point2=(6, 0))
trussSketch.Line(point1=(0, -1.5), point2=(2,-1.5))
trussSketch.Line(point1=(2, -1.5), point2=(4,-1.5))
trussSketch.Line(point1=(0, -1.5), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, -1.5))
trussSketch.Line(point1=(4, -1.5), point2=(6, 0))
trussSketch.Line(point1=(2, 0), point2=(2, -1.5))
trussSketch.Line(point1=(4, 0), point2=(4, -1.5))

trussPart = trussModel.Part(name='Truss', dimensionality=TWO_D_PLANAR,
                            type=DEFORMABLE_BODY)
trussPart.BaseWire(sketch=trussSketch)

# ----------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs modulus
# and poissons ratio
trussMaterial = trussModel.Material(name='AISI 1005 Steel')
trussMaterial.Density(table=((7872, ),       ))
trussMaterial.Elastic(table=((200E9, 0.29), ))

# ----------------------------------------------------------------------
# Create a section and assign the truss to it
import section

trussSection = trussModel.TrussSection(name='Truss Section',
                                       material='AISI 1005 Steel',
                                       area=3.14E-4)

edges_for_section_assignment = trussPart.edges.findAt(((1.0, 0.0, 0.0), ),
                                                       ((3.0, 0.0, 0.0), ),
                                                       ((5.0, 0.0, 0.0), ),
                                                       ((1.0, -1.5, 0.0), ),
                                                       ((3.0, -1.5, 0.0), ),
                                                       ((1.0, -0.75, 0.0), ),
                                                       ((3.0, -0.75, 0.0), ),
                                                       ((5.0, -0.75, 0.0), ),
                                                       ((2.0, -0.75, 0.0), ),
                                                       ((4.0, -0.75, 0.0), ))
```

```
truss_region = regionToolset.Region(edges=edges_for_section_assignment)
trussPart.SectionAssignment(region=truss_region, sectionName='Truss Section')

# -------------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
trussAssembly = trussModel.rootAssembly
trussInstance = trussAssembly.Instance(name='Truss Instance', part=trussPart,
                                                              dependent=ON)

# -------------------------------------------------------------------------
# Create the step

import step

# Create a static general step
trussModel.StaticStep(name='Loading Step', previous='Initial',
                      description='Loads are applied to the truss in this step')

# -------------------------------------------------------------------------
# Create the field output request

# Change the name of field output request 'F-Output-1' to 'Selected Field Outputs'
trussModel.fieldOutputRequests.changeKey(fromName='F-Output-1',
                                          toName='Selected Field Outputs')

# Since F-Output-1 is applied at the 'Loading Step' step by default,
# 'Selected Field Outputs' will be too
# We only need to set the required variables
trussModel.fieldOutputRequests['Selected Field Outputs'].setValues(variables=('S',
                                                                   'U', 'RF', 'CF'))

# -------------------------------------------------------------------------
# Create the history output request
# We want the defaults so we'll leave this section blank

# -------------------------------------------------------------------------
# Apply loads

# Concentrated load of 3000 N on first node
vertex_coords_for_first_force = (2.0, 0.0, 0.0)
vertex_for_first_force = trussInstance.vertices \
                                 .findAt((vertex_coords_for_first_force,))
trussModel.ConcentratedForce(name='Force1', createStepName='Loading Step',
                             region=(vertex_for_first_force,), cf2=-3000.0,
                             distributionType=UNIFORM)

# Concentrated load of 5000 N on second node
```

```
vertex_coords_for_second_force = (4.0, 0.0, 0.0)
vertex_for_second_force = trussInstance.vertices \
                                          .findAt((vertex_coords_for_second_force,))
trussModel.ConcentratedForce(name='Force2', createStepName='Loading Step',
                              region=(vertex_for_second_force,), cf2=-5000.0,
                              distributionType=UNIFORM)

# Concentrated load of 6000 N on third node
vertex_for_third_force = trussInstance.vertices.findAt(((6.0, 0.0, 0.0),))
trussModel.ConcentratedForce(name='Force3', createStepName='Loading Step',
                              region=(vertex_for_third_force,), cf2=-6000.0,
                              distributionType=UNIFORM)

# -----------------------------------------------------------------------
# Apply boundary conditions

# Pin left end of upper member
vertex_coords_for_first_pin = (0.0, 0.0, 0.0)
vertex_for_first_pin = trussInstance.vertices \
                                            .findAt((vertex_coords_for_first_pin,))
trussModel.DisplacementBC(name='Pin1', createStepName='Initial',
                           region=(vertex_for_first_pin,),
                           u1=SET, u2=SET, ur3=UNSET,
                           amplitude=UNSET, distributionType=UNIFORM)

# Pin left end of lower member
vertex_coords_for_second_pin = (0.0, -1.5, 0.0)
vertex_for_second_pin = trussInstance.vertices \
                                           .findAt((vertex_coords_for_second_pin,))
trussModel.DisplacementBC(name='Pin2', createStepName='Initial',
                           region=(vertex_for_second_pin,),
                           u1=SET, u2=SET, ur3=UNSET,
                           amplitude=UNSET, distributionType=UNIFORM)

#ALTERNATIVE METHOD
#vertex_coords_for_first_pin = (0.0, 0.0, 0.0)
#vertex_coords_for_second_pin = (0.0, -1.5, 0.0)
#vertices_for_pins = trussInstance.vertices.findAt((vertex_coords_for_first_pin,),
(vertex_coords_for_second_pin,))
#trussModel.DisplacementBC(name='Pins', createStepName='Initial',
region=(vertices_for_pins,), u1=SET, u2=SET, ur3=UNSET, amplitude=UNSET,
#distributionType=UNIFORM)

# -----------------------------------------------------------------------
# Create the mesh

import mesh

truss_mesh_region = truss_region
edges_for_meshing = edges_for_section_assignment

mesh_element_type=mesh.ElemType(elemCode=T2D2, elemLibrary=STANDARD)
```

```
trussPart.setElementType(regions=truss_mesh_region,
                         elemTypes=(mesh_element_type, ))
trussPart.seedEdgeByNumber(edges=edges_for_meshing, number=1)
trussPart.generateMesh()

# -------------------------------------------------------------------------
# Create and run the job

import job

# Create the job
mdb.Job(name='TrussAnalysisJob', model='Truss Structure', type=ANALYSIS,
        explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE,
        description='Analysis of truss under concentrated loads',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)

# Run the job
mdb.jobs['TrussAnalysisJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['TrussAnalysisJob'].waitForCompletion()

# End of run job

# -------------------------------------------------------------------------
# Post processing

import visualization

truss_Odb_Path = 'TrussAnalysisJob.odb'
odb_object = session.openOdb(name=truss_Odb_Path)

session.viewports['Viewport: 1'].setValues(displayedObject=odb_object)
session.viewports['Viewport: 1'].odbDisplay.display \
                                        .setValues(plotState=(DEFORMED, ))

# Plot the deformed state of the truss
truss_deformed_viewport = session.Viewport(name='Truss in Deformed State')
truss_deformed_viewport.setValues(displayedObject=odb_object)
truss_deformed_viewport.odbDisplay.display.setValues(plotState=(UNDEFORMED,
                                                               DEFORMED, ))
truss_deformed_viewport.odbDisplay.commonOptions.setValues(nodeLabels=ON)
truss_deformed_viewport.odbDisplay.commonOptions.setValues(elemLabels=ON)
truss_deformed_viewport.setValues(origin=(0.0, 0.0), width=250, height=160)

# Plot the output variable U (spatial displacements at nodes) as its Magnitude
# invariant
# This is the equivalent of going to Report > Field Output and choosing to
# output U with Invariant: Magnitude
```

```
truss_displacements_magnitude_viewport= session \
                       .Viewport(name='Truss Displacements at Nodes (Magnitude)')
truss_displacements_magnitude_viewport.setValues(displayedObject=odb_object)
truss_displacements_magnitude_viewport.odbDisplay \
                                    .setPrimaryVariable(variableLabel='U',
                                                        outputPosition=NODAL,
                                                        refinement=(INVARIANT,
                                                                    'Magnitude'))
truss_displacements_magnitude_viewport.odbDisplay.display \
                                       .setValues(plotState=(CONTOURS_ON_DEF, ))
truss_displacements_magnitude_viewport.setValues(width=250, height=160)
truss_displacements_magnitude_viewport.offset(20,-10)

# Plot the output variable U (spatial displacements at nodes) as its U1 component
# This is the equivalent of going to Report > Field Output and choosing to output
# U with Component: U1
truss_displacements_U1_viewport= session \
                    .Viewport(name='Truss Displacements at Nodes (U1 Component')
truss_displacements_U1_viewport.setValues(displayedObject=odb_object)
truss_displacements_U1_viewport.odbDisplay \
                            .setPrimaryVariable(variableLabel='U',
                                                outputPosition=NODAL,
                                                refinement=(COMPONENT, 'U1'))
truss_displacements_U1_viewport.odbDisplay.display \
                                       .setValues(plotState=(CONTOURS_ON_DEF, ))
truss_displacements_U1_viewport.setValues(width=250, height=160)
truss_displacements_U1_viewport.offset(40,-20)

session.viewports['Viewport: 1'].sendToBack()
```

7.4 Examining the Script

Let's go through the entire script, statement by statement, and understand how it works.

7.4.1 Initialization (import required modules)

The block dealing with this initialization is

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)
```

These statements are identical to those used in the Cantilever Beam example and were explained in section 4.3.1 on page65

7.4.2 Create the model

The following code block creates the model

```
# ---------------------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Truss Structure')
trussModel = mdb.models['Truss Structure']
```

These statements rename the model from 'Model-1' to 'Truss Structure'. They are almost identical to those used in the Cantilever Beam example and were explained in section 4.3.2 on page 67.

7.4.3 Create the part

The following block creates the part

```
# ---------------------------------------------------------------------
# Create the part

import sketch
import part

trussSketch = trussModel.ConstrainedSketch(name='2D Truss Sketch', sheetSize=10.0)
trussSketch.Line(point1=(0, 0), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, 0))
trussSketch.Line(point1=(4, 0), point2=(6, 0))
trussSketch.Line(point1=(0, -1.5), point2=(2,-1.5))
trussSketch.Line(point1=(2, -1.5), point2=(4,-1.5))
trussSketch.Line(point1=(0, -1.5), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, -1.5))
trussSketch.Line(point1=(4, -1.5), point2=(6, 0))
trussSketch.Line(point1=(2, 0), point2=(2, -1.5))
trussSketch.Line(point1=(4, 0), point2=(4, -1.5))

trussPart = trussModel.Part(name='Truss', dimensionality=TWO_D_PLANAR,
                            type=DEFORMABLE_BODY)
trussPart.BaseWire(sketch=trussSketch)
```

```
import sketch
import part
```

These statements import the sketch and part modules into the script, thus providing access to the objects related to sketches and parts. They were explained in section 4.3.3 on page69.

```
trussSketch = trussModel.ConstrainedSketch(name='2D Truss Sketch', sheetSize=10.0)
```

This statement creates a constrained sketch object by calling the **ConstrainedSketch()** method of the **Model** object. This was explained in section 4.3.3 on page 69.

```
trussSketch.Line(point1=(0, 0), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, 0))
trussSketch.Line(point1=(4, 0), point2=(6, 0))
trussSketch.Line(point1=(0, -1.5), point2=(2,-1.5))
trussSketch.Line(point1=(2, -1.5), point2=(4,-1.5))
trussSketch.Line(point1=(0, -1.5), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, -1.5))
trussSketch.Line(point1=(4, -1.5), point2=(6, 0))
trussSketch.Line(point1=(2, 0), point2=(2, -1.5))
trussSketch.Line(point1=(4, 0), point2=(4, -1.5))
```

The statements use the **Line()** method of the **ConstrainedSketchGeometry** object. The **ConstrainedSketchGeometry** object stores the geometry of a sketch, such as lines, circles, arcs, and construction lines. The sketch module defines **ConstrainedSketchGeometry** objects. The first parameter **point1** is a pair of floats specifying the coordinates of the first endpoint of the line. The second parameter **point2** is a pair of floats specifying the coordinates of the second endpoint.

```
trussPart = trussModel.Part(name='Truss', dimensionality=TWO_D_PLANAR,
                            type=DEFORMABLE_BODY)
```

This statement creates a **Part** object and places it in the parts repository. The **name** of the part (its key in the repository) is set to 'Truss' and its **dimensionality** is set to a SymbolicConstant **TWO_D_PLANAR** which defines it to be a 2D part. It is defined to be of the type deformable body using the **DEFORMABLE_BODY** SymbolicConstant.

```
trussPart.BaseWire(sketch=trussSketch)
```

This statement calls the **BaseWire()** method which creates a Feature object by creating a planar wire from the ConstrainedSketch object **trussSketch** which is passed to it as an argument. Feature objects were explained in section 4.3.3 on page 70.

7.4.4 Define the materials

The following block of code creates the material for the simulation

```
# -------------------------------------------------------------------------
# Create material

import material
```

```
# Create material AISI 1005 Steel by assigning mass density, youngs modulus
# and poissons ratio
trussMaterial = trussModel.Material(name='AISI 1005 Steel')
trussMaterial.Density(table=((7872, ),       ))
trussMaterial.Elastic(table=((200E9, 0.29), ))
```

The statements are almost identical to those used in the Cantilever Beam example and were explained in section 4.3.4 on page 71.

7.4.5 Create sections and make section assignments

The following block creates the section and assigns it to the truss

```
# -------------------------------------------------------------------------
# Create a section and assign the truss to it
import section

trussSection = trussModel.TrussSection(name='Truss Section',
                                       material='AISI 1005 Steel',
                                       area=3.14E-4)

edges_for_section_assignment = trussPart.edges.findAt(((1.0, 0.0, 0.0), ),
                                                      ((3.0, 0.0, 0.0), ),
                                                      ((5.0, 0.0, 0.0), ),
                                                      ((1.0, -1.5, 0.0), ),
                                                      ((3.0, -1.5, 0.0), ),
                                                      ((1.0, -0.75, 0.0), ),
                                                      ((3.0, -0.75, 0.0), ),
                                                      ((5.0, -0.75, 0.0), ),
                                                      ((2.0, -0.75, 0.0), ),
                                                      ((4.0, -0.75, 0.0), ))

truss_region = regionToolset.Region(edges=edges_for_section_assignment)
trussPart.SectionAssignment(region=truss_region, sectionName='Truss Section')
```

```
import section
```

This statement imports the section module making its properties and methods accessible to the script.

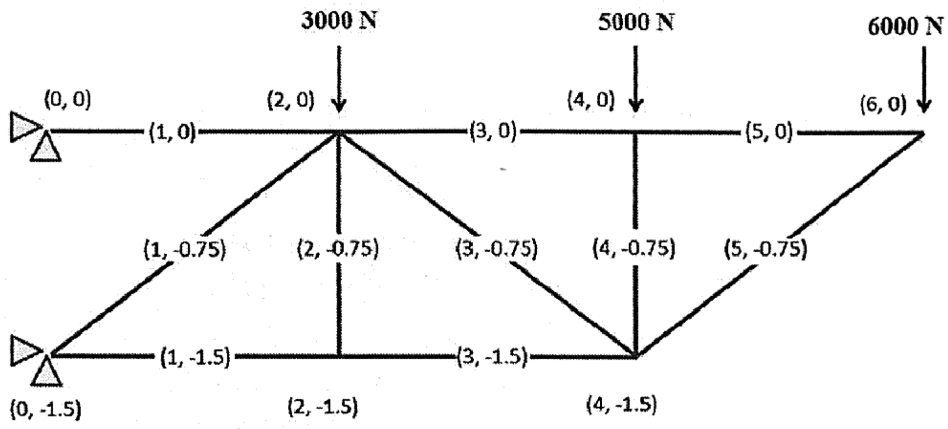
```
trussSection = trussModel.TrussSection(name='Truss Section',
                                       material='AISI 1005 Steel',
                                       area=3.14E-4)
```

This statement creates a **TrussSection** object using the **TrussSection()** method. The **TrussSection** object is derived from the **Section** object which is defined in the section module. The first parameter given to the method is a String for the name, which is used as the repository key. The second parameter is the material, which has been defined. Note that this material parameter must be a String, it cannot be a **material** object. That means we cannot say *material=trussMaterial* even though we had defined the **trussMaterial** variable earlier. 'AISI1005 Steel' on the other hand is a String, and it is the key assigned to that material in the **materials** repository. The third argument, area, is an optional one. It is a Float specifying the cross-sectional area of the truss members. Since our truss members have a radius of 1 cm (or 0.01 m), their cross-sectional area is 0.000314 m^2.

```
edges_for_section_assignment = trussPart.edges.findAt(((1.0, 0.0, 0.0), ),
                                                       ((3.0, 0.0, 0.0), ),
                                                       ((5.0, 0.0, 0.0), ),
                                                       ((1.0, -1.5, 0.0), ),
                                                       ((3.0, -1.5, 0.0), ),
                                                       ((1.0, -0.75, 0.0), ),
                                                       ((3.0, -0.75, 0.0), ),
                                                       ((5.0, -0.75, 0.0), ),
                                                       ((2.0, -0.75, 0.0), ),
                                                       ((4.0, -0.75, 0.0), ))
```

This statement uses the **findAt()** method to find any objects in the **EdgeArray** (basically edges) at the specified points or at a distance of less than 1E-6 from them. **trussPart** is the part, **trussPart.edges** exposes the **EdgeArray**, and **trussPart.edges.findAt()** finds the **edge** in the **EdgeArray**.

The coordinates used were obtained by drawing a rough sketch and determining the midpoints of each of the truss members. They are displayed in the figure below. Note that the Z coordinate was added when using the **findAt()** method. Being a 2D object the Z coordinate is 0.0 for all points.



```
truss_region = regionToolset.Region(edges=edges_for_section_assignment)
```

This statement creates a **Region** object using the **Region()** method. The **Region()** method has no required arguments, only optional ones such as **elements, nodes, vertices, edges, faces, cells** and a few more listed in the documentation. We use the **edges** argument, and assign it the edges obtained in the previous statement, which are the member elements of the truss.

The **Region** object itself was discussed in section 4.3.5 of the Cantilever Beam example on page 73. Note how the method used to create the region in this example is different from that used in the Cantilever Beam example. With the beam, a 3D object, we created **beam_region** with the statement *beam_region=(beamPart.cells,)* With the truss, a 2D planar object, we instead use the **Region()** method and passing the edges as arguments.

```
trussPart.SectionAssignment(region=truss_region, sectionName='Truss Section')
```

This statement creates a **SectionAssignment** object using the **SectionAssignment()** method. It is almost identical to the one used in the Cantilever Beam example, section 4.3.5 on page 73.The first parameter is the **Region** object created in the previous statement, and the second parameter is the name we wish to give the section, which is also its key in the sections repository.

7.4.6 Create an assembly

The following block creates the assembly

```
# ---------------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
trussAssembly = trussModel.rootAssembly
trussInstance = trussAssembly.Instance(name='Truss Instance', part=trussPart,
                                                               dependent=ON)
```

These statements are almost identical to the ones used in the Cantilever Beam example. If you wish to refer back, they are explained in section 4.3.6 on page 74.

7.4.7 Create steps

The following block creates the steps

```
# ----------------------------------------------------------------------
# Create the step

import step

# Create a static general step
trussModel.StaticStep(name='Loading Step', previous='Initial',
                      description='Loads are applied to the truss in this step')
```

These statements are also the same as the ones used in the Cantilever Beam example. You may refer back to them in section 4.3.7 on page 75.

7.4.8 Create and define field output requests

The following block creates the field output requests

```
# ----------------------------------------------------------------------
# Create the field output request

# Change the name of field output request 'F-Output-1' to 'Selected Field Outputs'
trussModel.fieldOutputRequests.changeKey(fromName='F-Output-1',
                                         toName='Selected Field Outputs')

# Since F-Output-1 is applied at the 'Loading Step' step by default,
# 'Selected Field Outputs' will be too
# We only need to set the required variables
trussModel.fieldOutputRequests['Selected Field Outputs'].setValues(variables=('S',
                                                                   'U', 'RF', 'CF'))
```

You have seen these commands in the Cantilever Beam example, section 4.3.8 on page 76..

7.4.9 Create and define history output requests

No history output requests were defined in this simulation

```
# ----------------------------------------------------------------------
# Create the history output request
# We want the defaults so we'll leave this section blank
```

7.4.10 Apply loads

The following block applies the loads

```
# ------------------------------------------------------------------------
# Apply loads

# Concentrated load of 3000 N on first node
vertex_coords_for_first_force = (2.0, 0.0, 0.0)
vertex_for_first_force = trussInstance.vertices \
                                        .findAt((vertex_coords_for_first_force,))
trussModel.ConcentratedForce(name='Force1', createStepName='Loading Step',
                             region=(vertex_for_first_force,), cf2=-3000.0,
                             distributionType=UNIFORM)

# Concentrated load of 5000 N on second node
vertex_coords_for_second_force = (4.0, 0.0, 0.0)
vertex_for_second_force = trussInstance.vertices \
                                        .findAt((vertex_coords_for_second_force,))
trussModel.ConcentratedForce(name='Force2', createStepName='Loading Step',
                             region=(vertex_for_second_force,), cf2=-5000.0,
                             distributionType=UNIFORM)

# Concentrated load of 6000 N on third node
vertex_for_third_force = trussInstance.vertices.findAt(((6.0, 0.0, 0.0),))
trussModel.ConcentratedForce(name='Force3', createStepName='Loading Step',
                             region=(vertex_for_third_force,), cf2=-6000.0,
                             distributionType=UNIFORM)
```

```
vertex_coords_for_first_force = (2.0, 0.0, 0.0)
```

This statement assigns to a variable the coordinates of the node on which the 3000 N force is applied.

```
vertex_for_first_force = trussInstance.vertices \
                                        .findAt((vertex_coords_for_first_force,))
```

This statement uses the **findAt()** method to find any **Vertex** object that is at the specified coordinates or at a distance of less than 1E-6 from them. **trussInstance** is the part instance in the assembly, **trussInstance.vertices** exposes the **VertexArray**, and **trussInstance.vertices.findAt()** finds the **vertex** in the **VertexArray**.

```
trussModel.ConcentratedForce(name='Force1', createStepName='Loading Step',
                             region=(vertex_for_first_force,), cf2=-3000.0,
                             distributionType=UNIFORM)
```

This statement applies a concentrated force on the selected node. It uses the **ConcentratedForce()** method to create a **ConcentratedForce** object, which is derived from the **Load** object. The first argument is the **name** or repository key for which the String 'Force1' is given. The second argument is the name/key of the step in which the concentrated force will be applied. The third argument is required to be a Region object. However **vertex_for_first_force** is a **Vertex** object. So we put it in parenthesis and add a comma indicating a sequence, and it becomes a **Region** object. Hence *region=(vertex_for_first_force,)*. The forth argument **cf2** is the Y-component of the force. (**cf1** is X-component, and **cf3** is Z-component). It is set to -3000 with the negative sign indicating the force is downward (since +ve y is up). The fifth argument sets the distribution type to uniform using the SymbolicConstant **UNIFORM**.

The statements

```
# Concentrated load of 5000 N on second node
vertex_coords_for_second_force = (4.0, 0.0, 0.0)
vertex_for_second_force = trussInstance.vertices \
                                        .findAt((vertex_coords_for_second_force,))
trussModel.ConcentratedForce(name='Force2', createStepName='Loading Step',
                              region=(vertex_for_second_force,), cf2=-5000.0,
                              distributionType=UNIFORM)

# Concentrated load of 6000 N on third node
vertex_for_third_force = trussInstance.vertices.findAt(((6.0, 0.0, 0.0),))
trussModel.ConcentratedForce(name='Force3', createStepName='Loading Step',
                              region=(vertex_for_third_force,), cf2=-6000.0,
                              distributionType=UNIFORM)
```

are similar to the previous ones. They apply the 5000 N and 6000 N forces on the corresponding vertices.

7.4.11 Apply boundary conditions

The following block applies the boundary conditions

```
# -----------------------------------------------------------------------
# Apply boundary conditions

# Pin left end of upper member
vertex_coords_for_first_pin = (0.0, 0.0, 0.0)
vertex_for_first_pin = trussInstance.vertices \
                                        .findAt((vertex_coords_for_first_pin,))
trussModel.DisplacementBC(name='Pin1', createStepName='Initial',
                          region=(vertex_for_first_pin,),
                          u1=SET, u2=SET, ur3=UNSET,
                          amplitude=UNSET, distributionType=UNIFORM)
```

```
# Pin left end of lower member
vertex_coords_for_second_pin = (0.0, -1.5, 0.0)
vertex_for_second_pin = trussInstance.vertices \
                                        .findAt((vertex_coords_for_second_pin,))
trussModel.DisplacementBC(name='Pin2', createStepName='Initial',
                          region=(vertex_for_second_pin,),
                          u1=SET, u2=SET, ur3=UNSET,
                          amplitude=UNSET, distributionType=UNIFORM)
```

```
vertex_coords_for_first_pin = (0.0, 0.0, 0.0)
```

This statement assigns the coordinates of the first node to be pinned to a variable.

```
vertex_for_first_pin = trussInstance.vertices \
                                        .findAt((vertex_coords_for_first_pin,))
```

This statement uses the **findAt()** method to find the **Vertex** object and assign it to a variable.

```
trussModel.DisplacementBC(name='Pin1', createStepName='Initial',
                          region=(vertex_for_first_pin,),
                          u1=SET, u2=SET, ur3=UNSET,
                          amplitude=UNSET, distributionType=UNIFORM)
```

This statement creates a **DisplacementBC** object which stores the data for a displacement/rotation. The **DisplacementBC** object is derived from the **BoundaryCondition** object. The first required argument is a String for the name. The second is the name/key of the step in which the boundary condition is to be applied. In this case we apply it to the 'Initial' step. The third argument must be a **Region** object. Once again we convert the **Vertex** object **vertex_for_first_pin** into a **Region** object by adding the parenthesis and a comma. The remaining arguments are optional. Note however that even though **u1**, **u2**, **u3**, **ur1**, **ur2** and **ur3** are optional arguments, at least one of them must be specified. For **u1** and **u2** we use the **SymbolicConstant SET** thus preventing translation in the 1 and 2 directions (a.k.a. X and Y directions). For **u3** we use the SymbolicConstant UNSET which is self-explanatory, and is the default value. **ur1**, **ur2** and **ur3** are not specified, and will default to **UNSET**. Since no **amplitude** is used, it is set to **UNSET** and the **distributionType** is once again set to **UNIFORM** ensuring a uniform special distribution of the boundary condition in the applied region.

```
vertex_coords_for_second_pin = (0.0, -1.5, 0.0)
vertex_for_second_pin = trussInstance.vertices \
                                        .findAt((vertex_coords_for_second_pin,))
trussModel.DisplacementBC(name='Pin2', createStepName='Initial',
                          region=(vertex_for_second_pin,),
                                    u1=SET, u2=SET, ur3=UNSET,
                                    amplitude=UNSET, distributionType=UNIFORM)
```

These statements repeat the process to pin the second node.

Instead of using two sets of statements to create two boundary conditions, we could instead have combined them as

```
vertex_coords_for_first_pin = (0.0, 0.0, 0.0)
vertex_coords_for_second_pin = (0.0, -1.5, 0.0)
vertices_for_pins = trussInstance.vertices.findAt((vertex_coords_for_first_pin,),
                                                   (vertex_coords_for_second_pin,))
trussModel.DisplacementBC(name='Pins', createStepName='Initial',
                          region=(vertices_for_pins,),
                          u1=SET, u2=SET, ur3=UNSET, amplitude=UNSET,
                          distributionType=UNIFORM)
```

Which method you choose is a matter of personal preference.

7.4.12 Mesh

The following block creates the mesh

```
# -----------------------------------------------------------------------
# Create the mesh

import mesh

truss_mesh_region = truss_region
edges_for_meshing = edges_for_section_assignment

mesh_element_type=mesh.ElemType(elemCode=T2D2, elemLibrary=STANDARD)
trussPart.setElementType(regions=truss_mesh_region,
                         elemTypes=(mesh_element_type, ))
trussPart.seedEdgeByNumber(edges=edges_for_meshing, number=1)
trussPart.generateMesh()
```

```
import mesh
```

This statement makes the methods and attributes of the **mesh** module available to the script.

```
truss_mesh_region = truss_region
```

We have already created a **Region** object of the truss earlier in our script. It is stored in the variable **truss_region**. We are storing a copy of it in a new variable **truss_mesh_region** which will be used in a subsequent statement.

```
edges_for_meshing = edges_for_section_assignment
```

We have also already identified and stored the edges (members) of the truss earlier in our script in the **edges_for_section_assignment** variable. We are now storing a copy of it in a new variable **edges_for_meshing** which will be used in a subsequent statement.

```
mesh_element_type=mesh.ElemType(elemCode=T2D2, elemLibrary=STANDARD)
```

This statement creates an **ElementType** object using the **ElemType()** method. This method was described in the Cantilever Beam example, section 4.3.12, page 85.

```
trussPart.setElementType(regions=truss_mesh_region,
                         elemTypes=(mesh_element_type, ))
```

This statement uses the **setElementType()** method to set the element type of the mesh. It requires the regions to mesh to be provided as one of the parameters. The **truss_mesh_region** variable is used here. In addition it requires a sequence of **Elem Type** objects, hence we use **mesh_element_type**, and use parenthesis and a comma to convert it to a sequence.

```
trussPart.seedEdgeByNumber(edges=edges_for_meshing, number=1)
```

This statement uses the **seedEdgeByNumber()** method to seed the given edges uniformly based on the number of elements along the edges. The first parameter required is a sequence of **Edge** objects, which we have stored in the **edges_for_meshing** variable. The second required parameter is an integer specifying the number of elements along each edge. We set this to 1. Of course this is a very coarse mesh and therefore not a very accurate simulation, but this example is for demonstration purposes. Feel free to refine the mesh as an exercise.

```
trussPart.generateMesh()
```

This statement uses the **generateMesh()** method to generate the mesh on the truss. The **generateMesh()** method is defined in the **mesh** module which has been imported.

7.4.13 Create and run the job

The following code runs the job

```
# ----------------------------------------------------------------------
# Create and run the job

import job

# Create the job
mdb.Job(name='TrussAnalysisJob', model='Truss Structure', type=ANALYSIS,
        explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE,
        description='Analysis of truss under concentrated loads',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)

# Run the job
mdb.jobs['TrussAnalysisJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['TrussAnalysisJob'].waitForCompletion()

# End of run job
```

All of this should look familiar to you from the Cantilever Beam example. You may refer back to section 4.3.13, page 88 for a refresher on these job commands.

7.4.14 Post processing – setting the viewport

The following code begins the post processing

```
# ----------------------------------------------------------------------
# Post processing

import visualization

truss_Odb_Path = 'TrussAnalysisJob.odb'
odb_object = session.openOdb(name=truss_Odb_Path)

session.viewports['Viewport: 1'].setValues(displayedObject=odb_object)
session.viewports['Viewport: 1'].odbDisplay.display\
                                                      .setValues(plotState=(DEFORMED, ))
```

You have seen these statements used in the Cantilever Beam example. To refresh your memory refer back to section 4.3.14 on page 89.

7.4.15 Plot the deformed state and modify common options

The following post processing block plots the deformed state of the truss and enables node and element labels through the common options

```
# Plot the deformed state of the truss
truss_deformed_viewport = session.Viewport(name='Truss in Deformed State')
truss_deformed_viewport.setValues(displayedObject=odb_object)
truss_deformed_viewport.odbDisplay.display.setValues(plotState=(UNDEFORMED,
                                                                DEFORMED, ))
truss_deformed_viewport.odbDisplay.commonOptions.setValues(nodeLabels=ON)
truss_deformed_viewport.odbDisplay.commonOptions.setValues(elemLabels=ON)
truss_deformed_viewport.setValues(origin=(0.0, 0.0), width=250, height=160)
```

```
truss_deformed_viewport = session.Viewport(name='Truss in Deformed State')
truss_deformed_viewport.setValues(displayedObject=odb_object)
```

These 2 statements should look familiar to you. The first one creates a new **Viewport** object (a new window on your screen) called 'Truss in Deformed State'. The second statement assigns the output database of the simulation to the viewport.

```
truss_deformed_viewport.odbDisplay.display.setValues(plotState=(UNDEFORMED,
                                                                DEFORMED, ))
```

You have seen the **setValues()** method used in the Cantilever Beam example. The difference here is that two symbolic keywords **UNDEFORMED** and **DEFORMED** have been used together. This causes both to be displayed overlaid on one another in the viewport window.

```
truss_deformed_viewport.odbDisplay.commonOptions.setValues(nodeLabels=ON)
```

This statement is the equivalent of clicking on the Common Options tool in the viewport and checking off 'show node labels'. Notice how we have again used the **setValues()** method, just as in the last statement, but the arguments supplied to it are very different. The parameters of the **setValues()** method depend on the context you are using it in.

```
truss_deformed_viewport.odbDisplay.commonOptions.setValues(elemLabels=ON)
```

This statement is the equivalent of clicking on the Common Options tool in the viewport and checking off 'show element labels'.

```
truss_deformed_viewport.setValues(origin=(0.0, 0.0), width=250, height=160)
```

Once again we use the **setValues()** method on the **Viewport** object. This time we provide 3 optional arguments, the **origin** of the new viewport window, its **width** and its **height**.

7.4.16 Plot the field outputs

The following post processing block plots the field output variables

```
# Plot the output variable U (spatial displacements at nodes) as its Magnitude
# invariant
# This is the equivalent of going to Report > Field Output and choosing to
# output U with Invariant: Magnitude
truss_displacements_magnitude_viewport= session \
                        .Viewport(name='Truss Displacements at Nodes (Magnitude)')
truss_displacements_magnitude_viewport.setValues(displayedObject=odb_object)
truss_displacements_magnitude_viewport.odbDisplay \
                                    .setPrimaryVariable(variableLabel='U',
                                                        outputPosition=NODAL,
                                                        refinement=(INVARIANT,
                                                                    'Magnitude'))
truss_displacements_magnitude_viewport.odbDisplay.display \
                                    .setValues(plotState=(CONTOURS_ON_DEF, ))
truss_displacements_magnitude_viewport.setValues(width=250, height=160)
truss_displacements_magnitude_viewport.offset(20,-10)

# Plot the output variable U (spatial displacements at nodes) as its U1 component
# This is the equivalent of going to Report > Field Output and choosing to output
# U with Component: U1
truss_displacements_U1_viewport= session \
                    .Viewport(name='Truss Displacements at Nodes (U1 Component')
truss_displacements_U1_viewport.setValues(displayedObject=odb_object)
truss_displacements_U1_viewport.odbDisplay \
                            .setPrimaryVariable(variableLabel='U',
                                                outputPosition=NODAL,
                                                refinement=(COMPONENT, 'U1'))
truss_displacements_U1_viewport.odbDisplay.display \
                                    .setValues(plotState=(CONTOURS_ON_DEF, ))
truss_displacements_U1_viewport.setValues(width=250, height=160)
truss_displacements_U1_viewport.offset(40,-20)

session.viewports['Viewport: 1'].sendToBack()
```

```
truss_displacements_magnitude_viewport= session \
                        .Viewport(name='Truss Displacements at Nodes (Magnitude)')
truss_displacements_magnitude_viewport.setValues(displayedObject=odb_object)
```

You are very familiar by now with the above 2 statements. We are creating a new viewport window called 'Truss Displacements at Nodes (Magnitude)' and setting it to draw its data from the output database file.

```
truss_displacements_magnitude_viewport.odbDisplay \
                                        .setPrimaryVariable(variableLabel='U',
                                                            outputPosition=NODAL,
                                                            refinement=(INVARIANT,
                                                                        'Magnitude'))
```

The **setPrimaryVariable()** method is used, which specifies the field output variable for which to obtain results from the output database. The first required argument **variableLabel** is a String specifying the field output variable we wish to plot. The second required argument, **outputPosition** requires a SymbolicConstant specifying the position from which to obtain data. One of the possible values is **NODAL**, which indicates we are drawing the data from a node. The documentation lists other possible values. The third argument is an optional one called **refinement**. It is only required if a refinement is available for the specified **variableLabel**, which is the case here. It must be a sequence of a SymbolicConstant and a String. We set the SymbolicConstant to **INVARIANT** and the String to 'Magnitude'.

```
truss_displacements_magnitude_viewport.odbDisplay.display \
                                        .setValues(plotState=(CONTOURS_ON_DEF, ))
```

You once again see the **setValues()** method being used on the **Display** object. Previously we set the **plotState** variable to the SymbolicConstants **DEFORMED** or **UNDEFORMED** (or both). In this situation we are setting the plot state to **CONTOURS_ON_DEF** which tells Abaqus to display the deformed state with a color contour of the specified variable/quantity (ie, U) displayed on it.

```
truss_displacements_magnitude_viewport.setValues(width=250, height=160)
```

Once again we use the **setValues()** method on the viewport and provide the optional width and height arguments to set the dimensions of the window.

```
truss_displacements_magnitude_viewport.offset(20,-10)
```

The **offset()** method is used on the viewport to offset the location of this viewport window from its current location by the specified X and Y coordinates. The offsets are floats specified in millimeters. This is done so that our windows are not one on top of another. It is not necessary to do this, it's only done here for aesthetic purposes and to demonstrate the **offset()** method to you.

```
truss_displacements_U1_viewport= session \
                    .Viewport(name='Truss Displacements at Nodes (U1 Component')
truss_displacements_U1_viewport.setValues(displayedObject=odb_object)
```

```
truss_displacements_U1_viewport.odbDisplay \
                          .setPrimaryVariable(variableLabel='U',
                                              outputPosition=NODAL,
                                              refinement=(COMPONENT, 'U1'))
truss_displacements_U1_viewport.odbDisplay.display \
                                     .setValues(plotState=(CONTOURS_ON_DEF, ))
truss_displacements_U1_viewport.setValues(width=250, height=160)
truss_displacements_U1_viewport.offset(40,-20)
```

These statements repeat the process except this time the SymbolicConstant is set to **COMPONENT** and the String to 'U1' in order to display the X component of the displacement. Also the window has been offset by a different amount in order to reveal the previous two underlying windows.

```
session.viewports['Viewport: 1'].sendToBack()
```

This statement uses the **sendToBack()** method to ensure that the default viewport window Viewport:1, which is the biggest window since we have not resized it, is behind all the newly created ones. In Abaqus 6.10 it is not really necessary since the newer windows automatically appear over the older ones but it might be helpful in older or newer versions of the software.

7.5 Summary

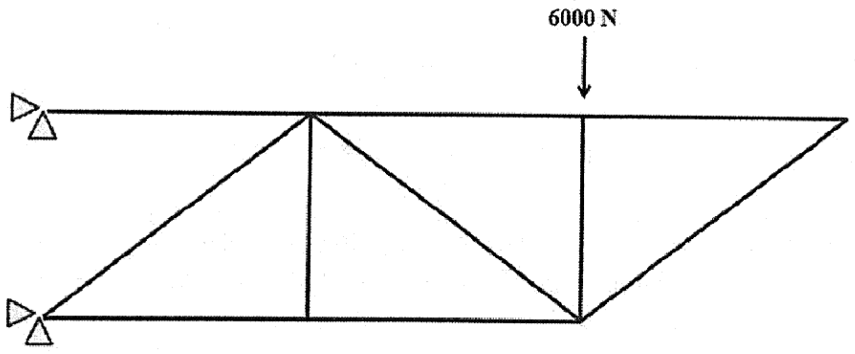
You just performed a 2D static truss analysis using a script. You are now familiar with the scripting commands most commonly used with such a simulation. Many of these commands will be used again in subsequent examples, just as ones from the Cantilever Beam example have been used here. There is no need to memorize these, you can always refer back to the examples in this book and copy and paste code suitably modifying it to fit your needs. Or you can use the replay file to assist you as well.

8

Explicit Analysis of a Dynamically Loaded Truss

8.1 Introduction

In this chapter we will perform a explicit analysis on a truss under dynamic loading. The problem is displayed in the figure. It is similar to the static general truss analysis of the previous chapter except that there is only one concentrated force and it is applied for 0.01 seconds.



In this exercise the following tasks will be demonstrated, first using the Abaqus/CAE, and then using a Python script.

- Create a part
- Assign materials
- Assign sections
- Create an Assembly
- Identify sets

- Create a dynamic, explicit step
- Request history outputs
- Assign loads
- Assign boundary conditions
- Create a mesh
- Create and submit a job
- Retrieve history outputs

The new topics covered are:

- Model / Preprocessing
  - Create sets in the assembly
  - Change step time period and tell Abaqus to include non-linear geometry effects
  - Use history output requests specifying the domain and frequency of history outputs
  - Specify point of application of loads using sets
- Results / Post-processing
  - Plot history outputs
  - Save XY data of history output plots
  - Write XY data to a report
  - Display Field Output as color contours

8.2 Procedure in GUI

You can perform the simulation in Abaqus/CAE by following the steps listed below. You can either read through these, or watch the video demonstrating the process on the book website. Note that much of the procedure is identical to that used to perform the static analysis of the truss in the previous example.

1. Rename **Model-1** to **Truss Structure**
   a. Right-click on Model-1 in Model Database
   b. Choose **Rename..**
   c. Change name to **Truss Structure**
2. Create the part
   a. Double-click on **Parts** in Model Database. **Create Part** window is displayed.

   b. Set **Name** to **Truss**
   c. Set **Modeling Space** to **2D Planar**
   d. Set **Type** to **Deformable**
   e. Set **Base Feature** to **Wire**
   f. Set **Approximate Size** to **10**
   g. Click **OK**. You will enter Sketcher mode.
3. Sketch the truss
   a. Use the **Create Lines: Connected** tool to draw the profile of the truss
   b. Split the lines using the **Split** tool
   c. Use **Add Constraints > Equal Length** tool to set the lengths of the required truss elements to be equal
   d. Use the **Add Dimension** tool to set the length of the horizontal elements to 2 m and the length of the vertical elements to 1.5 m.
   e. Click **Done** to exit the sketcher.
4. Create the material
   a. Double-click on **Materials** in the Model Database. **Edit Material** window is displayed
   b. Set **Name** to **AISI 1005 Steel**
   c. Select **General > Density**. Set **Mass Density** to **7872** (which is 7.872 g/cc)
   d. Select **Mechanical > Elasticity > Elastic**. Set **Young's Modulus** to **200E9** (which is 200 GPa) and **Poisson's Ratio** to **0.29.**
5. Assign sections
   a. Double-click on **Sections** in the Model Database. **Create Section** window is displayed
   b. Set **Name** to **Truss Section**
   c. Set **Category** to **Beam**
   d. Set **Type** to **Truss**
   e. Click **Continue...** The **Edit Section** window is displayed.
   f. In the **Basic** tab, set **Material** to **the AISI 1005 Steel** which was defined in the material creation step.
   g. Set **Cross-sectional Area** to **3.14E-4**
   h. Click **OK.**
6. Assign the section to the truss
   a. Expand the **Parts** container in the Model Database. Expand the part **Truss.**
   b. Double-click on **Section Assignments**

c. You see the message **Select the regions to be assigned a section** displayed below the viewport
d. Click and drag with the mouse to select the entire truss.
e. Click **Done**. The **Edit Section Assignment** window is displayed.
f. Set **Section** to **Truss Section.**
g. Click **OK**.
h. Click **Done.**

7. Create the Assembly
   a. Double-click on **Assembly** in the Model Database. The viewport changes to the **Assembly Module.**
   b. Expand the **Assembly** container.
   c. Double-click on **Instances.** The **Create Instance** window is displayed.
   d. Set **Parts** to **Truss**
   e. Set **Instance Type** to **Dependent (mesh on part)**
   f. Click **OK**.
8. Identify Sets
   a. Expand the **Assembly** container in the Model Database.
   b. Double-click on **Sets**. The **Create Set** window is displayed.
   c. Set **Name** to **force point set**
   d. Click **Continue...**
   e. You see the message **Select the geometry for the set** displayed below the viewport
   f. Select the node on which the force will be applied by clicking on it
   g. Click **Done.**
   h. Once again double-click on **Sets**. The **Create Set** window is displayed.
   i. Set **Name** to **end point set**
   j. Click **Continue...**
   k. You see the message **Select the geometry for the set** displayed below the viewport
   l. Select the node on the extreme right
   m. Click **Done**
9. Create Steps
   a. Double-click on **Steps** in the Model Database. The **Create Step** window is displayed.
   b. Set **Name** to **Loading Step**
   c. Set **Insert New Step After** to **Initial**

d. Set **Procedure Type** to **General >Dynamic, Explicit**
e. Click **Continue..** The **Edit Step** window is displayed
f. In the **Basic** tab, set **Description** to **Loads are applied to the trussfor 0.01s in this step.**
g. Set **Time period** to **0.01**
h. Click **OK.**

10. Request History Outputs
    a. Expand **the History Output Requests** container in the Model Database
    b. Right-click on **H-Output-1** and choose Rename…
    c. Change the name to **Force Point Output**
    d. Double-click on **Force Point Output** in the Model Database. The **Edit History Output Request** window is displayed
    e. Set **Domain** to **Set.** A new dropdown list appears next to it.
    f. Choose **force point set** from this list
    g. Set **Frequency** to **Every n time increments.**
    h. Set **n:** to **1**
    i. Select the desired variables by checking them off in the **Output Variables** list. The variable we want is **UT (translations)** from the **Displacement/Velocity/Acceleration** group. Uncheck the rest. You will notice that the text box above the output variable list displays **UT**
    j. Click **OK**
    k. We need to create the second history output request. Double-click on **History Output Requests** in the Model Database. The **Create History** window is displayed
    l. Set **Name** to **End Point Output**
    m. Set **Step** to **Loading Step**
    n. Click **Continue…** The **Edit History Output Request** window is displayed
    o. Set **Domain** to **Set.** A new dropdown list appears next to it.
    p. Choose **end point set** from this list.
    q. Set **Frequency** to **Every n time increments**
    r. Set **n:** to **1**
    s. Select the desired variables by checking them off in the **Output Variables** list. The variable we want is **UT (translations)** from the **Displacement/Velocity/Acceleration** group. Uncheck the rest. You will notice that the text box above the output variable list displays **UT**
    t. Click **OK**

11. Assign Loads
    a. Double-click on **Loads** in the Model Database. The **Create Load** window is displayed
    b. Set **Name** to **ForcePulse**
    c. Set **Step** to **Loading Step**
    d. Set **Category** to **Mechanical**
    e. Set **Type for Selected Step** to **Concentrated Force**
    f. Click **Continue...**
    g. You see the message **Select points for the load** displayed below the viewport
    h. We could select the required node by clicking on it. However we have already created a set for it. So click on the button **Sets** at the bottom of the viewport. The **Region Selection** window is displayed
    i. Choose **force point set** from the list. You may check off **Highlight selections in viewport** if you wish to see the selected node light up
    j. Click **Continue...**. The **Edit Load** window is displayed
    k. Set **CF2** to **-6000** to apply a 6000 N force in downward (negative Y) direction. Notice that Amplitude is set to **(Instantaneous)** although you cannot change it here.
    l. Click **OK**
    m. You will see the force displayed with an arrow in the viewport on the selected node
12. Apply boundary conditions
    a. Double-click on **BCs** in the Model Database. The **Create Boundary Condition** window is displayed
    b. Set **Name** to **Pin**
    c. Set **Step** to **Initial**
    d. Set **Category** to **Mechanical**
    e. Set **Types for Selected Step** to **Displacement/Rotation**
    f. Click **Continue...**
    g. Since you earlier selected vertices in the viewport by clicking the **Sets** button, you will now see the **Region Selection** window asking you to choose the set on which to apply boundary conditions. You also see the message **Select a region from the dialog** at the bottom of the viewport. However we do not wish to apply it on either **force point set** or **end point set**. Notice also the button **Select in Viewport** at the bottom right of the viewport. Click it.

You now see the message **Select regions for the boundary condition** displayed below the viewport

h. Select the two nodes on the extreme left. You can press the 'Shift' key on your keyboard to select both at the same time.
i. Click **Done**. The **Edit Boundary Condition** window is displayed.
j. Check off **U1** and **U2**. This will create a pin joint which does not allow translation but permits rotation.
k. Click **OK**.

13. Create the mesh
    a. Expand the **Parts** container in the Model Database.
    b. Expand **Truss**
    c. Double-click on **Mesh (Empty)**. The viewport window changes to the **Mesh module** and the tools in the toolbar are now meshing tools.
    d. Using the menu bar click on **Mesh > Element Type ...**
    e. You see the message **Select the regions to be assigned element types** displayed below the viewport
    f. Click and drag using your mouse to select the entire truss.
    g. Click **Done.** The **Element Type** window is displayed.
    h. Set **Element Library** to **Standard**
    i. Set **Geometric Order** to **Linear**
    j. Set **Family** to **Truss**
    k. You will notice the message **T2D2: A 2-node linear 2-D truss**
    l. Click **OK**
    m. Click **Done**
    n. Using the menu bar lick on **Seed > Edge by Number**
    o. You see the message **Select the regions to be assigned local seeds** displayed below the viewport
    p. Click and drag using your mouse to select the entire truss
    q. Click **Done.**
    r. You see the prompt **Number of elements along the edges** displayed below the viewport.
    s. Set it to 1 and press the 'Enter' key on your keyboard
    t. Click **Done**
    u. Using the menu bar lick on **Mesh > Part**
    v. You see the prompt **OK to mesh the part?** displayed below the viewport
    w. Click **Yes**

14. Create and submit the job
    a. Double-click on **Jobs** in the Model Database. The **Create Job** window is displayed
    b. Set **Name** to **TrussExplicitAnalysisJob**
    c. Set **Source** to **Model**
    d. Select **Truss Structure** (it is the only option displayed)
    e. Click **Continue..** The **Edit Job** window is displayed
    f. Set **Description** to **Analysis of truss under a pulse load**
    g. Set **Job Type** to **Full Analysis**.
    h. Leave all other options at defaults
    i. Click **OK**
    j. Expand the **Jobs** container in the Model Database
    k. Right-click on **TrussExplicitAnalysisJob** and choose **Submit**. This will run the simulation. You will see the following messages in the message window:
    The job input file "TrussExplicitAnalysisJob.inp" has been submitted for analysis.
    Job TrussExplicitAnalysisJob: Analysis Input File Processor completed successfully
    Job TrussExplicitAnalysisJob: Abaqus/Standard completed successfully
    Job TrussExplicitAnalysisJob completed successfully
15. Plot History Outputs
    a. Using the menu bar click on **Result >History Output...** The **History Output** window is displayed.
    b. In the **Output Variable** list select **Spatial displacement: U2 at Node 4 in NSET END POINT SET**
    c. Click the **Plot** button. A plot of the vertical displacement of the node at the extreme right of the truss is displayed in the viewport.
    d. Click the **Save As...** button. The **Save XY Data As** window is displayed.
    e. Set **Name** to **Data for end point**
    f. Click **OK**
    g. In the **Output Variable** list select **Spatial displacement: U2 at Node 4 in NSET FORCE POINT SET**
    h. Click the **Plot** button. A plot of the vertical displacement of the node at which the force was applied is displayed in the viewport.
    i. Click the **Save As...** button. The **Save XY Data As** window is displayed.
    j. Set **Name** to **Data for force point**

k. Click the **Dismiss** button
l. Using the menu bar click on **Report>XY...** The **Report XY Data** window is displayed
m. In the **XY Data** tab, make sure **Select from:** is set to **All XY data**. **Data for end point** and **Data for force point** should be displayed in the list. However sometimes due to a bug in Abaqus the list may appear empty and needs to be refreshed. To remedy this change **Select from:** to **XY plot in current view** and then back to **All XY data**. You should now see our XY data sets in the list.
n. Click **Data for end point** to make sure it is selected.
o. Click on the **Setup** tab.
p. In the **File** section, set **Name** to **end_point_xydata_output.txt.**
q. Uncheck **Append to file.**
r. In the **Data** section, for **Write:** check **XY data, Columns totals** and **Column min/max**
s. Switch back to **XY Data** tab
t. Make sure **Data for end point** is selected.
u. Click **Apply**. The file **end_point_xydata_output.txt** will be written to your Abaqus working directory.
v. Click **Data for force point** to make sure it is selected.
w. Click on the **Setup** tab.
x. In the **File** section, set **Name** to **force_point_xydata_output.txt.**
y. Uncheck **Append to file.**
z. In the **Data** section, for **Write:** check **XY data, Columns totals** and **Column min/max**
aa. Switch back to **XY Data** tab
bb. Make sure **Data for end point** is selected.
cc. Click **Apply**. The file **force_point_xydata_output.txt** will be written to your Abaqus working directory.
dd. Click **Cancel** to close the **Report XY Data** window.

8.3 Python Script

The following Python script replicates the above procedure for the dynamic explicit analysis of the truss. You can find it in the source code accompanying the book in **truss_dynamic.py**. You can run it by opening a new model in Abaqus/CAE (**File > New**

Model Database > With Standard/Explicit Model) and running it with **File > Run Script…**

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)


# -------------------------------------------------------------------------
# Create the model


mdb.models.changeKey(fromName='Model-1', toName='Truss Structure')
trussModel = mdb.models['Truss Structure']

# -------------------------------------------------------------------------
# Create the part

import sketch
import part

trussSketch = trussModel.ConstrainedSketch(name='2D Truss Sketch', sheetSize=10.0)
trussSketch.Line(point1=(0, 0), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, 0))
trussSketch.Line(point1=(4, 0), point2=(6, 0))
trussSketch.Line(point1=(0, -1.5), point2=(2,-1.5))
trussSketch.Line(point1=(2, -1.5), point2=(4,-1.5))
trussSketch.Line(point1=(0, -1.5), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, -1.5))
trussSketch.Line(point1=(4, -1.5), point2=(6, 0))
trussSketch.Line(point1=(2, 0), point2=(2, -1.5))
trussSketch.Line(point1=(4, 0), point2=(4, -1.5))

trussPart = trussModel.Part(name='Truss', dimensionality=TWO_D_PLANAR,
                                          type=DEFORMABLE_BODY)
trussPart.BaseWire(sketch=trussSketch)

# -------------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs modulus and
# poissons ratio
trussMaterial = trussModel.Material(name='AISI 1005 Steel')
trussMaterial.Density(table=((7872, ),       ))
trussMaterial.Elastic(table=((200E9, 0.29), ))

# -------------------------------------------------------------------------
# Create a section and assign the truss to it
import section

trussSection = trussModel.TrussSection(name='Truss Section',
                                       material='AISI 1005 Steel',
                                       area=3.14E-4)

edges_for_section_assignment = trussPart.edges.findAt(((1.0, 0.0, 0.0), ),
                                                      ((3.0, 0.0, 0.0), ),
                                                      ((5.0, 0.0, 0.0), ),
                                                      ((1.0, -1.5, 0.0), ),
                                                      ((3.0, -1.5, 0.0), ),
                                                      ((1.0, -0.75, 0.0), ),
                                                      ((3.0, -0.75, 0.0), ),
                                                      ((5.0, -0.75, 0.0), ),
                                                      ((2.0, -0.75, 0.0), ),
                                                      ((4.0, -0.75, 0.0), ))

truss_region = regionToolset.Region(edges=edges_for_section_assignment)
trussPart.SectionAssignment(region=truss_region, sectionName='Truss Section')

# -----------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
trussAssembly = trussModel.rootAssembly
trussInstance = trussAssembly.Instance(name='Truss Instance', part=trussPart,
                                                              dependent=ON)

# -----------------------------------------------------------------------
# Create sets

# Create set for load point
vertex_coords_for_force = (4.0, 0.0, 0.0)
vertex_for_force = trussInstance.vertices.findAt((vertex_coords_for_force,))
trussAssembly.Set(vertices=vertex_for_force, name='force point set')

# Create set for end point
vertex_coords_for_end = trussInstance.vertices.findAt(((6.0, 0.0, 0.0),))
trussAssembly.Set(vertices=vertex_coords_for_end, name='end point set')

# -----------------------------------------------------------------------
# Create the step

import step

# Create a dynamic explicit step
trussModel.ExplicitDynamicsStep(name='Loading Step', previous='Initial',
           description='Loads are applied to the truss for 0.01s in this step',
           timePeriod=0.01)
```

```
# ---------------------------------------------------------------------------
# Create the history output request

force_point_region = trussAssembly.sets['force point set']
trussModel.historyOutputRequests.changeKey(fromName='H-Output-1',
                                            toName='Force point output')
trussModel.historyOutputRequests['Force point output'] \
                             .setValues(variables=('UT',), frequency=1,
                             region=force_point_region, sectionPoints=DEFAULT,
                             rebar=EXCLUDE)

end_point_region = trussAssembly.sets['end point set']
trussModel.HistoryOutputRequest(name='End point output',
                                 createStepName='Loading Step',
                                 variables=('UT',),  frequency=1,
                                 region=end_point_region, sectionPoints=DEFAULT,
                                 rebar=EXCLUDE)

# ---------------------------------------------------------------------------
# Apply loads

# Concentrated load of 6000 N on second node

# We aleady have the vertex for force from the assembly step so we use that
trussModel.ConcentratedForce(name='ForcePulse', createStepName='Loading Step',
                              region=(vertex_for_force,), cf2=-6000.0,
                              distributionType=UNIFORM, field='', localCsys=None)

# ---------------------------------------------------------------------------
# Apply boundary conditions

# Pin left end of upper beam
vertex_coords_for_first_pin = (0.0, 0.0, 0.0)
vertex_for_first_pin = trussInstance.vertices \
                                    .findAt((vertex_coords_for_first_pin,))
trussModel.DisplacementBC(name='Pin1', createStepName='Initial',
                           region=(vertex_for_first_pin,),
                           u1=SET, u2=SET, ur3=UNSET,
                           amplitude=UNSET, distributionType=UNIFORM)

# Pin left end of lower beam
vertex_coords_for_second_pin = (0.0, -1.5, 0.0)
vertex_for_second_pin = trussInstance.vertices \
                                    .findAt((vertex_coords_for_second_pin,))
trussModel.DisplacementBC(name='Pin2', createStepName='Initial',
                           region=(vertex_for_second_pin,),
                           u1=SET, u2=SET, ur3=UNSET,
                           amplitude=UNSET, distributionType=UNIFORM)

#ALTERNATIVE METHOD
#vertex_coords_for_first_pin = (0.0, 0.0, 0.0)
```

```
#vertex_coords_for_second_pin = (0.0, -1.5, 0.0)
#vertices_for_pins = trussInstance.vertices \
#           .findAt((vertex_coords_for_first_pin,),
#                   (vertex_coords_for_second_pin,))
#trussModel.DisplacementBC(name='Pins', createStepName='Initial',
#                          region=(vertices_for_pins,),
#                          u1=SET, u2=SET, ur3=UNSET,
#                          amplitude=UNSET, #distributionType=UNIFORM)

# ----------------------------------------------------------------------
# Create the mesh

import mesh

truss_mesh_region = truss_region
edges_for_meshing = edges_for_section_assignment

mesh_element_type=mesh.ElemType(elemCode=T2D2, elemLibrary=STANDARD)
trussPart.setElementType(regions=truss_mesh_region,
                         elemTypes=(mesh_element_type, ))
trussPart.seedEdgeByNumber(edges=edges_for_meshing, number=1)
trussPart.generateMesh()

# ----------------------------------------------------------------------
# Create and run the job

import job

# Create the job
mdb.Job(name='TrussExplicitAnalysisJob', model='Truss Structure', type=ANALYSIS,
        explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE,
        description='Analysis of truss under a pulse load',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)

# Run the job
mdb.jobs['TrussExplicitAnalysisJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['TrussExplicitAnalysisJob'].waitForCompletion()

# End of run job


# ======================================================================
# ----------------------------------------------------------------------
# Post processing
# ----------------------------------------------------------------------
# ======================================================================
import odbAccess
import visualization

truss_Odb_Path = 'TrussExplicitAnalysisJob.odb'
odb_object = session.openOdb(name=truss_Odb_Path)

session.viewports['Viewport: 1'].setValues(displayedObject=odb_object)
session.viewports['Viewport: 1'].odbDisplay.display \
                                            .setValues(plotState=(DEFORMED, ))

#---------------------------------------------------------------------------
# Plot the deformed state of the truss

truss_deformed_viewport = session.Viewport(name='Truss in Deformed State')
truss_deformed_viewport.setValues(displayedObject=odb_object)
truss_deformed_viewport.odbDisplay.display.setValues(plotState=(UNDEFORMED,
                                                                DEFORMED, ))
truss_deformed_viewport.odbDisplay.commonOptions.setValues(nodeLabels=ON)
truss_deformed_viewport.odbDisplay.commonOptions.setValues(elemLabels=ON)
truss_deformed_viewport.setValues(origin=(0.0, 0.0), width=250, height=160)

# --------------------------------------------------------------------------
# Make XY plots of U2 displacement for force point and end point

# We need to find the variable names for the history variables
# Abaqus tends to give them names like "Spatial displacement: U2 at Node 2 in
# NSET FORCE POINT SET"
# So basically we will just search for variables with the letters 'U2' in them
# and save the variable names in an array called theoutputvariablename to use later

keyarray=session.odbData['TrussExplicitAnalysisJob.odb'].historyVariables.keys()

theoutputvariablename=[]
for x in keyarray:
    if (x.find('U2')>-1):
        theoutputvariablename.append(x)

# ---------------------------------
# a) XY plot and data output of U2 displacement for force point

xydata_force_pt=session.XYDataFromHistory(name = 'Data for force point',
                                      odb=odb_object,
                                      outputVariableName=theoutputvariablename[0],
                                      steps=('Loading Step', ), )
curve_force_pt = session.Curve(xyData=xydata_force_pt)

# Before plotting we make sure the name 'Plot of forcepoint' is not already in
# use, and delete it if it is, because Abaqus does not allow overwriting of plots
if 'Plot of forcepoint' in session.xyPlots.keys():
    del session.xyPlots['Plot of forcepoint']

xyplot_force_pt = session.XYPlot('Plot of forcepoint')
```

```
chartName = xyplot_force_pt.charts.keys()[0]
chart = xyplot_force_pt.charts[chartName]
chart.setValues(curvesToPlot=(curve_force_pt, ), )
xyplot_force_pt_viewport = session \
                          .Viewport(name='Displacement U2 plot of force point')
xyplot_force_pt_viewport.setValues(displayedObject=xyplot_force_pt)

# Output the xy data as a txt file
xydataobject_force_point = session.xyDataObjects['Data for force point']
session.xyReportOptions.setValues(totals=ON, minMax=ON)
session.writeXYReport(fileName='force_point_xydata_output.txt',
                      xyData=(xydataobject_force_point, ),appendMode=OFF)

# -------------------------------
# b) XY plot of U2 displacement for end point

xydata_end_pt=session.XYDataFromHistory(name = 'Data for end point',
                                        odb=odb_object,
                                        outputVariableName=theoutputvariablename[1],
                                        steps=('Loading Step', ), )
curve_end_pt = session.Curve(xyData=xydata_end_pt)

# Before plotting we make sure the name 'Plot of endpoint' is not already in use,
# and delete it if it is, because Abaqus does not allow overwriting of plots
if 'Plot of endpoint' in session.xyPlots.keys():
    del session.xyPlots['Plot of endpoint']

xyplot_end_pt = session.XYPlot('Plot of endpoint')
chartName = xyplot_end_pt.charts.keys()[0]
chart = xyplot_end_pt.charts[chartName]
chart.setValues(curvesToPlot=(curve_end_pt, ), )
xyplot_end_pt_viewport = session.Viewport(name='Displacement U2 plot of end point')
xyplot_end_pt_viewport.setValues(displayedObject=xyplot_end_pt)

# Output the xy data as a txt file
xydataobject_end_point = session.xyDataObjects['Data for end point']
session.xyReportOptions.setValues(totals=ON, minMax=ON)
session.writeXYReport(fileName='end_point_xydata_output.txt',
                      xyData=(xydataobject_end_point, ),appendMode=OFF)
```

8.3.1 Part, material, section and assembly blocks

Much of the code is identical or at least very similar to the static analysis of a truss performed in the previous chapter. The blocks dealing with creating the model, the part, the material, the section and the assembly are the same hence require no further explanation.

8.3.2 Creating sets

The new stuff begins in the block for creating sets.

```
# -----------------------------------------------------------------------
# Create sets

# Create set for load point
vertex_coords_for_force = (4.0, 0.0, 0.0)
vertex_for_force = trussInstance.vertices.findAt((vertex_coords_for_force,))
trussAssembly.Set(vertices=vertex_for_force, name='force point set')

# Create set for end point
vertex_coords_for_end = trussInstance.vertices.findAt(((6.0, 0.0, 0.0),))
trussAssembly.Set(vertices=vertex_coords_for_end, name='end point set')
```

The above block creates two sets - one for the node on which the load will be applied, and the other for the node furthest away from the wall.

```
vertex_coords_for_force = (4.0, 0.0, 0.0)
```

This statement assigns the coordinates of the node on which the force is to be applied to a variable for later use

```
vertex_for_force = trussInstance.vertices.findAt((vertex_coords_for_force,))
```

This statement uses the **findAt()** method to find the vertex at the given coordinates (or one within 1E-6 of it). The **Vertex** object is then stored in a variable **vertex_for_force** to be used later.

```
trussAssembly.Set(vertices=vertex_for_force, name='force point set')
```

This statement uses the **Set()** method to create a **Set** object in the assembly. Its first argument, **vertices**, is an optional argument. In place of **vertices** you might have used **nodes**, **elements**, **edges**, **faces**, **cells**, among other possible arguments (all of which are listed in the Abaqus documentation). Since we are using **vertices**, we provide a **Vertex** object **vertex_for_force**. The second argument, **name**, is a required parameter. It is a String which is the name of the set and its key in the repository.

```
vertex_coords_for_end = trussInstance.vertices.findAt(((6.0, 0.0, 0.0),))
trussAssembly.Set(vertices=vertex_coords_for_end, name='end point set')
```

These two statements repeat the process for the end point. However a slight change has been made to demonstrate another correct form of syntax. Notice that the coordinates are not first assigned to a variable, instead they are typed in directly as arguments to the **findAt()** method. Parenthesis must be included with the coordinates.

8.3.3 Creating steps

The following block creates the step

```
# ----------------------------------------------------------------------
# Create the step

import step

# Create a dynamic explicit step
trussModel.ExplicitDynamicsStep(name='Loading Step', previous='Initial',
              description='Loads are applied to the truss for 0.01s in this step',
              timePeriod=0.01)
```

```
import step
```

This statement imports the **step** module so the script can access its methods and properties

```
trussModel.ExplicitDynamicsStep(name='Loading Step', previous='Initial',
              description='Loads are applied to the truss for 0.01s in this step',
              timePeriod=0.01)
```

This statement creates a dynamic analysis step by creating a **ExplicitDynamicsStep** object using the **ExplicitDynamicsStep()** method. The **ExplicitDynamicsStep** object is derived from the **AnalysisStep** object which in turn is derived from the **Step()** object.

The first argument, **name**, is a required argument. It is a String specifying the repository key. The second argument, **previous**, is also a required parameter. It is a String specifying the name of the previous step. The third argument, **description**, is an optional one. It is a String describing what the step does or some other comments the author of the script wishes to type in. The fourth argument, **timePeriod**, is optional. It is a Float specifying the total time period of the step. We set it to 0.01 because we are applying the concentrated force in this step, and we only wish it to be applied for 0.01 seconds as a pulse.

8.3.4 Create and define history output requests

The following code block creates the history output requests

```
# -----------------------------------------------------------------------
# Create the history output request

force_point_region = trussAssembly.sets['force point set']
trussModel.historyOutputRequests.changeKey(fromName='H-Output-1',
                                           toName='Force point output')
trussModel.historyOutputRequests['Force point output'] \
                            .setValues(variables=('UT',), frequency=1,
                            region=force_point_region, sectionPoints=DEFAULT,
                            rebar=EXCLUDE)

end_point_region = trussAssembly.sets['end point set']
trussModel.HistoryOutputRequest(name='End point output',
                                createStepName='Loading Step',
                                variables=('UT',),  frequency=1,
                                region=end_point_region, sectionPoints=DEFAULT,
                                rebar=EXCLUDE)
```

```
force_point_region = trussAssembly.sets['force point set']
```

This statement assigns the set ‘force point set’ created previously to a variable. Note that this **Set** object is interchangeable with a **Region** object. Hence the variable has been named **force_point_region** and it will be used a few statements later as an argument where a **Region** object is expeccted.

```
trussModel.historyOutputRequests.changeKey(fromName='H-Output-1',
                                           toName='Force point output')
```

You’ve seen the **changeKey()** method used previously. Since the default history output request is ‘H-Output-1’, we rename it ‘Force point output’.

```
trussModel.historyOutputRequests['Force point output'] \
                            .setValues(variables=('UT',), frequency=1,
                            region=force_point_region, sectionPoints=DEFAULT,
                            rebar=EXCLUDE)
```

The **setValues()** method is used to tell Abaqus what data is desired in the history output. The first argument, **variables**, is a sequence of Strings indicating which quantities should be included in the history output (or the SymbolicConstants **PRESELECT** or **ALL**). The second argument, **frequency**, is an integer specifying the output frequency in increments. The default is 1. The third argument, **region**, is a **Region** object specifying the region from which output is requested. The fourth argument **sectionPoints** is a sequence of integers specifying the section points for which output is requested (the default is DEFAULT). The fifth argument, **rebar**, is a SymbolicConstant specifying whether output is requested for rebar. The default value is **EXCLUDE**.

```
end_point_region = trussAssembly.sets['end point set']
trussModel.HistoryOutputRequest(name='End point output',
                                createStepName='Loading Step',
                                variables=('UT',), frequency=1,
                                region=end_point_region, sectionPoints=DEFAULT,
                                rebar=EXCLUDE)
```

These statements repeat the process for the end point.

8.3.5 Apply loads

The following block applies the loads

```
# ------------------------------------------------------------------------
# Apply loads

# Concentrated load of 6000 N on second node

# We aleady have the vertex for force from the assembly step so we use that
trussModel.ConcentratedForce(name='ForcePulse', createStepName='Loading Step',
                             region=(vertex_for_force,), cf2=-6000.0,
                             distributionType=UNIFORM, field='', localCsys=None)
```

```
trussModel.ConcentratedForce(name='ForcePulse', createStepName='Loading Step',
                             region=(vertex_for_force,), cf2=-6000.0,
                             distributionType=UNIFORM, field='', localCsys=None)
```

The **ConcentratedForce()** method is used to apply a force. The syntax is identical to that used in the static truss analysis example, which was described in section 7.4.10 on page 136.

8.3.6 Boundary conditions, mesh, running the job and initial post processing

The blocks dealing with assigning boundary conditions, creating the mesh and creating and running the job are the same as those used in the previous example, hence they are not reexamined here again. The same is true for the initial post processing operation of plotting the deformed state of the truss.

8.3.7 XY plots of displacement

The following block generates XY plots of U2 (displacement in y direction) for the force application point and the end point of the truss.

```
# -----------------------------------------------------------------------
# Make XY plots of U2 displacement for force point and end point

# We need to find the variable names for the history variables
# Abaqus tends to give them names like "Spatial displacement: U2 at Node 2 in
# NSET FORCE POINT SET"
# So basically we will just search for variables with the letters 'U2' in them
# and save the variable names in an array called theoutputvariablename to use later

keyarray=session.odbData['TrussExplicitAnalysisJob.odb'].historyVariables.keys()

theoutputvariablename=[]
for x in keyarray:
    if (x.find('U2')>-1):
        theoutputvariablename.append(x)

# ---------------------------------
# a) XY plot and data output of U2 displacement for force point

xydata_force_pt=session.XYDataFromHistory(name = 'Data for force point',
                                      odb=odb_object,
                                      outputVariableName=theoutputvariablename[0],
                                      steps=('Loading Step', ), )
curve_force_pt = session.Curve(xyData=xydata_force_pt)

# Before plotting we make sure the name 'Plot of forcepoint' is not already in
# use, and delete it if it is, because Abaqus does not allow overwriting of plots
if 'Plot of forcepoint' in session.xyPlots.keys():
    del session.xyPlots['Plot of forcepoint']

xyplot_force_pt = session.XYPlot('Plot of forcepoint')
chartName = xyplot_force_pt.charts.keys()[0]
chart = xyplot_force_pt.charts[chartName]
chart.setValues(curvesToPlot=(curve_force_pt, ), )
xyplot_force_pt_viewport = session \
                           .Viewport(name='Displacement U2 plot of force point')
xyplot_force_pt_viewport.setValues(displayedObject=xyplot_force_pt)

# output the xy data as a txt file
xydataobject_force_point = session.xyDataObjects['Data for force point']
session.xyReportOptions.setValues(totals=ON, minMax=ON)
session.writeXYReport(fileName='force_point_xydata_output.txt',
                      xyData=(xydataobject_force_point, ),appendMode=OFF)

# ---------------------------------
# b) XY plot of U2 displacement for end point
```

```
xydata_end_pt=session.XYDataFromHistory(name = 'Data for end point',
                                    odb=odb_object,
                                    outputVariableName=theoutputvariablename[1],
                                    steps=('Loading Step', ), )
curve_end_pt = session.Curve(xyData=xydata_end_pt)

# Before plotting we make sure the name 'Plot of endpoint' is not already in use,
# and delete it if it is, because Abaqus does not allow overwriting of plots
if 'Plot of endpoint' in session.xyPlots.keys():
    del session.xyPlots['Plot of endpoint']

xyplot_end_pt = session.XYPlot('Plot of endpoint')
chartName = xyplot_end_pt.charts.keys()[0]
chart = xyplot_end_pt.charts[chartName]
chart.setValues(curvesToPlot=(curve_end_pt, ), )
xyplot_end_pt_viewport = session.Viewport(name='Displacement U2 plot of end point')
xyplot_end_pt_viewport.setValues(displayedObject=xyplot_end_pt)

# Output the xy data as a txt file
xydataobject_end_point = session.xyDataObjects['Data for end point']
session.xyReportOptions.setValues(totals=ON, minMax=ON)
session.writeXYReport(fileName='end_point_xydata_output.txt',
                      xyData=(xydataobject_end_point, ),appendMode=OFF)
```

In our history output requests, we have asked Abaqus to store U2 (displacement in 2-direction or y-direction) data. Abaqus tends to store all such data in dictionaries. From Python 101, you know that dictionaries are made up of key – value pairs. We will need to access the data from the output database, but in order to access the values we need to know the keys. The problem is we do not know the name of the keys off hand, so we must attempt to determine them.

```
keyarray=session.odbData['TrussExplicitAnalysisJob.odb'].historyVariables.keys()
```

This first statement accesses the names of all the keys and puts them in a variable **keyarray** which is automatically turned into a list variable.

```
theoutputvariablename=[]
```

Since we have two history output requests, we can the keys for both of them in an array. We therefore initialize the list variable **theoutputvariablename**. We could of course have created two variables instead, but this technique can also be used if you wished to plot multiple history output data, hence it is one you can reuse in your own scripts.

```
for x in keyarray:
    if (x.find('U2')>-1):
        theoutputvariablename.append(x)
```

When storing the history variables in its repository, Abaqus tends to give them names such as "Spatial displacement: U2 at Node 2 in NSET FORCE POINT SET". The node number will change with each point (it is 2 in this case), and so will the name of the set. Hence instead of trying to determine the entire text of the key, it makes sense to just search for keys with "U2" in them.

We iterate through all the keys which we have stored in the **keyarray** variable using a for-loop. We then use an if-statement to test for the presence of U2 in the key. For this we use the **find()** method. The **find()** method operates on Strings. It accepts a subString (basically a String) as an argument, and returns the lowest index at which that String is found within the original String. If it is not found, it returns -1. Hence our if condition tests to see if **find()** returns anything greater than -1, which would indicate the presence of U2 in the key.

If U2 is present in the key, we store the entire key in the variable **theoutputvariablename** by using the **append()** method which adds its argument to the end of the list it is operating on. The **append()** method was discussed in section 3.4 on page 44.

```
xydata_force_pt=session.XYDataFromHistory(name = 'Data for force point',
                                          odb=odb_object,
                                          outputVariableName=theoutputvariablename[0],
                                          steps=('Loading Step', ), )
```

The **XYDataFromHistory()** method creates an **XYData** object by reading history data from an Odb object. The first argument supplied here, **name**, is optional. It is the repository key which will be associated with this **XYData** object. The second argument, **odb**, is required. It is an **Odb** object specifying the output database from which data will be read. The third argument, **outputVariableName** is a required parameter. It is a String, and it is the repository key of the output variable from which the X-Y data will be read. This is why we had to earlier find the keys of the output variables. Here we use the first one, hence the [0]. The fourth argument, **steps**, is also required. It is a sequence of Strings specifying the names of the steps from which data will be extracted. Since it is a sequence, we do not just write 'Loading Step' but instead ('Loading Step',) with parenthesis and a comma.

```
curve_force_pt = session.Curve(xyData=xydata_force_pt)
```

The **Curve()** method creates an **XYCurve** object from an **XYData** object. We created the **XYData** object in the previous statement, and we now need to make an **XYCurve** object from it which will soon be plotted on a chart.

```
if 'Plot of forcepoint' in session.xyPlots.keys():
    del session.xyPlots['Plot of forcepoint']
```

Our next step is to create an XY Plot. However Abaqus does not allow multiple plots to exist with the same name. If the user runs this script twice, one of these may exist from a previous run of this simulation. Hence an if-statement is used to check for this. The **keys()** method is used to obtain the keys of plots currently in the session repository with the statement **session.xyPlots.keys()**. We then search for our plot 'Plot of forcepoint' among these keys. Notice the keyword 'in'. This is interesting Python syntax that has endeared the language to some programmers. It does exactly what the code reads as: look to see if 'Plot of forcepoint' is among the xyplot keys in the current session.

If the condition returns true, the xyPlot is promptly deleted using the del keyword. Notice the syntax, we refer to the xyplot with **session.xyPlots['Plot of forcepoint']**

```
xyplot_force_pt = session.XYPlot('Plot of forcepoint')
```

The **XYPlot()** method creates an **XYPlot** object. The argument is a String specifying the name (repository key) of the new **XYPlot** object. The plot itself is currently blank.

```
chartName = xyplot_force_pt.charts.keys()[0]
chart = xyplot_force_pt.charts[chartName]
```

An **XYPlot** object can consist of multiple charts. Each of these charts will have a name or repository key with which one can refer to it through a script. The first one, by default, should be 'Chart-1'. However it is safest to obtain the name programmatically. We use the **keys()** method to obtain a list of the keys, and since we should only have one chart by default we refer to it with the index [0]. Hence the statement **xyplot_force_pt.charts.keys()[0]**. The key is assigned to the **chartName** variable. This variable is then used to assign the chart object to a variable 'chart'.

```
chart.setValues(curvesToPlot=(curve_force_pt, ), )
```

The **setValues()** method, when used with the chart object can be used to plot the chart. The **curvesToPlot** argument refers to the **curve_force_pt**, the curve we had created earlier using the **Curve()** method. However the argument must be a sequence, hence the syntax **(curve_force_pt,)**

```
xyplot_force_pt_viewport = session \
                            .Viewport(name='Displacement U2 plot of force point')
xyplot_force_pt_viewport.setValues(displayedObject=xyplot_force_pt)
```

We display the plot in the viewport using the now familiar methods **Viewport()** and **setValues()**.

We also wish to output the XY data as a text file

```
xydataobject_force_point = session.xyDataObjects['Data for force point']
```

We earlier created an **XYData** object with the name 'Data for force point' using the **XYDataFromHistory()** method. We now assign that object to another variable to make our code easier to read.

```
session.xyReportOptions.setValues(totals=ON, minMax=ON)
session.writeXYReport(fileName='force_point_xydata_output.txt',
                      xyData=(xydataobject_force_point, ),appendMode=OFF)
```

An **XYReportOptions** object stores the setting used by **writeXYReport()** method. The **XYReportOptions** object did not need to be created by us because Abaqus creates it the moment the **visualization** module is imported.

The **setValues()** method of the **XYReportOptions** object has no required arguments, but many options ones. The ones we use are **totals**, which specifies whether or not column totals are output, and **minMax**, which specifies whether the maximum and minimum values are output. Both are set using the boolean ON. The default value for both is OFF.

The **writeXYReport()** method writes an **XYData** object to an ASCII file. The first argument, **filename**, is required. It is a String specifying the name of the file to which the XY data will be written. The second argument, **xyData**, is also required. It is a sequence of **XYData** objects to be written to the output file. The third argument, **appendMode**, is optional. It accepts a Boolean value, and specifies whether or not the XY data will be appended to an existing file or placed in a new file.

The entire process is then repeated for the other node of the truss marked as ‘end point’.

```
xydata_end_pt=session.XYDataFromHistory(name = 'Data for end point',
                                        odb=odb_object,
                                        outputVariableName=theoutputvariablename[1],
                                        steps=('Loading Step', ), )
curve_end_pt = session.Curve(xyData=xydata_end_pt)

# Before plotting we make sure the name 'Plot of endpoint' is not already in use,
# and delete it if it is, because Abaqus does not allow overwriting of plots
if 'Plot of endpoint' in session.xyPlots.keys():
    del session.xyPlots['Plot of endpoint']

xyplot_end_pt = session.XYPlot('Plot of endpoint')
chartName = xyplot_end_pt.charts.keys()[0]
chart = xyplot_end_pt.charts[chartName]
chart.setValues(curvesToPlot=(curve_end_pt, ), )
xyplot_end_pt_viewport = session.Viewport(name='Displacement U2 plot of end point')
xyplot_end_pt_viewport.setValues(displayedObject=xyplot_end_pt)

# Output the xy data as a txt file
xydataobject_end_point = session.xyDataObjects['Data for end point']
session.xyReportOptions.setValues(totals=ON, minMax=ON)
session.writeXYReport(fileName='end_point_xydata_output.txt',
                      xyData=(xydataobject_end_point, ),appendMode=OFF)
```

8.4 Summary

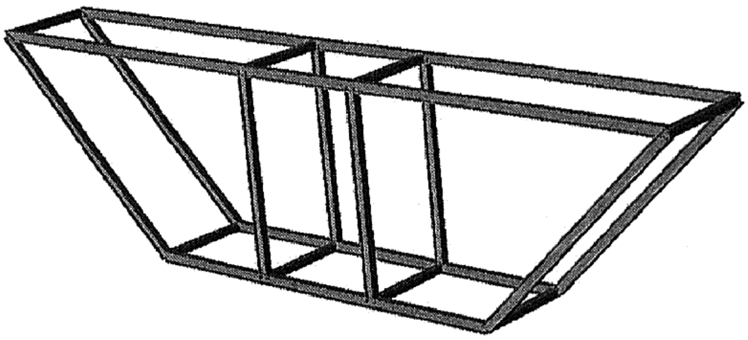
A few more concepts were covered in this chapter among which are creating sets, and post processing methods such as plotting XY data on a chart, and reporting it to an output file. We used some interesting tactics to discover the keys of the XY Data and latch onto it. These methods will likely be used by you in many scripts in the future.

9

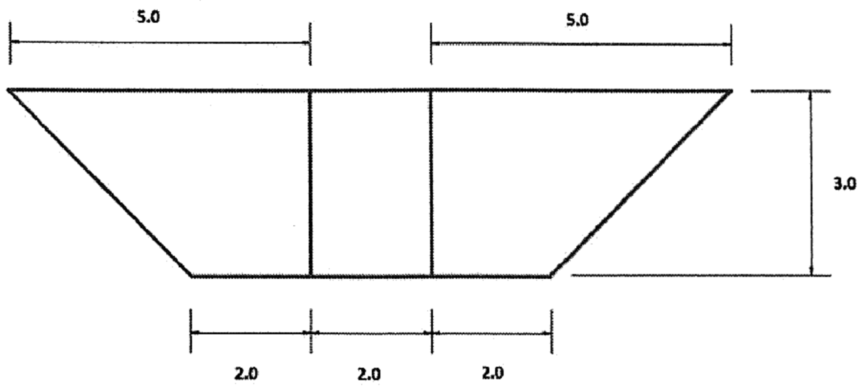
Analysis of a Frame of I-Beams

9.1 Introduction

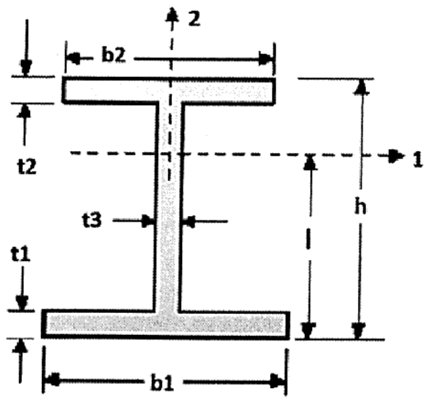
In this chapter we will perform an analysis on a frame made up of I-beams. The structure is displayed in the figure.



The dimensions of the beam frame are displayed in the following figure. All dimensions are in meters. In addition the distance between the two frames (ie, the length of the cross members) is 1.5 m.

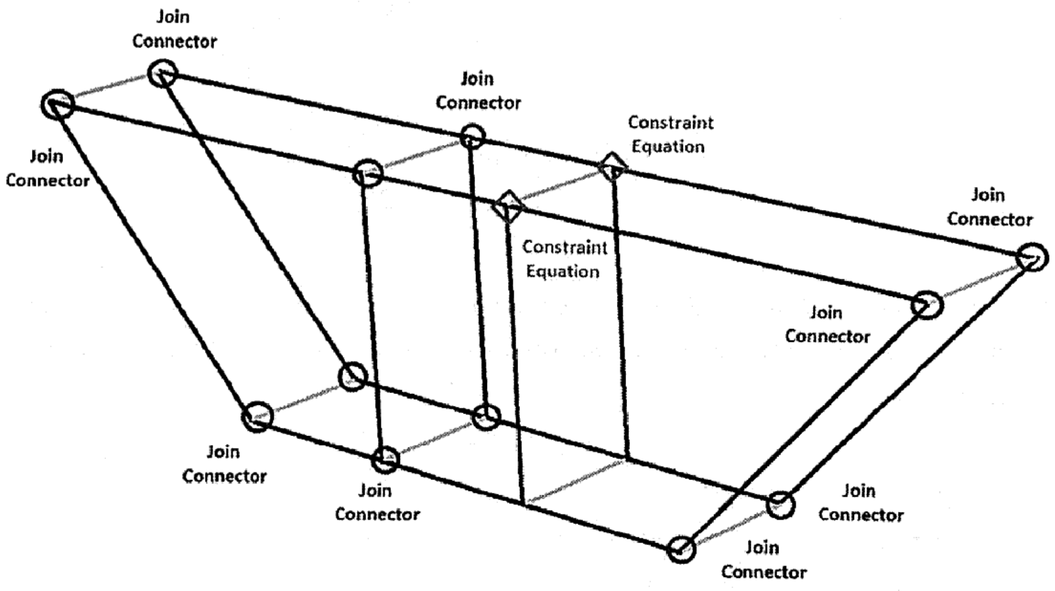


The beam profile dimensions are displayed in the figure.

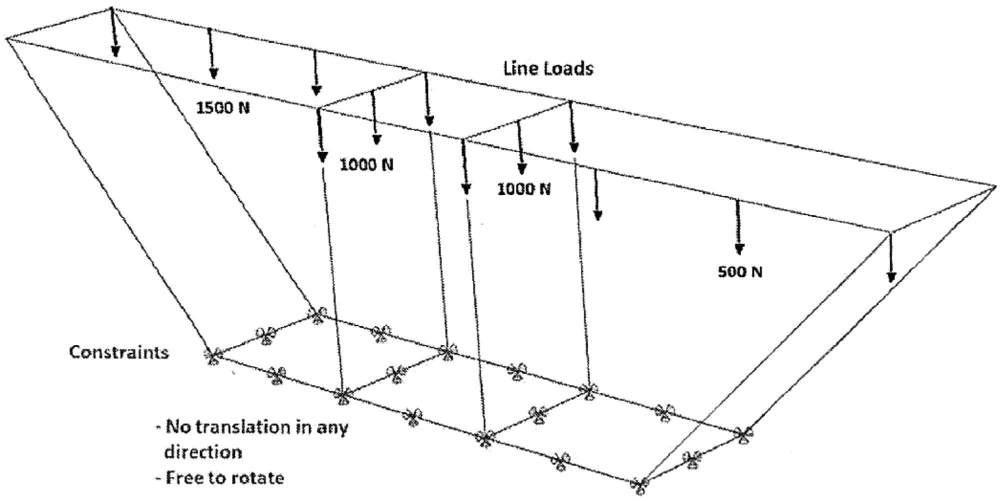


Dimension	Frame member	Crossbracing member
l:	0.075	0.06
h:	0.15	0.12
b1:	0.12	0.11
b2:	0.12	0.08
t1:	0.02	0.01
t2:	0.02	0.01
t3:	0.04	0.02

We will use both join connectors and constrain equations to create the pin joints between the frames and cross members in order to demonstrate how you can use both methods.



The loads and boundary conditions are displayed in the figure.



In this exercise the following tasks will be performed first using Abaqus/CAE, and then using a Python script.

- Create a part
- Create and offset datum points and datum planes
- Assign materials
- Create profiles
- Assign sections
- Set orientation
- Create an Assembly
- Create connector sections
- Perform connector assignments
- Identify sets
- Assign constraints with constraint equations
- Create a step
- Assign loads
- Assign boundary conditions
- Create a mesh
- Create and submit a job

The following new topics are covered in this example:

- Model / Preprocessing
    - Create a part starting with a reference point
    - Create datum planes and datum lines
    - Create beam elements in 3D using the 'Create Lines: Connected' and 'Create Wire: Point to Point' tools
    - Create beam sections and define beam profile geometry
    - Orient beams and render the orientations in the viewport
    - Use connectors (wire features + connector sections) to create joints
    - Use constraint equations to simulate joints
    - Use line loads

9.2 Procedure in GUI

You can perform the simulation in Abaqus/CAE by following the steps listed below. You can either read through these, or watch the video demonstrating the process on the book website.

1. Rename **Model-1** to **Beam Frame**
    a. Right-click on Model-1 in Model Database
    b. Choose **Rename..**
    c. Change name to **Beam Frame**
2. Create the frame part
    a. Double-click on **Parts** in Model Database. **Create Part** window is displayed.
    b. Set **Name** to **Frame**
    c. Set **Modeling Space** to **3D**
    d. Set **Type** to **Deformable**
    e. Set **Base Feature** to **Point**
    f. Set **Type** to **Coordinates**
    g. Set **Approximate Size** to **20**
    h. Click **Continue..** You see the message **Enter the coordinates of the point** displayed below the viewport
    i. Type in **0.0,0.0,0.0** and hit the "Enter" key on your keyboard. A reference point marked with an **X** and the letters **RP** appears in the viewport.

j. Click on the **Create Datum Point: Offset From Point** tool in the toolbar. You see the message **Select a point from which to offset** displayed below the viewport.
k. Click on the reference point. You see **Offset (X, Y, Z)** displayed below the viewport. Type in **13.0,0.0,0.0** and hit the “Enter” key on your keyboard.
l. Click on **Autofit view** on the **View Manipulation** toolbar to see the point
m. You again see the message **Select a point from which to offset** displayed below the viewport.Again click on the reference point.
n. You see **Offset (X, Y, Z)** displayed below the viewport. Type in **4.0,-3.0,0.0** and hit the “Enter” key on your keyboard.
o. You again see the message **Select a point from which to offset** displayed below the viewport.Again click on the reference point.
p. You see **Offset (X, Y, Z)** displayed below the viewport. Type in **1.0,0.0,0.0** and hit the “Enter” key on your keyboard.
q. Click on the **Create Datum Plane: 3 Points** tool in the toolbar. You see the message **Select the first point in the datum plane** displayed below the viewport
r. Click on the point on the left. You see the message **Select the second point in the datum plane** displayed below the viewport.
s. Click on the point in the middle which is lower than the other two. You see the message **Select the third point in the datum plane** displayed below the viewport.
t. Click on the point on the right.
u. Click on **Autofit view** on the **View Manipulation** toolbar to see the plane.
v. Click on the **Create Datum Axis: Principal Axis** tool in the toolbar.
w. From the Principal axis choice buttons at the bottom of the viewport click on Y-Axis. The Y-axis is displayed in the viewport
x. Click on the **Create Wire: Planar** tool in the toolbar. You see the message **Select a plane for the planar wire** displayed below the viewport
y. Click on the edge of the plane in the viewport. You see the message **Select an edge or axis that will appear vertical and on the right** displayed below the viewport.
z. Change the selection in the dropdown list to **vertical and on the left**
aa. Click on the datum axis (the Y-axis) in the viewport. You enter the sketcher.
bb. Use the **Create Lines:Connected** tool to draw the profile of one frame
cc. Split the lines using the **Split** tool

dd. Use the **Add Dimension** tool to set the required dimensions

ee. Click **Done** to exit the sketcher

ff. Click on the **Create Datum Plane: Offset from Plane** tool in the toolbar. You see the message **Select a plane from which to offset** displayed below the viewport

gg. Click on the plane.You see the message **How do you want to specify the offset?** displayed below the viewport.

hh. Click on Enter Value

ii. You see the message **Arrow shows the offset direction** displayed below the viewport. You may wish to use the Rotate View tool from the View Manipulation toolbar to rotate the view so you can better see which way the arrow is pointed. It should be pointed along the positive Z axis (out of the screen, towards you)

jj. Click OK

kk. For Offset type in 1.5 and hit the "Enter" key on your keyboard. A second plane appears 1.5 m in front of the original. We can now draw the second frame on this.

ll. Click on the **Create Datum Point: Enter Coordinates** tool in the toolbar. You see the prompt **Coordinates for datum point (X, Y, Z)** displayed below the viewport

mm. Type in 1.0,0.0,1.5 and hit the "Enter" key on your keyboard

nn. Then type in 13.0,0.0,1.5 and hit the "Enter" key on your keyboard

oo. Then type in 4.0,-3.0,1.5 and hit the "Enter" key on your keyboard

pp. Click on the **Create Wire: Planar** tool in the toolbar. You see the message **Select a plane for the planar view** displayed below the viewport

qq. Click on the new plane. You see the message **Select an edge or axis that will appear vertical and on the right** displayed below the viewport

rr. Change the selection in the dropdown list to **vertical and on the left**

ss. Once again click on the datum axis (the Y-axis) in the viewport. You enter the sketcher

tt. Use the **Create Lines:Connected** tool to draw the profile of the second frame

uu. Split the lines using the **Split** tool

vv. Use the **Add Dimension** tool to set the required dimensions

ww. Click **Done** to exit the sketcher

3. Create the cross bracing

a. Double-click on **Parts** in Model Database. **Create Part** window is displayed
b. Set **Name** to **CrossBracing**
c. Set **Modeling Space** to **3D**
d. Set **Type** to **Deformable**
e. Set **Base Feature** to **Point**
f. Set **Type** to **Coordinates**
g. Set **Approximate Size** to **20**
h. Click **Continue..** You see the message **Enter the coordinates of the point** displayed below the viewport
i. Type in **0.0,0.0,0.0** and hit the "Enter" key on your keyboard. A reference point marked with an **X** and the letters **RP** appears in the viewport
j. Click on the **Create Datum Point: Enter Coordinates** tool in the toolbar. You see the prompt **Coordinates for datum point (X, Y, Z)** displayed below the viewport
k. Type in **1.0,0.0,0.0** and hit the "Enter" key on your keyboard
l. Then type in **1.0,0.0,1.5** and hit the "Enter" key on your keyboard
m. Repeat for **1.0,0.0,0.0, 1.0,0.0,1.5, 4.0,-3.0,0.0, 4.0,-3.0,1.5, 6.0,0.0,0.0, 6.0,0.0,1.5, 6.0,-3.0,0.0, 6.0,-3.0,1.5, 8.0,0.0,0.0, 8.0,0.0,1.5, 8.0,-3.0,0.0, 8.0,-3.0,1.5, 10.0,-3.0,0.0, 10.0,-3.0,1.5, 13.0,0.0,0.0, 13.0,0.0,1.5**
n. Click on the **Create Wire: Point to Point** tool in the toolbar. The**Create Wire Feature** window is displayed
o. Set **Add method** to **Disjoint wires**
p. Click the **Add** button
q. You see the prompt **Select the first point** displayed below the viewport. You can either type in **1.0,0.0,0.0**or click on it with the mouse.
r. You see the prompt **Select the second point** displayed below the viewport. You can either type in **1.0,0.0,1.5** or click on it with the mouse. A line is drawn connecting the two points. You are once again prompted to **Select the first point**
s. Repeat the process till all the cross braces have been drawn
t. Click **Done**. All the datum points selected are filled into the table in the **Create Wire Feature** window.
u. Check **Create set of wires**
v. Click **OK.**

4. Create the material

   a. Double-click on **Materials** in the Model Database. **Edit Material** window is displayed
   b. Set **Name** to **AISI 1005 Steel**
   c. Select **General > Density**. Set **Mass Density** to **7872** (which is 7.872 g/cc)
   d. Select **Mechanical > Elasticity > Elastic**. Set **Young's Modulus** to **200E9** (which is 200 GPa) and **Poisson's Ratio** to **0.29.**
5. Assign Profiles
   a. Double-click on **Profiles** in the Model Database. **Create Profile** window is displayed
   b. Set **Name** to **FrameProfile**
   c. Set **Shape** to **I**
   d. Click **Continue..**
   e. The **EditProfile** window is displayed
   f. Set **l** to **0.075**
   g. Set **h** to **0.15**
   h. Set **b1** to **0.12**
   i. Set **b2** to **0.12**
   j. Set **t1** to **0.02**
   k. Set **t2** to **0.02**
   l. Set **t3** to **0.04**
   m. Click **OK**
   n. Double-click on **Profiles** in the Model Database. **Create Profile** window is displayed
   o. Set **Name** to Cross**Profile**
   p. Set **Shape** to **I**
   q. Click **Continue..**
   r. The **EditProfile** window is displayed
   s. Set **l** to **0.06**
   t. Set **h** to **0.12**
   u. Set **b1** to **0.11**
   v. Set **b2** to **0.08**
   w. Set **t1** to **0.01**
   x. Set **t2** to **0.01**
   y. Set **t3** to **0.02**
   z. Click **OK**
6. Assign sections

a. Double-click on **Sections** in the Model Database. **Create Section** window is displayed
b. Set **Name** to **Frame Section**
c. Set **Category** to **Beam**
d. Set **Type** to **Beam**
e. Click **Continue…** The **Edit Section** window is displayed.
f. Set **Section Integration** to **During Analysis**
g. Set **Profile name** to **FrameProfile** which we created earlier
h. In the **Basic** tab, set **Material** to **AISI 1005 Steel** which was defined in the create material step.
i. Leave**Section Poisson's ratio** atthe default of **0**
j. Click **OK**.
k. Again double-click on **Sections** in the Model Database. **Create Section** window is displayed
l. Set **Name** to **Cross Section**
m. Set **Category** to **Beam**
n. Set **Type** to **Beam**
o. Click **Continue…** The **Edit Section** window is displayed.
p. Set **Section Integration** to **During Analysis**
q. Set **Profile name** to **CrossProfile** which we created earlier
r. In the **Basic** tab, set **Material** to **AISI 1005 Steel** which was defined in the create material step.
s. Leave **Section Poisson's ratio** at the default of **0**
t. Click **OK**

7. Assign the sections to the frame and cross bracing
   a. Expand the **Parts** container in the Model Database. Expand the part **Frame.**
   b. Double-click on **Section Assignments**
   c. You see the message **Select the regions to be assigned a section** displayed below the viewport
   d. Click and drag with the mouse to select the entire frame (both sides).
   e. Click **Done**. The **Edit Section Assignment** window is displayed.
   f. Set **Section** to **Frame Section.**
   g. Click **OK.**
   h. Click **Done**.
   i. Expand the **Parts** container in the Model Database. Expand the part **CrossBracing.**

   j. Double-click on **Section Assignments**
   k. You see the message **Select the regions to be assigned a section** displayed below the viewport
   l. Click and drag with the mouse to select the entire frame (both sides).
   m. Click **Done.** The **Edit Section Assignment** window is displayed.
   n. Set **Section** to **Cross Section**.
   o. Click **OK**.
   p. Click **Done.**

8. Define Beam Orientations
   a. Change the **Module** (displayed above viewport) to **Property** if it isn't already the case using the dropdown menu.
   b. Using the menu bar click on **Assign>Beam Section Orientation...**
   c. You see the message **Select the regions to be assigned a beam section orientation** displayed below the viewport
   d. Click and drag with the mouse to select the entire crossbracing
   e. Click **Done**. You see the prompt **Enter an approximate n1 direction (tangent vectors are shown)** displayed below the viewport.
   f. Type in 1.0,0.0,0.0 and hit the "Enter" key on your keyboard. You notice orientation arrows have been displayed in the viewport. You see the prompt **Click OK to confirm input** displayed below the viwport.
   g. Click **OK.**
   h. By default you may not be able to see how the beams are oriented in the viewport. Using the menu bar click on **View>Part Display Options.** The **Part Display Options** window is displayed.
   i. In the **General** tab, in the **Idealizations** section, check **Render beam profiles.**
   j. Click **OK.**You now see the cross beam profiles rendered in the viewport.
   k. At the top of the viewport, ensure the **Module** is still set to **Property** and change the **Part** to **Frame** .
   l. Using the menu bar click on **Assign>Beam Section Orientation...**
   m. You see the message **Select the regions to be assigned a beam section orientation** displayed below the viewport
   n. Click and drag with the mouse to select the entire frame (both sides)
   o. Click **Done**. You see the prompt **Enter an approximate n1 direction (tangent vectors are shown)** displayed below the viewport.

p. Type in 0.0,0.0,1.0 and hit the “Enter” key on your keyboard. You notice orientation arrows have been displayed in the viewport. You see the prompt **Click OK to confirm input** displayed below the viwport.
q. Click **OK**
r. You now see the cross beam profiles rendered in the viewport. You can now disable rendering of beam profiles.Using the menu bar click on **View>Part Display Options.** The **Part Display Options** window is displayed. In the **General** tab, in the **Idealizations** section, uncheck **Render beam profiles.** Click **OK.**

9. Create the Assembly
   a. Double-click on **Assembly** in the Model Database. The viewport changes to the **Assembly Module.**
   b. Expand the **Assembly** container.
   c. Double-click on **Instances**. The **Create Instance** window is displayed.
   d. Set **Parts** to **Frame**
   e. Set **Instance Type** to **Dependent (mesh on part)**
   f. Click **Apply**. The Frame is displayed in the viewport.
   g. Set **Parts** to **CrossBracing**
   h. Set **Instance Type** to **Dependent (mesh on part)**
   i. Click **OK.** Now both Frame and CrossBracing are displayed in the viewport. Note that they are not actually connected together but only look that way since we created the parts in the correct locations.
   j. If you wish to see the rendered beam profiles, using the menu bar click on **View>Assembly Display Options.** The **Assembly Display Options** window is displayed. In the **General** tab, in the **Idealizations** section, check **Render beam profiles.** Click **OK.** Disable the beam profile rendering by repeating the process and unchecking **Render beam profiles.**
10. Create the connector wires
    a. Change the **Module** (displayed above viewport) to **Interaction** if it isn't already the case using the dropdown menu
    b. Click on the **Create Wire Feature** tool in the toolbar. The **Create Wire Feature** window is displayed
    c. Set **Add Method** to **Disjoint wires**
    d. Click the **Add...** button. You see the message **Select the first point** displayed below the viewport.

e. Click on the first point. You see the message **Select the second point** displayed below the viewport
f. Click on the same point again. You again see the message **Select the first point** displayed below the viewport
g. Repeat the procedure till 12 of the 16 nodes are selected. Do not select the 4 nodes of the second loop as we will demonstrate a different method for these.
h. Click **Done**. All the selected points are displayed in the list.
i. Ensure that **Create set of wires** is checked
j. Click **OK**.
k. Expand the **Assembly** container in the Model Database. Expand the **Sets** container. You see **Wire-1-Set-1**. This is the set of connector wires we have just created. Right-mouse-click on it and choose **Rename.** The **Rename Set** window is displayed
l. Set the name to **Set of connector wires**
m. Click **OK**

11. Create connector sections
    a. Double-click on **Connector Sections** in the Model Database. The **Create Connector Section** window is displayed
    b. Set **Name** to **FrameCrossConnSect**
    c. Set **Connection Category** to **Basic**
    d. For **Connection Type** set **Translation type** to **Join** and leave **Rotational type** at the default of **None**. You will see **Constrained CORM: U1, U2, U3**
    e. Click **Continue...**The **Edit Connector Section** window is displayed
    f. Leave everything as it is and click **OK.**
12. Assign connectors
    a. Expand the **Assembly** container in the Model Database. Double-click **Connector Assignment.** You see the message **Select wires or attachment lines to be assigned a section** displayed below the viewport
    b. At the right of the message is a button **Sets...** Since we earlier assigned the connector wires to a set during their creation, we can use this. Click it. The **Region Selection** window is displayed
    c. Choose **Set of connector wires** from the list.
    d. Click **Continue...**The **Edit Connector Section Assignment** window is displayed
    e. Set Section to **FrameCrossConnSect**
    f. Click **OK**

13. Identify Sets for remaining 4 nodes
    a. Expand the **Assembly** container in the Model Database. Expand the **Instances** container.
    b. Right-click on **Frame-1** and choose **Suppress. Frame-1** becomes invisible.
    c. Double-click on **Sets**. The **Create Set** window is displayed.
    d. Set **Name** to **crossnode1**
    e. Click **Continue...**You see the message **Select the geometry for the set** displayed below the viewport
    f. Select one of the upper nodes of the crossbracing which was not used as a connector.
    g. Click **Done.**
    h. Once again double-click on **Sets**. The **Create Set** window is displayed.
    i. Set **Name** to **crossnode2**
    j. Click **Continue...**
    k. Yousee the message **Select the geometry for the set** displayed below the viewport
    l. Select the other upper node of the crossbracing
    m. Click **Done**
    n. Right-click on **Frame-1** and choose **Resume**. **Frame-1** becomes visible again
    o. Right-click on **CrossBracing-1** and choose **Suppress**. **CrossBracing-1** becomes invisible
    p. Double-click on **Sets.** The **Create Set** window is displayed.
    q. Set **Name** to **framenode1**
    r. Click **Continue...**You see the message **Select the geometry for the set** displayed below the viewport
    s. Select the node on the frame which corresponds to crossbracing1
    t. Click **Done.**
    u. Once again double-click on **Sets**. The **Create Set** window is displayed.
    v. Set **Name** to **framenode2**
    w. Click **Continue...**
    x. You see the message **Select the geometry for the set** displayed below the viewport
    y. Select the node on the frame which corresponds to crossbracing2
    z. Click **Done**

aa. Right-click on **CrossBracing-1** and choose **Resume**. **CrossBracing-1** becomes visible again

14. Create constraints for the 4 nodes
    a. Double-click on **Constraints** in the Model Database. The **Create Constraint** window is displayed
    b. Set **Name** to **JoinConstraint1**
    c. Set **Type** to **Equation**
    d. Click **Continue…**
    e. The **Edit Constraint** window is displayed
    f. Set the following values in the table

	Coefficient	Set Name	DOF	CSYS ID
1	1	Crossnode1	1	(global)
2	-1	Framenode1	1	(global)

    g. Click **OK**.
    h. Repeat the process to create JoinConstraint2, JoinConstraint3, JoinConstraint4, JoinConstraint5 and JoinConstraint6 with the following values in the respective tables

JoinConstraint 2

	Coefficient	Set Name	DOF	CSYS ID
1	1	Crossnode1	2	(global)
2	-1	Framenode1	2	(global)

JoinConstraint3

	Coefficient	Set Name	DOF	CSYS ID
1	1	Crossnode1	3	(global)
2	-1	Framenode1	3	(global)

JoinConstraint4

	Coefficient	Set Name	DOF	CSYS ID
1	1	Crossnode2	1	(global)
2	-1	Framenode2	1	(global)

JoinConstraint5

	Coefficient	Set Name	DOF	CSYS ID
1	1	Crossnode2	2	(global)
2	-1	Framenode2	2	(global)

JoinConstraint6

	Coefficient	Set Name	DOF	CSYS ID
1	1	Crossnode2	3	(global)

2	-1	Framenode2	3	(global)

15. Create Steps
    i. Double-click on **Steps** in the Model Database. The **Create Step** window is displayed.
    j. Set **Name** to **Apply Loads**
    k. Set **Insert New Step After** to **Initial**
    l. Set **Procedure Type** to **General >Static, General**
    m. Click **Continue..** The **Edit Step** window is displayed
    n. In the **Basic** tab, set **Description** to **Loads are applied in this step**.
    o. Click **OK**.
16. Assign Loads
    a. Double-click on **Loads** in the Model Database. The **Create Load** window is displayed
    b. Set **Name** to **CrossLoad1**
    c. Set **Step** to **ApplyLoads**
    d. Set **Category** to **Mechanical**
    e. Set **Type for Selected Step** to **Line load**
    f. Click **Continue...**
    g. The **Region Selection** window is displayed. You see the message **Select a region from the dialog** displayed below the viewport. However we wish to select the elements by clicking in the viewport. Click on **Select in Viewport**
    h. You see the message **Select bodies for the load** displayed below the viewport. Select the crossbar by clicking on it.
    i. Click **Done**.The **EditLoad** window is displayed
    j. Set **Component 2** to **-1000**.
    k. Click **OK**.
    l. You will see the force displayed with arrows in the viewport on the selected crossbrace
    m. Repeat the process to create a line load on the adjacent cross bracing. Name the load **CrossLoad2**.
    n. Repeat the process to create a line load on the frontal frame element. Name the load **FrameLoad1**. Set Component 2 to **-1500**
    o. Repeat the process to create a line load on the frame element diagonally across from this one. Name the load **FrameLoad2**.Set Component 2 to **-500**
17. Apply boundary conditions

a. Double-click on **BCs** in the Model Database. The **Create Boundary Condition** window is displayed
b. Set **Name** to **FixBottom**
c. Set **Step** to **Initial**
d. Set **Category** to **Mechanical**
e. Set **Types for Selected Step** to **Displacement/Rotation**
f. Click **Continue...**
g. You see the message **Select regions for the boundary condition** at the bottom of the viewport. Click while pressing the "Shift" key on your keyboard to select all the elements (beams) at the base of the structure.
h. Click **Done**. The **Edit Boundary Condition** window is displayed.
i. Check off **U1, U2** and **U3**. This will pin these beams allowing them to rotate but preventing any translational motion.
j. Click **OK**.

18. Create the mesh
    a. Expand the **Parts** container in the Model Database.
    b. Expand **CrossBracing**
    c. Double-click on **Mesh (Empty)**. The viewport window changes to the **Mesh module** and the tools in the toolbar are now meshing tools.
    d. Using the menu bar click on **Mesh > Element Type ...**
    e. You see the message **Select the regions to be assigned element types** displayed below the viewport
    f. Click and drag using your mouse to select all the crossbraces.
    g. Click **Done**. The **Element Type** window is displayed.
    h. Set **Element Library** to **Standard**
    i. Set **Geometric Order** to **Linear**
    j. Set **Family** to **Beam**
    k. You will notice the message **B31: A 2-node linear beam in space**
    l. Click **OK**
    m. Click **Done**
    n. Using the menu bar lick on **Seed > Edge by Number**
    o. You see the message **Select the regions to be assigned local seeds** displayed below the viewport
    p. Click and drag using your mouse to select all the cross braces
    q. Click **Done**.

r. You see the prompt **Number of elements along the edges** displayed below the viewport.
s. Set it to 4 and press the "Enter" key on your keyboard
t. Click Done
u. Using the menu bar lick on **Mesh > Part**
v. You see the prompt **OK to mesh the part?** displayed below the viewport
w. Click **Yes**
x. Repeat the above process to mesh the frame as well.

19. Create and submit the job
    a. Double-click on **Jobs** in the Model Database. The **Create Job** window is displayed
    b. Set **Name** to **BeamFrameAnalysisJob**
    c. Set **Source** to **Model**
    d. Select **BeamFrame** (it is the only option displayed)
    e. Click **Continue..** The **Edit Job** window is displayed
    f. Set **Description** to **Analysis of loaded beam frame**
    g. Set **Job Type** to **Full Analysis**.
    h. Leave all other options at defaults
    i. Click **OK**
    j. Expand the **Jobs** container in the Model Database
    k. Right-click on **BeamFrameAnalysisJob** and choose **Submit**.
    l. It is quite possible that you will get an error message stating that connector assignments reference regions are empty or have been deleted or suppressed. Click **Dismiss**
    y. Expand the **Assembly** container in the model tree. Expand the **Features** container. You will find that Wire-1 has been crossed off. Right click on it and select **Resume**.
    m. Try running the simulation again. This time it will run. You will see the following messages in the message window:

       The job input file "BeamFrameAnalysisJob.inp" has been submitted for analysis.
       Job BeamFrameAnalysisJob: Analysis Input File Processor completed successfully
       Job BeamFrameAnalysisJob: Abaqus/Standard completed successfully
       Job BeamFrameAnalysisJob completed successfully

9.3 Python Script

The following Python script replicates the above procedure for the analysis of the beam frame. You can find it in the source code accompanying the book in **beamframe.py**. You can run it by opening a new model in Abaqus (**File > New Model Database > With Standard/Explicit Model**) and running it with **File > Run Script...**

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# -------------------------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Beam Frame')
beamModel = mdb.models['Beam Frame']

# -------------------------------------------------------------------------
# Create the parts

import sketch
import part

# -------------------------------------------------------------------------
# Create the frame

# Start with a 3D Point Deformable Body
framePart = beamModel.Part(name='Frame', dimensionality=THREE_D,
                                         type=DEFORMABLE_BODY)
framePart.ReferencePoint(point=(0.0, 0.0, 0.0))

# ---------------------------------------------
# a) Create one side of the frame

# Create other datum points by offsetting from the reference point
the_reference_point = framePart.referencePoints[1]
framePart.DatumPointByOffset(point=the_reference_point, vector=(13.0,0.0,0.0))
framePart.DatumPointByOffset(point=the_reference_point, vector=(4.0,-3.0,0.0))
framePart.DatumPointByOffset(point=the_reference_point, vector=(1.0,0.0,0.0))

# There are 3 items in the datums repository, the 3 datum points that have been
# created. Extract the keys of the datums repository, these will be used to get
# the datum points. The keys will be numbers but might be in random order as
# dictionaries are unordered. Sort them to get them in ascending order as Abaqus
# assigns keys in ascending order as a datum point is created. Once the points
# are available a datum plane is created using the 3 points
framePart_datums_keys = framePart.datums.keys()
framePart_datums_keys.sort()
```

```
frame_datum_point_1 = framePart.datums[framePart_datums_keys[2]]
frame_datum_point_2 = framePart.datums[framePart_datums_keys[1]]
frame_datum_point_3 = framePart.datums[framePart_datums_keys[0]]
framePart.DatumPlaneByThreePoints(point1=frame_datum_point_1,
                                  point2=frame_datum_point_2,
                                  point3=frame_datum_point_3)

# Create a datum axis
framePart.DatumAxisByPrincipalAxis(principalAxis=YAXIS)

# There are 5 objects in the datums repository, 3 datum points, a datum plane,
# and the datum axis
# The datum plane will be the one whose key is the second to highest number
# The datum axis will be the one whose key is the highest number
framePart_datums_keys = framePart.datums.keys()
framePart_datums_keys.sort()
index_of_plane = (len(framePart_datums_keys) - 2)
index_of_axis = (len(framePart_datums_keys) - 1)
frame_datum_plane = framePart.datums[framePart_datums_keys[index_of_plane]]
frame_datum_axis = framePart.datums[framePart_datums_keys[index_of_axis]]

# create the sketch
sketch_transform1 = framePart.MakeSketchTransform(sketchPlane=frame_datum_plane,
sketchUpEdge=frame_datum_axis, sketchPlaneSide=SIDE1, sketchOrientation=LEFT,
                                                     origin=(0.0, 0.0, 0.0))
framePart_sketch = mdb.models['Beam Frame'] \
                                   .ConstrainedSketch(name='frame sketch 1',
                                                     sheetSize=20,
                                                     gridSpacing=1,
                                                     transform=sketch_transform1)

# In place of last two statements, could instead do:
# framePart_sketch = beamModel.ConstrainedSketch(name='sketch frame',
#                                                sheetSize=30,
#                                                gridSpacing=10)
# However it is better to use MakeSketchTransform when sketching onto a plane

framePart_sketch.Line(point1=(1.0,0.0), point2=(4.0,-3.0))
framePart_sketch.Line(point1=(4.0,-3.0), point2=(6.0,-3.0))
framePart_sketch.Line(point1=(6.0,-3.0), point2=(8.0,-3.0))
framePart_sketch.Line(point1=(8.0,-3.0), point2=(10.0,-3.0))
framePart_sketch.Line(point1=(10.0,-3.0), point2=(13.0,0.0))
framePart_sketch.Line(point1=(13.0,0.0), point2=(8.0,0.0))
framePart_sketch.Line(point1=(8.0,0.0), point2=(6.0,0.0))
framePart_sketch.Line(point1=(6.0,0.0), point2=(1.0,0.0))
framePart_sketch.Line(point1=(6.0,0.0), point2=(6.0,-3.0))
framePart_sketch.Line(point1=(8.0,0.0), point2=(8.0,-3.0))

# use the sketch to create a wire
framePart.Wire(sketchPlane=frame_datum_plane, sketchUpEdge=frame_datum_axis,
               sketchPlaneSide=SIDE1, sketchOrientation=LEFT,
               sketch=framePart_sketch)
```

```
# ---------------------------------------------
# b) Create other side of the frame

# create a datum plane by offsetting from existing one
framePart.DatumPlaneByOffset(plane=frame_datum_plane, flip=SIDE1, offset=1.5)

framePart_datums_keys = framePart.datums.keys()
framePart_datums_keys.sort()
index_of_plane2 = (len(framePart_datums_keys) - 1)
frame_datum_plane2=framePart.datums[framePart_datums_keys[index_of_plane2]]

framePart.DatumPointByCoordinate(coords=(1.0, 0.0, 1.5))
framePart.DatumPointByCoordinate(coords=(13.0, 0.0, 1.5))
framePart.DatumPointByCoordinate(coords=(4.0, -3.0, 1.5))

framePart.DatumAxisByTwoPoint(point1=(0.0,0.0,1.5), point2=(0.0,5.0,1.5))

framePart_datums_keys = framePart.datums.keys()
framePart_datums_keys.sort()
index_of_axis2 = (len(framePart_datums_keys) -1)
frame_datum_axis2 = framePart.datums[framePart_datums_keys[index_of_axis2]]


# Make the sektch
sketch_transform2 = framePart.MakeSketchTransform(sketchPlane=frame_datum_plane2,
                                                  sketchUpEdge=frame_datum_axis2,
                                                  sketchPlaneSide=SIDE1,
                                                  sketchOrientation=LEFT,
                                                  origin=(0.0, 0.0, 1.5))
# We could also have used frame_datum_axis instead of frame_datum_axis2
# sketch_transform2 = framePart \
#                          .MakeSketchTransform(sketchPlane=frame_datum_plane2,
#                                               sketchUpEdge=frame_datum_axis2,
#                                               sketchPlaneSide=SIDE1,
#                                               sketchOrientation=LEFT,
#                                               origin=(0.0, 0.0, 1.5))
framePart_sketch2 = mdb.models['Beam Frame'] \
                             .ConstrainedSketch(name='frame sketch 2',
                                                sheetSize=20,
                                                gridSpacing=1,
                                                transform=sketch_transform2)

framePart_sketch2.Line(point1=(1.0,0.0), point2=(4.0,-3.0))
framePart_sketch2.Line(point1=(4.0,-3.0), point2=(6.0,-3.0))
framePart_sketch2.Line(point1=(6.0,-3.0), point2=(8.0,-3.0))
framePart_sketch2.Line(point1=(8.0,-3.0), point2=(10.0,-3.0))
framePart_sketch2.Line(point1=(10.0,-3.0), point2=(13.0,0.0))
framePart_sketch2.Line(point1=(13.0,0.0), point2=(8.0,0.0))
framePart_sketch2.Line(point1=(8.0,0.0), point2=(6.0,0.0))
framePart_sketch2.Line(point1=(6.0,0.0), point2=(1.0,0.0))
framePart_sketch2.Line(point1=(6.0,0.0), point2=(6.0,-3.0))
```

```
framePart_sketch2.Line(point1=(8.0,0.0), point2=(8.0,-3.0))


"""
# The following code also works. You set the origin to something different and
# change all the coordinates accordingly.
# Make the sektch
sketch_transform2 = framePart.MakeSketchTransform(sketchPlane=frame_datum_plane2,
                                                  sketchUpEdge=frame_datum_axis2,
                                                  sketchPlaneSide=SIDE1,
                                                  sketchOrientation=LEFT,
                                                  origin=(7.0, -1.5, 1.5))
# we could also have used frame_datum_axis instead of frame_datum_axis2
sketch_transform2 = framePart.MakeSketchTransform(sketchPlane=frame_datum_plane2,
                                                  sketchUpEdge=frame_datum_axis2,
                                                  sketchPlaneSide=SIDE1,
                                                  sketchOrientation=LEFT,
                                                  origin=(7.0, -1.5, 1.5))
framePart_sketch2 = mdb.models['Beam Frame'] \
                                  .ConstrainedSketch(name='frame sketch 2',
                                                   sheetSize=20,
                                                   gridSpacing=1,
                                                   transform=sketch_transform2)
framePart_sketch2.Line(point1=(-6.0,1.5), point2=(-3.0,-1.5))
framePart_sketch2.Line(point1=(-3.0,-1.5), point2=(-1.0,-1.5))
framePart_sketch2.Line(point1=(-1.0,-1.5), point2=(1.0,-1.5))
framePart_sketch2.Line(point1=(1.0,-1.5), point2=(3.0,-1.5))
framePart_sketch2.Line(point1=(3.0,-1.5), point2=(6.0,1.5))
framePart_sketch2.Line(point1=(6.0,1.5), point2=(1.0,1.5))
framePart_sketch2.Line(point1=(1.0,1.5), point2=(-1.0,1.5))
framePart_sketch2.Line(point1=(-1.0,1.5), point2=(-6.0,1.5))
framePart_sketch2.Line(point1=(-1.0,1.5), point2=(-1.0,-1.5))
framePart_sketch2.Line(point1=(1.0,1.5), point2=(1.0,-1.5))

"""


# Use the sketch to create a wire
framePart.Wire(sketchPlane=frame_datum_plane2, sketchUpEdge=frame_datum_axis2,
               sketchPlaneSide=SIDE1, sketchOrientation=LEFT,
               sketch=framePart_sketch2)

# -----------------------------------------------------------------------
# Create the cross bracing

# Start with a 3D Point Deformable Body
crossPart = beamModel.Part(name='CrossBracing', dimensionality=THREE_D,
                                                 type=DEFORMABLE_BODY)
crossPart.ReferencePoint(point=(0.0, 0.0, 0.0))

crossPart.DatumPointByCoordinate(coords=(1.0, 0.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(1.0, 0.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(4.0, -3.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(4.0, -3.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(6.0, 0.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(6.0, 0.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(6.0, -3.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(6.0, -3.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(8.0, 0.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(8.0, 0.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(8.0, -3.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(8.0, -3.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(10.0, -3.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(10.0, -3.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(13.0, 0.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(13.0, 0.0, 1.5))


crossPart_datums_keys = crossPart.datums.keys()
crossPart_datums_keys.sort()

datum_points = crossPart.datums
crossPart.WirePolyLine(points=((datum_points[crossPart_datums_keys[0]],
                                datum_points[crossPart_datums_keys[1]]),
                               (datum_points[crossPart_datums_keys[2]],
                                datum_points[crossPart_datums_keys[3]]),
                               (datum_points[crossPart_datums_keys[4]],
                                datum_points[crossPart_datums_keys[5]]),
                               (datum_points[crossPart_datums_keys[6]],
                                datum_points[crossPart_datums_keys[7]]),
                               (datum_points[crossPart_datums_keys[8]],
                                datum_points[crossPart_datums_keys[9]]),
                               (datum_points[crossPart_datums_keys[10]],
                                datum_points[crossPart_datums_keys[11]]),
                               (datum_points[crossPart_datums_keys[12]],
                                datum_points[crossPart_datums_keys[13]]),
                               (datum_points[crossPart_datums_keys[14]],
                                datum_points[crossPart_datums_keys[15]])),
                                mergeWire=OFF, meshable=ON)

# -----------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs modulus
# and poissons ratio
beamMaterial = beamModel.Material(name='AISI 1005 Steel')
beamMaterial.Density(table=((7872, ),       ))
beamMaterial.Elastic(table=((200E9, 0.29), ))

# -----------------------------------------------------------------------
# Create profiles
```

```
beamModel.IProfile(name='FrameProfile', l=0.075, h=0.15, b1=0.12, b2=0.12,
                                                 t1=0.02, t2=0.02, t3=0.04)
beamModel.IProfile(name='CrossProfile', l=0.06, h=0.12, b1=0.11, b2=0.08,
                                                 t1=0.01, t2=0.01, t3=0.02)

# ---------------------------------------------------------------------
# Create sections and assign frame and crosbracing to them

import section

frameSection = beamModel.BeamSection(name='Frame Section', profile='FrameProfile',
                                     integration=DURING_ANALYSIS,
                                     material='AISI 1005 Steel')
crossSection = beamModel.BeamSection(name='Cross Section', profile='CrossProfile',
                                     integration=DURING_ANALYSIS,
                                     material='AISI 1005 Steel')

edges_for_frame_section_assignment = framePart.edges.findAt(((3.5, 0.0, 0.0), ),
                                                            ((7.0, 0.0, 0.0), ),
                                                            ((10.5, 0.0, 0.0), ),
                                                            ((2.5, -1.5, 0.0), ),
                                                            ((5.0, -3.0, 0.0), ),
                                                            ((7.0, -3.0, 0.0), ),
                                                            ((9.0, -3.0, 0.0), ),
                                                            ((11.5, -1.5, 0.0), ),
                                                            ((6.0, -1.5, 0.0), ),
                                                            ((8.0, -1.5, 0.0), ),
                                                            ((3.5, 0.0, 1.5), ),
                                                            ((7.0, 0.0, 1.5), ),
                                                            ((10.5, 0.0, 1.5), ),
                                                            ((2.5, -1.5, 1.5), ),
                                                            ((5.0, -3.0, 1.5), ),
                                                            ((7.0, -3.0, 1.5), ),
                                                            ((9.0, -3.0, 1.5), ),
                                                            ((11.5, -1.5, 1.5), ),
                                                            ((6.0, -1.5, 1.5), ),
                                                            ((8.0, -1.5, 1.5), ),)

frame_region = regionToolset.Region(edges=edges_for_frame_section_assignment)
framePart.SectionAssignment(region=frame_region, sectionName='Frame Section')


edges_for_cross_section_assignment = crossPart.edges.findAt(((1.0, 0.0, 0.75), ),
                                                            ((6.0, 0.0, 0.75), ),
                                                            ((8.0, 0.0, 0.75), ),
                                                            ((13.0, 0.0, 0.75), ),
                                                            ((4.0, -3.0, 0.75), ),
                                                            ((6.0, -3.0, 0.75), ),
                                                            ((8.0, -3.0, 0.75), ),
                                                            ((10, -3.0, 0.75), ),)

cross_region = regionToolset.Region(edges=edges_for_cross_section_assignment)
```

```
crossPart.SectionAssignment(region=cross_region, sectionName='Cross Section')

# -----------------------------------------------------------------------
# Assign beam orientations

framePart.assignBeamSectionOrientation(region=frame_region, method=N1_COSINES,
                                                            n1=(0.0, 0.0, 1.0))

crossPart.assignBeamSectionOrientation(region=cross_region, method=N1_COSINES,
                                                            n1=(1.0, 0.0, 0.0))

# -----------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
beamAssembly = beamModel.rootAssembly

frameInstance = beamAssembly.Instance(name='Frame Instance', part=framePart,
                                                             dependent=ON)
crossInstance = beamAssembly.Instance(name='Cross Instance', part=crossPart,
                                                             dependent=ON)

# -----------------------------------------------------------------------
# Create the wire features

vertices_for_frame_side_1 = frameInstance.vertices.findAt(((1.0, 0.0, 0.0),),
                                                          ((4.0, -3.0, 0.0),),
                                                          ((6.0, -3.0, 0.0),),
                                                          ((10.0, -3.0, 0.0),),
                                                          ((13.0, 0.0, 0.0),),
                                                          ((6.0, 0.0, 0.0),),)

vertices_for_frame_side_2 = frameInstance.vertices.findAt(((1.0, 0.0, 1.5),),
                                                         ((4.0, -3.0, 1.5),),
                                                         ((6.0, -3.0, 1.5),),
                                                         ((10.0, -3.0, 1.5),),
                                                         ((13.0, 0.0, 1.5),),
                                                         ((6.0, 0.0, 1.5),),)

vertices_for_crossbars_side_1 =  crossInstance.vertices.findAt(((1.0, 0.0, 0.0),),
                                                               ((4.0, -3.0, 0.0),),
                                                               ((6.0, -3.0, 0.0),),
                                                               ((10.0, -3.0, 0.0),),
                                                               ((13.0, 0.0, 0.0),),
                                                               ((6.0, 0.0, 0.0),),)

vertices_for_crossbars_side_2 = crossInstance.vertices.findAt(((1.0, 0.0, 1.5),),
                                                              ((4.0, -3.0, 1.5),),
                                                              ((6.0, -3.0, 1.5),),
                                                              ((10.0, -3.0, 1.5),),
                                                                    ((13.0, 0.0, 1.5),),
                                                                    ((6.0, 0.0, 1.5),),)

# We need to create the connectors using these points
# The format of the WirePolyLine function is WirePolyLine(points=((v1[1], v2[1]),
# (v1[2], v2[2]), mergeWire=OFF, meshable=OFF)
# Notice that the points argument is a tuple of point pairs, in other words a
# tuple of tuples
# We first create the tuples of the point pairs specifying the individual
# connections such as (v1[1], v2[1]) and put them in a list and then create
# a tuple of those tuples from that list using the "tuple()" function

list_of_point_tuples = []
for i in range(len(vertices_for_frame_side_1)):
    list_of_point_tuples.append((vertices_for_frame_side_1[i],
                                 vertices_for_crossbars_side_1[i]))
    list_of_point_tuples.append((vertices_for_frame_side_2[i],
                                 vertices_for_crossbars_side_2[i]))

tuple_of_point_tuples = tuple(list_of_point_tuples)

beamAssembly.WirePolyLine(points=tuple_of_point_tuples, mergeWire=OFF,
                                                        meshable=OFF)

# ---------------------------------------------------------------------------
# Assign these wire features/connectors to a set that can be used later
# This is the equivalent of checking off the "Create set of wires" checkbox in
# the "Create Wire Feature" window

# This of course requires that we first find the edges that are the wire
# connectors

edges_for_connector_set = beamAssembly.edges.findAt(((1.0, 0.0, 0.0),),
                                                    ((4.0, -3.0, 0.0),),
                                                    ((6.0, -3.0, 0.0),),
                                                    ((10.0, -3.0, 0.0),),
                                                    ((13.0, 0.0, 0.0),),
                                                    ((6.0, 0.0, 0.0),),
                                                    ((1.0, 0.0, 1.5),),
                                                    ((4.0, -3.0, 1.5),),
                                                    ((6.0, -3.0, 1.5),),
                                                    ((10.0, -3.0, 1.5),),
                                                    ((13.0, 0.0, 1.5),),
                                                    ((6.0, 0.0, 1.5),),)

# Now assign them to a set
beamAssembly.Set(edges=edges_for_connector_set, name='Set of connector wires')

# ---------------------------------------------------------------------------
# Create a connector section

beamModel.ConnectorSection(name='FrameCrossConnSect', translationalType=JOIN)
```

```
# -----------------------------------------------------------------------
# Assign this connector section to the wire features using the set created earlier
conn_wire_region = beamAssembly.sets['Set of connector wires']
beamAssembly.SectionAssignment(sectionName='FrameCrossConnSect',
                               region=conn_wire_region)

# -----------------------------------------------------------------------
# Use constraint equations on the other two nodes

# We did not apply the JOIN condition to four of the nodes
# We will instead use an equation constraint to achieve the same effect on the
# top two
# However we won't use the equation constraint on the lower two because we are
# going to fix that edge anyway and having an equation constraint as well as a
# fixed boundary condition might give an error

# First we need to assign the nodes, both on the frame part and on the crossbars,
# to sets

vertex_for_framenode_1 = frameInstance.vertices.findAt(((8.0, 0.0, 0.0),),)
beamAssembly.Set(vertices=vertex_for_framenode_1, name='framenode1')
vertex_for_framenode_2 = frameInstance.vertices.findAt(((8.0, 0.0, 1.5),),)
beamAssembly.Set(vertices=vertex_for_framenode_2, name='framenode2')
vertex_for_crossnode_1 = crossInstance.vertices.findAt(((8.0, 0.0, 0.0),),)
beamAssembly.Set(vertices=vertex_for_crossnode_1, name='crossnode1')
vertex_for_crossnode_2 = crossInstance.vertices.findAt(((8.0, 0.0, 1.5),),)
beamAssembly.Set(vertices=vertex_for_crossnode_2, name='crossnode2')

# Create the equation constraints
beamModel.Equation(name='JoinConstraint1', terms=((1.0, 'crossnode1', 1),
                                                  (-1.0, 'framenode1', 1)))
beamModel.Equation(name='JoinConstraint2', terms=((1.0, 'crossnode1', 2),
                                                  (-1.0, 'framenode1', 2)))
beamModel.Equation(name='JoinConstraint3', terms=((1.0, 'crossnode1', 3),
                                                  (-1.0, 'framenode1', 3)))
beamModel.Equation(name='JoinConstraint4', terms=((1.0, 'crossnode2', 1),
                                                  (-1.0, 'framenode2', 1)))
beamModel.Equation(name='JoinConstraint5', terms=((1.0, 'crossnode2', 2),
                                                  (-1.0, 'framenode2', 2)))
beamModel.Equation(name='JoinConstraint6', terms=((1.0, 'crossnode2', 3),
                                                  (-1.0, 'framenode2', 3)))

# -----------------------------------------------------------------------
# Create the step

import step

# Create a static general step
beamModel.StaticStep(name='Apply Loads', previous='Initial',
                     description='Loads are applied in this step')
```

```
# -------------------------------------------------------------------------
# Field output requests

# Leave at defaults

# -------------------------------------------------------------------------
# History output request

# Leave at defaults

# -------------------------------------------------------------------------
# Apply loads

edge_for_crossload1 = crossInstance.edges.findAt(((6.0, 0.0, 0.75), ),)
region_for_crossload1 = regionToolset.Region(edges=edge_for_crossload1)
edge_for_crossload2 = crossInstance.edges.findAt(((8.0, 0.0, 0.75), ),)
region_for_crossload2 = regionToolset.Region(edges=edge_for_crossload2)
edge_for_frameload1 = frameInstance.edges.findAt(((3.5, 0.0, 0.0), ),)
region_for_frameload1 = regionToolset.Region(edges=edge_for_frameload1)
edge_for_frameload2 = frameInstance.edges.findAt(((10.5, 0.0, 1.5), ),)
region_for_frameload2 = regionToolset.Region(edges=edge_for_frameload2)

beamModel.LineLoad(name='CrossLoad1', createStepName='Apply Loads',
                   region=region_for_crossload1, comp2=-1000.0)
beamModel.LineLoad(name='CrossLoad2', createStepName='Apply Loads',
                   region=region_for_crossload2, comp2=-1000.0)
beamModel.LineLoad(name='FrameLoad1', createStepName='Apply Loads',
                   region=region_for_frameload1, comp2=-1500.0)
beamModel.LineLoad(name='FrameLoad2', createStepName='Apply Loads',
                   region=region_for_frameload2, comp2=-500.0)

# -------------------------------------------------------------------------
# Apply boundary conditions

frame_edges_for_bc = frameInstance.edges.findAt(((5.0,-3.0,0.0),),
                                                ((7.0,-3.0,0.0),),
                                                ((9.0,-3.0,0.0),),
                                                ((5.0,-3.0,1.5),),
                                                ((7.0,-3.0,1.5),),
                                                ((9.0,-3.0,1.5),),)

cross_edges_for_bc = crossInstance.edges.findAt(((4.0,-3.0,0.75),),
                                                ((6.0,-3.0,0.75),),
                                                ((8.0,-3.0,0.75),),
                                                ((10.0,-3.0,0.75),),)

edges_for_bc = frame_edges_for_bc + cross_edges_for_bc
bc_region = regionToolset.Region(edges=edges_for_bc)

beamModel.DisplacementBC(name='FixBottom', createStepName='Initial',
                         region=bc_region, u1=SET, u2=SET, u3=SET,
                         ur1=UNSET, ur2= UNSET, ur3=UNSET,
                            amplitude=UNSET, distributionType=UNIFORM,
                            fieldName='', localCsys=None)

# -----------------------------------------------------------------------
# Create the mesh

import mesh

frame_mesh_region = frame_region
frame_edges_for_meshing = edges_for_frame_section_assignment
frame_mesh_element_type=mesh.ElemType(elemCode=B31, elemLibrary=STANDARD)
framePart.setElementType(regions=frame_mesh_region,
                            elemTypes=(frame_mesh_element_type, ))
framePart.seedEdgeByNumber(edges=frame_edges_for_meshing, number=4)
framePart.generateMesh()

cross_mesh_region = cross_region
cross_edges_for_meshing = edges_for_cross_section_assignment
cross_mesh_element_type=mesh.ElemType(elemCode=B31, elemLibrary=STANDARD)
crossPart.setElementType(regions=cross_mesh_region,
                            elemTypes=(cross_mesh_element_type, ))
crossPart.seedEdgeByNumber(edges=cross_edges_for_meshing, number=4)
crossPart.generateMesh()

# -----------------------------------------------------------------------
# Create and run the job

import job

# Create the job
mdb.Job(name='BeamFrameAnalysisJob', model='Beam Frame', type=ANALYSIS,
explicitPrecision=SINGLE,
    nodalOutputPrecision=SINGLE, description='Bending of loaded beam frame',
    parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
    numDomains=1, userSubroutine='', numCpus=1, memory=50, memoryUnits=PERCENTAGE,
scratch='',
    echoPrint=OFF, modelPrint=OFF, contactPrint=OFF, historyPrint=OFF)

# Run the job
mdb.jobs['BeamFrameAnalysisJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['BeamFrameAnalysisJob'].waitForCompletion()

# End of run job
```

9.4 Examining the Script

Let's go through the entire script, statement by statement, and understand how it works.

9.4.1 Initialization (import required modules)

The block dealing with this initialization is

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)
```

These statements are identical to those used in the Cantilever Beam example and were explained in section 4.3.1 on page 65

9.4.2 Create the model

The following code block creates the model

```
# -----------------------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Beam Frame')
beamModel = mdb.models['Beam Frame']
```

These statements rename the model from 'Model-1' to 'Beam Frame'. They are almost identical to those used in the Cantilever Beam example and were explained in section 4.3.2 on page 67.

9.4.3 Create the part

The following block begins the part creation process. (The majority of the code has been removed to save space, only a few comments are included)

```
# -----------------------------------------------------------------------
# Create the parts

import sketch
import part

# -----------------------------------------------------------------------
# Create the frame

# ----------------------------------------------
# a) Create one side of the frame

# --------------------------------------------
# b) Create other side of the frame

# Make the sketch

# use the sketch to create a wire

# -------------------------------------------------
```

Let's examine the statements.

```
import sketch
import part
```

These statements import the sketch and part modules into the script, thus providing access to the objects related to sketches and parts. They were explained in section 4.3.3 on page 69.

```
framePart = beamModel.Part(name='Frame', dimensionality=THREE_D,
                                         type=DEFORMABLE_BODY)
```

This statement creates a **Part** object and places it in the **parts** repository. The **name** of the part (its key in the repository) is set to 'Frame' and its **dimensionality** is set to a SymbolicConstant **THREE_D** which defines it to be a 3D part. It is defined to be of the type deformable body using the **DEFORMABLE_BODY** SymbolicConstant.

```
framePart.ReferencePoint(point=(0.0, 0.0, 0.0))
```

The **ReferencePoint()** method creates a **ReferencePoint** object, which is a **Feature** object, at the specified location. The argument **point** is a required argument. Here we specify it using a sequence of three Floats representing the X, Y and Z coordinates.

```
the_reference_point = framePart.referencePoints[1]
```

The created reference point is automatically assigned the key '1' since it is the first point. We therefore refer to it using **framePart.referencePoints[1]** and assign it to the variable **the_reference_point** for later use.

```
framePart.DatumPointByOffset(point=the_reference_point, vector=(13.0,0.0,0.0))
framePart.DatumPointByOffset(point=the_reference_point, vector=(4.0,-3.0,0.0))
framePart.DatumPointByOffset(point=the_reference_point, vector=(1.0,0.0,0.0))
```

The **DatumPointByOffsetI()** method creates **DatumPoint** objects which are offset from the original point (specified by the **point** argument) by a vector (specified by the **vector** argument).

```
framePart_datums_keys = framePart.datums.keys()
framePart_datums_keys.sort()
frame_datum_point_1 = framePart.datums[framePart_datums_keys[2]]
frame_datum_point_2 = framePart.datums[framePart_datums_keys[1]]
frame_datum_point_3 = framePart.datums[framePart_datums_keys[0]]
framePart.DatumPlaneByThreePoints(point1=frame_datum_point_1,
                                  point2=frame_datum_point_2,
                                  point3=frame_datum_point_3)
```

We now wish to assign the 3 **DatumPoint** objects to variables. All three datum points have repository keys which can be used to refer to them. However we do not know what key Abaqus has assigned to them. We can access the keys using **framePart.datums.keys()**. Since dictionaries are stored in any random order, we use the **sort()** command to place them in ascending order. This is possible because Abaqus assigns the keys in ascending order as they are created. Once this has been done we can refer to the keys using the regular list notation such as **framePart_datums_keys[2]**. And we can use each key to refer to the corresponding datum point such as **framePart.datums[framePart_datums_keys[2]]**. Finally we use the **DatumPlaneByThreePoints()** methods to create a **DatumPlane** object (which, by the way, is a **Feature** object). Three points are passed as arguments to this method.

```
framePart.DatumAxisByPrincipalAxis(principalAxis=YAXIS)
```

The **DatumAxisByPrincipalAxis()** method creates a **DatumAxis** object (a **Feature** object) along one of the principal axes. We use the **principalAxis** argument to specify which axis using **XAXIS, YAXIS** or **ZAXIS** SymbolicConstants.

```
framePart_datums_keys = framePart.datums.keys()
framePart_datums_keys.sort()
index_of_plane = (len(framePart_datums_keys) - 2)
index_of_axis = (len(framePart_datums_keys) - 1)
frame_datum_plane = framePart.datums[framePart_datums_keys[index_of_plane]]
frame_datum_axis = framePart.datums[framePart_datums_keys[index_of_axis]]
```

We need to assign the datum plane and the datum axis to variables so that they can be used later. Once again we will extract these variables from the **datums** list using their corresponding keys. Since we do not know the keys, we will have to figure them out. We sort them in order using the **sort()** method as done before. Since the datum axis was the last object to be created, its key will be one less than the length of the **datums** array. And the datum plane being the second to last datum object created, its key will be two less than the length of the **datums** array. We use the Python **len()** command to obtain the length of a list.

```
sketch_transform1 = framePart.MakeSketchTransform(sketchPlane=frame_datum_plane,
sketchUpEdge=frame_datum_axis, sketchPlaneSide=SIDE1, sketchOrientation=LEFT,
                                                    origin=(0.0, 0.0, 0.0))
```

This statement identifies the datum plane we have created as the plane on which the frame will be sketched. The **MakeSketchTransform()** method creates a **Transform** object, which basically represents the transformation from sketch coordinates to part coordinates. **sketchPlane** is a required argument, and it specifies a datum plane or planar face object which will be the sketch plane. **sketchUpEdge** is an optional argument, an **Edge** or **DatumAxis** object, which specifies the orientation of the sketch. **sketchPlaneSide**, an optional parameter, specifies which side of **sketchPlane** the sketch will be positioned. This can either be **SIDE1** or **SIDE2**. **sketchOrientation**, another optional parameter, specifies the orientation of **sketchUpEdge** on the sketch. Possible SymbolicConstant values are **TOP**, **BOTTOM**, **LEFT** and **RIGHT**. **origin** is also an optional argument. It is a sequence of floats specifying the coordinates of the point that will be the origin of the sketch.

```
framePart_sketch = mdb.models['Beam Frame'] \
                                    .ConstrainedSketch(name='frame sketch 1',
                                                       sheetSize=20,
                                                       gridSpacing=1,
                                                       transform=sketch_transform1)
```

This statement creates a **ConstrainedSketch** object by calling the **ConstrainedSketch()** method of the **Model** object. This was explained in section 4.3.3on page 69.

```
framePart_sketch.Line(point1=(1.0,0.0), point2=(4.0,-3.0))
framePart_sketch.Line(point1=(4.0,-3.0), point2=(6.0,-3.0))
framePart_sketch.Line(point1=(6.0,-3.0), point2=(8.0,-3.0))
framePart_sketch.Line(point1=(8.0,-3.0), point2=(10.0,-3.0))
framePart_sketch.Line(point1=(10.0,-3.0), point2=(13.0,0.0))
framePart_sketch.Line(point1=(13.0,0.0), point2=(8.0,0.0))
framePart_sketch.Line(point1=(8.0,0.0), point2=(6.0,0.0))
framePart_sketch.Line(point1=(6.0,0.0), point2=(1.0,0.0))
framePart_sketch.Line(point1=(6.0,0.0), point2=(6.0,-3.0))
framePart_sketch.Line(point1=(8.0,0.0), point2=(8.0,-3.0))
```

The statements use the **Line()** method of the **ConstrainedSketchGeometry** object. The **ConstrainedSketchGeometry** object stores the geometry of a sketch, such as lines, circles, arcs, and construction lines. The sketch module defines **ConstrainedSketchGeometry** objects. The first parameter **point1** is a pair of floats specifying the coordinates of the first endpoint of the line. The second parameter **point2** is a pair of floats specifying the coordinates of the second endpoint.

```
framePart.Wire(sketchPlane=frame_datum_plane, sketchUpEdge=frame_datum_axis,
               sketchPlaneSide=SIDE1, sketchOrientation=LEFT,
               sketch=framePart_sketch)
```

The **Wire()** method creates a Feature object, a planar wire, using a given **ConstrainedSketch** object. There are 4 required arguments. **sketchPlane** is a datum plane object or a face object which specifies the plane on which to sketch. In our case it is **frame_datum_plane**. **sketchUpEdge** is an **Edge** object or a **DatumAxis** object which specifies the direction. **sketchPlaneSide** specifies the direction of the feature creation, and it can be either **SIDE1** or **SIDE2**. **sketch** is the **ConstrainedSketch** object. An optional argument is **sketchOrientation** which specifies the orientation of **sketchUpEdge** on the sketch, and it can be **TOP**, **BOTTOM**, **LEFT** or **RIGHT**.

```
# ----------------------------------------------
# b) Create other side of the frame

# create a datum plane by offsetting from existing one
framePart.DatumPlaneByOffset(plane=frame_datum_plane, flip=SIDE1, offset=1.5)

framePart_datums_keys = framePart.datums.keys()
framePart_datums_keys.sort()
index_of_plane2 = (len(framePart_datums_keys) - 1)
frame_datum_plane2=framePart.datums[framePart_datums_keys[index_of_plane2]]

framePart.DatumPointByCoordinate(coords=(1.0, 0.0, 1.5))
framePart.DatumPointByCoordinate(coords=(13.0, 0.0, 1.5))
framePart.DatumPointByCoordinate(coords=(4.0, -3.0, 1.5))

framePart.DatumAxisByTwoPoint(point1=(0.0,0.0,1.5), point2=(0.0,5.0,1.5))

framePart_datums_keys = framePart.datums.keys()
framePart_datums_keys.sort()
index_of_axis2 = (len(framePart_datums_keys) -1)
frame_datum_axis2 = framePart.datums[framePart_datums_keys[index_of_axis2]]


# Make the sektch
sketch_transform2 = framePart.MakeSketchTransform(sketchPlane=frame_datum_plane2,
                                                    sketchUpEdge=frame_datum_axis2,
                                                    sketchPlaneSide=SIDE1,
                                                    sketchOrientation=LEFT,
                                                    origin=(0.0, 0.0, 1.5))
# We could also have used frame_datum_axis instead of frame_datum_axis2
# sketch_transform2 = framePart \
#                             .MakeSketchTransform(sketchPlane=frame_datum_plane2,
#                                                  sketchUpEdge=frame_datum_axis2,
#                                                  sketchPlaneSide=SIDE1,
#                                                  sketchOrientation=LEFT,
#                                                  origin=(0.0, 0.0, 1.5))
framePart_sketch2 = mdb.models['Beam Frame'] \
                                   .ConstrainedSketch(name='frame sketch 2',
                                                      sheetSize=20,
                                                      gridSpacing=1,
                                                      transform=sketch_transform2)

framePart_sketch2.Line(point1=(1.0,0.0), point2=(4.0,-3.0))
framePart_sketch2.Line(point1=(4.0,-3.0), point2=(6.0,-3.0))
framePart_sketch2.Line(point1=(6.0,-3.0), point2=(8.0,-3.0))
framePart_sketch2.Line(point1=(8.0,-3.0), point2=(10.0,-3.0))
framePart_sketch2.Line(point1=(10.0,-3.0), point2=(13.0,0.0))
framePart_sketch2.Line(point1=(13.0,0.0), point2=(8.0,0.0))
framePart_sketch2.Line(point1=(8.0,0.0), point2=(6.0,0.0))
framePart_sketch2.Line(point1=(6.0,0.0), point2=(1.0,0.0))
framePart_sketch2.Line(point1=(6.0,0.0), point2=(6.0,-3.0))
framePart_sketch2.Line(point1=(8.0,0.0), point2=(8.0,-3.0))


# Use the sketch to create a wire
framePart.Wire(sketchPlane=frame_datum_plane2, sketchUpEdge=frame_datum_axis2,
               sketchPlaneSide=SIDE1, sketchOrientation=LEFT,
               sketch=framePart_sketch2)
```

The above lines repeat the process in order to create the other frame. Other than the location of the points there is nothing new in them.

```
# -------------------------------------------------------------------------
# Create the cross bracing

# Start with a 3D Point Deformable Body
crossPart = beamModel.Part(name='CrossBracing', dimensionality=THREE_D,
                                                 type=DEFORMABLE_BODY)
crossPart.ReferencePoint(point=(0.0, 0.0, 0.0))
Once again the Part() and ReferencePoint() methods are used as was done for the
frame.
crossPart.DatumPointByCoordinate(coords=(1.0, 0.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(1.0, 0.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(4.0, -3.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(4.0, -3.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(6.0, 0.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(6.0, 0.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(6.0, -3.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(6.0, -3.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(8.0, 0.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(8.0, 0.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(8.0, -3.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(8.0, -3.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(10.0, -3.0, 0.0))
crossPart.DatumPointByCoordinate(coords=(10.0, -3.0, 1.5))
crossPart.DatumPointByCoordinate(coords=(13.0, 0.0, 0.0))

crossPart.DatumPointByCoordinate(coords=(13.0, 0.0, 1.5))
```

The **DatumPointByCoordinate()** method creates a **DatumPoint** object (a **Feature** object) at the coordinates specified by the required **coords** argument, which is a sequence of three Floats specifying the X, Y and Z coordinates.

```
crossPart_datums_keys = crossPart.datums.keys()
crossPart_datums_keys.sort()
```

The keys of the datum points are obtained using the **keys()** method and assigned to a variable. They are then sorted using the **sort()** method.

```
datum_points = crossPart.datums
```

The datum points of the cross bracing are all assigned to a variable which can be used in the next statement.

```
crossPart.WirePolyLine(points=((datum_points[crossPart_datums_keys[0]],
                                 datum_points[crossPart_datums_keys[1]]),
                                (datum_points[crossPart_datums_keys[2]],
                                 datum_points[crossPart_datums_keys[3]]),
                                (datum_points[crossPart_datums_keys[4]],
                                 datum_points[crossPart_datums_keys[5]]),
                                (datum_points[crossPart_datums_keys[6]],
                                 datum_points[crossPart_datums_keys[7]]),
                                (datum_points[crossPart_datums_keys[8]],
                                 datum_points[crossPart_datums_keys[9]]),
                                (datum_points[crossPart_datums_keys[10]],
                                 datum_points[crossPart_datums_keys[11]]),
                                (datum_points[crossPart_datums_keys[12]],
                                 datum_points[crossPart_datums_keys[13]]),
                                (datum_points[crossPart_datums_keys[14]],
                                 datum_points[crossPart_datums_keys[15]])),
                                 mergeWire=OFF, meshable=ON)
```

The **WirePolyLine()** method creates a **Feature** object consisting of wires. The points are provided using the **points** parameter and they are joined together in pairs. When this function is used at the **Part** level, as is the case here, each point can be a datum point, a reference point, a vertex or coordinates of a point.

9.4.4 Define the materials

The following block of code creates the material for the simulation

```
# -----------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs modulus
# and poissons ratio
beamMaterial = beamModel.Material(name='AISI 1005 Steel')
beamMaterial.Density(table=((7872, ),       ))
beamMaterial.Elastic(table=((200E9, 0.29), ))
```

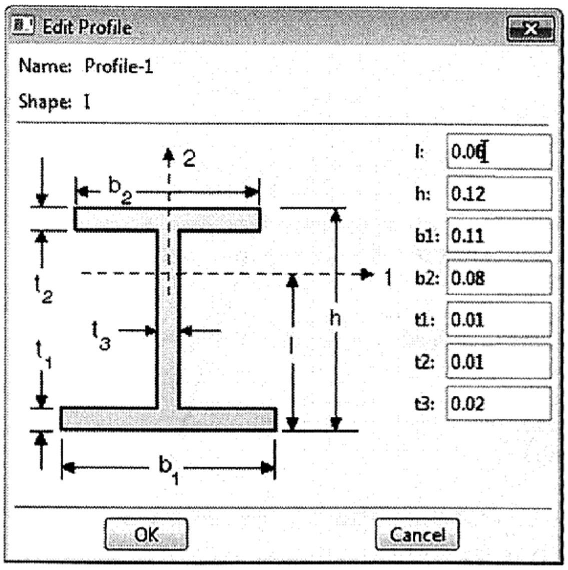
The statements are identical to those used in the Cantilever Beam example and were explained in section 4.3.4 on page 71.

9.4.5 Create profiles

The following block creates the beam profiles

```
# -----------------------------------------------------------------------
# Create profiles

beamModel.IProfile(name='FrameProfile', l=0.075, h=0.15, b1=0.12, b2=0.12,
                                                 t1=0.02, t2=0.02, t3=0.04)
beamModel.IProfile(name='CrossProfile', l=0.06, h=0.12, b1=0.11, b2=0.08,
                                                 t1=0.01, t2=0.01, t3=0.02)
```

The **IProfile()** method is used to specify that we have an I-beam, and to define its dimensions. The **name** parameter gives each profile a key for the repository, and **l**, **h**, **b1**, **b2**, **t1**, **t2** and **t3** correspond to the dimensions displayed in the GUI.



Name: Profile-1

Shape: I

l: 0.06

h: 0.12

b1: 0.11

b2: 0.08

t1: 0.01

t2: 0.01

t3: 0.02

9.4.6 Create sections and make section assignments

The following block creates the sections and assigns them to the frames and cross bracing

```
# ---------------------------------------------------------------------
# Create sections and assign frame and crosbracing to them

import section

frameSection = beamModel.BeamSection(name='Frame Section', profile='FrameProfile',
                                     integration=DURING_ANALYSIS,
                                     material='AISI 1005 Steel')
crossSection = beamModel.BeamSection(name='Cross Section', profile='CrossProfile',
                                     integration=DURING_ANALYSIS,
                                     material='AISI 1005 Steel')

edges_for_frame_section_assignment = framePart.edges.findAt(((3.5, 0.0, 0.0), ),
                                                            ((7.0, 0.0, 0.0), ),
                                                            ((10.5, 0.0, 0.0), ),
                                                            ((2.5, -1.5, 0.0), ),
                                                            ((5.0, -3.0, 0.0), ),
                                                            ((7.0, -3.0, 0.0), ),
                                                            ((9.0, -3.0, 0.0), ),
                                                            ((11.5, -1.5, 0.0), ),
                                                            ((6.0, -1.5, 0.0), ),
                                                            ((8.0, -1.5, 0.0), ),
                                                            ((3.5, 0.0, 1.5), ),
                                                            ((7.0, 0.0, 1.5), ),
                                                            ((10.5, 0.0, 1.5), ),
                                                            ((2.5, -1.5, 1.5), ),
                                                                  ((5.0, -3.0, 1.5), ),
                                                                  ((7.0, -3.0, 1.5), ),
                                                                  ((9.0, -3.0, 1.5), ),
                                                                  ((11.5, -1.5, 1.5), ),
                                                                  ((6.0, -1.5, 1.5), ),
                                                                  ((8.0, -1.5, 1.5), ),)

frame_region = regionToolset.Region(edges=edges_for_frame_section_assignment)
framePart.SectionAssignment(region=frame_region, sectionName='Frame Section')


edges_for_cross_section_assignment = crossPart.edges.findAt(((1.0, 0.0, 0.75), ),
                                                            ((6.0, 0.0, 0.75), ),
                                                            ((8.0, 0.0, 0.75), ),
                                                            ((13.0, 0.0, 0.75), ),
                                                            ((4.0, -3.0, 0.75), ),
                                                            ((6.0, -3.0, 0.75), ),
                                                            ((8.0, -3.0, 0.75), ),
                                                            ((10, -3.0, 0.75), ),)

cross_region = regionToolset.Region(edges=edges_for_cross_section_assignment)
crossPart.SectionAssignment(region=cross_region, sectionName='Cross Section')
```

```
import section
```

This statement imports the **section** module making its properties and methods accessible to the script.

```
frameSection = beamModel.BeamSection(name='Frame Section', profile='FrameProfile',
                                     integration=DURING_ANALYSIS,
                                     material='AISI 1005 Steel')
crossSection = beamModel.BeamSection(name='Cross Section', profile='CrossProfile',
                                     integration=DURING_ANALYSIS,
                                     material='AISI 1005 Steel')
```

These statements create **BeamSection** objects using the **BeamSection()** method. **BeamSection** objects are derived from the **Section** object which is defined in the **section** module. The first required parameter is **name**, a String which specifies the repository key. The second is **profile**, which is the name of an already defined beam profile. The third is **integration** which specifies the integration method for the section. The possible values are SymbolicConstants **DURING_ANALYSIS** and **BEFORE_ANALYSIS**. An optional parameter **material** specifies the name of the material being used.

```
edges_for_frame_section_assignment = framePart.edges.findAt(((3.5, 0.0, 0.0), ),
                                                            ((7.0, 0.0, 0.0), ),
                                                            ((10.5, 0.0, 0.0), ),
                                                            ((2.5, -1.5, 0.0), ),
                                                            ((5.0, -3.0, 0.0), ),
                                                            ((7.0, -3.0, 0.0), ),
                                                            ((9.0, -3.0, 0.0), ),
                                                            ((11.5, -1.5, 0.0), ),
                                                            ((6.0, -1.5, 0.0), ),
                                                            ((8.0, -1.5, 0.0), ),
                                                            ((3.5, 0.0, 1.5), ),
                                                            ((7.0, 0.0, 1.5), ),
                                                            ((10.5, 0.0, 1.5), ),
                                                            ((2.5, -1.5, 1.5), ),
                                                            ((5.0, -3.0, 1.5), ),
                                                            ((7.0, -3.0, 1.5), ),
                                                            ((9.0, -3.0, 1.5), ),
                                                            ((11.5, -1.5, 1.5), ),
                                                            ((6.0, -1.5, 1.5), ),
                                                            ((8.0, -1.5, 1.5), ),)
```

This statement uses the **findAt()** method to find any objects in the **EdgeArray** (basically edges) at the specified points or at a distance of less than 1E-6 from them. **framePart** is the part, **framePart.edges** exposes the **EdgeArray**, and **framePart.edges.findAt()** finds the edge in the **EdgeArray**. The coordinates used were obtained by drawing a rough sketch and determining the midpoints of each of the frame members.

```
frame_region = regionToolset.Region(edges=edges_for_frame_section_assignment)
```

This creates a **Region** object using the **Region()** method. The **Region()** method has no required arguments, but only optional ones such as **elements, nodes**, **vertices**, **edges**, **faces**, **cells** and a few more listed in the documentation. We use the **edges** argument, and assign it the edges obtained in the previous statement, which are the member elements of the frame. The **Region** object itself was discussed in section 4.3.5 of the Cantilever Beam example on page 73.

```
framePart.SectionAssignment(region=frame_region, sectionName='Frame Section')
```

This creates a **SectionAssignment** object using the **SectionAssignment()** method. It is almost identical to the one used in the Cantilever Beam example, Section 4.3.5 on page 73. The first parameter is the **Region** object created in the previous statement, and the second parameter is the name we wish to give the section, which is also its key in the sections repository.

```
edges_for_cross_section_assignment = crossPart.edges.findAt(((1.0, 0.0, 0.75), ),
                                                            ((6.0, 0.0, 0.75), ),
                                                            ((8.0, 0.0, 0.75), ),
                                                            ((13.0, 0.0, 0.75), ),
                                                            ((4.0, -3.0, 0.75), ),
                                                            ((6.0, -3.0, 0.75), ),
                                                            ((8.0, -3.0, 0.75), ),
                                                            ((10, -3.0, 0.75), ),)

cross_region = regionToolset.Region(edges=edges_for_cross_section_assignment)
crossPart.SectionAssignment(region=cross_region, sectionName='Cross Section')
```

The process of assigning sections is repeated for the cross bracing.

9.4.7 Assign section orientations

The following block assigns the orientation to the beam sections

```
# -----------------------------------------------------------------------
# Assign beam orientations

framePart.assignBeamSectionOrientation(region=frame_region, method=N1_COSINES,
                                                             n1=(0.0, 0.0, 1.0))

crossPart.assignBeamSectionOrientation(region=cross_region, method=N1_COSINES,
                                                             n1=(1.0, 0.0, 0.0))
```

The **assignBeamSectionOrientation()** method assigns a beam section orientation to a region of a part. The required argument **region** specifies the **Edge** objects. The variable **frame_region** was defined earlier when creating sections. The second required argument is **method**. This specifies the assignment method, and as of this writing the only value supported is the SymbolicConstant **N1_COSINES**. The third required argument is **n1**, which defines the local n1 direction of the beam profile using a sequence of three Floats.

9.4.8 Create an assembly

The following code block creates the assembly

```
# -----------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
beamAssembly = beamModel.rootAssembly

frameInstance = beamAssembly.Instance(name='Frame Instance', part=framePart,
                                                              dependent=ON)
crossInstance = beamAssembly.Instance(name='Cross Instance', part=crossPart,
                                                              dependent=ON)
```

These statements are almost identical to the ones used in the Cantilever Beam example. If you wish to refer back to them they are explained in Section 4.3.6 on page 74.

9.4.9 Create connectors using wire features

The following code block creates the wire features

```
# ----------------------------------------------------------------------
# Create the wire features

vertices_for_frame_side_1 = frameInstance.vertices.findAt(((1.0, 0.0, 0.0),),
                                                           ((4.0, -3.0, 0.0),),
                                                           ((6.0, -3.0, 0.0),),
                                                           ((10.0, -3.0, 0.0),),
                                                           ((13.0, 0.0, 0.0),),
                                                           ((6.0, 0.0, 0.0),),)

vertices_for_frame_side_2 = frameInstance.vertices.findAt(((1.0, 0.0, 1.5),),
                                                          ((4.0, -3.0, 1.5),),
                                                          ((6.0, -3.0, 1.5),),
                                                          ((10.0, -3.0, 1.5),),
                                                          ((13.0, 0.0, 1.5),),
                                                          ((6.0, 0.0, 1.5),),)

vertices_for_crossbars_side_1 =  crossInstance.vertices.findAt(((1.0, 0.0, 0.0),),
                                                               ((4.0, -3.0, 0.0),),
                                                               ((6.0, -3.0, 0.0),),
                                                               ((10.0, -3.0, 0.0),),
                                                               ((13.0, 0.0, 0.0),),
                                                               ((6.0, 0.0, 0.0),),)

vertices_for_crossbars_side_2 = crossInstance.vertices.findAt(((1.0, 0.0, 1.5),),
                                                              ((4.0, -3.0, 1.5),),
                                                              ((6.0, -3.0, 1.5),),
                                                              ((10.0, -3.0, 1.5),),
                                                              ((13.0, 0.0, 1.5),),
                                                              ((6.0, 0.0, 1.5),),)

# We need to create the connectors using these points
# The format of the WirePolyLine function is WirePolyLine(points=((v1[1], v2[1]),
# (v1[2], v2[2]), mergeWire=OFF, meshable=OFF)
# Notice that the points argument is a tuple of point pairs, in other words a
# tuple of tuples
# We first create the tuples of the point pairs specifying the individual
# connections such as (v1[1], v2[1]) and put them in a list and then create
# a tuple of those tuples from that list using the "tuple()" function

list_of_point_tuples = []
for i in range(len(vertices_for_frame_side_1)):
    list_of_point_tuples.append((vertices_for_frame_side_1[i],
                                    vertices_for_crossbars_side_1[i]))
    list_of_point_tuples.append((vertices_for_frame_side_2[i],
                                    vertices_for_crossbars_side_2[i]))

tuple_of_point_tuples = tuple(list_of_point_tuples)

beamAssembly.WirePolyLine(points=tuple_of_point_tuples, mergeWire=OFF,
                                                        meshable=OFF)

# ---------------------------------------------------------------------
# Assign these wire features/connectors to a set that can be used later
# This is the equivalent of checking off the "Create set of wires" checkbox in
# the "Create Wire Feature" window

# This of course requires that we first find the edges that are the wire
# connectors

edges_for_connector_set = beamAssembly.edges.findAt(((1.0, 0.0, 0.0),),
                                                     ((4.0, -3.0, 0.0),),
                                                     ((6.0, -3.0, 0.0),),
                                                     ((10.0, -3.0, 0.0),),
                                                     ((13.0, 0.0, 0.0),),
                                                     ((6.0, 0.0, 0.0),),
                                                     ((1.0, 0.0, 1.5),),
                                                     ((4.0, -3.0, 1.5),),
                                                     ((6.0, -3.0, 1.5),),
                                                     ((10.0, -3.0, 1.5),),
                                                     ((13.0, 0.0, 1.5),),
                                                     ((6.0, 0.0, 1.5),),)

# Now assign them to a set
beamAssembly.Set(edges=edges_for_connector_set, name='Set of connector wires')

# ---------------------------------------------------------------------
# Create a connector section

beamModel.ConnectorSection(name='FrameCrossConnSect', translationalType=JOIN)

# ---------------------------------------------------------------------
# Assign this connector section to the wire features using the set created earlier
conn_wire_region = beamAssembly.sets['Set of connector wires']
beamAssembly.SectionAssignment(sectionName='FrameCrossConnSect',
                               region=conn_wire_region)
```

```
vertices_for_frame_side_1 = frameInstance.vertices.findAt(((1.0, 0.0, 0.0),),
                                                          ((4.0, -3.0, 0.0),),
                                                          ((6.0, -3.0, 0.0),),
                                                          ((10.0, -3.0, 0.0),),
                                                          ((13.0, 0.0, 0.0),),
                                                          ((6.0, 0.0, 0.0),),)
```

The vertices for the wire features need to be found and stored so that they can be used to create the wire features. We need the vertices on both the frames, and both sides of the cross bracing where it meets each frame. We start by storing the vertices of one frame in the variable **vertices_for_frame_side_1**. The syntax **frameInstance.vertices** exposes the vertices array for **frameInstance**, and **findAt()** is used to find the vertices supplied as arguments, which are the coordinates for the nodes of the frame.

```
vertices_for_frame_side_2 = frameInstance.vertices.findAt(((1.0, 0.0, 1.5),),
                                                          ((4.0, -3.0, 1.5),),
                                                          ((6.0, -3.0, 1.5),),
                                                          ((10.0, -3.0, 1.5),),
                                                          ((13.0, 0.0, 1.5),),
                                                          ((6.0, 0.0, 1.5),),)

vertices_for_crossbars_side_1 =  crossInstance.vertices.findAt(((1.0, 0.0, 0.0),),
                                                               ((4.0, -3.0, 0.0),),
                                                               ((6.0, -3.0, 0.0),),
                                                               ((10.0, -3.0, 0.0),),
                                                               ((13.0, 0.0, 0.0),),
                                                               ((6.0, 0.0, 0.0),),)

vertices_for_crossbars_side_2 = crossInstance.vertices.findAt(((1.0, 0.0, 1.5),),
                                                              ((4.0, -3.0, 1.5),),
                                                              ((6.0, -3.0, 1.5),),
                                                              ((10.0, -3.0, 1.5),),
                                                              ((13.0, 0.0, 1.5),),
                                                              ((6.0, 0.0, 1.5),),)
```

Similarly the nodes of the other frame, and those on both sides of the cross bracing are found.

```
list_of_point_tuples = []
for i in range(len(vertices_for_frame_side_1)):
    list_of_point_tuples.append((vertices_for_frame_side_1[i],
                                 vertices_for_crossbars_side_1[i]))
    list_of_point_tuples.append((vertices_for_frame_side_2[i],
                                 vertices_for_crossbars_side_2[i]))

tuple_of_point_tuples = tuple(list_of_point_tuples)

beamAssembly.WirePolyLine(points=tuple_of_point_tuples, mergeWire=OFF,
                                                         meshable=OFF)
```

These statements are best explained together. We are trying to create the tuple which will later be passed to the **WirePolyLine()** function to create the wire features. The **WirePolyLine()** method expects the argument to be in the format:

```
WirePolyLine(points=((v1[1], v2[1]), (v1[2], v2[2]), mergeWire=OFF, meshable=OFF)
```

Here v1[1] is the first point (a tuple of x, y, z coordinate for one point). (v1[1], v2[1]) is a tuple of two such tuples – the two points which will have a wire feature between them. And ((v1[1], v2[1]),(v1[2], v2[2]))is a tuple of these tuples. So we need to put our points in this format.

The for-loop allows us to do this. It loops from 0 to one less than the number of coordinates which is obtained using the **len()** function. In each iteration it appends to a list called **list_of_point_tuples** a new tuple consisting of start and end coordinates of a wire feature. In fact it does this twice at each iteration, one for each side of the cross bracing.

Once this list has been created, it is converted into a tuple using Python's **tuple()** function which can take a sequence or list and convert it into a tuple.

This tuple can then be used as the **points** argument for the **WirePolyLine()** method. The **WirePolyLine()** method creates **Feature** objects by creating a series of wires joining points provided in pairs. When this method is used at the **Assembly** level, the points provided must be vertices, reference points or ophan mesh nodes. We have provided vertices in our script. The only required argument is **points**, which as stated before is a tuple of point pairs, where each pair is a tuple. The other 2 arguments supplied are optional. **mergeWire** specifies whether to merge the wire with existing geometry such as faces and solid regions, and the default is ON. We do not wish for the connecting wires to be merged into the parts hence we set it to OFF. **meshable** specifies whether the wire can be selected in meshing operations and the default value is ON. If we set it to OFF, as we have done here, the wire can be used for connector section assignment which is exactly what we intend to do with it.

```
# -----------------------------------------------------------------------
# Assign these wire features/connectors to a set that can be used later

edges_for_connector_set = beamAssembly.edges.findAt(((1.0, 0.0, 0.0),),
                                                    ((4.0, -3.0, 0.0),),
                                                    ((6.0, -3.0, 0.0),),
                                                    ((10.0, -3.0, 0.0),),
                                                    ((13.0, 0.0, 0.0),),
                                                    ((6.0, 0.0, 0.0),),
                                                    ((1.0, 0.0, 1.5),),
                                                    ((4.0, -3.0, 1.5),),
                                                    ((6.0, -3.0, 1.5),),
                                                                  ((10.0, -3.0, 1.5),),
                                                                  ((13.0, 0.0, 1.5),),
                                                                  ((6.0, 0.0, 1.5),),)

# Now assign them to a set
beamAssembly.Set(edges=edges_for_connector_set, name='Set of connector wires')
```

When creating wire features in Abaqus/CAE, you check off the **Create set of wires** option in the **Create Wire Feature** window. In the above lines we try to replicate this through the script by finding all the edges/wires we just created using **beamAssembly.edges.findAt()**. We then assign them to a set using the **Set()** method. You have already encountered the **Set()** method used in section 8.3.2 on page 161. The difference in that example was that the Set was created using vertices whereas here it is created using edges. To refresh your memory, the **Set()** method creates a Set object in the assembly. Its first argument, **edges**, is an optional argument. In place of edges you might have used **nodes**, **elements**, **vertices**, **faces**, **cells**, among other possible arguments (all of which are listed in the documentation). Since we are using **edges**, we provide a list of edges stored previously in **edges_for_connector_set**. The second argument, **name**, is a required parameter. It is a String which is the name of the set and its key in the repository.

```
# -------------------------------------------------------------------------
# Create a connector section

beamModel.ConnectorSection(name='FrameCrossConnSect', translationalType=JOIN)
```

The **ConnectorSection()** method is used to create a **ConnectorSection** object. A **ConnectorSection** object is derived from the **Section** object and describes the connection type and behavior of a connector. The **ConnectorSection()** method has a required method, **name**, which is a String specifying the repository key. There are a number of optional arguments such as **assembledType**, **rotationalType**, **translationalType**, and so on which are described in the documentation, but at least one of these 3 must be specified. The one used here is **translationalType** with which you specify the translational connection type. The possible values are all SymbolicConstants, and there are number of them listed in the documentation. The one used here is **JOIN** which constrains the translation along all 3 axes to be equal for both nodes connected by the connector section.

```
# -------------------------------------------------------------------------
# Assign this connector section to the wire features using the set created earlier
conn_wire_region = beamAssembly.sets['Set of connector wires']
```

```
beamAssembly.SectionAssignment(sectionName='FrameCrossConnSect',
                               region=conn_wire_region)
```

We first assign the set 'Set of connector wires' to a variable **conn_wire_region**. A set of edges can be used as a **Region** object which we will need in for the **SectionAssignment()** method, hence I've put the word region in the name of the variable to make things clear. You have seen the **SectionAssignment()** method used in section 4.3.5 on page 72. To refresh your memory, the **SectionAssignment()** method creates a **SectionAssignment** object, which is an object that is used to assign sections to a part, an assembly or an instance. Its first parameter is a **region**, in this case the region is **conn_wire_region**. The second argument **sectionName** is the name we wish to give the section, which is also the key in the of the sections dictionary/repository. This argument must be a String.

9.4.10 Use constraint equations for two nodes

The following code block implements constraint equations

```
# -----------------------------------------------------------------------
# Use constraint equations on the other two nodes

# We did not apply the JOIN condition to four of the nodes
# We will instead use an equation constraint to achieve the same effect on the
# top two
# However we won't use the equation constraint on the lower two because we are
# going to fix that edge anyway and having an equation constraint as well as a
# fixed boundary condition might give an error

# First we need to assign the nodes, both on the frame part and on the crossbars,
# to sets

vertex_for_framenode_1 = frameInstance.vertices.findAt(((8.0, 0.0, 0.0),),)
beamAssembly.Set(vertices=vertex_for_framenode_1, name='framenode1')
vertex_for_framenode_2 = frameInstance.vertices.findAt(((8.0, 0.0, 1.5),),)
beamAssembly.Set(vertices=vertex_for_framenode_2, name='framenode2')
vertex_for_crossnode_1 = crossInstance.vertices.findAt(((8.0, 0.0, 0.0),),)
beamAssembly.Set(vertices=vertex_for_crossnode_1, name='crossnode1')
vertex_for_crossnode_2 = crossInstance.vertices.findAt(((8.0, 0.0, 1.5),),)
beamAssembly.Set(vertices=vertex_for_crossnode_2, name='crossnode2')

# Create the equation constraints
beamModel.Equation(name='JoinConstraint1', terms=((1.0, 'crossnode1', 1),
                                                  (-1.0, 'framenode1', 1)))
beamModel.Equation(name='JoinConstraint2', terms=((1.0, 'crossnode1', 2),
                                                  (-1.0, 'framenode1', 2)))
beamModel.Equation(name='JoinConstraint3', terms=((1.0, 'crossnode1', 3),
                                                  (-1.0, 'framenode1', 3)))
beamModel.Equation(name='JoinConstraint4', terms=((1.0, 'crossnode2', 1),
                                                (-1.0, 'framenode2', 1)))
beamModel.Equation(name='JoinConstraint5', terms=((1.0, 'crossnode2', 2),
                                                (-1.0, 'framenode2', 2)))
beamModel.Equation(name='JoinConstraint6', terms=((1.0, 'crossnode2', 3),
                                                (-1.0, 'framenode2', 3)))
```

```
# First we need to assign the nodes, both on the frame part and on the crossbars,
# to sets

vertex_for_framenode_1 = frameInstance.vertices.findAt(((8.0, 0.0, 0.0),),)
beamAssembly.Set(vertices=vertex_for_framenode_1, name='framenode1')
vertex_for_framenode_2 = frameInstance.vertices.findAt(((8.0, 0.0, 1.5),),)
beamAssembly.Set(vertices=vertex_for_framenode_2, name='framenode2')
vertex_for_crossnode_1 = crossInstance.vertices.findAt(((8.0, 0.0, 0.0),),)
beamAssembly.Set(vertices=vertex_for_crossnode_1, name='crossnode1')
vertex_for_crossnode_2 = crossInstance.vertices.findAt(((8.0, 0.0, 1.5),),)
beamAssembly.Set(vertices=vertex_for_crossnode_2, name='crossnode2')
```

Sets must be created to represent the nodes so that they can be used as parameters for the **Equation()** method. The **findAt()** method is used to find the vertices on their respective part instances as **[partInstance].vertices.findAt()**. These are then assigned to sets using the **Set()** method which was explained in the previous section.

```
# Create the equation constraints
beamModel.Equation(name='JoinConstraint1', terms=((1.0, 'crossnode1', 1),
                                                (-1.0, 'framenode1', 1)))
beamModel.Equation(name='JoinConstraint2', terms=((1.0, 'crossnode1', 2),
                                                (-1.0, 'framenode1', 2)))
beamModel.Equation(name='JoinConstraint3', terms=((1.0, 'crossnode1', 3),
                                                (-1.0, 'framenode1', 3)))
beamModel.Equation(name='JoinConstraint4', terms=((1.0, 'crossnode2', 1),
                                                (-1.0, 'framenode2', 1)))
beamModel.Equation(name='JoinConstraint5', terms=((1.0, 'crossnode2', 2),
                                                (-1.0, 'framenode2', 2)))
beamModel.Equation(name='JoinConstraint6', terms=((1.0, 'crossnode2', 3),
                                                (-1.0, 'framenode2', 3)))
```

The **Equation()** method creates an **Equation** object, which is derived from the **Constraint** object. The **Equation** object defines a linear multi-point constraint between a set of degrees of freedom. It has 2 required arguments. The first is **name** which is a String specifying the repository key of the constraint. The second is **terms**, which is a sequence of Float, String, Int and Int specifying a coefficient, a set name, a degree of freedom and a coordinate system ID. The last one is optional and has not been used here. The actual numbers are the ones you would type into the **Edit Constraint** window in Abaqus/CAE.

9.4.11 Create steps

The following code block creates the steps

```
# ------------------------------------------------------------------------
# Create the step

import step

# Create a static general step
beamModel.StaticStep(name='Apply Loads', previous='Initial',
                                    description='Loads are applied in this step')
```

These statements are the same as the ones used in the Cantilever Beam example hence require no explanation. You may refer to in Section 4.3.7 on page 75.

9.4.12 Create and define field output requests

No field output requests were defined in this simulation

```
# ------------------------------------------------------------------------
# Field output requests

# Leave at defaults
```

9.4.13 Create and define history output requests

No history output requests were defined in this simulation

```
# ------------------------------------------------------------------------
# History output request

# Leave at defaults
```

9.4.14 Apply loads

The following block applies loads

```
# ------------------------------------------------------------------------
# Apply loads

edge_for_crossload1 = crossInstance.edges.findAt(((6.0, 0.0, 0.75), ),)
region_for_crossload1 = regionToolset.Region(edges=edge_for_crossload1)
edge_for_crossload2 = crossInstance.edges.findAt(((8.0, 0.0, 0.75), ),)
region_for_crossload2 = regionToolset.Region(edges=edge_for_crossload2)
edge_for_frameload1 = frameInstance.edges.findAt(((3.5, 0.0, 0.0), ),)
region_for_frameload1 = regionToolset.Region(edges=edge_for_frameload1)
edge_for_frameload2 = frameInstance.edges.findAt(((10.5, 0.0, 1.5), ),)
```

```
region_for_frameload2 = regionToolset.Region(edges=edge_for_frameload2)

beamModel.LineLoad(name='CrossLoad1', createStepName='Apply Loads',
                   region=region_for_crossload1, comp2=-1000.0)
beamModel.LineLoad(name='CrossLoad2', createStepName='Apply Loads',
                   region=region_for_crossload2, comp2=-1000.0)
beamModel.LineLoad(name='FrameLoad1', createStepName='Apply Loads',
                   region=region_for_frameload1, comp2=-1500.0)
beamModel.LineLoad(name='FrameLoad2', createStepName='Apply Loads',
                   region=region_for_frameload2, comp2=-500.0)
```

```
edge_for_crossload1 = crossInstance.edges.findAt(((6.0, 0.0, 0.75), ),)
```

The edges on which the line load will be applied are selected using the **findAt()** method as **[partInstance].edges.findAt()** and assigned to variables such as **edge_for_crossload1**.

```
region_for_crossload1 = regionToolset.Region(edges=edge_for_crossload1)
```

This selects a region using the **Edge** object we have just created. It uses the **Region()** method to create a **Region** object. You have seen the **Region()** method previously in Sections 4.3.10 and 4.3.11. To refresh your memory, the **Region()** method works a little differently depending on the kind of arguments you give it. In Section 4.3.10 we saw it return a surface like region and in Section 4.3.11 it returned a set-like region. Since a line-load is applied on the set of points and not a surface, we need **Region()** to return a set-like region. Hence the argument we use is **edges**, which requires a sequence of **Edge** objects. Remember that **findAt()** will return an **Edge** object or a sequence of **Edge** objects, hence **edge_for_crossload1** is valid argument here even though you might have thought of it as an **Edge** object as opposed to a sequence of **Edge** objects. The **Region** object is defined in the **regionToolset** module, which is why we used the '*import regionToolset*' statement in the initialization section of our script.

```
beamModel.LineLoad(name='CrossLoad1', createStepName='Apply Loads',
                   region=region_for_crossload1, comp2=-1000.0)
```

The **LineLoad()** method is used to create a **LineLoad** object. The **LineLoad** object, which is derived from the **Load** object, stores the data of an applied line load. It has 3 required arguments. The first is **name**, which is a String specifying the load repository key. We have named it 'CrossLoad1'. The second is **createStepName**, which is a String specifying the name of the step in which the load is to be created. We want our load to be created in the ‘Apply Loads’ step which was defined previously. The third is **region** which is a **Region** object specifying the region on which the load is applied. We supply **region_for_crossload1** which was created in the previous statement. One of the optional arguments is **distributionType**, whose possible values are the SymbolicConstants **UNIFORM**, **USER_DEFINED** and **FIELD**. It specifies how the load is distributed spatially. Since we have not defined it, it defaults to **UNIFORM**. Some other optional arguments are **comp1**, **comp2** and **comp3**. We have defined **comp3**, which is a Float specifying the component of the load in the global 3-direction. Even though **comp1**, **comp2** and **comp3** are optional arguments, at least one of them must be nonzero unless **distributionType** is set to **USER_DEFINED**. Other optional arguments are listed in the documentation.

9.4.15 Apply boundary conditions

The following block applies the boundary conditions

```
# -------------------------------------------------------------------
# Apply boundary conditions

frame_edges_for_bc = frameInstance.edges.findAt(((5.0,-3.0,0.0),),
                                                ((7.0,-3.0,0.0),),
                                                ((9.0,-3.0,0.0),),
                                                ((5.0,-3.0,1.5),),
                                                ((7.0,-3.0,1.5),),
                                                ((9.0,-3.0,1.5),),)

cross_edges_for_bc = crossInstance.edges.findAt(((4.0,-3.0,0.75),),
                                                ((6.0,-3.0,0.75),),
                                                ((8.0,-3.0,0.75),),
                                                ((10.0,-3.0,0.75),),)

edges_for_bc = frame_edges_for_bc + cross_edges_for_bc
bc_region = regionToolset.Region(edges=edges_for_bc)

beamModel.DisplacementBC(name='FixBottom', createStepName='Initial',
                         region=bc_region, u1=SET, u2=SET, u3=SET,
                         ur1=UNSET, ur2= UNSET, ur3=UNSET,
                         amplitude=UNSET, distributionType=UNIFORM,
                         fieldName='', localCsys=None)
```

```
frame_edges_for_bc = frameInstance.edges.findAt(((5.0,-3.0,0.0),),
                                                ((7.0,-3.0,0.0),),
                                                ((9.0,-3.0,0.0),),
                                                ((5.0,-3.0,1.5),),
                                                ((7.0,-3.0,1.5),),
                                                ((9.0,-3.0,1.5),),)
```

This statement uses the **findAt()** method to locate all the edges of the frame instance which we will constrain.

```
cross_edges_for_bc = crossInstance.edges.findAt(((4.0,-3.0,0.75),),
                                                ((6.0,-3.0,0.75),),
                                                ((8.0,-3.0,0.75),),
                                                ((10.0,-3.0,0.75),),)
```

This statement uses the **findAt()** method to locate all the edges of the cross bracing instance which we will constrain.

```
edges_for_bc = frame_edges_for_bc + cross_edges_for_bc
```

We then combine all the edges and place them in one variable by using the addition + sign. This syntax may come as a surprise to you, but it is perfectly legal and works very well in Abaqus. The **findAt()** methods of the two previous statements returned sequences of **Edge** objects, and this statement adds both sequences together to create another sequence of **Edge** objects.

```
bc_region = regionToolset.Region(edges=edges_for_bc)
```

We once again use the **Region()** method in the same manner as in the previous section to create a **Region** object using the edges. This **Region** object can then be used in the **DisplacementBC()** method.

```
beamModel.DisplacementBC(name='FixBottom', createStepName='Initial',
                         region=bc_region, u1=SET, u2=SET, u3=SET,
                         ur1=UNSET, ur2= UNSET, ur3=UNSET,
                         amplitude=UNSET, distributionType=UNIFORM,
                         fieldName='', localCsys=None)
```

This statement creates a **DisplacementBC** object which you encountered earlier in section 7.4.11. To jog your memory, it stores the data for a displacement/rotation. The **DisplacementBC** object is derived from the **BoundaryCondition** object. The first required argument is a String for the name. The second is the name/key of the step in which the boundary condition is to be applied. In this case we apply it to the 'Initial' step. The third argument must be a **Region** object. The remaining arguments are optional. Note however that even though **u1**, **u2**, **u3**, **ur1**, **ur2** and **ur3** are optional arguments, at least one of them must be specified. For **u1**, **u2** and **u3** we use the SymbolicConstant **SET** thus preventing translation in the 1, 2 and 3 directions (a.k.a. x, y and z directions). **Ur1**, **ur2** and **ur3** are not specified, and will default to **UNSET** thus allowing rotation along all 3 axes. Since no amplitude is used, it is set to **UNSET** and the **distributionType** is set to **UNIFORM** ensuring a uniform special distribution of the boundary condition in the applied region.

9.4.16 Mesh

The following code block creates the mesh

```
# -----------------------------------------------------------------------
# Create the mesh

import mesh

frame_mesh_region = frame_region
frame_edges_for_meshing = edges_for_frame_section_assignment
frame_mesh_element_type=mesh.ElemType(elemCode=B31, elemLibrary=STANDARD)
framePart.setElementType(regions=frame_mesh_region,
                         elemTypes=(frame_mesh_element_type, ))
framePart.seedEdgeByNumber(edges=frame_edges_for_meshing, number=4)
framePart.generateMesh()

cross_mesh_region = cross_region
cross_edges_for_meshing = edges_for_cross_section_assignment
cross_mesh_element_type=mesh.ElemType(elemCode=B31, elemLibrary=STANDARD)
crossPart.setElementType(regions=cross_mesh_region,
                         elemTypes=(cross_mesh_element_type, ))
crossPart.seedEdgeByNumber(edges=cross_edges_for_meshing, number=4)
crossPart.generateMesh()
```

All of these statements are similar to the ones used in section 7.4.12 on page 139. Refer back to that section to refresh your memory as the methods used are identical. The only differences are in the element type used and the number of seeds per edge. Also since we have 2 part instances in this example, they have both been meshed separately.

9.4.17 Create and run the job

The following code runs the job

```
# -----------------------------------------------------------------------
# Create and run the job

import job

# Create the job
mdb.Job(name='BeamFrameAnalysisJob', model='Beam Frame', type=ANALYSIS,
explicitPrecision=SINGLE,
    nodalOutputPrecision=SINGLE, description='Bending of loaded beam frame',
    parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
    numDomains=1, userSubroutine='', numCpus=1, memory=50, memoryUnits=PERCENTAGE,
scratch='',
    echoPrint=OFF, modelPrint=OFF, contactPrint=OFF, historyPrint=OFF)

# Run the job
mdb.jobs['BeamFrameAnalysisJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['BeamFrameAnalysisJob'].waitForCompletion()

# End of run job
```

All of this should look familiar to you from the Cantilever Beam example. You may refer back to section 4.3.13, page **88** for a refresher on these job commands

9.5 Summary

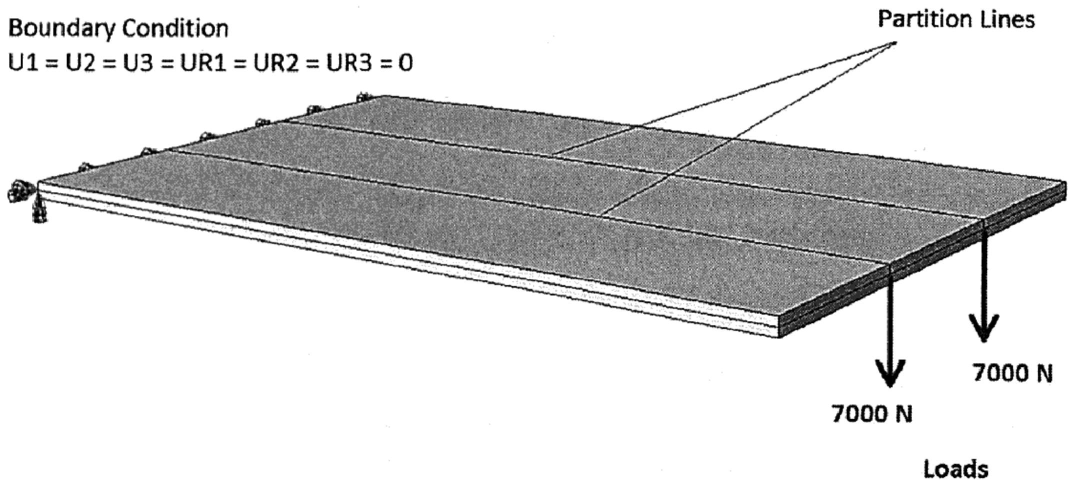
Some of the new topics covered in this chapter included creating datum planes and datum lines using a script. We also created connectors and constraint equations to simulate joints. You created a line load by using the **Region()** method a little differently to return a set-based region as opposed to a surface based one. These build on your knowledge of Python scripting in Abaqus.

10

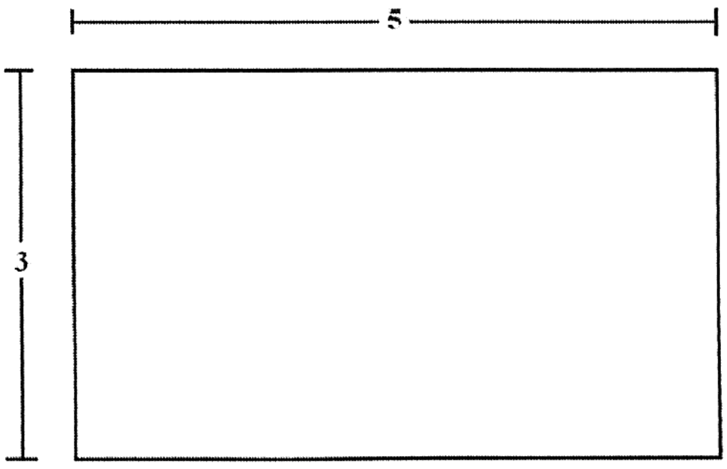
Bending of a Planar Shell (Plate)

10.1 Introduction

In this chapter we will perform a static analysis on a plate being bent by a concentrated force. The problem is displayed in the figure.



The dimensions are displayed in the following figure. All lengths are in meters and the shell thickness is 0.1 m.



In this example the following tasks will be demonstrated first using Abaqus/CAE, and then using a Python script.

- Create a part
- Assign materials
- Assign sections
- Create an Assembly
- Create a static, general step
- Request field outputs
- Delete history outputs
- Create datum points and partition faces
- Assign loads
- Assign boundary conditions
- Create a mesh
- Create and submit a job
- Report field outputs to an external file

The following new topics are covered in this example:

- Model / Preprocessing
    - Work in 3D with a planar shell

    - Create sections of type 'shell', specify section integration properties and assign shell thickness
    - Define shell offset when assigning sections
    - Turn NLGEOM (non-linear geometry) option on/off as required
    - Delete history outputs
    - Create partitions for the purpose of generating selectable nodess
- Results / Post-processing
    - Show element labels on meshed model
    - Change the sort variable and sort order in the report profile
    - View/Change the work directory

10.2 Procedure in GUI

You can perform the simulation in Abaqus/CAE by following the steps listed below. You can either read through these, or watch the video demonstrating the process on the book website.

1. Rename **Model-1** to **Plate Bending Model**
    a. Right-click on Model-1 in the Model Database
    b. Choose **Rename..**
    c. Change name to **Blate Bending Model**
2. Create the part
    a. Double-click on **Parts** in Model Database. **Create Part** window is displayed.
    b. Set **Name** to **Plate**
    c. Set **Modeling Space** to **3D**
    d. Set **Type** to **Deformable**
    e. Set **Base Feature Shape** to **Shell**
    f. Set **Base Feature Type** to **Planar**
    g. Set **Approximate Size** to **20**
    h. Click **OK**. You will enter Sketcher mode.
3. Sketch the plate
    a. Use the **Create Lines:Rectangle (4 lines)** tool to draw the profile of the plate. Start at the origin and drag to the top and left so that the rectangle has positive X and Y coordinates.
    b. Use the **Add Dimension** tool to set the length of the horizontal elements to 5 m and the length of the vertical elements to 3 m.

   c. Click **Done** to exit the sketcher.
4. Create the material
   a. Double-click on **Materials** in the Model Database. **Edit Material** window is displayed
   b. Set **Name** to **AISI 1005 Steel**
   c. Select **General > Density**. Set **Mass Density** to **7872** (which is 7.872 g/cc)
   d. Select **Mechanical > Elasticity > Elastic**. Set **Young's Modulus** to **200E9** (which is 200 GPa) and **Poisson's Ratio** to **0.29**.
5. Create sections
   a. Double-click on **Sections** in the Model Database. **Create Section** window is displayed
   b. Set **Name** to **Plate Section**
   c. Set **Category** to **Shell**
   d. Set **Type** to **Homogeneous**
   e. Click **Continue...** The **Edit Section** window is displayed.
   f. In the **Basic** tab , set **Section integration** to **During Analysis**
   g. Set **Shell thickness Value** to **0.1**
   h. Set **Material** to **the AISI 1005 Steel** which was defined in the material creation step.
   i. Set **Thickness integration rule** to **Simpson**
   j. Click **OK**.
6. Assign the section to the plate
   a. Expand the **Parts** container in the Model Database. Expand the part **Plate.**
   b. Double-click on **Section Assignments**
   c. You see the message **Select the regions to be assigned a section** displayed below the viewport
   d. Click and drag with the mouse to select the entire plate.
   e. Click **Done.** The **Edit Section Assignment** window is displayed.
   f. Set **Section** to **Plate Section.**
   g. Set **Shell Offset Definition** to **Middle surface.**
   h. Click **OK.**
7. Create the Assembly
   a. Double-click on **Assembly** in the Model Database. The viewport changes to the **Assembly Module.**
   b. Expand the **Assembly** container.
   c. Double-click on **Instances**. The **Create Instance** window is displayed.

   d. Set **Parts** to **Plate**
   e. Set **Instance Type** to **Dependent (mesh on part)**
   f. Click **OK**.
8. Create Steps
   a. Double-click on **Steps** in the Model Database. The **Create Step** window is displayed.
   b. Set **Name** to **Load Step**
   c. Set **Insert New Step After** to **Initial**
   d. Set **Procedure Type** to **General > Static, General**
   e. Click **Continue..** The **Edit Step** window is displayed
   f. In the **Basic** tab, set **Description** to **Apply concentrated forces in this step.**
   g. Set **Time period** to **1**
   h. Set **Nlgeom** to **On**
   i. Click **OK**.
9. Request Field Outputs
   a. Expand the **Field Output Requests** container in the Model Database.
   b. Right-click on **F-Output-1** and choose **Rename...**
   c. Change the name to **Output Stresses and Displacements**
   d. Double-click on **Output Stresses and Displacements** in the Model Database. The **Edit Field Output Request** window is displayed.
   e. Select the desired variables by checking them off in the **Output Variables** list. The variables we want are **S (stress components and invariants)** and **U (translations and rotations)**. Uncheck the rest. You will notice that the text box above the output variable list displays **S,U**
   f. Click **OK**.
10. Delete History Outputs
    a. Expand the **History Output Requests** container in the Model Database.
    b. Right-click on **H-Output-1** and choose **Delete...**
    c. You see a prompt **OK to delete "H-Output-1"?** Click **Yes**.
11. Apply boundary conditions
    a. Double-click on **BCs** in the Model Database. The **Create Boundary Condition** window is displayed
    b. Set **Name** to **Fix Edge**
    c. Set **Step** to **Initial**
    d. Set **Category** to **Mechanical**
    e. Set **Types for Selected Step** to **Displacement/Rotation**

f. Click **Continue...**
g. You see the message **Select regions for the boundary condition** displayed below the viewport
h. Select the left edge of the plate.
i. Click **Done**. The **Edit Boundary Condition** window is displayed.
j. Check off **U1, U2, U3, UR1, UR2** and **UR3**. This will fix the edge and not allow translation or rotation.
k. Click **OK**.

12. Partition the plate to create points for the concentrated loads
    a. Expand the **Parts** container in the model tree.
    b. Double-click the part **Plate**. The viewport changes to the Part module and plate part.
    c. Click the **Create Datum Point: Enter coordinates** tool. You see the prompt **Coordinates for datum point (X, Y, Z):** below the viewport
    d. Type in the coordinates **0.0, 2.0, 0.0** and press the "Enter" key on your keyboard. You see a datum point appear on the left edge of the plate in the viewport. You again see the prompt **Coordinates for datum point (X, Y, Z):** below the viewport
    e. Type in the coordinates **0.0, 1.0, 0.0** and press the "Enter" key on your keyboard.. You see another datum point appear on the left edge of the plate in the viewport.
    f. Similarly proceed to enter in the coordinates of the next 2 datum points which are **5.0, 2.0, 0.0** and **5.0, 1.0, 0.0** respectively. There are now 4 datum points, 2 on the left edge and 2 on the right edge of the plate.
    g. Click the **Partition Face: Use Shortest Path Between 2 Points** tool. You see the message **Select a start point** below the viewport
    h. Click on the top left datum point (whose coordinates are **0.0, 2.0, 0.0**). You see the message **Select an end point** below the viewport.
    i. Click on the top right datum point (whose coordinates are **5.0, 2.0, 0.0**). You see the message **Partition definition complete** below the viewport
    j. Click on the **Create Partition** button. The partition is displayed in the viewport.
    k. You see the prompt **Select the faces to partition** below the viewport. Use the drop down to set it to individually.
    l. Hover the mouse over the lower half of the plate (below the partition line). It will light up as you are hoving over it. Click it to select it.

- m. Click **Done.** You see the message **Select a start point** below the viewport
- n. Click on the bottom left datum point (whose coordinates are **0.0, 1.0, 0.0**). You see the message **Select an end point** below the viewport
- o. Click on the bottom right datum point (whose coordinates are **5.0, 1.0, 0.0**). You see the message **Partition definition complete** below the viewport
- p. Click on the **Create Partition** button. The second partition is displayed in the viewport and the plate now consists of 3 different partitions.
- q. Click **Done**

13. Assign Loads
    - a. Double-click on **Loads in** the Model Database. The **Create Load** window is displayed
    - b. Set **Name** to **Concentrated Forces**
    - c. Set **Step** to **Load Step**
    - d. Set **Category** to **Mechanical**
    - e. Set **Type for Selected Step** to **Concentrated force**
    - f. Click **Continue...**
    - g. You see the message **Select points for the load**displayed below the viewport
    - h. Select the two points on the right edge where the partition line meets the edge. The reason for creating the partitions was to be able to select these two points. Hold the "Shift" key on your keyboard to select both points.
    - i. Click **Done**. The **Edit Load** window is displayed
    - j. Set **CF3** to **-7000.0** to apply a 7000 N force in downward (negative Y) direction
    - k. Click **OK**
    - l. You will see the forces displayed with an arrows in the viewport on the selected points although you may need to rotate the view to see them clearly
14. Create the mesh
    - a. Expand the **Parts** container in the Model Database.
    - b. Expand **Plate**
    - c. Double-click on **Mesh (Empty)**. The viewport window changes to the **Mesh module** and the tools in the toolbar are now meshing tools.
    - d. Using the menu bar click on **Mesh > Element Type ...**
    - e. You see the message **Select the regions to be assigned element types** displayed below the viewport
    - f. Click and drag using your mouse to select the entire plate.
    - g. Click **Done**. The **Element Type** window is displayed.

h. Set **Element Library** to **Standard**
i. Set **Geometric Order** to **Quadratic**
j. Set **Family** to **Shell**
k. You will notice the message **S8R: An 8-node doubly curved thick shell, reduced integration**
l. Click **OK**
m. Click **Done**
n. Using the menu bar lick on **Seed > Edge by Number**
o. You see the message **Select the regions to be assigned local seeds** displayed below the viewport
p. Click on the 6 vertical edges (3 on left edge and 3 on right edge). You will need to press the 'Shift' key on your keyboard to select all 6 of them
q. Click **Done**. You see the prompt **Number of elements along the edges** displayed below the viewport
r. Set it to **3** and press the "Enter" key on your keyboard.
s. Again you see the message **Select the regions to be assigned local seeds** displayed below the viewport
t. Click on the 4 horizontal edges (top edge, bottom edge and 2 partition lines). You will need to press the 'Shift' key on your keyboard to select all 4 of them
u. Click **Done**. You see the prompt **Number of elements along the edges** displayed below the viewport
v. Set it to **10** and press the 'Enter' key on your keyboard
w. Click **Done**
x. Using the menu bar click on **Mesh > Part**
y. You see the prompt **OK to mesh the part?** displayed below the viewport
z. Click **Yes**. The meshed plate appears in the viewport.

15. Create and submit the job
    a. Double-click on **Jobs** in the Model Database. The **Create Job** window is displayed
    b. Set **Name** to **PlateJob**
    c. Set **Source** to **Model**
    d. Select **Plate Bending Model** (it is the only option displayed)
    e. Click **Continue..** The **Edit Job** window is displayed
    f. Set **Description** to **Job simulates the bending of a plate**
    g. Set **Job Type** to **Full Analysis**.

h. Leave all other options at defaults
i. Click **OK**
j. Expand the **Jobs** container in the Model Database
k. Right-click on **PlateJob** and choose **Submit**.
l. You will see a popup saying **History output is not requested in the following steps: Load Step. OK to continue with job submission?** Click **Yes**.
m. This will run the simulation. You will see the following messages in the message window:

   The job input file "PlateJob.inp" has been submitted for analysis.
   Job PlateJob: Analysis Input File Processor completed successfully
   Job PlateJob: Abaqus/Standard completed successfully
   Job PlateJob completed successfully

16. Show element labels and plot contours
    a. Right-click on **PlateJob (Completed)** in the Model Database. Choose **Results**. The viewport changes to the **Visualization** module.
    b. In the toolbar click the **Plot Undeformed Shape** tool. The plate is displayed in its undeformed state.
    c. In the toolbar click the **Common Options** tool. The **Common Plot Options** window is displayed.
    d. In the **Labels** tab check **Show element labels**
    e. Click **OK**. The elements are now numbered on the truss in the viewport.
    f. In the toolbar click the **Plot Contours on Deformed Shape** tool. A color contour of S. Mises stresses is plotted over the plate
17. Report Field Outputs
    a. Using the menu bar click on **Report> Field Output...** The **Report Field Output** window is displayed.
    b. In the **Variable** tab, set the **Output Variables Position** to **Integration Point**.
    c. In the list you see **S: Stress components**. Click the arrow next to it to expand the list. Select **Mises** by checking it off
    d. In the **Setup** tab, set the **File Name** to **platestresses.rpt**.
    e. Uncheck the **Append to file** option
    f. Set **Sort by** to **S.Mises** using the dropdown
    g. Set it to **Descending**
    h. For **Write** check **Field output, Column totals** and **column min/max**.

i. Click **OK** to close the **Field Output** window. In the message area you see **The field output report was appended to file "platestresses.rpt".**

j. You can now use windows explorer to navigate to the Abaqus temporary files directory. Open platestresses.rpt using WordPad. You will find that the stresses have been tabulated with element labels. In addition the maximum and minimum stresses are displayed at the bottom of the report.

10.3 Python Script

The following Python script replicates the above procedure for the analysis of the planar shell. You can find it in the source code accompanying the book in **plate_bending.py**. You can run it by opening a new model in Abaqus (**File > New Model Database > With Standard/Explicit Model**) and running it with **File > Run Script...**

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# -------------------------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Plate Bending Model')
plateModel = mdb.models['Plate Bending Model']

# -------------------------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the plate using the rectangle tool
plateProfileSketch = plateModel.ConstrainedSketch(name='Plate Sketch',
                                                  sheetSize=20)
plateProfileSketch.rectangle(point1=(0.0,0.0), point2=(5.0,3.0))

# b) Create a shell named "Plate" using the sketch
platePart=plateModel.Part(name='Plate', dimensionality=THREE_D,
                                        type=DEFORMABLE_BODY)
platePart.BaseShell(sketch=plateProfileSketch)

# -------------------------------------------------------------------------
# Create material

import material
```

```
# Create material AISI 1005 Steel by assigning mass density, youngs modulus and
# poissons ratio
plateMaterial = plateModel.Material(name='AISI 1005 Steel')
plateMaterial.Density(table=((7872, ),      ))
plateMaterial.Elastic(table=((200E9, 0.29), ))

# -----------------------------------------------------------------------
# Create homogeneous shell section of thickness 0.1m and assign the plate to it

import section

# Create a section to assign to the plate
plateSection = plateModel.HomogeneousShellSection(name='Plate Section',
                                                  material='AISI 1005 Steel',
                                                  thicknessType=UNIFORM,
                                                  thickness=0.1)

#assign the plate to this section
plate_face_point = (2.5, 1.5, 0.0)
plate_face = platePart.faces.findAt((plate_face_point,))
plate_region = (plate_face,)

platePart.SectionAssignment(region=plate_region, sectionName='Plate Section',
                            offset=0.0, offsetType=MIDDLE_SURFACE, offsetField='')

# -----------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
plateAssembly = plateModel.rootAssembly
plateInstance = plateAssembly.Instance(name='Plate Instance', part=platePart,
                                       dependent=ON)

# -----------------------------------------------------------------------
# Create the step

import step

# Create a static general step
plateModel.StaticStep(name='Load Step', previous='Initial',
                      description='Apply concentrated forces in this step',
                      nlgeom=ON)

# -----------------------------------------------------------------------
# Create the field output request

# change the name of field output request 'F-Output-1' to 'Output Stresses and
# Displacements'
plateModel.fieldOutputRequests.changeKey(fromName='F-Output-1',
                                         toName='Output Stresses and Displacements')
```

```
# since F-Output-1 is applied at the 'Apply Load' step by default, 'Selected
# Field Outputs' will be too
# We only need to set the required variables
plateModel.fieldOutputRequests['Output Stresses and Displacements'] \
                                                .setValues(variables=('S','UT'))

# -----------------------------------------------------------------------
# Create the history output request

# We don't want any history outputs so lets delete the existing one 'H-Output-1'
del plateModel.historyOutputRequests['H-Output-1']

# -----------------------------------------------------------------------
# Apply boundary conditions - fix one edge

fixed_edge = plateInstance.edges.findAt(((0.0, 1.5, 0.0), ))
fixed_edge_region=regionToolset.Region(edges=fixed_edge)

plateModel.DisplacementBC(name='FixEdge', createStepName='Initial',
                          region=fixed_edge_region, u1=SET, u2=SET, u3=SET,
                          ur1=SET, ur2=SET, ur3=SET,
                          amplitude=UNSET, distributionType=UNIFORM,
                          fieldName='', localCsys=None)

# Instead of using the displacements/rotations boundary condition and setting all
# six DOF to zero
# We could have just used the Encastre condition with the following statement
# plateModel.EncastreBC(name='Encaster edge', createStepName='Initial',
#                                          region=fixed_edge_region)

# -----------------------------------------------------------------------
# Create vertices on which to apply concentrated forces by partitioning part

# Create the datum points
platePart.DatumPointByCoordinate(coords=(0.0, 1.0, 0.0))
platePart.DatumPointByCoordinate(coords=(0.0, 2.0, 0.0))
platePart.DatumPointByCoordinate(coords=(5.0, 1.0, 0.0))
platePart.DatumPointByCoordinate(coords=(5.0, 2.0, 0.0))

# Assign the datum points to variables
# Abaqus stores the 4 datum points in platePart.datums
# Since their keys may or may not start at zero, put the keys in an array sorted
# in ascending order
platePart_datums_keys = platePart.datums.keys()
platePart_datums_keys.sort()
plate_datum_point_1 = platePart.datums[platePart_datums_keys[0]]
plate_datum_point_2 = platePart.datums[platePart_datums_keys[1]]
plate_datum_point_3 = platePart.datums[platePart_datums_keys[2]]
plate_datum_point_4 = platePart.datums[platePart_datums_keys[3]]

# Select the entire face and partition it using two points
```

```
partition_face_pt = (2.5, 1.5, 0.0)
partition_face = platePart.faces.findAt((partition_face_pt,))
platePart.PartitionFaceByShortestPath(point1=plate_datum_point_1,
                                      point2=plate_datum_point_3,
                                      faces=partition_face)

# Now two faces exist, select the one that needs to be partitioned
partition_face_pt = (2.5, 2.0, 0.0)
partition_face = platePart.faces.findAt((partition_face_pt,))
platePart.PartitionFaceByShortestPath(point1=plate_datum_point_2,
                                      point2=plate_datum_point_4,
                                      faces=partition_face)

# Since the partitions have been created, vertices can be extracted
vertices_for_concentrated_force = plateInstance.vertices.findAt(((5.0, 1.0, 0.0),),
                                                               ((5.0, 2.0, 0.0),),)

# -------------------------------------------------------------------------
# Apply concentrated forces

plateModel.ConcentratedForce(name='Concentrated Forces',
                             createStepName='Load Step',
                             region=(vertices_for_concentrated_force,),
                             cf3=-7000.0, distributionType=UNIFORM)

# -------------------------------------------------------------------------
# Create the mesh

import mesh

# set element type
plate_mesh_region = plate_region

elemType1 = mesh.ElemType(elemCode=S8R, elemLibrary=STANDARD)

platePart.setElementType(regions=plate_mesh_region, elemTypes=(elemType1,))

# seed edges by number
mesh_edges_vertical = platePart.edges.findAt(((0.0, 0.5, 0.0), ),
                                             ((0.0, 1.5, 0.0), ),
                                             ((0.0, 2.5, 0.0), ),
                                             ((5.0, 0.5, 0.0), ),
                                             ((5.0, 1.5, 0.0), ),
                                             ((5.0, 2.5, 0.0), ))

mesh_edges_horizontal = platePart.edges.findAt(((2.5, 0.0, 0.0), ),
                                               ((2.5, 1.0, 0.0), ),
                                               ((2.5, 2.0, 0.0), ),
                                               ((2.5, 3.0, 0.0), ))

platePart.seedEdgeByNumber(edges=mesh_edges_vertical, number = 3)
platePart.seedEdgeByNumber(edges=mesh_edges_horizontal, number=10)
platePart.generateMesh()

# -----------------------------------------------------------------------
# Create and run the job

import job

# create the job

mdb.Job(name='PlateJob', model='Plate Bending Model', type=ANALYSIS,
                         description='Job simulates the bending of a plate')

# run the job
mdb.jobs['PlateJob'].submit(consistencyChecking=OFF)

# do not return control till job is finished running
mdb.jobs['PlateJob'].waitForCompletion()


#########################################################################
#------------------------------------------------------------------------
#Post Processing
#------------------------------------------------------------------------
#########################################################################

# -----------------------------------------------------------------------
# Display deformed state with contours

import visualization

plate_viewport = session.Viewport(name='Plate Results Viewport')
plate_Odb_Path = 'PlateJob.odb'
an_odb_object = session.openOdb(name=plate_Odb_Path)
plate_viewport.setValues(displayedObject=an_odb_object)
plate_viewport.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))

# -----------------------------------------------------------------------
# Report stresses in descending order

import odbAccess

# The main session viewport must be set to the odb object using the following line.
# If not you might receive an error message that states
# "There are no active entities. No report has been generated."
session.viewports['Viewport: 1'].setValues(displayedObject=an_odb_object)

# Set the option to display the reported quantity (in our case the stresses) in
# descending order
session.fieldReportOptions.setValues(sort=DESCENDING)

# Name the report and give it a path. If you do not assign a path (as is done here)
# it will be stored in the default abaqus temporary directory
report_name_and_path='PlateStresses.rpt'

# ..........................................................................
# You may enter an entire path if you wish to have the report stored in a
# particular location.
# One way to do it is using the following syntax.
# report_name='PlateReport'
# report_path='C:/MyNewFolder/'
# report_name_and_path = report_path + report_name + '.rpt'
# Alternatively you could have used 1 statement instead of these 3 :
# report_name_and_path='C:/MyNewFolder/PlateReport.rpt'

# Note however that the folder 'MyNewFolder' must exist otherwise you will likely
# get the following error
# "IOError:C/MyNewFolder: Directory not found"
# You must either create the folder in Windows before running the script
# Or if you wish to create it using Python commands you must use the os.makedir()
# or os.makedirs() function
# os.makedirs() is preferable because you can create multiple nested directories
# in one statent if you wish
# Note that this function returns an exception if the directory already exists
# hence it is a good idea to use a try block

#try:
#    os.makedirs(report_path)
#except:
#    print "Directory exists hence no need to recreate it. \
#                                                     Move on to next statement"


# Here it is rewritten without all the comments
"""
report_name='PlateStresses'
report_path='C:/MyNewFolder/'
report_name_and_path = report_path + report_name + '.rpt'
try:
    os.makedirs(report_path)
except:
    print "Directory exists hence no need to recreate it. \
                                                     Move on to next statement"
"""
# ..........................................................................


# Write the field report outputting the Mises stresses
session.writeFieldReport(fileName=report_name_and_path, append=OFF,
                   sortItem='S.Mises', odb=an_odb_object, step=0, frame=1,
                   outputPosition=INTEGRATION_POINT,
                   variable=(('S', INTEGRATION_POINT, ((INVARIANT, 'Mises'), )), ))
```

10.4 Examining the Script

Let's examine the script statement by statement.

10.4.1 Initialization (import required modules)

The block dealing with this initialization is

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)
```

These statements are identical to those used in the Cantilever Beam example and were explained in section 4.3.1on page 65

10.4.2 Create the model

The following block creates the model

```
# -----------------------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Plate Bending Model')
plateModel = mdb.models['Plate Bending Model']
```

These statements rename the model from 'Model-1' to 'Plate Bending Model'. They are almost identical to those used in the Cantilever Beam example which were explained in section 4.3.2 on page 67.

10.4.3 Create the part

The following block of code creates the part

```
# -----------------------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the plate using the rectangle tool
plateProfileSketch = plateModel.ConstrainedSketch(name='Plate Sketch',
                                                  sheetSize=20)
plateProfileSketch.rectangle(point1=(0.0,0.0), point2=(5.0,3.0))
```

```
# b) Create a shell named "Plate" using the sketch
platePart=plateModel.Part(name='Plate', dimensionality=THREE_D,
                                        type=DEFORMABLE_BODY)
platePart.BaseShell(sketch=plateProfileSketch)
```

All the statements except the last one are very similar to the ones used in the Cantilever Beam example. To refresh your memory on the **ConstrainedSketch(), rectangle()** and **Part()** methods, refer back to section 4.3.3.

```
platePart.BaseShell(sketch=plateProfileSketch)
```

This creates a **Feature** object by calling the **BaseShell()** method. **Feature** objects were explained in section 4.3.3. To jog your memory, **Feature** objects contain parameters specified by the user as well as modifications made to the model by Abaqus based on those parameters. **BaseShell()** creates a planar shell from the given **ConstrainedSketch** object which is passed to it as its one required argument. **BaseShell()** has no optional arguments. Notice that no depth/thickness is specified in the **BaseShell()** method unlike the **BaseSolidExtrude()** method used in the Cantilever Beam example.

10.4.4 Define the materials

The following block of code creates the material for the simulation

```
# -----------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs modulus and
# poissons ratio
plateMaterial = plateModel.Material(name='AISI 1005 Steel')
plateMaterial.Density(table=((7872, ),       ))
plateMaterial.Elastic(table=((200E9, 0.29), ))
```

The statements are almost identical to those used in the Cantilever Beam example which were explained in section 4.3.4 on page 71.

10.4.5 Create solid sections and make section assignments

The following block creates the sections and makes assignments

```
# -----------------------------------------------------------------------
# Create homogeneous shell section of thickness 0.1m and assign the plate to it

```

```
import section

# Create a section to assign to the plate
plateSection = plateModel.HomogeneousShellSection(name='Plate Section',
                                                  material='AISI 1005 Steel',
                                                  thicknessType=UNIFORM,
                                                  thickness=0.1)

#assign the plate to this section
plate_face_point = (2.5, 1.5, 0.0)
plate_face = platePart.faces.findAt((plate_face_point,))
plate_region = (plate_face,)

platePart.SectionAssignment(region=plate_region, sectionName='Plate Section',
                            offset=0.0, offsetType=MIDDLE_SURFACE, offsetField='')
```

The statement

```
import section
```

imports the section module making its properties available to the script.

```
plateSection = plateModel.HomogeneousShellSection(name='Plate Section',
                                                  material='AISI 1005 Steel',
                                                  thicknessType=UNIFORM,
                                                  thickness=0.1)
```

This statement creates a **HomogeneousShellSection** object using the **HomogeneousShellSection()** method. The **HomogeneousShellSection** object defines the properties of a shell section. The first parameter, **name**, given to the method is the name which is used as the repository key. The second parameter, **material**, is the repository key of the material (a String), which has been defined before. The third parameter, an optional one called **thicknessType**, is a SymbolicConstant which identifies the type of distribution which is used for defining the thickness of the elements. We have assigned it the value **UNIFORM**, The fourth parameter is an optional one called **thickness**, which is a Float specifying the thickness of the section. It is only applicable when **thicknessType** has been set to **UNIFORM**.

```
plate_face_point = (2.5, 1.5, 0.0)
plate_face = platePart.faces.findAt((plate_face_point,))
plate_region = (plate_face,)
```

The **faces.findAt()** method returns the face (**Face** object) at the specified coordinates. The next statement turns it into a sequence of **Face** objects by using a comma, giving us a **Region** object (remember a **Region** is a sequence of other objects such as **Face** objects).

```
platePart.SectionAssignment(region=plate_region, sectionName='Plate Section',
                            offset=0.0, offsetType=MIDDLE_SURFACE, offsetField='')
```

The **SectionAssignment()** method is used to create a **SectionAssignment** object, which as mentioned in section 4.3.5 on page 72 is an object that is used to assign sections to a part, an assembly or an instance. The first parameter is a **region**, in this case the region is the entire part. The second argument, **sectionName**, is the repository key we give to the section. The third argument, **offset**, is an optional one. It is a Float specifying the offset with a default value of 0.0. The fourth argument, **offsetType**, is also optional. It is a SymbolicConstant specifying the method used to define the shell offset. We have used **MIDDLE_SURFACE**. The last argument is an optional one called **offsetField** which is a String specifying the name of the field specifying the offset. We set it to ''.

10.4.6 Create an assembly

The following code block creates the assembly.

```
# -----------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
plateAssembly = plateModel.rootAssembly
plateInstance = plateAssembly.Instance(name='Plate Instance', part=platePart,
                                       dependent=ON)
```

These statements are almost identical to the ones used in the Cantilever Beam example in section 4.3.6 on page 74.

10.4.7 Create steps

The following code block creates the steps.

```
# -----------------------------------------------------------------------
# Create the step

import step

# Create a static general step
plateModel.StaticStep(name='Load Step', previous='Initial',
                      description='Apply concentrated forces in this step',
                      nlgeom=ON)
```

The statements are almost identical to the ones used in the Cantilever Beam example in section 4.3.7 on page 75. The one big difference is the use of the **nlgeom** parameter. This is an optional argument which specifies whether Abaqus should account for geometric nonlinearity during the analysis. It accepts a Boolean value (ON or OFF) with default being OFF.

10.4.8 Create and define field output requests

The following code block creates the field output requests.

```
# -------------------------------------------------------------------------
# Create the field output request

# change the name of field output request 'F-Output-1' to 'Output Stresses and
# Displacements'
plateModel.fieldOutputRequests.changeKey(fromName='F-Output-1',
                                         toName='Output Stresses and Displacements')

# since F-Output-1 is applied at the 'Apply Load' step by default, 'Selected
# Field Outputs' will be too
# We only need to set the required variables
plateModel.fieldOutputRequests['Output Stresses and Displacements'] \
                                                    .setValues(variables=('S','UT'))
```

The statements are similar to ones used in the Cantilever Beam example in section 4.3.8 on page 76.

10.4.9 Create and define history output requests

The following code block defines the history output requests:

```
# -------------------------------------------------------------------------
# Create the history output request

# We don't want any history outputs so lets delete the existing one 'H-Output-1'
del plateModel.historyOutputRequests['H-Output-1']
```

We do not want any history outputs in this example so we shall delete the existing default 'H-Output-1' using the Python del command.

10.4.10 Apply boundary conditions

The following block of code applies the boundary condition:

```
# ----------------------------------------------------------------------
# Apply boundary conditions - fix one edge

fixed_edge = plateInstance.edges.findAt(((0.0, 1.5, 0.0), ))
fixed_edge_region=regionToolset.Region(edges=fixed_edge)

plateModel.DisplacementBC(name='FixEdge', createStepName='Initial',
                          region=fixed_edge_region, u1=SET, u2=SET, u3=SET,
                          ur1=SET, ur2=SET, ur3=SET,
                          amplitude=UNSET, distributionType=UNIFORM,
                          fieldName='', localCsys=None)

# Instead of using the displacements/rotations boundary condition and setting all
# six DOF to zero
# We could have just used the Encastre condition with the following statement
# plateModel.EncastreBC(name='Encaster edge', createStepName='Initial',
#                                              region=fixed_edge_region)
```

```
fixed_edge = plateInstance.edges.findAt(((0.0, 1.5, 0.0), ))
```

uses the **findAt()** method to find the edge to be fixed using the coordinates of its center (midpoint).

```
fixed_edge_region=regionToolset.Region(edges=fixed_edge)
```

uses the **Region()** method to create a **Region** object out of the edge. This Region object can then be used in the **DisplacementBC()** method. The **Region** object was discussed in Section 4.3.5 of the Cantilever Beam example on page 72.

```
plateModel.DisplacementBC(name='FixEdge', createStepName='Initial',
                          region=fixed_edge_region, u1=SET, u2=SET, u3=SET,
                          ur1=SET, ur2=SET, ur3=SET,
                          amplitude=UNSET, distributionType=UNIFORM,
                          fieldName='', localCsys=None)
```

This statement creates a **DisplacementBC** object which you encountered earlier in section 7.4.11. To jog your memory, it stores the data for a displacement/rotation. The **DisplacementBC** object is derived from the **BoundaryCondition** object. The first required argument is a String for the name. The second is the name/key of the step in which the boundary condition is to be applied. In this case we apply it to the 'Initial' step. The third argument must be a region object. The remaining arguments are optional. Note however that even though **u1, u2, u3**, **ur1**, **ur2** and **ur3** are optional arguments, at least one of them must be specified. For **u1**, **u2** and **u3** we use the SymbolicConstant **SET** thus preventing translation in the 1, 2 and 3 directions (a.k.a. X, Y and Z directions). **ur1**, **ur2** and **ur3** are not specified, and will default to **UNSET** thus allowing rotation along all 3 axes. Since no amplitude is used, it is set to **UNSET** and the **distributionType** is set to **UNIFORM** ensuring a uniform special distribution of the boundary condition in the applied region.

Note: Instead of using the displacements/rotations boundary condition and setting all six DOF to zero, we could have just used the Encastre condition with the following statement.

```
plateModel.EncastreBC(name='Encaster edge', createStepName='Initial',
                                            region=fixed_edge_region)
```

Here the **EncastreBC()** method is used to create a **TypeBC** object, which is an object that stores data on several types of predefined boundary conditions commonly used in stress/displacement analysis. The **EncastreBC()** method was previously discussed in section 4.3.11 of the Cantilever Beam example.

10.4.11 Partition part to create vertices

The following block of code applies the constraints:

```
# -------------------------------------------------------------------
# Create vertices on which to apply concentrated forces by partitioning part

# Create the datum points
platePart.DatumPointByCoordinate(coords=(0.0, 1.0, 0.0))
platePart.DatumPointByCoordinate(coords=(0.0, 2.0, 0.0))
platePart.DatumPointByCoordinate(coords=(5.0, 1.0, 0.0))
platePart.DatumPointByCoordinate(coords=(5.0, 2.0, 0.0))

# Assign the datum points to variables
# Abaqus stores the 4 datum points in platePart.datums
# Since their keys may or may not start at zero, put the keys in an array sorted
# in ascending order
platePart_datums_keys = platePart.datums.keys()
platePart_datums_keys.sort()
plate_datum_point_1 = platePart.datums[platePart_datums_keys[0]]
plate_datum_point_2 = platePart.datums[platePart_datums_keys[1]]
plate_datum_point_3 = platePart.datums[platePart_datums_keys[2]]
plate_datum_point_4 = platePart.datums[platePart_datums_keys[3]]

# Select the entire face and partition it using two points
partition_face_pt = (2.5, 1.5, 0.0)
```

```
partition_face = platePart.faces.findAt((partition_face_pt,))
platePart.PartitionFaceByShortestPath(point1=plate_datum_point_1,
                                      point2=plate_datum_point_3,
                                      faces=partition_face)

# Now two faces exist, select the one that needs to be partitioned
partition_face_pt = (2.5, 2.0, 0.0)
partition_face = platePart.faces.findAt((partition_face_pt,))
platePart.PartitionFaceByShortestPath(point1=plate_datum_point_2,
                                      point2=plate_datum_point_4,
                                      faces=partition_face)

# Since the partitions have been created, vertices can be extracted
vertices_for_concentrated_force = plateInstance.vertices.findAt(((5.0, 1.0, 0.0),),
                                                                 ((5.0, 2.0, 0.0),),)
```

```
platePart.DatumPointByCoordinate(coords=(0.0, 1.0, 0.0))
platePart.DatumPointByCoordinate(coords=(0.0, 2.0, 0.0))
platePart.DatumPointByCoordinate(coords=(5.0, 1.0, 0.0))
platePart.DatumPointByCoordinate(coords=(5.0, 2.0, 0.0))
```

These statements use the **DatumPointByCoordinate()** method to create datum points on the part. The **DatumPointByCoordinate()** method creates a **Feature** object and a **DatumPoint** object at the point specified in the coordinates. **coords** is a sequence of three Floats specifying X, Y and Z coordinates of the datum point and is a required argument.

```
platePart_datums_keys = platePart.datums.keys()
platePart_datums_keys.sort()
plate_datum_point_1 = platePart.datums[platePart_datums_keys[0]]
plate_datum_point_2 = platePart.datums[platePart_datums_keys[1]]
plate_datum_point_3 = platePart.datums[platePart_datums_keys[2]]
plate_datum_point_4 = platePart.datums[platePart_datums_keys[3]]
```

We need to assign each of the 4 datum points we have created to variables so that we can use them with the **PartitionFaceByShortestPath()** method in subsequent statements. Abaqus stores all the datum points in **platePart.datums**. We need to extract these and assign them to variables.

We get the keys, or repository names, of each of them by using the **keys()** method as **platePart.datums.keys()** and assign them to the variable **platePart_datums_keys**. The next statement uses the **sort()** method to sort these keys in ascending order and places them back in the variable **platePart_datums_keys**. We can then refer to each of the keys using index notation, such as **platePart_datums_keys[0]** for the first one. And we can refer to the corresponding **DatumPoint** object using these keys as **platePart.datums[platePart_datums_keys[0]]**.

```
partition_face_pt = (2.5, 1.5, 0.0)
partition_face = platePart.faces.findAt((partition_face_pt,))
```

These statements locate the face to be partitioned using the **findAt()** method and place it in a variable.

```
platePart.PartitionFaceByShortestPath(point1=plate_datum_point_1,
                                      point2=plate_datum_point_3,
                                      faces=partition_face)
```

The **PartitionFaceByShortestPath()** method can be used to partition one or more faces using the shortest path between two given points (a straight line). It has 3 required arguments. **point1** and **point2** can both be a **Vertex** object, an **InterestingPoint** object, or a **DatumPoint** object. In our case we have used **DatumPoint** objects which we obtained using the previous statements. The third required argument **faces** is a sequence of **Face** objects specifying the faces to partition.

```
# Now two faces exist, select the one that needs to be partitioned
partition_face_pt = (2.5, 2.0, 0.0)
partition_face = platePart.faces.findAt((partition_face_pt,))
platePart.PartitionFaceByShortestPath(point1=plate_datum_point_2,
                                      point2=plate_datum_point_4,
                                      faces=partition_face)
```

These statements are similar to the previous 3 and are used to create the second partition. Note that since the original face has been partitioned, it is necessary to use **findAt()** again to choose one of the 2 new faces which we wish to partition.

```
vertices_for_concentrated_force = plateInstance.vertices.findAt(((5.0, 1.0, 0.0),),
                                                                ((5.0, 2.0, 0.0),),)
```

Now that the faces have been partitioned, vertices have been created as part of the partitioning where the partition lines intersect with the edges of the plate. Abaqus can now be instructed to extract these vertices using **vertices.findAt()**.

10.4.12 Apply loads

The following block applies the loads:

```
# ---------------------------------------------------------------------
# Apply concentrated forces

plateModel.ConcentratedForce(name='Concentrated Forces',
                             createStepName='Load Step',
                             region=(vertices_for_concentrated_force,),
                             cf3=-7000.0, distributionType=UNIFORM)
```

The **ConcentratedForce()** method creates a **ConcentratedForce** object, which is derived from the **Load** object. The first argument is the name or repository key for which the String 'Concentrated Forces' is given. The second argument is the name/key of the step in which the concentrated force will be applied. The third argument is required to be a **Region** object. However **vertices_for_concentrated_force** is a **GeomSequence**. A **GeomSequence** is a sequence of **Geometry** objects, such as **Vertices** or **Edges**. We put it in parenthesis and add a comma to make it a **Region** object. Hence *region=(vertices_for_concentrated_force,)*. The forth argument **cf3** is the Z-component of the force. (**cf1** is X-component, and **cf2** is Y-component). It is set to -7000.0 with the negative sign indicating the force is pointing toward the plane (since +ve Z is out from the plane). The fifth argument sets the **distributionType** to uniform using the SymbolicConstant **UNIFORM**.

10.4.13 Mesh

The following block creates the mesh

```
# ---------------------------------------------------------------------
# Create the mesh

import mesh

# set element type
plate_mesh_region = plate_region

elemType1 = mesh.ElemType(elemCode=S8R, elemLibrary=STANDARD)

platePart.setElementType(regions=plate_mesh_region, elemTypes=(elemType1,))

# seed edges by number
mesh_edges_vertical = platePart.edges.findAt(((0.0, 0.5, 0.0), ),
                                             ((0.0, 1.5, 0.0), ),
                                             ((0.0, 2.5, 0.0), ),
                                                ((5.0, 0.5, 0.0), ),
                                                ((5.0, 1.5, 0.0), ),
                                                ((5.0, 2.5, 0.0), ))

mesh_edges_horizontal = platePart.edges.findAt(((2.5, 0.0, 0.0), ),
                                                   ((2.5, 1.0, 0.0), ),
                                                   ((2.5, 2.0, 0.0), ),
                                                   ((2.5, 3.0, 0.0), ))

platePart.seedEdgeByNumber(edges=mesh_edges_vertical, number = 3)
platePart.seedEdgeByNumber(edges=mesh_edges_horizontal, number=10)

platePart.generateMesh()
```

```
plate_mesh_region = plate_region
elemType1 = mesh.ElemType(elemCode=S8R, elemLibrary=STANDARD)
platePart.setElementType(regions=plate_mesh_region, elemTypes=(elemType1,))
```

These statements are similar to the ones you have used previously, refer to section 7.4.12 on page 139 to refresh your memory on the **ElemType()** and **setElementType()** methods.

```
mesh_edges_vertical = platePart.edges.findAt(((0.0, 0.5, 0.0), ),
                                                ((0.0, 1.5, 0.0), ),
                                                ((0.0, 2.5, 0.0), ),
                                                ((5.0, 0.5, 0.0), ),
                                                ((5.0, 1.5, 0.0), ),
                                                ((5.0, 2.5, 0.0), ))

mesh_edges_horizontal = platePart.edges.findAt(((2.5, 0.0, 0.0), ),
                                                   ((2.5, 1.0, 0.0), ),
                                                   ((2.5, 2.0, 0.0), ),
                                                   ((2.5, 3.0, 0.0), ))
```

We wish to specify how many seeds are on each edge. However we have partitioned the plate so we now have 6 edges along one axis, and 4 along the other. The above statements identify these edges using the **findAt()** method and their midpoints. The edges along X and Y axes have been collected into two separate variables **mesh_edges_vertical** and **mesh_edges_horizontal**.

```
platePart.seedEdgeByNumber(edges=mesh_edges_vertical, number = 3)
platePart.seedEdgeByNumber(edges=mesh_edges_horizontal, number=10)
platePart.generateMesh()
```

You have seen the **seedEdgeByNumber()** and **generateMesh()** methods used before, refer to section 7.4.12 on page 139.

10.4.14 Create and run the job

The following block runs the job

```
# -----------------------------------------------------------------------
# Create and run the job

import job

# create the job

mdb.Job(name='PlateJob', model='Plate Bending Model', type=ANALYSIS,
                         description='Job simulates the bending of a plate')

# run the job
mdb.jobs['PlateJob'].submit(consistencyChecking=OFF)

# do not return control till job is finished running
mdb.jobs['PlateJob'].waitForCompletion()
```

These statements are similar to ones used previously. You may refer to Section 4.3.13 on page 88.

10.4.15 Display deformed state with contours

The following code displays the deformed state of the plate

```
# -----------------------------------------------------------------------
# Display deformed state with contours

import visualization

plate_viewport = session.Viewport(name='Plate Results Viewport')
plate_Odb_Path = 'PlateJob.odb'
an_odb_object = session.openOdb(name=plate_Odb_Path)
plate_viewport.setValues(displayedObject=an_odb_object)
plate_viewport.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))
```

These statements are almost identical to the ones used in the Cantilever Beam example. Refer to section 4.3.14 on page 89 to review the **Viewport()**, **openOdb()** and **setValues()** methods.. The only difference here is that **plotState** has been set to a different SymbolicConstant **CONTOURS_ON_DEF** instead of **DEFORMED** to plot a color contour.

10.4.16 Write Field Output Report

The following statements create a new file and then write the field output report to it.

```
# -----------------------------------------------------------------------------
# Report stresses in descending order

import odbAccess

# The main session viewport must be set to the odb object using the following line.
# If not you might receive an error message that states
# "There are no active entities. No report has been generated."
session.viewports['Viewport: 1'].setValues(displayedObject=an_odb_object)

# Set the option to display the reported quantity (in our case the stresses) in
# descending order
session.fieldReportOptions.setValues(sort=DESCENDING)

# Name the report and give it a path. If you do not assign a path (as is done here)
# it will be stored in the default abaqus temporary directory
report_name_and_path='PlateStresses.rpt'

# Write the field report outputting the Mises stresses
session.writeFieldReport(fileName=report_name_and_path, append=OFF,
                  sortItem='S.Mises', odb=an_odb_object, step=0, frame=1,
                  outputPosition=INTEGRATION_POINT,
                  variable=(('S', INTEGRATION_POINT, ((INVARIANT, 'Mises'), )), ))
```

`import odbAccess`

This makes the **Odb** objects, methods and members available to the script.

`session.viewports['Viewport: 1'].setValues(displayedObject=an_odb_object)`

You've seen the **setValues()** method used in the Cantilever Beam example in both the initialization block as well as the post processing. To refresh your memory, **setValues()** sets the display to the selected output database.

If we leave out this line, Abaqus will give us an error message in subsequent statements when we try to generate the report saying "There are no active entities. No report has been generated."

`report_name_and_path='PlateStresses.rpt'`

```
session.writeFieldReport(fileName=report_name_and_path, append=OFF,
sortItem='S.Mises', odb=an_odb_object, step=0, frame=1,
    outputPosition=INTEGRATION_POINT, variable=(('S', INTEGRATION_POINT,
((INVARIANT,'Mises'), )), ))
```

The **writeFieldReport()** method writes a **FieldOutput** object to an ASCII file. It has a number of required arguments. **filename** is a String specifying the name of the file to which the output will be written. **append** is a Boolean which specifies whether or not to append field output to an existing file. **sortItem** specifies the item or column by which the tabulated values are sorted, we are sorting with respect to Mises stress. **odb** is the output database. Step can be either an Int or an **OdbStep** object (we have used Int) specifying which step the output values should be obtained from. **frame** can be an Int or an **OdbFrame** object specifying which frame to obtain the field output values from. **outputPosition** is a SymbolicConstant which specifies the position from which to obtain data, we set it to **INTEGRATION_POINT**. **variable** is a sequence of variable descriptions specifying one or more field output variables for which to obtain data. Each variable description must have a String specifying the name of the variable (in our case 'S'), a SymbolicConstant specifying the output position at which to report the data, and a sequence of tuples each consisting of a SymbolicConstant specifying refinement **(COMPONENT** or **INVARIANT)** followed by a String with the name of the component or invariant.

10.5 Summary

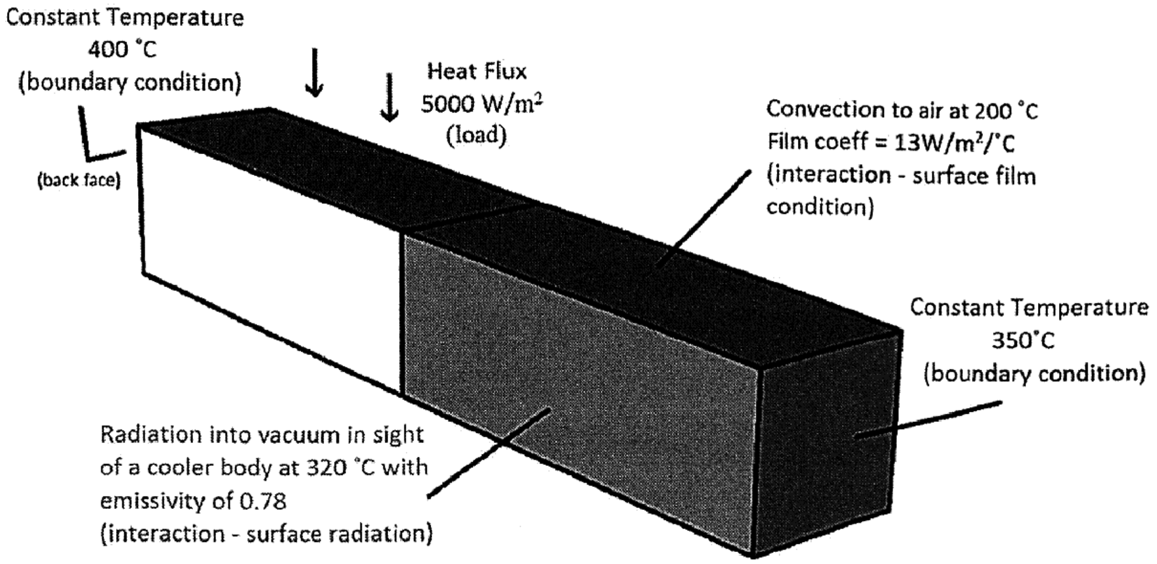
In this chapter we partitioned faces, displayed contours on a deformed plot, and reported field output to an external file. These are tasks you will undoubtedly script again in future.

11

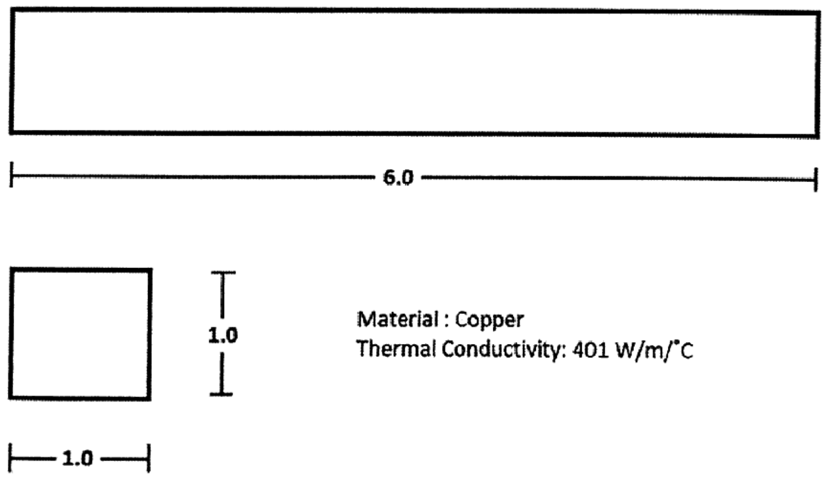
Heat Transfer Analysis

11.1 Introduction

In this chapter we will perform a heat transfer analysis on a rectangular block. The problem is displayed in the figure.


Constant Temperature
400 °C
(boundary condition)
(back face)
Heat Flux
5000 W/m²
(load)
Convection to air at 200 °C
Film coeff = 13W/m²/°C
(interaction - surface film condition)
Constant Temperature
350°C
(boundary condition)
Radiation into vacuum in sight of a cooler body at 320 °C with emissivity of 0.78
(interaction - surface radiation)

The dimensions and material properties are displayed in the following figure. The unit of length is meters.



In this exercise the following tasks will be performed first using the Abaqus GUI, and then using a Python script.

- Create a part
- Assign materials
- Assign sections
- Create an Assembly
- Create a datum plane and partition a part
- Create a heat transfer step
- Assign boundary conditions
- Assign loads
- Create a mesh
- Create and submit a job
- Plot contours
- Change view orientation

The following new topics are covered in this example:

- Model / Preprocessing
  - Create a steady state or transient heat transfer step
  - Assign heat flux loads and constant temperature boundary conditions
  - Use interactions to define convection and radiation heat loss mechanisms

- Modify model attributes to define the Stefan-Boltzmann constant and absolute zero of temperature scale
- Results / Post-processing
  - Display nodal temperatures as a color contour
  - Orient the viewport display and save custom views

11.2 Procedure in GUI

You can perform the simulation in Abaqus/CAE by following the steps listed below. You can either read through these, or watch the video demonstrating the process on the book website.

1. Rename **Model-1** to **Heat Transfer**
   a. Right-click on Model-1 in Model Database
   b. Choose **Rename..**
   c. Change name to **Heat Transfer**
2. Create the part
   a. Double-click on **Parts** in Model Database. **Create Part** window is displayed.
   b. Set **Name** to **Block**
   c. Set **Modeling Space** to **3D**
   d. Set **Type** to **Deformable**
   e. Set **Base Feature Shape** to **Solid**
   f. Set **Base Feature Type** to **Extrusion**
   g. Set **Approximate Size** to **5**
   h. Click **OK**. You will enter Sketcher mode.
3. Sketch the profile
   a. Use the **Create Lines:Rectangle (4 lines)** tool to draw the square profile of the block
   b. Use the **Add Dimension** tool to set the length of the horizontal and vertical elements to 1 m.
   c. Click **Done** to exit the sketcher. The **Edit Base Extrusion** window is displayed.
   d. . Set **Depth** to **6.0**
   e. Click **OK**. The extruded block is displayed.
4. Create the material

  a. Double-click on **Materials** in the Model Database. **Edit Material** window is displayed
  b. Set **Name** to **Copper**
  c. Select **Thermal>Conductivity**. Set **Conductivity** to **400** (which is 400 W/mK)
  d. Click **OK**

5. Assign sections
   a. Double-click on **Sections** in the Model Database. **Create Section** window is displayed
   b. Set **Name** to **Block Section**
   c. Set **Category** to **Solid**
   d. Set **Type** to **Homogeneous**
   e. Click **Continue...** The **Edit Section** window is displayed.
   f. In the **Basic** tab, set **Material** to **Copper** which was defined in the create material step.
   g. Click **OK.**
6. Assign the section to the block
   a. Expand the **Parts** container in the Model Database. Expand the part **Block.**
   b. Double-click on **Section Assignments**
   c. You see the message **Select the regions to be assigned a section** displayed below the viewport
   d. Click and drag with the mouse to select the entire block.
   e. Click **Done**. The **Edit Section Assignment** window is displayed.
   f. Set **Section** to **Block Section.**
   g. Click **OK.**
7. Create the Assembly
   a. Double-click on **Assembly** in the Model Database. The viewport changes to the **Assembly Module.**
   b. Expand the **Assembly** container.
   c. Double-click on **Instances**. The **Create Instance** window is displayed.
   d. Set **Parts** to **Block**
   e. Set **Instance Type** to **Dependent (mesh on part)**
   f. Click **OK.**
8. Partition the block
   a. At the top of the viewport, change **Module** to **Part** using the dropdown menu.

   b. Click the **Create Datum Plane: Midway between 2 points** tool. You see the message **Select the first point to create datum plane** displayed below the viewport.
   c. Click on a corner (such as 0.0,0.0,0.0).You see the message **Select the second point to create datum plane** displayed below the viewport.
   d. Click on the same corner on the opposite face (such as 0.0,0.0,6.0). You may need to use the **Rotate View** tool in order to be able to see that corner. The datum plane is displayed in the viewport in the middle of the block
   e. Click the **Partition cell: Use datum plane** tool. You see the message **Select a datum plane** displayed below the viewport
   f. Click on the datum plane to select it.
   g. Click the **Create Partition** button below the viewport. The block is partitioned in two.
   h. Click **Done**.
9. Create Steps
   a. Double-click on **Steps** in the Model Database. The **Create Step** window is displayed.
   b. Set **Name** to **Heating Step**
   c. Set **Insert New Step After** to **Initial**
   d. Set **Procedure Type** to **General >Heat transfer**
   e. Click **Continue..** The **Edit Step** window is displayed
   f. In the **Basic** tab, set **Description** to **Apply heat in this step**.
   g. Set **Response** to **Transient**.
   h. You may see a message **Default load variation with time has been changed to Ramp linearly over step**. Click **Dismiss**.
   i. Click **OK**.
10. Leave Field Outputs at default
11. No History Outputs.
12. Apply boundary conditions
   a. Double-click on **BCs** in the Model Database. The **Create Boundary Condition** window is displayed
   b. Set **Name** to **Const Temp Surf 1**
   c. Set **Step** to **Heating Step**
   d. Set **Category** to **Other**
   e. Set **Types for Selected Step** to **Temperature**
   f. Click **Continue…**

g. You see the message **Select regions for the boundary condition** displayed below the viewport
h. Select one end face of the block by clicking on it.
i. Click **Done**. The **Edit Boundary Condition** window is displayed.
j. Set **Distribution** to **Uniform**.
k. Set **Magnitude** to **400**.
l. Click **OK**.
m. In the same manner create another boundary condition **Const Temp Surf 2** for the opposite face of the block setting the magnitude to **350**.

13. Assign Loads
    a. Double-click on **Loads** in the Model Database. The **Create Load** window is displayed
    b. Set **Name** to **Heat Flux**
    c. Set **Step** to **Heating Step**
    d. Set **Category** to **Thermal**
    e. Set **Type for Selected Step** to **Surface heat flux**
    f. Click **Continue…**You see the message **Select surfaces for the load** displayed below the viewport
    g. Set it to individually from the drop down list
    h. Click on the top surface of the block, on the side of the partition closer to the 400 K constant temperature surface
    i. Click **Done. The Edit Load** window is displayed
    n. Set **Distribution** to **Uniform.**
    j. Set **Magnitude** to **5000**
    k. Click **OK**
    l. You will see the flux displayed with an arrows in the viewport on the selected top face
14. Assign Interactions
    a. Double-click on **Interactions** in the Model Database. The **Create Interaction** window is displayed.
    b. Set **Name** to **Convection**
    c. Set **Step** to **Heating Step**
    d. Set **Types for Selected Step** to **Surface film condition**
    e. Click **Continue…**
    f. You see the message **Select the surface** displayed below the viewport
    g. Select the face which will lose heat by convection by clicking on it.

h. Click **Done**. The **Edit Interaction** window is displayed
i. Set **Definition** to **Embedded Coefficient**
j. Set **Film coefficient** to **13**
k. Set **Film coefficient amplitude** to **Instantaneous**.
l. Set **Sink temperature** to **200**
m. Set **Sink amplitude** to **Ramp**
n. Click **OK**
o. Double-click on **Interactions** in the Model Database. The **Create Interaction** window is displayed.
p. Set **Name** to **Radiation**
q. Set **Step** to **Heating Step**
r. Set **Types for Selected Step** to **Surface radiation**
s. Click **Continue...**
t. You see the message **Select the surface** displayed below the viewport
u. Select the face which will lose heat by radiation by clicking on it.
v. Click **Done**. The **Edit Interaction** window is displayed
w. Set **Emissivity distribution** to **Uniform**
x. Set **Emissivity** to **0.78**
y. Set **Ambient temperature** to **320**
z. Set **Ambient temperature amplitude** to **Ramp**
aa. Click **OK**

15. Create the mesh
    a. Expand the**Parts** container in the Model Database.
    b. Expand **Block**
    c. Double-click on **Mesh (Empty)**. The viewport window changes to the **Mesh module** and the tools in the toolbar are now meshing tools.
    d. Using the menu bar click on **Mesh > Element Type ...**
    e. You see the message **Select the regions to be assigned element types** displayed below the viewport
    f. Click and drag using your mouse to select the entire block.
    g. Click **Done**. The **Element Type** window is displayed.
    h. Set **Element Library** to **Standard**
    i. Set **Geometric Order** to **Linear**
    j. Set **Family** to **Heat Transfer**
    k. You will notice the message **DC3D8: An8-node linear heat transfer block**
    l. Click **OK**

m. Click **Done**
n. Using the menu bar lick on **Seed >Part...**The **Global Seeds** window is displayed
o. Set **Approximate global size** to **0.5**. Leave everything else at default values.
p. Click **OK.**
q. You see the message **Seeding definition complete** displayed below the viewport. Click **Done.**
r. Using the menu bar click on **Mesh > Part**
s. You see the prompt **OK to mesh the part?** displayed below the viewport
t. Click **Yes**

16. Create and submit the job
    a. Double-click on **Jobs** in the Model Database. The **Create Job** window is displayed
    b. Set **Name** to **HeatTransferJob**
    c. Set **Source** to **Model**
    d. Select **Heat Transfer** (it is the only option displayed)
    e. Click **Continue..** The **Edit Job** window is displayed
    f. Set **Description** to **Job simulates heat conduction through block**
    g. Set **Job Type** to **Full Analysis**. Leave all other options at defaults
    h. Click **OK**
    i. Expand the**Jobs** container in the Model Database
    j. Right-click on **HeatTransferJob** and choose **Submit.**
    k. You will see a popup saying **History output is not requested in the following steps: Heating Step. OK to continue with job submission?** Click **Yes.**
    l. This will run the simulation. You will see the following messages in the message window:
       The job input file "HeatTransferJob.inp" has been submitted for analysis.
       Job HeatTransferJob: Analysis Input File Processor completed successfully
       Job HeatTransferJob: Abaqus/Standard completed successfully
       Job HeatTransferJob completed successfully

17. Plot heat contour
    a. Right-click on **HeatTransferJob (Completed)** in the Model Database. Choose **Results.**The viewport changes to the **Visualization** module.

b. Using the menu bar click on **Result> Field Output...** The **Field Output** window is displayed.
c. In the Primary **Variable** tab, set the **Output Variable** to **NT11** which has the description **Nodal temperature at nodes.**
d. Click **OK**
e. You see the **Select Plot State** window. Set **Plot state** to **Contour**. Click **OK**.

18. Change view to left view
    a. Expose the Views toolbar by using the menu bar and clicking on **Views>Toolbars > Views**.
    b. Click the **Apply left view** button on the **Views** toolbar.

11.3 Python Script

The following Python script replicates the above procedure for the heat transfer analysis problem. You can find it in the source code accompanying the book in **heat_transfer.py**. You can run it by opening a new model in Abaqus **(File > New Model Database > With Standard/Explicit Model)** and running it with **File > Run Script…**

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)


# ---------------------------------------------------------------------
# Create the model


mdb.models.changeKey(fromName='Model-1', toName='Heat Transfer')
heatModel = mdb.models['Heat Transfer']

# ---------------------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the beam cross section using rectangle tool
beamProfileSketch = heatModel.ConstrainedSketch(name='Beam CS', sheetSize=5)
beamProfileSketch.rectangle(point1=(0.0,0.0), point2=(1.0,1.0))

# b) Create a 3D deformable part by extruding the sketch
```

```
beamPart=heatModel.Part(name='Beam', dimensionality=THREE_D, type=DEFORMABLE_BODY)
beamPart.BaseSolidExtrude(sketch=beamProfileSketch, depth=6)

# -----------------------------------------------------------------------
# Create material

import material

# Create material Copper by assigning thermal conductivity (dont need youngs
# modulus and poissons ratio)
beamMaterial = heatModel.Material(name='Copper')
beamMaterial.Conductivity(table=((400.0, ),  ))

# -----------------------------------------------------------------------
# Create solid section and assign the beam to it

import section

# Create a section to assign to the beam
beamSection = heatModel.HomogeneousSolidSection(name='Beam Section',
                                                material='Copper')

#assign the beam to this section
beam_region = (beamPart.cells,)
beamPart.SectionAssignment(region=beam_region, sectionName='Beam Section')

# -----------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
beamAssembly = heatModel.rootAssembly
beamInstance = beamAssembly.Instance(name='Beam Instance', part=beamPart,
                                                           dependent=ON)

# -----------------------------------------------------------------------
# Create the datum plane and partition the part

vertex_at_one_end_coords = (0.0, 0.0, 0.0)
vertex_at_one_end = beamInstance.vertices.findAt((vertex_at_one_end_coords,))

vertex_at_other_end_coords = (0.0, 0.0, 6.0)
vertex_at_other_end = beamInstance.vertices.findAt((vertex_at_other_end_coords,))

beamPart.DatumPlaneByTwoPoint(point1=vertex_at_one_end_coords,
                              point2=vertex_at_other_end_coords)

beam_inside_xcoord=0.5
beam_inside_ycoord=0.5
beam_inside_zcoord=3.0
beamCells=beamPart.cells
```

```
selectedBeamCells=beamCells.findAt((beam_inside_xcoord,beam_inside_ycoord,
                                                    beam_inside_zcoord),)

beamPart.PartitionCellByDatumPlane(datumPlane=beamPart.datums[3],
                                   cells=selectedBeamCells)

# -------------------------------------------------------------------------
# Create the step

import step

# Create a heat transfer step
heatModel.HeatTransferStep(name='Heating Step', previous='Initial',
                           description='Apply heat in this step',
                           response=STEADY_STATE, amplitude=RAMP)

# -------------------------------------------------------------------------
# Apply boundary conditions

end_face_1_pt = (0.5, 0.5, 0.0)
end_face_1 = beamInstance.faces.findAt((end_face_1_pt,))
end_face_1_region=regionToolset.Region(faces=end_face_1)
heatModel.TemperatureBC(name='Const Temp Surf 1', createStepName='Heating Step',
                        region=end_face_1_region,  distributionType=UNIFORM,
                        fieldName='', magnitude=400.0, amplitude=UNSET)

end_face_2_pt = (0.5, 0.5, 6.0)
end_face_2 = beamInstance.faces.findAt((end_face_2_pt,))
end_face_2_region=regionToolset.Region(faces=end_face_2)
heatModel.TemperatureBC(name='Const Temp Surf 2', createStepName='Heating Step',
                        region=end_face_2_region, distributionType=UNIFORM,
                        fieldName='', magnitude=350.0, amplitude=UNSET)

# -------------------------------------------------------------------------
# Apply loads

flux_face_pt = (0.5, 1.0, 1.5)
flux_face = beamInstance.faces.findAt((flux_face_pt,))
flux_face_region=regionToolset.Region(side1Faces=flux_face)
heatModel.SurfaceHeatFlux(name='Heat Flux', createStepName='Heating Step',
                          region=flux_face_region, magnitude=5000.0)


# -------------------------------------------------------------------------
# Create interactions (convection and radiation)

# Convection
convection_face_pt = (0.5, 1.0, 4.5)
convection_face = beamInstance.faces.findAt((convection_face_pt,))
convection_face_region=regionToolset.Region(side1Faces=convection_face)
heatModel.FilmCondition(name='Convection', createStepName='Heating Step',
                        surface=convection_face_region, definition=EMBEDDED_COEFF,
                         filmCoeff=13.0, filmCoeffAmplitude='',
                         sinkTemperature=200.0, sinkAmplitude='')

# Radiation
radiation_face_pt = (0.0, 0.5, 4.5)
radiation_face = beamInstance.faces.findAt((radiation_face_pt,))
radiation_face_region=regionToolset.Region(side1Faces=radiation_face)
heatModel.RadiationToAmbient(name='Radiation', createStepName='Heating Step',
                             surface=radiation_face_region, radiationType=AMBIENT,
                             distributionType=UNIFORM, field='',
                             emissivity=0.78, ambientTemperature=320.0,
                             ambientTemperatureAmp='')

# Set absolute zero and Stefan-Boltzmann constant in model attributes (these must be
set for problems involving radiation)
heatModel.setValues(absoluteZero=-273.15, stefanBoltzmann=5.67E-8)

# ----------------------------------------------------------------------
# Create the mesh

import mesh

mesh_element_type = mesh.ElemType(elemCode=DC3D8, elemLibrary=STANDARD)

# One side of partition
beam_one_half_inside_xcoord=0.5
beam_one_half_inside_ycoord=0.5
beam_one_half_inside_zcoord=1.5
selectedBeamCells_one_half=beamCells.findAt((beam_one_half_inside_xcoord,
                                             beam_one_half_inside_ycoord,
                                             beam_one_half_inside_zcoord),)

# Other side of partition
beam_other_half_inside_xcoord=0.5
beam_other_half_inside_ycoord=0.5
beam_other_half_inside_zcoord=4.5
selectedBeamCells_other_half=beamCells.findAt((beam_other_half_inside_xcoord,
                                               beam_other_half_inside_ycoord,
                                               beam_other_half_inside_zcoord),)

# Combine both partitions into one region
beamMeshRegion=(selectedBeamCells_one_half,selectedBeamCells_other_half)

# Set the element type
beamPart.setElementType(regions=beamMeshRegion, elemTypes=(mesh_element_type,))

# Seed the part
beamPart.seedPart(size=0.25, deviationFactor=0.1)

# Generate the mesh
beamPart.generateMesh()
```

```
# -----------------------------------------------------------------------
# Create and run the job

import job

# Create the job

mdb.Job(name='HeatTransferJob', model='Heat Transfer', type=ANALYSIS,
                        description='Job simulates heat conduction through beam')

# Run the job
mdb.jobs['HeatTransferJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['HeatTransferJob'].waitForCompletion()

# End of run job

# -----------------------------------------------------------------------
# Post processing

import visualization

heattransfer_viewport = session.Viewport(name='Beam Results Viewport')
heattransfer_Odb_Path = 'HeatTransferJob.odb'
an_odb_object = session.openOdb(name=heattransfer_Odb_Path)
heattransfer_viewport.setValues(displayedObject=an_odb_object)

# display nodal temperatures
heattransfer_viewport.odbDisplay.setPrimaryVariable(variableLabel='NT11',
                                                    outputPosition=NODAL)

# plot these nodal temperatures as contours
heattransfer_viewport.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))

# Move camera angle to adjust view
# Possible values are 'Front', 'Back', 'Top', 'Bottom', 'Left', `Right', 'Iso',
# 'User-1', 'User-2', 'User-3', and 'User-4'.
heattransfer_viewport.view.setValues(session.views['Top'])
heattransfer_viewport.view.setValues(session.views['Right'])

# Save a view as user-1
session.View(name='User-1', nearPlane=9.0695, farPlane=15.588, width=5.116,
             height=3.0197, projection=PERSPECTIVE,
             cameraPosition=(7.258, 6.2162, 11.582),
             cameraUpVector=(-0.29634, 0.88595, -0.35675),
             cameraTarget=(0.5, 0.5, 3), viewOffsetX=0, viewOffsetY=0, autoFit=OFF)
```

11.4 Examining the Script

Let's examine the script line by line.

11.4.1 Initialization, creation of the model, part, materials, sections and assembly

Most of these statements are practically identical to those used in the Cantilever Beam example. The code has essentially been copied and pasted here with few modifications to the variable names. The one change however is with the material. This time 'Copper' is used. We do not specify the density, Young's modulus or Poisson's ratio, instead we specify the conductivity.

```
beamMaterial.Conductivity(table=((400.0, ),  ))
```

The **Conductivity()** method creates a **Conductivity** object, which is an object that specifies thermal conductivity. It has one required argument **table** which is a sequence of sequences of Floats. The data expected here depends on the type of conductivity. If one uses the optional argument **type**, one can specify **ISOTROPIC**, **ORTHOTROPIC** and **ANISOTROPIC**. In our case we do not specify the type hence it defaults to **ISOTROPIC**. For this type of conductivity, the **table** argument expects temperature dependent conductivity data. In our case we only specify one conductivity, but since **table** needs to be a sequence of sequences we write it as ((400.0,),).

11.4.2 Create a datum plane and partition the part

The following block creates the assembly.

```
# -----------------------------------------------------------------------
# Create the datum plane and partition the part

vertex_at_one_end_coords = (0.0, 0.0, 0.0)
vertex_at_one_end = beamInstance.vertices.findAt((vertex_at_one_end_coords,))

vertex_at_other_end_coords = (0.0, 0.0, 6.0)
vertex_at_other_end = beamInstance.vertices.findAt((vertex_at_other_end_coords,))

beamPart.DatumPlaneByTwoPoint(point1=vertex_at_one_end_coords,
                              point2=vertex_at_other_end_coords)

beam_inside_xcoord=0.5
beam_inside_ycoord=0.5
beam_inside_zcoord=3.0
beamCells=beamPart.cells
selectedBeamCells=beamCells.findAt((beam_inside_xcoord,beam_inside_ycoord,
                                                        beam_inside_zcoord),)
```

```
beamPart.PartitionCellByDatumPlane(datumPlane=beamPart.datums[3],
                                   cells=selectedBeamCells)
```

```
vertex_at_one_end_coords = (0.0, 0.0, 0.0)
vertex_at_one_end = beamInstance.vertices.findAt((vertex_at_one_end_coords,))
```

These two statements use the **findAt()** method to find a vertex at one end of the beam.

```
vertex_at_other_end_coords = (0.0, 0.0, 6.0)
vertex_at_other_end = beamInstance.vertices.findAt((vertex_at_other_end_coords,))
```

These statements find the corresponding point at the other end of the beam.

```
beamPart.DatumPlaneByTwoPoint(point1=vertex_at_one_end_coords,
                              point2=vertex_at_other_end_coords)
```

This statement uses the **DatumPlaneByTwoPoint()** method to create a **Feature** object and **DatumPlane** object. (A **DatumPlane** object has no direct constructor and is created when a **Feature** object is created). Two points are required as arguments and these can be **vertices, meshnodes, datum** objects or **InterestingPoints** (such as midpoint of an edge). The datum plane is created midway between the two points and normal to the line connecting them.

```
beam_inside_xcoord=0.5
beam_inside_ycoord=0.5
beam_inside_zcoord=3.0
beamCells=beamPart.cells
selectedBeamCells=beamCells.findAt((beam_inside_xcoord,beam_inside_ycoord,
                                    beam_inside_zcoord),)
```

In order to partition the beam using the datum plane we just created, we need to first select all its cells. We refer to all the cells in the part file as **beamPart.cells**. We then pick a point (0.5, 0.5, 3.0) which is right in the middle of the beam and use the **findAt()** method to select the cells of the beam.

```
beamPart.PartitionCellByDatumPlane(datumPlane=beamPart.datums[3],
                                   cells=selectedBeamCells)
```

The **PartitionCellByDatumPlane()** method partitions cells using a datum plane. The cells and the datum plane are provided as arguments to the method.

11.4.3 Create steps

The following block creates the step.

```
# --------------------------------------------------------------------
# Create the step

import step

# Create a heat transfer step
heatModel.HeatTransferStep(name='Heating Step', previous='Initial',
                           description='Apply heat in this step',
                           response=STEADY_STATE, amplitude=RAMP)
```

The **HeatTransferStep()** method creates a **HeatTransferStep** object which can control uncoupled heat transfer for either transient or steady-state response (which is what we are simulating). The required arguments are **name**, which is a String specifying the repository key you wish to assign the step, and **previous** which is the repository key of the step which occurs before it. **description** is an optional String argument.

11.4.4 Apply constraints/boundary conditions

The following block applies the constraints:

```
# --------------------------------------------------------------------
# Apply boundary conditions

end_face_1_pt = (0.5, 0.5, 0.0)
end_face_1 = beamInstance.faces.findAt((end_face_1_pt,))
end_face_1_region=regionToolset.Region(faces=end_face_1)
heatModel.TemperatureBC(name='Const Temp Surf 1', createStepName='Heating Step',
                        region=end_face_1_region,  distributionType=UNIFORM,
                        fieldName='', magnitude=400.0, amplitude=UNSET)

end_face_2_pt = (0.5, 0.5, 6.0)
end_face_2 = beamInstance.faces.findAt((end_face_2_pt,))
end_face_2_region=regionToolset.Region(faces=end_face_2)
heatModel.TemperatureBC(name='Const Temp Surf 2', createStepName='Heating Step',
                        region=end_face_2_region, distributionType=UNIFORM,
                        fieldName='', magnitude=350.0, amplitude=UNSET)
```

Much of this should look similar to the boundary condition code in the previous chapters. You can refer back to section 4.3.11 on page 81 of the Cantilever Beam example where **faces.findAt()** and **regionToolset.Region()** were used. The only new method here is the **TemperatureBC()** method.

```
heatModel.TemperatureBC(name='Const Temp Surf 1', createStepName='Heating Step',
                              region=end_face_1_region,  distributionType=UNIFORM,
                              fieldName='', magnitude=400.0, amplitude=UNSET)
```

This statement creates a **TemperatureBC** object. **name**, **createStepName** and **region** are required arguments. **name** is a String specifying the repository key for the boundary condition. **createStepName** is a String which specifies the name of the step in which the boundary condition must be created. **region** refers to the region on which the boundary condition is applied – it must be a **Region** object. The other arguments used here are optional ones. **magnitude** is a Float specifying the temperature magnitude. **distributionType** is a SymbolicConstant specifying the spatial distribution of the boundary condition. We set it to the default of **UNIFORM**. Other possible values are **USER_DEFINED** and **FIELD**. If we were to use **FIELD**, **fieldName** would be a String referring to the **AnalyticalField** object. **amplitude** can be either a String specifying the name of the amplitude reference, or the SymbolicConstant **UNSET**.

11.4.5 Apply loads

The following block applies the loads:

```
# -----------------------------------------------------------------------
# Apply loads

flux_face_pt = (0.5, 1.0, 1.5)
flux_face = beamInstance.faces.findAt((flux_face_pt,))
flux_face_region=regionToolset.Region(side1Faces=flux_face)
heatModel.SurfaceHeatFlux(name='Heat Flux', createStepName='Heating Step',
                           region=flux_face_region, magnitude=5000.0)
```

```
flux_face_pt = (0.5, 1.0, 1.5)
flux_face = beamInstance.faces.findAt((flux_face_pt,))
flux_face_region=regionToolset.Region(side1Faces=flux_face)
```

In order to apply the heat flux on the surface of the beam we need to first assign that surface to a region. Since the beam has been partitioned, we need to make sure the coordinates we supply to the **findAt()** method lie on the correct side of the partition or we will select the wrong surface. The **Region()** method then creates a **Region** object out of this surface which can then be used in the next statement.

```
heatModel.SurfaceHeatFlux(name='Heat Flux', createStepName='Heating Step',
                           region=flux_face_region, magnitude=5000.0)
```

The **SurfaceHeatFlux()** method creates a **SurfaceHeatFlux** object which describes the surface heat flux into or out of a region. The **SurfaceHeatFlux** object is derived from the

Load object. The four arguments that have been passed to the **SurfaceHeatFlux()** method are required arguments. **name** is a String specifying the load repository key. **createStepName** is a String specifying which step the load must be applied in. **region** is a **Region** object which specifies the region on which the load must be applied. We created **flux_face_region** in the previous statement to use here. **magnitude** is a Float specifying the heat flux magnitude. If however the optional **distributionType** argument is used to specify how the spatial distribution of the surface heat flux then **magnitude** is no longer a required argument. The Abaqus Scripting Reference describes other optional arguments.

11.4.6 Create interactions

The following block applies the loads:

```
# -----------------------------------------------------------------------
# Create interactions (convection and radiation)

# Convection
convection_face_pt = (0.5, 1.0, 4.5)
convection_face = beamInstance.faces.findAt((convection_face_pt,))
convection_face_region=regionToolset.Region(side1Faces=convection_face)
heatModel.FilmCondition(name='Convection', createStepName='Heating Step',
                        surface=convection_face_region, definition=EMBEDDED_COEFF,
                        filmCoeff=13.0, filmCoeffAmplitude='',
                        sinkTemperature=200.0, sinkAmplitude='')

# Radiation
radiation_face_pt = (0.0, 0.5, 4.5)
radiation_face = beamInstance.faces.findAt((radiation_face_pt,))
radiation_face_region=regionToolset.Region(side1Faces=radiation_face)
heatModel.RadiationToAmbient(name='Radiation', createStepName='Heating Step',
                             surface=radiation_face_region, radiationType=AMBIENT,
                             distributionType=UNIFORM, field='',
                             emissivity=0.78, ambientTemperature=320.0,
                             ambientTemperatureAmp='')

# Set absolute zero and Stefan-Boltzmann constant in model attributes (these must be
set for problems involving radiation)
heatModel.setValues(absoluteZero=-273.15, stefanBoltzmann=5.67E-8)
```

Many of the statements are similar to those used to apply the heat flux.

```
convection_face_pt = (0.5, 1.0, 4.5)
convection_face = beamInstance.faces.findAt((convection_face_pt,))
convection_face_region=regionToolset.Region(side1Faces=convection_face)
```

To define convection on one of the surfaces of the beam, we first assign that surface to a region. Keeping in mind the presence of a partition, the coordinates to get the correct surface are supplied to the **findAt()** method. The **Region()** method then creates a **Region** object out of this surface which can then be used in the **FilmCondition()** method.

```
heatModel.FilmCondition(name='Convection', createStepName='Heating Step',
                        surface=convection_face_region, definition=EMBEDDED_COEFF,
                        filmCoeff=13.0, filmCoeffAmplitude='',
                        sinkTemperature=200.0, sinkAmplitude='')
```

The **FilmCondition()** method creates a **FilmCondition** object which defines the film coefficients and sink temperatures in an analysis. The **FilmCondition** object is derived from the **Interaction** object. The four required arguments are **name**, **createStepName**, **surface** and **definition**. **name** is a String used as the repository key, **createStepName** is a String identifying the name of the step in which the **FilmCondition** object should be created, **surface** is a **Region** object indicating which surface the film condition interaction should be applied to, and **definition** is a SymbolicConstant specifying how it will be defined with possible values of **EMBEDDED_COEFF**, **PROPERTY_REF**, **USER_SUB**, and **FIELD**. **filmCoeff**, **filmCoeffAmplitude**, **sinkTemperature** and **sinkAmplitude** are optional arguments. **filmCoeff** is a Float specifying the convection coefficient, its default value is 0.0. **filmCoeffAmplitude** is a String specifying the name of the Amplitude object (if any) for the variation of film coefficient with time. The default value is an empty String. **sinkTemperature** is a Float specifying the reference ambient temperature, it defaults to 0.0. **sinkAmplitude** is a String specifying the name of the Amplitude object (if any) for the variation of sink temperature with time. The default is an empty String. The Abaqus Scripting Reference describes other optional arguments.

```
radiation_face_pt = (0.0, 0.5, 4.5)
radiation_face = beamInstance.faces.findAt((radiation_face_pt,))
radiation_face_region=regionToolset.Region(side1Faces=radiation_face)
```

Similar to convection, to define radiation from/to part of the beam, we first assign that surface to a region. Accounting for the presence of a partition, the coordinates to get the correct surface are supplied to the **findAt()** method. The **Region()** method then creates a **Region** object out of this surface which can then be used in the **RadiationToAmbient()** method.

```
heatModel.RadiationToAmbient(name='Radiation', createStepName='Heating Step',
                             surface=radiation_face_region, radiationType=AMBIENT,
                             distributionType=UNIFORM, field='',
                                    emissivity=0.78, ambientTemperature=320.0,
                                    ambientTemperatureAmp='')
```

The **RadiationToAmbient()** method creates a **RadiationToAmbient** object which defines the heat transfer between a surface and its environment. The **RadiationToAmbient** object is derived from the Interaction object. The four required arguments are **name, createStepName, surface** and **emissivity**. **name** is a String used as the repository key, **createStepName** is a String identifying the name of the step in which the **RadiationToAmbient** object should be created, **surface** is a **Region** object indicating which surface the radiation interaction should be applied to, and **emissivity** is a Float specifying the emissivity. **radiationType, distributionType, field, ambientTemperature** and **ambientTemperatureAmp** are optional arguments. **radiationType** is a SymbolicConstant specifying whether Abaqus should use surface radiation behavior **(AMBIENT)** or cavity radiation approximation **(CAVITY). distributionType** is a SymbolicConstant specifying the radiation distribution when **radiationType** is set to **AMBIENT**. The possible values are **UNIFORM** (the default) and **ANALYTICAL_FIELD. field** is a String specifying the name of the **AnalyticalField** object and when **distributionType** = **ANALYTICAL_FIELD**. The default is an empty String. **ambientTemperature** is a Float which sets the reference ambient temperature (default is 0.0) when **radiationType** = **AMBIENT**. **ambientTemperatureAmp** is a String indicating the name of the Amplitude object which defines the variation of temperature with time. The Abaqus Scripting Reference describes other optional arguments.

```
heatModel.setValues(absoluteZero=-273.15, stefanBoltzmann=5.67E-8)
```

The **[model].setValue()** method is used to modify a **Model** object. It has no required arguments and a number of optional ones spelled out in the Abaqus Scripting Reference Manual. The ones we have used are **absoluteZero** and **stefanBoltzmann. absoluteZero** is a Float specifying the value of absolute zero on the temperature scale being used. Since we are using Celsius as the unit of temperature throughout the analysis, we set absoluteZero to -273.15 ˚C. **stefanBoltzmann** is a Float specifying the Stefan-Boltzmann constant. To remain consistent with the rest of the model which is in SI units we give it a value of 5.67E-8 J/s/m^2/ ˚C^4

11.4.7 Mesh

The following block creates the mesh

```
# -----------------------------------------------------------------------
# Create the mesh

import mesh

mesh_element_type = mesh.ElemType(elemCode=DC3D8, elemLibrary=STANDARD)

# One side of partition
beam_one_half_inside_xcoord=0.5
beam_one_half_inside_ycoord=0.5
beam_one_half_inside_zcoord=1.5
selectedBeamCells_one_half=beamCells.findAt((beam_one_half_inside_xcoord,
                                             beam_one_half_inside_ycoord,
                                             beam_one_half_inside_zcoord),)


# Other side of partition
beam_other_half_inside_xcoord=0.5
beam_other_half_inside_ycoord=0.5
beam_other_half_inside_zcoord=4.5
selectedBeamCells_other_half=beamCells.findAt((beam_other_half_inside_xcoord,
                                               beam_other_half_inside_ycoord,
                                               beam_other_half_inside_zcoord),)

# Combine both partitions into one region
beamMeshRegion=(selectedBeamCells_one_half,selectedBeamCells_other_half)

# Set the element type
beamPart.setElementType(regions=beamMeshRegion, elemTypes=(mesh_element_type,))


# Seed the part
beamPart.seedPart(size=0.25, deviationFactor=0.1)

# Generate the mesh
beamPart.generateMesh()
```

`mesh_element_type = mesh.ElemType(elemCode=DC3D8, elemLibrary=STANDARD)`

You have seen the **ElemType()** method in previous examples such as section 4.3.12 on page 83 of the Cantilever Beam example. To jog your memory, **the ElemType()** method creates an **ElementType** object which can later be used as an argument to the **setElementType()** method. The only required argument is **elemCode** which is a symbolic constant in Abaqus that specifies the element code. We use **DC3D8** which is an 8-node linear brick heat transfer element.

```
# One side of partition
beam_one_half_inside_xcoord=0.5
beam_one_half_inside_ycoord=0.5
beam_one_half_inside_zcoord=1.5
selectedBeamCells_one_half=beamCells.findAt((beam_one_half_inside_xcoord,
                                             beam_one_half_inside_ycoord,
                                             beam_one_half_inside_zcoord),)
```

Remember that we have partitioned the beam. In order to mesh it we will need to select the cells on both sides of the partition. The above statements select the cells on one side of the partition. The coordinates supplied to the **findAt()** method are (0.5,0.5,1.5) which is the center of this side of the partition. The variable **selectedBeamCells_one_half** refers to the cells on this side of the partition.

```
# Other side of partition
beam_other_half_inside_xcoord=0.5
beam_other_half_inside_ycoord=0.5
beam_other_half_inside_zcoord=4.5
selectedBeamCells_other_half=beamCells.findAt((beam_other_half_inside_xcoord,
                                               beam_other_half_inside_ycoord,
                                               beam_other_half_inside_zcoord),)
```

These statements repeat the procedure for the other side of the partition. The variable **selectedBeamCells_other_half** refers to the cells on this other side.

```
# Combine both partitions into one region
beamMeshRegion=(selectedBeamCells_one_half,selectedBeamCells_other_half)
```

Now that we have the cells on both sides of the partition stored in two variables **selectedBeamCells_one_half** and **selectedBeamCells_other_half**, we can create a **Region** object which consists of all the cells by separating them with a comma and putting parentheses around them.

```
# Set the element type
beamPart.setElementType(regions=beamMeshRegion, elemTypes=(mesh_element_type,))

# Seed the part
beamPart.seedPart(size=0.25, deviationFactor=0.1)

# Generate the mesh
beamPart.generateMesh()
```

You have seen the **setElementType()**, **seedPart()** and **generateMesh()** methods in previous examples and can refer to section 4.3.12 on page 83 of the Cantilever Beam example for an explanation.

11.4.8 Create and run the job

The following code runs the job

```
# ------------------------------------------------------------------------
# Create and run the job

import job

# Create the job

mdb.Job(name='HeatTransferJob', model='Heat Transfer', type=ANALYSIS,
                         description='Job simulates heat conduction through beam')

# Run the job
mdb.jobs['HeatTransferJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['HeatTransferJob'].waitForCompletion()

# End of run job
```

These statements are similar to ones used previously. You may refer back to Section 4.3.13,on page 88.

11.4.9 Post Processing

The following code displays the nodal temperatures as a contour plot. It also changes the camera angle to view the beam from the left.

```
# ------------------------------------------------------------------------
# Post processing

import visualization

heattransfer_viewport = session.Viewport(name='Beam Results Viewport')
heattransfer_Odb_Path = 'HeatTransferJob.odb'
an_odb_object = session.openOdb(name=heattransfer_Odb_Path)
heattransfer_viewport.setValues(displayedObject=an_odb_object)

# display nodal temperatures
heattransfer_viewport.odbDisplay.setPrimaryVariable(variableLabel='NT11',
                                                    outputPosition=NODAL)

# plot these nodal temperatures as contours
heattransfer_viewport.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))

# Move camera angle to adjust view
# Possible values are 'Front', 'Back', 'Top', 'Bottom', 'Left', `Right', 'Iso',
# 'User-1', 'User-2', 'User-3', and 'User-4'.
heattransfer_viewport.view.setValues(session.views['Top'])
heattransfer_viewport.view.setValues(session.views['Right'])

# Save a view as user-1
session.View(name='User-1', nearPlane=9.0695, farPlane=15.588, width=5.116,
             height=3.0197, projection=PERSPECTIVE,
             cameraPosition=(7.258, 6.2162, 11.582),
             cameraUpVector=(-0.29634, 0.88595, -0.35675),
             cameraTarget=(0.5, 0.5, 3), viewOffsetX=0, viewOffsetY=0, autoFit=OFF)
```

```
heattransfer_viewport = session.Viewport(name='Beam Results Viewport')
heattransfer_Odb_Path = 'HeatTransferJob.odb'
an_odb_object = session.openOdb(name=heattransfer_Odb_Path)
heattransfer_viewport.setValues(displayedObject=an_odb_object)
```

These statements are almost identical to the ones used in the Cantilever Beam example. Refer back to section 4.3.14 on page 89 to review the **Viewport()**, **openOdb()** and **viewport.setValues()** methods.

```
heattransfer_viewport.odbDisplay.setPrimaryVariable(variableLabel='NT11',
                                                     outputPosition=NODAL)
```

The **setPrimaryVariable()** method tells Abaqus which of the field output variables you wish to explore. In the next statement when we use **display.setValues()** to plot a contour plot Abaqus will plot the primary variable. **variableLabel**, **field**, and **outputPosition** are required arguments. **variableLabel** is a String referring to the field output variable whereas **field** is a String specifying the **FieldOutput** object. You must use either **variableLabel** or **field** but not both together. **outputPosition** specifies what position to obtain the data from using a SymbolicConstant such as **NODAL** or **ELEMENT_FACE**, see the Abaqus Scripting Reference for other options.

You are probably wondering how we knew the variable label is called 'NT11'. Abaqus uses the nodal variable NTxx. For continuum elements there is only one temperature value per node hence only NT11 is applicable. For shells and beams a temperature gradient can exist through the shell thickness or beam cross-section hence other variables such as NT12 and NT13 specify the gradients in the local 1- and 2-directions and so on. However the easiest way to figure out the variable label is the run the simulation once in CAE, and in the **Visualization** mode go to **Result > Field Output**. The output variable names for the simulation such as NT11, RFL11 etc are listed here in the first column of the Field Output table.

```
heattransfer_viewport.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))
```

The **display.setValues()** method was also used in the Cantilever Beam example. Refer back to section 4.3.14 on page 89 to review it. The only difference here is that **plotState** has been set to a different SymbolicConstant **CONTOURS_ON_DEF** instead of **DEFORM** in order to plot a color contour of NT11.

```
heattransfer_viewport.view.setValues(session.views['Top'])
heattransfer_viewport.view.setValues(session.views['Right'])
```

(viewport).view.setValues() allows you to set the camera angle for viewing the model in the visualization viewport. Here we set it to top view using **sessions.views['Top']** and to right view using **sessions.views['Right']**. The possible values are **Front**, **Back**, **Top**, **Bottom**, **Left**, **Right**, **Iso**, **User-1**, **User-2**, **User-3** and **User-4**. One way to see the list of **sessions.views** available is to go to the Kernal Command Line Interface (below your viewport in Abaqus) and type '*print session.views*'. Here is what you see:

>>> print session.views

{'Front': 'SavedView object', 'Back': 'SavedView object', 'Top': 'SavedView object', 'Bottom': 'SavedView object', 'Left': 'SavedView object', 'Right': 'SavedView object', 'Iso': 'SavedView object', 'User-1': 'SavedView object', 'User-2': 'SavedView object', 'User-3': 'SavedView object', 'User-4': 'SavedView object'}

Needless to say, since we asked Abaqus to display a 'Right' view immediately after the 'Top' view, we will never actually see the top view. Since the GUI (Abaqus/CAE) procedure involved displaying both views, the equivalent code for both has been used in the script to maintain consistency.

For your information, the statements could also have been written as

```
session.viewports['Beam Results Viewport'].view.setValues(session.views['Top'])
session.viewports['Beam Results Viewport'].view.setValues(session.views['Right'])
```

if you didn't already have the viewport assigned to the variable **heattransfer_viewport**.

```
session.View(name='User-1', nearPlane=9.0695, farPlane=15.588, width=5.116,
             height=3.0197, projection=PERSPECTIVE,
             cameraPosition=(7.258, 6.2162, 11.582),
             cameraUpVector=(-0.29634, 0.88595, -0.35675),
             cameraTarget=(0.5, 0.5, 3), viewOffsetX=0, viewOffsetY=0, autoFit=OFF)
```

The **View()** method is used to save a certain view orientation to the 'User-1' button. It creates a new **View** object. All the arguments passed to the method here are required arguments. **name** is a String that names the view. **nearPlane** is a Float that sets the distance from the camera to the clipping plane and must be > 0.0. **farPlane** is a Float that sets the distance from the camera to the far clipping plane when **farPlaneMode** is set to **SPECIFY**. It is required that **farPlane** > **nearPlane**. **width** and **height** are Floats specifying the width and height of the front clipping plane and must be > 0.0. **projection** is a SymbolicConstant defining the projection mode – either **PERSPECTIVE** or **PARALLEL**. **cameraPosition** is a sequence of 3 Floats which specify the position of the camera using the global coordinate system. **cameraUpVector** is a sequence of 3 Floats specifying the positive Y-axis of the camera body, it allows you to partially orient the camera. **cameraTarget** is a sequence of 3 Floats which specify what point the camera must look at. So **cameraPosition, cameraUpVector** and **cameraTarget** together specify the position and orientation of the camera. **viewOffsetX** and **viewOffsetY** are Floats which specify the amount to pan the model in the screen X- and Y-directions as a fraction of the viewport width and height respectively. Since the camera orientation and target have been set, the **viewOffsetX** and **viewOffsetY** are actually panning the display, hence no rotation is produced in the view. Positive values are up in the Y-direction and right in the X-direction.

11.5 Summary

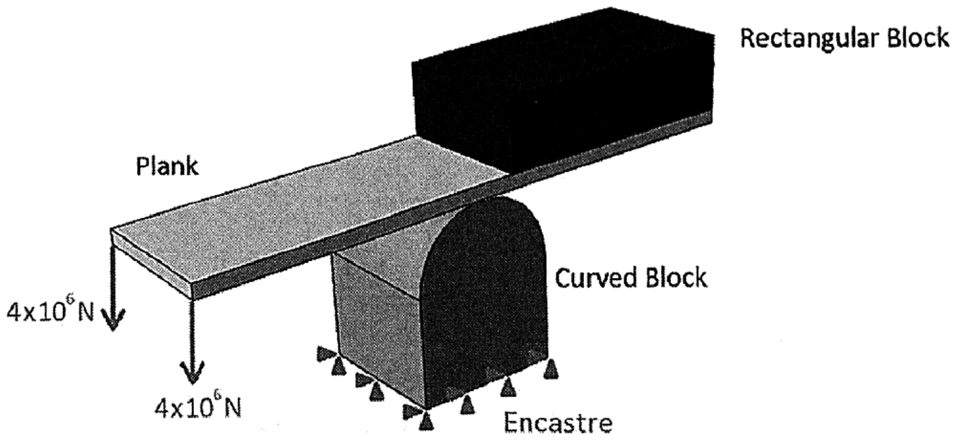
In this chapter we scripted a steady state heat transfer model. This included applying heat flux loads and constant temperature boundary conditions. You also learnt to change the primary variable in Abaqus/Viewer to plot a color contour and to change the camera angle. The heat transfer example used here was a very simple one, the aim was to introduce you to a few of the commands you are likely to use in a Python script. The Abaqus Scripting Reference explains in detail all of the options available to you for heat transfer analyses.

12

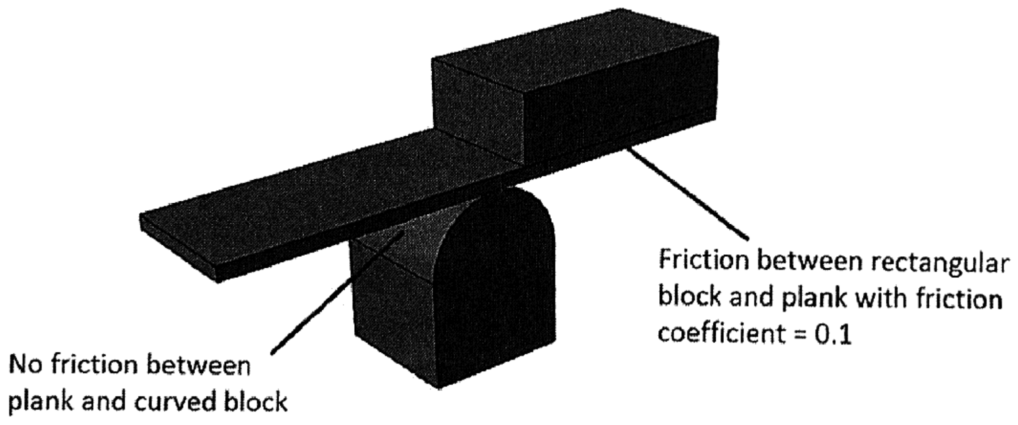
Contact Analysis (Contact Pairs Method)

12.1 Introduction

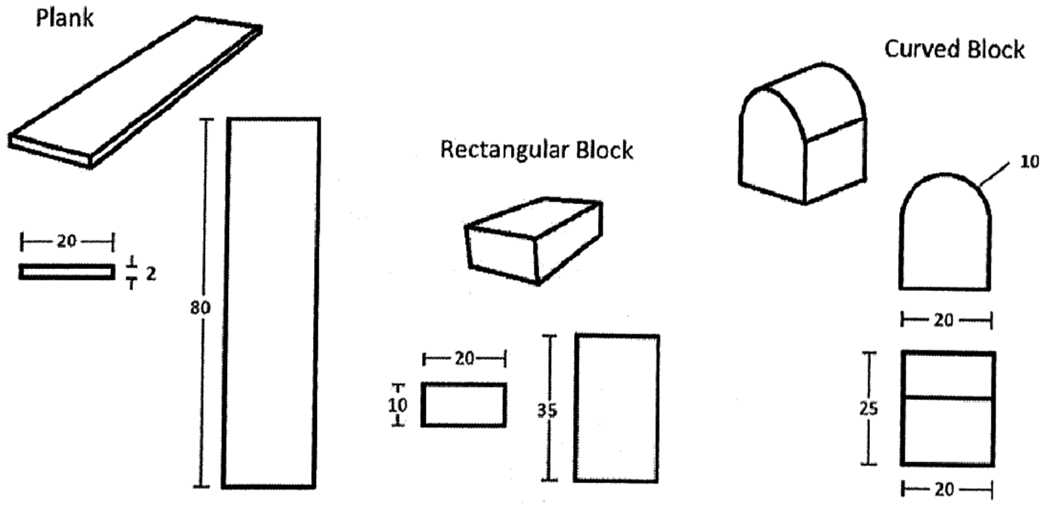
In this chapter we will perform a contact analysis. The problem is displayed in the figure. We will use the contact pairs method (as opposed to the general contact method).



We use frictional properties for the contact interaction between the rectangular block and the plank, and frictionless contact between the plank and the curved block.


Friction between rectangular block and plank with friction coefficient = 0.1
No friction between plank and curved block

The dimensions the parts are displayed in the figure. All dimensions are in SI with length in meters.



In this example the following tasks will be performed first using Abaqus/CAE, and then using a Python script.

- Create a part
- Assign materials
- Assign sections
- Create an Assembly using face to face constraints
- Create multiple steps
- Assign boundary conditions

- Assign loads
- Identify surfaces
- Assign interaction properties
- Create interactions
- Create a mesh
- Create and submit a job

The following new topics are covered in this example:

- Model / Preprocessing
    - Define surfaces in the assembly
    - Create interaction properties (specifically contact with and without friction)
    - Specify interaction pairs (contact surfaces)
- Results / Post-processing
    - Plot contact pressures to identify contact

12.2 Procedure in GUI

You can perform the simulation in Abaqus/CAE by following the steps listed below. You can either read through these, or watch the video demonstrating the process on the book website.

1. Rename **Model-1** to **Contact Simulation**
    a. Right-click on Model-1 in Model Database
    b. Choose **Rename..**
    c. Change name to **Contact Simulation**
2. Create the Plank
    a. Double-click on **Parts** in Model Database. **Create Part** window is displayed.
    b. Set **Name** to **Plank**
    c. Set **Modeling Space** to **3D**
    d. Set **Type** to **Deformable**
    e. Set **Base Feature Shape** to **Solid**
    f. Set **Base Feature Type** to **Extrusion**
    g. Set **Approximate Size** to **20**
    h. Click **OK.** You will enter Sketcher mode.

i. Use the **Create Lines: Rectangle (4 lines)** tool to draw the profile of the plank
j. Use the **Add Dimension** tool to set the width to 20 m and the thickness to 2 m.
k. Click **Done** to exit the sketcher. The **Edit Base Extrusion** window is displayed.
l. Set **Depth** to **80.0**
m. Click **OK**.

3. Create the Curved Block
   a. Double-click on **Parts** in Model Database. **Create Part** window is displayed.
   b. Set **Name** to **Curved Block**
   c. Set **Modeling Space** to **3D**
   d. Set **Type** to **Deformable**
   e. Set **Base Feature Shape** to **Solid**
   f. Set **Base Feature Type** to **Extrusion**
   g. Set **Approximate Size** to **20**
   h. Click **OK**. You will enter Sketcher mode.
   i. Use the **Create Circle: Center and Perimeter** tool to draw a circle
   j. Use the **Add Dimension** tool to set the radius to 10 m.
   k. Use the **Create Lines: Rectangle (4 lines)** tool to draw a rectangle
   l. Use the **Add Dimension** tool to set the height to 15 m
   m. Use the **Auto Trim** tool to trim out parts of the circle and the rectangle to create the desired profile
   n. Click **Done** to exit the sketcher. The **Edit Base Extrusion** window is displayed.
   o. Set **Depth** to **20.0**
   p. Click **OK**.
4. Create the Rectangular Block
   a. Double-click on **Parts** in Model Database. **Create Part** window is displayed.
   b. Set **Name** to **Rectangular Block**
   c. Set **Modeling Space** to **3D**
   d. Set **Type** to **Deformable**
   e. Set **Base Feature Shape** to **Solid**
   f. Set **Base Feature Type** to **Extrusion**
   g. Set **Approximate Size** to **20**
   h. Click **OK**. You will enter Sketcher mode.

   i. Use the **Create Lines: Rectangle (4 lines)** tool to draw the profile of the plank
   j. Use the **Add Dimension** tool to set the width to 20 m and the height to 10 m.
   k. Click **Done** to exit the sketcher. The **Edit Base Extrusion** window is displayed.
   l. Set **Depth** to **35.0**
   m. Click **OK**.

5. Create the 2 materials
   a. Double-click on **Materials** in the Model Database. **Edit Material** window is displayed
   b. Set **Name** to **AISI 1005 Steel**
   c. Select **General > Density**. Set **Mass Density** to **7800** (which is 7.800 g/cc)
   d. Select **Mechanical > Elasticity > Elastic**. Set **Young's Modulus** to **200E9** (which is 200 GPa) and **Poisson's Ratio** to **0.29**.
   e. Again double-click on **Materials** in the Model Database. **Edit Material** window is displayed
   f. Set **Name** to **Aluminum 2024-T3**
   g. Select **General > Density**. Set **Mass Density** to **2770** (which is 2.770 g/cc)
   h. Select **Mechanical > Elasticity > Elastic**. Set **Young's Modulus** to **73.1E9** (which is 73.1 GPa) and **Poisson's Ratio** to **0.33**.
6. Assign sections
   a. Double-click on **Sections** in the Model Database. **Create Section** window is displayed
   b. Set **Name** to **Steel Section**
   c. Set **Category** to **Solid**
   d. Set **Type** to **Homogeneous**
   e. Click **Continue...** The **Edit Section** window is displayed.
   f. In the **Basic** tab, set **Material** to **the AISI 1005 Steel** which was defined in the create material step.
   g. Click **OK**.
   h. Again double-click on **Sections** in the Model Database. **Create Section** window is displayed
   i. Set **Name** to **Aluminum Section**
   j. Set **Category** to **Solid**
   k. Set **Type** to **Homogeneous**
   l. Click **Continue...** The **Edit Section** window is displayed.

   m. In the **Basic** tab, set **Material** to **the Aluminum 2024 – T3** which was defined in the material creation step.
   n. Click **OK**.
7. Assign the sections to the parts
   a. Expand the **Parts** container in the Model Database. Expand the part **Plank**.
   b. Double-click on **Section Assignments**
   c. You see the message **Select the regions to be assigned a section** displayed below the viewport
   d. Click and drag with the mouse to select the entire plank.
   e. Click **Done**. The **Edit Section Assignment** window is displayed.
   f. Set **Section** to **Aluminum Section**.
   g. Click **OK**.
   h. Similarly assign Steel Section to the curved block and the rectangular block.
8. Create the Assembly
   a. Double-click on **Assembly** in the Model Database. The viewport changes to the **Assembly Module**.
   b. Expand the **Assembly** container.
   c. Double-click on **Instances**. The **Create Instance** window is displayed.
   d. Set **Parts** to **Plank**
   e. Set **Instance Type** to **Dependent (mesh on part)**
   f. Click **OK**. The plank is instanced in the assembly.
   g. Again double-click on **Instances**. The **Create Instance** window is displayed.
   h. Set **Parts** to **Curved block**
   i. Set **Instance Type** to **Dependent (mesh on part)**
   j. Check **Auto-offset from other instances**
   k. Click **OK**. The curved block is instanced in the assembly.
   l. Click the **Create Constraint: Face to Face** tool. You see the message **Select a planar face or datum plane of the movable instance** below the viewport.
   m. Click the bottom face of the curved block. You see the message **Select a planar face or datum plane of the fixed instance** below the vieport
   n. Click the bottom face of the plank. Arrows appear on the faces and you see the message **The instance will be moved so that the arrows point in the same direction** below the viewport.
   o. Click **OK** or **Flip** as required to have the arrows pointing in the same direction. You see the prompt **Distance from the fixed plane along its normal** below the viewport.

p. Set it to **25.0**
q. Similarly use face to face constraints on the other two surfaces to align the parts as displayed in the figure.
r. Again double-click on **Instances.** The **Create Instance** window is displayed.
s. Set **Parts** to **Rectangular block**
t. Set **Instance Type** to **Dependent (mesh on part)**
u. Check **Auto-offset from other instances**
v. Click **OK**. The rectangular block is instanced in the assembly.
w. Use the **Create Constraint: Face to Face** tool 3 more times to align the parts as shown in the figure.

9. Create Steps
    a. Double-click on **Steps** in the Model Database. The **Create Step** window is displayed.
    b. Set **Name** to **Make Contact**
    c. Set **Insert New Step After** to **Initial**
    d. Set **Procedure Type** to **General > Static, General**
    e. Click **Continue..** The **Edit Step** window is displayed
    f. In the **Basic** tab, set **Description** to **Push parts together to avoid chatter**.
    g. Click **OK.**
    h. Once again double-click on **Steps** in the Model Database. The **Create Step** window is displayed.
    i. Set **Name** to **Apply Force**
    j. Set **Insert New Step After** to **Initial**
    k. Set **Procedure Type** to **General > Static, General**
    l. Click **Continue..** The **Edit Step** window is displayed
    m. In the **Basic** tab, set **Description** to **Apply force on one end of the plank.**
    n. Click **OK.**
10. Leave Field Output Requests at defaults
11. Leave History Output Requests at defaults
12. Apply boundary conditions
    a. Double-click on **BCs** in the Model Database. The **Create Boundary Condition** window is displayed
    b. Set **Name** to **Fix Plank End**
    c. Set **Step** to **Make Contact**
    d. Set **Category** to **Mechanical**
    e. Set **Types for Selected Step** to **Symmetry/Antisymmetry/Encastre**

f. Click **Continue…**

g. You see the message **Select regions for the boundary condition** displayed below the viewport

h. Select the end face of the shaft.

i. Click **Done.** The **Edit Boundary Condition** window is displayed.

j. Choose **ENCASTRE (U1=U2=U3=UR1=UR2=UR3=0).**

k. Click **OK.**

l. Similarly create a second boundary condition called **Fix Curved Block,** applied during the **Make Contact** step with **ENCASTRE.** This is applied to the bottom of the curved block.

m. Create a third boundary condition called **Fix Rectangular Block,** applied during the **Make Contact** step with **ENCASTRE.** This is applied to the end face of the rectangular block.

n. Create a forth boundary condition called **Press Plank Curved,** applied during the **Make Contact** step. Set **Type for Selected Step** to **Displacement/Rotation.** Select the top surface of the plank to apply the boundary condition. When the **Edit Boundary Condition** window is displayed, set **U2** to **-1E-8** and **U1=U3=UR1=UR2=UR3=0.** Click **OK**

o. Create a fifth boundary condition called **Press Rect Plank,** applied during the **Make Contact** step. Set **Type for Selected Step** to **Displaccement/Rotation.** Select the top surface of the rectangular block to apply the boundary condition. When the **Edit Boundary Condition** window is displayed, set **U2** to **-1E-8** and **U1=U3=UR1=UR2=UR3=0.** Click **OK**

p. Right click on **BCs** in the Model Database. Choose Manager… The Boundary Condition Manager window is displayed

q. For the boundary conditions **Press Plank Curved** and **Press Rect Plank,** deactivate both in the **Apply Force** step using the **Deactivate** button. For the boundary condition **Fix Curved Block** use the **Move Left** button to activate it in the **Initial** step. For the boundary conditions **Fix Plank End** and **Fix Rectangular Block** use the **Move Right** button to activate them in the **Apply Force** step. The table should look as follows:

Name	Initial	Make Contact	Apply Force
Fix Curved Block	Created	Propagated	Propagated
Fix Plank End			Created
Fix Rectangular Block			Created
Press Plank Curved		Created	Inactive

Press Rect Plank		Created	Inactive

r. Click **Dismiss**

13. Assign Loads
    a. Double-click on **Loads** in the Model Database. The **Create Load** window is displayed
    b. Set **Name** to **Concentrated forces at corners**
    c. Set **Step** to **Apply Force**
    d. Set **Category** to **Mechanical**
    e. Set **Type for Selected Step** to **Concentrated Force**
    f. Click **Continue…**
    g. You see the message **Select points for the load displayed below the viewport**
    h. Select the two corners of the plank by clicking on them. You will need to use the "Shift" key on the keyboard to select both of them.
    i. Click **Done**. The **Edit Load** window is displayed
    j. Set **CF2** to **-4E6**to apply a 4000000 N force in downward (negative Y) direction
    k. Click **OK**
    l. You will see the forces displayed with arrows in the viewport on the selected nodes
14. Assign surfaces
    a. Expand the **Assembly** container in the Model Database.
    b. Double-click on **Surfaces**. The **Create Surface** window is displayed
    c. Set **Name** to **rect block bottom**
    d. Click **Continue…** You see the message **Select the regions for the surface** displayed below the viewport
    e. Set it to **individually** with the dropdown menu
    f. Select the bottom surface of the rectangular block. You might need to suppress the face to face relationship between the rectangular block and the plank in order to make the bottom surface visible. Then resume the relationship.
    g. Similarly assign the surface **curved block top** to the top surface of the curved block
    h. Similarly assign the surface **plank bottom** to the bottom surface of the plank
    i. Similarly assign the surface **plank top** to the top surface of the plank
15. Assign interaction properties

a. Double click **Interaction Properties** in the Model Database
b. Set **Name** to **Frictionless**
c. Set **Type** to **Contact**
d. Click **Continue...** The **Edit Contact Property** window is displayed
e. Select **Mechanical** > **Tangential Behavior**. It is added to the **Contact Property Options** list.
f. Set **Frictional formulation** to **Frictionless**
g. Once again double click **Interaction Properties** in the Model Database
h. Set **Name** to **Frictional**
i. Set **Type** to **Contact**
j. Click **Continue...** The **Edit Contact Property** window is displayed
k. Select **Mechanical** > **Tangential Behavior**. It is added to the **Contact Property Options** list.
l. Set **Frictional formulation** to **Penalty**
m. Set **Friction Coeff** to **0.1**
n. Click **OK**

16. Create interactions
    a. Double click **Interactions**in the Model Database
    b. Set **Name** to **Curved Plank Interaction**
    c. Set **Step** to **Apply Force**
    d. Click **Continue...**
    e. You see the message **Select the master surface** displayed below the viewport
    f. Set it to **individually**
    g. Click the **Surfaces...** button. The **Region Selection** window is displayed.
    h. Choose **curved block top**
    i. Click **Continue...**
    j. You see the prompt **Select the slave type** displayed below the viewport
    k. Click **Surface**. The **Region Selection** window is displayed
    l. Choose **plank bottom**
    m. Click **Continue...**The **Edit Interaction** window is displayed
    n. Set **Contact interaction property** to **Frictionless**. Leave all other options at default.
    o. Click **OK**
    p. Double click **Interactions**in the Model Database
    q. Set **Name** to **Rectangular Plank Interaction**

r. Set **Step** to **Apply Force**
s. Click **Continue…**
t. You see the message **Select the master surface from the dialog** displayed below the viewport. The **Region Selection** window is displayed.
u. Choose **rectangular block bottom**
v. Click **Continue...**
w. You see the prompt **Select the slave type** displayed below the viewport
x. Click **Surface**. The **Region Selection** window is displayed
y. Choose **plank top**
z. Click **Continue...**The **Edit Interaction** window is displayed
aa. Set **Contact interaction property** to **Frictional**. Leave all other options at default.
bb. Click **OK**

17. Create the mesh
    a. Expand the**Parts** container in the Model Database.
    b. Expand **Plank**
    c. Double-click on **Mesh (Empty)**. The viewport window changes to the **Mesh module** and the tools in the toolbar are now meshing tools.
    d. Using the menu bar click on **Mesh > Element Type …**
    e. You see the message **Select the regions to be assigned element types** displayed below the viewport
    f. Click and drag using your mouse to select the entire plank.
    g. Click **Done**. The **Element Type** window is displayed.
    h. Set **Element Library** to **Standard**
    i. Set **Geometric Order** to **Linear**
    j. Set **Family** to **3D Stress**
    k. Check **Reduced Integration**
    l. You will notice the message **C3D8R: An 8-node linear brick, reduced integration, hourglass control**
    m. Click **OK**
    u. Using the menu bar lick on **Seed >Part…**The **Global Seeds** window is displayed
    v. Set **Approximate global size** to **4**. Leave everything else at default values.
    w. Click **OK**.
    x. You see the message **Seeding definition complete** displayed below the viewport. Click **Done**.

- y. Using the menu bar click on **Mesh > Part**
- z. You see the prompt **OK to mesh the part?** displayed below the viewport
- n. Click **Yes**
- o. Repeat the same process for **Curved Block** and **Rectangular Block**

18. Create and submit the job
    - a. Double-click on **Jobs in** the Model Database. The **Create Job** window is displayed
    - b. Set **Name** to **ContactSimulationJob**
    - c. Set **Source** to **Model**
    - d. Select **Contact Simulation** (it is the only option displayed)
    - e. Click **Continue..** The **Edit Job** window is displayed
    - f. Set **Description** to **Run the contact simulation**
    - g. Set **Job Type** to **Full Analysis.**
    - h. Leave all other options at defaults
    - i. Click **OK**
    - j. Expand the**Jobs** container in the Model Database
    - k. Right-click on **ContactSimulationJob** and choose **Submit.** This will run the simulation. You will see the following messages in the message window:
      The job input file "ContactSimulationJob.inp" has been submitted for analysis.
      Job ContactSimulationJob: Analysis Input File Processor completed successfully
      Job ContactSimulationJob: Abaqus/Standard completed successfully
      Job ContactSimulationJobcompleted successfully
19. Plot mises stress and contact pressures
    - a. Right-click on **ContactSimulationJob (Completed)** in the Model Database. Choose **Results**. The viewport changes to the **Visualization** module.
    - b. In the toolbar click the **Plot Contours on Deformed Shape** tool. The Mises stresses are displayed on the deformed plank and on the rectangular and curved blocks.
    - c. Using the **Field Output** toolbar change **Primary** to **CPRESS**. The contact pressures are displayed on the parts.
    - d. Click the **Frame Selector** tool and use the slider to observe contact pressures over a few frames
    - e. In the **Display Group** toolbar click the **Replace Selected** tool
    - f. You see the message **Select entities to replace** displayed below the viewport

g. Set it to **Part instances** with the dropdown
h. Click the curved block in the viewport
i. Click **Done**. The view of the assembly in the viewport has been replaced with the curved block.
j. Use the slider of the **Frame Selector** tool to go to frame 6 (last frame) of the **Make Contact** step. You see contact pressures displayed on the curved block indicating that contact was established in this frame.
k. In the **Display Group** toolbar click the **Replace All** tool to bring back the view of the assembly in the viewport

12.3 Python Script

The following Python script replicates the above procedure for the contact simulations. You can find it in the source code accompanying the book in **contact.py**. You can run it by opening a new model in Abaqus/CAE (**File** > **New Model Database** > **With Standard/Explicit Model)** and running it with **File** > **Run Script…**

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# ---------------------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Contact Simulation')
contactModel = mdb.models['Contact Simulation']

# ---------------------------------------------------------------------
# Create the parts

import sketch
import part

# a) Plank

# i) Sketch the plank cross section using rectangle tool
plankProfileSketch = contactModel.ConstrainedSketch(name='Plank Sketch',
                                                     sheetSize=40)
plankProfileSketch.rectangle(point1=(-10.0,0.0), point2=(10.0,2.0))

# ii) Create a 3D deformable part named "Plank" by extruding the sketch
plankPart=contactModel.Part(name='Plank', dimensionality=THREE_D,
                                          type=DEFORMABLE_BODY)
```

```
plankPart.BaseSolidExtrude(sketch=plankProfileSketch, depth=80.0)

# b) Curved block

# i) Sketch the plank cross section using rectangle tool
curvedBlockProfileSketch = contactModel \
                     .ConstrainedSketch(name='Curved Block Sketch', sheetSize=40)

curvedBlockProfileSketch.Arc3Points(point1=(-10.0, 0.0), point2=(10.0, 0.0),
                                                         point3=(0.0, 10.0))
curvedBlockProfileSketch.Line(point1=(-10.0, 0.0), point2=(-10.0, -15.0))
curvedBlockProfileSketch.Line(point1=(-10.0, -15.0), point2=(10.0, -15.0))
curvedBlockProfileSketch.Line(point1=(10.0, -15.0), point2=(10.0, 0.0))

# ii) Create a 3D deformable part named "Curved Block" by extruding the sketch
curvedBlockPart=contactModel.Part(name='Curved Block', dimensionality=THREE_D,
                                                       type=DEFORMABLE_BODY)
curvedBlockPart.BaseSolidExtrude(sketch=curvedBlockProfileSketch, depth=20.0)

# c) Rectangular block

# i) Sketch the plank cross section using rectangle tool
rectangularBlockProfileSketch = contactModel \
                .ConstrainedSketch(name='Rectangular Block Sketch', sheetSize=40)
rectangularBlockProfileSketch.rectangle(point1=(0.0,0.0), point2=(20.0,10.0))

# ii) Create a 3D deformable part named "Rectangular Block" by extruding the sketch
rectangularBlockPart=contactModel.Part(name='Rectangular Block',
                                    dimensionality=THREE_D, type=DEFORMABLE_BODY)
rectangularBlockPart.BaseSolidExtrude(sketch=rectangularBlockProfileSketch,
                                                                   depth=35.0)

# -----------------------------------------------------------------------
# Create materials

import material

# Create material AISI 1005 Steel by assigning mass density, youngs modulus and
# poissons ratio
steelMaterial = contactModel.Material(name='AISI 1005 Steel')
steelMaterial.Density(table=((7872, ),       ))
steelMaterial.Elastic(table=((200E9, 0.29), ))

# Create material Aluminum 2024-T3 by assigning mass density, youngs modulus and
# poissons ratio
aluminumMaterial = contactModel.Material(name='Aluminum 2024-T3')
aluminumMaterial.Density(table=((2770, ),       ))
aluminumMaterial.Elastic(table=((73.1E9, 0.33), ))

# -----------------------------------------------------------------------
# Create solid sections and assign the parts to them
```

```
import section

# Create a section to assign to the beam
steelSection = contactModel.HomogeneousSolidSection(name='Steel Section',
                                                    material='AISI 1005 Steel')
aluminumSection = contactModel.HomogeneousSolidSection(name='Aluminum Section',
                                                       material='Aluminum 2024-T3')


#assign aluminum section to plank
plank_region = (plankPart.cells,)
plankPart.SectionAssignment(region=plank_region, sectionName='Aluminum Section')

#assign steel section to curved block
curved_block_region = (curvedBlockPart.cells,)
curvedBlockPart.SectionAssignment(region=curved_block_region,
                                  sectionName='Steel Section')

#assign steel section to rectangular block
rectangular_block_region = (rectangularBlockPart.cells,)
rectangularBlockPart.SectionAssignment(region=rectangular_block_region,
                                       sectionName='Steel Section')

# -------------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instances
contactAssembly = contactModel.rootAssembly
plankInstance = contactAssembly.Instance(name='Plank Instance', part=plankPart,
                                                                dependent=ON)
curvedBlockInstance = contactAssembly.Instance(name='Curved Block Instance',
                                               part=curvedBlockPart, dependent=ON)
rectangularBlockInstance = contactAssembly \
                                    .Instance(name='Rectangular Block Instance',
                                              part=rectangularBlockPart,
                                              dependent=ON)



# ++++++++++++++++++++++++++++++++++++++++++++++++
# Identify all the faces used to constrain the assembly

# plank front face
plank_constraint_face_1_point = (10.0,1.0,0.0)
plank_constraint_face_1 = plankInstance.faces \
                                        .findAt(plank_constraint_face_1_point,)

# plank bottom face
plank_constraint_face_2_point = (0.0,0.0,30.0)
plank_constraint_face_2 = plankInstance.faces \
                                        .findAt(plank_constraint_face_2_point,)
```

```
# plank side face
plank_constraint_face_3_point = (-10.0,1.0,30.0)
plank_constraint_face_3 = plankInstance.faces \
                                       .findAt(plank_constraint_face_3_point,)

# curved block front face
curvedBlock_constraint_face_1_point = (-10.0,-7.5,10.0)
curvedBlock_constraint_face_1 = curvedBlockInstance.faces \
                                   .findAt(curvedBlock_constraint_face_1_point,)

# curved block bottom face
curvedBlock_constraint_face_2_point = (0.0,-15.0,10.0)
curvedBlock_constraint_face_2 = curvedBlockInstance.faces \
                                   .findAt(curvedBlock_constraint_face_2_point,)

# curved block side face
curvedBlock_constraint_face_3_point = (0.0,-7.5,0.0)
curvedBlock_constraint_face_3 = curvedBlockInstance.faces \
                                   .findAt(curvedBlock_constraint_face_3_point,)

# rectangular block front face
rectangularBlock_constraint_face_1_point = (10.0,5.0,0.0)
rectangularBlock_constraint_face_1 = rectangularBlockInstance.faces \
                               .findAt(rectangularBlock_constraint_face_1_point,)

# rectangular block bottom face
rectangularBlock_constraint_face_2_point = (10.0,0.0,17.5)
rectangularBlock_constraint_face_2 = rectangularBlockInstance.faces \
                               .findAt(rectangularBlock_constraint_face_2_point,)

# rectangular block side face
rectangularBlock_constraint_face_3_point = (0.0,5.0,17.5)
rectangularBlock_constraint_face_3 = rectangularBlockInstance.faces \
                               .findAt(rectangularBlock_constraint_face_3_point,)


# ++++++++++++++++++++++++++++++++++++++++++++++
# Identify all the faces used for boundary conditions

# plank front face, will be fixed
plank_encastre_face_point = (10.0,1.0,0.0)
plank_encastre_face = plankInstance.faces.findAt((plank_encastre_face_point,))
plank_encastre_region=regionToolset.Region(faces=(plank_encastre_face))


# curved block bottom face, will be fixed
curvedBlock_encastre_face_point = (0.0,-15.0,10.0)
curvedBlock_encastre_face = curvedBlockInstance.faces \
                                     .findAt((curvedBlock_encastre_face_point,))
curvedBlock_encastre_region = regionToolset \
                                     .Region(faces=(curvedBlock_encastre_face))
```

```
# rectangular block front face, will be fixed
rectangularBlock_encastre_face_point = (10.0,5.0,0.0)
rectangularBlock_encastre_face = rectangularBlockInstance.faces \
                                  .findAt((rectangularBlock_encastre_face_point,))
rectangularBlock_encastre_region = regionToolset \
                                    .Region(faces=(rectangularBlock_encastre_face))

# plank top face, will be pushed down
plank_displacement_face_point = (0.0,2.0,30.0)
plank_displacement_face = plankInstance.faces \
                                         .findAt((plank_displacement_face_point,))
plank_displacement_region=regionToolset.Region(faces=(plank_displacement_face))

# rectangular block top face, will be pushed down
rectangularBlock_displacement_face_point = (10.0,10.0,17.5)
rectangularBlock_displacement_face = rectangularBlockInstance.faces \
                              .findAt((rectangularBlock_displacement_face_point,))
rectangularBlock_displacement_region=regionToolset \
                              .Region(faces=(rectangularBlock_displacement_face))

# ++++++++++++++++++++++++++++++++++++++++++++++++
# Identify vertices used for loads

vertices_for_force = plankInstance.vertices.findAt(((-10.0, 0.0, 80.0),),
                                                   ((10.0, 0.0, 80.0),),)

# ++++++++++++++++++++++++++++++++++++++++++++++++

# Identify faces used to define Surfaces in the assembly. These will later be used
# for contact interactions.

# rectangular block bottom surface
rectangularBlock_bottom_surface_point = (10.0,0.0,17.5)
rectangularBlock_bottom_surface = rectangularBlockInstance.faces \
                                 .findAt((rectangularBlock_bottom_surface_point,))

# curved block top surface
curvedBlock_top_surface_point = (0.0,10.0,10.0)
curvedBlock_top_surface = curvedBlockInstance.faces \
                                 .findAt((curvedBlock_top_surface_point,))

# plank bottom surface
plank_bottom_surface_point = (0.0,0.0,30.0)
plank_bottom_surface = plankInstance.faces.findAt((plank_bottom_surface_point,))

# plank top surface
plank_top_surface_point = (0.0,2.0,30.0)
plank_top_surface = plankInstance.faces.findAt((plank_top_surface_point,))

# ++++++++++++++++++++++++++++++++++++++++++++++++
# assemble the parts using face to face relationships
```

✓

✓

```
# establish face to face relationships between plank and curved block
contactAssembly.FaceToFace(movablePlane=curvedBlock_constraint_face_1,
                           fixedPlane=plank_constraint_face_1,
                           flip=OFF, clearance=-30.0)
contactAssembly.FaceToFace(movablePlane=curvedBlock_constraint_face_2,
                           fixedPlane=plank_constraint_face_2,
                           flip=OFF, clearance=25.0)
contactAssembly.FaceToFace(movablePlane=curvedBlock_constraint_face_3,
                           fixedPlane=plank_constraint_face_3,
                           flip=ON, clearance=-20.0)

# establish face to face relationships between plank and rectangular block
contactAssembly.FaceToFace(movablePlane=rectangularBlock_constraint_face_1,
                           fixedPlane=plank_constraint_face_1,
                           flip=OFF, clearance=0.0)
contactAssembly.FaceToFace(movablePlane=rectangularBlock_constraint_face_2,
                           fixedPlane=plank_constraint_face_2,
                           flip=OFF, clearance=-2.0)
contactAssembly.FaceToFace(movablePlane=rectangularBlock_constraint_face_3,
                           fixedPlane=plank_constraint_face_3,
                           flip=OFF, clearance=0.0)

# -----------------------------------------------------------------------
# Create the steps

import step

# Create a step in which the parts will be pushed together to avoid chatter
contactModel.StaticStep(name='Make Contact', previous='Initial',
                        description='Push parts together to avoid chatter',
                        nlgeom=ON, initialInc=0.1)

# Create a step in which the forces will be applied
contactModel.StaticStep(name='Apply Force', previous='Make Contact',
                        description='Apply force on one end of the plank',
                        initialInc=0.1)


# -----------------------------------------------------------------------
# Create the field output request
# Leave at defaults


# -----------------------------------------------------------------------
# Create the history output request
# Leave at defaults


# -----------------------------------------------------------------------
# Apply boundary conditions

contactModel.EncastreBC(name='Fix Plank End', createStepName='Apply Force',
                                                region=plank_encastre_region)

contactModel.EncastreBC(name='Fix Curved Block', createStepName='Initial',
                                                region=curvedBlock_encastre_region)

contactModel.EncastreBC(name='Fix Rectangular Block',
                        createStepName='Apply Force',
                        region=rectangularBlock_encastre_region)

contactModel.DisplacementBC(name='Press Plank Curved',
                            createStepName='Make Contact',
                            region=plank_displacement_region,
                            u1=0.0, u2=-0.2, u3=0.0, ur1=0.0, ur2=0.0, ur3=0.0)

contactModel.DisplacementBC(name='Press Rectangular Plank',
                            createStepName='Make Contact',
                            region=rectangularBlock_displacement_region,
                            u1=0.0, u2=-0.21, u3=0.0, ur1=0.0, ur2=0.0, ur3=0.0)

# Boundary conditions will be propagated from one step to the other by default
# We want this to happen for 'Fix Curved Block'
# However we want to remove the BC for 'Press Plank Curved' and 'Press Rectangular
# Plank'
contactModel.boundaryConditions['Press Plank Curved'].deactivate('Apply Force')
contactModel.boundaryConditions['Press Rectangular Plank'] \
                                                  .deactivate('Apply Force')

# -------------------------------------------------------------------------
# Apply concentrated forces

contactModel.ConcentratedForce(name='Concentrated forces at corners',
                               createStepName='Apply Force',
                               region=(vertices_for_force,),
                               cf2=-4E+6, distributionType=UNIFORM)



# -------------------------------------------------------------------------
# Define surfaces to use in contact interactions

contactAssembly.Surface(side1Faces=rectangularBlock_bottom_surface,
                        name='Rect Block Bottom')
contactAssembly.Surface(side1Faces=curvedBlock_top_surface,
                        name='Curved Block Top')
contactAssembly.Surface(side1Faces=plank_bottom_surface, name='Plank Bottom')
contactAssembly.Surface(side1Faces=plank_top_surface, name='Plank Top')

# -------------------------------------------------------------------------
# Create interaction properties

# Create a frictionless property
frictionless_interaction = contactModel.ContactProperty('Frictionless')
frictionless_interaction.TangentialBehavior(formulation=FRICTIONLESS)
```

```
# Create a frictional property
friction_interaction = contactModel.ContactProperty('Frictional')
friction_interaction.TangentialBehavior(formulation=PENALTY,  table=((0.1,),),
                                                               fraction=0.005)

# -------------------------------------------------------------------------
# Create interactions

frictionless_master_surface_region = contactAssembly.surfaces['Curved Block Top']
frictionless_slave_surface_region = contactAssembly.surfaces['Plank Bottom']

contactModel.SurfaceToSurfaceContactStd(name='Curved Plank Interaction',
                                        createStepName='Make Contact',
                                        master=frictionless_master_surface_region,
                                        slave=frictionless_slave_surface_region,
                                        sliding=FINITE,
                                        interactionProperty='Frictionless')

frictional_master_surface_region = contactAssembly.surfaces['Rect Block Bottom']
frictional_slave_surface_region = contactAssembly.surfaces['Plank Top']

contactModel.SurfaceToSurfaceContactStd(name='Rect Plank Interaction',
                                        createStepName='Make Contact',
                                        master=frictional_master_surface_region,
                                        slave=frictional_slave_surface_region,
                                        sliding=FINITE,
                                        interactionProperty='Frictional')

# -------------------------------------------------------------------------
# Create the mesh

import mesh

# ++++++++++++++++++++++++++++++++++++++++++++++++
# Mesh the plank
# We place a point somewhere inside it based on our knowledge of the geometry
plank_inside_coord=(0.0,1.0,40.0)

elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN, secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)

plankCells=plankPart.cells
selectedPlankCells=plankCells.findAt(plank_inside_coord,)
plankMeshRegion=(selectedPlankCells,)
plankPart.setElementType(regions=plankMeshRegion, elemTypes=(elemType1,))

plankPart.seedPart(size=4, deviationFactor=0.1)

plankPart.generateMesh()
```

```
# +++++++++++++++++++++++++++++++++++++++++++++++++
# Mesh the curved block
curvedBlock_inside_coord=(0.0,-5.0,10.0)

elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN, secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)

curvedBlockCells=curvedBlockPart.cells
selectedCurvedBlockCells=curvedBlockCells.findAt(curvedBlock_inside_coord,)
curvedBlockMeshRegion=(selectedCurvedBlockCells,)
curvedBlockPart.setElementType(regions=curvedBlockMeshRegion,
                               elemTypes=(elemType1,))

curvedBlockPart.seedPart(size=4, deviationFactor=0.1)

curvedBlockPart.generateMesh()

# +++++++++++++++++++++++++++++++++++++++++++++++++
# Mesh the rectangular block
rectangularBlock_inside_coord=(10.0,5.0,17.5)

elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN, secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)

rectangularBlockCells=rectangularBlockPart.cells
selectedRectangularBlockCells=rectangularBlockCells \
                                          .findAt(rectangularBlock_inside_coord,)
rectangularBlockMeshRegion=(selectedRectangularBlockCells,)
rectangularBlockPart.setElementType(regions=rectangularBlockMeshRegion,
                                    elemTypes=(elemType1,))

rectangularBlockPart.seedPart(size=4, deviationFactor=0.1)

rectangularBlockPart.generateMesh()

# -----------------------------------------------------------------------
# Create and run the job

import job

# Create the job
mdb.Job(name='ContactSimulationJob', model='Contact Simulation',
        type=ANALYSIS, explicitPrecision=SINGLE,
        nodalOutputPrecision=SINGLE, description='Run the contact simulation',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)

# Run the job
```

```
mdb.jobs['ContactSimulationJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['ContactSimulationJob'].waitForCompletion()

# End of run job
# -----------------------------------------------------------------------

import visualization

contact_viewport = session.Viewport(name='Contact Simulation Results Viewport')
contact_Odb_Path = 'ContactSimulationJob.odb'
an_odb_object = session.openOdb(name=contact_Odb_Path)
contact_viewport.setValues(displayedObject=an_odb_object)
contact_viewport.odbDisplay.display.setValues(plotState=(DEFORMED, ))
```

12.4 Examining the Script

Let's explore the workings of this script.

12.4.1 Initialization (import required modules)

The block dealing with this initialization is

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)
```

These statements are identical to those used in the Cantilever Beam example and were explained in section 4.3.1 on page 65

12.4.2 Create the model

The following block creates the model

```
# -----------------------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Contact Simulation')
contactModel = mdb.models['Contact Simulation']
```

These statements rename the model from 'Model-1' to 'Contact Simulation'. They are almost identical to those used in the Cantilever Beam example and were explained in section 4.3.2 on page 67.

12.4.3 Create the part

The following block creates the part

```
# -------------------------------------------------------------------------
# Create the parts

import sketch
import part

# a) Plank

# i) Sketch the plank cross section using rectangle tool
plankProfileSketch = contactModel.ConstrainedSketch(name='Plank Sketch',
                                                     sheetSize=40)
plankProfileSketch.rectangle(point1=(-10.0,0.0), point2=(10.0,2.0))

# ii) Create a 3D deformable part named "Plank" by extruding the sketch
plankPart=contactModel.Part(name='Plank', dimensionality=THREE_D,
                                          type=DEFORMABLE_BODY)
plankPart.BaseSolidExtrude(sketch=plankProfileSketch, depth=80.0)

# b) Curved block

# i) Sketch the plank cross section using rectangle tool
curvedBlockProfileSketch = contactModel \
                   .ConstrainedSketch(name='Curved Block Sketch', sheetSize=40)

curvedBlockProfileSketch.Arc3Points(point1=(-10.0, 0.0), point2=(10.0, 0.0),
                                                          point3=(0.0, 10.0))
curvedBlockProfileSketch.Line(point1=(-10.0, 0.0), point2=(-10.0, -15.0))
curvedBlockProfileSketch.Line(point1=(-10.0, -15.0), point2=(10.0, -15.0))
curvedBlockProfileSketch.Line(point1=(10.0, -15.0), point2=(10.0, 0.0))

# ii) Create a 3D deformable part named "Curved Block" by extruding the sketch
curvedBlockPart=contactModel.Part(name='Curved Block', dimensionality=THREE_D,
                                                       type=DEFORMABLE_BODY)
curvedBlockPart.BaseSolidExtrude(sketch=curvedBlockProfileSketch, depth=20.0)

# c) Rectangular block

# i) Sketch the plank cross section using rectangle tool
rectangularBlockProfileSketch = contactModel \
                .ConstrainedSketch(name='Rectangular Block Sketch', sheetSize=40)
rectangularBlockProfileSketch.rectangle(point1=(0.0,0.0), point2=(20.0,10.0))

# ii) Create a 3D deformable part named "Rectangular Block" by extruding the sketch
rectangularBlockPart=contactModel.Part(name='Rectangular Block',
                                    dimensionality=THREE_D, type=DEFORMABLE_BODY)
rectangularBlockPart.BaseSolidExtrude(sketch=rectangularBlockProfileSketch,
                                                                  depth=35.0)
```

This block of code creates the 3 parts – a rectangular plank, a block with a curved top, and a block. Aside from the **Arc3Points()** method, everything else here should look familiar to you. The **ConstrainedSketch()**, **rectangle()**, **Part()** and **BaseSolidExtrude()** methods were explained in section 4.3.3 on page 68 of the Cantilever Beam example.

```
curvedBlockProfileSketch.Arc3Points(point1=(-10.0, 0.0), point2=(10.0, 0.0),
                                                          point3=(0.0, 10.0))
```

The **Arc3Points()** method of the **ConstrainedSketchGeometry** object is used to draw an arc using two endpoints (**point1** and **point2**) and a third point that lies somewhere on the arc (**point3**). It returns a **ConstrainedSketchGeometry** object, or **None** if the arc cannot be created. The **ConstrainedSketchGeometry** object is an object that stores the geometry of a sketch such as arcs and lines.

```
curvedBlockProfileSketch.Line(point1=(-10.0, 0.0), point2=(-10.0, -15.0))
curvedBlockProfileSketch.Line(point1=(-10.0, -15.0), point2=(10.0, -15.0))
curvedBlockProfileSketch.Line(point1=(10.0, -15.0), point2=(10.0, 0.0))
```

The **Line()** method of the **ConstrainedSketchGeometry** has been used earlier in the truss example, section 7.4.3 on page 130. To refresh your memory, it draws a line using two points supplied as arguments. It returns a **ConstrainedSketchGeometry** object, or **None** if the line cannot be created.

All the statements except the last one are very similar to the ones used in the Cantilever Beam example. To refresh your memory on the **ConstrainedSketch()**, **rectangle()** and **Part()** methods, refer back to section 4.3.3.

12.4.4 Define the materials

The following block creates the material for the simulation

```
# -------------------------------------------------------------------
# Create materials

import material

# Create material AISI 1005 Steel by assigning mass density, youngs modulus and
# poissons ratio
steelMaterial = contactModel.Material(name='AISI 1005 Steel')
steelMaterial.Density(table=((7872, ),       ))
steelMaterial.Elastic(table=((200E9, 0.29), ))

# Create material Aluminum 2024-T3 by assigning mass density, youngs modulus and
# poissons ratio
```

```
aluminumMaterial = contactModel.Material(name='Aluminum 2024-T3')
aluminumMaterial.Density(table=((2770, ),       ))
aluminumMaterial.Elastic(table=((73.1E9, 0.33), ))
```

The statements are similar to those used in the Cantilever Beam example and were explained in section 4.3.4 on page 71.

12.4.5 Create solid sections and make section assignments

The following block creates the sections and makes assignments

```
# -----------------------------------------------------------------------
# Create solid sections and assign the parts to them

import section

# Create a section to assign to the beam
steelSection = contactModel.HomogeneousSolidSection(name='Steel Section',
                                                    material='AISI 1005 Steel')
aluminumSection = contactModel.HomogeneousSolidSection(name='Aluminum Section',
                                                      material='Aluminum 2024-T3')

# Assign aluminum section to plank
plank_region = (plankPart.cells,)
plankPart.SectionAssignment(region=plank_region, sectionName='Aluminum Section')

# Assign steel section to curved block
curved_block_region = (curvedBlockPart.cells,)
curvedBlockPart.SectionAssignment(region=curved_block_region,
                                  sectionName='Steel Section')

# Assign steel section to rectangular block
rectangular_block_region = (rectangularBlockPart.cells,)
rectangularBlockPart.SectionAssignment(region=rectangular_block_region,
                                       sectionName='Steel Section')
```

The procedure followed here is similar to that in section 4.3.5 of the Cantilever Beam example on page 72. The **HomogeneousSolidSection()** and **SectionAssignment()** methods are used as before. Two sections 'steelSection' and 'aluminumSection' are created, which are then assigned to the appropriate parts.

12.4.6 Create an assembly

The following block creates the assembly.

```
# -----------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instances
contactAssembly = contactModel.rootAssembly
plankInstance = contactAssembly.Instance(name='Plank Instance', part=plankPart,
                                          dependent=ON)
curvedBlockInstance = contactAssembly.Instance(name='Curved Block Instance',
                                          part=curvedBlockPart, dependent=ON)
rectangularBlockInstance = contactAssembly \
                                   .Instance(name='Rectangular Block Instance',
                                             part=rectangularBlockPart,
                                             dependent=ON)


# ++++++++++++++++++++++++++++++++++++++++++++++++++
# Identify all the faces used to constrain the assembly

# plank front face
plank_constraint_face_1_point = (10.0,1.0,0.0)
plank_constraint_face_1 = plankInstance.faces \
                                    .findAt(plank_constraint_face_1_point,)

# plank bottom face
plank_constraint_face_2_point = (0.0,0.0,30.0)
plank_constraint_face_2 = plankInstance.faces \
                                    .findAt(plank_constraint_face_2_point,)

# plank side face
plank_constraint_face_3_point = (-10.0,1.0,30.0)
plank_constraint_face_3 = plankInstance.faces \
                                    .findAt(plank_constraint_face_3_point,)

# curved block front face
curvedBlock_constraint_face_1_point = (-10.0,-7.5,10.0)
curvedBlock_constraint_face_1 = curvedBlockInstance.faces \
                              .findAt(curvedBlock_constraint_face_1_point,)

# curved block bottom face
curvedBlock_constraint_face_2_point = (0.0,-15.0,10.0)
curvedBlock_constraint_face_2 = curvedBlockInstance.faces \
                              .findAt(curvedBlock_constraint_face_2_point,)

# curved block side face
curvedBlock_constraint_face_3_point = (0.0,-7.5,0.0)
curvedBlock_constraint_face_3 = curvedBlockInstance.faces \
                                    .findAt(curvedBlock_constraint_face_3_point,)

# rectangular block front face
rectangularBlock_constraint_face_1_point = (10.0,5.0,0.0)
rectangularBlock_constraint_face_1 = rectangularBlockInstance.faces \
                                .findAt(rectangularBlock_constraint_face_1_point,)

# rectangular block bottom face
rectangularBlock_constraint_face_2_point = (10.0,0.0,17.5)
rectangularBlock_constraint_face_2 = rectangularBlockInstance.faces \
                                .findAt(rectangularBlock_constraint_face_2_point,)

# rectangular block side face
rectangularBlock_constraint_face_3_point = (0.0,5.0,17.5)
rectangularBlock_constraint_face_3 = rectangularBlockInstance.faces \
                                .findAt(rectangularBlock_constraint_face_3_point,)


# ++++++++++++++++++++++++++++++++++++++++++++++++
# Identify all the faces used for boundary conditions

# plank front face, will be fixed
plank_encastre_face_point = (10.0,1.0,0.0)
plank_encastre_face = plankInstance.faces.findAt((plank_encastre_face_point,))
plank_encastre_region=regionToolset.Region(faces=(plank_encastre_face))


# curved block bottom face, will be fixed
curvedBlock_encastre_face_point = (0.0,-15.0,10.0)
curvedBlock_encastre_face = curvedBlockInstance.faces \
                                       .findAt((curvedBlock_encastre_face_point,))
curvedBlock_encastre_region = regionToolset \
                                        .Region(faces=(curvedBlock_encastre_face))

# rectangular block front face, will be fixed
rectangularBlock_encastre_face_point = (10.0,5.0,0.0)
rectangularBlock_encastre_face = rectangularBlockInstance.faces \
                                  .findAt((rectangularBlock_encastre_face_point,))
rectangularBlock_encastre_region = regionToolset \
                                   .Region(faces=(rectangularBlock_encastre_face))

# plank top face, will be pushed down
plank_displacement_face_point = (0.0,2.0,30.0)
plank_displacement_face = plankInstance.faces \
                                        .findAt((plank_displacement_face_point,))
plank_displacement_region=regionToolset.Region(faces=(plank_displacement_face))

# rectangular block top face, will be pushed down
rectangularBlock_displacement_face_point = (10.0,10.0,17.5)
rectangularBlock_displacement_face = rectangularBlockInstance.faces \
                              .findAt((rectangularBlock_displacement_face_point,))
rectangularBlock_displacement_region=regionToolset \
                                    .Region(faces=(rectangularBlock_displacement_face))

# +++++++++++++++++++++++++++++++++++++++++++++++++
# Identify vertices used for loads

vertices_for_force = plankInstance.vertices.findAt(((-10.0, 0.0, 80.0),),
                                                    ((10.0, 0.0, 80.0),),)

# +++++++++++++++++++++++++++++++++++++++++++++++++

# Identify faces used to define Surfaces in the assembly. These will later be used
# for contact interactions.

# rectangular block bottom surface
rectangularBlock_bottom_surface_point = (10.0,0.0,17.5)
rectangularBlock_bottom_surface = rectangularBlockInstance.faces \
                                  .findAt((rectangularBlock_bottom_surface_point,))

# curved block top surface
curvedBlock_top_surface_point = (0.0,10.0,10.0)
curvedBlock_top_surface = curvedBlockInstance.faces \
                                  .findAt((curvedBlock_top_surface_point,))

# plank bottom surface
plank_bottom_surface_point = (0.0,0.0,30.0)
plank_bottom_surface = plankInstance.faces.findAt((plank_bottom_surface_point,))

# plank top surface
plank_top_surface_point = (0.0,2.0,30.0)
plank_top_surface = plankInstance.faces.findAt((plank_top_surface_point,))

# +++++++++++++++++++++++++++++++++++++++++++++++++
# assemble the parts using face to face relationships

# establish face to face relationships between plank and curved block
contactAssembly.FaceToFace(movablePlane=curvedBlock_constraint_face_1,
                           fixedPlane=plank_constraint_face_1,
                           flip=OFF, clearance=-30.0)
contactAssembly.FaceToFace(movablePlane=curvedBlock_constraint_face_2,
                           fixedPlane=plank_constraint_face_2,
                           flip=OFF, clearance=25.0)
contactAssembly.FaceToFace(movablePlane=curvedBlock_constraint_face_3,
                           fixedPlane=plank_constraint_face_3,
                           flip=ON, clearance=-20.0)

# establish face to face relationships between plank and rectangular block
contactAssembly.FaceToFace(movablePlane=rectangularBlock_constraint_face_1,
                           fixedPlane=plank_constraint_face_1,
                           flip=OFF, clearance=0.0)
contactAssembly.FaceToFace(movablePlane=rectangularBlock_constraint_face_2,
                           fixedPlane=plank_constraint_face_2,
                           flip=OFF, clearance=-2.0)
```

```
contactAssembly.FaceToFace(movablePlane=rectangularBlock_constraint_face_3,
                           fixedPlane=plank_constraint_face_3,
                           flip=OFF, clearance=0.0)
```

```
# Create the part instances
contactAssembly = contactModel.rootAssembly
plankInstance = contactAssembly.Instance(name='Plank Instance', part=plankPart,
                                                               dependent=ON)
curvedBlockInstance = contactAssembly.Instance(name='Curved Block Instance',
                                               part=curvedBlockPart, dependent=ON)
rectangularBlockInstance = contactAssembly \
                                    .Instance(name='Rectangular Block Instance',
                                              part=rectangularBlockPart,
                                              dependent=ON)
```

The procedure followed in these statements is identical to that of the Cantilever Beam example, section 4.3.6 on page 74. The **Instance()** method is used 3 times to create the 3 part instances.

```
# +++++++++++++++++++++++++++++++++++++++++++++++
# Identify all the faces used to constrain the assembly

# plank front face
plank_constraint_face_1_point = (10.0,1.0,0.0)
plank_constraint_face_1 = plankInstance.faces \
                                         .findAt(plank_constraint_face_1_point,)...
...
...
...
... (and so on)
```

The **faces.findAt()** method is used to select the front face of the instance of the plank and assign it to the variable 'plank_constraint_face_1'. The coordinates (10.0, 1.0, 0.0) are based on the fact that all the parts are instanced at the same coordinates in the assembly as they were created in the part module (even if this means they overlap in the assembly) and also based on your knowledge of the part geometry/dimensions.

The reason we are identifying the faces and storing them in variables is that the part instances will later be assembled together using face-to-face position constraints, and we will need to pass the constraint faces to the **FaceToFace()** method. Therefore the front, bottom and side faces of all 3 part instances are found and assigned to variables.

```
# +++++++++++++++++++++++++++++++++++++++++++++++
# Identify all the faces used for boundary conditions

# plank front face, will be fixed
```

```
plank_encastre_face_point = (10.0,1.0,0.0)
plank_encastre_face = plankInstance.faces.findAt((plank_encastre_face_point,))
plank_encastre_region=regionToolset.Region(faces=(plank_encastre_face))
...
...
...
... (and so on)
```

The front face of the plank is found using the **faces.findAt()** method. It is then assigned to a region using the **regionToolset.Region()** method when can later be used when assigning boundary conditions with the **EncastreBC()** method. In a similar manner, 4 other faces are also found, which will be used with either the **EncastreBC()** or **DisplacementBC()** methods when assigning boundary conditions.

```
# +++++++++++++++++++++++++++++++++++++++++++++++++
# Identify vertices used for loads

vertices_for_force = plankInstance.vertices.findAt(((-10.0, 0.0, 80.0),),
                                                   ((10.0, 0.0, 80.0),),)
```

The **vertices.findAt()** method is used to identify the two vertices on the instance of the plank where the loads will later be applied.

```
# +++++++++++++++++++++++++++++++++++++++++++++++++

# Identify faces used to define Surfaces in the assembly. These will later be used
# for contact interactions.

# rectangular block bottom surface
rectangularBlock_bottom_surface_point = (10.0,0.0,17.5)
rectangularBlock_bottom_surface = rectangularBlockInstance.faces \
                                .findAt((rectangularBlock_bottom_surface_point,))
...
...
...
... (and so on)
```

The **faces.findAt()** method is once again used to pick a face of the block part instance and assign it to a variable **rectangularBlock_bottom_surface**. This face however will not be used for boundary conditions but rather as a surface when defining the surface contact interactions. Similarly a few other surfaces are identified that will later be used to define the contact interactions.

```
# +++++++++++++++++++++++++++++++++++++++++++++++++
# Assemble the parts using face to face relationships
# establish face to face relationships between plank and curved block
contactAssembly.FaceToFace(movablePlane=curvedBlock_constraint_face_1,
                           fixedPlane=plank_constraint_face_1,
                           flip=OFF, clearance=-30.0)
contactAssembly.FaceToFace(movablePlane=curvedBlock_constraint_face_2,
                           fixedPlane=plank_constraint_face_2,
                           flip=OFF, clearance=25.0)
contactAssembly.FaceToFace(movablePlane=curvedBlock_constraint_face_3,
                           fixedPlane=plank_constraint_face_3,
                           flip=ON, clearance=-20.0)

# establish face to face relationships between plank and rectangular block
contactAssembly.FaceToFace(movablePlane=rectangularBlock_constraint_face_1,
                           fixedPlane=plank_constraint_face_1,
                           flip=OFF, clearance=0.0)
contactAssembly.FaceToFace(movablePlane=rectangularBlock_constraint_face_2,
                           fixedPlane=plank_constraint_face_2,
                           flip=OFF, clearance=-2.0)
contactAssembly.FaceToFace(movablePlane=rectangularBlock_constraint_face_3,
                           fixedPlane=plank_constraint_face_3,
                           flip=OFF, clearance=0.0)
```

The **FaceToFace()** method is used 6 times to create 6 face-to-face assembly constraints. The **FaceToFace()** method moves an instance (movable instance) so that its face is coincident with another instance (fixed instance). The face on the movable instance is the **movablePlane** and the face on the fixed instance is the **fixedPlane**. These must be passed as arguments to the **FaceToFace()** method. The other two required arguments are **flip**, a Boolean which specifies if the normal to the faces are in the same direction (OFF) or opposite direction (ON), and **clearance**, a Float which specifies the distance between the two faces once they have been aligned together.

12.4.7 Create steps

The following block creates the steps.

```
# -----------------------------------------------------------------
# Create the steps

import step

# Create a step in which the parts will be pushed together to avoid chatter
contactModel.StaticStep(name='Make Contact', previous='Initial',
                        description='Push parts together to avoid chatter',
                        nlgeom=ON, initialInc=0.1)

# Create a step in which the forces will be applied
```

```
contactModel.StaticStep(name='Apply Force', previous='Make Contact',
                        description='Apply force on one end of the plank',
                        initialInc=0.1)
```

The statements are similar to the ones used in the Cantilever Beam example in section 4.3.7 on page 75 with a few minor differences. Non-linear geometry has been turned on using **nlgeom**, the initial increment has been set to 0.1 using **initialInc**, and there are two steps as opposed to one.

The first step, 'Make Contact', will be used to push the parts together to establish contact between them. This helps since contact is a severe discontinuity and this makes helps Abaqus/Standard make the transition as smoothly as possible. The second step 'Apply Force' is the one in which the forces will be applied.

12.4.8 Create and define field output requests

The following block creates the field output requests.

```
# -------------------------------------------------------------------------
# Create the field output request
# Leave at defaults
```

No field output requests are created (or deleted), therefore field output variables are left at the defaults.

12.4.9 Create and define history output requests

The following block defines the history output requests:

```
# -------------------------------------------------------------------------
# Create the history output request
# Leave at defaults
```

No history output requests are created (or deleted), therefore history output variables are left at the defaults.

12.4.10 Apply boundary conditions

The following block applies the boundary conditions:

```
# -------------------------------------------------------------------------
# Apply boundary conditions
```

```
contactModel.EncastreBC(name='Fix Plank End', createStepName='Apply Force',
                                               region=plank_encastre_region)

contactModel.EncastreBC(name='Fix Curved Block', createStepName='Initial',
                                                 region=curvedBlock_encastre_region)

contactModel.EncastreBC(name='Fix Rectangular Block',
                        createStepName='Apply Force',
                        region=rectangularBlock_encastre_region)

contactModel.DisplacementBC(name='Press Plank Curved',
                            createStepName='Make Contact',
                            region=plank_displacement_region,
                            u1=0.0, u2=-0.2, u3=0.0, ur1=0.0, ur2=0.0, ur3=0.0)

contactModel.DisplacementBC(name='Press Rectangular Plank',
                            createStepName='Make Contact',
                            region=rectangularBlock_displacement_region,
                            u1=0.0, u2=-0.21, u3=0.0, ur1=0.0, ur2=0.0, ur3=0.0)

# Boundary conditions will be propagated from one step to the other by default
# We want this to happen for 'Fix Curved Block'
# However we want to remove the BC for 'Press Plank Curved' and 'Press Rectangular
# Plank'
contactModel.boundaryConditions['Press Plank Curved'].deactivate('Apply Force')
contactModel.boundaryConditions['Press Rectangular Plank'] \
                                                     .deactivate('Apply Force')
```

You've seen the **EncastreBC()** method used in the Cantilever Beam example, section 4.3.11 on page 81. You've also seen the **DisplacementBC()** method used in the Beam Frame Analysis example, section 9.4.15 on page 220. These **EncastreBC()** method is used here to fix the end of the plank, the block and the bottom of the curved block. The **DisplacementBC()** condition is used to move, and therefore press, the plank against the curved block and the block against the plank.

```
contactModel.boundaryConditions['Press Plank Curved'].deactivate('Apply Force')
contactModel.boundaryConditions['Press Rectangular Plank'] \
                                                     .deactivate('Apply Force')
```

The **deactivate()** method of the **BoundaryCondition** object deactivates a boundary condition in a step, and the boundary condition remains deactivated in subsequent steps. The one required argument is **stepName** which is a String indicating the name of the step in which to deactivate the boundary condition.

12.4.11 Apply loads

The following block applies the concentrated forces

```
# -----------------------------------------------------------------------
# Apply concentrated forces

contactModel.ConcentratedForce(name='Concentrated forces at corners',
                               createStepName='Apply Force',
                               region=(vertices_for_force,),
                               cf2=-4E+6, distributionType=UNIFORM)
```

The **ConcentratedForce()** method was explained in the plate bending example, section 10.4.12 on page 248. Here it is used to apply a force in the negative Y direction (downward) of 400,000 N on the region **vertices_for_force** which was created earlier in the assembly module for this purpose using the **vertices.findAt()** method.

12.4.12 Create Surfaces

The following block identifies and stores the surfaces which will later be used to define interactions

```
# -----------------------------------------------------------------------
# Define surfaces to use in contact interactions

contactAssembly.Surface(side1Faces=rectangularBlock_bottom_surface,
                        name='Rect Block Bottom')
contactAssembly.Surface(side1Faces=curvedBlock_top_surface,
                        name='Curved Block Top')
contactAssembly.Surface(side1Faces=plank_bottom_surface, name='Plank Bottom')
contactAssembly.Surface(side1Faces=plank_top_surface, name='Plank Top')
```

The **Surface()** method creates a surface object, which stores surfaces identified in an assembly. The surface refers to the side that was specified in the possible required arguments, which for a 3D solid face are **side1Faces** or **side2Faces**. By using **side1Faces** we are telling Abaqus that the normal of our newly created Surface object is in the same direction as the normal of the face passed as the argument. Basically we are selecting the outer surface. The other required argument is **name**, which as you might have guessed is a String specifying the repository key.

Four surfaces are created, and these will be used to define the contact interactions.

12.4.13 Create Interaction Properties

The following block defines the interaction properties which will later be assigned to interactions

```
# -----------------------------------------------------------------
# Create interaction properties

# Create a frictionless property
frictionless_interaction = contactModel.ContactProperty('Frictionless')
frictionless_interaction.TangentialBehavior(formulation=FRICTIONLESS)

# Create a frictional property
friction_interaction = contactModel.ContactProperty('Frictional')
friction_interaction.TangentialBehavior(formulation=PENALTY,  table=((0.1,),),
                                                              fraction=0.005)
```

```
frictionless_interaction = contactModel.ContactProperty('Frictionless')
```

The **ContactProperty()** method is used to create a **ContactProperty** object. A **ContactProperty** object, which is derived from the **InteractionProperty** object, defines the contact interaction property. The only required argument for the **ContactProperty()** method is **name** which is a String specifying the repository key for the interaction property. We name it 'Frictionless'. The **ContactProperty()** method returns a **ContactProperty** object, which we store in a variable **frictionless_interaction**.

```
friction_interaction = contactModel.ContactProperty('Frictional')
```

This is similar to the previous statement.

```
frictionless_interaction.TangentialBehavior(formulation=FRICTIONLESS)
```

Here the **TangentialBehavior()** method is used to create a **ContactTangentialBehavior** object. The **TangentialBehavior()** method has no required arguments, the optional one used here is **formulation**. The default is **FRICTIONLESS**, other possible values are **PENALTY**, **EXPONENTIAL_DECAY**, **ROUGH**, **LAGRANGE**, and **USER_DEFINED**. We use a **FRICTIONLESS** formulation

```
friction_interaction.TangentialBehavior(formulation=PENALTY,  table=((0.1,),),
                                                              fraction=0.005)
```

Here the **TangentialBehavior()** method is used to create a **ContactTangentialBehavior** object. The **TangentialBehavior()** method has no required arguments, the optional ones used here are **formulation**, **table** and **fraction**. **formulation** is a SymbolicConstant which specifies the friction formulation. The default is **FRICTIONLESS**, other possible values are **PENALTY**, **EXPONENTIAL_DECAY**, **ROUGH**, **LAGRANGE**, and **USER_DEFINED**. We use a **PENALTY** formulation. **table** is a sequence of sequences of Floats which specify the tangential behavior. The table data specifies a number of values which are defined in the Abaqus Scripting Reference Manual, and these depend on the type of formulation. For a **PENALTY** or **LAGRANGE** formulation, the first value is friction coefficient in the first slip direction, which we specify to be 0.1. **fraction** is a Float specifying the fraction of a characteristic surface dimension for maximum elastic slip. We specify 0.005.

Just so you know, the last two statements can be combined and rewritten as:

```
contactModel.interactionProperties['Frictional'] /
        .TangentialBehavior(formulation=PENALTY, table=((0.1,),), fraction=0.005)
```

12.4.14 Create Interactions

The following block creates the interactions

```
# -----------------------------------------------------------------------
# Create interactions

frictionless_master_surface_region = contactAssembly.surfaces['Curved Block Top']
frictionless_slave_surface_region = contactAssembly.surfaces['Plank Bottom']

contactModel.SurfaceToSurfaceContactStd(name='Curved Plank Interaction',
                                        createStepName='Make Contact',
                                        master=frictionless_master_surface_region,
                                        slave=frictionless_slave_surface_region,
                                        sliding=FINITE,
                                        interactionProperty='Frictionless')

frictional_master_surface_region = contactAssembly.surfaces['Rect Block Bottom']
frictional_slave_surface_region = contactAssembly.surfaces['Plank Top']

contactModel.SurfaceToSurfaceContactStd(name='Rect Plank Interaction',
                                        createStepName='Make Contact',
                                        master=frictional_master_surface_region,
                                        slave=frictional_slave_surface_region,
                                        sliding=FINITE,
                                        interactionProperty='Frictional')
```

```
frictionless_master_surface_region = contactAssembly.surfaces['Curved Block Top']
frictionless_slave_surface_region = contactAssembly.surfaces['Plank Bottom']
```

The surfaces 'Curved Block Top' and 'Plank Bottom' are assigned to the variables **frictionless_master_surface_region** and **frictionless_slave_surface_region**. The word 'region' is used in the variable names to point out that these variables will subsequently be passed to the **SurfaceToSurfaceContactStd()** method which requires **Region** objects as arguments. This is possible because a **Region** object is a link between a **Set** or a **Surface** object and its attributes. Here Abaqus will implicitly create **Region** objects associated with these surfaces.

```
contactModel.SurfaceToSurfaceContactStd(name='Curved Plank Interaction',
                                        createStepName='Make Contact',
                                        master=frictionless_master_surface_region,
                                        slave=frictionless_slave_surface_region,
                                        sliding=FINITE,
                                        interactionProperty='Frictionless')
```

This statement creates the interaction between the curved block and the plank. The **SurfaceToSurfaceContactStd()** method creates a **SurfaceToSurfaceContactStd** object which is derived from the **Interaction** object and defines surface-to-surface contact in an Abaqus/Standard analysis. All the arguments supplied here are required arguments. **name** is a String specifying the repository key of the **SurfaceToSurfaceContactStd** object. **createStepName** is a String specifying the name of the step in which the object is to be created. **master** and **slave** are **Region** objects specifying the master and slave surfaces for the contact interaction. It was for this purpose that the variables **frictionless_master_surface_region** and **frictionless_slave_surface_region** were created in the previous statements. **sliding** is a SymbolicConstant - either **FINITE** or **SMALL** – which specifies the contact formulation. **interactionProperty** is the String specifying the repository key of the **ContactProperty** object to associate with this interaction. We had named our interaction property 'Frictionless' and are now assigning that interaction property to this interaction.

```
frictional_master_surface_region = contactAssembly.surfaces['Rect Block Bottom']
frictional_slave_surface_region = contactAssembly.surfaces['Plank Top']

contactModel.SurfaceToSurfaceContactStd(name='Rect Plank Interaction',
                                        createStepName='Make Contact',
                                        master=frictional_master_surface_region,
                                        slave=frictional_slave_surface_region,
                                        sliding=FINITE,
                                        interactionProperty='Frictional')
```

The same procedure is repeated for the interaction between the plank and the rectangular block, except this time the interaction property 'Frictional' is assigned to the interaction.

12.4.15 Mesh

The following block creates the mesh

```
# -----------------------------------------------------------------------
# Create the mesh

import mesh

# +++++++++++++++++++++++++++++++++++++++++++++++
# Mesh the plank
# We place a point somewhere inside it based on our knowledge of the geometry
plank_inside_coord=(0.0,1.0,40.0)

elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN, secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)

plankCells=plankPart.cells
selectedPlankCells=plankCells.findAt(plank_inside_coord,)
plankMeshRegion=(selectedPlankCells,)
plankPart.setElementType(regions=plankMeshRegion, elemTypes=(elemType1,))

plankPart.seedPart(size=4, deviationFactor=0.1)

plankPart.generateMesh()

# +++++++++++++++++++++++++++++++++++++++++++++++
# Mesh the curved block
curvedBlock_inside_coord=(0.0,-5.0,10.0)

elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN, secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)

curvedBlockCells=curvedBlockPart.cells
selectedCurvedBlockCells=curvedBlockCells.findAt(curvedBlock_inside_coord,)
curvedBlockMeshRegion=(selectedCurvedBlockCells,)
curvedBlockPart.setElementType(regions=curvedBlockMeshRegion,
                               elemTypes=(elemType1,))

curvedBlockPart.seedPart(size=4, deviationFactor=0.1)

curvedBlockPart.generateMesh()

# +++++++++++++++++++++++++++++++++++++++++++++++
# Mesh the rectangular block
rectangularBlock_inside_coord=(10.0,5.0,17.5)

elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN, secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)
```

```
rectangularBlockCells=rectangularBlockPart.cells
selectedRectangularBlockCells=rectangularBlockCells \
                                        .findAt(rectangularBlock_inside_coord,)
rectangularBlockMeshRegion=(selectedRectangularBlockCells,)
rectangularBlockPart.setElementType(regions=rectangularBlockMeshRegion,
                                    elemTypes=(elemType1,))

rectangularBlockPart.seedPart(size=4, deviationFactor=0.1)

rectangularBlockPart.generateMesh()
```

There is nothing new for you to see here, the same meshing procedure has been followed in previous examples. The only difference is that there are 3 part objects in this model, hence the meshing script has been split into 3 blocks.

12.4.16 Create and run the job

The following code runs the job

```
# -----------------------------------------------------------------------
# Create and run the job

import job

# Create the job
mdb.Job(name='ContactSimulationJob', model='Contact Simulation',
        type=ANALYSIS, explicitPrecision=SINGLE,
        nodalOutputPrecision=SINGLE, description='Run the contact simulation',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)

# Run the job
mdb.jobs['ContactSimulationJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['ContactSimulationJob'].waitForCompletion()

# End of run job
```

These statements are similar to ones used previously. You may refer to Section 4.3.13, on page 88.

12.4.17 Post Processing - Display deformed state

The following block displays the deformed state of the plank

```
# -----------------------------------------------------------------------

import visualization

contact_viewport = session.Viewport(name='Contact Simulation Results Viewport')
contact_Odb_Path = 'ContactSimulationJob.odb'
an_odb_object = session.openOdb(name=contact_Odb_Path)
contact_viewport.setValues(displayedObject=an_odb_object)
contact_viewport.odbDisplay.display.setValues(plotState=(DEFORMED, ))
```

These statements are similar to ones used in previous examples. You can refer to the Cantilever Beam example, section 4.3.14 on page 89.

12.5 Summary

In this chapter you worked with contact, created interactions and assigned interaction properties. Contact is commonly encountered both in real life and in simulations that you will be creating in Abaqus.

12.6 What's Next?

At this point we've worked through a number of model setups. Everything we've done so far could also have been implemented in Abaqus/CAE so you haven't really harnessed the power of scripting yet. In subsequent chapters we will reuse some of the scripts you have created here to demonstrate important concepts such as optimization and parameterization.

13

Optimization – Determine the Maximum Plate Bending Loads

13.1 Introduction

We've looked at a number of scripting examples over the last few chapters. In each of these examples we ran not just one aspect of a simulation, but rather the entire simulation from model setup to job execution to post processing using Python scripts. The benefit of having an entire simulation in the form of a script is that you now have the power to programmatically control it, parameterize it, add conditions and loops, and easily alter it for different scenarios. One of the primary uses of scripting is optimization.

In this chapter we shall look at an example of optimization using the planar shell (plate) bending model from Chapter 10. Let's assume you have a large supply of these plates and you'll be using them for construction or in a manufacturing project. It has been decided (for whatever reason) that you can save on material and component costs by maximizing the load borne by each plate. The materials expert has told you that the maximum allowable Mises stress in these plates is 35 MPa. You now need to figure out the maximum load these plates can withstand in bending while experiencing a stress less than 35 MPa in order to optimize your design. Since you aren't really modifying the plate based on the analysis, you aren't optimizing the design of the plate itself, however you will be optimizing your use of resources by loading each of the plates to their maximum capacity – and it is that maximum that you are tasked to find in this example.

13.2 Methodology

We wrote a script in Chapter 10 to run the plate bending simulation. We can modify this same script to run our optimization procedure. The majority of the script will remain the same. This includes the blocks that deal with model, part, material, section, assembly, step, field output request, history output request (we didn't have any), boundary condition, partition and mesh creation. This means over 90% of the script remains unchanged.

The part of the script that needs modification is the application of the load. Since we are using the same concentrated forces and applying them at the same nodes, most of these statements will remain the same too. However we will put them inside a loop. At each iteration of the loop we will increase the magnitude of the concentrated forces. The block that creates and runs the job, as well as the post processing code, will need to be included inside of this loop so that the simulation can be rerun at each iteration of the loop and the results compared to our max stress criteria.

We will need to specify an initial force to use. We shall go with 5N. We will also need to specify how much to increase the force for the next iteration. We can go with a 5N increase at each iteration, so in the next iteration a 10N force will be applied, then 15N and so on. Each analysis job will be given a new name which states the amount of force applied such as PlateJob5N, PlateJob10N and so on. This way all the jobs will be listed in the model tree and output database list as they are created and run, and the user will be able to view the results of any of them if necessary. The results of each analysis will also be displayed in a new viewport which will pop-up over the previous one.

In the plate bending simulation a field output report file was written at the end. In this optimization we will continue to write this field output report file at every iteration. We will then read from this report, and extract the maximum stress from it. We will record this maximum stress by storing it in a file called 'iterative_analysis.txt' in a folder called 'Simulation results' so at the end of all the iterations we will have a table of force vs maximum stress. We will also compare this maximum stress to our maximum allowable stress of 35 MPa and if it has been exceeded we will break out of the loop.

At the end of the analysis we will highlight the elements of the plate which exceeded the maximum allowable stress and display the plate in the viewport so we can see at a glance where the stresses were too high. This gives me a chance to demonstrate how to change an element color within the visualization module.

13.3 Python Script

The following is the completed Python script to accomplish this task. You can find it in the source code accompanying the book in **plate_bending_optimization.py**. You can run it by opening a new model in Abaqus/CAE (**File > New Model Database > With Standard/Explicit Model**) and running it with **File > Run Script…**

```
from abaqus import *
from abaqusConstants import *
import regionToolset
import displayGroupOdbToolset as dgo

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# ----------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Plate Bending Model')
plateModel = mdb.models['Plate Bending Model']

# ----------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the plate using the rectangle tool
plateProfileSketch = plateModel.ConstrainedSketch(name='Plate Sketch',
                                                  sheetSize=20)
plateProfileSketch.rectangle(point1=(0.0,0.0), point2=(5.0,3.0))

# b) Create a shell named "Plate" using the sketch
platePart=plateModel.Part(name='Plate', dimensionality=THREE_D,
                                        type=DEFORMABLE_BODY)
platePart.BaseShell(sketch=plateProfileSketch)

# ----------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs modulus and
# poissons ratio
plateMaterial = plateModel.Material(name='AISI 1005 Steel')
plateMaterial.Density(table=((7872, ),        ))
plateMaterial.Elastic(table=((200E9, 0.29), ))

# ----------------------------------------------------
# Create homogeneous shell section of thickness 0.1m and assign the plate to it
```

```
import section

# Create a section to assign to the plate
plateSection = plateModel.HomogeneousShellSection(name='Plate Section',
                                                  material='AISI 1005 Steel',
                                                  thicknessType=UNIFORM,
                                                  thickness=0.1, preIntegrate=OFF,
                                                  idealization=NO_IDEALIZATION,
                                                  thicknessField='',
                                                  poissonDefinition=DEFAULT,
                                                  thicknessModulus=None,
                                                  temperature=GRADIENT,
                                                  useDensity=OFF,
                                                  nodalThicknessField='',
                                                  integrationRule=SIMPSON,
                                                  numIntPts=5)

# Assign the plate to this section
plate_face_point = (2.5, 1.5, 0.0)
plate_face = platePart.faces.findAt((plate_face_point,))
plate_region = (plate_face,)

platePart.SectionAssignment(region=plate_region, sectionName='Plate Section',
                            offset=0.0, offsetType=MIDDLE_SURFACE, offsetField='')

# -------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
plateAssembly = plateModel.rootAssembly
plateInstance = plateAssembly.Instance(name='Plate Instance', part=platePart,
                                       dependent=ON)

# -------------------------------------------------------
# Create the step

import step

# Create a static general step
plateModel.StaticStep(name='Load Step', previous='Initial',
                      description='Apply concentrated forces in this step')

# -------------------------------------------------------
# Create the field output request

# Change the name of field output request 'F-Output-1' to 'Output Stresses and
# Displacements'
plateModel.fieldOutputRequests.changeKey(fromName='F-Output-1',
                                         toName='Output Stresses and Displacements')
```

```
# Since F-Output-1 is applied at the 'Apply Load' step by default, 'Selected Field
# Outputs' will be too
# We only need to set the required variables
plateModel.fieldOutputRequests['Output Stresses and Displacements'] \
                                                .setValues(variables=('S','UT'))

# -------------------------------------------------------
# Create the history output request

# We don't want any history outputs so lets delete the existing one 'H-Output-1'
del plateModel.historyOutputRequests['H-Output-1']

# -------------------------------------------------------
# Apply boundary conditions - fix one edge

fixed_edge = plateInstance.edges.findAt(((0.0, 1.5, 0.0), ))
fixed_edge_region=regionToolset.Region(edges=fixed_edge)

plateModel.DisplacementBC(name='FixEdge', createStepName='Initial',
                          region=fixed_edge_region,
                          u1=SET, u2=SET, u3=SET, ur1=SET, ur2=SET, ur3=SET,
                          amplitude=UNSET, distributionType=UNIFORM, fieldName='',
                          localCsys=None)

# Instead of using the displacements/rotations boundary condition and setting all
# six DOF to zero
# We could have just used the Encastre condition with the following statement
# plateModel.EncastreBC(name='Encaster edge', createStepName='Initial',
#                                             region=fixed_edge_region)

# -------------------------------------------------------
# Create vertices on which to apply concentrated forces by partitioning part

# Create the datum points
platePart.DatumPointByCoordinate(coords=(0.0, 1.0, 0.0))
platePart.DatumPointByCoordinate(coords=(0.0, 2.0, 0.0))
platePart.DatumPointByCoordinate(coords=(5.0, 1.0, 0.0))
platePart.DatumPointByCoordinate(coords=(5.0, 2.0, 0.0))

# Assign the datum points to variables
# Abaqus stores the 4 datum points in platePart.datums
# Since their keys may or may not start at zero, put the keys in an array sorted
# in ascending order
platePart_datums_keys = platePart.datums.keys()
platePart_datums_keys.sort()
plate_datum_point_1 = platePart.datums[platePart_datums_keys[0]]
plate_datum_point_2 = platePart.datums[platePart_datums_keys[1]]
plate_datum_point_3 = platePart.datums[platePart_datums_keys[2]]
plate_datum_point_4 = platePart.datums[platePart_datums_keys[3]]

# Select the entire face and partition it using two points
partition_face_pt = (2.5, 1.5, 0.0)
```

```
partition_face = platePart.faces.findAt((partition_face_pt,))
platePart.PartitionFaceByShortestPath(point1=plate_datum_point_1,
                                      point2=plate_datum_point_3,
                                      faces=partition_face)

# Now two faces exist, select the one that needs to be partitioned
partition_face_pt = (2.5, 2.0, 0.0)
partition_face = platePart.faces.findAt((partition_face_pt,))
platePart.PartitionFaceByShortestPath(point1=plate_datum_point_2,
                                      point2=plate_datum_point_4,
                                      faces=partition_face)

# Since the partitions have been created, vertices can be extracted
vertices_for_concentrated_force = plateInstance.vertices \
                                                  .findAt(((5.0, 1.0, 0.0),),
                                                          ((5.0, 2.0, 0.0),),)

# -------------------------------------------------------
# Create the mesh

import mesh

# Set element type
plate_mesh_region = plate_region

elemType1 = mesh.ElemType(elemCode=S8R, elemLibrary=STANDARD)
elemType2 = mesh.ElemType(elemCode=STRI65, elemLibrary=STANDARD)

platePart.setElementType(regions=plate_mesh_region, elemTypes=(elemType1,
                                                               elemType2))

# Seed edges by number
mesh_edges_vertical = platePart.edges.findAt(((0.0, 0.5, 0.0), ),
                                             ((0.0, 1.5, 0.0), ),
                                             ((0.0, 2.5, 0.0), ),
                                             ((5.0, 0.5, 0.0), ),
                                             ((5.0, 1.5, 0.0), ),
                                             ((5.0, 2.5, 0.0), ))

mesh_edges_horizontal = platePart.edges.findAt(((2.5, 0.0, 0.0), ),
                                               ((2.5, 1.0, 0.0), ),
                                               ((2.5, 2.0, 0.0), ),
                                               ((2.5, 3.0, 0.0), ))

platePart.seedEdgeByNumber(edges=mesh_edges_vertical, number = 3)
platePart.seedEdgeByNumber(edges=mesh_edges_horizontal, number=10)

platePart.generateMesh()


#IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
#*************************************************************************
```

```
#          ITERATIVE ANALYSIS BEGINS HERE

# Make changes here
concentrated_force_initial_guess = 5
concentrated_force_guess_increment_size = 5
max_acceptable_stress = 35E+3

max_stress = 0
concentrated_force = concentrated_force_initial_guess
force_list=[]
stress_list=[]

while float(max_stress) < max_acceptable_stress :

        # -----------------------------------------------------------------
        # Apply concentrated forces

        plateModel.ConcentratedForce(name='Concentrated Forces',
                                     createStepName='Load Step',
                                     region=(vertices_for_concentrated_force,),
                                     cf3=-concentrated_force,
                                     distributionType=UNIFORM)

        # -----------------------------------------------------------------
        # Create and run the job

        import job

        # Create the job

        job_name='PlateJob'+repr(concentrated_force)+'N'

        mdb.Job(name=job_name, model='Plate Bending Model', type=ANALYSIS,
                               description='Job simulates the bending of a plate')

        # Run the job
        mdb.jobs[job_name].submit(consistencyChecking=OFF)

        # Do not return control till job is finished running
        mdb.jobs[job_name].waitForCompletion()


        ####################################################################
        #-------------------------------------------------------------------
        #Post Processing
        #-------------------------------------------------------------------
        ####################################################################

        # -----------------------------------------------------
        # Display deformed state

        import visualization
```

```
plate_viewport = session \
               .Viewport(name='Plate Results Viewport for force of ' + \
               repr(concentrated_force) + 'N')
plate_Odb_Path = job_name + '.odb'
an_odb_object = session.openOdb(name=plate_Odb_Path)
plate_viewport.setValues(displayedObject=an_odb_object)
plate_viewport.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))

# -----------------------------------------------------------------
# Report stresses in descending order

import odbAccess

# The main session viewport must be set to the odb object using the
# following line. If not you might receive an error message that states
# "There are no active entities. No report has been generated."
session.viewports['Viewport: 1'].setValues(displayedObject=an_odb_object)

# Set the option to display the reported quantity (in our case the
# stresses) in descending order
session.fieldReportOptions.setValues(sort=DESCENDING)

# Name the report and give it a path. If you do not assign a path (as is
# done here) it will be stored in the default abaqus temporary directory
report_name_and_path='PlateReport'+repr(concentrated_force)+'N.rpt'

# .................................................................
# You may enter an entire path if you wish to have the report stored in a
# particular location.
# One way to do it is using the following syntax.
# report_name='PlateReport'
# report_path='C:/MyNewFolder/'
# report_name_and_path = report_path + report_name + '.rpt'
# Alternatively you could have used 1 statement instead of these 3 :
# report_name_and_path='C:/MyNewFolder/PlateReport.rpt'

# Note however that the folder 'MyNewFolder' must exist otherwise you will
# likely get the following error
# "IOError:C/MyNewFolder: Directory not found"
# You must either create the folder in Windows before running the script
# Or if you wish to create it using Python commands you must use the
# os.makedir() or os.makedirs() function
# os.makedirs() is preferable because you can create multiple nested
# directories in one statent if you wish
# Note that this function returns an exception if the directory already
# exists hence it is a good idea to use a try block

#try:
#    os.makedirs(report_path)
#except:
#    print "Directory exists hence no need to recreate it. Move on to next
```

```
    #                                                              statement"

    # Rewriting with the comments removed
    """
    report_name='PlateReport'
    report_path='C:/MyNewFolder/'
    report_name_and_path = report_path + report_name + '.rpt'
    try:
        os.makedirs(report_path)
    except:
        print "Directory exists hence no need to recreate it. Move on to \
                                                          next statement"
    """
    # ................................................................

    # Write the field report outputting the Mises stresses
    session.writeFieldReport(fileName=report_name_and_path, append=OFF,
              sortItem='S.Mises', odb=an_odb_object, step=0,
              frame=1, outputPosition=INTEGRATION_POINT,
              variable=(('S', INTEGRATION_POINT, ((INVARIANT, 'Mises'), )), ))

    # ----------------------------------------------------------------
    # Read the maximum stress from the report

    extracted_line=''

    f=open(report_name_and_path)
    for line in f:
        str=line
        if 'Maximum' in str:
        # or you can use the statement
        # if str.find('Maximum') != -1 :
            extracted_line = str
            break

    extracted_list = extracted_line.split()
    max_stress = extracted_list[2]

    f.close()

    # Place the force and corresponding stress in a list
    force_list.append(concentrated_force)
    stress_list.append(max_stress)

    # Increase the concentrated force by the defined step size for the next
    # iteration
    concentrated_force = concentrated_force + \
                                concentrated_force_guess_increment_size


# Output a table of force and stress to specified location
```

```
try:
    os.makedirs('C:/SimulationResults/')
except:
    print "! Something went wrong.. maybe this directory already exists?"

g = open('C:/SimulationResults/iterative_analysis.txt','w')
g.write('Force \t Max Stress \n')

for i in range(len(force_list)):
    g.write(repr(force_list[i]) + '\t' + stress_list[i] + '\n')

g.write('\n\n')
g.write('The force that will cause the maximum allowable stress of ' + \
                        repr(max_acceptable_stress) + 'N is between ' + \
                        repr(force_list[len(force_list)-2]) + 'N and ' + \
                        repr(force_list[len(force_list)-1]) + 'N.')
g.close()

#           ITERATIVE ANALYSIS ENDS HERE
#***************************************************************************
#IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII


# Determine which elements exceed the maximum allowable stress

highlight_element_list = []

h = open(report_name_and_path)

# In the report file, the table begins after a line full of '-' characters.
# Extract lines one by one until this line is reached.
for line in h:
    str=line
    if '----------------------------------' in str:
        break

# Read in lines one at a time from the table
for line in h:
    str = line
    # Test if a blank line has occurred as this will signal the end of the table
    if not line.strip():
        break
    str_list = str.split()
    if float(str_list[3]) > max_acceptable_stress:
        highlight_element_list.append(str_list[0])

h.close()

# Convert list to tuple
highlight_element_tuple = tuple(highlight_element_list)
```

```
# Change the color of elements in the viewport where the stress exceeded the
# maximum acceptable stress
plate_viewport.setColor(initialColor='#FFFF00', translucency=0.4)
leaf = dgo.LeafFromModelElemLabels(elementLabels=(('PLATE INSTANCE',
                                                   highlight_element_tuple), ))
plate_viewport.odbDisplay.display.setValues(plotState=(DEFORMED, ))
plate_viewport.setColor(leaf=leaf, fillColor='Red')

# Print a message for the user in the message area
print '************************'
print 'Optimization complete'
print 'Multiple viewports have been created, one for each simulation. \n'
print 'These may appear one above the other and you will need to move them ' + \
                                   'around your screen to reveal the ones behind. \n'
print 'The last viewport to be created (the one on top) highlights in red the' + \
                             'elements which exceeded the maximum allowable stress. \n'
```

13.4 Examining the Script

Let's understand how this script works. The majority of the script is the same as the one in the Planar Shell bending example of Chapter 10 so we'll only examine the new parts.

13.4.1 Model, Part, Material, Section, Assembly, Step, Field Output Request, Boundary Condition, Partition and Mesh creation.

These lines are identical to those used in Chapter 10 (with the addition of a new **import** statement) and should require no explanation. The new **import** statement is discussed in section 13.4.6 on page 345.

13.4.2 Initialization

The following block initializes the variables for the loop.

```
#IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
#**************************************************************************
#            ITERATIVE ANALYSIS BEGINS HERE

# Make changes here
concentrated_force_initial_guess = 5
concentrated_force_guess_increment_size = 5
max_acceptable_stress = 35E+3

max_stress = 0
concentrated_force = concentrated_force_initial_guess
force_list=[]
stress_list=[]
```

We store our initial force guess of 5N in a variable called **concentrated_force_initial_guess**. The amount by which to increment the force for the next iteration (5N) is assigned to the variable **concentrated_force_guess_increment_size**. And our maximum allowable stress criterion of 35MPa (or 35000 Pa) is assigned to **max_acceptable_stress**.

At the end of each iteration, the maximum stress will be extracted from the report file and stored in the variable **max_stress**. The value in this variable will be tested at the beginning of each iteration to make sure it is less than 35 MPa. We set it to 0 Pa for the first iteration to ensure that this condition is satisfied and the loop executes at least once.

We will use the variable **concentrated_force** to store the magnitude of the force being used in each iteration. For the first iteration we will assign it the value stored in **concentrated_force_initial_guess**. You might ask why we do not just write

```
concentrated_force = 5
```

This is because it is better to have all the variables that can be changed by a user listed in one spot. Hence **concentrated_force_initial_guess**, **concentrated_force_guess_increment_size** and **max_acceptable_stress** are all listed one after another so any changes that need to be made can be made in one spot. In fact it might be an even better idea to put these variables all the way at the top of your script.

13.4.3 Modify and run the analysis at each iteration

At each iteration the following lines test to see if the criterion has been met, and then modify and run the analysis.

```
while float(max_stress) < max_acceptable_stress :

        # ----------------------------------------------------------------
        # Apply concentrated forces

        plateModel.ConcentratedForce(name='Concentrated Forces',
                                     createStepName='Load Step',
                                     region=(vertices_for_concentrated_force,),
                                     cf3=-concentrated_force,
                                     distributionType=UNIFORM)

        # ----------------------------------------------------------------
        # Create and run the job

        import job
```

```
# Create the job

job_name='PlateJob'+repr(concentrated_force)+'N'

mdb.Job(name=job_name, model='Plate Bending Model', type=ANALYSIS,
                       description='Job simulates the bending of a plate')

# Run the job
mdb.jobs[job_name].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs[job_name].waitForCompletion()


####################################################################
#------------------------------------------------------------------
#Post Processing
#------------------------------------------------------------------
####################################################################

# -----------------------------------------------------
# Display deformed state

import visualization

plate_viewport = session \
                .Viewport(name='Plate Results Viewport for force of ' + \
                repr(concentrated_force) + 'N')
plate_Odb_Path = job_name + '.odb'
an_odb_object = session.openOdb(name=plate_Odb_Path)
plate_viewport.setValues(displayedObject=an_odb_object)
plate_viewport.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))

# -----------------------------------------------------------------
# Report stresses in descending order

import odbAccess

# The main session viewport must be set to the odb object using the
# following line. If not you might receive an error message that states
# "There are no active entities. No report has been generated."
session.viewports['Viewport: 1'].setValues(displayedObject=an_odb_object)

# Set the option to display the reported quantity (in our case the
# stresses) in descending order
session.fieldReportOptions.setValues(sort=DESCENDING)

# Name the report and give it a path. If you do not assign a path (as is
# done here) it will be stored in the default abaqus temporary directory
report_name_and_path='PlateReport'+repr(concentrated_force)+'N.rpt'
```

```
        # Write the field report outputting the Mises stresses
        session.writeFieldReport(fileName=report_name_and_path, append=OFF,
                  sortItem='S.Mises', odb=an_odb_object, step=0,
                  frame=1, outputPosition=INTEGRATION_POINT,
                  variable=(('S', INTEGRATION_POINT, ((INVARIANT, 'Mises'), )), ))

        # -------------------------------------------------------------------
        # Read the maximum stress from the report

        extracted_line=''

        f=open(report_name_and_path)
        for line in f:
            str=line
            if 'Maximum' in str:
            # or you can use the statement
            # if str.find('Maximum') != -1 :
                extracted_line = str
                break

        extracted_list = extracted_line.split()
        max_stress = extracted_list[2]

        f.close()

        # Place the force and corresponding stress in a list
        force_list.append(concentrated_force)
        stress_list.append(max_stress)

        # Increase the concentrated force by the defined step size for the next
        # iteration
        concentrated_force = concentrated_force + \
                                        concentrated_force_guess_increment_size
```

```
while float(max_stress) < max_acceptable_stress :
```

This is the condition of the **while** loop. It tests to see if the maximum stress recorded at the previous iteration (**max_stress**) is less than the maximum allowable stress (**max_acceptable_stress**). If this condition is true, the contents of the **while** loop will be executed. If not, Abaqus will break out of the **while** loop and control will be passed to the parts of the script after it.

The **float()** method is used here is a built-in function of the Python language. It is used to convert arguments passed to it into a Float data type. For example, an integer 435 can be converted to a float as

```
float_var = float(435)
```

float_var now contains the value 435.0 (notice the .0 which makes it a float as opposed to an integer).

Similarly a String can be converted into a float.

```
String_var = "435.0"
float_var = float(String_var)
```

float_var now contatins the value 435.0.

The reason the **float()** method is used here is because **max_stress**, which is extracted from the report file at the end of each iteration, will be of type String. This is because text extracted from a file will always be treated by Python as a String, even if it appears to you as a number. It is not really possible to compare a string to a float. Python will not give you an error, it will just return FALSE for the condition. For example,

```
while "5" == 5 :
```

would return false even though it is actually true, because Python sees a String on one side and a float on the other. Hence we must use the **float()** method to convert **max_stress** to a float.

```
# ----------------------------------------------------------------
# Apply concentrated forces

plateModel.ConcentratedForce(name='Concentrated Forces',
                             createStepName='Load Step',
                             region=(vertices_for_concentrated_force,),
                             cf3=-concentrated_force,
                             distributionType=UNIFORM)
```

This statement is a modified version of the one used in the Planar Shell Bending example, section 10.4.12 on page 248. The only difference is that the force in the 3-direction has been replaced with the variable **concentrated_force** so that it changes to the value of the force guess at each iteration.

```
# ----------------------------------------------------------------
# Create and run the job

import job

# Create the job

job_name='PlateJob'+repr(concentrated_force)+'N'
```

```
mdb.Job(name=job_name, model='Plate Bending Model', type=ANALYSIS,
                       description='Job simulates the bending of a plate')

# Run the job
mdb.jobs[job_name].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs[job_name].waitForCompletion()
```

Few modifications have been made to these statements from section 10.4.14 on page 250 of the Planar Shell Bending example. The only new statement here is the first one which assigns the name of the job to the variable 'job_name'. We give each job a different name so that the the output databases have different names and each job and output database is listed in the model tree. This way the results for any of the iterations can be visualized later if required.

The **repr()** command used is a built-in function of Python that returns a String containing a printable representation of an object. Since **job_name** will be assigned to the **name** argument of the **Job()** method, it needs to be a String. Earlier we used the **float()** function to convert a String to a Float, here we use the **repr()** function to convert an integer to a String. So if **concentrated_force** = 5, then the statement will evaluate to *job_name = 'PlateJob5N'*.

```
####################################################################
#-------------------------------------------------------------------
#Post Processing
#-------------------------------------------------------------------
####################################################################

# ------------------------------------------------------
# Display deformed state

import visualization

plate_viewport = session \
                .Viewport(name='Plate Results Viewport for force of ' + \
                repr(concentrated_force) + 'N')
plate_Odb_Path = job_name + '.odb'
an_odb_object = session.openOdb(name=plate_Odb_Path)
plate_viewport.setValues(displayedObject=an_odb_object)
plate_viewport.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))

# -------------------------------------------------------------------
# Report stresses in descending order

import odbAccess
```

```
# The main session viewport must be set to the odb object using the
# following line. If not you might receive an error message that states
# "There are no active entities. No report has been generated."
session.viewports['Viewport: 1'].setValues(displayedObject=an_odb_object)

# Set the option to display the reported quantity (in our case the
# stresses) in descending order
session.fieldReportOptions.setValues(sort=DESCENDING)

# Name the report and give it a path. If you do not assign a path (as is
# done here) it will be stored in the default abaqus temporary directory
report_name_and_path='PlateReport'+repr(concentrated_force)+'N.rpt'
```

A few modifications have been made to this code from the Planar Shell Bending example, section 10.4.14 on page 250, to include the new analysis job naming convention. This is seen in the third, fourth and last lines of the snippet above.

The name of the viewport is the string "Plate Results Viewport for force of" concatenated with the force magnitude and the unit "N". So for example if the concentrated force is 5N, our viewport will have the name 'Plate Results Viewport for force of 5N'.

The name of the job is stored in the **job_name** variable, and that is concatenated with the String '.odb' and assigned to the variable **plate_Odb_Path**. So if the **job_name** variable holds "PlateJob5N", the variable **plate_Odb_Path** is now "PlateJob5N.odb". Similarly **report_name_and_path** will be assigned "PlateReport5N.rpt".

```
# Write the field report outputting the Mises stresses
session.writeFieldReport(fileName=report_name_and_path, append=OFF,
          sortItem='S.Mises', odb=an_odb_object, step=0,
          frame=1, outputPosition=INTEGRATION_POINT,
          variable=(('S', INTEGRATION_POINT, ((INVARIANT, 'Mises'), )), ))
```

This **writeFieldReport()** command has been copied from the Planar Shell Bending example, section 10.4.16 on page 251 .

```
# ---------------------------------------------------------------
# Read the maximum stress from the report

extracted_line=''

f=open(report_name_and_path)
for line in f:
    str=line
    if 'Maximum' in str:
    # or you can use the statement
```

```
        # if str.find('Maximum') != -1 :
            extracted_line = str
            break

    extracted_list = extracted_line.split()
    max_stress = extracted_list[2]

    f.close()
```

This code extracts the maximum value of the Mises stress from the report. If you examine the report, the bottom half looks something like:

```
                10            4          940.912          940.912
                71            1          867.384          867.384
                30            1          798.683          798.683
                30            3          665.845          665.845
                81            1          492.472          492.472
                20            4          474.511          474.511
                30            2          314.078          314.078
                30            4          152.557          152.557


  Minimum                                152.557          152.557
      At Element                              30               30
          Int Pt                               4                4

  Maximum                            9.27365E+03      9.27365E+03
      At Element                              50               50
          Int Pt                               1                1

           Total                     881.550E+03      881.550E+03
```

We open this file using the **open()** command, where you provide the filename as an argument, and assign it to the variable **f** which becomes a file object, that you can think of as a file handle. The file can then be accessed in subsequent parts of the script using this handle **f**.

Aside from the mandatory filename argument, there are other optional ones such as the **mode**. The commonly used modes are **r** for read, **w** for write (which will delete the file if it already exists), and **a** for append (which will add on to an existing file, or create a new one if it doesn't already exist). The default is **r** which suits our purposes since we wish to read from this file. The Python documentation lists other modes and optional arguments for the **open()** function.

When Python sees the statement *for line in f:*, it begins a **for** loop, and at each iteration it extracts one line from the file. It will sequentially move down the report file extracting one line at each iteration and storing it in the variable **line**.

We assign the String in **line** to another variable **str** for convenience.

We are trying to find the line in the file that looks something like (refer to image): "`Maxium                    9.27365E+03               9.27365E+03`"

The way to check if the variable **str** contains this line of output is to look for the word 'Maximum' in it since this word will only occur in that one line of our report file. The statement **if 'Maximum' in str:** is an **if** statement which checks to see if the String value in variable **str** contains the String 'Maximum'. If this condition is true, the statements within the **if** block will be executed.

This could also have been accomplished using the Python built-in function **(String).find()** as

```
if str.find('Maximum') != -1 :
```

where the **find()** function returns the lowest index in the String where the argument is found, or -1 if it is not found.

Once found, the contents of the line are assigned to the variable **extracted_line**. A **break** statement then moves control out of the loop.

Right now the value of the **extracted_line** variable looks something like this: "`Maxium                    9.27365E+03               9.27365E+03`"

We need to somehow extract the first number 9.27365E+03 from it. We can use the Python **(String).split()** command. **split()** does not have any required arguments, and when no arguments are passed it will split up the String wherever it encounters a space, and return a list of the words. What we will end up with here is *extracted_list[0] = 'Maximum'*, *extracted_list[1] = '9.27365E+0.3'* and *extracted_list[2] = '9.2736E+03'*. (Note that the each list item is a String even if these look like floats, because Python reads in a line from a file as a String).

As an aside, FYI, the **split()** function can be told to split a line at a character other than a space; for this you would need to specify that character as an argument. For example, *str.split('9')* would split up our line wherever the letter/number 9 appears. You can also specify, as a second argument, the maximum number of times a String can be split, thus limiting the number of elements in your list. Refer to the Python documentation for more on the **split()** function.

Once the line has been split up into a list of 3 elements in our case, the second one will refer to the maximum Mises stress. We can refer to it using index notation as **extracted_list[2]**.

The file is then closed using **f.close()**. The built-in Python **(file).close()** command closes a file and frees up the system resources taken up by it while it is open. Also the file object **f** can no longer be used to access the file after this.

```
# Place the force and corresponding stress in a list
force_list.append(concentrated_force)
stress_list.append(max_stress)
```

At the end of the simulation our plan is to print out a table of the magnitude of the concentrated force and the corresponding maximum stress. For this reason we use the built-in Python function **append()** to add **concentrated_force** and **max_stress** for the current analysis to the list variables **force_list** and **stress_list** respectively. The **(list).append()** function, which was mentioned in section 3.4 "Lists" on page 44, adds the argument provided to it to the end of a list making it the last element.

```
# Increase the concentrated force by the defined step size for the next
# iteration
concentrated_force = concentrated_force + \
                                concentrated_force_guess_increment_size
```

The concentrated force is then incremented for the next iteration of the optimization process.

13.4.4 Print a table of the results

The force and resultant maximum stress for each analysis is tabulated.

```
# Output a table of force and stress to specified location

try:
    os.makedirs('C:/SimulationResults/')
except:
    print "! Something went wrong.. maybe this directory already exists?"
```

```
g = open('C:/SimulationResults/iterative_analysis.txt','w')
g.write('Force \t Max Stress \n')

for i in range(len(force_list)):
    g.write(repr(force_list[i]) + '\t' + stress_list[i] + '\n')

g.write('\n\n')
g.write('The force that will cause the maximum allowable stress of ' + \
                       repr(max_acceptable_stress) + 'N is between ' + \
                       repr(force_list[len(force_list)-2]) + 'N and ' + \
                       repr(force_list[len(force_list)-1]) + 'N.')
g.close()
```

In Python, exceptions are errors which occur during program execution. They are not syntax errors and it is possible for a program to continue after this if you catch them. This is done using a **try-except** block. You place the statements that might generate an exception inside the **try** block and if an exception does occur the contents of the **except** block will be executed.

```
try:
    os.makedirs('C:/SimulationResults/')
except:
    print "! Something went wrong.. maybe this directory already exists?"
```

Here we use **try-except** when creating a new directory for our simulation results. It is possible that the script will be unable to create the new folder because it already exists, or because your Windows account doesn't have the necessary permissions, or some other unforeseeable reason. You can use an **except** block to deal with the situation. In this example we don't actually implement any additional code to deal with the problem, however having the **try-except** in place will prevent a known error from occurring (as we will see in a moment). Good programming practice of course would involve having some code to actually deal with the cause of the exception but I'm only trying to demonstrate the use of **try-except** in this example.

Also note the use of the exclamation point '!' in the print statement. This tells Abaqus to print this statement in red color font in the message area, and you might even hear a beep when it is printed. Just a cool feature I thought you'd want to know about.

If you wish to see it in action, run the script in Abaqus, then open a new model database and run the script again. Since the directory 'C:/SimulationResults' will be created the first time you run the script, the second time you will get an exception when trying to make the folder because one with the same name already exists. In this case you will see a message in the message area at the bottom of the window saying "Something is wrong.. maybe this directory already exists?" And Abaqus then proceeds to the next statement in the **except** block, or if there isn't any (as in this example) it then moves on to the next statement in the script.

If we did not have the **try-except** code in place, we would have got a Windows error here and the script would have terminated at that point. The presence of the **try-except** statements makes Python ignore the exception and it assumes you will do something to fix it in the except-block. In our example everything works out once the exception occurs because the directory we need does already exist and subsequent statements are therefore not affected. If on the other hand the directory did not exist, and our exception was raised for some other reason such as insufficient user privileges associated with the Windows account, our script would still continue on assuming we've dealt with the problem and we'd see some other errors further down the line since the directory to which we output the file will not exist.

As for the **os.makedirs()**, this is a built-in Python function which creates directories. The directory path must be provided as an argument. For your information, Python also has a **mkdir()** function. The different between **mkdir()** and **makedirs()** is that **makedirs()** allows you to create a whole path – such as a new directory within another new directory – using one command – whereas with **mkdir()** you'd have to create each nested directory in turn.

```
g = open('C:/SimulationResults/iterative_analysis.txt','w')
```

We use the built-in Python **open()** function which you have seen in the previous section. This time, aside from the required filename/path argument, we also provide an optional mode argument **w**. This opens the file in write mode. If a file with this name already exists it will be replaced.

```
g.write('Force \t Max Stress \n')
```

The built-in Python **write()** function is used to write the first line of the file, which will form the heading of the table. The **\t** translates into a tab space, while the **\n** is a newline (or carriage return) character; so the next time the **write()** function is used the text will be written on a new line.

```
for i in range(len(force_list)):
    g.write(repr(force_list[i]) + '\t' + stress_list[i] + '\n')
```

A **for** loop was used to iterate as many times as there are elements in **force_list**. The **write()** function is then used to write the forces and stresses. Remember that the elements of **force_list** are integers, hence the **repr()** function is used to convert them to Strings that can be printed into a file. The variables in **stress_list** on the other hand already Strings since they were read in from the report files for each analysis.

```
g.write('\n\n')
g.write('The force that will cause the maximum allowable stress of ' + \
                        repr(max_acceptable_stress) + 'N is between ' + \
                        repr(force_list[len(force_list)-2]) + 'N and ' + \
                        repr(force_list[len(force_list)-1]) + 'N.')
g.close()

#           ITERATIVE ANALYSIS ENDS HERE
#****************************************************************************
#IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
```

These lines add the finishing touches to our output file by stating that the magnitude of the force that causes the maximum allowable stress lies somewhere between the last two force attempts. We refer to them within the force list using index notation as *[len(force_list)-2]* and *[len(force_list)-1]*.

That brings us to the end of the iterative analysis procedure.

13.4.5 Read the report file to determine where the maximum stress was exceeded

```
# Determine which elements exceed the maximum allowable stress

highlight_element_list = []

h = open(report_name_and_path)

# In the report file, the table begins after a line full of '-' characters.
# Extract lines one by one until this line is reached.
for line in h:
    str=line
    if '----------------------------------' in str:
        break

# Read in lines one at a time from the table
for line in h:
    str = line
    # Test if a blank line has occurred as this will signal the end of the table
    if not line.strip():
            break
        str_list = str.split()
        if float(str_list[3]) > max_acceptable_stress:
            highlight_element_list.append(str_list[0])

h.close()

# Convert list to tuple
highlight_element_tuple = tuple(highlight_element_list)
```

We now read the last report file to determine which of the elements experienced a stress larger than the maximum allowable pressure. We will later highlight these elements in the viewport.

```
highlight_element_list = []
```

This variable is a placeholder for the list of elements with a stress exceeding the critical stress.

```
h = open(report_name_and_path)
```

The built-in Python function **open()** is used again here to open the last report file. The variable **report_name_and_path** still holds the path of the last report file from the final iteration.

The report file will look something like this:

```
********************************************************************
****************
Field Output Report, written Sun Mar 06 21:29:45 2011

Source 1
---------

   ODB: C:/AbaqusTemp/PlateJob20N.odb
   Step: Load Step
   Frame: Increment      1: Step Time =    1.000

Loc 1 : Integration point values at shell < "AISI 1005 STEEL" > <
5 section points > from source 1 : SNEG, (fraction = -1.0)
Loc 2 : Integration point values at shell < "AISI 1005 STEEL" > <
5 section points > from source 1 : SPOS, (fraction = 1.0)

Output sorted by column "S.Mises".

Field Output reported at integration points for part: PLATE
INSTANCE

         Element             Int        S.Mises        S.Mises
           Label              Pt         @Loc 1         @Loc 2
------------------------------------------------------------------
              50               1    37.0946E+03    37.0946E+03
              60               1    37.0573E+03    37.0573E+03
               1               3    36.5112E+03    36.5112E+03
              70               1    36.3201E+03    36.3201E+03
              40               1    36.2958E+03    36.2958E+03
               1               1    36.1592E+03    36.1592E+03
              21               4    36.0272E+03    36.0272E+03
              11               1    35.9607E+03    35.9607E+03
               1               2    35.3670E+03    35.3670E+03
              89               1    35.0950E+03    35.0950E+03
              22               3    34.5809E+03    34.5809E+03
              80               1    34.3091E+03    34.3091E+03
              11               3    34.1773E+03    34.1773E+03
              22               1    33.7113E+03    33.7113E+03
              21               2    33.5645E+03    33.5645E+03
```

(Many rows removed)

```
              81               1    1.96989E+03    1.96989E+03
              20               4    1.89805E+03    1.89805E+03
              30               2    1.25631E+03    1.25631E+03
              30               4        610.226        610.226


  Minimum                               610.226        610.226
      At Element                             30             30
          Int Pt                              4              4

  Maximum                           37.0946E+03    37.0946E+03
      At Element                             50             50
          Int Pt                              1              1

           Total                    3.52620E+06    3.52620E+06
```

We need to determine where the line of '-' characters appears and the data table begins after it.

```
# In the report file, the table begins after a line full of '-' characters.
# Extract lines one by one until this line is reached.
for line in h:
    str=line
    if '----------------------------------' in str:
         break
```

The **for** loop reads in lines from the file one by one checking to see if a large number of '-' characters appear in sequence. When this happens, the program control breaks out of the loop.

```
# Read in lines one at a time from the table
for line in h:
    str = line
    # Test if a blank line has occurred as this will signal the end of the table
    if not line.strip():
         break
    str_list = str.split()
    if float(str_list[3]) > max_acceptable_stress:
         highlight_element_list.append(str_list[0])
```

The next **for** loop continues to read in lines. Note that it does not start reading lines from the top of the report file. Instead it continues to read from where it left off in the previous **for** loop, since the file was not closed after that. This is Python behavior that you should be aware of.

We now check to see if a blank line appears, since the line after the table is followed by two blank lines in the report file. The **line.strip()** method is used. **strip()** is a built-in Python function that strips off leading and trailing characters in a String. The character (or multiple characters) needs to be specified as an argument, or **strip()** defaults to a whitespace character, which is what happens in this case. The **strip()** function is good for removing leading or trailing whitespaces in a String. In our case however, since the entire line is made up of whitespaces, **strip()** returns ''. This is basically a false condition. This is because values such as 0, None and '' are treated as 'false' in a Boolean sense. Hence the **if** condition returns true (since a 'not' is used), and the **break** statement removes program control from the loop.

You might be interested to know that Python also has 2 built-in functions called **lstrip()** and **rstrip()** which function similar to **strip()**. **lstrip()** removes leading characters, whereas **rstrip()** removes trailing characters.

The Python **split()** function is used once again to split each line of the table into list elements where the whitespaces occur. In this case we should get 4 elements since there are 4 columns. The subsequent **if** statement compares the value of the 4th (since indices begin at 0) element of the new list, which is the stress on the top surface of the shell elements, with the maximum acceptable stress. If the critical stress is exceeded, the element label (which is the first column or list index 0) is added to **highlight_element_list** with the **append()** function.

```
h.close()
```

The **close()** function closes the file and frees up any resources it was using while open.

```
# convert list to tuple
highlight_element_tuple = tuple(highlight_element_list)
```

The built-in Python function **tuple()** is used to convert the list to a tuple. This is because a tuple is required by the **LeafFrommModelElemLabels()** method which we will be using shortly.

13.4.6 Light up elements in the viewport where max stress is exceeded

The following code lights up in red the elements in the viewport where the stress exceeds the maximum allowable stress. The rest of the elements are colored yellow.

```
# Change the color of elements in the viewport where the stress exceeded the
# maximum acceptable stress
plate_viewport.setColor(initialColor='#FFFF00', translucency=0.4)
leaf = dgo.LeafFromModelElemLabels(elementLabels=(('PLATE INSTANCE',
                                                    highlight_element_tuple), ))
plate_viewport.odbDisplay.display.setValues(plotState=(DEFORMED, ))
plate_viewport.setColor(leaf=leaf, fillColor='Red')
```

There are 3 different **setColor()** methods available. 2 of them have been used here.

```
plate_viewport.setColor(initialColor='#FFFF00', translucency=0.4)
```

The **setColor()**method used in the first line assigns an initial color and translucency to the elements of the plate before we highlight the ones that have exceeded critical stress. This

setColor() method has one required argument, **initialColor**, which is a String specifying the initial color of the objects. It also has an optional argument **translucency** which specifies how translucent the object drawn with **initialColor** will be. It must be a float between 0.0 and 1.0. We do not use this optional argument here.

Note that we specified **initialColor** using the hexadecimal #FFFF00. Common colors can also be represented by their names, in this case 'Yellow'. Hence the statement could have been written as:

```
plate_viewport.setColor(initialColor='Yellow')
```

The next statement

```
leaf = dgo.LeafFromModelElemLabels(elementLabels=(('PLATE INSTANCE',
                                                   highlight_element_tuple), ))
```

creates a **Leaf** object. **Leaf** objects are used for items in a display group. They are temporary objects and they are created to use in **DisplayGroup** commands such as **setColor()**. The **LeafFromModelElemLabels()** method creates a **LeafFromModelElemLabels** object, which is derived from the **Leaf** object and can therefore be used in place of a **Leaf** object in **DisplayGroup** commands. **LeafFromModelElemLabels()** has one required argument, **elementLabels**, which is a sequence of Strings denoting part instances in the model and a sequence of the labels of the desired elements in that part instance. Hence it should be of the form

```
(('part_instance_1',    ('elem_label_1',    'elem_label_2',    'elem_label_3'),),
('part_instance_2', ('elem_label_1', 'elem_label_2', 'elem_label_3'),)).
```

We already have ('elem_label_1', 'elem_label_2', 'elem_label_3') in the form of a tuple/sequence **highlight_element_tuple** created earlier specifically for this purpose.

The **dgo** refers to the **displayGroupOdbToolset** which was imported using an import statement at the start of the script:

```
import displayGroupOdbToolset as dgo
```

The **displayGroupOdbToolset** defines **Leaf** objects. Hence we must use **dgo.LeafFromModelElemLabels()** to refer to it.

```
plate_viewport.odbDisplay.display.setValues(plotState=(DEFORMED, ))
```

This statement changes the displayed view in the viewport to a deformed view. You have seen this method used in many previous examples. The reason we are using it is because the display is currently a color contour of the stresses, and it will not be possible to highlight elements if a color contour is currently displayed on them. By changing it to deformed view it will be deformed but will not have a stress contour displayed on it.

```
plate_viewport.setColor(leaf=leaf, fillColor='Red')
```

Once again we use the **setColor()** method, but this one is a little different from the one used a few statements earlier even though it shares the same name. This **setColor()** method also specifies the color of a **Leaf** object, but it has a different set of required and optional arguments. The required argument **leaf** is a **Leaf** object. We use the one created with the **LeafFromModelElemLabels()** method. There are numerous optional arguments - **fillColor**, **edgeColorFillShade**, **edgeColorWireHide**, **nodeSymbolColor**, **nodeSymbolType** and **nodeSymbolSize**, all of which are described in the Abaqus Scripting Reference Manual. The only one we use is **fillColor**, which is a String specifying the color to be used for the faces of the elements. Needless to say, it is only applicable when the render style is filled or shaded. Just like the previously used **setColor()** method, we can use a hexadecimal value or a common color name, in this case 'Red'.

13.4.7 Print messages to the message area

The following code ads some finishing touches to the script by writing a message to Abaqus' message area.

```
# Print a message for the user in the message area
print '************************'
print 'Optimization complete'
print 'Multiple viewports have been created, one for each simulation. \n'
print 'These may appear one above the other and you will need to move them ' + \
                                'around your screen to reveal the ones behind. \n'
print 'The last viewport to be created (the one on top) highlights in red the' + \
                         'elements which exceeded the maximum allowable stress. \n'
```

The **print** statement is used to send messages to the message area. The newline (aka carriage return) symbol **\n** is the equivalent of hitting the Enter key at the end of each sentence, causing the next one to begin on a new line. The contents of the print statements are quite self-explanatory.

13.5 Summary

After reading through this chapter you should now be able to perform an optimization by placing the bulk of your script inside of a loop and iterating through it. This is the standard procedure when performing optimizations using Python scripts. You also performed some of the most common file handling (input/output) tasks using the generated report files. In the process you were introduced to **try-catch** blocks for catching exceptions. And you learnt how to change the color of interesting elements in the viewport, adding to your knowledge of post-processing through a script.

14

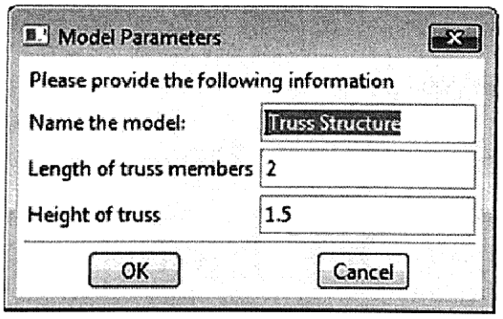
Parameterization, Prompt Boxes and XY Plots

14.1 Introduction

One of the most basic reasons for writing a script is that it gives you the ability to parameterize your model. This allows you to specify quantities in the form of variables whose values can be changed at runtime. If one of your dimensions is a variable, you can create your model geometry making use of that variable, and you'll then have the ability to change your model by changing that variable.

You already got a taste of this concept in the previous chapter with the plate, where the concentrated force was stored in the form of a variable whose value changed at every iteration. But this was a relatively simple example. You can in fact have many quantities in the form of variables which depend on the other variables. For example, you could specify the length of a truss member as a variable, and the cross sectional area as a variable which is related to the length by some mathematical relation. If you change the first variable, your script not only changes the length of the wire feature in the sketcher, it also changes the section properties accordingly. Or if you were working with beams you could have the script change the profile dimensions to make them some fraction of the length.

We will perform a similar parameterization in this chapter using the truss structure under dynamic loading from Chapter 9. In addition we will obtain the length of the beam members, as well as the magnitude of the concentrated force, as inputs from the user at runtime using prompt boxes. The ability to accept user input through a prompt box is a neat feature which allows the analyst to easily define a few variable values and observe the response of the model. We will demonstrate the use of a prompt box which accepts one input, as well as a prompt box that accepts multiple inputs.

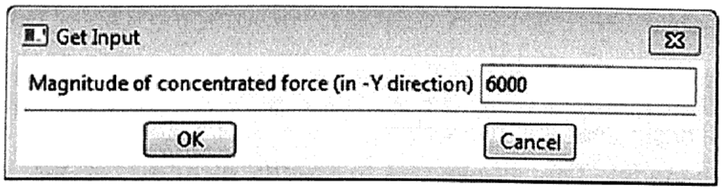

Model Parameters

Please provide the following information

Name the model: Truss Structure

Length of truss members 2

Height of truss 1.5


Get Input

Magnitude of concentrated force (in -Y direction) 6000

In addition we will revisit the XY plots created using history outputs, and play around with the plot characteristics. We'll change the characteristics and styles of the plot titles, axes, legends and so on. Quite often you will find yourself performing the same repetitive steps to visualize a result every time you run an analysis, and you can save some time and effort by writing these steps as a script. Although not the case in this example, it is quite popular to create standalone scripts for post-processing tasks which are only run after the analysis has completed.

14.2 Methodology

When the analyst runs the script, he or she will be prompted to type in the length of the truss members (they are all of equal length) and the height of the truss within a single prompt box. The script will be modified or parameterized so the part sketch will scale to these dimensions. The truss cross section area, which is a property assigned in the section module, will also be recalculated based on these dimensions. The radius of the truss members will be 0.05% of the length, and the cross section area will be calculated using this radius.

Recall that the **findAt()** method is used to find (and select) the truss members in order to assign section properties to them. Since the truss dimensions will now change based on user input, the locations of these members will also change, hence the arguments of the

findAt() method will need to be parameterized as well so they can dynamically update with the model geometry.

The user will also be prompted to enter the magnitude of the concentrated force, and this will be applied to the correct node (the one in the center). The history output will be requested from the node at the end of the structure. Note that the coordinates of both these nodes will depend on the geometry of the truss hence the **findAt()** method will once again be parameterized here.

14.3 Python Script

The following listing is the completed Python script to accomplish this. You can find it in the source code accompanying the book in **truss_dynamic_parameterized.py**. You can run it by opening a new model in Abaqus/CAE (**File > New Model Database > With Standard/Explicit Model**) and running it with **File > Run Script...**

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)

user_inputs = getInputs(fields = (('Name the model:', 'Truss Structure'),
                                   ('Length of truss members', '2'),
                                   ('Height of truss', '1.5')),
                        label = 'Please provide the following information',
                        dialogTitle = 'Model Parameters')

# If the user left the model name field blank we will need to give it a default
# name. This will also be the case if the user hits 'cancel' because then this
# field will have None
if user_inputs[0]:
    model_name = user_inputs[0]
else:
    print '!You did not type in a name for the model ' + \
          'Assuming name - Truss Structure '
    model_name = 'Truss Structure'

# If the user enters a character where a number (float or integer) is expected,
# the float() function will throw an error "ValueError: invalid literal for
# float(): xxxx". This will also be the case if the user hits 'cancel' because
# then this field will have 'None'
try:
    truss_member_length = float(user_inputs[1])
except:
    print '!You did not type in an integer or float. Assuming a length of 2'
    truss_member_length = 2

try:
    truss_height = float(user_inputs[2])
except:
    print '!You did not type in an integer or float. Assuming a height of 1.5'
    truss_height = 1.5

# -------------------------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName=model_name)
trussModel = mdb.models[model_name]

# -------------------------------------------------------------------------
# Create the part

import sketch
import part

trussSketch = trussModel.ConstrainedSketch(name='2D Truss Sketch', sheetSize=10.0)
trussSketch.Line(point1=(0, 0), point2=(truss_member_length, 0))
trussSketch.Line(point1=(truss_member_length, 0),
                 point2=(2*truss_member_length, 0))
trussSketch.Line(point1=(2*truss_member_length, 0),
                 point2=(3*truss_member_length, 0))
trussSketch.Line(point1=(0, -truss_height),
                 point2=(truss_member_length,-truss_height))
trussSketch.Line(point1=(truss_member_length, -truss_height),
                 point2=(2*truss_member_length,-truss_height))
trussSketch.Line(point1=(0, -truss_height),
                 point2=(truss_member_length, 0))
trussSketch.Line(point1=(truss_member_length, 0),
                 point2=(2*truss_member_length, -truss_height))
trussSketch.Line(point1=(2*truss_member_length, -truss_height),
                 point2=(3*truss_member_length, 0))
trussSketch.Line(point1=(truss_member_length, 0),
                 point2=(truss_member_length, -truss_height))
trussSketch.Line(point1=(2*truss_member_length, 0),
                 point2=(2*truss_member_length, -truss_height))

trussPart = trussModel.Part(name='Truss', dimensionality=TWO_D_PLANAR,
                            type=DEFORMABLE_BODY)
trussPart.BaseWire(sketch=trussSketch)

# -------------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs modulus
# and poissons ratio
```

```
trussMaterial = trussModel.Material(name='AISI 1005 Steel')
trussMaterial.Density(table=((7872, ),       ))
trussMaterial.Elastic(table=((200E9, 0.29), ))

# -------------------------------------------------------------------------
# Create a section and assign the truss to it
import section

# Set the radius to 0.5% of the length
truss_member_radius = 0.005*truss_member_length
truss_member_area = 3.14*(truss_member_radius**2)

trussSection = trussModel.TrussSection(name='Truss Section',
                               material='AISI 1005 Steel', area=truss_member_area)

edges_for_section_assignment = trussPart.edges \
 .findAt(((truss_member_length/2, 0.0, 0.0), ),
        ((truss_member_length + truss_member_length/2, 0.0, 0.0), ),
        ((2*truss_member_length + truss_member_length/2, 0.0, 0.0), ),
        ((truss_member_length/2, -truss_height, 0.0), ),
        ((truss_member_length + truss_member_length/2, -truss_height, 0.0), ),
        ((truss_member_length/2, -truss_height/2, 0.0), ),
        ((truss_member_length + truss_member_length/2, -truss_height/2, 0.0), ),
        ((2*truss_member_length + truss_member_length/2, -truss_height/2, 0.0), ),
        ((truss_member_length, -truss_height/2, 0.0), ),
        ((2*truss_member_length, -truss_height/2, 0.0), ))

truss_region = regionToolset.Region(edges=edges_for_section_assignment)
trussPart.SectionAssignment(region=truss_region, sectionName='Truss Section')

# -------------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
trussAssembly = trussModel.rootAssembly
trussInstance = trussAssembly.Instance(name='Truss Instance', part=trussPart,
                                                              dependent=ON)

# -------------------------------------------------------------------------
# Create sets

# create set for load point
vertex_coords_for_force = (2*truss_member_length, 0.0, 0.0)
vertex_for_force = trussInstance.vertices.findAt((vertex_coords_for_force,))
trussAssembly.Set(vertices=vertex_for_force, name='force point set')

# create set for end point
vertex_coords_for_end = trussInstance.vertices.findAt(((3*truss_member_length,
                                                                0.0, 0.0),))
trussAssembly.Set(vertices=vertex_coords_for_end, name='end point set')
```

```
# -----------------------------------------------------------------------
# Create the step

import step

# Create a dynamic explicit step
trussModel.ExplicitDynamicsStep(name='Loading Step', previous='Initial',
              description='Loads are applied to the truss for 0.01s in this step',
              timePeriod=0.01)

# -----------------------------------------------------------------------
# Create the history output request

force_point_region = trussAssembly.sets['force point set']
trussModel.historyOutputRequests.changeKey(fromName='H-Output-1',
                                           toName='Force point output')
trussModel.historyOutputRequests['Force point output'] \
                                        .setValues(variables=('UT',),
                                                   frequency=1,
                                                   region=force_point_region,
                                                   sectionPoints=DEFAULT,
                                                   rebar=EXCLUDE)

end_point_region = trussAssembly.sets['end point set']
trussModel.HistoryOutputRequest(name='End point output',
                                createStepName='Loading Step',
                                variables=('UT',), frequency=1,
                                region=end_point_region, sectionPoints=DEFAULT,
                                rebar=EXCLUDE)

# -----------------------------------------------------------------------
# Apply loads

# Ask user for magnitude of concentrated force
user_input_2 = getInput(prompt = 'Magnitude of concentrated force (in -Y
direction)', default = '6000')
try:
    force_input = float(user_input_2)
except:
    print '!You did not type in an integer or float. Assuming force ' + \
            'magnitude of 6000'
    force_input = 6000

# Apply concentrated force on second node

# We aleady have the vertex for force from the assembly step so we use that
trussModel.ConcentratedForce(name='ForcePulse', createStepName='Loading Step',
                             region=(vertex_for_force,), cf2=-force_input,
                             distributionType=UNIFORM, field='', localCsys=None)

# -----------------------------------------------------------------------
# Apply boundary conditions

# Pin left end of upper beam
vertex_coords_for_first_pin = (0.0, 0.0, 0.0)
vertex_for_first_pin = trussInstance.vertices \
                                        .findAt((vertex_coords_for_first_pin,))
trussModel.DisplacementBC(name='Pin1', createStepName='Initial',
                          region=(vertex_for_first_pin,),
                          u1=SET, u2=SET, ur3=UNSET,
                          amplitude=UNSET, distributionType=UNIFORM)

# Pin left end of lower beam
vertex_coords_for_second_pin = (0.0, -truss_height, 0.0)
vertex_for_second_pin = trussInstance.vertices \
                                        .findAt((vertex_coords_for_second_pin,))
trussModel.DisplacementBC(name='Pin2', createStepName='Initial',
                          region=(vertex_for_second_pin,),
                          u1=SET, u2=SET, ur3=UNSET,
                          amplitude=UNSET, distributionType=UNIFORM)

# -----------------------------------------------------------------------
# Create the mesh

import mesh

truss_mesh_region = truss_region
edges_for_meshing = edges_for_section_assignment

mesh_element_type=mesh.ElemType(elemCode=T2D2, elemLibrary=STANDARD)
trussPart.setElementType(regions=truss_mesh_region,
                         elemTypes=(mesh_element_type, ))
trussPart.seedEdgeByNumber(edges=edges_for_meshing, number=1)
trussPart.generateMesh()

# -----------------------------------------------------------------------
# Create and run the job

import job

# Create the job
mdb.Job(name='TrussExplicitParameterizedJob', model=model_name, type=ANALYSIS,
        explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE,
        description='Analysis of truss under a pulse load',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)


# Run the job
mdb.jobs['TrussExplicitParameterizedJob'].submit(consistencyChecking=OFF)
```

```
# Do not return control till job is finished running
mdb.jobs['TrussExplicitParameterizedJob'].waitForCompletion()

# End of run job


# ==========================================================================
# --------------------------------------------------------------------------
# Post processing
# --------------------------------------------------------------------------
# ==========================================================================


import odbAccess
import visualization

truss_Odb_Path = 'TrussExplicitParameterizedJob.odb'
odb_object = session.openOdb(name=truss_Odb_Path)

session.viewports['Viewport: 1'].setValues(displayedObject=odb_object)
session.viewports['Viewport: 1'].odbDisplay.display \
                                               .setValues(plotState=(DEFORMED, ))

#-------------------------------------------------------------------------
# Plot the deformed state of the truss

truss_deformed_viewport = session.Viewport(name='Truss in Deformed State')
truss_deformed_viewport.setValues(displayedObject=odb_object)
truss_deformed_viewport.odbDisplay.display.setValues(plotState=(UNDEFORMED,
                                                                 DEFORMED, ))
truss_deformed_viewport.odbDisplay.commonOptions.setValues(nodeLabels=ON)
truss_deformed_viewport.odbDisplay.commonOptions.setValues(elemLabels=ON)
truss_deformed_viewport.setValues(origin=(0.0, 0.0), width=250, height=160)

# -----------------------------------------------------------------------
# Make XY plots of U2 displacement for force point and end point

# We need to find the variable names for the history variables
# Abaqus tends to give them names like "Spatial displacement: U2 at Node 2 in
# NSET FORCE POINT SET"
# So basically we will search for variables with the letters 'U2' in them
# and save the variable names in an array called theoutputvariablename to use later

keyarray=session.odbData['TrussExplicitParameterizedJob.odb'] \
                                                     .historyVariables.keys()

theoutputvariablename=[]
for x in keyarray:
    if (x.find('U2')>-1):
        theoutputvariablename.append(x)

# ------------------------------------------
# a) XY plot and data output of U2 displacement for force point

xydata_force_pt=session.XYDataFromHistory(name = 'Data for force point',
                                          odb=odb_object,
                                          outputVariableName=theoutputvariablename[0],
                                          steps=('Loading Step', ), )
curve_force_pt = session.Curve(xyData=xydata_force_pt)

# Before plotting we make sure the name 'Plot of forcepoint' is not already in
# use, and delete it if it is, because Abaqus does not allow overwriting of plots
if 'Plot of forcepoint' in session.xyPlots.keys():
    del session.xyPlots['Plot of forcepoint']

xyplot_force_pt = session.XYPlot('Plot of forcepoint')
chartName = xyplot_force_pt.charts.keys()[0]
chart = xyplot_force_pt.charts[chartName]
chart.setValues(curvesToPlot=(curve_force_pt, ), )
xyplot_force_pt_viewport = session \
                           .Viewport(name='Displacement U2 plot of force point')
xyplot_force_pt_viewport.setValues(displayedObject=xyplot_force_pt)

# Output the XY data as a .txt file
xydataobject_force_point = session.xyDataObjects['Data for force point']
session.xyReportOptions.setValues(totals=ON, minMax=ON)
session.writeXYReport(fileName='force_point_xydata_output.txt',
                      xyData=(xydataobject_force_point, ),
                      appendMode=OFF)


# ---------------------------------------
# b) XY plot of U2 displacement for end point

xydata_end_pt=session.XYDataFromHistory(name = 'Data for end point',
                                        odb=odb_object,
                                        outputVariableName=theoutputvariablename[1],
                                        steps=('Loading Step', ), )
curve_end_pt = session.Curve(xyData=xydata_end_pt)

# Before plotting we make sure the name 'Plot of endpoint' is not already in use,
# and delete it if it is, because Abaqus does not allow overwriting of plots
if 'Plot of endpoint' in session.xyPlots.keys():
    del session.xyPlots['Plot of endpoint']

xyplot_end_pt = session.XYPlot('Plot of endpoint')
chartName = xyplot_end_pt.charts.keys()[0]
chart = xyplot_end_pt.charts[chartName]
chart.setValues(curvesToPlot=(curve_end_pt, ), )
xyplot_end_pt_viewport = session.Viewport(name='Displacement U2 plot of end point')
xyplot_end_pt_viewport.setValues(displayedObject=xyplot_end_pt)

# Output the XY data as a .txt file
xydataobject_end_point = session.xyDataObjects['Data for end point']
```

```
session.xyReportOptions.setValues(totals=ON, minMax=ON)
session.writeXYReport(fileName='end_point_xydata_output.txt',
                      xyData=(xydataobject_end_point, ), appendMode=OFF)


# -----------------------------------------------------------------------
# Create a combined XY plot of the U2 displacement for force point and end point

xy_data_1 = session.xyDataObjects['Data for force point']
curve_1 = session.Curve(xyData=xy_data_1)
xy_data_2 = session.xyDataObjects['Data for end point']
curve_2 = session.Curve(xyData=xy_data_2)

# Before plotting we make sure the name 'Combined plot' is not already in use, and
# delete it if it is, because Abaqus does not allow overwriting of plots
if 'Combined Plot' in session.xyPlots.keys():
       del session.xyPlots['Combined Plot']

combined_plot = session.XYPlot('Combined Plot')
combined_plot_chart_name = combined_plot.charts.keys()[0]
combined_plot_chart = combined_plot.charts[combined_plot_chart_name]
combined_plot_chart.setValues(curvesToPlot=(curve_1, curve_2, ), )

combined_plot_viewport = session.Viewport(name='Combined Plot Viewport')
combined_plot_viewport.setValues(displayedObject=combined_plot)

# -----------------------------------------------------------------------
# Modify the chart using the chart options

# Plot major and minor vertical gridlines
combined_plot_chart.majorAxis1GridStyle.setValues(show=True)
combined_plot_chart.majorAxis1GridStyle.setValues(color='#FF0000')
combined_plot_chart.majorAxis1GridStyle.setValues(style=DASHED)
combined_plot_chart.majorAxis1GridStyle.setValues(thickness=0.5)
combined_plot_chart.minorAxis1GridStyle.setValues(show=True)
combined_plot_chart.minorAxis1GridStyle.setValues(color='#00FFFF')
combined_plot_chart.minorAxis1GridStyle.setValues(style=DOTTED)
combined_plot_chart.minorAxis1GridStyle.setValues(thickness=0.2)

# Plot major and minor horizontal gridlines
combined_plot_chart.majorAxis2GridStyle.setValues(show=True)
combined_plot_chart.majorAxis2GridStyle.setValues(color='#FF0000')
combined_plot_chart.majorAxis2GridStyle.setValues(style=DASHED)
combined_plot_chart.majorAxis2GridStyle.setValues(thickness=0.5)
combined_plot_chart.minorAxis2GridStyle.setValues(show=True)
combined_plot_chart.minorAxis2GridStyle.setValues(color='#00FFFF')
combined_plot_chart.minorAxis2GridStyle.setValues(style=DOTTED)
combined_plot_chart.minorAxis2GridStyle.setValues(thickness=0.2)

# Add a border to the grid
combined_plot_chart.gridArea.border.setValues(show=True)
```

```
# Change the grid display color
combined_plot_chart.gridArea.style.setValues(color='#FFFC95')

# Set grid area size to square (set the aspect ratio to 1)
session.charts['Chart-3'].setValues(aspectRatio=1.0)

# Set the grid position to auto-align with an alighnment of 'center'
combined_plot_chart.gridArea.setValues(positionMethod=AUTOMATIC, alignment=CENTER)

# ---------------------------------------------------------------------------
# Modify the axes of the chart using the axis options

# Set the X-axis to linear (as opposed to log scale)
combined_plot_chart.axes1[0].axisData.setValues(scale=LINEAR)

# Set the Y-axis to linear (as opposed to log scale)
combined_plot_chart.axes2[0].axisData.setValues(scale=LINEAR)

#set the scale of the X axis
combined_plot_chart.axes1[0].axisData.setValues(maxValue=0.01,
                                                maxAutoCompute=False)

combined_plot_chart.axes1[0].axisData.setValues(minValue=0, minAutoCompute=False)

#set the scale of the Y axis
combined_plot_chart.axes2[0].axisData.setValues(maxAutoCompute=True)
combined_plot_chart.axes2[0].axisData.setValues(minAutoCompute=True)

# Set the frequency at which X-axis values are displayed
combined_plot_chart.axes1[0].axisData.setValues(tickMode=INCREMENT)

# Set the frequency at which Y-axis values are displayed
combined_plot_chart.axes2[0].axisData.setValues(tickMode=AUTOCOMPUTE)

# Set the frequency at which minor vertical gridlines are displayed
combined_plot_chart.axes1[0].axisData.setValues(minorTickCount=5)

# Set the frequency at which minor horizontal gridlines are displayed
combined_plot_chart.axes2[0].axisData.setValues(minorTickCount=1)

# Display the X-axis tickmarks on the inner side of the grid
combined_plot_chart.axes1[0].setValues(tickPlacement=INSIDE)

# Display the Y-axis tickmarks on the inner side of the grid
combined_plot_chart.axes2[0].setValues(tickPlacement=OUTSIDE)

# Set the frequency and style of tick marks displayed on the X-axis
combined_plot_chart.axes1[0].setValues(tickLength=4)
combined_plot_chart.axes1[0].tickStyle.setValues(style=SOLID)
combined_plot_chart.axes1[0].tickStyle.setValues(thickness=1.2)
combined_plot_chart.axes1[0].tickStyle.setValues(color='#000000')
```

```
# Set the frequency and style of tick marks displayed on the Y-axis
combined_plot_chart.axes2[0].setValues(tickLength=2)
combined_plot_chart.axes2[0].tickStyle.setValues(style=SOLID)
combined_plot_chart.axes2[0].tickStyle.setValues(thickness=1.2)
combined_plot_chart.axes2[0].tickStyle.setValues(color='#000000')

# Let Abaqus/CAE assign the default X-axis title
combined_plot_chart.axes1[0].axisData.setValues(useSystemTitle=True)

# Set the Y-axis title
combined_plot_chart.axes2[0].axisData.setValues(useSystemTitle=False,
                                                 title='Displacement of node')

# Set the font style and color of the X-axis title
combined_plot_chart.axes1[0].titleStyle \
               .setValues(font='-*-arial-medium-r-normal-*-*-180-*-*-p-*-*-*')
combined_plot_chart.axes1[0].titleStyle.setValues(color='#000000')

# Set the font style and color of the Y-axis title
combined_plot_chart.axes2[0].titleStyle \
               .setValues(font='-*-arial-medium-r-normal-*-*-180-*-*-p-*-*-*')
combined_plot_chart.axes2[0].titleStyle.setValues(color='#000000')

# Set the placement of the X-axis (top, bottom or center of chart ?)
combined_plot_chart.axes1[0].setValues(placement=MAX_EDGE)

# Let Abaqus CAE decide the placement of the Y-axis
combined_plot_chart.axes2[0].setValues(placement=MIN_MAX_EDGE)

# Set the format of the X-axis labels to decimal with precision of 3 sig figs
combined_plot_chart.axes1[0].axisData.setValues(labelFormat=DECIMAL)
combined_plot_chart.axes1[0].axisData.setValues(labelNumDigits=3)

# Set the format of the Y-axis labels to decimal with precision of 3 sig figs
combined_plot_chart.axes2[0].axisData.setValues(labelFormat=AUTOMATIC)
combined_plot_chart.axes2[0].axisData.setValues(labelNumDigits=2)

# Set how frequently labels are displayed on the X-axis (1 = every major gridline,
# 2 = every second major gridline etc)
combined_plot_chart.axes1[0].setValues(labelFrequency=1)

# Set how frequently labels are displayed on the Y-axis (1 = every major gridline,
# 2 = every second major gridline etc)
combined_plot_chart.axes2[0].setValues(labelFrequency=1)

# Set the font style and color of the X-axis labels
combined_plot_chart.axes1[0].labelStyle \
               .setValues(font='-*-verdana-medium-r-normal-*-*-100-*-*-p-*-*-*')
combined_plot_chart.axes1[0].labelStyle.setValues(color='#000000')

# Set the font style and color of the Y-axis labels
combined_plot_chart.axes2[0].labelStyle \
                .setValues(font='-*-verdana-medium-r-normal-*-*-100-*-*-p-*-*-*')
combined_plot_chart.axes2[0].labelStyle.setValues(color='#000000')

# Set the style and color of the X-axis line
combined_plot_chart.axes1[0].lineStyle.setValues(style=SOLID)
combined_plot_chart.axes1[0].lineStyle.setValues(thickness=0.2)

# Set the style and color of the Y-axis line
combined_plot_chart.axes2[0].lineStyle.setValues(style=SOLID)
combined_plot_chart.axes2[0].lineStyle.setValues(thickness=0.2)

# -----------------------------------------------------------------------
# Create/Modify the chart title using the plot title options

# Set the plot title
combined_plot.title.setValues(text='Displacement Vs Time')

# Set the plot title font style and color
combined_plot.title.style \
                .setValues(font='-*-arial-medium-r-normal-*-*-240-*-*-p-*-*-*')
combined_plot.title.style.setValues(color='#008000')

# Do not inset the plot title
combined_plot.title.area.setValues(inset=False)

# Auto-align the plot title at the top-center
combined_plot.title.area.setValues(positionMethod=AUTOMATIC, alignment=TOP_CENTER)

# Give the plot title a border
combined_plot.title.area.border.setValues(show=True)

# Set the color of the plot title border
combined_plot.title.area.border.setValues(color='#000000')


# -----------------------------------------------------------------------
# Modify the plot legend using the chart legend options

# Display the legend
combined_plot_chart.legend.setValues(show=True)

# Give the legend a title
combined_plot_chart.legend.setValues(title='Legend:')

combined_plot_chart.legend.textStyle \
                .setValues(font='-*-verdana-medium-r-normal-*-*-120-*-*-p-*-*-*')
combined_plot_chart.legend.titleStyle \
                .setValues(font='-*-verdana-medium-r-normal-*-*-120-*-*-p-*-*-*')

combined_plot_chart.legend.textStyle.setValues(color='#FF0000')
combined_plot_chart.legend.titleStyle.setValues(color='#FF0000')
```

```
# Display minimum and maximum values on the legend
combined_plot_chart.legend.setValues(showMinMax=True)

# Set the format to decimal with precision of 3 sig figs
combined_plot_chart.legend.setValues(numberFormat=DECIMAL)
combined_plot_chart.legend.setValues(numDigits=3)

# Set the position to inset so the legend can be displayed over the grid
combined_plot_chart.legend.area.setValues(inset=True)

# Choose manual positioning, and position the legend at desired location
combined_plot_chart.legend.area.setValues(positionMethod=MANUAL)
combined_plot_chart.legend.area.setValues(originOffset=(0.2, 0.1))

# Give the legend a border
combined_plot_chart.legend.area.border.setValues(show=True)

# Set the color of the legend border
combined_plot_chart.legend.area.border.setValues(color='#0000FF')

# Fill the legend with a white background so the grid is not visible through it
combined_plot_chart.legend.area.style.setValues(fill=True)
combined_plot_chart.legend.area.style.setValues(color='#FFFFFF')

# Resize the viewport window
combined_plot_viewport.setValues(width=250.0, height=200.0)


# ----------------------------------------------------------------------
# Change the plot curve line style, color, thickness etc using the XY Curve options

# Use the name assigned to the curve object as the name of the curve in the legend
curve_1.setValues(useDefault=True, legendSource=CURVE_NAME)

# Create a new name for the name of the curve in the legend
curve_2.setValues(useDefault=False, legendLabel='Displacement of the end point')

# Set the line style and thickness for the force point curve
curve_1.lineStyle.setValues(show=True)
curve_1.lineStyle.setValues(style=DASHED)
curve_1.lineStyle.setValues(color='#0099FF')
curve_1.lineStyle.setValues(thickness=0.8)

# Set the line style and thickness for the end point curve
curve_2.lineStyle.setValues(color='#000080')
curve_2.lineStyle.setValues(style=SOLID)
curve_2.lineStyle.setValues(thickness=0.5)

# Show symbols on the end point curve
curve_2.symbolStyle.setValues(show=True)
curve_2.symbolStyle.setValues(color='#000080')
curve_2.symbolStyle.setValues(marker=FILLED_CIRCLE)
```

```
curve_2.symbolStyle.setValues(size=1.6)

# ----------------------------------------------------------------------
# Save the plot by printing it to a png file

# Set the size of the image
session.pngOptions.setValues(imageSize=SIZE_ON_SCREEN)

# Opt not to reduce the image to 256 colors
session.printOptions.setValues(reduceColors=False)

# Store the name and path you wish to give the file in a variable
# If you only specify a name and no path, it will be saved in the Abaqus Work
directory
png_name_and_path = 'combined_plot_image'

# Print the image to a file
# Note that if the file already exists it will be replaced
session.printToFile(fileName=png_name_and_path, format=PNG,
                    canvasObjects=(combined_plot_viewport, ))
```

14.4 Examining the Script

Let's understand how this script works. A large portion of the script is copied from the dynamic explicit truss analysis script of Chapter 8 so we'll only discuss the new parts.

14.4.1 Accept inputs

The following block accepts inputs from the user with the help of prompt boxes

```
user_inputs = getInputs(fields = (('Name the model:', 'Truss Structure'),
                                  ('Length of truss members', '2'),
                                  ('Height of truss', '1.5')),
                        label = 'Please provide the following information',
                        dialogTitle = 'Model Parameters')

# If the user left the model name field blank we will need to give it a default
# name. This will also be the case if the user hits 'cancel' because then this
# field will have None
if user_inputs[0]:
    model_name = user_inputs[0]
else:
    print '!You did not type in a name for the model ' + \
          'Assuming name - Truss Structure '
    model_name = 'Truss Structure'

# If the user enters a character where a number (float or integer) is expected,
# the float() function will throw an error "ValueError: invalid literal for
# float(): xxxx". This will also be the case if the user hits 'cancel' because
# then this field will have 'None'
try:
    truss_member_length = float(user_inputs[1])
except:
    print '!You did not type in an integer or float. Assuming a length of 2'
    truss_member_length = 2

try:
    truss_height = float(user_inputs[2])
except:
    print '!You did not type in an integer or float. Assuming a height of 1.5'
    truss_height = 1.5
```

```
user_inputs = getInputs(fields = (('Name the model:', 'Truss Structure'),
                                  ('Length of truss members', '2'),
                                  ('Height of truss', '1.5')),
                        label = 'Please provide the following information',
                        dialogTitle = 'Model Parameters')
```

The **getInputs()** method is used to obtain multiple inputs from a user. It displays a modal dialog box with a column of labels and text input fields as well as an 'OK' button and a 'Cancel' button. The required argument **fields** is a sequence of sequence of Strings. Each sequence is a pair of Strings specifying the label that appears next to the text field and a default value that appears in the text field. If you don't want a default value to appear in the text field this second String must be an empty String. The option arguments are **label** and **dialogTitle**. **dialogTitle** as the name suggests is a String specifying the text in the title bar of the dialog box. If you don't provide this argument the dialog box will be titled "Get Inputs". **label** is a String specifying a label to be placed at the top of the dialog box above column of labels and text fields. By default this is an empty String so no label is displayed.

Since we have a single variable on the left hand side of the statement, **user_inputs** will be a list. We can then refer to the model name, member length and truss height as **user_inputs[0]**, **user_inputs[1]** and **user_inputs[2]**. We could instead have written this statement as:

```
[model_name, truss_member_length] =
         getInputs(fields = (('Name the model:', 'Truss Structure'),
                             ('Length of truss members', '2'),
                             ('Height of truss', '1.5')),
                   label = 'Please provide the following information',
                    dialogTitle = 'Model Parameters'
```

Note that **getInputs()** cannot be used if you run the script from the command like with a command line option such as **noGUI** because the GUI must be present to display the prompt box.

```
if user_inputs[0]:
    model_name = user_inputs[0]
else:
    print '!You did not type in a name for the model ' + \
            'Assuming name - Truss Structure '
    model_name = 'Truss Structure'
```

We need to make sure the user actually entered a String to name the model. If the user leaves the field blank, **getInputs()** will return '' (an empty String) for that field. Also if the user clicks **Cancel** in the prompt, **getInputs()** will return a **None** object for this field (and for the other two as well). If we try to use this empty string '' or **Null** object as our model name we will get an error. The **if** statement helps us catch this. For those of you without much programming experience it might look like there is no condition in our **if** statement. You would expect to see

```
if user_inputs[0] != '' and user_inputs[0] != None:
```

in which either of the two conditions (user_inputs[0] != '' and user_inputs[0] = None) evaluate to TRUE. So basically you end up with

```
if (TRUE):
```

What's important is not the fact that you used a condition, it's the fact that it resulted in a TRUE. In Python, as in most programming languages, any non-zero value or value that is not **0**, **None,** or an empty string '' is TRUE, and a **0**, **None** or '' is FALSE. Therefore if **user_inputs** is not an empty string or a Null object, the statement becomes

```
If(TRUE):
```

If this is confusing for you, feel free to write the entire condition.

The model name will be assigned the String entered by the user if the condition is true. And if it happens to be a **None**, then the default of 'Truss Structure' will be the model name.

```
try:
    truss_member_length = float(user_inputs[1])
except:
    print '!You did not type in an integer or float. Assuming a length of 2'
    truss_member_length = 2
```

Everything the user types into the text boxes is stored as a String. For example if the user types 4.2 for the height, **user_inputs[2]** is the String '4.2'. We use **foat()** to convert it to a float.

However if the user typed in letters of the alphabet or some special characters instead of numbers, the **float()** method will throw an exception. If we catch this exception we will know immediately that the user has not entered a number so we use a **try** block to catch this. This is not the first time you are seeing a try block, but in case you've forgotten, if a statement inside a **try** block throws an exception, the program does not terminate. Instead it executes the code inside the **except** block. Our **except** block prints out a message in the message area (it will be in red because we preceded it with an exclamation point), and assigns the truss members a default length.

```
try:
    truss_height = float(user_inputs[2])
except:
    print '!You did not type in an integer or float. Assuming a height of 1.5'
    truss_height = 1.5
```

The procedure is repeated for the truss height.

14.4.2 Create the model

The following block creates the model.

```
# -----------------------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName=model_name)
trussModel = mdb.models[model_name]
```

Here we have modified the code that creates the model to use the String value stored in the variable **model_name** as the name of our model.

14.4.3 Create part

The following block creates the part.

```
# -----------------------------------------------------------------------
# Create the part
```

```
import sketch
import part

trussSketch = trussModel.ConstrainedSketch(name='2D Truss Sketch', sheetSize=10.0)
trussSketch.Line(point1=(0, 0), point2=(truss_member_length, 0))
trussSketch.Line(point1=(truss_member_length, 0),
                 point2=(2*truss_member_length, 0))
trussSketch.Line(point1=(2*truss_member_length, 0),
                 point2=(3*truss_member_length, 0))
trussSketch.Line(point1=(0, -truss_height),
                 point2=(truss_member_length,-truss_height))
trussSketch.Line(point1=(truss_member_length, -truss_height),
                 point2=(2*truss_member_length,-truss_height))
trussSketch.Line(point1=(0, -truss_height),
                 point2=(truss_member_length, 0))
trussSketch.Line(point1=(truss_member_length, 0),
                 point2=(2*truss_member_length, -truss_height))
trussSketch.Line(point1=(2*truss_member_length, -truss_height),
                 point2=(3*truss_member_length, 0))
trussSketch.Line(point1=(truss_member_length, 0),
                 point2=(truss_member_length, -truss_height))
trussSketch.Line(point1=(2*truss_member_length, 0),
                 point2=(2*truss_member_length, -truss_height))

trussPart = trussModel.Part(name='Truss', dimensionality=TWO_D_PLANAR,
                            type=DEFORMABLE_BODY)
trussPart.BaseWire(sketch=trussSketch)
```

All of these statements have been suitable modified so that the variable values accepted by the **getInputs()** prompt box now form the dimensions of the truss. This is a good example of parameterization. Whatever dimensions the user enters in the prompt box are used to build the truss accordingly.

14.4.4 Create a section

The following block creates the section.

```
# -----------------------------------------------------------------------
# Create a section and assign the truss to it
import section

# Set the radius to 0.5% of the length
truss_member_radius = 0.005*truss_member_length
truss_member_area = 3.14*(truss_member_radius**2)

trussSection = trussModel.TrussSection(name='Truss Section',
                                material='AISI 1005 Steel', area=truss_member_area)

edges_for_section_assignment = trussPart.edges \
 .findAt(((truss_member_length/2, 0.0, 0.0), ),
        ((truss_member_length + truss_member_length/2, 0.0, 0.0), ),
        ((2*truss_member_length + truss_member_length/2, 0.0, 0.0), ),
        ((truss_member_length/2, -truss_height, 0.0), ),
        ((truss_member_length + truss_member_length/2, -truss_height, 0.0), ),
        ((truss_member_length/2, -truss_height/2, 0.0), ),
        ((truss_member_length + truss_member_length/2, -truss_height/2, 0.0), ),
        ((2*truss_member_length + truss_member_length/2, -truss_height/2, 0.0), ),
        ((truss_member_length, -truss_height/2, 0.0), ),
        ((2*truss_member_length, -truss_height/2, 0.0), ))

truss_region = regionToolset.Region(edges=edges_for_section_assignment)
trussPart.SectionAssignment(region=truss_region, sectionName='Truss Section')
```

Here you once again see the section creation and assignment block has been parameterized and uses variable names instead of hard coded values. All the arguments of the **edges.findAt()** method have been parameterized since the locations of the truss members depend on the parameters used when creating the truss.

14.4.5 Create sets

The following block creates sets for later use.

```
# -------------------------------------------------------------------------
# Create sets

# create set for load point
vertex_coords_for_force = (2*truss_member_length, 0.0, 0.0)
vertex_for_force = trussInstance.vertices.findAt((vertex_coords_for_force,))
trussAssembly.Set(vertices=vertex_for_force, name='force point set')

# create set for end point
vertex_coords_for_end = trussInstance.vertices.findAt(((3*truss_member_length,
                                                          0.0, 0.0),))
trussAssembly.Set(vertices=vertex_coords_for_end, name='end point set')
```

You see that parameterization required the modification of these statements as well. The locations of the vertices depend on the parameters of the truss.

14.4.6 Request and use load magnitude

The following block creates the loads.

```
# -------------------------------------------------------------------------
# Apply loads

# Ask user for magnitude of concentrated force
```

```
user_input_2 = getInput(prompt = 'Magnitude of concentrated force (in -Y
direction)', default = '6000')
try:
    force_input = float(user_input_2)
except:
    print '!You did not type in an integer or float. Assuming force ' + \
            'magnitude of 6000'
    force_input = 6000

# Apply concentrated force on second node

# We aleady have the vertex for force from the assembly step so we use that
trussModel.ConcentratedForce(name='ForcePulse', createStepName='Loading Step',
                             region=(vertex_for_force,), cf2=-force_input,
                             distributionType=UNIFORM, field='', localCsys=None)
```

The **getInput()** method is used to prompt the user for the magnitude of the concentrated force. **getInput()** serves a purpose similar to **getInputs()**. The difference is that it obtains a single input from the user in the dialog box whereas **getInputs()** obtains multiple inputs.

getInput() has one required argument **prompt** which is the String to be displayed next to the text field. It has one optional argument **default** which is the default String to place in the text field.

We follow **getInput()** with a try block similar to the ones we used after **getInputs()**.

The variable **force_input** is then used as an argument for **ConcentratedForce()**.

14.4.7 Boundary conditions

The following block creates the boundary conditions.

```
# -----------------------------------------------------------------------
# Apply boundary conditions

# Pin left end of upper beam
vertex_coords_for_first_pin = (0.0, 0.0, 0.0)
vertex_for_first_pin = trussInstance.vertices \
                                            .findAt((vertex_coords_for_first_pin,))
trussModel.DisplacementBC(name='Pin1', createStepName='Initial',
                          region=(vertex_for_first_pin,),
                          u1=SET, u2=SET, ur3=UNSET,
                          amplitude=UNSET, distributionType=UNIFORM)

# Pin left end of lower beam
```

```
vertex_coords_for_second_pin = (0.0, -truss_height, 0.0)
vertex_for_second_pin = trussInstance.vertices \
                                            .findAt((vertex_coords_for_second_pin,))
trussModel.DisplacementBC(name='Pin2', createStepName='Initial',
                          region=(vertex_for_second_pin,),
                          u1=SET, u2=SET, ur3=UNSET,
                          amplitude=UNSET, distributionType=UNIFORM)
```

This block now refers to the truss dimension variables instead of hard coded values.

14.4.8 Initial post processing

The following begins the post processing.

```
import odbAccess
import visualization

                        ...
                        ...

#-----------------------------------------------------------------------
# Plot the deformed state of the truss

                        ...
                        ...

# ---------------------------------------------------------------------
# Make XY plots of U2 displacement for force point and end point

                        ...
                        ...

# ------------------------------------------
# a) XY plot and data output of U2 displacement for force point

                        ...
                        ...

# ------------------------------------------
# b) XY plot of U2 displacement for end point

                        ...
                        ...
```

The statements that plot the deformed state of the truss, and XY plots of U2 displacement for the point of application of force and the end point of the truss, are left unchanged.

14.4.9 Combined XY plot

The following creates a combined U2 displacement plot for the two sets.

```
# -------------------------------------------------------------------------
# Create a combined XY plot of the U2 displacement for force point and end point

xy_data_1 = session.xyDataObjects['Data for force point']
curve_1 = session.Curve(xyData=xy_data_1)
xy_data_2 = session.xyDataObjects['Data for end point']
curve_2 = session.Curve(xyData=xy_data_2)

# Before plotting we make sure the name 'Combined plot' is not already in use, and
# delete it if it is, because Abaqus does not allow overwriting of plots
if 'Combined Plot' in session.xyPlots.keys():
       del session.xyPlots['Combined Plot']

combined_plot = session.XYPlot('Combined Plot')
combined_plot_chart_name = combined_plot.charts.keys()[0]
combined_plot_chart = combined_plot.charts[combined_plot_chart_name]
combined_plot_chart.setValues(curvesToPlot=(curve_1, curve_2, ), )

combined_plot_viewport = session.Viewport(name='Combined Plot Viewport')
combined_plot_viewport.setValues(displayedObject=combined_plot)
```

Here a combined XY plot is created. Notice that the statements are similar to those for plotting a single XY plot. The difference is that that in the **(chart).setValues()** method the **curvesToPlot** argument is set to two curves instead of one.

14.4.10 Chart Options

The following statements use the chart options to modify the chart.

```
# -------------------------------------------------------------------------
# Modify the chart using the chart options

# Plot major and minor vertical gridlines
combined_plot_chart.majorAxis1GridStyle.setValues(show=True)
combined_plot_chart.majorAxis1GridStyle.setValues(color='#FF0000')
combined_plot_chart.majorAxis1GridStyle.setValues(style=DASHED)
combined_plot_chart.majorAxis1GridStyle.setValues(thickness=0.5)
combined_plot_chart.minorAxis1GridStyle.setValues(show=True)
combined_plot_chart.minorAxis1GridStyle.setValues(color='#00FFFF')
combined_plot_chart.minorAxis1GridStyle.setValues(style=DOTTED)
combined_plot_chart.minorAxis1GridStyle.setValues(thickness=0.2)

# Plot major and minor horizontal gridlines
combined_plot_chart.majorAxis2GridStyle.setValues(show=True)
combined_plot_chart.majorAxis2GridStyle.setValues(color='#FF0000')
combined_plot_chart.majorAxis2GridStyle.setValues(style=DASHED)
combined_plot_chart.majorAxis2GridStyle.setValues(thickness=0.5)
combined_plot_chart.minorAxis2GridStyle.setValues(show=True)
combined_plot_chart.minorAxis2GridStyle.setValues(color='#00FFFF')
combined_plot_chart.minorAxis2GridStyle.setValues(style=DOTTED)
combined_plot_chart.minorAxis2GridStyle.setValues(thickness=0.2)

# Add a border to the grid
combined_plot_chart.gridArea.border.setValues(show=True)

# Change the grid display color
combined_plot_chart.gridArea.style.setValues(color='#FFFC95')

# Set grid area size to square (set the aspect ratio to 1)
session.charts['Chart-3'].setValues(aspectRatio=1.0)

# Set the grid position to auto-align with an alighnment of 'center'
combined_plot_chart.gridArea.setValues(positionMethod=AUTOMATIC, alignment=CENTER)
```

```
combined_plot_chart.majorAxis1GridStyle.setValues(show=True)
combined_plot_chart.majorAxis1GridStyle.setValues(color='#FF0000')
combined_plot_chart.majorAxis1GridStyle.setValues(style=DASHED)
combined_plot_chart.majorAxis1GridStyle.setValues(thickness=0.5)
```

XYPlots have 4 **LineStyle** objects which are used to define the line style to be used for drawing XY-Plot objects. These are **majorAxis1GridStyle**, **majorAxis2GridStyle**, **minorAxis1GridStyle**, **minorAxis2GridStyle**. All of them have a **setValues()** method which is used here. This **setValues()** method has 4 optional arguments. **color** is a String specifying the color to be used. **show** is a Boolean specifying whether the line should be shown. **style** is a SymbolicConstant specifying the line style with possible values of **SOLID**, **DASHED**, **DOTTED**, and **DOT_DASH**. **thickness** is a Float specifying the line thickness in mm to be used when drawing the lines.

The 4 statements could have been combined into one statement with all the arguments of **setValues()** provided together.

```
# Add a border to the grid
combined_plot_chart.gridArea.border.setValues(show=True)

# Change the grid display color
combined_plot_chart.gridArea.style.setValues(color='#FFFC95')

# Set grid area size to square (set the aspect ratio to 1)
session.charts['Chart-3'].setValues(aspectRatio=1.0)

# Set the grid position to auto-align with an alighnment of 'center'
combined_plot_chart.gridArea.setValues(positionMethod=AUTOMATIC, alignment=CENTER)
```

The comments interspersed with the code explain the purpose of each statement.

14.4.11 Axis Options

The following statements use the axis options to modify the chart axes.

```
# ---------------------------------------------------------------------------
# Modify the axes of the chart using the axis options

# Set the X-axis to linear (as opposed to log scale)
combined_plot_chart.axes1[0].axisData.setValues(scale=LINEAR)

# Set the Y-axis to linear (as opposed to log scale)
combined_plot_chart.axes2[0].axisData.setValues(scale=LINEAR)

#set the scale of the X axis
combined_plot_chart.axes1[0].axisData.setValues(maxValue=0.01,
                                                maxAutoCompute=False)

combined_plot_chart.axes1[0].axisData.setValues(minValue=0, minAutoCompute=False)

#set the scale of the Y axis
combined_plot_chart.axes2[0].axisData.setValues(maxAutoCompute=True)
combined_plot_chart.axes2[0].axisData.setValues(minAutoCompute=True)

# Set the frequency at which X-axis values are displayed
combined_plot_chart.axes1[0].axisData.setValues(tickMode=INCREMENT)

# Set the frequency at which Y-axis values are displayed
combined_plot_chart.axes2[0].axisData.setValues(tickMode=AUTOCOMPUTE)

# Set the frequency at which minor vertical gridlines are displayed
combined_plot_chart.axes1[0].axisData.setValues(minorTickCount=5)

# Set the frequency at which minor horizontal gridlines are displayed
combined_plot_chart.axes2[0].axisData.setValues(minorTickCount=1)

# Display the X-axis tickmarks on the inner side of the grid
combined_plot_chart.axes1[0].setValues(tickPlacement=INSIDE)

# Display the Y-axis tickmarks on the inner side of the grid
combined_plot_chart.axes2[0].setValues(tickPlacement=OUTSIDE)

# Set the frequency and style of tick marks displayed on the X-axis
combined_plot_chart.axes1[0].setValues(tickLength=4)
combined_plot_chart.axes1[0].tickStyle.setValues(style=SOLID)
combined_plot_chart.axes1[0].tickStyle.setValues(thickness=1.2)
combined_plot_chart.axes1[0].tickStyle.setValues(color='#000000')

# Set the frequency and style of tick marks displayed on the Y-axis
```

```
combined_plot_chart.axes2[0].setValues(tickLength=2)
combined_plot_chart.axes2[0].tickStyle.setValues(style=SOLID)
combined_plot_chart.axes2[0].tickStyle.setValues(thickness=1.2)
combined_plot_chart.axes2[0].tickStyle.setValues(color='#000000')

# Let Abaqus/CAE assign the default X-axis title
combined_plot_chart.axes1[0].axisData.setValues(useSystemTitle=True)

# Set the Y-axis title
combined_plot_chart.axes2[0].axisData.setValues(useSystemTitle=False,
                                                 title='Displacement of node')

# Set the font style and color of the X-axis title
combined_plot_chart.axes1[0].titleStyle \
                  .setValues(font='-*-arial-medium-r-normal-*-*-180-*-*-p-*-*-*')
combined_plot_chart.axes1[0].titleStyle.setValues(color='#000000')

# Set the font style and color of the Y-axis title
combined_plot_chart.axes2[0].titleStyle \
                  .setValues(font='-*-arial-medium-r-normal-*-*-180-*-*-p-*-*-*')
combined_plot_chart.axes2[0].titleStyle.setValues(color='#000000')

# Set the placement of the X-axis (top, bottom or center of chart ?)
combined_plot_chart.axes1[0].setValues(placement=MAX_EDGE)

# Let Abaqus CAE decide the placement of the Y-axis
combined_plot_chart.axes2[0].setValues(placement=MIN_MAX_EDGE)

# Set the format of the X-axis labels to decimal with precision of 3 sig figs
combined_plot_chart.axes1[0].axisData.setValues(labelFormat=DECIMAL)
combined_plot_chart.axes1[0].axisData.setValues(labelNumDigits=3)

# Set the format of the Y-axis labels to decimal with precision of 3 sig figs
combined_plot_chart.axes2[0].axisData.setValues(labelFormat=AUTOMATIC)
combined_plot_chart.axes2[0].axisData.setValues(labelNumDigits=2)

# Set how frequently labels are displayed on the X-axis (1 = every major gridline,
# 2 = every second major gridline etc)
combined_plot_chart.axes1[0].setValues(labelFrequency=1)

# Set how frequently labels are displayed on the Y-axis (1 = every major gridline,
# 2 = every second major gridline etc)
combined_plot_chart.axes2[0].setValues(labelFrequency=1)

# Set the font style and color of the X-axis labels
combined_plot_chart.axes1[0].labelStyle \
                  .setValues(font='-*-verdana-medium-r-normal-*-*-100-*-*-p-*-*-*')
combined_plot_chart.axes1[0].labelStyle.setValues(color='#000000')

# Set the font style and color of the Y-axis labels
combined_plot_chart.axes2[0].labelStyle \
                  .setValues(font='-*-verdana-medium-r-normal-*-*-100-*-*-p-*-*-*')
```

```
combined_plot_chart.axes2[0].labelStyle.setValues(color='#000000')

# Set the style and color of the X-axis line
combined_plot_chart.axes1[0].lineStyle.setValues(style=SOLID)
combined_plot_chart.axes1[0].lineStyle.setValues(thickness=0.2)

# Set the style and color of the Y-axis line
combined_plot_chart.axes2[0].lineStyle.setValues(style=SOLID)
combined_plot_chart.axes2[0].lineStyle.setValues(thickness=0.2)
```

The code has been heavily commented. Please read these comments for an explanation. A lot of the actions performed here by the script can be reused by you for projects of your own.

14.4.12 Title Options

The following statements use the title options to modify the plot title.

```
# -----------------------------------------------------------------------
# Create/Modify the chart title using the plot title options

# Set the plot title
combined_plot.title.setValues(text='Displacement Vs Time')

# Set the plot title font style and color
combined_plot.title.style \
                   .setValues(font='-*-arial-medium-r-normal-*-*-240-*-*-p-*-*-*')
combined_plot.title.style.setValues(color='#008000')

# Do not inset the plot title
combined_plot.title.area.setValues(inset=False)

# Auto-align the plot title at the top-center
combined_plot.title.area.setValues(positionMethod=AUTOMATIC, alignment=TOP_CENTER)

# Give the plot title a border
combined_plot.title.area.border.setValues(show=True)

# Set the color of the plot title border
combined_plot.title.area.border.setValues(color='#000000')
```

This code has also been heavily commented so there is no need for further explanation. All the changes you might wish to make to the plot title are demonstrated here and you can copy and reuse these statements into scripts of your own.

To make sure you understand the object structure here

```
combined_plot.title.setValues(text='Displacement Vs Time')
```

can be written as

```
session.xyPlots['Combined Plot'].title.setValues(text='Displacement Vs Time')
```

14.4.13 Chart Legend Options

The following statements use the chart legend options to modify the legend on the plot

```
# -----------------------------------------------------------------------------
# Modify the plot legend using the chart legend options

# Display the legend
combined_plot_chart.legend.setValues(show=True)

# Give the legend a title
combined_plot_chart.legend.setValues(title='Legend:')

combined_plot_chart.legend.textStyle \
                .setValues(font='-*-verdana-medium-r-normal-*-*-120-*-*-p-*-*-*')
combined_plot_chart.legend.titleStyle \
                .setValues(font='-*-verdana-medium-r-normal-*-*-120-*-*-p-*-*-*')

combined_plot_chart.legend.textStyle.setValues(color='#FF0000')
combined_plot_chart.legend.titleStyle.setValues(color='#FF0000')

# Display minimum and maximum values on the legend
combined_plot_chart.legend.setValues(showMinMax=True)

# Set the format to decimal with precision of 3 sig figs
combined_plot_chart.legend.setValues(numberFormat=DECIMAL)
combined_plot_chart.legend.setValues(numDigits=3)

# Set the position to inset so the legend can be displayed over the grid
combined_plot_chart.legend.area.setValues(inset=True)

# Choose manual positioning, and position the legend at desired location
combined_plot_chart.legend.area.setValues(positionMethod=MANUAL)
combined_plot_chart.legend.area.setValues(originOffset=(0.2, 0.1))

# Give the legend a border
combined_plot_chart.legend.area.border.setValues(show=True)

# Set the color of the legend border
combined_plot_chart.legend.area.border.setValues(color='#0000FF')

# Fill the legend with a white background so the grid is not visible through it
combined_plot_chart.legend.area.style.setValues(fill=True)
combined_plot_chart.legend.area.style.setValues(color='#FFFFFF')
```

```
# Resize the viewport window
combined_plot_viewport.setValues(width=250.0, height=200.0)
```

The heavily commented code explains itself. Chances are any changes you wish to make to the plot legend using a Python script are demonstrated here and you can reuse these statements in projects of your own.

Once again, to make sure you understand the object structure,

```
combined_plot_viewport.setValues(width=250.0, height=200.0)
```

can be written as

```
session.viewports['Combined Plot Viewport'].setValues(width=250.0, height=200.0)
```

14.4.14 XY Curve Options

The following statements use the XY Curve options to modify the legend on the plot

```
# -----------------------------------------------------------------------
# Change the plot curve line style, color, thickness etc using the XY Curve options

# Use the name assigned to the curve object as the name of the curve in the legend
curve_1.setValues(useDefault=True, legendSource=CURVE_NAME)

# Create a new name for the name of the curve in the legend
curve_2.setValues(useDefault=False, legendLabel='Displacement of the end point')

# Set the line style and thickness for the force point curve
curve_1.lineStyle.setValues(show=True)
curve_1.lineStyle.setValues(style=DASHED)
curve_1.lineStyle.setValues(color='#0099FF')
curve_1.lineStyle.setValues(thickness=0.8)

# Set the line style and thickness for the end point curve
curve_2.lineStyle.setValues(color='#000080')
curve_2.lineStyle.setValues(style=SOLID)
curve_2.lineStyle.setValues(thickness=0.5)

# Show symbols on the end point curve
curve_2.symbolStyle.setValues(show=True)
curve_2.symbolStyle.setValues(color='#000080')
curve_2.symbolStyle.setValues(marker=FILLED_CIRCLE)
curve_2.symbolStyle.setValues(size=1.6)
```

The statements are commented and should not require further explanation. They can be copied and reused in your own scripts.

Just so you're paying attention,

```
curve_1.setValues(useDefault=True, legendSource=CURVE_NAME)
```

can be written as

```
session.curves['Data for force point'].setValues(useDefault=True,
legendSource=CURVE_NAME)
```

14.4.15 Print the plot to an image

The following statements save the plot as an image file, in this case a .png (Portable Network Graphics)

```
# ---------------------------------------------------------------------------
# Save the plot by printing it to a png file

# Set the size of the image
session.pngOptions.setValues(imageSize=SIZE_ON_SCREEN)

# Opt not to reduce the image to 256 colors
session.printOptions.setValues(reduceColors=False)

# Store the name and path you wish to give the file in a variable
# If you only specify a name and no path, it will be saved in the Abaqus Work
directory
png_name_and_path = 'combined_plot_image'

# Print the image to a file
# Note that if the file already exists it will be replaced
session.printToFile(fileName=png_name_and_path, format=PNG,
                    canvasObjects=(combined_plot_viewport, ))
```

The comments interspersed with the statements explain the code and you should read those. In addition to them I'll point out a couple of useful facts in case you try to accomplish a similar task in your own scripts.

The statement

```
session.pngOptions.setValues(imageSize=SIZE_ON_SCREEN)
```

sets the size of the png image to be the same as the size of the plot on the users screen using the **SIZE_ON_SCREEN** SymbolicConstant. If you instead wished to set it to an exact size with width and height, you might write it as

```
session.pngOptions.setValues(imageSize=(709, 566))
```

The statement

```
png_name_and_path = 'combined_plot_image'
```

assigns the name of the to-be png image to the variable **png_name_and_path**. When used in conjunction with the **printToFile()** method in the next statement, this places the png in a file called **combined_plot_image.png** in the Abaqus Work Directory. If on the other hand you want to give it a specific location on your hard drive, you could give it a full path as:

```
png_name_and_path='C:/Users/Gautam/Desktop/combined_plot_image'
```

14.5 Summary

In this chapter you saw a good demonstration of the parameterization procedure. Parameterization is the foundation of almost any optimization analysis as it allows you to treat quantities as variables and change them easily without having to recreate the model manually. In addition you now have a few blocks of script code that can modify all aspects of an XY plot, and you can reuse these in your own scripts.

15

Optimization of a Parameterized Sandwich Structure

15.1 Introduction

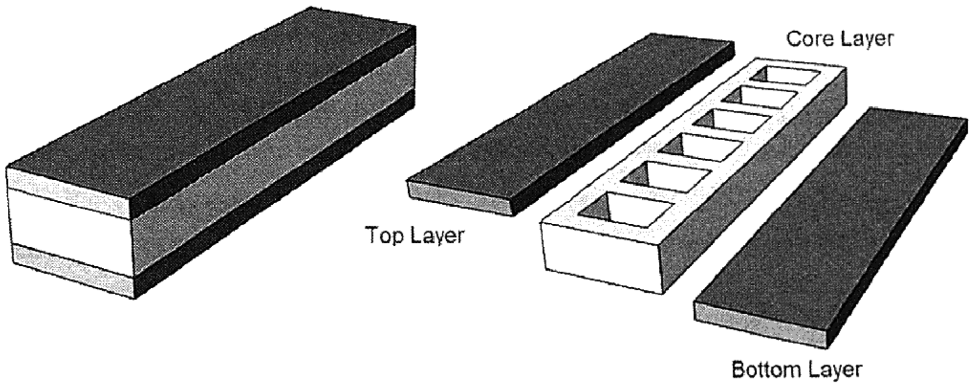
This chapter is another example of both parameterization and optimization studies. We will conduct an iterative optimization study on a parameterized sandwich structure. A sandwich structure consists of a layer of material sandwiched between two other layers which may or may not be of the same material. In our sandwich structure the two outer layers are solid planks or plates whereas the inner layer is a square honeycomb core. One end of the sandwich structure is fixed while the other end is free giving us something similar to a cantilever beam. Tie constraints will be used between the sandwich layers to hold them together.

We will write a parameterized script where the dimensions such as length, width, layer thicknesses and core cell dimensions will be specified at the start of the script, and the entire model will be created on the basis of these.

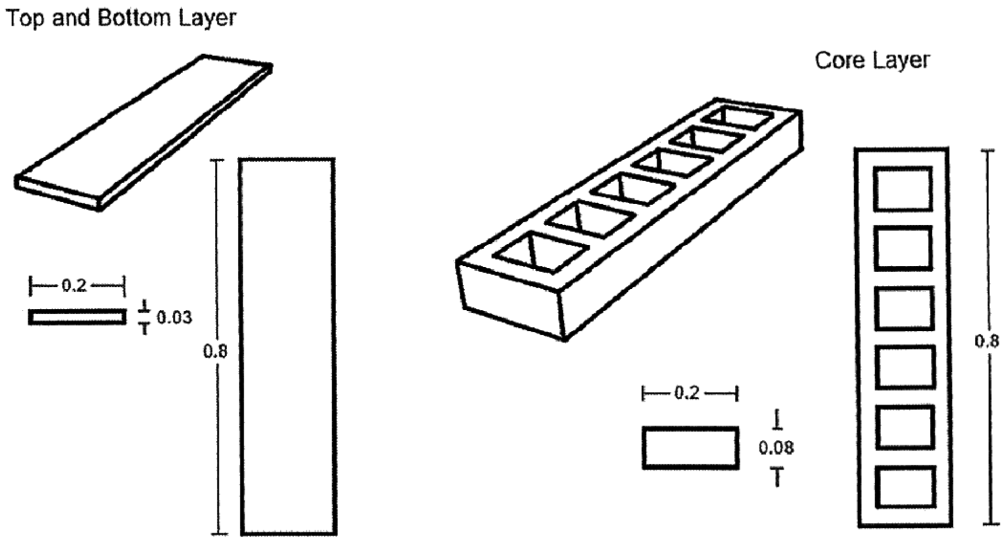
The user will provide input using a text file. Here each line of the text file will consist of tab separated values of all of the variables. For each line of this input file the script will extract the dimensions and perform an analysis. Therefore the bulk of the script will be inside a **for** loop iterating as many times as there are lines in the input file.

The results of each analysis (the displacement of nodes near the end of the sandwich beam) will be printed to an output file along with the input variables as tab separated values. The benefit of having such output is that these values can then be imported into a program such as Microsoft Excel or Matlab for creating plots and observing trends.

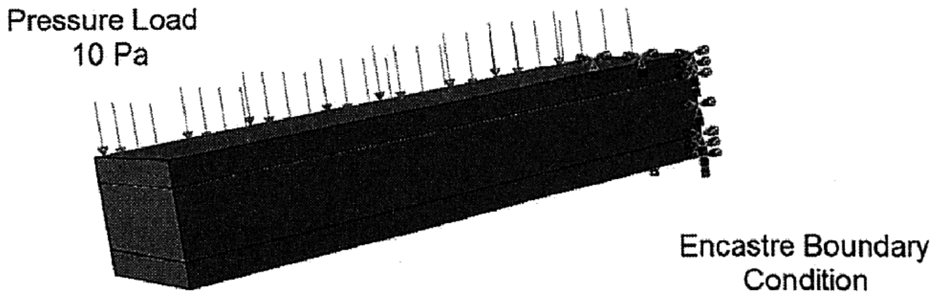
The geometry of our sandwich structure is displayed in the figure.



The following dimensions will be used:



The loads and boundary conditions are displayed in the next figure.



15.2 Procedure in GUI

You can perform the simulation in Abaqus/CAE by following the steps listed below. You can either read through these, or watch the video demonstrating the process on the book website.

1. Rename **Model-1** to **Sandwich Structure**
   a. Right-click on Model-1 in Model Database
   b. Choose **Rename..**
   c. Change name to **Sandwich Structure**
2. Create the Top Layer
   a. Double-click on **Parts** in Model Database. **Create Part** window is displayed.
   b. Set **Name** to **Top Layer**
   c. Set **Modeling Space** to **3D**
   d. Set **Type** to **Deformable**
   e. Set **Base Feature Shape** to **Solid**
   f. Set **Base Feature Type** to **Extrusion**
   g. Set **Approximate Size** to **20**
   h. Click **OK**. You will enter the sketcher mode.
   i. Use the **Create Lines:Rectangle (4 lines)** tool to draw the profile of the plank
   j. Use the **Add Dimension** tool to set the width to 0.2 m and the thickness to 0.03 m.
   k. Click **Done** to exit the sketcher. The **Edit Base Extrusion** window is displayed.
   l. Set **Depth to 0.8**
   m. Click **OK**.
3. Create the Core Layer
   a. Double-click on **Parts** in Model Database. **Create Part** window is displayed.
   b. Set **Name** to **Core Layer**
   c. Set **Modeling Space** to **3D**
   d. Set **Type** to **Deformable**
   e. Set **Base Feature Shape** to **Solid**
   f. Set **Base Feature Type** to **Extrusion**
   g. Set **Approximate Size** to **20**
   h. Click **OK**. You will enter Sketcher mode.

i. Use the **Create Lines: Rectangle (4 lines)** tool to draw the profile of the plank
j. Use the **Add Dimension** tool to set the width to 0.2 m and the thickness to 0.08 m.
k. Click **Done** to exit the sketcher. The **Edit Base Extrusion** window is displayed.
l. Set **Depth** to **0.8**
m. Click **OK**.
n. Click **Create Cut: Extrude** tool.
o. You see the message **Select a plane for the extruded cut** displayed below the viewport
p. Select the top face of the core layer.
q. You see the message **Select an edge or axis that will appear vertical and on the right** displayed below the viewport
r. Select the left edge of the core layer block. You will enter the Sketcher.
s. Use the **Create Lines: Rectangle (4 lines)** tool to draw 6 rectangles that will be cut out of the core.
t. Use the **Add Dimension** tool on each of these rectangles to set the x-dimension to 0.087 m and the y-dimension to 0.12.
u. Click **Done** to exit the sketcher. The **Edit Cut Extrusion** window is displayed.
v. Use the **Add Dimension** tool to set the width to 0.2 m and the thickness to 0.08 m.
w. Click **Done** to exit the sketcher. The **Edit Base Extrusion** window is displayed.
x. Set **Type** to **Through All.**
y. If **Extrude Direction** is not through the block (see arrow) then click **Flip**
z. Click **OK**.

4. Create the Bottom Layer
   n. Double-click on **Parts** in Model Database. **Create Part** window is displayed.
   o. Set **Name** to **Bottom Layer**
   p. Set **Modeling Space** to **3D**
   q. Set **Type** to **Deformable**
   r. Set **Base Feature Shape** to **Solid**
   s. Set **Base Feature Type** to **Extrusion**
   t. Set **Approximate Size** to **20**

u. Click **OK**. You will enter Sketcher mode.
v. Use the **Create Lines:Rectangle (4 lines)** tool to draw the profile of the plank
w. Use the **Add Dimension** tool to set the width to 0.2 m and the thickness to 0.03 m.
x. Click **Done** to exit the sketcher. The **Edit Base Extrusion** window is displayed.
y. Set **Depth** to **0.8**
z. Click **OK**.

5. Create the material
   a. Double-click on **Materials** in the Model Database. **Edit Material** window is displayed
   b. Set **Name** to **AISI 1005 Steel**
   c. Select **General > Density**. Set **Mass Density** to **7800** (which is 7.800 g/cc)
   d. Select **Mechanical > Elasticity > Elastic**. Set **Young's Modulus** to **200E9** (which is 200 GPa) and **Poisson's Ratio** to **0.29.**
   e. Click **OK**.
6. Create 3 sections
   a. Double-click on **Sections** in the Model Database. **Create Section** window is displayed
   b. Set **Name** to **Top Layer Section**
   c. Set **Category** to **Solid**
   d. Set **Type** to **Homogeneous**
   e. Click **Continue...** The **Edit Section** window is displayed.
   f. In the **Basic** tab, set **Material** to **the AISI 1005 Steel** which was defined in the create material step.
   g. Click **OK**.
   h. Again double-click on **Sections** in the Model Database. **Create Section** window is displayed
   i. Set **Name** to **Top Layer Section**
   j. Set **Category** to **Solid**
   k. Set **Type** to **Homogeneous**
   l. Click **Continue...** The **Edit Section** window is displayed.
   m. In the **Basic** tab, set **Material** to **the AISI 1005 Steel** which was defined in the create material step.

- n. Repeat steps a thru m to create two more sections **Core Layer Section** and **Bottom Layer Section**
- o. Click **OK**.

7. Assign the sections to the parts
   - a. Expand the **Parts** container in the Model Database. Expand the part **Top Layer**.
   - b. Double-click on **Section Assignments**
   - c. You see the message **Select the regions to be assigned a section** displayed below the viewport
   - d. Click and drag with the mouse to select the entire top layer.
   - e. Click **Done**. The **Edit Section Assignment** window is displayed.
   - f. Set **Section** to **Top Layer Section.**
   - g. Click **OK.**
   - h. Similarly assign **Bottom Layer Section** to the bottom layer and **Core Layer Section** to the core.
8. Create the Assembly
   - a. Double-click on **Assembly** in the Model Database. The viewport changes to the **Assembly Module.**
   - b. Expand the **Assembly** container.
   - c. Double-click on **Instances**. The **Create Instance** window is displayed.
   - d. Set **Parts to Top Layer**
   - e. Set **Instance Type** to **Dependent (mesh on part)**
   - f. Click **OK**. The top layer is instanced in the assembly.
   - g. Again double-click on **Instances**. The **Create Instance** window is displayed.
   - h. Set **Parts** to **Core Layer**
   - i. Set **Instance Type** to **Dependent (mesh on part)**
   - j. Check **Auto-offset from other instances**
   - k. Click **OK**. The core layer is instanced in the assembly.
   - l. Click the **Create Constraint: Face to Face** tool. You see the message **Select a planar face or datum plane of the movable instance** below the viewport.
   - m. Click the bottom face of the top layer. You see the message **Select a planar face or datum plane of the fixed instance** below the vieport
   - n. Click the top face of the core. Arrows appear on the faces and you see the message **The instance will be moved so that the arrows point in the same direction** below the viewport.

o. Click **OK** or **Flip** as required to have the arrows pointing in the same direction. You see the prompt **Distance from the fixed plane along its normal** below the viewport.
p. Set it to **0.0**
q. Similarly use face to face constraints on the other two surfaces to align the parts as displayed in the figure.
r. Again double-click on **Instances**. The **Create Instance** window is displayed.
s. Set **Parts** to **Bottom Layer**
t. Set **Instance Type** to **Dependent (mesh on part)**
u. Check **Auto-offset from other instances**
v. Click **OK**. The bottom layer is instanced in the assembly.
w. Use the **Create Constraint: Face to Face** tool 3 more times to align the parts as shown in the figure.

9. Create Sets in the Assembly
   a. Expand the **Assembly** container.
   b. Double-click on **Sets**. The **Create Set** window is displayed.
   c. Set **Name** to **displacement point set 1**
   d. Set **Type** to **Geometry**
   e. You see the message **Select the geometry for the set** below the viewport. Select the lower left corner of the core cell closest to the free end of the structure.
   f. Click **Done**
   g. Double-click on **Sets**. The **Create Set** window is displayed.
   h. Set **Name** to **displacement point set 2**
   i. Set **Type** to **Geometry**
   j. You see the message **Select the geometry for the set** below the viewport. Select the lower left right corner of the bottom layer.
   k. Click **Done**
10. Create Steps
    a. Double-click on **Steps** in the Model Database. The **Create Step** window is displayed.
    b. Set **Name** to **Apply Load**
    c. Set **Insert New Step After** to **Initial**
    d. Set **Procedure Type** to **General > Static, General**
    e. Click **Continue..** The **Edit Step** window is displayed

   f. In the **Basic** tab, set **Description** to **Apply the pressure load on top surface of sandwich structure**.
   g. Click **OK**.
11. Leave Field Output Requests at defaults
12. Request History Outputs
   a. Expand **the History Output Requests** container in the Model Database
   b. Right-click on **H-Output-1** and choose Rename…
   c. Change the name to **Displacement output 1**
   d. Double-click on **Displacement output 1** in the Model Database. The **Edit History Output Request** window is displayed
   e. Set **Domain** to **Set**. A new dropdown list appears next to it.
   f. Choose **displacement point set 1** from this list
   g. Set **Frequency** to **Every n time increments**.
   h. Set **n:** to **1**
   i. Select the desired variables by checking them off in the **Output Variables** list. The variable we want is **UT (translations)** from the **Displacement/Velocity/Acceleration** group. Uncheck the rest. You will notice that the text box above the output variable list displays **UT**.
   j. Click **OK**
   k. We need to create the second history output request. Double-click on **History Output Requests** in the Model Database. The **Create History** window is displayed
   l. Set **Name** to **Displacement output 2**
   m. Set **Step** to **Apply Load**
   n. Click **Continue…** The **Edit History Output Request** window is displayed
   o. Set **Domain** to **Set**. A new dropdown list appears next to it.
   p. Choose **displacement point set 2** from this list.
   q. Set **Frequency** to **Every n time increments**
   r. Set **n:** to **1**
   s. Select the desired variables by checking them off in the **Output Variables** list. The variable we want is **UT (translations)** from the **Displacement/Velocity/Acceleration** group. Uncheck the rest. You will notice that the text box above the output variable list displays **UT**.
   t. Click **OK**
13. Apply boundary conditions

a. Double-click on **BCs** in the Model Database. The **Create Boundary Condition** window is displayed
b. Set **Name** to **Fix Top Layer Front**
c. Set **Step** to **Apply Load**
d. Set **Category** to **Mechanical**
e. Set **Types for Selected Step** to **Symmetry/Antisymmetry/Encastre**
f. Click **Continue...**
g. You see the message **Select regions for the boundary condition** displayed below the viewport
h. Select the end face of the top layer.
i. Click **Done**. The **Edit Boundary Condition** window is displayed.
j. Choose **ENCASTRE (U1=U2=U3=UR1=UR2=UR3=0).**
k. Click **OK**.
l. Similarly create a second boundary condition called **Fix Core Layer Front**, applied during the **Apply force** step with **ENCASTRE**. This is applied to the end face of the core layer.
m. Create a third boundary condition called **Fix Bottom Layer Front**, applied during the **Make Contact** step with **ENCASTRE**. This is applied to the end face of the bottom layer.

14. Assign Loads
    a. Double-click on **Loads** in the Model Database. The **Create Load** window is displayed
    b. Set **Name** to **Uniform Applied Pressure**
    c. Set **Step** to **Apply Load**
    d. Set **Category** to **Mechanical**
    e. Set **Type for Selected Step** to **Pressure**
    f. Click **Continue...**
    g. You see the message **Select surfaces for the load** displayed below the viewport
    h. Select the top surface of the top layer by clicking on it.
    i. Click **Done.** The **Edit Load** window is displayed
    j. Set **Distribution** to **Uniform**
    k. Set **Magnitude** to **10** to apply a 10 N force in downward (negative Y) direction
    l. Set **Amplitude** to **Ramp**
    m. Click **OK**

n. You will see the forces displayed with arrows in the viewport on the surface

15. Define surfaces
    a. Expand the **Assembly** container in the Model Database.
    b. Double-click on **Surfaces.** The **Create Surface** window is displayed
    c. Set **Name** to **Top Layer Bottom**
    d. Click **Continue**… You see the message **Select the regions for the surface** displayed below the viewport
    e. Set it to **individually** with the dropdown menu
    f. Select the bottom surface of the top layer. You might need to use the **Replace Selected** tool in the display group toolbar to display just the top layer in order to make its bottom surface visible. For the message **Set entities to replace at the bottom of the viewport** set the drop down item to **Instances**. Once you've selected the surface click **Replace All** to unhide the other part instances.
    g. Similarly assign the surface **Bottom Layer Top** to the top surface of the bottom layer
    h. Similarly assign the surface **Core Layer Bottom** to the bottom surface of the core
    i. Similarly assign the surface **Core Layer Top** to the top surface of the core
16. Assign tie constraints
    a. Double click **Constraints** in the Model Database
    b. Set **Name** to **Tie Constraint 1**
    c. Set **Type** to **Tie**
    d. Click **Continue…** You see the message **Choose the master type** displayed below the viewport
    e. Click **Surface.** You see the message **Select regions for the master surface individual** displayed below the viewport
    f. Click the **Surfaces..** button at the bottom right of the viewport. The **Region Selection** window is displayed
    g. Choose **Core Layer Top.** Check **Highlight selections in viewport** to make sure the correct surface is being selected
    h. Click **Continue…** You see the message **Choose the slave type** displayed below the viewport
    i. Click **Surface.** You see the message **Select regions for the master surface individual** displayed below the viewport

   j. Choose **Top Layer Bottom.** Check **Highlight selections in viewport** to make sure the correct surface is being selected
   k. The **Edit Constraint** window is displayed.
   l. Leave all the settings at defaults. Click **OK.**
   m. Repeat these steps to create another tie constraint **Tie Constraint 1** with **Core Layer Top** as the master and **Bottom Layer Top** as the slave.
17. Create the mesh
   a. Expand the **Parts** container in the Model Database.
   b. Expand **Top Layer**
   c. Double-click on **Mesh (Empty).** The viewport window changes to the **Mesh module** and the tools in the toolbar are now meshing tools.
   d. Using the menu bar click on **Mesh > Element Type …**
   e. You see the message **Select the regions to be assigned element types** displayed below the viewport
   f. Click and drag using your mouse to select the entire top layer.
   g. Click **Done**. The **Element Type** window is displayed.
   h. Set **Element Library** to **Standard**
   i. Set **Geometric Order** to **Linear**
   j. Set **Family** to **3D Stress**
   k. Check **Reduced Integration**
   l. You will notice the message **C3D8R: An 8-node linear brick, reduced integration, hourglass control**
   m. Click **OK**
   n. Using the menu bar lick on **Seed >Part…**The **Global Seeds** window is displayed
   o. Set **Approximate global size** to 0.04. Leave everything else at default values.
   p. Click **OK.**
   q. You see the message **Seeding definition complete** displayed below the viewport. Click **Done.**
   r. Using the menu bar click on **Mesh > Part**
   s. You see the prompt **OK to mesh the part?** displayed below the viewport
   t. Click **Yes**
   u. Repeat the same process for **Bottom Layer** and **Core Layer.**
18. Create and submit the job

a. Double-click on **Jobs** in the Model Database. The **Create Job** window is displayed
b. Set **Name** to **SandwichStructureJob**
c. Set **Source** to **Model**
d. Select **SandwichStructure** (it is the only option displayed)
e. Click **Continue..** The **Edit Job** window is displayed
f. Set **Description** to **Run the sandwich structure simulation**
g. Set **Job Type** to **Full Analysis.**
h. Leave all other options at defaults
i. Click **OK**
j. Expand the**Jobs** container in the Model Database
k. Right-click on **ContactSimulationJob** and choose **Submit**. This will run the simulation. You will see the following messages in the message window:
The job input file " SandwichStructureJob.inp" has been submitted for analysis.
Job SandwichStructureJob: Analysis Input File Processor completed successfully
Job SandwichStructureJob: Abaqus/Standard completed successfully
Job SandwichStructureJob completed successfully

20. Plot mises stress and contact pressures
    a. Right-click on **SandwichStructureJob (Completed)** in the Model Database. Choose **Results.**The viewport changes to the **Visualization** module.
    b. Expand the **SandwichStructure.odb** container in the Results tree
    c. Expand the **History Output** container.
    d. You see two spatial displacement variables U2 for bottom layer instance and core instance.
    e. Right-click on each of them and choose **Save As..** Save them as **SandwichXYData-1** and **SandwichXYData-2.**
    f. Using the menu bar click on **Report>XY...** The **Report XY Data** window is displayed
    g. In the **XY Data** tab, make sure **Select from:** is set to **All XY data. sandwichXYData-1** and **sandwichXYData-2** should be displayed in the list. However sometimes due to a bug in Abaqus (Abaqus v 6.10 does not appear to have this bug) the list may appear empty and needs to be refreshed. To remedy this change **Select from:** to **XY plot in current view** and then back to **All XY data.** You should now see our XY data sets in the list.

h. Click **sandwichXYData-1** to make sure it is selected.
i. Click on the **Setup** tab.
j. In the **File** section, set **Name** to **SandwichXYData.txt** in **C:\SandwichFolder** (you will need to create this folder).
k. Uncheck **Append to file**.
l. In the **Data** section, for **Write:** check **XY data, Columns totals** and **Column min/max**
m. Switch back to **XY Data** tab
n. Make sure **sandwichXYData-1** is selected.
o. Click **Apply**. The file **SandwichXYData.txt** will be written to your Abaqus working directory.
p. Click **sandwichXYData-2** to make sure it is selected.
q. Click on the **Setup** tab.
r. In the **File** section, once again set **Name** to the same **SandwichXYData.txt.**
s. Check **Append to file.**
t. In the **Data** section, for **Write:** check **XY data, Columns totals** and **Column min/max**
u. Switch back to **XY Data** tab
v. Make sure **sandwichXYData-2** is selected.
w. Click **Apply**. The file **SandwichXYData.txt** will be written to your Abaqus working directory.
x. Click **Cancel** to close the **Report XY Data** window.

15.3 Python Script

The following Python script replicates the above procedure for the static analysis of the truss. You can find it in the source code accompanying the book in **sandwich_structure_parameterized_enhanced.py**. You can run it by opening a new model in Abaqus/CAE (**File > New Model database > With Standard/Explicit Model**) and running it with **File > Run Script…**

```
from abaqus import *
from abaqusConstants import *
import regionToolset

iteration_count = 1

input_file=open('C:/sandwichstructure_input.txt')

for line in input_file:
```

```
extracted_line = line
extracted_list = extracted_line.split()
sandwich_width = float(extracted_list[0])
sandwich_length = float(extracted_list[1])
top_layer_thickness = float(extracted_list[2])
bottom_layer_thickness = float(extracted_list[3])
core_layer_thickness = float(extracted_list[4])
no_of_core_cells = int(extracted_list[5])
wall_thickness_core_cells = float(extracted_list[6])

# ---------------------------------------------------------------------
# Variable initialization (units = metres)

reportxy_name = 'SandwichXYData'
reportxy_path = 'C:/SandwichFolder/'

# ---------------------------------------------------------------------
# Initial calculations

# We will draw the cells in the core by making square cutouts
core_layer_cell_cutout_length = (sandwich_length - \
        ((no_of_core_cells + 1)*wall_thickness_core_cells)) / no_of_core_cells

# Point used to find the top surface of the core
# We will be using the findAt command to find the top surface of the core
# As an argument we need to pass the coordinates of a point on this surface
# For an even number of cells, the exact center point of the top surface of
# the core can be used
# However for an odd number of cells, this point will lie over one of the holes
# In that case we'll pick a point offset a little way from it

if no_of_core_cells % 2 == 0:
    coreLayer_top_face_point = (sandwich_width/2, core_layer_thickness,
                                                  sandwich_length/2)
    coreLayer_bottom_face_point = (sandwich_width/2, 0.0, sandwich_length/2)
    coreLayer_inside_coord = (sandwich_width/2, core_layer_thickness/2,
                                                sandwich_length/2)
else:
    coreLayer_top_face_point = (sandwich_width/2,
                                core_layer_thickness,
                                sandwich_length/2 + \
                                   core_layer_cell_cutout_length/2 + \
                                   wall_thickness_core_cells/2)
    coreLayer_bottom_face_point = (sandwich_width/2, 0.0,
                                   sandwich_length/2 + \
                                      core_layer_cell_cutout_length/2 + \
                                      wall_thickness_core_cells/2)
    coreLayer_inside_coord = (sandwich_width/2,
                              core_layer_thickness/2,
                              sandwich_length/2 + \
                                 core_layer_cell_cutout_length/2 + \
                                 wall_thickness_core_cells/2)
```

```
# -------------------------------------------------------------------------
# Abaqus statements start here

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# -------------------------------------------------------------------------
# Create the model

# Change the model name with each iteration by adding the iteration count to
# end of its name
modelname = 'Sandwich Structure' + repr(iteration_count)
sandwichModel = mdb.Model(name=modelname)

# -------------------------------------------------------------------------
# Create the parts

import sketch
import part

# a) Top layer

# i) Sketch the top layer cross section using rectangle tool
topLayerProfileSketch = sandwichModel \
                  .ConstrainedSketch(name='Top Layer Sketch', sheetSize=40)
topLayerProfileSketch.rectangle(point1=(0.0,0.0),
                              point2=(sandwich_width,top_layer_thickness))

# ii) Create a 3D deformable part named "Top Layer" by extruding the sketch
topLayerPart=sandwichModel.Part(name='Top Layer', dimensionality=THREE_D,
                                               type=DEFORMABLE_BODY)
topLayerPart.BaseSolidExtrude(sketch=topLayerProfileSketch,
                             depth=sandwich_length)


# b) Bottom layer

# i) Sketch the bottom layer cross section using rectangle tool
bottomLayerProfileSketch = sandwichModel \
                .ConstrainedSketch(name='Bottom Layer Sketch', sheetSize=40)
bottomLayerProfileSketch.rectangle(point1=(0.0,0.0),
                             point2=(sandwich_width,bottom_layer_thickness))

# ii) Create a 3D deformable part named "Bottom Layer" by extruding the sketch
bottomLayerPart=sandwichModel.Part(name='Bottom Layer', dimensionality=THREE_D,
                                                   type=DEFORMABLE_BODY)
bottomLayerPart.BaseSolidExtrude(sketch=bottomLayerProfileSketch,
                                depth=sandwich_length)

# c) Core layer

# i) Sketch the core layer cross section using rectangle tool
```

```
coreLayerProfileSketch = sandwichModel \
                    .ConstrainedSketch(name='Core Layer Sketch', sheetSize=40)
coreLayerProfileSketch.rectangle(point1=(0.0,0.0),
                                 point2=(sandwich_width,core_layer_thickness))

# ii) Create a 3D deformable part named "Core Layer" by extruding the sketch
coreLayerPart=sandwichModel.Part(name='Core Layer', dimensionality=THREE_D,
                                 type=DEFORMABLE_BODY)
coreLayerPart.BaseSolidExtrude(sketch=coreLayerProfileSketch,
                               depth=sandwich_length)

# iii) Cut out square holes to create the core
# We need to sketch out the squares onto the surface
# In order to enter the sketcher we need to select the surface and an axis
# that will appear vertical and left (or any orientation we choose)

# Select the top surface of core layer block
sketch_core_top_face_point = (sandwich_width/2, core_layer_thickness,
                                                         sandwich_length/2)
sketch_core_top_face = coreLayerPart.faces.findAt(sketch_core_top_face_point,)

# Select the left edge of core layer block
sketch_core_left_edge_point = (sandwich_width/2, core_layer_thickness, 0)
sketch_core_left_edge = coreLayerPart.edges \
                                       .findAt(sketch_core_left_edge_point,)

sketch_transform = coreLayerPart \
                .MakeSketchTransform(sketchPlane=sketch_core_top_face,
                                     sketchUpEdge=sketch_core_left_edge,
                                     sketchPlaneSide=SIDE1,
                                     sketchOrientation=LEFT,
                                     origin=(0.0,0.0,0.0))

coreLayerCutoutSketch = sandwichModel \
                     .ConstrainedSketch(name='core layer cutout sketch',
                                        sheetSize = 50,
                                        transform=sketch_transform)

rect_pt_1_y = wall_thickness_core_cells
rect_pt_2_y = sandwich_width - wall_thickness_core_cells

rect_pt_1_x = wall_thickness_core_cells

for i in range(0,no_of_core_cells):

    rect_pt_2_x =rect_pt_1_x + core_layer_cell_cutout_length
    coreLayerCutoutSketch.rectangle(point1=(rect_pt_1_x,rect_pt_1_y),
                                    point2=(rect_pt_2_x,rect_pt_2_y))
    rect_pt_1_x = rect_pt_2_x + wall_thickness_core_cells

coreLayerPart.CutExtrude(sketchPlane=sketch_core_top_face,
                         sketchUpEdge=sketch_core_left_edge,
                              sketchPlaneSide=SIDE1, sketchOrientation=LEFT,
                              sketch=coreLayerCutoutSketch,
                              flipExtrudeDirection=OFF)


# ---------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs modulus
# and poissons ratio
steelMaterial = sandwichModel.Material(name='AISI 1005 Steel')
steelMaterial.Density(table=((7872, ),       ))
steelMaterial.Elastic(table=((200E9, 0.29), ))


# ---------------------------------------------------------------------
# Create solid section and assign the beam to it

import section

# Create a section to assign to the top layer
topLayerSection = sandwichModel \
                        .HomogeneousSolidSection(name='Top Layer Section',
                                                 material='AISI 1005 Steel')

# Assign the top layer to this section
top_layer_region = (topLayerPart.cells,)
topLayerPart.SectionAssignment(region=top_layer_region,
                               sectionName='Top Layer Section')

# Create a section to assign to the bottom layer
bottomLayerSection = sandwichModel \
                        .HomogeneousSolidSection(name='Bottom Layer Section',
                        material='AISI 1005 Steel')

# Assign the bottom layer to this section
bottom_layer_region = (bottomLayerPart.cells,)
bottomLayerPart.SectionAssignment(region=bottom_layer_region,
                                  sectionName='Bottom Layer Section')

# Create a section to assign to the core layer
coreLayerSection = sandwichModel \
                        .HomogeneousSolidSection(name='Core Layer Section',
                                                 material='AISI 1005 Steel')

# Assign the core layer to this section
core_layer_region = (coreLayerPart.cells,)
coreLayerPart.SectionAssignment(region=core_layer_region,
                                sectionName='Core Layer Section')
```

```
# -------------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instances
sandwichAssembly = sandwichModel.rootAssembly
topLayerInstance = sandwichAssembly.Instance(name='Top Layer Instance',
                                             part=topLayerPart,
                                             dependent=ON)
bottomLayerInstance = sandwichAssembly.Instance(name='Bottom Layer Instance',
                                                part=bottomLayerPart,
                                                dependent=ON)
coreLayerInstance = sandwichAssembly.Instance(name='Core Layer Instance',
                                              part=coreLayerPart,
                                              dependent=ON)

# +++++++++++++++++++++++++++++++++++++++++++++++
# Identify all the faces used to constrain the assembly

topLayer_front_face_point = (sandwich_width/2, top_layer_thickness/2, 0.0)
topLayer_front_face = topLayerInstance.faces.findAt(topLayer_front_face_point,)

topLayer_side_face_point = (0.0, top_layer_thickness/2, sandwich_length/2)
topLayer_side_face = topLayerInstance.faces.findAt(topLayer_side_face_point,)

topLayer_bottom_face_point = (sandwich_width/2, 0.0, sandwich_length/2)
topLayer_bottom_face = topLayerInstance.faces \
                                        .findAt(topLayer_bottom_face_point,)

bottomLayer_front_face_point = (sandwich_width/2, bottom_layer_thickness/2,
                                                                        0.0)
bottomLayer_front_face = bottomLayerInstance.faces \
                                       .findAt(bottomLayer_front_face_point,)

bottomLayer_side_face_point = (0.0, bottom_layer_thickness/2,
                                     sandwich_length/2)
bottomLayer_side_face = bottomLayerInstance.faces \
                                       .findAt(bottomLayer_side_face_point,)

bottomLayer_top_face_point = (sandwich_width/2,
                              bottom_layer_thickness,
                              sandwich_length/2)
bottomLayer_top_face = bottomLayerInstance.faces \
                                       .findAt(bottomLayer_top_face_point,)

coreLayer_front_face_point = (sandwich_width/2, core_layer_thickness/2, 0.0)
coreLayer_front_face = coreLayerInstance.faces \
                                        .findAt(coreLayer_front_face_point,)

coreLayer_side_face_point = (0.0, core_layer_thickness/2, sandwich_length/2)
coreLayer_side_face = coreLayerInstance.faces \
                                        .findAt(coreLayer_side_face_point,)

coreLayer_top_face = coreLayerInstance.faces.findAt(coreLayer_top_face_point,)

coreLayer_bottom_face = coreLayerInstance.faces \
                                      .findAt(coreLayer_bottom_face_point,)

# ++++++++++++++++++++++++++++++++++++++++++++
# Identify all the faces used for boundary conditions

topLayer_fix_front_face_point = (sandwich_width/2, top_layer_thickness/2, 0.0)
topLayer_fix_front_face = topLayerInstance.faces \
                                   .findAt((topLayer_fix_front_face_point,))
topLayer_fix_front_region = regionToolset \
                                   .Region(faces=(topLayer_fix_front_face))

bottomLayer_fix_front_face_point = (sandwich_width/2, bottom_layer_thickness/2,
                                                                        0.0)
bottomLayer_fix_front_face = bottomLayerInstance.faces \
                                .findAt((bottomLayer_fix_front_face_point,))
bottomLayer_fix_front_region = regionToolset \
                                 .Region(faces=(bottomLayer_fix_front_face))

coreLayer_fix_front_face_point = (sandwich_width/2, core_layer_thickness/2,
                                                                       0.0)
coreLayer_fix_front_face = coreLayerInstance.faces \
                                  .findAt((coreLayer_fix_front_face_point,))
coreLayer_fix_front_region = regionToolset \
                                  .Region(faces=(coreLayer_fix_front_face))

# ++++++++++++++++++++++++++++++++++++++++++++
# Identify all the faces used for loads

topLayer_load_top_face_point = (sandwich_width/2, top_layer_thickness,
                                                        sandwich_length/2)
topLayer_load_top_face = topLayerInstance.faces \
                                   .findAt((topLayer_load_top_face_point,))

# ++++++++++++++++++++++++++++++++++++++++++++
# Create a set to measure displacement history output

# For the first history point we use one of the upper corners of the core
# layer inside one of its holes
vertex_coords_for_displacement_1 = (wall_thickness_core_cells, 0.0,
                                    sandwich_length-wall_thickness_core_cells)
vertex_for_displacement_1 = coreLayerInstance.vertices \
                                 .findAt((vertex_coords_for_displacement_1,))
sandwichAssembly.Set(vertices=vertex_for_displacement_1,
                     name='displacement point set 1')

# For the second history point we use one of the lower corners at the end of
# the bottom layer
```

```
vertex_coords_for_displacement_2 = (sandwich_width, 0.0, sandwich_length)
vertex_for_displacement_2 = bottomLayerInstance.vertices \
                                .findAt((vertex_coords_for_displacement_2,))
sandwichAssembly.Set(vertices=vertex_for_displacement_2,
                     name='displacement point set 2')

# +++++++++++++++++++++++++++++++++++++++++++
# Identify faces used to define Surfaces in the assembly. These will later be
# used for tie constraints.

topLayer_bottom_surface_point = (sandwich_width/2, 0.0, sandwich_length/2)
topLayer_bottom_surface = topLayerInstance.faces \
                                  .findAt((topLayer_bottom_surface_point,))

bottomLayer_top_surface_point = (sandwich_width/2, bottom_layer_thickness,
                                                   sandwich_length/2)
bottomLayer_top_surface = bottomLayerInstance.faces \
                                  .findAt((bottomLayer_top_surface_point,))

coreLayer_top_surface_point = coreLayer_top_face_point
coreLayer_top_surface = coreLayerInstance.faces \
                                    .findAt((coreLayer_top_surface_point,))

coreLayer_bottom_surface_point = coreLayer_bottom_face_point
coreLayer_bottom_surface = coreLayerInstance.faces \
                                 .findAt((coreLayer_bottom_surface_point,))

# +++++++++++++++++++++++++++++++++++++++++++
# Assemble the parts using face to face relationships

# Establish face to face relationships between top layer and core layer
sandwichAssembly.FaceToFace(movablePlane=topLayer_front_face,
                            fixedPlane=coreLayer_front_face, flip=OFF,
                            clearance=0.0)
sandwichAssembly.FaceToFace(movablePlane=topLayer_side_face,
                            fixedPlane=coreLayer_side_face, flip=OFF,
                            clearance=0.0)
sandwichAssembly.FaceToFace(movablePlane=topLayer_bottom_face,
                            fixedPlane=coreLayer_top_face, flip=ON,
                            clearance=0.0)

# Establish face to face relationships between bottom layer and core layer
sandwichAssembly.FaceToFace(movablePlane=bottomLayer_front_face,
                            fixedPlane=coreLayer_front_face, flip=OFF,
                            clearance=0.0)
sandwichAssembly.FaceToFace(movablePlane=bottomLayer_side_face,
                            fixedPlane=coreLayer_side_face, flip=OFF,
                            clearance=0.0)
sandwichAssembly.FaceToFace(movablePlane=bottomLayer_top_face,
                            fixedPlane=coreLayer_bottom_face, flip=ON,
                            clearance=0.0)
```

```
# ------------------------------------------------------------------
# Create the steps

import step

# Create the loading step
sandwichModel.StaticStep(name='Apply Load', previous='Initial',
   description='Apply the pressure load on top surface of sandwich structure')

# ------------------------------------------------------------------
# Create the field output request
# Leave at defaults

# ------------------------------------------------------------------
# Create the history output request

displacement_point_region_1 = sandwichAssembly.sets['displacement point set 1']
sandwichModel.historyOutputRequests.changeKey(fromName='H-Output-1',
                                         toName='Displacement output 1')
sandwichModel.historyOutputRequests['Displacement output 1'] \
                            .setValues(variables=('UT',), frequency=1,
                                      region=displacement_point_region_1,
                                      sectionPoints=DEFAULT, rebar=EXCLUDE)

displacement_point_region_2 = sandwichAssembly \
                                          .sets['displacement point set 2']
sandwichModel.HistoryOutputRequest(name='Displacement output 2',
                                   createStepName='Apply Load',
                                   variables=('UT',),
                                   region=displacement_point_region_2,
                                   sectionPoints=DEFAULT, rebar=EXCLUDE)

# ------------------------------------------------------------------
# Apply boundary conditions

sandwichModel.EncastreBC(name='Fix Top Layer Front',
                         createStepName='Apply Load',
                         region=topLayer_fix_front_region)
sandwichModel.EncastreBC(name='Fix Bottom Layer Front',
                         createStepName='Apply Load',
                         region=bottomLayer_fix_front_region)
sandwichModel.EncastreBC(name='Fix Core Layer Front',
                         createStepName='Apply Load',
                         region=coreLayer_fix_front_region)

# ------------------------------------------------------------------
# Apply loads

# we extract the region of the face choosing which direction its normal points
in
topLayer_top_face_region=regionToolset \
                                   .Region(side1Faces=topLayer_load_top_face)
```

```
# apply the pressure load on this region in the 'Apply Load' step
sandwichModel.Pressure(name='Uniform Applied Pressure',
                      createStepName='Apply Load',
                      region=topLayer_top_face_region,
                      distributionType=UNIFORM, magnitude=10, amplitude=UNSET)

# -------------------------------------------------------------------
# Define surfaces to use in tie constraints

sandwichAssembly.Surface(side1Faces=topLayer_bottom_surface,
                                                   name='Top Layer Bottom')
sandwichAssembly.Surface(side1Faces=bottomLayer_top_surface,
                                                   name='Bottom Layer Top')
sandwichAssembly.Surface(side1Faces=coreLayer_bottom_surface,
                                                   name='Core Layer Bottom')
sandwichAssembly.Surface(side1Faces=coreLayer_top_surface,
                                                   name='Core Layer Top')

# -------------------------------------------------------------------
# Create tie constraints

import interaction

region1=sandwichAssembly.surfaces['Core Layer Top']
region2=sandwichAssembly.surfaces['Top Layer Bottom']

sandwichModel.Tie(name='Constraint-1', master=region1, slave=region2,
                  adjust=ON, tieRotations=ON,
                  constraintEnforcement=SURFACE_TO_SURFACE)

region1=sandwichAssembly.surfaces['Core Layer Bottom']
region2=sandwichAssembly.surfaces['Bottom Layer Top']

sandwichModel.Tie(name='Constraint-2', master=region1, slave=region2,
                  adjust=ON, tieRotations=ON,
                  constraintEnforcement=SURFACE_TO_SURFACE)

# -------------------------------------------------------------------
# Create the mesh

import mesh

# +++++++++++++++++++++++++++++++++++++++++++
# Mesh the top layer
# We place a point somewhere inside it based on our knowledge of the geometry
topLayer_inside_coord=(sandwich_width/2, top_layer_thickness/2,
                                        sandwich_length/2)

elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN,
                          secondOrderAccuracy=OFF,
                              hourglassControl=DEFAULT, distortionControl=DEFAULT)

topLayerCells=topLayerPart.cells
selectedTopLayerCells=topLayerCells.findAt(topLayer_inside_coord,)
topLayerMeshRegion=(selectedTopLayerCells,)
topLayerPart.setElementType(regions=topLayerMeshRegion, elemTypes=(elemType1,))

topLayerPart.seedPart(size=0.04, deviationFactor=0.1)

topLayerPart.generateMesh()

# +++++++++++++++++++++++++++++++++++++++++++++
# Mesh the bottom layer
# We place a point somewhere inside it based on our knowledge of the geometry
bottomLayer_inside_coord=(sandwich_width/2, bottom_layer_thickness/2,
                                            sandwich_length/2)

elemType2 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN,
                          secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)

bottomLayerCells=bottomLayerPart.cells
selectedBottomLayerCells=bottomLayerCells.findAt(bottomLayer_inside_coord,)
bottomLayerMeshRegion=(selectedBottomLayerCells,)
bottomLayerPart.setElementType(regions=bottomLayerMeshRegion,
                               elemTypes=(elemType2,))

bottomLayerPart.seedPart(size=0.04, deviationFactor=0.1)

bottomLayerPart.generateMesh()

# +++++++++++++++++++++++++++++++++++++++++++++
# Mesh the core layer
# We place a point somewhere inside it based on our knowledge of the geometry
# This point has already been defined in the initial calculations section as
# coreLayer_inside_coord

elemType3 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN,
                          secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)

coreLayerCells=coreLayerPart.cells
selectedCoreLayerCells=coreLayerCells.findAt(coreLayer_inside_coord,)
coreLayerMeshRegion=(selectedCoreLayerCells,)
coreLayerPart.setElementType(regions=coreLayerMeshRegion,
                             elemTypes=(elemType3,))

coreLayerPart.seedPart(size=0.04, deviationFactor=0.1)

coreLayerPart.generateMesh()
```

```
# ---------------------------------------------------------------------
# Create and run the job

import job

# Create the job
job_name = 'SandwichStructureJob' + repr(iteration_count)

mdb.Job(name=job_name, model=modelname, type=ANALYSIS,
        explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE,
        description='Run the contact simulation',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)

# Run the job
mdb.jobs[job_name].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs[job_name].waitForCompletion()

# End of run job
# ---------------------------------------------------------------------


# =====================================================================
# ---------------------------------------------------------------------
# Post processing
# ---------------------------------------------------------------------
# =====================================================================



# ---------------------------------------------------------------------
# Send XY Data for U3 displacement of bottom and top center points to an
# output file

import odbAccess
import visualization

sandwich_odb_path = job_name + '.odb'
sandwich_odb_object = session.openOdb(name=sandwich_odb_path)

# The main session viewport must be set to the odb object using the following
# line. If not you might receive an error message that states
# "The current viewport is not associated with an output database file.
# Request operation cancelled."
session.viewports['Viewport: 1'].setValues(displayedObject=sandwich_odb_object)

keyarray=session.odbData[sandwich_odb_path].historyVariables.keys()
```

```
theoutputvariablename=[]

for x in keyarray:
    if (x.find('U2')>-1):
        theoutputvariablename.append(x)

# You may enter an entire path if you wish to have the report stored in a
# particular location.
# One way to do it is using the following syntax.

reportxy_name_and_path = reportxy_path + reportxy_name + '.txt'

# Note however that the folder 'MyNewFolder' must exist otherwise you will
# likely get the following error
# "IOError:C/MyNewFolder: Directory not found"
# You must either create the folder in Windows before running the script
# Or if you wish to create it using Python commands you must use the
# os.makedir() or os.makedirs() function
# os.makedirs() is preferable because you can create multiple nested
# directories in one statent if you wish
# Note that this function returns an exception if the directory already exists
# hence it is a good idea to use a try block

try:
    os.makedirs(reportxy_path)
except:
    print "Directory exists hence no need to recreate it. Move on to " + \
           "next statement"


session.XYDataFromHistory(name='sandwichXYData-1', odb=sandwich_odb_object,
                          outputVariableName=theoutputvariablename[0])
sandwich_xydata_object = session.xyDataObjects['sandwichXYData-1']
session.xyReportOptions.setValues(totals=ON, minMax=ON)
session.writeXYReport(fileName=reportxy_name_and_path,
                      xyData=(sandwich_xydata_object, ), appendMode=OFF)

session.XYDataFromHistory(name='sandwichXYData-2', odb=sandwich_odb_object,
                          outputVariableName=theoutputvariablename[1])
sandwich_xydata_object = session.xyDataObjects['sandwichXYData-2']
session.xyReportOptions.setValues(totals=ON, minMax=ON)
session.writeXYReport(fileName=reportxy_name_and_path,
                      xyData=(sandwich_xydata_object, ), appendMode=ON)

sandwich_odb_object.close()

# ----------------------------------------------------------------------
# Read the displacement from the report

extracted_line = ''
# Need a boolean variable to state whether we are reading the correct section
# of the file
```

```
    file_xydata_section = 0

    f=open(reportxy_name_and_path)
    for line in f:
        str=line
        if 'sandwichXYData-2' in str:
            file_xydata_section = 1
        if 'MINIMUM' in str and file_xydata_section == 1:
                extracted_line = str
                extracted_list = extracted_line.split()
                max_displacement = extracted_list[2]
                print "!The displacement of the node at end of beam is " + \
                                                repr(-1 * float(max_displacement))

    f.close()

    # -----------------------------------------------------------------
    # Write this value as well as inputs to the output file

    file_output = open(reportxy_path + 'sandwichstructure_output.txt','a')
    if iteration_count == 1:
        file_output.write("Sandwich Structure Iterative Simulation Output" + "\n")
        file_output.write("Beam width" + "\t" + "Beam length" + "\t" + \
                          "Top Layer" + "\t" + "Bottom Layer" + "\t" +  \
                          "Core Layer" + "\t" + "#cells" + "\t" + \
                          "wall thickness" + "\t" + "Deflection" + "\n ")

    file_output.write(repr(sandwich_width) + "\t\t" + repr(sandwich_length) + \
                      "\t\t" + repr(top_layer_thickness) + "\t\t" + \
                      repr(bottom_layer_thickness) + "\t\t" + \
                      repr(core_layer_thickness) + "\t\t" + \
                      repr(no_of_core_cells) + "\t" +  \
                      repr(wall_thickness_core_cells) + "\t\t" + \
                      repr(-1 * float(max_displacement)) + "\n")
    file_output.close()

    iteration_count = iteration_count + 1

input_file.close()
```

15.4 Examining the Script

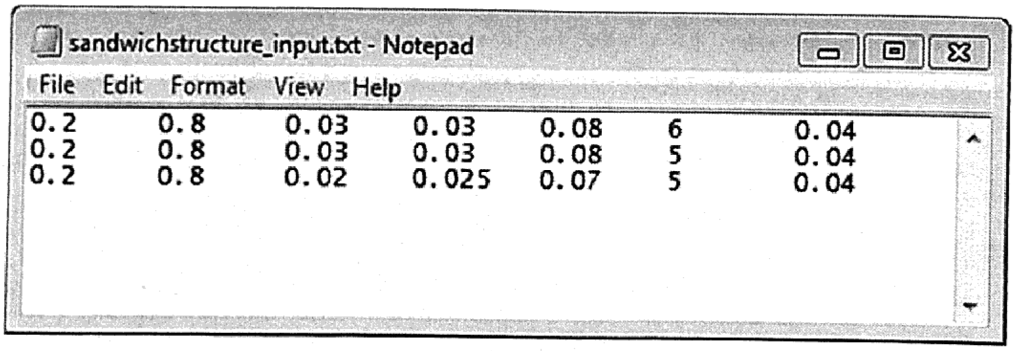
Let's examine this script in detail.

15.4.1 Accept inputs

The following block accepts inputs from the user by presenting him/her with prompt boxes

```
from abaqus import *
from abaqusConstants import *
import regionToolset

iteration_count = 1

input_file=open('C:/sandwichstructure_input.txt')

for line in input_file:
    extracted_line = line
    extracted_list = extracted_line.split()
    sandwich_width = float(extracted_list[0])
    sandwich_length = float(extracted_list[1])
    top_layer_thickness = float(extracted_list[2])
    bottom_layer_thickness = float(extracted_list[3])
    core_layer_thickness = float(extracted_list[4])
    no_of_core_cells = int(extracted_list[5])
    wall_thickness_core_cells = float(extracted_list[6])

                                ...
                                ...
                 (REST OF SCRIPT APPEARS HERE )
                                ...
                                ...

input_file.close()
```

The input file might look something like this:


sandwichstructure_input.txt - Notepad
File Edit Format View Help

```
0.2     0.8     0.03    0.03    0.08    6       0.04
0.2     0.8     0.03    0.03    0.08    5       0.04
0.2     0.8     0.02    0.025   0.07    5       0.04
```

The **open()** method is used to access the input file. A **for** loop extracts each line one by one using the statement '*for line in input_file*' where each line of the input file gets stored in the variable **line**. The **split()** command is used to split up this line at the separators (in this case tabs) and store each String (they are Strings even though they look like floats because we are reading from a file) in a list. We then assign these to variables using the **float()** and **int()** methods to convert them from Strings to floats or integers.

These variables are used to run the parameterized script, and then the next iteration begins till all lines of the file have been used. Then the input file is closed using the **close()** method.

15.4.2 Variable initialization and preliminary calculations

This block initializes variables and performs some preliminary calculations.

```
# ---------------------------------------------------------------------
# Variable initialization (units = metres)

reportxy_name = 'SandwichXYData'
reportxy_path = 'C:/SandwichFolder/'

# ---------------------------------------------------------------------
# Initial calculations

# We will draw the cells in the core by making square cutouts
core_layer_cell_cutout_length = (sandwich_length - \
        ((no_of_core_cells + 1)*wall_thickness_core_cells)) / no_of_core_cells

# Point used to find the top surface of the core
# We will be using the findAt command to find the top surface of the core
# As an argument we need to pass the coordinates of a point on this surface
# For an even number of cells, the exact center point of the top surface of
# the core can be used
# However for an odd number of cells, this point will lie over one of the holes
# In that case we'll pick a point offset a little way from it

if no_of_core_cells % 2 == 0:
    coreLayer_top_face_point = (sandwich_width/2, core_layer_thickness,
                                                  sandwich_length/2)
    coreLayer_bottom_face_point = (sandwich_width/2, 0.0, sandwich_length/2)
    coreLayer_inside_coord = (sandwich_width/2, core_layer_thickness/2,
                                                sandwich_length/2)
else:
    coreLayer_top_face_point = (sandwich_width/2,
                                core_layer_thickness,
                                sandwich_length/2 + \
                                   core_layer_cell_cutout_length/2 + \
                                   wall_thickness_core_cells/2)
    coreLayer_bottom_face_point = (sandwich_width/2, 0.0,
                                   sandwich_length/2 + \
                                      core_layer_cell_cutout_length/2 + \
                                      wall_thickness_core_cells/2)
    coreLayer_inside_coord = (sandwich_width/2,
                              core_layer_thickness/2,
                              sandwich_length/2 + \
                                 core_layer_cell_cutout_length/2 + \
                                 wall_thickness_core_cells/2)
```

We perform some variable initializations for name of the report and the path it will be placed in. We also perform some initial calculations. Since we get the number of core cells, the wall thickness of the cells, as well as the length of the sandwich structure from the input file, we can calculate the length of each cell in order to cut-extrude them from the core layer in the part creation block.

We will later need to select the top and bottom faces of the core to create surfaces and the center of the core to create cells for the mesh using **findAt()**. The problem is that the center point of the core might be a hole after the part is created if there is an odd number of cells in the core. In such a case **findAt()** will not be able to locate any point at the center coordinates. Hence an appropriate algorithm is used depending on whether the number of cells is even or odd to find a center point or a point close enough to the center. If an odd number of cells are present, the point used will be moved away from the center such that it lies on an actual surface or inside an actual solid material.

15.4.3 Create the model

This block creates the model

```
# -------------------------------------------------------------------
# Abaqus statements start here

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# -------------------------------------------------------------------
# Create the model

# Change the model name with each iteration by adding the iteration count to
# end of its name
modelname = 'Sandwich Structure' + repr(iteration_count)
sandwichModel = mdb.Model(name=modelname)
```

The model name has the iteration number appended to it so that at the end of all the simulation runs the Abaqus/CAE model tree will contain all the models created. This will be useful in case you wish to examine them. The **repr()** method provided by Python is used to convert the integer to a String.

15.4.4 Create the parts, material, section and assembly

The following blocks create the parts, material, and section, and assemble the parts.

```
    # ---------------------------------------------------
```

```
        # Create the parts
                ..
                ..
                ..

        # -----------------------------------------------------
        # Create material
                ..
                ..
                ..

        # -------------------------------------------------------
        # Create solid section and assign the beam to it
                ..
                ..
                ..

        # -------------------------------------------------------
        # Create the assembly
                ..
                ..
                ..
```

All of these blocks are familiar to you. The part creation process is fairly simple using **Part()** and **BaseSolidExtrude()**. For the core the **MakeSketchTransform()** method is used to transform the surface to a 2D plane on which you can draw the cutouts. **CutExtrude()** is used to extrude the sketch and cut the rectangular holes out of the core.

15.4.5 Identify faces and sets

The following block identifies faces and sets for later use.

```
        # +++++++++++++++++++++++++++++++++++++++++++
        # Identify all the faces used to constrain the assembly
                ..
                ..
                ..


        # +++++++++++++++++++++++++++++++++++++++++++
    # Identify all the faces used for boundary conditions
            ..
            ..
            ..

    # ++++++++++++++++++++++++++++++++++++++++++++
    # Identify all the faces used for loads
            ..
            ..
            ..

    # ++++++++++++++++++++++++++++++++++++++++++++
    # Create a set to measure displacement history output
            ..
            ..
            ..

    # ++++++++++++++++++++++++++++++++++++++++++++
    # Identify faces used to define Surfaces in the assembly. These will
      later be used for tie constraints.
            ..
            ..
            ..
```

A number of faces are identified using the **findAt()** method. These faces will later be used to constrain the assembly, assign loads and boundary conditions, and define surfaces in the assembly. The sets identified are the points at which history output information will be requested. All the coordinates supplied to the **findAt()** method are obtained using simple formulas that factor in the dimensions of the parts.

15.4.6 Assemble parts

The following block assembles the layers together.

```
# ++++++++++++++++++++++++++++++++++++++++++++
# Assemble the parts using face to face relationships

# Establish face to face relationships between top layer and core layer
sandwichAssembly.FaceToFace(movablePlane=topLayer_front_face,
                            fixedPlane=coreLayer_front_face, flip=OFF,
                            clearance=0.0)
sandwichAssembly.FaceToFace(movablePlane=topLayer_side_face,
                            fixedPlane=coreLayer_side_face, flip=OFF,
                            clearance=0.0)
sandwichAssembly.FaceToFace(movablePlane=topLayer_bottom_face,
                            fixedPlane=coreLayer_top_face, flip=ON,
                            clearance=0.0)
```

```
    # Establish face to face relationships between bottom layer and core layer
    sandwichAssembly.FaceToFace(movablePlane=bottomLayer_front_face,
                                fixedPlane=coreLayer_front_face, flip=OFF,
                                clearance=0.0)
    sandwichAssembly.FaceToFace(movablePlane=bottomLayer_side_face,
                                fixedPlane=coreLayer_side_face, flip=OFF,
                                clearance=0.0)
    sandwichAssembly.FaceToFace(movablePlane=bottomLayer_top_face,
                                fixedPlane=coreLayer_bottom_face, flip=ON,
                                clearance=0.0)
```

The **FaceToFace()** method is used to geometrically constrain the parts in the assembly. **FaceToFace()** moves an instance (the movable instance) so that its face is coincident with that of the other instance (the fixed instance). It has 4 required arguments. **movablePlane** is a planar face, datum plane or orphan mesh face on the movable part instance whereas **fixedPlane** is on the fixed part instance. **flip** is a Boolean specifying whether the normal to the faces are forward aligned (OFF) or reverse aligned (ON). **clearance** is the distance that should separate the faces once they are constrained.

15.4.7 Create steps, boundary conditions and loads

The following blocks create the steps, loads and boundary conditions.

```
        # -------------------------------------------------------------------
        # Create the steps
                ..
                ..
                ..

        # -------------------------------------------------------------------
        # Apply boundary conditions
                ..
                ..
                ..

        # -------------------------------------------------------------------
        # Apply loads
                ..
                ..
                ..
```

There is nothing new here that you haven't seen in previous examples, these blocks create the steps and assign the loads and boundary conditions in the usual manner.

15.4.8 Surfaces and Tie constraints

This block ties the layers together.

```
# -------------------------------------------------------------------
# Define surfaces to use in tie constraints

sandwichAssembly.Surface(side1Faces=topLayer_bottom_surface,
                                                    name='Top Layer Bottom')
sandwichAssembly.Surface(side1Faces=bottomLayer_top_surface,
                                                    name='Bottom Layer Top')
sandwichAssembly.Surface(side1Faces=coreLayer_bottom_surface,
                                                    name='Core Layer Bottom')
sandwichAssembly.Surface(side1Faces=coreLayer_top_surface,
                                                    name='Core Layer Top')

# -------------------------------------------------------------------
# Create tie constraints

import interaction

region1=sandwichAssembly.surfaces['Core Layer Top']
region2=sandwichAssembly.surfaces['Top Layer Bottom']

sandwichModel.Tie(name='Constraint-1', master=region1, slave=region2,
                  adjust=ON, tieRotations=ON,
                  constraintEnforcement=SURFACE_TO_SURFACE)

region1=sandwichAssembly.surfaces['Core Layer Bottom']
region2=sandwichAssembly.surfaces['Bottom Layer Top']

sandwichModel.Tie(name='Constraint-2', master=region1, slave=region2,
                  adjust=ON, tieRotations=ON,
                  constraintEnforcement=SURFACE_TO_SURFACE)
```

The **surface()** method has already been explained in section 12.4.12 on page 312. It is used to define the surfaces which will subsequently be utilized for the tie constraints.

The surfaces are then assigned to variables named **region1** and **region2** and supplied as parameters to the **Tie()** method which expects **Region** objects as its parameters. **Surface** objects are a type of **Region** hence this type of usage is possible.

The **Tie()** method creates a **Tie** object. It requires 3 arguments – **name** which is a String specifying its repository key, **master** which is a **Region** object specifying the master surface, and **slave** which is a **Region** object specifying the slave surface. Optional arguments supplied here include **adjust** which is a Boolean specifying whether the initial positions of the slave nodes should be adjusted to lie on the master surface, **tieRotations** which is a Boolean specifying whether rotation degrees of freedom should be tied, **constraintEnforcement** which is a SymbolicConstant specifying the discretization method with possible values of **SOLVER_DEFAULT**, **NODE_TO_SURFACE** and **SURFACE_TO_SURFACE**, **thickness** which is a Boolean specifying whether shell element thickness is considered, **positionTolerance** which is a Float specifying the position tolerance

15.4.9 Mesh and Run Job

These blocks mesh the parts and run the analysis.

```
    # -------------------------------------------------------------------
    # Create the mesh
            ..
            ..
            ..

    # -------------------------------------------------------------------
    # Create and run the job

    import job

    # Create the job
    job_name = 'SandwichStructureJob' + repr(iteration_count)

            ..
            ..
            ..

# Run the job
            ..

# Do not return control till job is finished running
            ..
```

The mesh generation procedure is quite standard. So is the job creation procedure. The only point to note here is that the job name includes the iteration number using the Python **repr()** method to convert the integer to a String. This means all the output databases will be stored rather than being overwritten as they have different names. This allows the analyst to access any of them for further study if necessary.

15.4.10 XY Reports

This block creates XY reports from the history output.

```
# ----------------------------------------------------------------------
# Send XY Data for U3 displacement of bottom and top center points to an
# output file

import odbAccess
import visualization

sandwich_odb_path = job_name + '.odb'
sandwich_odb_object = session.openOdb(name=sandwich_odb_path)

# The main session viewport must be set to the odb object using the following
# line. If not you might receive an error message that states
# "The current viewport is not associated with an output database file.
# Request operation cancelled."
session.viewports['Viewport: 1'].setValues(displayedObject=sandwich_odb_object)

keyarray=session.odbData[sandwich_odb_path].historyVariables.keys()

theoutputvariablename=[]

for x in keyarray:
    if (x.find('U2')>-1):
        theoutputvariablename.append(x)

# You may enter an entire path if you wish to have the report stored in a
# particular location.
# One way to do it is using the following syntax.

reportxy_name_and_path = reportxy_path + reportxy_name + '.txt'

# Note however that the folder 'MyNewFolder' must exist otherwise you will
# likely get the following error
# "IOError:C/MyNewFolder: Directory not found"
# You must either create the folder in Windows before running the script
# Or if you wish to create it using Python commands you must use the
# os.makedir() or os.makedirs() function
# os.makedirs() is preferable because you can create multiple nested
# directories in one statent if you wish
# Note that this function returns an exception if the directory already exists
# hence it is a good idea to use a try block

try:
    os.makedirs(reportxy_path)
except:
    print "Directory exists hence no need to recreate it. Move on to " + \
          "next statement"


session.XYDataFromHistory(name='sandwichXYData-1', odb=sandwich_odb_object,
                          outputVariableName=theoutputvariablename[0])
sandwich_xydata_object = session.xyDataObjects['sandwichXYData-1']
session.xyReportOptions.setValues(totals=ON, minMax=ON)
```

```
    session.writeXYReport(fileName=reportxy_name_and_path,
                          xyData=(sandwich_xydata_object, ), appendMode=OFF)

    session.XYDataFromHistory(name='sandwichXYData-2', odb=sandwich_odb_object,
                              outputVariableName=theoutputvariablename[1])
    sandwich_xydata_object = session.xyDataObjects['sandwichXYData-2']
    session.xyReportOptions.setValues(totals=ON, minMax=ON)
    session.writeXYReport(fileName=reportxy_name_and_path,
                          xyData=(sandwich_xydata_object, ), appendMode=ON)

    sandwich_odb_object.close()
```

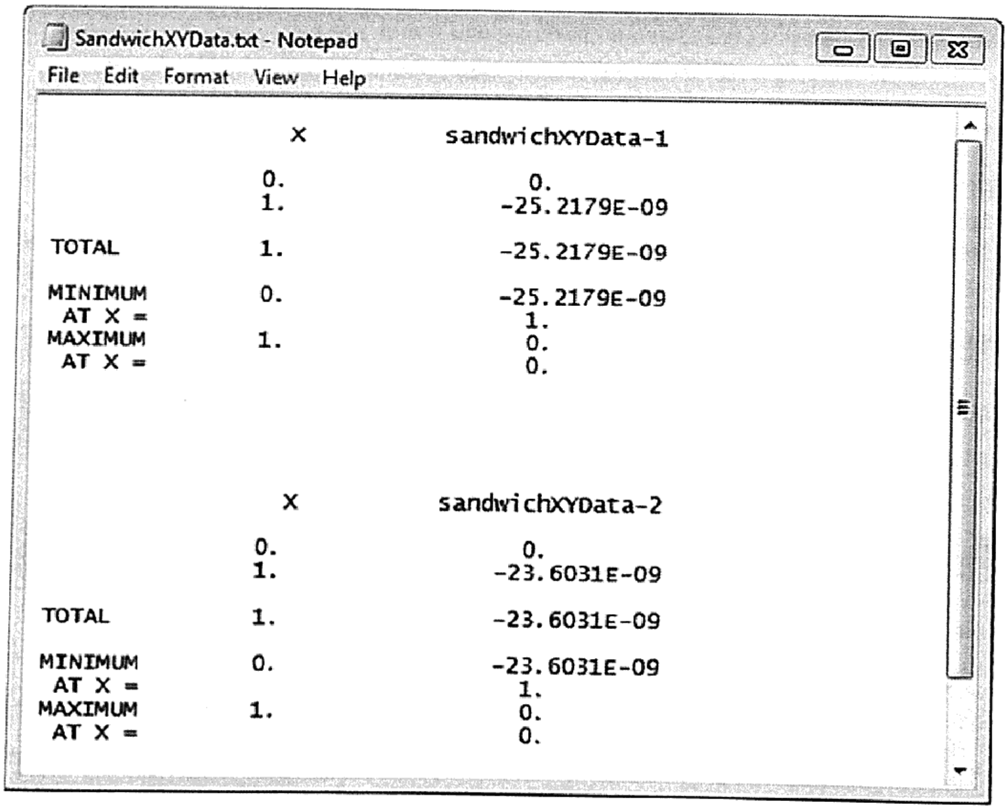
These statements write the displacements of the two nodes to XYReports. The procedure is similar to the dynamic explicit truss analysis example, section 8.3.7 on page 165.

15.4.11 Read from report

```
    # -------------------------------------------------------------------
    # Read the displacement from the report

    extracted_line = ''
    # Need a boolean variable to state whether we are reading the correct section
    # of the file
    file_xydata_section = 0

    f=open(reportxy_name_and_path)
    for line in f:
        str=line
        if 'sandwichXYData-2' in str:
            file_xydata_section = 1
        if 'MINIMUM' in str and file_xydata_section == 1:
                extracted_line = str
                extracted_list = extracted_line.split()
                max_displacement = extracted_list[2]
                print "!The displacement of the node at end of beam is " + \
                                                 repr(-1 * float(max_displacement))

    f.close()
```

The above statements open the XY-reports written earlier and read in the displacement from them.

The XY-report will look something like this (although the actual numbers will vary depending on your inputs).



```
                     X                 sandwichXYData-1

                  0.                      0.
                  1.                    -25.2179E-09

 TOTAL            1.                    -25.2179E-09

 MINIMUM          0.                    -25.2179E-09
   AT X =                                 1.
 MAXIMUM          1.                      0.
   AT X =                                 0.


                     X                 sandwichXYData-2

                  0.                      0.
                  1.                    -23.6031E-09

 TOTAL            1.                    -23.6031E-09

 MINIMUM          0.                    -23.6031E-09
   AT X =                                 1.
 MAXIMUM          1.                      0.
   AT X =                                 0.
```

We wish to read the minimum value of the displacement of the second node since displacement is in the negative Z direction We first need to test for the presence of 'sandwichXYData-2' in order to make sure the read head has reached the second node, and then we look for the word 'MINIMUM'. We extract out this line, split it into a list, and the last element in this list is our required number.

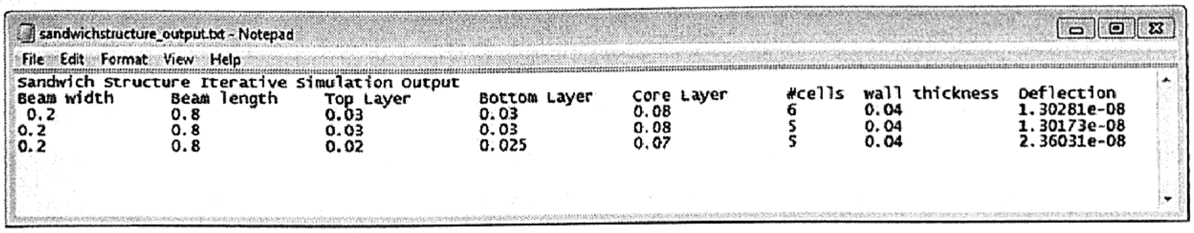
15.4.12 Write to output file

This block writes the output of each iteration to an output file.

```
    # -----------------------------------------------------------------------
    # Write this value as well as inputs to the output file

    file_output = open(reportxy_path + 'sandwichstructure_output.txt','a')
    if iteration_count == 1:
        file_output.write("Sandwich Structure Iterative Simulation Output" + "\n")
        file_output.write("Beam width" + "\t" + "Beam length" + "\t" + \
                          "Top Layer" + "\t" + "Bottom Layer" + "\t" +  \
                          "Core Layer" + "\t" + "#cells" + "\t" + \
                              "wall thickness" + "\t" + "Deflection" + "\n ")

    file_output.write(repr(sandwich_width) + "\t\t" + repr(sandwich_length) + \
                          "\t\t" + repr(top_layer_thickness) + "\t\t" + \
                          repr(bottom_layer_thickness) + "\t\t" + \
                          repr(core_layer_thickness) + "\t\t" + \
                          repr(no_of_core_cells) + "\t" +  \
                          repr(wall_thickness_core_cells) + "\t\t" + \
                          repr(-1 * float(max_displacement)) + "\n")
    file_output.close()

    iteration_count = iteration_count + 1

input_file.close()
```

These statements write all the parameters obtained from the input file as well as the displacement obtained from the analysis to an output file. If this is the first time the file is being written to (**iteration_count** = 1), then a title and column titles are added for presentation purposes.

This is what the final output might look like, depending of course on your inputs.


sandwichstructure_output.txt - Notepad
File Edit Format View Help

```
Sandwich Structure Iterative Simulation Output
Beam width      Beam length     Top Layer       Bottom Layer    Core Layer      #cells  wall thickness  Deflection
 0.2            0.8             0.03            0.03            0.08            6       0.04            1.30281e-08
0.2             0.8             0.03            0.03            0.08            5       0.04            1.30173e-08
0.2             0.8             0.02            0.025           0.07            5       0.04            2.36031e-08
```

15.5 Summary

In this script you parameterized a complex model and ran an optimization on it. You read parameters from an input file, and spit out results into an output file. You now have a good idea of how parameterization and optimization are carried out using Python scripts. The output file can of course be imported into software such as Microsoft Excel or Matlab where the trends can be analyzed for optimization purposes.

16

Explore an Output Database

16.1 Introduction

This chapter is going to introduce you to reading output databases, and gaining useful information from them. When you run an analysis in Abaqus, the data you request – the field and history outputs – as well as other information, such as the geometry of the part instance, is written to the output database (.odb) file. You might be required to extract some specific information from an odb as part of your analysis procedure. A script might be a more efficient then manually using the Abaqus/Viewer environment. In addition there are some tasks that are impossible to perform in the Viewer but possible through a script.

In this example we will experiment with the output database of the static truss analysis from Chapter 7 and the explicit dynamic truss analysis of Chapter 8. We will perform 4 tasks.

1) We will extract the stress field, and display a contour plot of one-half of its value. Each of the truss members will therefore appear to have only half of their original stress when viewed in Abaqus/Viewer. While this may not appear very useful, the purpose is to demonstrate how you can modify a field by performing a mathematical operation on it or a linear combination with another field. We will use the field output data of the static truss analysis for this.

2) We will extract information about the part instance used in the analysis, its nodes and elements, and find out which element and node experienced the maximum stress and displacement respectively. You saw an example of finding which element experiences the maximum stress in the plate optimization example (Chapter 13), but in that example you obtained this information by reading the

report file generated during post-processing. This time you will read the output database. You will also use the print command in a manner similar to the printf() command from C which allows you to format your printed output. We will use the field output data of the static truss analysis for this.

3) We will find out what regions of the part have history outputs available, what these history outputs are, and extract the history output data. You will also see how to find out which sets were defined in the model, and how to extract information about the history region these sets correspond to. History output information will be examined for both the output databases – the static truss analysis and the dynamic explicit truss analysis.

4) We will extract the material and section properties from the odb. We will also extract the entire material and section definitions from the static truss analysis odb and put them in a new Abaqus/CAE model for future use using some built-in methods provided by Abaqus.

In the process you will also learn of the various type of print statements, and how to format printed output to suit your needs (and also to make your code more readable). In addition you will discover the **hasattr()** and **type()** built-in functions offered by Python.

Performing these tasks will give you a good insight into working with Abaqus output databases using a Python script.

16.2 Methodology

For the first task, we will read in the stress [S] and displacement [U], both **FieldOutput** objects. We will divide the stresses by 2 to make them half their value, and leave the displacements at their present values. We will then create a new viewport window, set the primary variable to our new half stresses, and the deformed variable to the unchanged displacement, and plot these. We will also turn on element and node labels, so we can see the element and node numbers in the viewport to better understand what is going on in the next task.

For the second task, we will use the object model to examine field output values in the output database. Output databases consist of a very large amount of information, and this information is buried inside the object model at different levels –you have containers with information and more containers nested within them with additional information. To find the element with the maximum stress and the node with the maximum displacement, we will need to loop through all the elements and nodes examining their stress and displacement values respectively.

For the third task we will once again use the object model, but this time we will examine history output information.

For the fourth task we will use some methods provided by Abaqus to easily extract material and section information from an odb. We will create a new model file and place this information in it for demonstration purposes.

16.3 Before we begin – Odb Object Model

It will help for you to have some knowledge about the object model used in output databases before we examine the script.

An Odb must be opened using an **open()** method, passing the .odb file as a parameter. It will then persist until it is explicitly closed using a **close()** command. The **open()** command will differ based on whether you are inside the GUI - Abaqus/Viewer & Abaqus/CAE – or outside it.

From within the interactive products you use

```
import visualization
session.openOdb(name='file name with or without full path')
```

From outside the interactive products you use

```
from odbAccess import *
Odb_object = openOdb('file name with or without full path')
```

An output database consists of a repository of analysis steps. The name of the step a.k.a. its repository key, is the name assigned to the step when building the model (in Abaqus/CAE or an input file). So everything within the Odb is accessed by first accessing the steps as:

```
odb.steps['step name']
```

All the results data in an Odb is divided into two categories – Field Outputs and History Outputs. These are the big 'containers' of analysis result information, and they have objects and variables nested within them.

To access the field outputs, you first access the frame within the step. The frames are the increments of the analysis at which output is written to the Odb. The frames are stored as an **OdbFrameArray** object.

You access the frames using

```
odb.steps['step name'].frames[frame_index]
```

Where **frame_index** is an integer (0,1,2 …) or you can count backwards, such as -1 for last frame.

A **fieldOutputs** repository lies inside each frame. You access the field outputs within the frame as

```
Odb.steps['step name'].frames[frame_index].fieldOutputs[field_output_ key]
```

where **field_output_ key** is the name (or repository key) of the field output such as 'S', 'U' and 'UT'.

Each **FieldOutput** object contains a number of members, such as **values** which is a **FieldValueArray** object that represents the field data at that point.

```
Odb.steps['step name'].frames[frame_index].fieldOutputs[field_output_key].values
```

The **FieldValueArray** object has its own methods and members such as **elementLabel** and **nodeLabel** which are integers specifying the element and node labels of the element or node for which that field output exists. You would access these using statements of the form

```
odb.steps['step name'].frames[frame_index].fieldOutputs[field_output_key]
                                                 .values[value_index].elementLabel
```

and

```
odb.steps['step name'].frames[frame_index].fieldOutputs[field_output_variable_key]
                                                      .values[value_index].elementLabel
```

where **value_index** is an integer index that goes from 0 to the number of field output points for that field output variable.

The **FieldValueArray** also has members such as **mises** and **data** whose members contain the actual field output information. You would access these using statements such as

```
odb.steps['step name'].frames[frame_index].fieldOutputs['S']
                                                  .values[value_index].mises
```

for the Mises stress, and

```
odb.steps['step name'].frames[frame_index].fieldOutputs['U']
                                                .values[value_index].data[1]
```

for the displacement in the Y direction.

Another notable member of the **FieldValueArray** object is **instance** which is an **OdbInstance** object specifying which part the labels belong to. You would access it using

```
odb.steps['step name'].frames[frame_index].fieldOutputs[field_output_variable]
                                                .values[value_index].instance
```

And you can obtain more information, such as the name and type of the part using

```
odb.steps['step name'].frames[frame_index].fieldOutputs[field_output_variable]
                                           .values[value_index].instance.name
```

and

```
odb.steps['step name'].frames[frame_index].fieldOutputs[field_output_variable]
                                           .values[value_index].instance.type
```

You can even address the elements and nodes of the part instance from here using

```
odb.steps['step name'].frames[frame_index].fieldOutputs[field_output_variable]
                             .values[value_index].instance.elements[element_index]
```

and

```
odb.steps['step name'].frames[frame_index].fieldOutputs[field_output_variable]
                                          .values[value_index].nodes[node_index]
```

and access their members to get information such as their coordinates.

On the other hand to access the history outputs you examine the contents of **historyRegions** rather than **frames** from the **steps** repository. The history outputs for a point or a region exist in the **historyRegions** object. You would refer to it with statements of the form

```
odb.steps['step name'].historyRegions[history_region_key]
```

All of this should have given you a general idea of the output database object model. It contains many more members and methods which were not listed here, and the best way to discover them is by interrogating the object model (we will talk about interrogation at the end of the chapter).

16.4 How to run the script

Open a new model in Abaqus/CAE and run the script created for the static truss analysis using **File > Run Script...** The analysis will create an output database file 'TrussAnalysisJob.odb' and the script will open and display it in the Abaqus/Viewer viewport.

Then then open another new model in Abaqus/CAE and run the script created for the dynamic explicit truss analysis using **File > Run Script...** (It will be necessary to open a model to run the second script since both the scripts were originally written to be standalone and assume the existence of a default model 'Model-1' which they rename). The analysis will create an output database file 'TrussExplicitAnalysisJob.odb' and the script will open and display it in the Abaqus/Viewer viewport.

The reason both these scripts must be run is that they run the analysis and produce the output databases. The Odb exploration script in this example needs to access these output database files.

Once these scripts have been run, the Odb exploration script written in this chapter can be run using **File > Run Script..** either with those models still open in Abaqus/CAE, or in a new Abaqus/CAE model. (It does not make a difference since this script only accesses the .odb files and does not assume the existence or lack of any model in Abaqus/CAE).

16.5 Python Script

The following listing is the completed Python script. It is saved as the file **truss_odb.py**. You can run it by opening a new document in Abaqus/CAE (**File > New Model Database > With Standard/Explicit Model**) and running it with **File > Run Script...**

```
from abaqus import *
from abaqusConstants import *
import visualization

truss_Odb_Path = 'TrussAnalysisJob.odb'
trussOdb = session.openOdb(name=truss_Odb_Path)

truss_dynamic_Odb_Path = 'TrussExplicitAnalysisJob.odb'
trussDynamicOdb = session.openOdb(name=truss_dynamic_Odb_Path)

# -----------------------------------------------------------------------
# Change stresses on truss members to half their current value
# -----------------------------------------------------------------------

odb_stresses = trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S']
odb_displacements = trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['U']

half_odb_stresses = odb_stresses/2
same_odb_displacements = 1*odb_displacements

# Plot it in a new viewport window
truss_halfstress_viewport = session.Viewport(name='Truss with half the stresses')
truss_halfstress_viewport.setValues(displayedObject=trussOdb)

truss_halfstress_viewport.odbDisplay \
                              .setDeformedVariable(field=same_odb_displacements)
truss_halfstress_viewport.odbDisplay \
                            .setPrimaryVariable(field=half_odb_stresses,
                                               outputPosition=INTEGRATION_POINT,
                                               refinement=(INVARIANT, 'Mises'))
truss_halfstress_viewport.odbDisplay.display \
                                       .setValues(plotState=(CONTOURS_ON_DEF, ))
truss_halfstress_viewport.odbDisplay.commonOptions.setValues(nodeLabels=ON)
truss_halfstress_viewport.odbDisplay.commonOptions.setValues(elemLabels=ON)
truss_halfstress_viewport.setValues(origin=(0.0, 0.0), width=250, height=150)

# Change the deformation scale factor
truss_halfstress_viewport.odbDisplay.commonOptions.setValues(
deformationScaling=UNIFORM, uniformScaleFactor=45)

# -----------------------------------------------------------------------



# -----------------------------------------------------------------------
# Display information about the part, elements, nodes and maximum stress and
# displacement using the field output data
# -----------------------------------------------------------------------

print '\n\n'
print '************************************************'
print 'Some information obtained from the odb'
print '\n\n'

print 'The number of field output variables requested in the analysis is %d' \
                   %  len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs)


# The field outputs are in the form of dictionaries - key:value pairs
field_output_vars = ''
for j in range(len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs)):
    field_output_vars = field_output_vars + trussOdb.steps['Loading Step'] \
                                        .frames[-1].fieldOutputs.keys()[j] + ', '
print 'They are %s' % field_output_vars

print '\n\n'
print 'Stress field output [S] information is available at %d points.' \
        % len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'].values)



# Since we have only one part, all the stress field outputs are available for
# this part only
print 'The name of the part instance for which stress field outputs are ' + \
      'available is %s whose modeling space is %s and is of type %s.' \
      % ( trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
                  .values[0].instance.name, trussOdb.steps['Loading Step'] \
                  .frames[-1].fieldOutputs['S'].values[0].instance.embeddedSpace,
          trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
                  .values[0].instance.type )

print '\n\n'
print 'The part has %d elements and %d nodes.' \
        % ( len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
                                              .values[0].instance.elements),
            len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
                                              .values[0].instance.nodes) )

# Find element with maximum stress
max_stress = 0
for k in range(len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
               .values[0].instance.elements)):  #can use 0 or any other index too
    element_mises_stress = trussOdb.steps['Loading Step'].frames[-1] \
                                             .fieldOutputs['S'].values[k].mises
    if element_mises_stress > max_stress:
        max_stress_element_label  = trussOdb.steps['Loading Step'].frames[-1] \
                                        .fieldOutputs['S'].values[k].elementLabel
        field_output_stress_object_index = k
        max_stress = element_mises_stress

print '\n\n'
print 'The maximum mises stress is %(maximum stress)E and it is on ' + \
      'element %(element with maximum stress)d' % {"maximum stress": max_stress,
                          "element with maximum stress": max_stress_element_label}
```

```
print 'This element is of type %s' % trussOdb.steps['Loading Step'].frames[-1] \
                    .fieldOutputs['S'].values[field_output_stress_object_index] \
                    .instance.elements[max_stress_element_label - 1].type

print 'It connects node %d and node %d' \
       % (trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
              .values[field_output_stress_object_index].instance \
              .elements[max_stress_element_label - 1].connectivity[0],
          trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
              .values[field_output_stress_object_index].instance \
              .elements[max_stress_element_label - 1].connectivity[1] )


# Find node with maximum displacement in Y direction
max_displacement = 0
for x in range(len(trussOdb.steps['Loading Step'].frames[-1] \
                                    .fieldOutputs['U'].values[0].instance.nodes)):
    node_y_displacement = abs(trussOdb.steps['Loading Step'].frames[-1] \
                                           .fieldOutputs['U'].values[x].data[1])
    if node_y_displacement > max_displacement :
        max_disp_node_label = trussOdb.steps['Loading Step'].frames[-1] \
                                           .fieldOutputs['U'].values[x].nodeLabel
        field_output_disp_object_index = x
        max_displacement = node_y_displacement

print '\n\n'
print 'The node with the maximum Y displacement is %d' % max_disp_node_label
print 'The Y component of the displacement is %f' % max_displacement
print 'The magnitude of the displacement (length of disp vector) is %.5f' \
       % trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['U'] \
                              .values[field_output_disp_object_index].magnitude

print '\n\n'
print '=================================================='


# ---------------------------------------------------------------------
# Display history output information for static truss job
# ---------------------------------------------------------------------

print 'There is/are %d history region(s) in static truss odb.' \
       % len(trussOdb.steps['Loading Step'].historyRegions)

print 'In the odb its key is : %s' % trussOdb.steps['Loading Step'] \
                                       .historyRegions.keys()[0]

print 'In the odb its name is : %s' % trussOdb.steps['Loading Step'] \
                                       .historyRegions.values()[0].name

print 'Its description is : %s' % trussOdb.steps['Loading Step'] \
                                       .historyRegions.values()[0].description
```

```
print 'Its position is : %s' % trussOdb.steps['Loading Step'] \
                                          .historyRegions.values()[0].position


history_output_vars = ''
for y in range(len(trussOdb.steps['Loading Step'] \
                                    .historyRegions.values()[0].historyOutputs)):
    history_output_vars = history_output_vars + \
                        trussOdb.steps['Loading Step'] \
                       .historyRegions.values()[0].historyOutputs.keys()[y] + ', '
print 'The history outputs available are %s' % history_output_vars


print 'At time = %f Strain energy (ALLSE) is %f' \
       % (trussOdb.steps['Loading Step'].historyRegions.values()[0] \
                                          .historyOutputs['ALLSE'].data[0][0],
          trussOdb.steps['Loading Step'].historyRegions.values()[0] \
                                          .historyOutputs['ALLSE'].data[0][1])

print 'At time = %f Strain energy (ALLSE) is %f' \
       % (trussOdb.steps['Loading Step'].historyRegions.values()[0] \
                                            .historyOutputs['ALLSE'].data[1][0],
          trussOdb.steps['Loading Step'].historyRegions.values()[0] \
                                            .historyOutputs['ALLSE'].data[1][1])

print '\n\n'


# ----------------------------------------------------------------------
# Display history output information for dynamic explicit truss job
# ----------------------------------------------------------------------

print 'There is/are %d history region(s) in dynamic explicit truss odb' \
       % len(trussDynamicOdb.steps['Loading Step'].historyRegions)

print 'In the odb their keys are %s and %s' \
       % (trussDynamicOdb.steps['Loading Step'].historyRegions.keys()[0],
          trussDynamicOdb.steps['Loading Step'].historyRegions.keys()[1])

print 'The 2 sets in the model are %s and %s' \
       % (trussDynamicOdb.rootAssembly.nodeSets.keys()[0],
          trussDynamicOdb.rootAssembly.nodeSets.keys()[1])

print 'The label of the node which makes up the set END POINT SET is %d' \
       % trussDynamicOdb.rootAssembly.nodeSets['END POINT SET'].nodes[0][0].label

print '\n\n'


# Figure out which one has node 3
for z in trussDynamicOdb.steps['Loading Step'].historyRegions.keys():
    if z.find('Node') > -1:
        if z.split('.')[1] == '3':
            correct_key = z

print 'Here is the history output for U2 data at END POINT SET'

for m in range(len(trussDynamicOdb.steps['Loading Step'] \
                        .historyRegions[correct_key].historyOutputs['U2'].data)):
    time = trussDynamicOdb.steps['Loading Step'].historyRegions[correct_key] \
                                                .historyOutputs['U2'].data[m][0]
    Ydisp = trussDynamicOdb.steps['Loading Step'].historyRegions[correct_key] \
                                                .historyOutputs['U2'].data[m][1]
    print 'At time = %f Y-displacement = %f' % (time, Ydisp)


# ---------------------------------------------------------------------
# Extract part, material and section informatio from the odb
# ---------------------------------------------------------------------

from odbMaterial import *
from odbSection import *

print '\n\n'
print '=================================================='

print 'The first (and only) part instance in the model is %s ' \
      % trussOdb.rootAssembly.instances.keys()[0]

print 'The first (and only) material in the model is %s ' \
      % trussOdb.materials.keys()[0]

steel_material = trussOdb.materials.values()[0]

if hasattr(steel_material, 'elastic'):
    print 'The material is elastic'
else:
    print 'The material is not elastic'

print 'Its density is %f' % steel_material.density.table[0][0]
print 'Its Young\'s modulus is %.2E and its Poisson\'s ratio is %f ' \
      % (steel_material.elastic.table[0][0], steel_material.elastic.table[0][1])

print 'The first (and only) section in the model is %s ' \
      % trussOdb.sections.keys()[0]

truss_section = trussOdb.sections.values()[0]
if type(truss_section) == HomogeneousSolidSectionType:
    print 'The section is of type HomogeneousSolidSection'
else:
    print 'The section is not of type HomegeneousSolidSection'
```

```
# ----------------------------------------------------------------------
# Extract material and section definitions from the odb and place in a model
# ----------------------------------------------------------------------

mdb.Model(name='Model To Extract Odb Info', modelType=STANDARD_EXPLICIT)
mdb.models['Model To Extract Odb Info'].materialsFromOdb('TrussAnalysisJob.odb')
mdb.models['Model To Extract Odb Info'].sectionsFromOdb('TrussAnalysisJob.odb')

# ----------------------------------------------------------------------

# Close the output databases
trussOdb.close()
trussDynamicOdb.close()
```

16.5.1 Initialization

The following statements import the required modules and open the odb

```
from abaqus import *
from abaqusConstants import *
import visualization

truss_Odb_Path = 'TrussAnalysisJob.odb'
trussOdb = session.openOdb(name=truss_Odb_Path)

truss_dynamic_Odb_Path = 'TrussExplicitAnalysisJob.odb'
trussDynamicOdb = session.openOdb(name=truss_dynamic_Odb_Path)
```

You import the visualization module so that you can use the **session.openOdb()** method to open the .odb file. The **openOdb()** method has been used in a number of previous examples, and was first encountered and discussed in the Cantilever Beam, Section 4.3.14 on page 89. All we are doing here is opening the .odb file named 'TrussAnalysisJob.odb' which was created by the static truss analysis as well as the .odb file named 'TrussExplicitAnalysisJob.odb' which was created by the dynamic explicit truss analysis.

16.5.2 Mathematical operations on field data

The following statements perform the first task of displaying half the stresses as a color contour on the truss members

```
# ----------------------------------------------------------------------
# Change stresses on truss members to half their current value
# ----------------------------------------------------------------------

odb_stresses = trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S']
```

```
odb_displacements = trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['U']

half_odb_stresses = odb_stresses/2
same_odb_displacements = 1*odb_displacements

# Plot it in a new viewport window
truss_halfstress_viewport = session.Viewport(name='Truss with half the stresses')
truss_halfstress_viewport.setValues(displayedObject=trussOdb)

truss_halfstress_viewport.odbDisplay \
                               .setDeformedVariable(field=same_odb_displacements)
truss_halfstress_viewport.odbDisplay \
                             .setPrimaryVariable(field=half_odb_stresses,
                                                 outputPosition=INTEGRATION_POINT,
                                                 refinement=(INVARIANT, 'Mises'))
truss_halfstress_viewport.odbDisplay.display \
                                          .setValues(plotState=(CONTOURS_ON_DEF, ))
truss_halfstress_viewport.odbDisplay.commonOptions.setValues(nodeLabels=ON)
truss_halfstress_viewport.odbDisplay.commonOptions.setValues(elemLabels=ON)
truss_halfstress_viewport.setValues(origin=(0.0, 0.0), width=250, height=150)

# Change the deformation scale factor
truss_halfstress_viewport.odbDisplay.commonOptions \
                .setValues(deformationScaling=UNIFORM, uniformScaleFactor=45)
```

`odb_stresses = trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S']`

This statement accesses the field output quantity 'S' that was requested when building the model. It assigns the **FieldOutput** object to a variable **odb_stresses** which we can work with / modify. The analysis step in the static truss analysis was named 'Loading Step', hence this is the key in the **steps** repository. The frame is referred to as **frames[-1]** indicating the last frame in this step. Since this analysis was carried out with an initial step time of 1.0 and the total step time was also 1.0, this means that only one increment was performed by the Abaqus solver. Therefore there are 2 frames, frame at time 0.0, and frame at time 1.0 representing the first increment. Therefore we could use [1] instead of [-1] and rewrite the statement as

```
odb_stresses = trussOdb.steps['Loading Step'].frames[1].fieldOutputs['S']
```

Which one you choose is a matter of personal preference. Also you see the field output variable referred to with the key 'S' because this is the key used by Abaqus to represent stress field output requests, and it was used by our truss analysis script to request stress outputs.

`odb_displacements = trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['U']`

This is similar to the previous statement except that we are requesting and storing the field output for U – translations and rotations.

```
half_odb_stresses = odb_stresses/2
same_odb_displacements = 1*odb_displacements
```

Here the two **FieldOutput** variables **odb_stresses** and **odb_displacements** created in the previous two statements are subjected to a basic mathematical operation and assigned to new variables. The stresses are halved and assigned to the variable **half_odb_stresses**. This demonstrates a powerful feature – you can create new field data by performing basic mathematical operations on existing field data. Here we divide by 2 using the '/' operator. Other operators that can be used are +,- and *, trigonometric functions **cos()**, **sin()**, **tan()**, **acos()**, **asin()**, **atan()**, and other commonly used functions such as **abs()**, **exp()**, **log()**, **exp10()**, **log10()**, **power()**, **degreeToRadian()** and **radianToDegree()**.

In fact it is also possible to use linear combinations of two fields to create a third. For example you could add the stresses from two different odbs to create a new one as

```
new_odb_stresses = odbs_stresses_from_odb1 + odb_stresses_from_odb2
```

You could do the same with displacements

```
new_odb_displacements = odbs_displacements_from_odb1 +
                        odb_displacements_from_odb2
```

However the fields must be of the same time, so stresses and displacements cannot be combined as

```
new_odb_field = odbs_stresses_from_odb1 + odb_displacements_from_odb2
```

because stress data exists at integration points (**INTEGRATION_POINT**) and displacement data is nodal (**ELEMENT_NODAL**) and these cannot be combined. This aside from the fact that it would not make sense to combine them anyway.

You are probably wondering why we multiplied **odb_displacements** by 1 to create **same_odb_displacements**. Why not write it as follows? :

```
same_odb_displacements = odb_displacements
```

In fact why not just use the original variable? Well, this is not possible due to the way Abaqus works internally.

To answer the second question, when we create a new field data from an old one, we are only allowed to use all new field data or all old field data in any given viewport window. So if you were to plot the new stress data (half the original) in a viewport window, you would not be able to plot the old displacement data in the same viewport. Hence we must create a new field data variable.

To answer the first question – Abaqus does recognize the original field data as a new one by assigning it to a new variable alone, it is only when some sort of mathematical operation is carried out on it that it registers this fact. Hence we multiply it by 1.

Why does Abaqus work like this? To be honest, I'm not sure. Maybe the folks who develop Abaqus have a reason for this, or maybe this is a quirk or bug that will not exist in future versions of Abaqus. However for most of us the important thing is getting the job done, and it appears that multiplying by 1, and therefore performing a mathematical operation, forces Abaqus to internally create a new FieldOutput object in a different way from a simple assignment statement.

```
# Plot it in a new viewport window
truss_halfstress_viewport = session.Viewport(name='Truss with half the stresses')
truss_halfstress_viewport.setValues(displayedObject=trussOdb)

truss_halfstress_viewport.odbDisplay \
                              .setDeformedVariable(field=same_odb_displacements)
truss_halfstress_viewport.odbDisplay \
                            .setPrimaryVariable(field=half_odb_stresses,
                                                outputPosition=INTEGRATION_POINT,
                                                refinement=(INVARIANT, 'Mises'))
truss_halfstress_viewport.odbDisplay.display \
                                      .setValues(plotState=(CONTOURS_ON_DEF, ))
truss_halfstress_viewport.odbDisplay.commonOptions.setValues(nodeLabels=ON)
truss_halfstress_viewport.odbDisplay.commonOptions.setValues(elemLabels=ON)
truss_halfstress_viewport.setValues(origin=(0.0, 0.0), width=250, height=150)
```

Almost all of these statements are similar to the ones used in the static truss analysis script, and were explained in section 7.4.15 on page 142. The only new method used here is **setDeformedVariable()** which is similar to **setPrimaryVariable()** except that it specifies the field output variable or **FieldOutput** object to use when displaying the deformed shape. Either the field output variable **variableLabel** or the **FieldOutput** object **field** must be provided as an argument.

```
# Change the deformation scale factor
truss_halfstress_viewport.odbDisplay.commonOptions \
               .setValues(deformationScaling=UNIFORM, uniformScaleFactor=45)
```

You've used **odbDisplay.commonOptions.setValues()** numerous times before, in fact it is used in the previous statements as well, the difference here is the use of the parameters **deformationScaling** and **uniformScaleFactor** to scale the visible displacement in the viewport by 45 times its actual value. **deformationScaling** requires a SymbolicConstant which can be **AUTO**, **UNIFORM** and **NONUNIFORM**. **uniformScaleFactor** is the constant factor by which to multiply the deformation when **deformationScaling** is set to **UNIFORM**.

16.5.3 Access information about part, nodes, elements, stresses, displacements

The following statements perform the second task of querying the odb for information on the part, nodes, elements and finding the maximum stress and displacement

```
# -----------------------------------------------------------------------
# Display information about the part, elements, nodes and maximum stress and
# displacement using the field output data
# -----------------------------------------------------------------------

print '\n\n'
print '**************************************************'
print 'Some information obtained from the odb'
print '\n\n'

print 'The number of field output variables requested in the analysis is %d' \
                    %  len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs)


# The field outputs are in the form of dictionaries - key:value pairs
field_output_vars = ''
for j in range(len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs)):
    field_output_vars = field_output_vars + trussOdb.steps['Loading Step'] \
                                        .frames[-1].fieldOutputs.keys()[j] + ', '
print 'They are %s' % field_output_vars

print '\n\n'
print 'Stress field output [S] information is available at %d points.' \
         % len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'].values)



# Since we have only one part, all the stress field outputs are available for
# this part only
print 'The name of the part instance for which stress field outputs are ' + \
      'available is %s whose modeling space is %s and is of type %s.' \
       % ( trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
                   .values[0].instance.name, trussOdb.steps['Loading Step'] \
                   .frames[-1].fieldOutputs['S'].values[0].instance.embeddedSpace,
            trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
                    .values[0].instance.type )


print '\n\n'
print 'The part has %d elements and %d nodes.' \
        % ( len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
                                                .values[0].instance.elements),
            len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
                                                .values[0].instance.nodes) )


# Find element with maximum stress
max_stress = 0
for k in range(len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
               .values[0].instance.elements)):  #can use 0 or any other index too
    element_mises_stress = trussOdb.steps['Loading Step'].frames[-1] \
                                               .fieldOutputs['S'].values[k].mises
    if element_mises_stress > max_stress:
        max_stress_element_label  = trussOdb.steps['Loading Step'].frames[-1] \
                                        .fieldOutputs['S'].values[k].elementLabel
        field_output_stress_object_index = k
        max_stress = element_mises_stress

print '\n\n'
print 'The maximum mises stress is %(maximum stress)E and it is on ' + \
      'element %(element with maximum stress)d' % {"maximum stress": max_stress,
                        "element with maximum stress": max_stress_element_label}

print 'This element is of type %s' % trussOdb.steps['Loading Step'].frames[-1] \
                    .fieldOutputs['S'].values[field_output_stress_object_index] \
                    .instance.elements[max_stress_element_label - 1].type
print 'It connects node %d and node %d' \
       % (trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
              .values[field_output_stress_object_index].instance \
              .elements[max_stress_element_label - 1].connectivity[0],
          trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
              .values[field_output_stress_object_index].instance \
              .elements[max_stress_element_label - 1].connectivity[1] )


# Find node with maximum displacement in Y direction
max_displacement = 0
for x in range(len(trussOdb.steps['Loading Step'].frames[-1] \
                                .fieldOutputs['U'].values[0].instance.nodes)):
    node_y_displacement = abs(trussOdb.steps['Loading Step'].frames[-1] \
                                       .fieldOutputs['U'].values[x].data[1])
    if node_y_displacement > max_displacement :
        max_disp_node_label = trussOdb.steps['Loading Step'].frames[-1] \
                                       .fieldOutputs['U'].values[x].nodeLabel
        field_output_disp_object_index = x
        max_displacement = node_y_displacement

print '\n\n'
print 'The node with the maximum Y displacement is %d' % max_disp_node_label
print 'The Y component of the displacement is %f' % max_displacement
print 'The magnitude of the displacement (length of disp vector) is %.5f' \
      % trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['U'] \
                             .values[field_output_disp_object_index].magnitude

print '\n\n'
print '=============================================='
```

```
print '\n\n'
print '**********************************************'
print 'Some information obtained from the odb'
print '\n\n'
```

This is certainly not the first time we're using the **print** statement to print statements to the message area. I should point out however that this is the most basic format of the **print** statement and we will use a couple of others in this section. For the sake of convenience I will call this print statement type 1.

Type 1 print statement: `print 'some String within quotes'`

The '\n' character has special meaning to Python, namely that it forces a carriage return – the equivalent of hitting the 'Enter' key on the keyboard. Since every **print** statement will print on a new line, it doesn't really make much sense to have a single **\n** at the end of the print statement. However if we use two **\n**'s then Python will leave a line before the next statement.

```
print 'The number of field output variables requested in the analysis is %d' \
                    %  len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs)
```

Here we use another form of the print statement. It is more similar to the printf() command from C. It allows you to format the output in a particular manner by using the modulo (**%**) symbol, which is known as the String formatting or interpolation operator, followed by a character. So **%d** indicates a signed decimal number, **%f** indicates a float and so on. The following are the available format specifiers as listed in the Python documentation.

'd'	Signed integer decimal.
'i'	Signed integer decimal.
'o'	Signed octal value.
'u'	Obsolete type – it is identical to 'd'.
'x'	Signed hexadecimal (lowercase).

'X'	Signed hexadecimal (uppercase).
'e'	Floating point exponential format (lowercase).
'E'	Floating point exponential format (uppercase).
'f'	Floating point decimal format.
'F'	Floating point decimal format.
'g'	Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.
'G'	Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.
'c'	Single character (accepts integer or single character String).
'r'	String (converts any Python object using repr()).
's'	String (converts any Python object using str()).
'%'	No argument is converted, results in a '%' character in the result.

The values that will replace the String format placeholder must be specified after another **%** symbol. If a single value is present, it can be specified as a single non-tuple object as is done in this case. This is what I call a type 2 print statement

Type 2 print statement: `print ‘String  with %x formatting’ % var1`

The field data variables exist as key:value pairs for each frame of each step in the output database. The **len()** command is used to find the number of field output values stored for the last frame of the step called ‘Loading Step’ as

```
len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs)
```

```
# The field outputs are in the form of dictionaries - key:value pairs
field_output_vars = ''
for j in range(len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs)):
    field_output_vars = field_output_vars + trussOdb.steps['Loading Step'] \
                                        .frames[-1].fieldOutputs.keys()[j] + ', '
print 'They are %s' % field_output_vars
```

Here we print out the names of the field output variables. As mentioned previously, the field data variables exist in the **fieldOutputs** repository as key:value pairs. Here we use the **keys()** method to access them. In addition we attach an index to extract the key at that index location as

```
trussOdb.steps['Loading Step'].frames[-1].fieldOutputs.keys()[j]
```

This way we can use a **for** loop to extract each of the names (or keys) and store them in the variable **field_output_vars** with commas separating them. We later print this variable out with a print statement.

```
print 'Stress field output [S] information is available at %d points.' \
        % len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'].values)
```

We use the key 'S' to refer to the corresponding value in the **fieldOutputs** repository. **fieldOutputs['S'].values** gives us a list of values, hence **len()** can be used to get the number of field output values, which will be the same as the number of points at which field output data is available.

```
# Since we have only one part, all the stress field outputs are available for
# this part only
print 'The name of the part instance for which stress field outputs are ' + \
      'available is %s whose modeling space is %s and is of type %s.' \
       % ( trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
                   .values[0].instance.name, trussOdb.steps['Loading Step'] \
                   .frames[-1].fieldOutputs['S'].values[0].instance.embeddedSpace,
           trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
                   .values[0].instance.type )
```

The instance for which the field output is written is accessed with

```
trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'].values[0].instance
```

Its name, modeling space and type are accessed using

```
trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'].values[0].instance.name
```

```
trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S']
                                                  .values[0].instance.embeddedSpace
```

and

```
trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'].values[0].instance.type
```

Notice the difference in the print statement used here. Since there are multiple values being passed to the print statement (there are multiple String formatting operators), the values or variables themselves have to be in a tuple, i.e. within ().

Type 3 print statement: `print 'String  with %x and %y formatting' % (var1, var2)`

```
print 'The part has %d elements and %d nodes.' \
        % ( len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
                                                    .values[0].instance.elements),
            len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
                                                    .values[0].instance.nodes) )
```

The above statement refers to the nodes and elements of the part instance, and counts them using the **len()** function to give us the number of elements and nodes in the part.

Note that we used the index 0 in **fieldOutputs['S'].values[0].isntance.nodes**. We could have used any other index here between 0 and the total number of field outputs. This is because we have only one part hence the **instance** member of all of them refers to the same part.

In fact, we know that the stress field output data occurs on each element, hence the number of stress field outputs is the same as the number of elements. So we could instead have rewritten

```
len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S']
                                                    .values[0].instance.elements)
```

as

```
len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'].values)
```

However I avoided this to maintain the readability of the code and so as not to confuse you.

```
for k in range(len(trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
               .values[0].instance.elements)):  #can use 0 or any other index too
    element_mises_stress = trussOdb.steps['Loading Step'].frames[-1] \
                                               .fieldOutputs['S'].values[k].mises
    if element_mises_stress > max_stress:
        max_stress_element_label  = trussOdb.steps['Loading Step'].frames[-1] \
                                        .fieldOutputs['S'].values[k].elementLabel
        field_output_stress_object_index = k
        max_stress = element_mises_stress
```

In order to find the maximum stress we iterate through each element of the part and look at its Mises stress. The Mises stress is a member of the values **FieldValueArray**.

Once the maximum stress has been found, we get the element label using

```
trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'].values[k].elementLabel
```

because the element label is the same as its index in the elements array.

We store the iteration counter value for this element in a variable **field_output_stress_object_index** to use in a subsequent statement.

```
print 'The maximum mises stress is %(maximum stress)E and it is on ' + \
      'element %(element with maximum stress)d' % {"maximum stress": max_stress,
                        "element with maximum stress": max_stress_element_label}
```

This statement prints out the maximum stress and the element with the maximum stress, both of which were found in previous statements.

Notice that this is another type of print statement. Here the variables/values are expressed as dictionaries with key:value pairs. The key is placed in parenthesis after the % and then followed by the formatting specifying character. The dictionaries are written with curly braces {} separated by commas. I like to call this the type 4 print statement

Type 4 print statement: `print 'String  with %(key1)x and %(key2)y formatting' %`
`{"key 1": val1, "key 2": val2}`

By using keys, it might be easier to read the code, especially if there are a lot of String formatting operators within the print statement. More importantly, the variables no longer have to be listed in the same order as the String formatting operators since it is the key that identifies which value to use. Hence the statement can be written as:

```
print 'The maximum mises stress is %(maximum stress)E and it is on ' + \
      'element %(element with maximum stress)d' % {"maximum stress": max_stress,
                        "element with maximum stress": max_stress_element_label}

print 'This element is of type %s' % trussOdb.steps['Loading Step'].frames[-1] \
                    .fieldOutputs['S'].values[field_output_stress_object_index] \
                    .instance.elements[max_stress_element_label - 1].type
```

We use the variable **field_output_stress_object_index** obtained from the **for** loop to refer to the stress field output which we identified as having the largest Mises stress. We refer to the correct element of the instance by subtracting 1 from **max_stress_element_label** because the element label is always 1 more than its index in the elements list (since lists count from 0 upward).

Once again I should point out that since all the stress outputs are for the same part, we could have used any index for values[] between 0 and one minus the total number of stress field outputs. So the statement could instead have been written as

```
print 'This element is of type %s' % trussOdb.steps['Loading Step'].frames[-1] \
      .fieldOutputs['S'].values[0].instance.elements[max_stress_elemt_label -1].type
```

Or

```
print 'This element is of type %s' % trussOdb.steps['Loading Step'].frames[-1] \
   .fieldOutputs['S'].values[9].instance.elements[max_stress_elemt_label -1].type
```

However this may be a little confusing when reading code.

```
print 'It connects node %d and node %d' \
       % (trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
              .values[field_output_stress_object_index].instance \
              .elements[max_stress_element_label - 1].connectivity[0],
          trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['S'] \
              .values[field_output_stress_object_index].instance \
              .elements[max_stress_element_label - 1].connectivity[1] )
```

The **emenents** have a **connectivity** member which lists the labels of the nodes that the element is connected to. Since we are dealing with 2D truss members, each element has 2 connectivities.

```
max_displacement = 0
for x in range(len(trussOdb.steps['Loading Step'].frames[-1] \
                                   .fieldOutputs['U'].values[0].instance.nodes)):
    node_y_displacement = abs(trussOdb.steps['Loading Step'].frames[-1] \
                                          .fieldOutputs['U'].values[x].data[1])
    if node_y_displacement > max_displacement :
        max_disp_node_label = trussOdb.steps['Loading Step'].frames[-1] \
                                          .fieldOutputs['U'].values[x].nodeLabel
        field_output_disp_object_index = x
        max_displacement = node_y_displacement
```

Similar to the one used for finding the maximum stress, this **for** loop is used to find the maximum displacement and store some information about it to use in subsequent **print** statements. Notice that we are not investigating **fieldOutputs['U']** which stores information about translations and rotations, and the **data** member of **values[]** is referred to which stores the X, Y and Z displacements as **data[0]**, **data[1]** and **data[2]**. Since we are only looking at **data[1]** as the condition for the **for** loop, we are actually finding the node with the maximum Y-displacement.

```
print 'The Y component of the displacement is %f' % max_displacement
```

This prints the maximum displacement out to the screen. **%f** is used to print a float.

```
print 'The magnitude of the displacement (length of disp vector) is %.5f' \
        % trussOdb.steps['Loading Step'].frames[-1].fieldOutputs['U'] \
                              .values[field_output_disp_object_index].magnitude
```

The magnitude of the displacement (not just the Y-component) is also present in the odb as the **magnitude** member of each field output in **values[]**.

Notice the String formatting operator **%.5f**. This means the value to be printed is a float, however only print 5 digits after the decimal point and round up if necessary. With only a **%f**, the stress magnitude is printed with 6 digits after the decimal place, but with it we limit it to 5 digits.

16.5.4 Display history output information for static truss analysis

The following statements query the odb of the static truss analysis for history output information

```
# -------------------------------------------------------------------------
# Display history output information for static truss job
# -------------------------------------------------------------------------

print 'There is/are %d history region(s) in static truss odb.' \
       % len(trussOdb.steps['Loading Step'].historyRegions)

print 'In the odb its key is : %s' % trussOdb.steps['Loading Step'] \
                                          .historyRegions.keys()[0]

print 'In the odb its name is : %s' % trussOdb.steps['Loading Step'] \
                                          .historyRegions.values()[0].name

print 'Its description is : %s' % trussOdb.steps['Loading Step'] \
                                          .historyRegions.values()[0].description

print 'Its position is : %s' % trussOdb.steps['Loading Step'] \
                                          .historyRegions.values()[0].position


history_output_vars = ''
for y in range(len(trussOdb.steps['Loading Step'] \
                                    .historyRegions.values()[0].historyOutputs)):
    history_output_vars = history_output_vars + \
                         trussOdb.steps['Loading Step'] \
                        .historyRegions.values()[0].historyOutputs.keys()[y] + ', '
print 'The history outputs available are %s' % history_output_vars


print 'At time = %f Strain energy (ALLSE) is %f' \
       % (trussOdb.steps['Loading Step'].historyRegions.values()[0] \
                                          .historyOutputs['ALLSE'].data[0][0],
          trussOdb.steps['Loading Step'].historyRegions.values()[0] \
                                                .historyOutputs['ALLSE'].data[0][1])

print 'At time = %f Strain energy (ALLSE) is %f' \
      % (trussOdb.steps['Loading Step'].historyRegions.values()[0] \
                                                  .historyOutputs['ALLSE'].data[1][0],
          trussOdb.steps['Loading Step'].historyRegions.values()[0] \
                                                  .historyOutputs['ALLSE'].data[1][1])

print '\n\n'
```

```
print 'There is/are %d history region(s) in static truss odb.' \
      % len(trussOdb.steps['Loading Step'].historyRegions)
```

`trussOdb.steps['Loading Step'].historyRegions` returns the **HistoryRegion** objects, of which there will be one per history output request. **len()** gives us the number of history output regions..

```
print 'In the odb its key is : %s' % trussOdb.steps['Loading Step'] \
                                          .historyRegions.keys()[0]
```

`trussOdb.steps['Loading Step'].historyRegions.keys()` contains the keys of all the HistoryRegion objects. We already know (since we created the model ourselves) that there is one **HistoryRegion** object, and we refer to the its key with an index of [0].

```
print 'In the odb its name is : %s' % trussOdb.steps['Loading Step'] \
                                          .historyRegions.values()[0].name
```

We refer to the **HistoryRegion** objects in the repository using **values()**. We refer to the first one using the index of [0] and we access the **name** property to get the name which is the same as the repository key.

Since the previous statement informs us that the **key()** for this **HistoryRegion** object is 'Assembly ASSEMBLY' we can also write this statement as

```
print 'In the odb its name is : %s' % trussOdb.steps['Loading Step'] \
                                          .historyRegions['Assembly ASSEMBLY'].name
```

```
print 'Its description is : %s' % trussOdb.steps['Loading Step'] \
                                          .historyRegions.values()[0].description
```

The **description** property of the **HistoryOutput** object returns an Abaqus supplied description for the history output request.

```
print 'Its position is : %s' % trussOdb.steps['Loading Step'] \
                                        .historyRegions.values()[0].position
```

The **position** property of the **HistoryRegion** object provides a SymbolicConstant specifying the position of the history output. Possible values you will encounter are **NODAL**, **INTEGRATION_POINT**, **WHOLE_ELEMENT**, **WHOLE_REGION** and **WHOLE_MODEL**.

```
history_output_vars = ''
for y in range(len(trussOdb.steps['Loading Step'] \
                                   .historyRegions.values()[0].historyOutputs)):
    history_output_vars = history_output_vars + \
                         trussOdb.steps['Loading Step'] \
                        .historyRegions.values()[0].historyOutputs.keys()[y] + ', '
print 'The history outputs available are %s' % history_output_vars
```

The **for** loop iterates over the number of **HistoryOutput** objects in the **historyOutputs** repository. Each **HistoryRegion** object has a number of **HistoryOutput** objects because a number of history output variables are selected for each history output request (such as ALLE, LLWK, ETOTAL etc) and these are all requested from the region. For each of them this loop extracts the repository key and collects them in a comma separated String for printing.

```
print 'At time = %f Strain energy (ALLSE) is %f' \
       % (trussOdb.steps['Loading Step'].historyRegions.values()[0] \
                                         .historyOutputs['ALLSE'].data[0][0],
          trussOdb.steps['Loading Step'].historyRegions.values()[0] \
                                         .historyOutputs['ALLSE'].data[0][1])

print 'At time = %f Strain energy (ALLSE) is %f' \
       % (trussOdb.steps['Loading Step'].historyRegions.values()[0] \
                                           .historyOutputs['ALLSE'].data[1][0],
          trussOdb.steps['Loading Step'].historyRegions.values()[0] \
                                           .historyOutputs['ALLSE'].data[1][1])
```

The strain energy history output variable ALLSE is available in a list of tuples of (time, value) using **trussOdb.steps['Loading Step'].historyRegions.values()[0] .historyOut puts['ALLSE'].data.** In our odb there are two data points or tuples, one at time = 0 and the other at time = 1 but in some other analysis where history output requests occur many times there will be many more data points. We use the first index of [0] for the first point, and second index of [0] or [1] for the time and value respectively at this point. Similarly we repeat for the second point.

If you were to plot the history output for ALSE in Abaqus/Viewer, these are the two points that give you the plot you see.

16.5.5 Display history output information for dynamic explicit truss analysis

The following statements query the odb of the dynamic explicit truss analysis for history output information

```
# -------------------------------------------------------------------------
# Display history output information for dynamic explicit truss job
# -------------------------------------------------------------------------

print 'There is/are %d history region(s) in dynamic explicit truss odb' \
       % len(trussDynamicOdb.steps['Loading Step'].historyRegions)

print 'In the odb their keys are %s and %s' \
       % (trussDynamicOdb.steps['Loading Step'].historyRegions.keys()[0],
          trussDynamicOdb.steps['Loading Step'].historyRegions.keys()[1])

print 'The 2 sets in the model are %s and %s' \
       % (trussDynamicOdb.rootAssembly.nodeSets.keys()[0],
          trussDynamicOdb.rootAssembly.nodeSets.keys()[1])

print 'The label of the node which makes up the set END POINT SET is %d' \
       % trussDynamicOdb.rootAssembly.nodeSets['END POINT SET'].nodes[0][0].label

print '\n\n'


# Figure out which one has node 3
for z in trussDynamicOdb.steps['Loading Step'].historyRegions.keys():
    if z.find('Node') > -1:
        if z.split('.')[1] == '3':
            correct_key = z

print 'Here is the history output for U2 data at END POINT SET'

for m in range(len(trussDynamicOdb.steps['Loading Step'] \
                        .historyRegions[correct_key].historyOutputs['U2'].data)):
    time = trussDynamicOdb.steps['Loading Step'].historyRegions[correct_key] \
                                               .historyOutputs['U2'].data[m][0]
    Ydisp = trussDynamicOdb.steps['Loading Step'].historyRegions[correct_key] \
                                               .historyOutputs['U2'].data[m][1]
    print 'At time = %f Y-displacement = %f' % (time, Ydisp)
```

```
print 'There is/are %d history region(s) in dynamic explicit truss odb' \
       % len(trussDynamicOdb.steps['Loading Step'].historyRegions)
```

As was done for the static truss, this statement prints out the number of **HistoryRegion** objects, which is the same as the number of history output requests.

```
print 'In the odb their keys are %s and %s' \
      % (trussDynamicOdb.steps['Loading Step'].historyRegions.keys()[0],
         trussDynamicOdb.steps['Loading Step'].historyRegions.keys()[1])
```

Again, similar to the static truss, this statement prints out the names or repository keys of the history regions.

```
print 'The 2 sets in the model are %s and %s' \
      % (trussDynamicOdb.rootAssembly.nodeSets.keys()[0],
         trussDynamicOdb.rootAssembly.nodeSets.keys()[1])
```

We wish to display the history output at the end point (**END POINT SET**). In order to do that we need to identify the label of that node. The node sets in the model are stored in the repository **trussDynamicOdb.rootAssembly.nodeSets** and their keys are accessed with the **keys()** method. Since we already know there are two sets, we can access them with the indices [0] and [1]. If we were instead looking for element sets we would access them through **trussDynamicOdb.rootAssembly.nodeSets**

```
print 'The label of the node which makes up the set END POINT SET is %d' \
      % trussDynamicOdb.rootAssembly.nodeSets['END POINT SET'].nodes[0][0].label
```

We can use the key 'END POINT SET' to refer to the node in the **nodeSets** repository. It is an **OdbSet** object, and has a property called **nodes** which returns an array of **OdbMeshNodeArray** objects. This **OdbSet** object has only one element in our case – or more precisely one **OdbMeshNodeArray** object - which we refer to with the index [0]. This in turn has one **OdbMeshNode** object which we refer to with the second index [0]. This **OdbMeshNode** object has a member called **label** which returns the label of the node. In this manner we obtain the label of the node that is END POINT SET.

```
# Figure out which one has node 3
for z in trussDynamicOdb.steps['Loading Step'].historyRegions.keys():
    if z.find('Node') > -1:
        if z.split('.')[1] == '3':
            correct_key = z
```

Now that we know the label of the node for END POINT SET is 3, we search through the keys of the **HistoryRegion** objects looking for a 3. That is because these objects have strange names that are assigned by Abaqus such as 'Node TRUSS INSTANCE.2' and 'Node TRUSS INSTANCE.3'. In this case we already know this is what the key is, but in many situations you may not. Hence you need to do some sort of search within the String and find some clue as to whether this is the correct **HistoryRegion** object or not.

We use the **find()** method to do this. If the String 'Node' is present in the key (z) then the **find()** method will return its location in the String counting up from 0, and if is not found then it will return -1. So we first check to see if this **HistoryRegion** object is a node by looking for 'Node'. Once that is confirmed, we then split it where the dot occurs as the label is after the dot. We use the **split()** method for this, and pass '.' as the parameter. **split()** will return a list which will have 2 elements in this case, the String before the dot and the String after which is the node label. We use the index [1] to refer to this String. We compare it to '3' since we know that is the **nodelabel** because we obtained it from the previous statement. I have just gone ahead and hard coded it in here to make it more readable for learning purposes but in a real program you would instead write the statement as

```
if z.split('.')[1] == 'trussDynamicOdb.rootAssembly.nodeSets['END POINT SET']
                                                             .nodes[0][0].label':
```

If the key has the number 3 as the label after the dot, then this node is the END POINT SET and we can obtain our history data from it.

```
for m in range(len(trussDynamicOdb.steps['Loading Step'] \
                        .historyRegions[correct_key].historyOutputs['U2'].data)):
    time = trussDynamicOdb.steps['Loading Step'].historyRegions[correct_key] \
                                                .historyOutputs['U2'].data[m][0]
    Ydisp = trussDynamicOdb.steps['Loading Step'].historyRegions[correct_key] \
                                                .historyOutputs['U2'].data[m][1]
    print 'At time = %f Y-displacement = %f' % (time, Ydisp)
```

The **HistoryRegion** object has a member **historyOutputs** which contains the history output variables requested at that point. One of those is 'U2', the displacement in the Y direction. The displacement data for U2 can be accessed using its **data** member. It contains a number of tuples, each representing a point on the U2 displacement vs time curve. The points are accessed with the loop counter **m**, the time is the first coordinate which is accessed with index [0] and the Y displacement is the second coordinate which is accessed with the index [1].

This loop will print out all the U2 displacement versus time values available.

16.5.6 Extract material and section definitions

The following statements extract material and section information from the output database

```
# -----------------------------------------------------------------------
# Extract part, material and section informatio from the odb
# -----------------------------------------------------------------------

from odbMaterial import *
from odbSection import *

print '\n\n'
print '=================================================='

print 'The first (and only) part instance in the model is %s ' \
       % trussOdb.rootAssembly.instances.keys()[0]

print 'The first (and only) material in the model is %s ' \
       % trussOdb.materials.keys()[0]

steel_material = trussOdb.materials.values()[0]

if hasattr(steel_material, 'elastic'):
    print 'The material is elastic'
else:
    print 'The material is not elastic'

print 'Its density is %f' % steel_material.density.table[0][0]
print 'Its Young\'s modulus is %.2E and its Poisson\'s ratio is %f ' \
       % (steel_material.elastic.table[0][0], steel_material.elastic.table[0][1])

print 'The first (and only) section in the model is %s ' \
       % trussOdb.sections.keys()[0]

truss_section = trussOdb.sections.values()[0]
if type(truss_section) == HomogeneousSolidSectionType:
    print 'The section is of type HomogeneousSolidSection'
else:
    print 'The section is not of type HomegeneousSolidSection'
```

```
from odbMaterial import *
from odbSection import *
```

These **import** statements are required to access material and section information from the output database.

```
print 'The first (and only) part instance in the model is %s ' \
       % trussOdb.rootAssembly.instances.keys()[0]
```

trussOdb.rootAssembly.instances refers to a repository of **OdbInstance** objects, or part instances in plain English. Here we print out the key of the first part instance.

```
print 'The first (and only) material in the model is %s ' \
      % trussOdb.materials.keys()[0]
```

trussOdb.materials refers to the repository of **Material** objects in the model. Here we print out the key of the first **Material** object.

```
steel_material = trussOdb.materials.values()[0]
```

Here we assign the first **Material** lobject to the variable **steel_material**. The statement could also have been written as

```
steel_material = trussOdb.materials[trussOdb.materials.keys()[0]]
```

Next we use **if** else statements to test if the material has elastic properties defined

```
if hasattr(steel_material, 'elastic'):
    print 'The material is elastic'
else:
    print 'The material is not elastic'
```

hasattr() is a built in Python function. It accepts an object and a String as arguments. If the String is the name of one of the objects attributes it returns True, and if not it returns False.

He we test to see if the material has elasticity defined. If the **Material** object has elasticity – Young's modulus and Poisson's ratio were defined when creating the model, it will have a member called **elastic**.

```
print 'Its density is %f' % steel_material.density.table[0][0]
```

The density is accessed using the **density** attribute of the **Material** object. This in turn has an attribute called **table** which contains a tuple of density versus temperature values. We access the first cell of this table.

```
print 'Its Young\'s modulus is %.2E and its Poisson\'s ratio is %f ' \
      % (steel_material.elastic.table[0][0], steel_material.elastic.table[0][1])
```

The Young's modulus and Poisson's ratio of the material are found in the **elastic** attribute of the **Material** object, whose presence we tested in the **if** statement a moment ago.

elastic contains a **table** member with one row and two columns that store the Young's modulus and Poisson's ratio.

```
print 'The first (and only) section in the model is %s ' \
      % trussOdb.sections.keys()[0]
```

trussOdb.sections refers to the **Section** object repository. We get the name or repository key of the first section using **.keys()[0]**.

```
truss_section = trussOdb.sections.values()[0]
```

Here we assign the first **Section** object to the variable **truss_section**. The statement could also have been written as

```
print trussOdb.sections[trussOdb.sections.keys()[0]]
```

Next we use **if-else** statements to test the type of the **Section** object

```
if type(truss_section) == HomogeneousSolidSectionType:
    print 'The section is of type HomogeneousSolidSection'
else:
    print 'The section is not of type HomegeneousSolidSection'
```

type() is a built in Python function. It accepts an object a parameter and returns a **type** object.

If you wish to test for a **HomogeneousSolidSection**, you compare it to **HomogeneousSolidSectionType**. If you were to test for a **HomegeneousShellSection**, you would compare it to **HomogeneousShellType**. For a **TrussSection** it would be **TrussSectionType**. And so on. You will know (or you can figure it out) the type of object that was created when you created the shell section and only need to add the word 'Type' at the end of it.

16.5.7 Extract material and section definitions

The following statements extract the entire material and section definitions from the output database and transfer them to a new model

```
# ------------------------------------------------------------------------
# Extract material and section definitions from the odb and place in a model
# ------------------------------------------------------------------------
```

```
mdb.Model(name='Model To Extract Odb Info', modelType=STANDARD_EXPLICIT)
mdb.models['Model To Extract Odb Info'].materialsFromOdb('TrussAnalysisJob.odb')
mdb.models['Model To Extract Odb Info'].sectionsFromOdb('TrussAnalysisJob.odb')
```

Abaqus provides a few methods to quickly extract information from an odb and place it in a model. This cannot be done from Abaqus/Viewer and must be done through a script. These methods are **beamProfilesFromOdb()**, **materialsFromOdb()**, **ModelFromOdbFile()**, **PartFromOdb()**, and **sectionsFromOdb()**. Here you see 2 of them being used.

materialsFromOdb() is a method of the **Material** object. It creates a **Material** object by reading an output database and places the new material in the materials repository for that model. It has one required argument **fileName** which is a String representing the name of the .odb file to read. The String may also be a full path. Note that if a material with this name exists in the model it will be overwritten.

sectionsFromOdb() is a method of the **Section** object. It creates a **Section** object by reading an output database and places the new section in the sections repository for that model. It too has the same required argument **fileName** which is a String representing the name of the .odb file to read. Again the String may also be a full path. Note that if a section with this name exists in the model it will be overwritten.

It is because these methods add materials and sections to the materials and sections repositories of a model that we needed to first create a new model for this purpose. We used the **Model()** method to create the model database.

In all the previous examples we have not created a new model, but rather changed the key of an existing model using the **mdb.models.changekey()**. The **Model()** method creates a **Model** object with the name being the String specified as its required argument. Other optional arguments such as **description, stefanBoltzmann** and **absoluteZero** are available as well - see the Abaqus Scripting Reference Manual for a full list.

16.6 Object Model Interrogation

At this point it should be evident that the output database object model runs very deep and you need to know exactly where the information you need is nested. For example, to find the strain energy of the model at time = 1 of the 'Loading Step' you used the statement:

```
trussOdb.steps['Loading Step'].historyRegions.values()[0].
                                               historyOutputs['ALLSE'].data[1][1]
```

You do not need to memorize this entire path structure. Most times it is best to use print statements either within the script, or in the Kernel Command Line Interface in the Abaqus/CAE window. And on other occasions you can reuse and modify code you have written previously.

The process of determining the output database model structure and how you can access the information you need using it is referred to as object model interrogation. There are two methods that are invaluable for doing this – **print** and **prettyPrint()**.

You've seen the print command already. **prettyPrint()** on the other hand prints out a formatted version of the object passed to it as an argument.

We can experiment with this in the Kernel Command Line Interface. Run the script created in this chapter and remove the **close()** commands that close the odbs at the end of the script, so that we can then experiment with the odbs.

Use the print statement to find out what information is stored in trussOdb.

```
>>> print trussOdb
({'analysisTitle': 'Analysis of truss under concentrated loads', 'closed': False,
'customData': python object wrapper, 'description': 'DDB object', 'diagnosticData':
'OdbDiagnosticData object', 'isReadOnly': True, 'jobData': 'JobData object',
'materials': 'Repository object', 'name': 'C:/AbaqusTemp/TrussAnalysisJob.odb',
'parts': 'Repository object', 'path': 'C:/AbaqusTemp/TrussAnalysisJob.odb',
'profiles': 'Repository object', 'readInternalSets': False, 'rootAssembly':
'OdbAssembly object', 'sectionCategories': 'Repository object', 'sections':
'Repository object', 'sectorDefinition': None, 'steps': 'Repository object',
'userData': 'UserData object'})
```

You notice that it prints everything together on the same line which makes it hard to read. Try **prettyPrint()** instead

```
>>> prettyPrint(trussOdb)
({'analysisTitle': 'Analysis of truss under concentrated loads',
  'closed': False,
  'customData': None,
  'description': 'DDB object',
  'diagnosticData': 'OdbDiagnosticData object',
  'isReadOnly': True,
  'jobData': 'JobData object',
  'materials': 'Repository object',
  'name': 'C:/AbaqusTemp/TrussAnalysisJob.odb',
  'parts': 'Repository object',
  'path': 'C:/AbaqusTemp/TrussAnalysisJob.odb',
  'profiles': 'Repository object',
  'readInternalSets': False,
  'rootAssembly': 'OdbAssembly object',
  'sectionCategories': 'Repository object',
  'sections': 'Repository object',
  'sectorDefinition': None,
  'steps': 'Repository object',
  'userData': 'UserData object'})
```

prettyPrint() displays the contents of **trussOdb** in a much more readable fashion. You can now see what members and methods are contained in it. One of these is **steps** which is a **Repository** object. Let's explore steps.

```
>>> prettyPrint(trussOdb.steps)
{'Loading Step': 'OdbStep object'}
```

The steps repository appears to contain one name:value pair. The step is named 'Loading Step' which is its repository key. Let's dig deeper

```
>>> prettyPrint(trussOdb.steps['Loading Step'])
({'acousticMass': -1.0,
  'acousticMassCenter': 'tuple object',
  'description': 'Loads are applied to the truss in this step',
  'domain': TIME,
  'eliminatedNodalDofs': 'NodalDofsArray object',
  'frames': 'OdbFrameArray object',
  'historyRegions': 'Repository object',
  'inertiaAboutCenter': 'tuple object',
  'inertiaAboutOrigin': 'tuple object',
  'loadCases': 'Repository object',
  'mass': -1.0,
  'massCenter': 'tuple object',
  'name': 'Loading Step',
  'nlgeom': False,
  'number': 1,
  'previousStepName': 'Initial',
  'procedure': '*STATIC',
  'retainedEigenModes': 'tuple object',
  'retainedNodalDofs': 'NodalDofsArray object',
  'timePeriod': 1.0,
  'totalTime': 0.0})
```

Let's explore the **historyRegions** repository object.

```
>>> prettyPrint(trussOdb.steps['Loading Step'].historyRegions)
{'Assembly ASSEMBLY': 'HistoryRegion object'}
```

We see that it has one **HistoryRegion** object. To look inside it:

```
>>> prettyPrint(trussOdb.steps['Loading Step'].historyRegions['Assembly ASSEMBLY'])
({'description': 'Output at assembly ASSEMBLY',
  'historyOutputs': 'Repository object',
  'loadCase': None,
  'name': 'Assembly ASSEMBLY',
  'point': 'HistoryPoint object',
  'position': WHOLE_MODEL})
```

Alternatively we could have typed

```
>>> prettyPrint(trussOdb.steps['LoadingStep'].historyRegions.values()[0])
({'description': 'Output at assembly ASSEMBLY',
  'historyOutputs': 'Repository object',
  'loadCase': None,
  'name': 'Assembly ASSEMBLY',
  'point': 'HistoryPoint object',
  'position': WHOLE_MODEL})
```

To see which history outputs are available we check the **historyOutputs** repository

```
>>> prettyPrint(trussOdb.steps['Loading Step'].historyRegions['Assembly
ASSEMBLY'].historyOutputs)
{'ALLAE': 'HistoryOutput object',
 'ALLCD': 'HistoryOutput object',
 'ALLDMD': 'HistoryOutput object',
 'ALLEE': 'HistoryOutput object',
 'ALLFD': 'HistoryOutput object',
 'ALLIE': 'HistoryOutput object',
 'ALLJD': 'HistoryOutput object',
 'ALLKE': 'HistoryOutput object',
 'ALLKL': 'HistoryOutput object',
 'ALLPD': 'HistoryOutput object',
 'ALLQB': 'HistoryOutput object',
 'ALLSD': 'HistoryOutput object',
 'ALLSE': 'HistoryOutput object',
 'ALLVD': 'HistoryOutput object',
 'ALLWK': 'HistoryOutput object',
 'ETOTAL': 'HistoryOutput object'}
```

If we were to access one of these, such as 'ALLSE'

```
>>> prettyPrint(trussOdb.steps['Loading Step'].historyRegions['Assembly
ASSEMBLY'].historyOutputs['ALLSE'])
```

```
({'conjugateData': None,
  'data': 'tuple object',
  'description': 'Strain energy',
  'name': 'ALLSE',
  'type': SCALAR})
```

Let's examine at the **data** member

```
>>> prettyPrint(trussOdb.steps['Loading Step'].historyRegions['Assembly
ASSEMBLY'].historyOutputs['ALLSE'].data)
('tuple object', 'tuple object')
```

It has 2 tuples representing the ALLSE data. Let's find out what the second one contains.

```
>>> prettyPrint(trussOdb.steps['Loading Step'].historyRegions['Assembly
ASSEMBLY'].historyOutputs['ALLSE'].data[1])
(1.0, 65.4211)
```

It is giving us (time t2, ALLSE value v2). To access the ALLSE value at time = t2 we can write

```
>>> prettyPrint(trussOdb.steps['Loading Step'].historyRegions['Assembly
ASSEMBLY'].historyOutputs['ALLSE'].data[1][1])
65.4211
```

or

```
>>> print trussOdb.steps['Loading Step'].historyRegions['Assembly
ASSEMBLY'].historyOutputs['ALLSE'].data[1][1]
65.4210891724
```

I think you get the picture. By working our way through the object model using **prettyPrint**, we now know exactly where the ALLSE data is stored. This is how you would come up with the statement that we used in the script to find the ALLSE data at time = 1.

```
trussOdb.steps['Loading Step'].historyRegions.values()[0].
                                              historyOutputs['ALLSE'].data[1][1]
```

16.7 More object model interrogation techniques

You've seen how to use **prettyPrint()** in the previous section with the one required argument which is an Abaqus object. **prettyPrint()** also has a few optional arguments. One of them is **maxRecursionDepth**, which is an Int specifying the maximum depth to navigate and print. You can also set it to the SymbolicConstants **UNLIMITED** or **None**.

In the case of **UNLIMITED** you might get a lot of output. **None** on the other hand will resort to the default setting which is the current setting in the **extReprOptions** object.

You generally want to keep **maxRecursionDepth** set to a small number. Personally I never go beyond 2. This lets you see one level deeper without cluttering up the kernel command line interface.

To compare the difference **maxRecursionDepth** makes, here is **prettyPrint()** with **maxRecursionDepth = 1**

```
prettyPrint(trussOdb.steps['Loading Step'].historyRegions['Assembly ASSEMBLY'])
({'description': 'Output at assembly ASSEMBLY',
  'historyOutputs': 'Repository object',
  'loadCase': None,
  'name': 'Assembly ASSEMBLY',
  'point': 'HistoryPoint object',
  'position': WHOLE_MODEL})
```

And here it is with **maxRecursionDepth = 2**

```
>>> prettyPrint(trussOdb.steps['Loading Step'].historyRegions['Assembly ASSEMBLY'],
2)
({'description': 'Output at assembly ASSEMBLY',
  'historyOutputs': {'ALLAE': 'HistoryOutput object',
                     'ALLCD': 'HistoryOutput object',
                     'ALLDMD': 'HistoryOutput object',
                     'ALLEE': 'HistoryOutput object',
                     'ALLFD': 'HistoryOutput object',
                     'ALLIE': 'HistoryOutput object',
                     'ALLJD': 'HistoryOutput object',
                     'ALLKE': 'HistoryOutput object',
                     'ALLKL': 'HistoryOutput object',
                     'ALLPD': 'HistoryOutput object',
                     'ALLQB': 'HistoryOutput object',
                     'ALLSD': 'HistoryOutput object',
                     'ALLSE': 'HistoryOutput object',
                     'ALLVD': 'HistoryOutput object',
                     'ALLWK': 'HistoryOutput object',
                     'ETOTAL': 'HistoryOutput object'},
  'loadCase': None,
  'name': 'Assembly ASSEMBLY',
  'point': ({'assembly': 'OdbAssembly object',
             'element': None,
             'face': None,
             'instance': None,
             'ipNumber': None,
             'node': None,
             'position': WHOLE_MODEL,
             'region': None,
             'sectionPoint': None}),
  'position': WHOLE_MODEL})
```

Another way to interrogate object attributes is the **printPaths()** method. This returns information similar to print, except in the form of full paths of the attributes rather than just the attribute names. Its format is *printPaths(object)*. You can also change the depth as you can with **prettyPrint()** eg. **printPaths(object, 2)**.

```
>>> printPaths(trussOdb.steps['Loading Step'].historyRegions['Assembly ASSEMBLY'])
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].description
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].historyOutputs
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].loadCase
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].name
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].point
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].position
```

If you wish to store the paths as Strings in a variable for further processing, you can instead use **getPaths()**.

```
paths_String = getPaths(trussOdb.steps['Loading Step'].historyRegions['Assembly
ASSEMBLY'])

>>> prettyPrint(paths_String)
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].description
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].historyOutputs
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].loadCase
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].name
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].point
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].position
```

If you are more interested in the types of data, but also want to know the full path, you could use **printTypes()**

```
>>> printTypes(trussOdb.steps['Loading Step'].historyRegions['Assembly ASSEMBLY'])
```

```
str
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].description
Repository
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].historyOutputs
NoneType
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].loadCase
str
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].name
HistoryPoint
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].point
SymbolicConstant
session.openOdb(r'C:/AbaqusTemp/TrussAnalysisJob.odb').steps['Loading
Step'].historyRegions['Assembly ASSEMBLY'].position
```

One final way to interrogate object members and methods is to use **__members__** and **__methods__**. You type these in as part of the statement at the end, and Abaqus tells you what your options are

```
>>> trussOdb.steps['Loading Step'].historyRegions['Assembly ASSEMBLY'].__members__
['description', 'historyOutputs', 'loadCase', 'name', 'point', 'position']
```

__members__ returns the same member attributes as **prettyPrint()**. However it does not tell you the type of object each of the members is.

```
>>> trussOdb.steps['Loading Step'].historyRegions['Assembly ASSEMBLY'].__methods__
['HistoryOutput', 'getSubset']
```

__methods__ tells you what methods can be used at this point. This is information you do not get from **prettyPrint()**.

With so many interrogation options, all of which return similar data, which one you use is sometimes a matter of personal preference. This author has a tendency to use **print**, **prettyPrint()** and **__methods__** almost all of the time but you might prefer **printPaths()** and **printTypes()**.

16.8 Summary

You now have a good understanding of how you can access information stored in an output database using a Python script. There is a wealth of information available in an odb, and all you need to access it is a basic understanding of the output database object model. There is no sense in memorizing the entire tree structure which has hundreds of nested repositories, attributes and methods; you should instead use object model interrogation with **print** and **prettyPrint()** statements to determine how to access the information you need.

17

Combine Frames of two Output Databases and Create an Animation

17.1 Introduction

In the previous chapter we explored two output databases to understand the output database object model and learn how to obtain useful information from an .odb file. In this chapter we will demonstrate how to create a new output database file from scratch. To make things interesting we will open two other output databases, extract the required information from them, and combine this information from both of them into a new output database.

We will modify the plate bending example from Chapter 10 in order to include the effect of plasticity, and increase the loading on it to force it into plastic deformation. We shall request Abaqus to write restart information to the .res file during this analysis. We will then continue the analysis using the restart file and remove the load from the plate allowing it to spring back and recover its elastic deformation (the plastic deformation will not be recovered). The two analyses will generate two output databases. However these do not overlap, and the first frame of the restart analysis will correspond to the last frame of the original analysis. In order to view the results of the original analysis in Abaqus/Viewer, the first .odb needs to be opened, and for the second analysis (springback) the second .odb will need to be opened.

Our goal is to use a Python script to read both the output databases, extract the nodal displacement information, and create a new output database which combines the information of both analyses. This allows the analyst to view the entire set of results (that you choose to include in the combined odb) in Abaqus/Viewer since the frames of both analyses are joined together. We will then create an animation which includes both the bending and the springback.

17.2 Methodology

We will need to create 3 Python scripts for this example.

The first script will be a modification of the plate bending script from Chapter 10. We will update it to include plastic material properties, and increase the load to cause bending stresses that exceed the elastic limit. We will also need to request Abaqus to write restart information to the .res file. On running the simulation an output database file will be produced.

The second script will replicate the original model, and add a new step to it where the load is removed. It will then continue the analysis using this new model. On running this simulation a second output database file will be produced.

The third script will open and read the output databases created by the two analyses, and extract the nodal displacement information. It will then create a new output database, and in it create the part, instance it, create two steps, and add the displacement field output data to these steps from each of the .odb files. It will then open this .odb in Abaqus/Viewer, animate the time history and save the animation, which will include both the bending and the springback.

17.3 Procedure in GUI

You can perform the simulation in Abaqus/CAE by following the steps listed below. You can either read through these, or watch the video demonstrating the process on the book website.

1. Open the model created for the elastic plate bending example of Chapter 10.
2. Rename **Plate Bending Model** as **Plastic Plate Bending Model**
    a. Right-click on **Plate Bending Model** in the Model Database
    b. Choose **Rename..**
    c. Change name to **Plastic Plate Bending Model**
3. Modify the material
    a. Expand the **Materials** container in the Model Tree
    b. Right-click on **AISI 1005 Steel**
    c. Choose **Rename..**

   d. Change name to **Steel**
   e. Right-click on **AISI 1005 Steel**
   f. Choose **Edit.** The **Edit Material** window is displayed
   g. Select **Mechanical > Elasticity > Elastic**. Change **Poisson's Ratio** to **0.23**
   h. Select **Mechanical >Plasticity>Plastic**. Chang
   i. Right-click in the Data table and choose **Read from File.** The **Read Data from ASCII File** window is displayed.
   j. Set **File** to **plate_bending_steel_plasticity_data** by clicking the **Select** button.
   k. Set **Start reading values into table row** to **1**
   l. Set **Start reading values into table column** to **1**
   m. Set **Base Feature Shape** to **Shell**
   n. Set **Base Feature Type** to **Planar**
   o. Right-click in the table and choose **Create XY Data....** The **Create XY Data** window is displayed.
   p. Set **Name** to **steel stress vs. plastic strain**
   q. Set **Read X values from column** to **2**
   r. Set **Read Y values from column** to **1**
   s. Set **Approximate Size** to **20**
   t. Click **OK**. You will enter Sketcher mode.
   u. Switch to the **Visualization** module using the **Module** dropdown menu
   v. Expand the **XY Data** container in the Results Tree
   w. Double-click on **steel stress vs. plastic strain** in the Results tree. Aplot of the yield stress vs. plastic strain data is displayed
   x. Click the **XY Curve Options** tool. The **Curve Options** window is displayed.
   y. Set the **Style** to **dashed** using the dropdown menu.
   z. Check **Show symbol**.
   aa. Change **Symbol** to another shape using the dropdown menu
   bb. Change **Size** to **Large** using the dropdown menu
   cc. Click **Dismiss** to close the window
4. Edit the Step
   a. Expand the **Steps** container in the Model Tree.
   b. Double-click **Load Step**. The **EditStep** window is displayed
   c. Switch to the **Incrementation** tab
   d. Set **Increment size** to **0.1**
5. Edit Loads

   a. Expand the **Loads** container in the Model Tree.
   b. Double-click Concentrated Forces. The **Edit Load** window is displayed
   c. Change **CF3** to **-270000.0** to apply a 270000.0 N force in downward (negative Y) direction
   d. Click **OK**
6. Create and submit the job
   a. Double-click on **Jobs** in the Model Database. The **Create Job** window is displayed
   b. Set **Name** to **PlateJobPlastic**
   c. Set **Source** to **Plastic Plate Bending Model**
   d. Select **Plastic Plate Bending Model** (it is the only option displayed)
   e. Click **Continue..** The **Edit Job** window is displayed
   f. Set **Description** to **Job simulates the plastic bending of a plate**
   g. Set **Job Type** to **Full Analysis.**
   h. Leave all other options at defaults
   i. Click **OK**
   j. Expand the **Jobs** container in the Model Database
   k. Right-click on **PlateJobPlastic** and choose **Submit.**
   l. You will see a popup saying **History output is not requested in the following steps: Load Step. OK to continue with job submission?** Click **Yes.**
   m. This will run the simulation. You will see the following messages in the message window:

      Error in job PlateJobPlastic: THERE IS NO MATERIAL BY THE NAME AISI 1005 STEEL

      Error in job PlateJobPlastic: 90 elements have missing property definitions. The elements have been identified in error set ErrElemMissingSection

      Job PlateJobPlastic: Analysis Input File Processor aborted due to errors.

      Error in job PlateJobPlastic: Analysis Input File Processor exited with an error

      You will also see the word Aborted next to PlateJobPlastic in the Model Tree
7. Edit Sections
   a. Expand the **Sections** container in the Model tree

   b. Double-click **Concentrated Forces**. The **Edit Load** window is displayed. You also see a message **Section 'Plate Section' contains a reference to material 'AISI 1005 Steel', but that material no longer exists**
   c. Click **Dismiss**
   d. Set **Material** to **Steel** using the drop down menu
   e. Click **OK**to close the **Edit Section** window
8. Resubmit the job
   a. Right-click on **PlateJobPlastic** in the **Jobs** container of the Model tree and choose **Submit**.
   b. You will see a popup saying **Job files already exist for PlateJobPlastic. OK to overwrite?** Click **OK**.
   c. You will see a popup saying **History output is not requested in the following steps: Load Step. OK to continue with job submission?** Click **Yes**.
   d. This will run the simulation. You will see the following messages in the message window:
      The job input file "PlateJobPlastic.inp" has been submitted for analysis.
      Job PlateJobPlastic: Analysis Input File Processor completed successfully
      Job PlateJobPlastic: Abaqus/Standard completed successfully
      Job PlateJobPlastic completed successfully
9. Plot contour and change font of legend, title block and state block
   a. Right-click on **PlateJobPlastic (Completed)** in the Model Database. Choose **Results**. The viewport changes to the **Visualization** module.
   b. In the toolbar choose **Plot Contours on Deformed Shape** tool to plot the Mises stress contours on the plate
   c. In the menu bar click on **Viewport > Viewport Annotation Options**
   d. Switch to the **Legend** tab
   e. Click **Set Font**. The Select Font window is displayed.
   f. Set **Size** to **14** using the dropdown menu
   g. For **Apply To** check **Legend**
   h. Click Ok. The font size of the legend is now 14.
   i. Switch to the Title Block tab
   j. Click **Set Font**. The Select Font window is displayed.
   k. Set **Font** to **Times New Roman** using the dropdown menu
   l. Set **Size** to **14** using the dropdown menu

   m. For **Style** check **Italic**
   n. For **Apply To** check **Title block** and **State block**
   o. Click **OK**

10. Request Field Outputs
    a. Switch to the **Step** module using the **Module** dropdown menu
    b. Using the menu bar click on **Output > Restart Requests...** The **Edit Restart Requests** window is displayed.
    c. In the **Frequency** column, set the frequency to **1** for Load Step
    d. Check **Overlay**
    e. Click **OK**
11. Resubmit the job
    a. Right-click on **PlateJobPlastic** in the **Jobs** container of the Model tree and choose **Submit**.
    b. You will see a popup saying **Job files already exist for PlateJobPlastic. OK to overwrite?** Click **OK**.
    c. You will see a popup saying **History output is not requested in the following steps: Load Step. OK to continue with job submission?** Click **Yes**.
    d. This will run the simulation. You will see the following messages in the message window:
       The job input file "PlateJobPlastic.inp" has been submitted for analysis.
       Job PlateJobPlastic: Analysis Input File Processor completed successfully
       Job PlateJobPlastic: Abaqus/Standard completed successfully
       Job PlateJobPlastic completed successfully
12. Check the Abaqus work directory – it is C:\Temp by default – for the presence of a restart file PlateJobPlastic.res
13. Copy the model to create a restart model
    a. Right click on **Plastic Plate Bending Model** in the Model tree.
    b. Choose **Copy Model..** The **Copy Model** dialog box is displayed
    c. Set **Copy Plastic Plate Bending Model to:** to **Plate Springback Model**
    d. Click **OK**. A new model **Plate Springback Model** is displayed in the Model tree
    e. Right click on **Plate Springback Model**.
    f. Choose **Edit Attributes..** The **Edit Model Attributes** window is displayed
    g. In the **Restart** tab check **Read data from job**and type in **PlateJobPlastic**

   h. Set **Step name** to **Load Step**
   i. Click Ok..

14. Add a new step
    a. Double-click on **Steps** container in the Model Tree. The Create Step window is displayed
    b. Set **Name** to **Springback**
    c. Set **Insert New Step After** to **Load Step**
    d. Set **Procedure Type** to **General > Static, General**
    e. Click **Continue..** The **Edit Step** window is displayed
    f. In the **Basic** tab, set **Description** to **Remove load and allow elastic springback.**
    g. Set **Time period** to **1**
    h. Switch to the **Incrementation** tab
    i. Set **Initial Increment size** to **0.1**
    j. Click **OK.**

15. Deactivate the load in the Springback step
    a. Right click on **Loads** in the Model tree.
    b. Choose **Manager..** The **Load Manager** dialog box is displayed
    c. Click **Propagated** in the **Springback** step. The word **Propagated** changes to **Inactive**
    d. Click **Deactivate**
    e. Click **Dismiss.**
16. Create and submit the job
    a. Double-click on **Jobs** in the Model Database. The **Create Job** window is displayed
    b. Set **Name** to **PlateSpringbackJob**
    c. Set **Source** to **Plate Springback Model**
    d. Select **Plastic Plate Bending Model** (it is the only option displayed)
    e. Click **Continue..** The **Edit Job** window is displayed
    f. Set **Description** to **Job allows elastic springback**
    g. Set **Job Type** to **Restart.**
    h. Leave all other options at defaults
    i. Click **OK**

j. Expand the **Jobs** container in the Model Database
k. Right-click on **PlateSpringbackJob** and choose **Submit.**
l. You will see a popup saying **History output is not requested in the following steps: Load Step. OK to continue with job submission?** Click **Yes.**
m. This will run the simulation. You will see the following messages in the message window:
The job input file "PlateJobPlastic.inp" has been submitted for analysis.
Job PlateSpringbackJob: Analysis Input File Processor completed successfully
Job PlateSpringbackJob: Abaqus/Standard completed successfully
Job PlateSpringbackJob completed successfully

16. Plot contour
a. Right-click on **PlateSpringbackJob (Completed)** in the Model Tree. Choose **Results**. The viewport changes to the **Visualization** module.
b. In the toolbar choose **Plot Contours on Deformed Shape** tool to plot the Mises stress contours on the plate
c. In the menu bar click on **Viewport > Create.** A new viewport is created. It may be hidden behind the current viewport if you cannot seen
d. In the menu bar click on **Viewport > Tile Vertically.** The two viewports are placed side by side. The new viewport has the same contents as the old one.
e. Click the titlebar of Viewport 1 to make it the active viewport.
f. Right click on **Output Databases** in the Results tree and choose **Open**
g. Choose PlateJobPlastic.odb from the Open Database browse window. The results of the original analysis are displayed in the viewport.
h. In the toolbar choose **Plot Contours on Deformed Shape** tool to plot the Mises stress contours on the plate
i. Click the titlebar of Viewport 2 to make it the active viewport.
j. Use the **ODB** dropdown menu (above the viewport) to ensure that the ODB is set to PlateSpringbackJob.odb
k. Again click the titlebar of Viewport 1 to make it the active viewport.
l. Click the **Frame Selector** tool. You see the **Frame Selector** dialog box
m. Set the frame to the last frame (frame 14) of the Load step
n. Close the **Frame Selector** tool by clicking the red x at the top right corner of the dialog box
o. Again click the titlebar of Viewport 2 to make it the active viewport.

p. Click the **Frame Selector** tool. You see the **Frame Selector** dialog box
q. Set the frame to the first frame (frame 0) of the Springback step
r. Compare the contour plots and the legends. They should be identical.
s. Click the titlebar of Viewport 1 to make it the active viewport
t. Change the primary field variable to **UT** (translations and rotations) **U3** using the field output toolbar. The displacement contour is displayed on the plate
u. Click the titlebar of Viewport 2 to make it the active viewport
v. Change the primary field variable to **UT** (translations and rotations) **U3** using the field output toolbar. The displacement contour is displayed on the plate
w. Compare the contour plots and the legends. They should be identical

17.4 How to run the script

The user should open up a new model in Abaqus using **File > New Model Database > With Standard/Explicit Model** and run the first script to simulate plastic bending of the plate using using **File > Run Script...** The analysis will create an output database file 'PlateJobPlastic.odb'.

The user must then run the second script using **File > Run Script...** The analysis will create an output database file 'PlateSpringbackJob.odb'.

Finally the user should run the third script (again using **File > Run Script...**) which will read from both thee output databases and create a new one called Plate_plastic_bending_and_springback.odb.

17.5 Python Script to simulate plastic plate bending

The following Python script replicates the above procedure for setting up and analyzing plastic bending of the plate and requesting restart data. It is a modified version of the script from Chapter 10. You can find it in the source code accompanying the book in **plate_bending_plastic.py**. You can run it by opening a new document in Abaqus/CAE (**File > New Model database > With Standard/Explicit Model**) and running it with **File > Run Script**

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# -----------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Plastic Plate Bending Model')
plateModel = mdb.models['Plastic Plate Bending Model']

# -----------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the plate using the rectangle tool
plateProfileSketch = plateModel.ConstrainedSketch(name='Plate Sketch', sheetSize=20)
plateProfileSketch.rectangle(point1=(0.0,0.0), point2=(5.0,3.0))

# b) Create a shell named "Plate" using the sketch
platePart=plateModel.Part(name='Plate', dimensionality=THREE_D,
type=DEFORMABLE_BODY)
platePart.BaseShell(sketch=plateProfileSketch)

# -----------------------------------------------------
# Create material

import material

# Create material Steel by assigning mass density, youngs modulus and poissons ratio
plateMaterial = plateModel.Material(name='Steel')
plateMaterial.Density(table=((7872, ),       ))
plateMaterial.Elastic(table=((200E9, 0.29), ))

# Read plastic properties from an external file
material_file_name = 'plate_bending_steel_plasticity_data.txt'
stress_data_list = []
strain_data_list = []

f = open(material_file_name)
for line in f:
        stress_strain_line = line
        stress_strain_list = stress_strain_line.split()
        stress_data_list.append(stress_strain_list[0])
        strain_data_list.append(stress_strain_list[1])
f.close()

plasticity_curve_list = []
for i in range(len(stress_data_list)):
        plasticity_curve_list.append((float(stress_data_list[i]),
float(strain_data_list[i])))

plasticity_curve_tuple = tuple(plasticity_curve_list)

plateMaterial.Plastic(table=plasticity_curve_tuple)
```

```
# --------------------------------------------------------
# Create homogeneous shell section of thickness 0.1m and assign the plate to it

import section

# Create a section to assign to the plate
plateSection = plateModel.HomogeneousShellSection(name='Plate Section',
material='Steel', thicknessType=UNIFORM, thickness=0.1)

#assign the plate to this section
plate_face_point = (2.5, 1.5, 0.0)
plate_face = platePart.faces.findAt((plate_face_point,))
plate_region = (plate_face,)

platePart.SectionAssignment(region=plate_region, sectionName='Plate Section',
offset=0.0, offsetType=MIDDLE_SURFACE, offsetField='')

# --------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
plateAssembly = plateModel.rootAssembly
plateInstance = plateAssembly.Instance(name='Plate Instance', part=platePart,
dependent=ON)

# --------------------------------------------------------
# Create the step

import step

# Create a static general step
plateModel.StaticStep(name='Load Step', previous='Initial', description='Apply
concentrated forces in this step', nlgeom=ON, initialInc=0.1)

plateModel.steps['Load Step'].Restart(frequency=1, numberIntervals=0, overlay=ON,
timeMarks=OFF)

# --------------------------------------------------------
# Create the field output request

# change the name of field output request 'F-Output-1' to 'Output Stresses and
Displacements'
plateModel.fieldOutputRequests.changeKey(fromName='F-Output-1', toName='Output
Stresses and Displacements')

# since F-Output-1 is applied at the 'Apply Load' step by default, 'Selected Field
Outputs' will be too
# we only need to set the required variables
plateModel.fieldOutputRequests['Output Stresses and
Displacements'].setValues(variables=('S','UT'))
```

```
# -------------------------------------------------------
# Create the history output request

# We don't want any history outputs so lets delete the existing one 'H-Output-1'
del plateModel.historyOutputRequests['H-Output-1']

# -------------------------------------------------------
# Apply boundary conditions - fix one edge

fixed_edge = plateInstance.edges.findAt(((0.0, 1.5, 0.0), ))
fixed_edge_region=regionToolset.Region(edges=fixed_edge)

plateModel.DisplacementBC(name='FixEdge', createStepName='Initial',
region=fixed_edge_region, u1=SET, u2=SET, u3=SET, ur1=SET, ur2=SET, ur3=SET,
amplitude=UNSET, distributionType=UNIFORM, fieldName='', localCsys=None)

# Instead of using the displacements/rotations boundary condition and setting all
six DOF to zero
# We could have just used the Encastre condition with the following statement
# plateModel.EncastreBC(name='Encaster edge', createStepName='Initial',
region=fixed_edge_region)

# -------------------------------------------------------
# Create vertices on which to apply concentrated forces by partitioning part

# Create the datum points
platePart.DatumPointByCoordinate(coords=(0.0, 1.0, 0.0))
platePart.DatumPointByCoordinate(coords=(0.0, 2.0, 0.0))
platePart.DatumPointByCoordinate(coords=(5.0, 1.0, 0.0))
platePart.DatumPointByCoordinate(coords=(5.0, 2.0, 0.0))

# Assign the datum points to variables
# Abaqus stores the 4 datum points in platePart.datums
# Since their keys may or may not start at zero, put the keys in an array sorted in
ascending order
platePart_datums_keys = platePart.datums.keys()
platePart_datums_keys.sort()
plate_datum_point_1 = platePart.datums[platePart_datums_keys[0]]
plate_datum_point_2 = platePart.datums[platePart_datums_keys[1]]
plate_datum_point_3 = platePart.datums[platePart_datums_keys[2]]
plate_datum_point_4 = platePart.datums[platePart_datums_keys[3]]

# Select the entire face and partition it using two points
partition_face_pt = (2.5, 1.5, 0.0)
partition_face = platePart.faces.findAt((partition_face_pt,))
platePart.PartitionFaceByShortestPath(point1=plate_datum_point_1,
point2=plate_datum_point_3, faces=partition_face)

# Now two faces exist, select the one that needs to be partitioned
partition_face_pt = (2.5, 2.0, 0.0)
partition_face = platePart.faces.findAt((partition_face_pt,))
```

```
platePart.PartitionFaceByShortestPath(point1=plate_datum_point_2,
point2=plate_datum_point_4, faces=partition_face)

# Since the partitions have been created, vertices can be extracted
vertices_for_concentrated_force = plateInstance.vertices.findAt(((5.0, 1.0, 0.0),),
                                                               ((5.0, 2.0, 0.0),),)

# -----------------------------------------------------------
# Apply concentrated forces

plateModel.ConcentratedForce(name='Concentrated Forces', createStepName='Load Step',
region=(vertices_for_concentrated_force,), cf3=-270000.0, distributionType=UNIFORM)

# -----------------------------------------------------------
# Create the mesh

import mesh

# set element type
plate_mesh_region = plate_region

elemType1 = mesh.ElemType(elemCode=S8R, elemLibrary=STANDARD)

platePart.setElementType(regions=plate_mesh_region, elemTypes=(elemType1,))

# seed edges by number
mesh_edges_vertical = platePart.edges.findAt(((0.0, 0.5, 0.0), ),
                                             ((0.0, 1.5, 0.0), ),
                                             ((0.0, 2.5, 0.0), ),
                                             ((5.0, 0.5, 0.0), ),
                                             ((5.0, 1.5, 0.0), ),
                                             ((5.0, 2.5, 0.0), ))

mesh_edges_horizontal = platePart.edges.findAt(((2.5, 0.0, 0.0), ),
                                               ((2.5, 1.0, 0.0), ),
                                               ((2.5, 2.0, 0.0), ),
                                               ((2.5, 3.0, 0.0), ))

platePart.seedEdgeByNumber(edges=mesh_edges_vertical, number = 3)
platePart.seedEdgeByNumber(edges=mesh_edges_horizontal, number=10)

platePart.generateMesh()

# ------------------------------------------------------------
# Create and run the job

import job

# create the job

mdb.Job(name='PlateJobPlastic', model='Plastic Plate Bending Model', type=ANALYSIS,
description='Job simulates the plastic bending of a plate')
```

```
# run the job
mdb.jobs['PlateJobPlastic'].submit(consistencyChecking=OFF)

# do not return control till job is finished running
mdb.jobs['PlateJobPlastic'].waitForCompletion()


################################################
#-----------------------------------------------
#Post Processing
#-----------------------------------------------
################################################

# -------------------------------------------------------
# Display deformed state with contours

import visualization

plate_viewport = session.Viewport(name='Plate Plastic Bending Results Viewport')
plate_Odb_Path = 'PlateJobPlastic.odb'
an_odb_object = session.openOdb(name=plate_Odb_Path)
plate_viewport.setValues(displayedObject=an_odb_object)
plate_viewport.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))

# Displacement U3, the 3-component of displacement
plate_viewport.odbDisplay.setPrimaryVariable(variableLabel='UT',
outputPosition=NODAL, refinement=(COMPONENT, 'U3'),)


# -------------------------------------------------------
# Report stresses in descending order

import odbAccess

# The main session viewport must be set to the odb object using the following line.
If not you might receive an error message that states
# "There are no active entities. No report has been generated."
session.viewports['Viewport: 1'].setValues(displayedObject=an_odb_object)

# Set the option to display the reported quantity (in our case the stresses) in
descending order
session.fieldReportOptions.setValues(sort=DESCENDING)

# Name the report and give it a path. If you do not assign a path (as is done here)
it will be stored in the default abaqus temporary directory
report_name_and_path='PlateStresses.rpt'

# Write the field report outputting the Mises stresses
session.writeFieldReport(fileName=report_name_and_path, append=OFF,
sortItem='S.Mises', odb=an_odb_object, step=0, frame=1,
    outputPosition=INTEGRATION_POINT, variable=(('S', INTEGRATION_POINT,
((INVARIANT, 'Mises'), )), ))
```

Let's dissect the script and understand how it works. Since most of the script is almost identical to the one used in the elastic plate bending example, only the differences will be highlighted.

```
mdb.models.changeKey(fromName='Model-1', toName='Plastic Plate Bending Model')
plateModel = mdb.models['Plastic Plate Bending Model']
```

The name of the model has been set to 'Plastic Plate Bending Model' and the subsequent statement has been modified accordingly.

```
plateMaterial = plateModel.Material(name='Steel')
```

The material created in this script is given generic steel properties hence it has been named 'Steel'.

```
# Read plastic properties from an external file
material_file_name = 'plate_bending_steel_plasticity_data.txt'
stress_data_list = []
strain_data_list = []

f = open(material_file_name)
for line in f:
        stress_strain_line = line
        stress_strain_list = stress_strain_line.split()
        stress_data_list.append(stress_strain_list[0])
        strain_data_list.append(stress_strain_list[1])
f.close()

plasticity_curve_list = []
for i in range(len(stress_data_list)):
        plasticity_curve_list.append((float(stress_data_list[i]),
float(strain_data_list[i])))

plasticity_curve_tuple = tuple(plasticity_curve_list)

plateMaterial.Plastic(table=plasticity_curve_tuple)
```

If you were to hardcode the plasticity properties into the program, you would use the statement

```
plateMaterial.Plastic(table=((2.8E8, 0.0), (3.25E8, 0.025), (3.45E8, 0.06), (3.6E8,
0.1), (3.8E8, 0.2)))
```

What we are instead doing here is reading them in from an external text file 'plate_bending_steel_plasticity_data.txt'. We need to then put these properties in the format of the above statement.

You've encountered the **open()** command before. The **for** loop reads in each line of data one by one. These are pairs of stress and strain, and Python's **split()** command splits them at the whitespace between them creating a list with 2 members (**stress_strain_list**). The first member of this list is added to **stress_data_list** using the **append()** method and the second member is added to **strain_data_list**. In a second **for** loop, one stress and one strain from each of these lists is appended to a variable called **plasticity_curve_list** giving us a list of tuples. The **tuple()** method is then used to convert this to a tuple of tuples. Finally the **Plastic()** method is used to create the plastic material by passing the tuple of tuples to it as **table**.

```
plateSection = plateModel.HomogeneousShellSection(name='Plate Section',
material='Steel', thicknessType=UNIFORM, thickness=0.1)
```

This statement has been updated to use the new material 'Steel'.

```
plateModel.StaticStep(name='Load Step', previous='Initial', description='Apply
concentrated forces in this step', nlgeom=ON, initialInc=0.1)
plateModel.steps['Load Step'].Restart(frequency=1, numberIntervals=0, overlay=ON,
timeMarks=OFF)
```

The **StaticStep()** method has been modified to include the **initialInc** parameter. This sets the period of the initial increment. We set it to 0.1, which for a total step time of 1 is 10%.

The **Restart()** method tells Abaqus to write restart data to the .res file. It creates a **Restart** object which defines a restart request. It has no required arguments, only optional ones. **frequency** is an Int that specifies which increments the information will be written at. We set it to 1 indicating that restart information should be written at every increment. **numberIntervals** is an Int that specifies the number of intervals during a step at which restart information should be written. **overlay** is a Boolean which specifies whether the restart data should overwrite the one written at a previous increment or not. **timeMarks** is a Boolean specifying whether or not exact time marks should be used for writing during the analysis.

```
plateModel.ConcentratedForce(name='Concentrated Forces', createStepName='Load Step',
region=(vertices_for_concentrated_force,), cf3=-270000.0, distributionType=UNIFORM)
```

We modify the concentrated force to have a magnitude of 270,000.

```
# create the job

mdb.Job(name='PlateJobPlastic', model='Plastic Plate Bending Model', type=ANALYSIS,
description='Job simulates the plastic bending of a plate')

# run the job
mdb.jobs['PlateJobPlastic'].submit(consistencyChecking=OFF)

# do not return control till job is finished running
mdb.jobs['PlateJobPlastic'].waitForCompletion()
```

These statements have been modified to assign and use a new name for the job.

```
plate_viewport = session.Viewport(name='Plate Plastic Bending Results Viewport')
plate_Odb_Path = 'PlateJobPlastic.odb'
```

The name of the viewport and the .odb file have been changed in these statements to gel with previous changes.

```
# Displacement U3, the 3-component of displacement
plate_viewport.odbDisplay.setPrimaryVariable(variableLabel='UT',
outputPosition=NODAL, refinement=(COMPONENT, 'U3'),)
```

These statements tell Abaqus to display the translation in the 3-direction on the contour plot.

17.6 Python Script to simulate elastic springback

The following is the completed Python script to simulate the elastic springback of the bent plate when the load is removed. You can find it in the source code accompanying the book in You can run it by opening a new document in Abaqus/CAE (**File > New Model Database > With Standard/Explicit Model**) and running it with **File > Run Script...**

```
# If you know you will only run the restart analysis from within Abaqus/CAE
# using the same .cae file as the original analysis, then you can copy the old
# model 'Plate Bending Model' to a new one 'Plate Springback Model' using the
# following statement:
# mdb.Model(name='Plate Springback Model',
#                                   objectToCopy=mdb.models['Plate Bending Model'])
# and then add the relevant script lines after this
# On the other hand if you think you might later run the restart analysis from
# the command line, realize that you cannot copy the model to a new one
# Since you will not be within Abaqus/CAE viewing a .cae file. All you will be
# able to access is the output database (.odb) of the original analysis
# For this reason it is better to copy and paste the script of the original file
# and make modifications to it to create the new model
# Now all you need to do (once you've run the original analysis) is run this new
# script from the command line (and it will also work in Abaqus/CAE)
# It will access the output database file (.odb) of the original analysis to get
# what it needs


from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# ----------------------------------------------------------------------
# Create the model

mdb.Model(name='Plate Springback Model', modelType=STANDARD_EXPLICIT)
plateModel = mdb.models['Plate Springback Model']

plateModel.setValues(restartJob='PlateJobPlastic', restartStep='Load Step')

# ----------------------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the plate using the rectangle tool
plateProfileSketch = plateModel.ConstrainedSketch(name='Plate Sketch',
                                                   sheetSize=20)
plateProfileSketch.rectangle(point1=(0.0,0.0), point2=(5.0,3.0))

# b) Create a shell named "Plate" using the sketch
platePart=plateModel.Part(name='Plate', dimensionality=THREE_D,
                                         type=DEFORMABLE_BODY)
platePart.BaseShell(sketch=plateProfileSketch)

# ----------------------------------------------------------------------
# Create material

import material

# Create material Steel by assigning mass density, youngs modulus and
# poissons ratio
plateMaterial = plateModel.Material(name='Steel')
plateMaterial.Density(table=((7872, ),       ))
plateMaterial.Elastic(table=((200E9, 0.29), ))



# Read plastic properties from an external file
material_file_name = 'plate_bending_steel_plasticity_data.txt'
stress_data_list = []
strain_data_list = []

f = open(material_file_name)
for line in f:
    stress_strain_line = line
    stress_strain_list = stress_strain_line.split()
    stress_data_list.append(stress_strain_list[0])
    strain_data_list.append(stress_strain_list[1])
f.close()

plasticity_curve_list = []
for i in range(len(stress_data_list)):
    plasticity_curve_list.append((float(stress_data_list[i]),
                                  float(strain_data_list[i])))

plasticity_curve_tuple = tuple(plasticity_curve_list)

plateMaterial.Plastic(table=plasticity_curve_tuple)

# -------------------------------------------------------------------------
# Create homogeneous shell section of thickness 0.1m and assign the plate to it

import section

# Create a section to assign to the plate
plateSection = plateModel.HomogeneousShellSection(name='Plate Section',
                                                  material='Steel',
                                                  thicknessType=UNIFORM,
                                                  thickness=0.1)

#assign the plate to this section
plate_face_point = (2.5, 1.5, 0.0)
plate_face = platePart.faces.findAt((plate_face_point,))
plate_region = (plate_face,)

platePart.SectionAssignment(region=plate_region, sectionName='Plate Section',
                           offset=0.0, offsetType=MIDDLE_SURFACE, offsetField='')

# -------------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
plateAssembly = plateModel.rootAssembly
```

```
plateInstance = plateAssembly.Instance(name='Plate Instance', part=platePart,
                                                              dependent=ON)

# -----------------------------------------------------------------------
# Create the step

import step

# Create a static general step
plateModel.StaticStep(name='Load Step', previous='Initial',
                      description='Apply concentrated forces in this step',
                      nlgeom=ON, initialInc=0.1)

plateModel.steps['Load Step'].Restart(frequency=1, numberIntervals=0, overlay=ON,
                                                                 timeMarks=OFF)

# Create the new springback step
plateModel.StaticStep(name='Springback', previous='Load Step',
                      description='Remove load and allow elastic springback',
                      initialInc=0.1)

# -----------------------------------------------------------------------
# Create the field output request

# change the name of field output request 'F-Output-1' to 'Output Stresses and
# Displacements'
plateModel.fieldOutputRequests.changeKey(fromName='F-Output-1',
                                    toName='Output Stresses and Displacements')

# Since F-Output-1 is applied at the 'Apply Load' step by default, 'Selected
# Field Outputs' will be too
# We only need to set the required variables
plateModel.fieldOutputRequests['Output Stresses and Displacements'] \
                                              .setValues(variables=('S','UT'))

# -----------------------------------------------------------------------
# Create the history output request

# We don't want any history outputs so lets delete the existing one 'H-Output-1'
del plateModel.historyOutputRequests['H-Output-1']

# -----------------------------------------------------------------------
# Apply boundary conditions - fix one edge

fixed_edge = plateInstance.edges.findAt(((0.0, 1.5, 0.0), ))
fixed_edge_region=regionToolset.Region(edges=fixed_edge)

plateModel.DisplacementBC(name='FixEdge', createStepName='Initial',
                          region=fixed_edge_region,
                          u1=SET, u2=SET, u3=SET, ur1=SET, ur2=SET, ur3=SET,
                          amplitude=UNSET, distributionType=UNIFORM,
                          fieldName='', localCsys=None)
```

```
# Instead of using the displacements/rotations boundary condition and setting all
# six DOF to zero
# We could have just used the Encastre condition with the following statement
# plateModel.EncastreBC(name='Encaster edge', createStepName='Initial',
#                                                      region=fixed_edge_region)

# -----------------------------------------------------------------------
# Create vertices on which to apply concentrated forces by partitioning part

# Create the datum points
platePart.DatumPointByCoordinate(coords=(0.0, 1.0, 0.0))
platePart.DatumPointByCoordinate(coords=(0.0, 2.0, 0.0))
platePart.DatumPointByCoordinate(coords=(5.0, 1.0, 0.0))
platePart.DatumPointByCoordinate(coords=(5.0, 2.0, 0.0))

# Assign the datum points to variables
# Abaqus stores the 4 datum points in platePart.datums
# Since their keys may or may not start at zero, put the keys in an array sorted
# in ascending order
platePart_datums_keys = platePart.datums.keys()
platePart_datums_keys.sort()
plate_datum_point_1 = platePart.datums[platePart_datums_keys[0]]
plate_datum_point_2 = platePart.datums[platePart_datums_keys[1]]
plate_datum_point_3 = platePart.datums[platePart_datums_keys[2]]
plate_datum_point_4 = platePart.datums[platePart_datums_keys[3]]

# Select the entire face and partition it using two points
partition_face_pt = (2.5, 1.5, 0.0)
partition_face = platePart.faces.findAt((partition_face_pt,))
platePart.PartitionFaceByShortestPath(point1=plate_datum_point_1,
                                      point2=plate_datum_point_3,
                                      faces=partition_face)

# Now two faces exist, select the one that needs to be partitioned
partition_face_pt = (2.5, 2.0, 0.0)
partition_face = platePart.faces.findAt((partition_face_pt,))
platePart.PartitionFaceByShortestPath(point1=plate_datum_point_2,
                                      point2=plate_datum_point_4,
                                      faces=partition_face)

# Since the partitions have been created, vertices can be extracted
vertices_for_concentrated_force = plateInstance.vertices \
                              .findAt(((5.0, 1.0, 0.0),), ((5.0, 2.0, 0.0),),)

# -----------------------------------------------------------------------
# Apply concentrated forces

plateModel.ConcentratedForce(name='Concentrated Forces',
                             createStepName='Load Step',
                             region=(vertices_for_concentrated_force,),
                             cf3=-270000.0, distributionType=UNIFORM)
```

```
# Deactivate the concentrated force in the 'Springback' step where it has been
# propagated by default
plateModel.loads['Concentrated Forces'].deactivate('Springback')
# -------------------------------------------------------------------------
# Create the mesh

import mesh

# set element type
plate_mesh_region = plate_region

elemType1 = mesh.ElemType(elemCode=S8R, elemLibrary=STANDARD)

platePart.setElementType(regions=plate_mesh_region, elemTypes=(elemType1,))

# seed edges by number
mesh_edges_vertical = platePart.edges.findAt(((0.0, 0.5, 0.0), ),
                                             ((0.0, 1.5, 0.0), ),
                                             ((0.0, 2.5, 0.0), ),
                                             ((5.0, 0.5, 0.0), ),
                                             ((5.0, 1.5, 0.0), ),
                                             ((5.0, 2.5, 0.0), ))

mesh_edges_horizontal = platePart.edges.findAt(((2.5, 0.0, 0.0), ),
                                               ((2.5, 1.0, 0.0), ),
                                               ((2.5, 2.0, 0.0), ),
                                               ((2.5, 3.0, 0.0), ))

platePart.seedEdgeByNumber(edges=mesh_edges_vertical, number = 3)
platePart.seedEdgeByNumber(edges=mesh_edges_horizontal, number=10)

platePart.generateMesh()



# -------------------------------------------------------------------------
# Create and run the job

import job

# Create the job

mdb.Job(name='PlateSpringbackJob', model='Plate Springback Model', type=RESTART,
                              description='Job allows elastic springback')

# Run the job
mdb.jobs['PlateSpringbackJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['PlateSpringbackJob'].waitForCompletion()
```

```
##############################################################################
#--------------------------------------------------------------------------
#Post Processing
#--------------------------------------------------------------------------
##############################################################################

# -------------------------------------------------------
# Display deformed state with contours

import visualization

plate_viewport = session.Viewport(name='Plate Springback Results Viewport')
plate_Odb_Path = 'PlateSpringbackJob.odb'
an_odb_object = session.openOdb(name=plate_Odb_Path)
plate_viewport.setValues(displayedObject=an_odb_object)
plate_viewport.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))

# Displacement U3, the 3-component of displacement
plate_viewport.odbDisplay.setPrimaryVariable(variableLabel='UT',
                                             outputPosition=NODAL,
                                             refinement=(COMPONENT, 'U3'),)

# -----------------------------------------------------------------------
# Report stresses in descending order

import odbAccess

# The main session viewport must be set to the odb object using the following
# line. If not you might receive an error message that states
# "There are no active entities. No report has been generated."
session.viewports['Viewport: 1'].setValues(displayedObject=an_odb_object)

# Set the option to display the reported quantity (in our case the stresses) in
# descending order
session.fieldReportOptions.setValues(sort=DESCENDING)

# Name the report and give it a path. If you do not assign a path (as is done
# here) it will be stored in the default abaqus temporary directory
report_name_and_path='PlateStresses.rpt'

# Write the field report outputting the Mises stresses
session.writeFieldReport(fileName=report_name_and_path, append=OFF,
                 sortItem='S.Mises', odb=an_odb_object, step=0, frame=1,
                 outputPosition=INTEGRATION_POINT,
                 variable=(('S', INTEGRATION_POINT, ((INVARIANT, 'Mises'), )), ))



# -----------------------------------------------------------------------
# Display the first frame of the restart analysis so you can compare it to the
# last frame of the original analysis (they should be the same)
# Also offset the viewport to reveal the viewport of the original analysis
# behind it

plate_viewport.odbDisplay.setFrame(step='Springback', frame=0)
plate_viewport.offset(deltaX=50.0, deltaY=5.0)

# -----------------------------------------------------------------------
# The viewport from the original simulation loses its contour plot when the
# restart script is run because Abaqus closes the output database of that job.
# Hence redisplay the contour on it so that the last frame of the first analysis
# is displayed (and can be compared to the first frame of the restart analysis.

plate_viewport = session.viewports['Plate Plastic Bending Results Viewport']
plate_Odb_Path = 'PlateJobPlastic.odb'
an_odb_object = session.openOdb(name=plate_Odb_Path)
plate_viewport.setValues(displayedObject=an_odb_object)
plate_viewport.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))
plate_viewport.odbDisplay.setPrimaryVariable(variableLabel='UT',
                           outputPosition=NODAL, refinement=(COMPONENT, 'U3'),)


# -----------------------------------------------------------------------
# Change the fonts in the viewport annotation options to make the legend, title
# block and stateblocks more readable

plate_viewport = session.viewports['Plate Plastic Bending Results Viewport']
plate_viewport.viewportAnnotationOptions \
          .setValues(legendFont='-*-verdana-medium-r-normal-*-*-120-*-*-p-*-*-*')
plate_viewport.viewportAnnotationOptions \
   .setValues(titleFont='-*-times new roman-medium-i-normal-*-*-140-*-*-p-*-*-*',
              stateFont='-*-times new roman-medium-i-normal-*-*-140-*-*-p-*-*-*')

plate_viewport = session.viewports['Plate Springback Results Viewport']
plate_viewport.viewportAnnotationOptions \
        .setValues(legendFont='-*-verdana-medium-r-normal-*-*-120-*-*-p-*-*-*')
plate_viewport.viewportAnnotationOptions \
   .setValues(titleFont='-*-times new roman-medium-i-normal-*-*-140-*-*-p-*-*-*',
    stateFont='-*-times new roman-medium-i-normal-*-*-140-*-*-p-*-*-*')
```

In a restart analysis the new model remains almost identical to the original one up until the point of the restart, although a few changes are allowed such as creating more sets in the assembly. Therefore the bulk of this script is a copy of the previous one and only the differences will be highlighted..

```
mdb.Model(name='Plate Springback Model', modelType=STANDARD_EXPLICIT)
plateModel = mdb.models['Plate Springback Model']

plateModel.setValues(restartJob='PlateJobPlastic', restartStep='Load Step')
```

Most of the scripts we've written so far begin with renaming the existing default model 'Model-1' using the **changeKey()** method. In this script however we create a new model which is a copy of the original with changes. To create a new model we use the **Model()** method. **Model()** creates a **Model** object. Its only required argument is a String which specifies the repository key. Here we name the model 'Plate Springback Model'. One of the optional arguments is **modelType** which specifies the type of analysis which will be carried out in this model. The possible options are **STANDARD_EXPLICIT** and **CFD.**

The **(Model).setValues()** method is then used to indicate which job the restart analysis should be continued from. **setValues()** modifies an existing model object. It has no required arguments, but a number of optional ones, of which we use **restartJob** and **restartStep**. **restartJob** is a String which is the name of the job that originally generated the restart data. **restartStep** is the name of the step of the model where the restart analysis will begin.

```
# Create the new springback step
plateModel.StaticStep(name='Springback', previous='Load Step', description='Remove
load and allow elastic springback', initialInc=0.1)
```

This statement creates a new static step called 'Springback' which follows the 'Load Step'. It sets the initial increment to 0.1 using the **initialInc** parameter which works out to 10% since the total time is 1.

```
# Deactivate the concentrated force in the 'Springback' step where it has been
# propagated by default
plateModel.loads['Concentrated Forces'].deactivate('Springback')
```

This statement deactivates the force in our new 'Springback' step. It uses the **deactivate()** method of the Load object, which deactivates the load in the specified step and in all subsequent steps. It has one required argument **stepName** which is a String specifying the name of the step in which to deactivate the load. This statement is the equivalent of opening the load manager in Abaqus/CAE and deactivating a load in a particular step.

```
mdb.Job(name='PlateSpringbackJob', model='Plate Springback Model', type=RESTART,
                                   description='Job allows elastic springback')

# Run the job
mdb.jobs['PlateSpringbackJob'].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs['PlateSpringbackJob'].waitForCompletion()
```

These statements have been modified to give the job a new name 'PlateSpringbackJob' and submit this job.

```
import visualization

plate_viewport = session.Viewport(name='Plate Springback Results Viewport')
plate_Odb_Path = 'PlateSpringbackJob.odb'
```

These statements have also been modified to change the name of the viewport and point to the output database of the newly created job.

```
# ---------------------------------------------------------------------
# Display the first frame of the restart analysis so you can compare it to the
# last frame of the original analysis (they should be the same)
# Also offset the viewport to reveal the viewport of the original analysis
# behind it

plate_viewport.odbDisplay.setFrame(step='Springback', frame=0)
plate_viewport.offset(deltaX=50.0, deltaY=5.0)

# ---------------------------------------------------------------------
# The viewport from the original simulation loses its contour plot when the
# restart script is run because Abaqus closes the output database of that job.
# Hence redisplay the contour on it so that the last frame of the first analysis
# is displayed (and can be compared to the first frame of the restart analysis.

plate_viewport = session.viewports['Plate Plastic Bending Results Viewport']
plate_Odb_Path = 'PlateJobPlastic.odb'
an_odb_object = session.openOdb(name=plate_Odb_Path)
plate_viewport.setValues(displayedObject=an_odb_object)
plate_viewport.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))
plate_viewport.odbDisplay.setPrimaryVariable(variableLabel='UT',
                             outputPosition=NODAL, refinement=(COMPONENT, 'U3'),)


# ---------------------------------------------------------------------
# Change the fonts in the viewport annotation options to make the legend, title
# block and stateblocks more readable

plate_viewport = session.viewports['Plate Plastic Bending Results Viewport']
plate_viewport.viewportAnnotationOptions \
          .setValues(legendFont='-*-verdana-medium-r-normal-*-*-120-*-*-p-*-*-*')
plate_viewport.viewportAnnotationOptions \
   .setValues(titleFont='-*-times new roman-medium-i-normal-*-*-140-*-*-p-*-*-*',
              stateFont='-*-times new roman-medium-i-normal-*-*-140-*-*-p-*-*-*')

plate_viewport = session.viewports['Plate Springback Results Viewport']
plate_viewport.viewportAnnotationOptions \
        .setValues(legendFont='-*-verdana-medium-r-normal-*-*-120-*-*-p-*-*-*')
```

```
plate_viewport.viewportAnnotationOptions \
   .setValues(titleFont='-*-times new roman-medium-i-normal-*-*-140-*-*-p-*-*-*',
    stateFont='-*-times new roman-medium-i-normal-*-*-140-*-*-p-*-*-*')
```

```
plate_viewport.odbDisplay.setFrame(step='Springback', frame=0)
```

The **setFrame()** method is used to specify the step and the frame of the **OdbDisplay** object. It has 2 required arguments **step** and **frame**. **step** is a String specifying the name of the step. **frame** is an Int specifying which frame in the selected step to display.

```
plate_viewport.offset(deltaX=50.0, deltaY=5.0)
```

The **offset()** method modifies the X and Y coordinates of the current viewport by displacing them a certain distance. The arguments **delatX** and **deltaY** are Floats specifying the X and Y offsets of the viewport origin in millimeters.

```
# -------------------------------------------------------------------------
# The viewport from the original simulation loses its contour plot when the
# restart script is run because Abaqus closes the output database of that job.
# Hence redisplay the contour on it so that the last frame of the first analysis
# is displayed (and can be compared to the first frame of the restart analysis.

plate_viewport = session.viewports['Plate Plastic Bending Results Viewport']
plate_Odb_Path = 'PlateJobPlastic.odb'
an_odb_object = session.openOdb(name=plate_Odb_Path)
plate_viewport.setValues(displayedObject=an_odb_object)
plate_viewport.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))
plate_viewport.odbDisplay.setPrimaryVariable(variableLabel='UT',
                              outputPosition=NODAL, refinement=(COMPONENT, 'U3'),)
```

You've seen these methods used on numerous occasions before. The comment at the top of this chunk of code explains the reason for this. When the script is run, Abaqus closes the output database of the original job. This causes the results of the original analysis displayed in the viewport to disappear. All we are doing here is redisplaying it.

```
# -------------------------------------------------------------------------
# Change the fonts in the viewport annotation options to make the legend, title
# block and stateblocks more readable

plate_viewport = session.viewports['Plate Plastic Bending Results Viewport']
plate_viewport.viewportAnnotationOptions \
          .setValues(legendFont='-*-verdana-medium-r-normal-*-*-120-*-*-p-*-*-*')
plate_viewport.viewportAnnotationOptions \
   .setValues(titleFont='-*-times new roman-medium-i-normal-*-*-140-*-*-p-*-*-*',
             stateFont='-*-times new roman-medium-i-normal-*-*-140-*-*-p-*-*-*')

plate_viewport = session.viewports['Plate Springback Results Viewport']
```

```
plate_viewport.viewportAnnotationOptions \
        .setValues(legendFont='-*-verdana-medium-r-normal-*-*-120-*-*-p-*-*-*')
plate_viewport.viewportAnnotationOptions \
    .setValues(titleFont='-*-times new roman-medium-i-normal-*-*-140-*-*-p-*-*-*',
     stateFont='-*-times new roman-medium-i-normal-*-*-140-*-*-p-*-*-*')
```

The **ViewportAnnotationOptions** object stores the current settings dictating how annotations will be rendered in the viewport. The **viewportAnnotationOptions.setValues()** method modifies the **ViewportAnnotationOptions** object. It has a number of optional arguments. The ones we use are **legendFont**, **titleFont** and **stateFont**. **legendFont** is a String specifying the font to be used in the legend. **titleFont** is a String specifying the font to be used in the title. And **stateFont** is a String specifying the font to be used in the state block.

The names of the font used by Abaqus such as '-*-verdana-medium-r-normal-*-*-120-*-*-p-*-*-*' follow a strange and complicated format. In my experience it is best to perform these steps in Abaqus/CAE and look at the replay file, or perform them within a macro and then look at the Python script generated. This will give you the font String that you require.

17.7 Python Script to combine the output databases

The following listing is the completed Python script to read the output databases of the original and restart analyses, and combine them to produce a new .odb with the results of both. Once this .odb has been made, an animation of the plate bending under load and then springing back elastically will be written to the hard drive. You can find it in the source code accompanying the book in plate_bending_od_combine_restart.py. You can run it by opening a new model in Abaqus/CAE (**File > New Model database > With Standard/Explicit Model**) and running it with **File > Run Script...**

```
# ***********************************************************************
# CLASS DEFINITION
# ***********************************************************************

class FieldOutputDisplacements:

    def __init__(self, node_label_list, displacement_tuple_list):
        self.field_data_node_labels = node_label_list
        self.field_data_xyz_displacements  = displacement_tuple_list

# ***********************************************************************

from abaqus import *
from odbAccess import *
from abaqusConstants import *
import visualization

# ************************************************************************
# READ EXISTING OUTPUT DATABASE (.ODB)
# ************************************************************************
plate_Odb_Path = 'PlateJobPlastic.odb'
plateOdb = session.openOdb(name=plate_Odb_Path)

restart_Odb_Path = 'PlateSpringbackJob.odb'
restartOdb = session.openOdb(name=restart_Odb_Path)


node_labels_and_coords = []
# This will hold node labels and coordinates as
# [(label1, xcoord1, ycoord1, zcoord1), (label2, xcoord2, ycoord2, zcoord2), ...]
no_of_nodes = len(plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                                  .values[0].instance.nodes)

for i in range(no_of_nodes):
    node_label=plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                              .values[1].instance.nodes[i].label
    node_x_coord = plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                        .values[0].instance.nodes[i].coordinates[0]
    node_y_coord = plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                        .values[0].instance.nodes[i].coordinates[1]
    node_z_coord = plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                        .values[0].instance.nodes[i].coordinates[2]

    node_labels_and_coords.append((node_label, node_x_coord, node_y_coord,
                                                              node_z_coord))



element_labels_and_coords = []
# This contains element labels and connectivity
# (label, conn1, conn2, conn3, conn4)
no_of_elements = len(plateOdb.steps['Load Step'].frames[1] \
                              .fieldOutputs['UT'].values[0].instance.elements)

for j in range(no_of_elements):
    element_label = plateOdb.steps['Load Step'].frames[0].fieldOutputs['UT'] \
                                           .values[0].instance.elements[1].label
    element_connector_1 = plateOdb.steps['Load Step'].frames[0] \
               .fieldOutputs['UT'].values[0].instance.elements[j].connectivity[0]
    element_connector_2 = plateOdb.steps['Load Step'].frames[0] \
               .fieldOutputs['UT'].values[0].instance.elements[j].connectivity[1]
    element_connector_3 = plateOdb.steps['Load Step'].frames[0] \
               .fieldOutputs['UT'].values[0].instance.elements[j].connectivity[2]
    element_connector_4 = plateOdb.steps['Load Step'].frames[0] \
               .fieldOutputs['UT'].values[0].instance.elements[j].connectivity[3]
    element_labels_and_coords.append((element_label,
                                      element_connector_1,
                                      element_connector_2,
                                      element_connector_3,
                                      element_connector_4))



fieldoutput_node_labels_load_step = []
fieldoutput_node_disp_load_step =[]
no_of_field_output_ut_values = len(plateOdb.steps['Load Step'].frames[0] \
                                             .fieldOutputs['UT'].values)
no_of_frames = len(plateOdb.steps['Load Step'].frames)

field_output_frames_load_step = []

for p in range(no_of_frames):

    for q in range(no_of_field_output_ut_values):
        field_label = plateOdb.steps['Load Step'].frames[p] \
                                   .fieldOutputs['UT'].values[q].nodeLabel
        field_data_x = plateOdb.steps['Load Step'].frames[p] \
                                   .fieldOutputs['UT'].values[q].data[0]
        field_data_y = plateOdb.steps['Load Step'].frames[p] \
                                   .fieldOutputs['UT'].values[q].data[1]
        field_data_z = plateOdb.steps['Load Step'].frames[p] \
                                   .fieldOutputs['UT'].values[q].data[2]

        fieldoutput_node_labels_load_step.append(field_label)
        fieldoutput_node_disp_load_step.append((field_data_x, field_data_y,
                                                field_data_z))

    field_output_frames_load_step \
        .append(FieldOutputDisplacements(fieldoutput_node_labels_load_step,
                                   fieldoutput_node_disp_load_step))
    fieldoutput_node_labels_load_step = []
    fieldoutput_node_disp_load_step = []

# Make a list of the step times for frames of 'Load Step'
frame_step_times_load_step = []
for x in range(len(field_output_frames_load_step)):
    frame_step_times_load_step.append(plateOdb.steps['Load Step'].frames[x] \
                                                          .frameValue)


fieldoutput_node_labels_springback_step = []
fieldoutput_node_disp_springback_step =[]
no_of_field_output_ut_values = len(restartOdb.steps['Springback'].frames[0] \
                                             .fieldOutputs['UT'].values)
# This will be same as no_of_nodes since translations 'UT' are obtained at the
# nodes..
```

```
no_of_frames = len(restartOdb.steps['Springback'].frames)

field_output_frames_springback_step = []

for p in range(no_of_frames):

    for q in range(no_of_field_output_ut_values):
        field_label = restartOdb.steps['Springback'].frames[p] \
                                        .fieldOutputs['UT'].values[q].nodeLabel
        field_data_x = restartOdb.steps['Springback'] \
                                  .frames[p].fieldOutputs['UT'].values[q].data[0]
        field_data_y = restartOdb.steps['Springback'] \
                                  .frames[p].fieldOutputs['UT'].values[q].data[1]
        field_data_z = restartOdb.steps['Springback'] \
                                  .frames[p].fieldOutputs['UT'].values[q].data[2]

        fieldoutput_node_labels_springback_step.append(field_label)
        fieldoutput_node_disp_springback_step.append((field_data_x, field_data_y,
                                                                      field_data_z))

    field_output_frames_springback_step \
        .append(FieldOutputDisplacements(fieldoutput_node_labels_springback_step,
                                         fieldoutput_node_disp_springback_step))
    fieldoutput_node_labels_springback_step = []
    fieldoutput_node_disp_springback_step = []

# Make a list of the step times for frames of 'Springback' step
frame_step_times_springback_step = []
for x in range(len(field_output_frames_springback_step)):
    frame_step_times_springback_step.append(restartOdb.steps['Springback'] \
                                                           .frames[x].frameValue)

# **************************************************************************




# **************************************************************************
# WRITE TO NEW OUTPUT DATABASE (.ODB)
# **************************************************************************

import os

if os.path.exists('Plate_plastic_bending_and_springback.odb'):
    os.remove('Plate_plastic_bending_and_springback.odb')

new_odb = Odb('Plate Bending and Springback combined', analysisTitle='',
              description='', path='Plate_plastic_bending_and_springback.odb')

plate_part = new_odb.Part(name='Plate Part', embeddedSpace=THREE_D,
                                              type=DEFORMABLE_BODY)
```

```
plate_part.addNodes(nodeData=node_labels_and_coords, nodeSetName='node set')
del node_labels_and_coords

plate_part.addElements(elementData=element_labels_and_coords, type='S4R',
                                          elementSetName='element set')
del element_labels_and_coords


# Instance the part

plate_instance = new_odb.rootAssembly.Instance(name='Plate Instance',
                                          object=plate_part)

# Create a step 'Load Step'

load_step = new_odb.Step(name = 'Load Step', description='', domain=TIME,
                                                  timePeriod=1.0)

# Create frames with nodal displacements
for i in range(len(field_output_frames_load_step)):

    single_frame_load_step = load_step.Frame(incrementNumber=i,
                                 frameValue=frame_step_times_load_step[i],
                                 description='')

    disp_field_load_step = single_frame_load_step.FieldOutput(name='UT',
                                 description='Displacements and trans',
                                 type=VECTOR)

    disp_field_load_step.addData(position=NODAL, instance=plate_instance,
               labels=field_output_frames_load_step[i].field_data_node_labels,
               data=field_output_frames_load_step[i].field_data_xyz_displacements)

del field_output_frames_load_step
del frame_step_times_load_step

# Create a step 'Springback'
springback_step = new_odb.Step(name = 'Springback', description='', domain=TIME,
                                                     timePeriod=1.0)

# Create frames with nodal displacements
for i in range(len(field_output_frames_springback_step)):
    single_frame_springback_step = springback_step.Frame(incrementNumber=i,
                           frameValue=frame_step_times_springback_step[i],
                           description='')

    disp_field_springback_step = single_frame_springback_step \
                        .FieldOutput(name='UT',
                                  description='Displacements and trans',
                                  type=VECTOR)

    disp_field_springback_step.addData(position=NODAL, instance=plate_instance,
          labels=field_output_frames_springback_step[i].field_data_node_labels,
          data=field_output_frames_springback_step[i].field_data_xyz_displacements)

del field_output_frames_springback_step
del frame_step_times_springback_step

# Make this the default field for visualization
load_step.setDefaultDeformedField(disp_field_load_step)

# Save the odb in order to reopen it in abaqus/viewer

new_odb.save()
new_odb.close()

# **************************************************************************


# **************************************************************************
# CREATE AN ANIMATION
# **************************************************************************

import visualization
import animation

plate_Odb_Path = 'Plate_plastic_bending_and_springback.odb'
odb_object = session.openOdb(name=plate_Odb_Path)

session.viewports['Viewport: 1'].setValues(displayedObject=odb_object)
session.viewports['Viewport: 1'].odbDisplay.display \
                                        .setValues(plotState=(CONTOURS_ON_DEF, ))
session.viewports['Viewport: 1'].odbDisplay \
                               .setPrimaryVariable(variableLabel='UT',
                                                   outputPosition=NODAL,
                                                   refinement=(COMPONENT, 'UT3'),)

session.animationController.setValues(animationType=TIME_HISTORY,
                                      viewports=('Viewport: 1', ))
session.animationController.play(duration=UNLIMITED)
session.imageAnimationOptions.setValues(vpDecorations=ON, vpBackground=OFF,
                                        compass=ON, timeScale=1)
session.writeImageAnimation(fileName='plate_bend_springback_animation',
                            format=AVI,
                            canvasObjects=(session.viewports['Viewport: 1'], ))
```

17.8 Examining the Script

Let's take a closer look at this script.

17.8.1 Class Definition.

This block creates a class.

```
# ***************************************************************************
# CLASS DEFINITION
# ***************************************************************************

class FieldOutputDisplacements:

    def __init__(self, node_label_list, displacement_tuple_list):
        self.field_data_node_labels = node_label_list
        self.field_data_xyz_displacements  = displacement_tuple_list

# ***************************************************************************
```

You learnt about classes in Chapter 3, section 3.7. Now you see an example of a class in action. We will use instances of this class as a container for nodal displacements. Each frame of the analysis (both load step and springback step) will have one instance of this class associated with it, containing the node labels and displacements for that frame.

```
class FieldOutputDisplacements:
```

The class keyword indicates the beginning of a class which we name **FieldOutputDisplacements.**

```
    def __init__(self, node_label_list, displacement_tuple_list):
```

This class has one function definition, identified by the keyword **def**. The function is called __**init**__, and in Python any function in a class with the name __**init**__ is an initialization function and will be called the moment an object of that class is created. It is similar to a constructor in C, C++, Java, C# and a host of other object oriented language, although technically speaking there are some differences (but we won't go into that in this book as it is of no consequence to us).

The first argument in the function definition is **self. self** is a reference to the instance of the object that has called the method. It is similar to the **this** keyword in C++. In Python every function (or more accurately every non-static function) in a class must have **self** as its first argument. Strictly speaking you don't have to name it **self** and could give it another name, however **self** is accepted as convention in the Python world and it would make your code much less readable to other Python programmers or script writers if you used some other name.

The next 2 arguments **node_label_list** and **displacement_tuple_list** are expected to be passed to the object when an instance is first created.

```
self.field_data_node_labels = node_label_list
self.field_data_xyz_displacements  = displacement_tuple_list
```

These statements assign the values passed as the arguments **node_label_list** and **displacement_tuple_list** to the objects variables **field_data_node_labels** and **field_data_xyz_displacements**. Thus every instance of the **FieldOutputDisplacement** will be supplied a list of node labels and a list of x, y and z displacement tuples, and these will be stored in the instance's **field_data_node_labels** and **field_data_xyz_displacements** variables. Later in the script we will create an instance of this **FieldOutputDisplacements()** class for each frame of the analysis to store the node labels and displacements associated with it.

17.8.2 Read data from output databases

The following block obtains information from the output databases.

```
# ***************************************************************************
# READ EXISTING OUTPUT DATABASE (.ODB)
# ***************************************************************************
plate_Odb_Path = 'PlateJobPlastic.odb'
plateOdb = session.openOdb(name=plate_Odb_Path)

restart_Odb_Path = 'PlateSpringbackJob.odb'
restartOdb = session.openOdb(name=restart_Odb_Path)


node_labels_and_coords = []
# This will hold node labels and coordinates as
# [(label1, xcoord1, ycoord1, zcoord1), (label2, xcoord2, ycoord2, zcoord2), ...]
no_of_nodes = len(plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                                   .values[0].instance.nodes)

for i in range(no_of_nodes):
    node_label=plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                           .values[1].instance.nodes[i].label
    node_x_coord = plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                   .values[0].instance.nodes[i].coordinates[0]
    node_y_coord = plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                        .values[0].instance.nodes[i].coordinates[1]
    node_z_coord = plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                        .values[0].instance.nodes[i].coordinates[2]

    node_labels_and_coords.append((node_label, node_x_coord, node_y_coord,
                                                           node_z_coord))



element_labels_and_coords = []
# This contains element labels and connectivity
# (label, conn1, conn2, conn3, conn4)
no_of_elements = len(plateOdb.steps['Load Step'].frames[1] \
                          .fieldOutputs['UT'].values[0].instance.elements)

for j in range(no_of_elements):
    element_label = plateOdb.steps['Load Step'].frames[0].fieldOutputs['UT'] \
                                       .values[0].instance.elements[1].label
    element_connector_1 = plateOdb.steps['Load Step'].frames[0] \
               .fieldOutputs['UT'].values[0].instance.elements[j].connectivity[0]
    element_connector_2 = plateOdb.steps['Load Step'].frames[0] \
               .fieldOutputs['UT'].values[0].instance.elements[j].connectivity[1]
    element_connector_3 = plateOdb.steps['Load Step'].frames[0] \
               .fieldOutputs['UT'].values[0].instance.elements[j].connectivity[2]
    element_connector_4 = plateOdb.steps['Load Step'].frames[0] \
               .fieldOutputs['UT'].values[0].instance.elements[j].connectivity[3]

    element_labels_and_coords.append((element_label,
                                      element_connector_1,
                                      element_connector_2,
                                      element_connector_3,
                                      element_connector_4))



fieldoutput_node_labels_load_step = []
fieldoutput_node_disp_load_step =[]
no_of_field_output_ut_values = len(plateOdb.steps['Load Step'].frames[0] \
                                              .fieldOutputs['UT'].values)
no_of_frames = len(plateOdb.steps['Load Step'].frames)

field_output_frames_load_step = []

for p in range(no_of_frames):

    for q in range(no_of_field_output_ut_values):
        field_label = plateOdb.steps['Load Step'].frames[p] \
                                    .fieldOutputs['UT'].values[q].nodeLabel
        field_data_x = plateOdb.steps['Load Step'].frames[p] \
                                    .fieldOutputs['UT'].values[q].data[0]
        field_data_y = plateOdb.steps['Load Step'].frames[p] \
                                    .fieldOutputs['UT'].values[q].data[1]
        field_data_z = plateOdb.steps['Load Step'].frames[p] \
                                         .fieldOutputs['UT'].values[q].data[2]

        fieldoutput_node_labels_load_step.append(field_label)
        fieldoutput_node_disp_load_step.append((field_data_x, field_data_y,
                                                          field_data_z))

    field_output_frames_load_step \
            .append(FieldOutputDisplacements(fieldoutput_node_labels_load_step,
                                          fieldoutput_node_disp_load_step))
    fieldoutput_node_labels_load_step = []
    fieldoutput_node_disp_load_step = []

# Make a list of the step times for frames of 'Load Step'
frame_step_times_load_step = []
for x in range(len(field_output_frames_load_step)):
    frame_step_times_load_step.append(plateOdb.steps['Load Step'].frames[x] \
                                                                .frameValue)


fieldoutput_node_labels_springback_step = []
fieldoutput_node_disp_springback_step =[]
no_of_field_output_ut_values = len(restartOdb.steps['Springback'].frames[0] \
                                                  .fieldOutputs['UT'].values)
# This will be same as no_of_nodes since translations 'UT' are obtained at the
# nodes..

no_of_frames = len(restartOdb.steps['Springback'].frames)

field_output_frames_springback_step = []

for p in range(no_of_frames):

    for q in range(no_of_field_output_ut_values):
        field_label = restartOdb.steps['Springback'].frames[p] \
                                     .fieldOutputs['UT'].values[q].nodeLabel
        field_data_x = restartOdb.steps['Springback'] \
                               .frames[p].fieldOutputs['UT'].values[q].data[0]
        field_data_y = restartOdb.steps['Springback'] \
                               .frames[p].fieldOutputs['UT'].values[q].data[1]
        field_data_z = restartOdb.steps['Springback'] \
                               .frames[p].fieldOutputs['UT'].values[q].data[2]

        fieldoutput_node_labels_springback_step.append(field_label)
        fieldoutput_node_disp_springback_step.append((field_data_x, field_data_y,
                                                                field_data_z))

    field_output_frames_springback_step \
        .append(FieldOutputDisplacements(fieldoutput_node_labels_springback_step,
                                      fieldoutput_node_disp_springback_step))
    fieldoutput_node_labels_springback_step = []
    fieldoutput_node_disp_springback_step = []
```

```
# Make a list of the step times for frames of 'Springback' step
frame_step_times_springback_step = []
for x in range(len(field_output_frames_springback_step)):
    frame_step_times_springback_step.append(restartOdb.steps['Springback'] \
                                                   .frames[x].frameValue)
```

```
plate_Odb_Path = 'PlateJobPlastic.odb'
plateOdb = session.openOdb(name=plate_Odb_Path)

restart_Odb_Path = 'PlateSpringbackJob.odb'
restartOdb = session.openOdb(name=restart_Odb_Path)
```

You once again use the **session.openOdb()** method which was first encountered and explained in the Cantilever Beam example, section 4.3.14 on page 89. Here we open the two .odbs created by the original analysis and the restart analysis and assign them to the variables 'plateOdb' and 'restartOdb'.

```
node_labels_and_coords = []
# This will hold node labels and coordinates as
# [(label1, xcoord1, ycoord1, zcoord1), (label2, xcoord2, ycoord2, zcoord2), ...]
no_of_nodes = len(plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                              .values[0].instance.nodes)

for i in range(no_of_nodes):
    node_label=plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                          .values[1].instance.nodes[i].label
    node_x_coord = plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                   .values[0].instance.nodes[i].coordinates[0]
    node_y_coord = plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                   .values[0].instance.nodes[i].coordinates[1]
    node_z_coord = plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'] \
                                   .values[0].instance.nodes[i].coordinates[2]

    node_labels_and_coords.append((node_label, node_x_coord, node_y_coord,
                                                            node_z_coord))
```

In order to reconstruct the plate in a new output database we will need its nodal coordinates from this output database. What we need are the node labels, and the x, y and z coordinates of the nodes. We will store these in the variable **node_labels_and_coords** as a list of tuples ie [(label1, xcoord1, ycoord1, zcoord1), (label2, xcoord2, ycoord2, zcoord2), ...]

The nodes of the part instance can be accessed using **plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'].values[0].instance.nodes**, and we can use the **len()** method to get the number of nodes in the part.

We then access the label of the i'th node using

```
plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'].values[1].instance
                                                                 .nodes[i].label
```

and the x, y and z coordinates of the i'th node using

```
plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'].values[0].instance
                                                        .nodes[i].coordinates[0]
plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'].values[0].instance
                                                        .nodes[i].coordinates[1]
plateOdb.steps['Load Step'].frames[1].fieldOutputs['UT'].values[0].instance
                                                        .nodes[i].coordinates[2]
```

Finally we use the **append()** method to append these to the list.

```
element_labels_and_coords = []
# This contains element labels and connectivity
# (label, conn1, conn2, conn3, conn4)
no_of_elements = len(plateOdb.steps['Load Step'].frames[1] \
                                 .fieldOutputs['UT'].values[0].instance.elements)

for j in range(no_of_elements):
    element_label = plateOdb.steps['Load Step'].frames[0].fieldOutputs['UT'] \
                                                 .values[0].instance.elements[1].label
    element_connector_1 = plateOdb.steps['Load Step'].frames[0] \
               .fieldOutputs['UT'].values[0].instance.elements[j].connectivity[0]
    element_connector_2 = plateOdb.steps['Load Step'].frames[0] \
               .fieldOutputs['UT'].values[0].instance.elements[j].connectivity[1]
    element_connector_3 = plateOdb.steps['Load Step'].frames[0] \
               .fieldOutputs['UT'].values[0].instance.elements[j].connectivity[2]
    element_connector_4 = plateOdb.steps['Load Step'].frames[0] \
               .fieldOutputs['UT'].values[0].instance.elements[j].connectivity[3]

    element_labels_and_coords.append((element_label,
                                      element_connector_1,
                                      element_connector_2,
                                      element_connector_3,
                                      element_connector_4))
```

We repeat a similar process for the elements. The data required for the elements when building a new output database is the label and the connectivity of the elements (4 connectors nodes of the rectangular element).

We access the label of the j'th element using

```
element_label = plateOdb.steps['Load Step'].frames[0].fieldOutputs['UT'].
                                              values[0].instance.elements[1].label
```

and the first connector of the j'th node using

```
element_connector_1 = plateOdb.steps['Load Step'].frames[0].fieldOutputs['UT']
                                   .values[0].instance.elements[j].connectivity[0]
```

In the statements

```
fieldoutput_node_labels_load_step = []
fieldoutput_node_disp_load_step =[]
```

we store the labels of the nodes corresponding to field outputs in **fieldoutput_node_labels_load_step** in the form *(label1, label2, ...)* and the x, y and z displacements for a frame in the load step in *fieldoutput_node_disp_load_step* in the form *((UT-x_1, UT-y_1, UT-z_1), (UT-x_2, UT-y_2, UT-z_2), ...)*

```
no_of_field_output_ut_values = len(plateOdb.steps['Load Step'].frames[0] \
                                                 .fieldOutputs['UT'].values)
```

The displacement (UT) values can be accessed using **plateOdb.steps['Load Step'].frames[0].fieldOutputs['UT'].values**. The number of values available in the output database is obtained by using the **len()** method on it. Since nodal displacement is stored for the nodes, this number will be the same as the number of nodes in the part instance.

```
no_of_frames = len(plateOdb.steps['Load Step'].frames)
```

We find the number of frames in the load step for use in a loop.

```
field_output_frames_load_step = []
```

This variable will hold a list of **FieldOutputDisplacements** objects (the class we created at the beginning of the script).

```
for p in range(no_of_frames):

    for q in range(no_of_field_output_ut_values):
        field_label = plateOdb.steps['Load Step'].frames[p] \
                                    .fieldOutputs['UT'].values[q].nodeLabel
        field_data_x = plateOdb.steps['Load Step'].frames[p] \
                                        .fieldOutputs['UT'].values[q].data[0]
        field_data_y = plateOdb.steps['Load Step'].frames[p] \
                                        .fieldOutputs['UT'].values[q].data[1]
        field_data_z = plateOdb.steps['Load Step'].frames[p] \
                                        .fieldOutputs['UT'].values[q].data[2]

        fieldoutput_node_labels_load_step.append(field_label)
        fieldoutput_node_disp_load_step.append((field_data_x, field_data_y,
                                                      field_data_z))

    field_output_frames_load_step \
          .append(FieldOutputDisplacements(fieldoutput_node_labels_load_step,
                                          fieldoutput_node_disp_load_step))
    fieldoutput_node_labels_load_step = []
    fieldoutput_node_disp_load_step = []
```

The inner loop extracts the node labels and the x, y and z displacements for each frame. These are placed in the variables **fieldoutput_node_labels_load_step** and **fieldoutput_node_disp_load_step**.

The outer loop runs through all the frames of the load step. For each frame it uses the node labels and data gathered by the inner loop to create a **FieldOutputDisplacements** object which it adds to the variable **field_output_frames_load_step**.

```
# Make a list of the step times for frames of 'Load Step'
frame_step_times_load_step = []
for x in range(len(field_output_frames_load_step)):
    frame_step_times_load_step.append(plateOdb.steps['Load Step'].frames[x] \
                                                           .frameValue))
```

We need to know the step times of the frames. We obtain these using **plateOdb.steps['Load Step'].frames[x].frameValue** where **x** is the frame number, and store them in the variable **frame_step_times_load_step** for use when constructing the new odb.

```
fieldoutput_node_labels_springback_step = []
fieldoutput_node_disp_springback_step =[]
no_of_field_output_ut_values = len(restartOdb.steps['Springback'].frames[0] \
                                                .fieldOutputs['UT'].values)
# This will be same as no_of_nodes since translations 'UT' are obtained at the
# nodes..

no_of_frames = len(restartOdb.steps['Springback'].frames)

field_output_frames_springback_step = []

for p in range(no_of_frames):
        for q in range(no_of_field_output_ut_values):
            field_label = restartOdb.steps['Springback'].frames[p] \
                                          .fieldOutputs['UT'].values[q].nodeLabel
            field_data_x = restartOdb.steps['Springback'] \
                                   .frames[p].fieldOutputs['UT'].values[q].data[0]
            field_data_y = restartOdb.steps['Springback'] \
                                   .frames[p].fieldOutputs['UT'].values[q].data[1]
            field_data_z = restartOdb.steps['Springback'] \
                                   .frames[p].fieldOutputs['UT'].values[q].data[2]

            fieldoutput_node_labels_springback_step.append(field_label)
            fieldoutput_node_disp_springback_step.append((field_data_x, field_data_y,
                                                                        field_data_z))

        field_output_frames_springback_step \
            .append(FieldOutputDisplacements(fieldoutput_node_labels_springback_step,
                                             fieldoutput_node_disp_springback_step))
        fieldoutput_node_labels_springback_step = []
        fieldoutput_node_disp_springback_step = []

    # Make a list of the step times for frames of 'Springback' step
    frame_step_times_springback_step = []
    for x in range(len(field_output_frames_springback_step)):
        frame_step_times_springback_step.append(restartOdb.steps['Springback'] \
                                                                .frames[x].frameValue)
```

The same task is repeated for the frames of the 'springback' step. The code has just been copied and suitable modified.

17.8.3 Create a new output database

This block writes the new output database containing the desired contents of the two existing output databases.

```
# ****************************************************************************
# WRITE TO NEW OUTPUT DATABASE (.ODB)
# ****************************************************************************

import os

if os.path.exists('Plate_plastic_bending_and_springback.odb'):
    os.remove('Plate_plastic_bending_and_springback.odb')

new_odb = Odb('Plate Bending and Springback combined', analysisTitle='',
              description='', path='Plate_plastic_bending_and_springback.odb')

plate_part = new_odb.Part(name='Plate Part', embeddedSpace=THREE_D,
                                             type=DEFORMABLE_BODY)
```

```
plate_part.addNodes(nodeData=node_labels_and_coords, nodeSetName='node set')
del node_labels_and_coords

plate_part.addElements(elementData=element_labels_and_coords, type='S4R',
                                                    elementSetName='element set')
del element_labels_and_coords


# Instance the part

plate_instance = new_odb.rootAssembly.Instance(name='Plate Instance',
                                                object=plate_part)

# Create a step 'Load Step'

load_step = new_odb.Step(name = 'Load Step', description='', domain=TIME,
                                                              timePeriod=1.0)

# Create frames with nodal displacements
for i in range(len(field_output_frames_load_step)):

    single_frame_load_step = load_step.Frame(incrementNumber=i,
                                  frameValue=frame_step_times_load_step[i],
                                  description='')

    disp_field_load_step = single_frame_load_step.FieldOutput(name='UT',
                                  description='Displacements and trans',
                                  type=VECTOR)

    disp_field_load_step.addData(position=NODAL, instance=plate_instance,
              labels=field_output_frames_load_step[i].field_data_node_labels,
              data=field_output_frames_load_step[i].field_data_xyz_displacements)

del field_output_frames_load_step
del frame_step_times_load_step

# Create a step 'Springback'
springback_step = new_odb.Step(name = 'Springback', description='', domain=TIME,
                                                                timePeriod=1.0)

# Create frames with nodal displacements
for i in range(len(field_output_frames_springback_step)):
    single_frame_springback_step = springback_step.Frame(incrementNumber=i,
                             frameValue=frame_step_times_springback_step[i],
                             description='')

    disp_field_springback_step = single_frame_springback_step \
                              .FieldOutput(name='UT',
                                      description='Displacements and trans',
                                      type=VECTOR)

    disp_field_springback_step.addData(position=NODAL, instance=plate_instance,
          labels=field_output_frames_springback_step[i].field_data_node_labels,
          data=field_output_frames_springback_step[i].field_data_xyz_displacements)

del field_output_frames_springback_step
del frame_step_times_springback_step

# Make this the default field for visualization
load_step.setDefaultDeformedField(disp_field_load_step)

# Save the odb in order to reopen it in abaqus/viewer

new_odb.save()
new_odb.close()

# ********************************************************************
```

```
import os

if os.path.exists('Plate_plastic_bending_and_springback.odb'):
    os.remove('Plate_plastic_bending_and_springback.odb')
```

os.path is a Python module that implements many useful functions on pathnames. One of the methods it provides is **exists()** which accepts a path as its argument and returns True if the argument refers to an existing path (and False if it does not).

os is a Python module that allows a program to use operating system dependent functionality. One of the functions it provides is **remove()** which accepts a path as an argument and deletes the file at that path. Note that on Microsoft Windows, if a file is in use this statement will raise an exception.

```
new_odb = Odb('Plate Bending and Springback combined', analysisTitle='',
              description='', path='Plate_plastic_bending_and_springback.odb')
```

An Odb object represents an output database file. Here we use the **Odb()** method to create an **Odb** object, or more importantly a new .odb file. The only required argument of the **Odb()** method is **name** which is a String specifying the repository key. **analysisTitle** is a String specifying the title of the output database, **description** is a String specifying its description, and **path** is a String specifying the path to the new .odb file. We create a new output database file with the delightfully creative name 'Plate_plastic_bending_and_springback.odb' and assign it to the variable **new_odb** so we can refer to it later in the script.

```
plate_part = new_odb.Part(name='Plate Part', embeddedSpace=THREE_D,
                                             type=DEFORMABLE_BODY)
```

The **Part()** method is used to create a part object. You've used this method in almost every script written so far and were first introduced to it in Section 4.3.3 on page 68. However that **Part()** was a method of the **Part** object. This one on the other hand is a method of **OdbPart**, and therefore creates an **OdbPart** object which is similar to the kernel **Part** object in that it contains nodes and elements, however it does not contain geometry. The **OdbPart** object created by it does not have nodes and elements at this stage, and can be added later. Using this method, we tell Abaqus to create a part in the new output database by using the dot notation and our output database variable **new_odb**. This odb part is assigned to the variable **plate_part**.

```
plate_part.addNodes(nodeData=node_labels_and_coords, nodeSetName='node set')
del node_labels_and_coords
```

addNodes() is a method of the **OdbPart** object. The **addNodes()** method adds nodes to this **OdbPart** object. Its required argument is **nodeData** which is a sequence of tuples specifying the node labels and coordinates in the form ((label1,x1, y1, z1), (label2, x2, y2, z2), ...) which is exactly how we have formatted our data into the **node_labels_and_coords** variable. It also has an optional argument **nodeSetName** which is a String specifying a name for this node set. Here we use **addNodes()** to add the node data we obtained from the first output database (which will be the same as the node data from the second output database since the exact same part was used in the restart analysis).

We then use Python's **del** keyword to delete the variable and free up some memory. This may not be too important for a small .odb file like the one we are using but for simulations with very large .odbs you might be keen to free up as much memory as possible when you can.

```
plate_part.addElements(elementData=element_labels_and_coords, type='S4R',
                                                    elementSetName='element set')
del element_labels_and_coords
```

This time we add element data to the new output database. **addElements()** is also a method of the **OdbPart** object, and adds elements to it. It has 2 required arguments – the first one is **elementData** which is a sequence of sequences of Ints specifying the element labels and nodal connectivity in the form *((label1, c1_1, c1_2, c1_3, c1_4), (label2, c2_1, c2_2, c2_3, c2_4), ...)* which is precisely how we formatted our element data into the *element_labels_and_coords* variable. The second required argument is **type** which is a String specifying the element type, which in our case we know to be 'S4R' hence we hard coded this into the program. However you could extract the element type using the script, by picking any random eleent, let's say element 50, using

```
prettyPrint(plateOdb.steps['Load Step'].frames[0].fieldOutputs['UT'].values[0]
                                                          .instance.elements[50])
```

We then delete **element_labels_and_coords** to free up some memory.

```
# Instance the part
plate_instance = new_odb.rootAssembly.Instance(name='Plate Instance',
                                              object=plate_part)
```

The **Instance()** method is used to create an instance of a part in an assembly. You've used **rootAssembly.Instance()** in almost all the scripts so far, and it was first encountered in Section 4.3.6 on page 74. However that **Instance()** method was from the **PartInstance** object. This **Instance()** method is a method of the **OdbInstance** object and is a little different. It copies a **PartInstance** object from a specified model and creates a new **PartInstance** object. It has 2 required arguments, **name**, which is a String specifying the repository key, and **objectToCopy**, which is the **PartInstance** object to be copied. We tell Abaqus to create an instance in the new output database by using the dot notation, our output database variable **new_odb**, and our previously created **PartInstance** object **plate_part**. This instance is assigned to the variable **plate_instance**.

```
load_step = new_odb.Step(name = 'Load Step', description='', domain=TIME,
                                                          timePeriod=1.0)
```

The **Step()** method used here is similar to **StaticStep()**, **ExplicitDynamicStep()** and **HeatTransferStep()** that you have used in previous scripts, except that it creates an **OdbStep** object instead of a Step object. It has 3 required arguments – **name**, **description** and **domain**. **name** is a String specifying the repository key, **description** is a String specifying the step description, and **domain** is a SymbolicConstant specifying the domain of the step with possible values of **TIME**, **FREQUENCY**, **ARC_LENGTH**, and **MODAL**. **timePeriod** is an optional argument, and it is a Float specifying the time period of the step. It is required if **domain=TIME**.

```
# Create frames with nodal displacements
for i in range(len(field_output_frames_load_step)):
```

```
single_frame_load_step = load_step.Frame(incrementNumber=i,
                                         frameValue=frame_step_times_load_step[i],
                                         description='')

disp_field_load_step = single_frame_load_step.FieldOutput(name='UT',
                                         description='Displacements and trans',
                                         type=VECTOR)

disp_field_load_step.addData(position=NODAL, instance=plate_instance,
          labels=field_output_frames_load_step[i].field_data_node_labels,
          data=field_output_frames_load_step[i].field_data_xyz_displacements)
```

In this **for** loop, we create a new frames for the **OdbStep** object using the **Frame()** method. **Frame()** creates an **OdbFrame** object. It has 2 required arguments, **incrementNumber**, which is an Int specifying the frame increment number within the step, and **frameValue**, which is a Float specifying the step time, frequency or mode depending on whether you are in the time, frequency or mode domain. Since we are in the time domain, **frameValue** is the step time. **description** is an optional argument and it is a String specifying the contents of the frame. Here we give the frames the same number as the loop counter since we are counting over the total number of frames the step should have, and for **frameValue** we use the step times we had collected earlier when reading in the odb. The **Frame()** method returns an **OdbFrame** object which we store in the variable **single_frame_load_step**.

A **FieldOutput** object contains field data for a specific output value. The **FieldOutput()** method is used to create a **FieldOutput** object. **FieldOutput()** has 3 required arguments – **name**, which is a String specifying the output variable name, **description**, which is a String specifying the output variable, and **type**, which is a SymbolicConstant specifying the output type with possible values of **SCALAR**, **VECTOR**, **TENSOR_3D_FULL**, **TENSOR_3D_PLANAR**, **TENSOR_3D_SURFACE**, **TENSOR_2D_PLANAR** and **TENSOR_2D_SURFACE**. Here we create a field output named 'UT' of type **VECTOR**.

The **addData()** method adds data to a **FieldOutput** object. It has 4 required arguments – **position, instance**, **labels** and **data**. **position** is a SymbolicConstant which specifies the position of the output. Since our displacement data is calculated at the nodes, we use **NODAL**. Other possible options are **INTEGRATION_POINT**, **ELEMENT_NODAL** and **CENTROID**. **instance** is an **OdbInstance** object which specifies the namespace for the labels. **labels** is a sequence of Ints which specifies the labels of the nodes (or elements) where the values in data are stored. The nodes (or element) labels must be sorted in ascending order, and must be in the same order as the values in **data. data** is a sequence of sequences of Floats specifying the data values.

```
del field_output_frames_load_step
del frame_step_times_load_step
```

We delete these variables to free up space since they are no longer required.

```
# Create a step 'Springback'
springback_step = new_odb.Step(name = 'Springback', description='', domain=TIME,
                                                                  timePeriod=1.0)

# Create frames with nodal displacements
for i in range(len(field_output_frames_springback_step)):
    single_frame_springback_step = springback_step.Frame(incrementNumber=i,
                                   frameValue=frame_step_times_springback_step[i],
                                   description='')

    disp_field_springback_step = single_frame_springback_step \
                                .FieldOutput(name='UT',
                                             description='Displacements and trans',
                                             type=VECTOR)

    disp_field_springback_step.addData(position=NODAL, instance=plate_instance,
         labels=field_output_frames_springback_step[i].field_data_node_labels,
         data=field_output_frames_springback_step[i].field_data_xyz_displacements)

del field_output_frames_springback_step
del frame_step_times_springback_step
```

We repeat the procedure for the ‘springback’ step.

```
# Make this the default field for visualization
load_step.setDefaultDeformedField(disp_field_load_step)
```

The **setDefaultDeformedField()** method is used to set the default deformed field variable in the step passed to it as an argument. When the odb is opened in Abaqus/Viewer and the deformed shape is plotted, it is the default deformed variable that will be plotted unless the analyst instructs Abaqus to plot another field output.

```
new_odb.save()
new_odb.close()
```

The output database is saved using the **save()** method, which allows it to later be opened again. It is then closed using the **close()** method.

17.8.4 Create the animation using the new output database

This block creates the animation using the frames of the new output database.

```
# ***********************************************************************
# CREATE AN ANIMATION
# ***********************************************************************

import visualization
import animation

plate_Odb_Path = 'Plate_plastic_bending_and_springback.odb'
odb_object = session.openOdb(name=plate_Odb_Path)

session.viewports['Viewport: 1'].setValues(displayedObject=odb_object)
session.viewports['Viewport: 1'].odbDisplay.display \
                                    .setValues(plotState=(CONTOURS_ON_DEF, ))
session.viewports['Viewport: 1'].odbDisplay \
                            .setPrimaryVariable(variableLabel='UT',
                                                outputPosition=NODAL,
                                                refinement=(COMPONENT, 'UT3'),)

session.animationController.setValues(animationType=TIME_HISTORY,
                                      viewports=('Viewport: 1', ))
session.animationController.play(duration=UNLIMITED)
session.imageAnimationOptions.setValues(vpDecorations=ON, vpBackground=OFF,
                                        compass=ON, timeScale=1)
session.writeImageAnimation(fileName='plate_bend_springback_animation',
                            format=AVI,
                            canvasObjects=(session.viewports['Viewport: 1'], ))
```

```
import visualization
import animation

plate_Odb_Path = 'Plate_plastic_bending_and_springback.odb'
odb_object = session.openOdb(name=plate_Odb_Path)

session.viewports['Viewport: 1'].setValues(displayedObject=odb_object)
session.viewports['Viewport: 1'].odbDisplay.display \
                                    .setValues(plotState=(CONTOURS_ON_DEF, ))
session.viewports['Viewport: 1'].odbDisplay \
                            .setPrimaryVariable(variableLabel='UT',
                                                outputPosition=NODAL,
                                                refinement=(COMPONENT, 'UT3'),)
```

All of these statements should be familiar to you by now except the second import statement which is required to use the **AnimationController** object.

```
session.animationController.setValues(animationType=TIME_HISTORY,
                                      viewports=('Viewport: 1', ))
```

The **AnimationController** object controls the object-based animations in the viewport. Its **setValues()** method has two optional arguments – **animationType** and **viewports**. **animationType** is a SymbolicConstant specifying the type of movie that should be played, with possible values of **SCALE_FACTOR, HARMONIC, TIME_HISTORY** and the default of **NONE**. **viewports** is a sequence of pairs of Strings specifying the repository key of the viewport where the animation will be active followed by a layer name or the SymbolicConstant **ALL**. In our case no layer name is specified, hence the current layer will be used.

```
session.animationController.play(duration=UNLIMITED)
```

The **play()** method of the **AnimationController** object plays the animation. It has one optional argument **duration** which is an Int specifying how many seconds to play the animation, or it can be the SymbolicConstant **UNLIMITED** (which is the default).

```
session.imageAnimationOptions.setValues(vpDecorations=ON, vpBackground=OFF,
                                         compass=ON, timeScale=1)
```

The **ImageAnimationOptions** object stores the values and attributes associated with saving viewport animations. Its **setValues()** method has a few optional arguments, many of which are used here. **vpDecorations** is a Boolean specifying whether or not to include the viewport border and title. It defaults to ON. Here we choose ON (or leave it at default) so that the border and title will be included in the video. **vpBackground** is a Boolean specifying whether or not to display the viewport background. It defaults to OFF. We will set it to off (or leave it at default) so that the animation will be displayed on a blank white background. **compass** is a Boolean specifying whether to include the compass in the video. It defaults to OFF, but here we change it to ON. **timescale** is an Int specifying the time scale to apply to the frame rate. We have not specified **frameRate** which is the final optional argument, hence Abaqus will choose a frame rate, and by using a **timescale** of 1 we leave this framerate unchanged.

```
session.writeImageAnimation(fileName='plate_bend_springback_animation',
                            format=AVI,
                            canvasObjects=(session.viewports['Viewport: 1'], ))
```

writeImageAnimation() is a method of the session object. It creates an animation file. It has 2 required arguments – **fileName** which is a String specifying the name of the animation file to generate (and this can be a full path), and **format** which is a SymbolicConstant specifying the format of the generated file with possible values of

AVI, **QUICKTIME**, **VRML** and **COMPRESSED_VRML**. It also has one optional argument – **canvasObjects** – a sequence specifying the canvas objects to capture. We tell it to capture 'Viewport: 1' which is where the animation is displayed.

17.9 Summary

In this chapter we extracted data from 2 existing output databases and created a new one using this information. You now have a firm understanding of not only how to extract information from output databases using a Python script, but also how to construct one from scratch. Using this technique you can create output databases that contain only what you need - either for further processing tasks or to help you or another analyst visualize specific results.

Monitor an Analysis Job and Send an Email when Complete

18.1 Introduction

A single analysis job in Abaqus can take hours or even days to run. Multiple jobs running as part of an optimization routing can take a considerable amount of time to execute. It is possible to write a script that monitors a job and provide updates to the analyst.

In this example we shall monitor the running of the Cantilever Beam example from Chapter 4. We shall detect when the job completes or aborts. We will then log into a Gmail account, and send an email to another address informing the analyst that the job has either completed running or quit with errors.

18.2 Methodology

In our original Cantilever Beam script we submit the job and then wait for it to complete using the **WaitForCompletion()** function. On completion, program control returns to the script and subsequent statements, in our case post processing statements, are executed.

We will no longer use the **waitForCompletion()** function. Instead we will use the **addMessageCallback()** function of the **MonitorMgr** object provided by Abaqus to monitor messages generated by Abaqus during the analysis. Every time a message is generated a function **jobMonitorCallback()**,defined by us, will be called, which will check the type of the message. If the message type is either **ABORTED** or **COMPLETED** it will call another function **postProcess()**, also defined by us, to log into Gmail's SMTP server and send an email indicating that the job has been completed (or aborted).

18.3 Python Script

The following listing is the completed Python script to accomplish this. You can find it in the source code accompanying the book in **job_monitor_and_email.py**. You can run it by opening a new model in Abaqus/CAE (**File > New Model database > With Standard/Explicit Model**) and running it with **File > Run Script...**

```
# ---------------------------------------------------------------------------
# Define the callback function jobMonitorCallback()
def jobMonitorCallback(jobName, messageType, data, userData):
    if ((messageType==ABORTED) or (messageType==ERROR)):
        # Send an email
        sendEmailMessage("Bad news - The job has failed")
        # Stop monitoring the job
        monitorManager.removeMessageCallback(jobName='BeamDeflectionJob',
                                             messageType=ANY_MESSAGE_TYPE,
                                             callback=jobMonitorCallback,
                                             userData=None)
    elif (messageType==JOB_COMPLETED):
        # Send an email
        sendEmailMessage("Good news - The job has finished running !!!")
        # Stop monitoring the job
        monitorManager.removeMessageCallback(jobName='BeamDeflectionJob',
                                             messageType=ANY_MESSAGE_TYPE,
                                             callback=jobMonitorCallback,
                                             userData=None)
        # Call post processing function
        postProcess()
# ---------------------------------------------------------------------------


# ---------------------------------------------------------------------------
# Define a function sendEmailMessage() to send the email
def sendEmailMessage(email_message):

    # Import Pythons smtplib in order to send emails.
    # The smtplib.SMTP class encapsulates an SMTP connection and has methods for
    # SMTP and ESMTP operations
    import smtplib

    # Import the email module required to send text in MIME format
    from email.mime.text import MIMEText

    sender = 'abaquspython@gmail.com'
    recipient = 'garyofcourse@yahoo.com'
    subject = 'Email sent from Abaqus with Python script'
    contents = email_message

    # Create a text/plain message
    msg = MIMEText(contents)
    # Specify the email subject, sender and receipient
    msg['Subject'] =  subject
    msg['From'] = sender
    msg['To'] = recipient

    # This is Googles Outgoing Mail (SMTP) server
    gmail_smtp_server = 'smtp.gmail.com'

    # This is the port used by Gmail server for outgoing mail
    gmail_smtp_port = 587

    # Gmail username (SMTP username)
    gmail_username = 'abaquspython'

    # Gmail password (SMTP password)
    gmail_password = 'xxxxx'

    # Create an SMTP object. The SMTP connect() method is called using the
    # name and port.
    session = smtplib.SMTP(gmail_smtp_server, gmail_smtp_port)

    # Identify ourselves to the ESMTP server using EHLO
    session.ehlo()

    # Put the SMTP connection in Transport Layer Security (TLS) mode using
    # SMTP.starttls() so all SMTP commands that follow will be encrypted
    session.starttls()

    # Call EHLO again
    session.ehlo()

    # Login to the server using SMTP.login()
    session.login(gmail_username, gmail_password)

    # Send the email using SMTP.sendmail()
    session.sendmail(sender, [recipient], msg.as_string())

    # End the session
    session.close()

# sendEmailMessage() definition ends here
# -----------------------------------------------------------------------

# -----------------------------------------------------------------------
# Define a function postProcess() to display the deformed shape
def postProcess():
    # Post Processing

    import visualization

    beam_viewport = session.Viewport(name='Beam Results Viewport')
    beam_Odb_Path = 'BeamDeflectionJob.odb'
    an_odb_object = session.openOdb(name=beam_Odb_Path)
    beam_viewport.setValues(displayedObject=an_odb_object)
    beam_viewport.odbDisplay.display.setValues(plotState=(DEFORMED, ))
# -----------------------------------------------------------------------


from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)


# -----------------------------------------------------------------------
# Create the model (or more accurately, rename the existing one)

mdb.models.changeKey(fromName='Model-1', toName='Cantilever Beam')
beamModel = mdb.models['Cantilever Beam']

# -----------------------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the beam cross section using rectangle tool
beamProfileSketch = beamModel.ConstrainedSketch(name='Beam CS Profile',
                                                sheetSize=5)
beamProfileSketch.rectangle(point1=(0.1,0.1), point2=(0.3,-0.1))

# b) Create a 3D deformable part named "Beam" by extruding the sketch
beamPart=beamModel.Part(name='Beam', dimensionality=THREE_D,
                                     type=DEFORMABLE_BODY)
beamPart.BaseSolidExtrude(sketch=beamProfileSketch, depth=5)

# -----------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs modulus and
# poissons ratio
beamMaterial = beamModel.Material(name='AISI 1005 Steel')
beamMaterial.Density(table=((7872, ),       ))
beamMaterial.Elastic(table=((200E9, 0.29), ))

# -----------------------------------------------------------------------
# Create solid section and assign the beam to it

import section
```

```
# Create a section to assign to the beam
beamSection = beamModel.HomogeneousSolidSection(name='Beam Section',
                                                material='AISI 1005 Steel')

# Assign the beam to this section
beam_region = (beamPart.cells,)
beamPart.SectionAssignment(region=beam_region, sectionName='Beam Section')

# -----------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
beamAssembly = beamModel.rootAssembly
beamInstance = beamAssembly.Instance(name='Beam Instance', part=beamPart,
                                                          dependent=ON)

# -----------------------------------------------------------------------
# Create the step

import step

# Create a static general step
beamModel.StaticStep(name='Apply Load', previous='Initial',
                     description='Load is applied during this step')


# -----------------------------------------------------------------------
# Create the field output request

# change the name of field output request 'F-Output-1' to 'Selected Field Outputs'
beamModel.fieldOutputRequests.changeKey(fromName='F-Output-1',
                                        toName='Selected Field Outputs')

# since F-Output-1 is applied at the 'Apply Load' step by default, 'Selected
# Field Outputs' will be too
# we only need to set the required variables
beamModel.fieldOutputRequests['Selected Field Outputs'] \
                      .setValues(variables=('S', 'E', 'PEMAG', 'U', 'RF', 'CF'))


# -----------------------------------------------------------------------
# Create the history output request

# we try a slightly different method from that used in field output request
# create a new history output request called 'Default History Outputs' and assign
# both the step and the variables
beamModel.HistoryOutputRequest(name='Default History Outputs',
                               createStepName='Apply Load', variables=PRESELECT)

# now delete the original history output request 'H-Output-1'
```

```
del beamModel.historyOutputRequests['H-Output-1']

# ----------------------------------------------------------------------
# Apply pressure load to top surface

# First we need to locate and select the top surface
# We place a point somewhere on the top surface based on our knowledge of the
# geometry
top_face_pt_x = 0.2
top_face_pt_y = 0.1
top_face_pt_z = 2.5
top_face_pt = (top_face_pt_x,top_face_pt_y,top_face_pt_z)

# the face on which that point lies is the face we are looking for
top_face = beamInstance.faces.findAt((top_face_pt,))

# we extract the region of the face choosing which direction its normal points in
top_face_region=regionToolset.Region(side1Faces=top_face)

# apply the pressure load on this region in the 'Apply Load' step
beamModel.Pressure(name='Uniform Applied Pressure', createStepName='Apply Load',
                   region=top_face_region, distributionType=UNIFORM,
                   magnitude=10, amplitude=UNSET)

# ----------------------------------------------------------------------
# Apply encastre (fixed) constraint to one end to make it cantilever

# First we need to locate and select the top surface
# We place a point somewhere on the top surface based on our knowledge of the
# geometry
fixed_end_face_pt_x = 0.2
fixed_end_face_pt_y = 0.0
fixed_end_face_pt_z = 0.0
fixed_end_face_pt = (fixed_end_face_pt_x,fixed_end_face_pt_y,fixed_end_face_pt_z)

# the face on which that point lies is the face we are looking for
fixed_end_face = beamInstance.faces.findAt((fixed_end_face_pt,))

# we extract the region of the face choosing which direction its normal points in
fixed_end_face_region=regionToolset.Region(faces=fixed_end_face)

beamModel.EncastreBC(name='Encaster one end', createStepName='Initial',
                                              region=fixed_end_face_region)

# ----------------------------------------------------------------------
# Create the mesh

import mesh

# First we need to locate and select a point inside the solid
# We place a point somewhere inside it based on our knowledge of the geometry
beam_inside_xcoord=0.2
```

```
beam_inside_ycoord=0.0
beam_inside_zcoord=2.5

elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN, secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)


beamCells=beamPart.cells
selectedBeamCells=beamCells.findAt((beam_inside_xcoord,beam_inside_ycoord,
                                                              beam_inside_zcoord),)
beamMeshRegion=(selectedBeamCells,)
beamPart.setElementType(regions=beamMeshRegion, elemTypes=(elemType1,))

beamPart.seedPart(size=0.1, deviationFactor=0.1)

beamPart.generateMesh()

# -----------------------------------------------------------------------
# Create and run the job

import job
from jobMessage import *

# Create the job
mdb.Job(name='BeamDeflectionJob', model='Cantilever Beam', type=ANALYSIS,
        explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE,
        description='Job simulates the loaded cantilever beam',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)

# Monitor the job
monitorManager.addMessageCallback(jobName='BeamDeflectionJob',
                                  messageType=ANY_MESSAGE_TYPE,
                                  callback=jobMonitorCallback, userData=None)

# Run the job
mdb.jobs['BeamDeflectionJob'].submit(consistencyChecking=OFF)

# End of run job
# -----------------------------------------------------------------------
```

18.4 Examining the Script

We will examine this script in a different order, rather than the top-down approach we usually take. This is because I have defined the functions at the top of the script rather than at the locations where they are actually called. It is necessary for a function to be defined before the interpreter encounters it in the script.

Since this script is a modified version of the original Cantilever Beam script with a few new functions added, only the new parts of the script will be examined, and you can refer back to Chapter 4 for the rest.

18.4.1 Job submission and message callback

The block that submits the job and calls the job monitor is displayed below:

```
# -----------------------------------------------------------------------
# Create and run the job

import job
from jobMessage import *

# Create the job
mdb.Job(name='BeamDeflectionJob', model='Cantilever Beam', type=ANALYSIS,
        explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE,
        description='Job simulates the loaded cantilever beam',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)

# Monitor the job
monitorManager.addMessageCallback(jobName='BeamDeflectionJob',
                                  messageType=ANY_MESSAGE_TYPE,
                                  callback=jobMonitorCallback, userData=None)

# Run the job
mdb.jobs['BeamDeflectionJob'].submit(consistencyChecking=OFF)

# End of run job
# -----------------------------------------------------------------------
```

Most of these statements are almost identical to those of the Cantilever Beam example with a few modifications.

```
from jobMessage import *
```

This gives us access to the SymbolicConstants necessary for the **addMessageCallback()** method. When we call this function, we use **ANY_MESSAGE_TYPE** as one of the arguments. The **jobMessage** module defines this SymbolicConstant. We could in fact have written the statement as

```
from jobMessage import ANY_MESSAGE_TYPE
```

But I prefer not be so precise as it only complicates things.

```
monitorManager.addMessageCallback(jobName='BeamDeflectionJob',
                                  messageType=ANY_MESSAGE_TYPE,
                                  callback=jobMonitorCallback, userData=None)
```

The **monitorManager** object is used for monitoring a job. It provides a method **addMessageCallback()** that tells Abaqus which function to call when a particular message is received from the solver. It has 3 required arguments.

The first is **jobName** which needs to be a String corresponding to the name of the job to monitor. If you wish to monitor all jobs you can use the SymbolicConstant **ANY_JOB** which is available to us from the **jobMessage** module.

The second argument is **messageType** which is a SymbolicConstant specifying which type of messages will cause the callback function to be called. The options available here are **ABORTED**, **ANY_JOB**, **ANY_MESSAGE_TYPE**, **COMPLETED**, **END_STEP**, **ERROR**, **HEADING**, **HEALER_JOB**, **HEALER_TYPE**, **INTERRUPTED**, **ITERATION**, **JOB_ABORTED**, **JOB_COMPLETED**, **JOB_INTERRUPTED**, **JOB_SUBMITTED**, **MONITOR_DATA**, **ODB_FILE**, **ODB_FRAME**, **SIMULATION_ABORTED**, **SIMULATION_COMPLETED**, **SIMULATION_INTERRUPTED**, **SIMULATION_SUBMITTED**, **STARTED**, **STATUS**, **STEP** and **WARNING**. We use **ANY_MESSAGE_TYPE** which causes all messages to call the callback function. We will then process these messages within the callback function itself as this allows us to deal with **JOB_COMPLETED**, **ABORTED** and **ERROR** using just one **addMessageCallback()** method.

The third required argument is **callback** which is the name of the function to call when the specified message type is encountered. By default, this function will be called with **jobName**, **messageType**, **data** and **userData** as its parameters. **data** is useful if you are using **MONITOR_DATA** to monitor a degree of freedom during the analysis, and will contain information about that degree of freedom. **userData**, which is also an optional argument for **addMessageCallback()**, is any Python object (or **None**) that you choose to pass to the function. You could choose, for instance, to pass a reference to the Viewport so that the callback function can access it.

Notice that the **waitForCompletion()** method has been removed from the script. This is because we no longer have any statements after the **submit()** function and are placing all the post processing tasks within another function that the callback function will execute one the job has been completed. We have therefore replaced the **waitForCompletion()** method with **addMessageCallback()**.

18.4.2 Define the callback function

This block defines our callback function.

```
# -------------------------------------------------------------------------
# Define the callback function jobMonitorCallback()
def jobMonitorCallback(jobName, messageType, data, userData):
    if ((messageType==ABORTED) or (messageType==ERROR)):
        # Send an email
        sendEmailMessage("Bad news - The job has failed")
        # Stop monitoring the job
        monitorManager.removeMessageCallback(jobName='BeamDeflectionJob',
                                             messageType=ANY_MESSAGE_TYPE,
                                             callback=jobMonitorCallback,
                                             userData=None)
    elif (messageType==JOB_COMPLETED):
        # Send an email
        sendEmailMessage("Good news - The job has finished running !!!")
        # Stop monitoring the job
        monitorManager.removeMessageCallback(jobName='BeamDeflectionJob',
                                             messageType=ANY_MESSAGE_TYPE,
                                             callback=jobMonitorCallback,
                                             userData=None)
        # Call post processing function
        postProcess()
# -------------------------------------------------------------------------
```

As stated in the previous section, when **addMessageCallback()** calls the callback function – which we have named **jobMonitorCallback()** – it passes to it **jobName, messageType, data** and **userData** as arguments. Hence our function definition includes these as the accepted arguments.

An **if-elif** block is used to check the message type and act accordingly. If **messageType** is **ABORTED** or **ERROR** indicating that the job was aborted or exited with an error, the function **sendEmailMessage()** is called with the String "Bad news – The job has failed" as its argument. On the other hand if **messageType** is **JOB_COMPLETED** indicating that the job completed successfully, **sendEmailMessage()** is called with the String "Good news – The job has finished running !!!" as its argument. Also another function we define – **postProcess()** – is called to execute any post processing tasks such as plotting the deformed shape.

In both cases the **removeMessageCallback()** method is used to put an end to the job monitoring. **removeMessageCallback()** is the exact opposite of **addMessageCallback()** and the callback function is no longer called when messages are received from the solver. The arguments supplied to it must exactly match the arguments supplied to **addMessageCallback()**.

18.4.3 Define a function to handle post processing

This block defines the function **postProcess()** which will handle post processing tasks.

```
# -----------------------------------------------------------------------
# Define a function postProcess() to display the deformed shape
def postProcess():
        # Post Processing

        import visualization

        beam_viewport = session.Viewport(name='Beam Results Viewport')
        beam_Odb_Path = 'BeamDeflectionJob.odb'
        an_odb_object = session.openOdb(name=beam_Odb_Path)
        beam_viewport.setValues(displayedObject=an_odb_object)
        beam_viewport.odbDisplay.display.setValues(plotState=(DEFORMED, ))
# -----------------------------------------------------------------------
```

The **postProcess()** function defined by us contains nothing new - just the post processing statements from the original Cantilever Beam example. All we have done is packaged these statements inside this function so that it can be called from our callback function **addMessageCallback()** after the job has completed running successfully.

18.4.4 Define the email function

This block defines a function that connects to Gmail and sends an email.

```
# -----------------------------------------------------------------------
# Define a function sendEmailMessage() to send the email
def sendEmailMessage(email_message):

    # Import Pythons smtplib in order to send emails.
    # The smtplib.SMTP class encapsulates an SMTP connection and has methods for
    # SMTP and ESMTP operations
    import smtplib

    # Import the email module required to send text in MIME format
    from email.mime.text import MIMEText

    sender = 'abaquspython@gmail.com'
    recipient = 'garyofcourse@yahoo.com'
    subject = 'Email sent from Abaqus with Python script'
    contents = email_message

    # Create a text/plain message
    msg = MIMEText(contents)

    # Specify the email subject, sender and receipient
    msg['Subject'] =  subject
    msg['From'] = sender
    msg['To'] = recipient

    # This is Googles Outgoing Mail (SMTP) server
    gmail_smtp_server = 'smtp.gmail.com'

    # This is the port used by Gmail server for outgoing mail
    gmail_smtp_port = 587

    # Gmail username (SMTP username)
    gmail_username = 'abaquspython'

    # Gmail password (SMTP password)
    gmail_password = 'xxxxx'

    # Create an SMTP object. The SMTP connect() method is called using the
    # name and port.
    session = smtplib.SMTP(gmail_smtp_server, gmail_smtp_port)

    # Identify ourselves to the ESMTP server using EHLO
    session.ehlo()

    # Put the SMTP connection in Transport Layer Security (TLS) mode using
    # SMTP.starttls() so all SMTP commands that follow will be encrypted
    session.starttls()

    # Call EHLO again
    session.ehlo()

    # Login to the server using SMTP.login()
    session.login(gmail_username, gmail_password)

    # Send the email using SMTP.sendmail()
    session.sendmail(sender, [recipient], msg.as_string())

    # End the session
    session.close()

# sendEmailMessage() definition ends here
# -------------------------------------------------------------------
```

The **sendEmailMessage()** function contains the Python code required to send an email. Since this is not a book on Python, but rather a book on writing Python scripts for Abaqus, I do not want to get into advanced Python programming. However I think this is a very cool (and also very useful) example of how you can use Python scripts to make your life easier. I will limit this section to a very basic explanation of what the above code does, and if you ever decide to implement this in one of your scripts just copy and paste it in.

```
def sendEmailMessage(email_message):
```

sendEmailmessage() accepts a String as an argument which will be used as the content of the email.

```
    import smtplib
```

SMTP is Simple Mail Transfer Protocol. Python provides **smtplib** in order to send mail from a Python program. The **SMTP** class in **smtplib** encapsulates an SMTP connection and has functions that can be used for SMTP and ESMTP (extended SMTP) operations.

```
    # Import the email module required to send text in MIME format
    from email.mime.text import MIMEText
```

This statement imports the email module required to send text in Multipurpose Internet Mail Extensions (MIME) format. MIME defines the format of the email, and most internet email today is transmitted via SMTP in MIME format.

```
    sender = 'abaquspython@gmail.com'
    recipient = 'garyofcourse@yahoo.com'
    subject = 'Email sent from Abaqus with Python script'
    contents = email_message
```

Here we place the email addresses of the sender and recipient, and the subject of the email in variables for later use. The content of the email is obtained from **email_message** which was passed as an argument to **sendEmailMessage()**. The sender email address can be a Gmail address that you create specifically for sending emails about Abaqus jobs, since the email have to be sent from somewhere. The recipient email address will be the email address of the analyst conducting the study.

```
    # Create a text/plain message
    msg = MIMEText(contents)
```

The **MIMEText** class provided by Python is used to create **MIME** objects. Here we are giving it the String contained in the variable **contents**. When we pass **msg** to the **sendmail()** function, this String will form the body of the email.

```
# Specify the email subject, sender and receipient
msg['Subject'] =  subject
msg['From'] = sender
msg['To'] = recipient
```

Here we add the subject, sender and recipient to the **mimetext** object. When the email is received by the recipient, the sender, recipient and subject fields will be filled in due to these statements.

```
# This is Googles Outgoing Mail (SMTP) server
gmail_smtp_server = 'smtp.gmail.com'

# This is the port used by Gmail server for outgoing mail
gmail_smtp_port = 587

# Gmail username (SMTP username)
gmail_username = 'abaquspython'

# Gmail password (SMTP password)
gmail_password = 'xxxxx'
```

The variable names used here are quite descriptive. 'smtp.gmail.com' is Googles SMTP server, and it uses port 587. The username and password used here are for an existing Gmail account. You'll have to create one of your own and stick the username and password in here.

```
# Create an SMTP object. The SMTP connect() method is called using the
# name and port.
session = smtplib.SMTP(gmail_smtp_server, gmail_smtp_port)
```

The **smtplib.SMTP()** method is used to create an **SMTP** object. It accepts 2 arguments, the server name and the server port. It calls the **smtplib.connect()** method and connects to the server.

```
# Identify ourselves to the ESMTP server using EHLO
session.ehlo()
```

The **ehlo()** method utilizes the **EHLO** clause (a more advanced version to **HELO**) for the client to basically say hello to the server and identify itself and the server replies back.

```
# Put the SMTP connection in Transport Layer Security (TLS) mode using
# SMTP.starttls() so all SMTP commands that follow will be encrypted
session.starttls()
```

starttls() puts the SMTP connection in Transport Layer Security (TLS) mode. This means all further communication will be encrypted making it much safer.

```
# Call EHLO again
session.ehlo()
```

Now that we are in TLS mode we call **ehlo()** again.

```
# Login to the server using SMTP.login()
session.login(gmail_username, gmail_password)
```

The **login()** method logs into the server using the supplied username and password. It is safe to use these here since we have an encrypted connection to the server.

```
# Send the email using SMTP.sendmail()
session.sendmail(sender, [recipient], msg.as_string())
```

The **sendmail()** function sends the email. It accepts 3 arguments, a String for the sender, a list of Strings for recipients, and a message. We use the **as_String()** method to flatten the email into a String. When the **as_String()** method is used with a MIME object it encodes the text in a format suitable for email. The 'subject', 'from' and 'to' fields of the email will be represented appropriately.

```
# End the session
session.close()
```

This ends our session with the server.

18.5 Summary

In this chapter you were introduced to job monitoring. In the example script we monitored the messages **ABORTED**, **ERROR** and **JOB_COMPLETED**, which are only a few of the available message types. If job monitoring is an important topic in your work I strongly recommend looking up the other message types and experimenting with them. We also learnt how to send an email from a Python script. While this involved some advanced Python programming, it not only gave you some reusable code in case you wish to have your jobs email you on completion, but it also demonstrated the fact that you can harness powerful features of the Python language and are not only limited to Abaqus kernel commands.

PART 3 – GUI SCRIPTS

Up until this point all the scripts you have written have run without much interaction with the analyst, with the exception of the prompt boxes of Chapter 14. This is perfectly acceptable for most scripts, and possibly all scripts you ever write for Abaqus will be like this. However there may be times when you wish to create an interface for your script, just so you can type in values or select options at runtime. If you work in an environment where other analysts will be using your scripts, a visual interface can save them having to modify your scripts directly, and may therefore be beneficial for everyone involved. Taking things a step further, if you are in a large organization where individuals without much Abaqus experience will be working with your models, you may wish to alter the Abaqus/CAE interface itself so as to provide them with a pre-determined workflow and limit their exposure to the complexities of Abaqus.

In Part 3, you will learn how to create simple dialog boxes using the Really Simple GUI (RSG), as well as custom interfaces and vertical applications using the Abaqus GUI Toolkit. From my personal experience, most individuals working with Python scripts in Abaqus are not required to create GUIs, therefore most of the following chapters can be considered optional for most readers. However it wouldn't hurt to skim over them, just so you get an idea of what is involved.

The last chapter of the book deals with Plug-ins. These are useful for both kernel and GUI scripts, so browse through it even if you skip chapters 19 – 21.

19

A Really Simple GUI (RSG) for the Sandwich Structure Study

19.1 Introduction

In Chapter 15 we wrote a parameterized script to study the deflection of a pressure loaded sandwich structure. This script accepted parameters using a specially formatted input file and ran a complete analysis for each set of inputs. In this chapter we shall modify that script to instead accept inputs/parameters using a dialog box presented to the analyst in Abaqus/CAE. To simplify the example and focus on topic at hand, the analysis will only accept one set of inputs and run once using these. The dialog box will only be presented once at the beginning and there will be no looping.

The dialog box will be created using a facility known as the Really Simple GUI, abbreviated as RSG. RSG allows the analyst to quickly create a dialog box with text fields, checkboxes, combo boxes (dropdown menus), radio buttons and so on without using any complex GUI customization tools. The drawback is that you can only customize the appearance of the dialog box you create, not the rest of the Abaqus/CAE interface. In addition, the appearance of the dialog box itself cannot change dynamically, meaning that you cannot show and hide controls, or display different options based on previously selected ones.

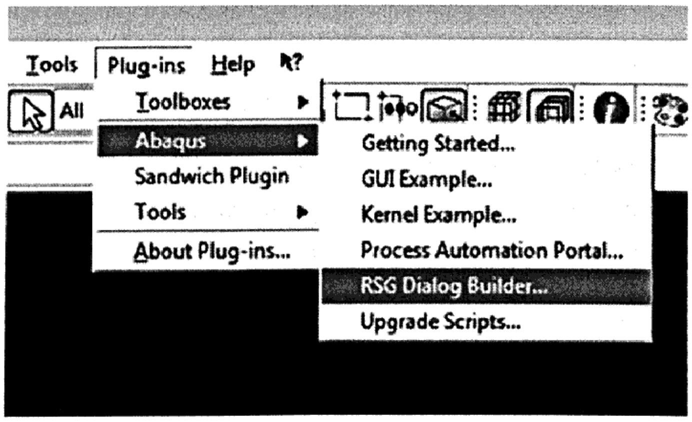
19.2 Methodology

We will modify the script from the sandwich structure analysis. It will be placed inside a function using the **def** keyword. This function will be called by the RSG dialog box when the user clicks OK, and the parameters provided to the script will be the values supplied by the user using the dialog box controls. Needless to say we will delete the parts of the script that read data from an input file. In addition the loop itself will be removed since the analysis will only be run once.

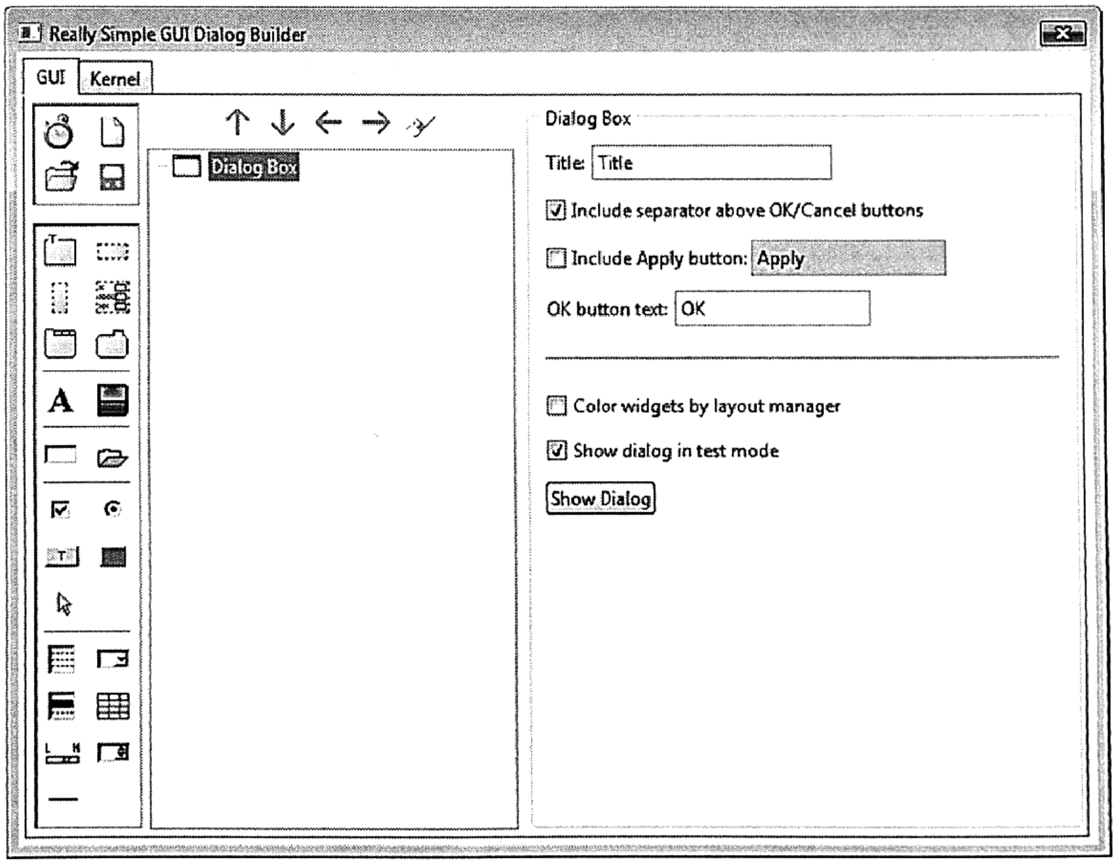
The RSG Dialog builder will be used to create the dialog box. It is a WYSIWYG (what you see is what you get) interface where you select which controls you would like to place on the dialog box from the available options, and the finished product will look identical to it.

19.3 Getting Started with RSG

In Abaqus v6.10 the RSG Dialog builder can be accessed from **Plugins > Abaqus > RSG Dialog Builder...** as displayed in the figure.



The Really Simple GUI Dialog Builder appears as shown in the following figure. On the left hand side you see a set of tools you can use. Most of these are controls/widgets that can be added to the dialog box. As you click on them they will populate the tree in the center giving you a hierarchy which can be rearranged using the arrow keys.



Dialog Box

Title: Title

[✓] Include separator above OK/Cancel buttons

[ ] Include Apply button: Apply

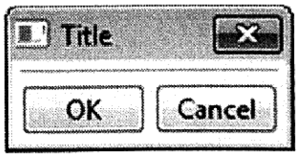
OK button text: OK

[ ] Color widgets by layout manager

[✓] Show dialog in test mode

Show Dialog

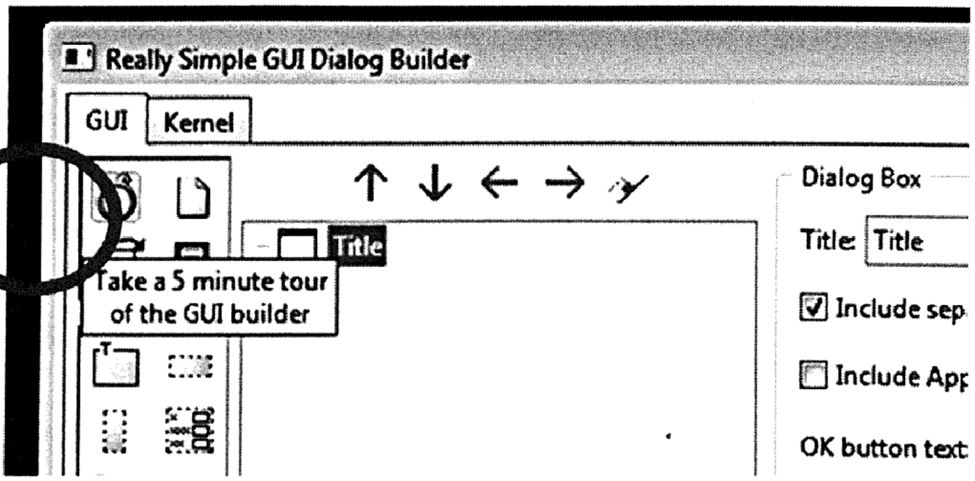
In the right side of the window, where you see a few dialog box options, check 'Show dialog in test mode' and click the 'Show Dialog' button.



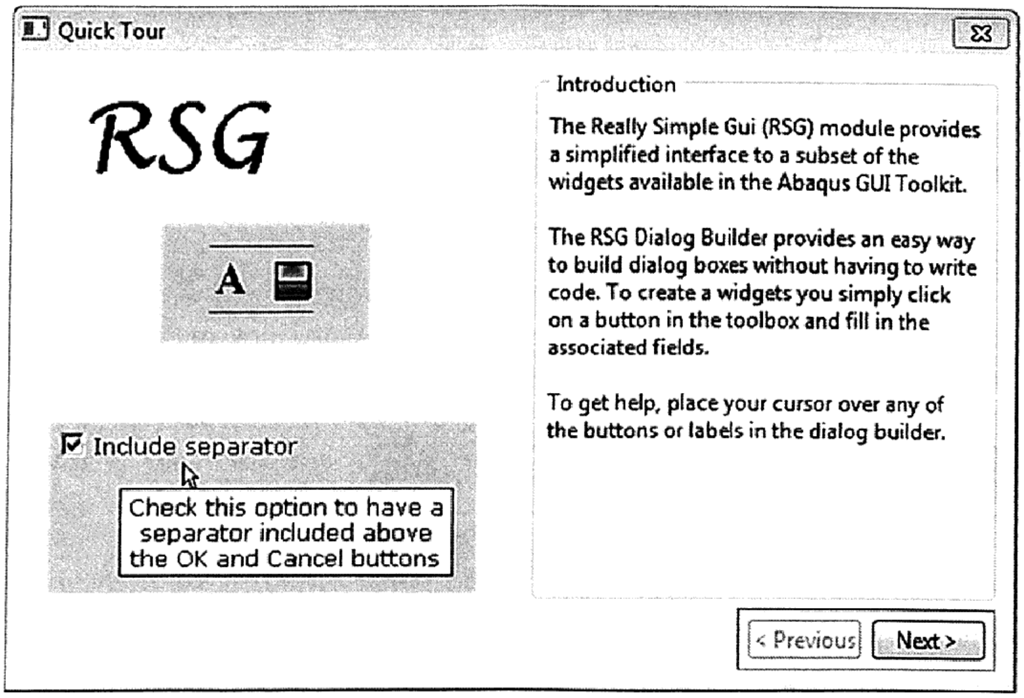
A dialog box is displayed. At the moment you haven't added any controls to it hence all it contains is **OK** and **Cancel** buttons.

The RSG comes with a basic 5 minute (or shorter) tutorial. It makes little sense for me to rehash what is already covered in this tutorial especially since it is available to everyone. You can either run through it in Abaqus, or follow along using the screenshots below. These screenshots were taken in Abaqus/CAE Student Edition 6.10-2.

Click on the "Take a 5 minute tour of the GUI builder" tool.



The ‘Quick Tour’ begins.

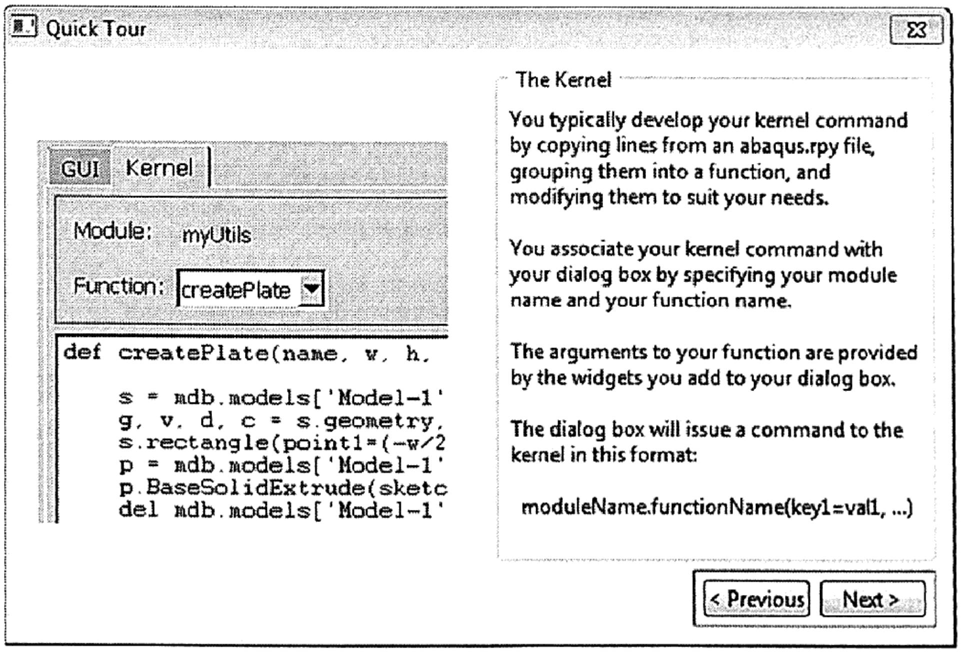

Quick Tour
RSG
A
Include separator
Check this option to have a separator included above the OK and Cancel buttons
< Previous
Next >

Introduction

The Really Simple Gui (RSG) module provides a simplified interface to a subset of the widgets available in the Abaqus GUI Toolkit.

The RSG Dialog Builder provides an easy way to build dialog boxes without having to write code. To create a widgets you simply click on a button in the toolbox and fill in the associated fields.

To get help, place your cursor over any of the buttons or labels in the dialog builder.



```
def createPlate(name, w, h,

    s = mdb.models['Model-1'
    g, v, d, c = s.geometry,
    s.rectangle(point1=(-w/2
    p = mdb.models['Model-1'
    p.BaseSolidExtrude(sketc
    del mdb.models['Model-1'
```

The Kernel

You typically develop your kernel command by copying lines from an abaqus.rpy file, grouping them into a function, and modifying them to suit your needs.

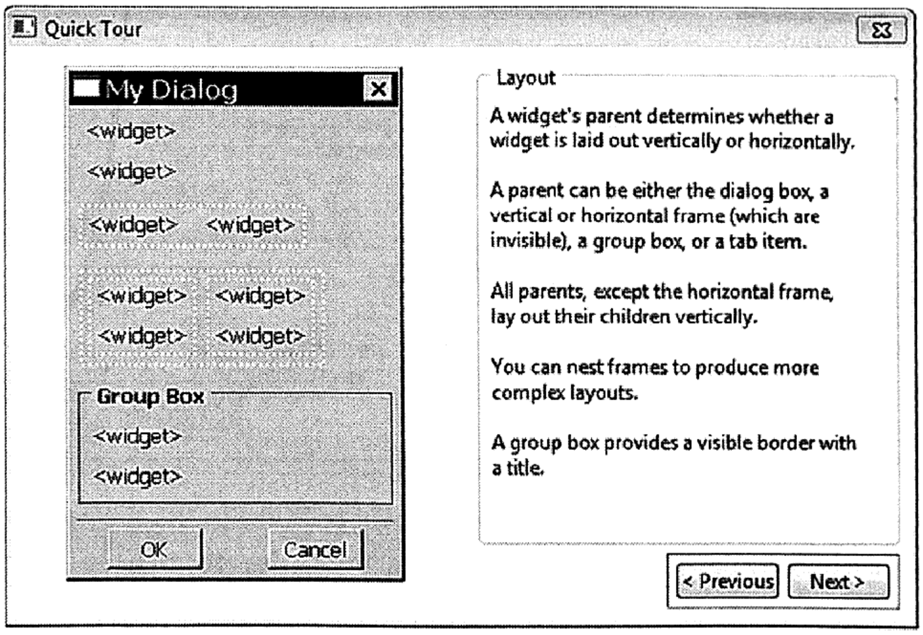
You associate your kernel command with your dialog box by specifying your module name and your function name.

The arguments to your function are provided by the widgets you add to your dialog box.

The dialog box will issue a command to the kernel in this format:

moduleName.functionName(key1=val1, ...)

This window is where we will link the RSG to our Python script. The script itself will form what is labeled at the module, and the function within the script will be the function called when the **OK** button is clicked in the dialog box. In the above figure, the module is 'myUtils' and the function is 'createPlate', which means that a function called 'createPlate()' will be called in a script called 'myUtils.py'.



Layout

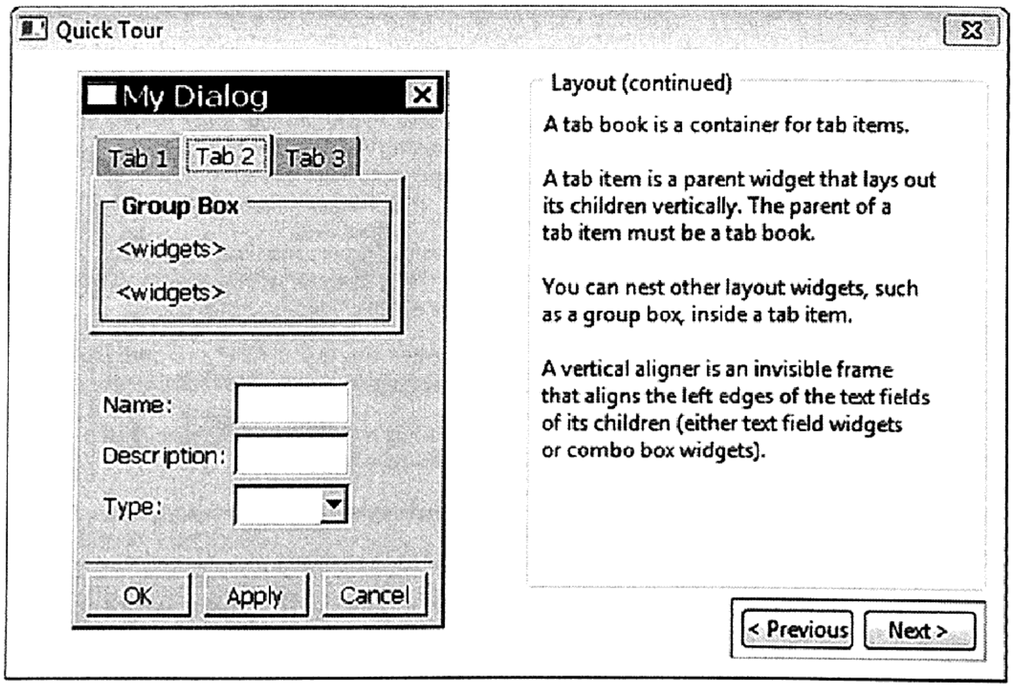
A widget's parent determines whether a widget is laid out vertically or horizontally.

A parent can be either the dialog box, a vertical or horizontal frame (which are invisible), a group box, or a tab item.

All parents, except the horizontal frame, lay out their children vertically.

You can nest frames to produce more complex layouts.

A group box provides a visible border with a title.



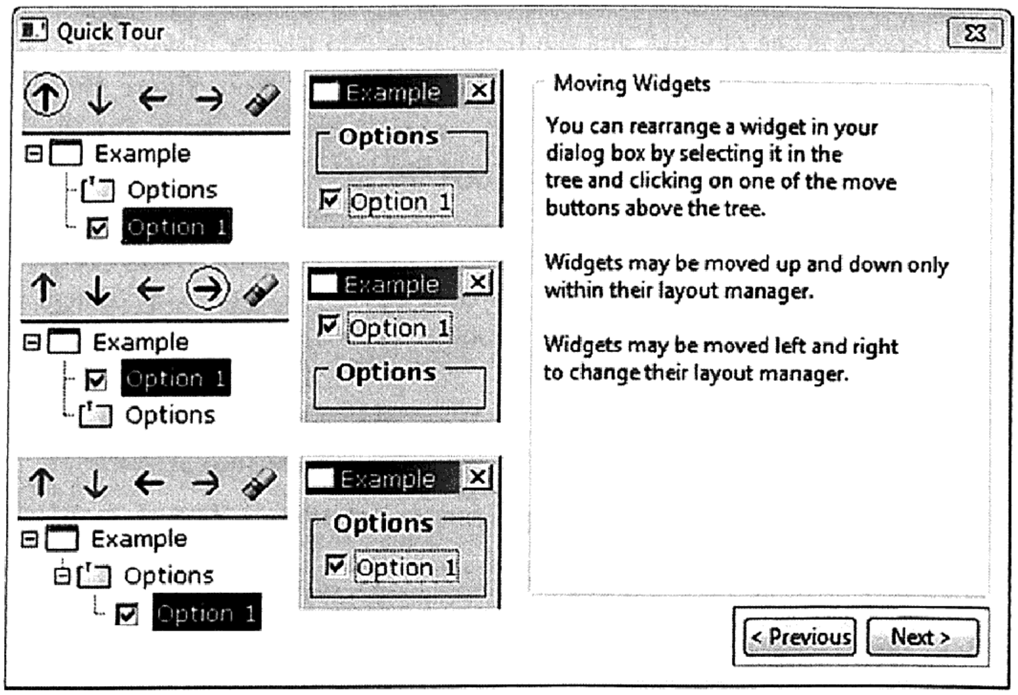
Layout (continued)

A tab book is a container for tab items.

A tab item is a parent widget that lays out its children vertically. The parent of a tab item must be a tab book.

You can nest other layout widgets, such as a group box, inside a tab item.

A vertical aligner is an invisible frame that aligns the left edges of the text fields of its children (either text field widgets or combo box widgets).



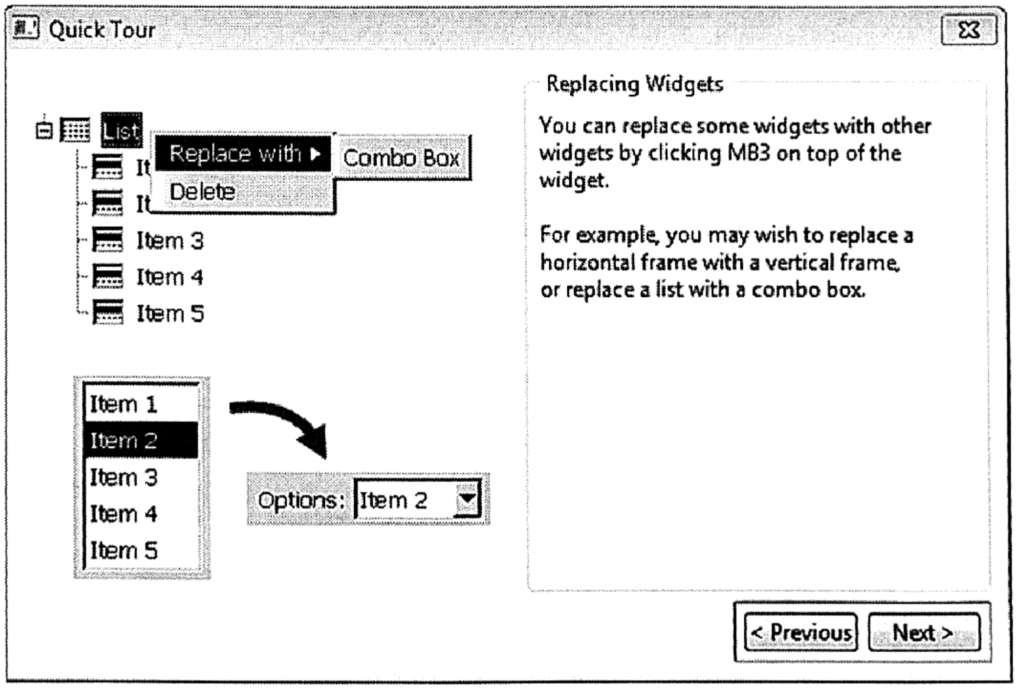
Moving Widgets

You can rearrange a widget in your dialog box by selecting it in the tree and clicking on one of the move buttons above the tree.

Widgets may be moved up and down only within their layout manager.

Widgets may be moved left and right to change their layout manager.

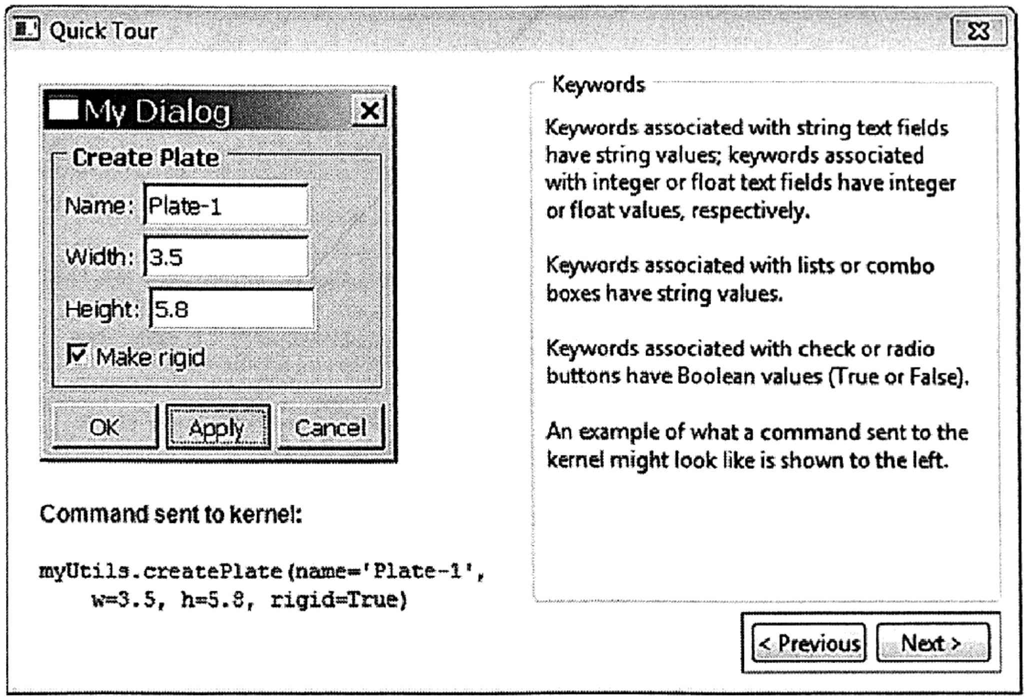
Moving widgets up and down tends to change their position in the dialog box. Moving widgets left and right allows you to nest them within a layout manager thus allowing them to be affected by the layout.



Replacing Widgets

You can replace some widgets with other widgets by clicking MB3 on top of the widget.

For example, you may wish to replace a horizontal frame with a vertical frame, or replace a list with a combo box.



Command sent to kernel:

```
myUtils.createPlate(name='Plate-1',
    w=3.5, h=5.8, rigid=True)
```

Keywords

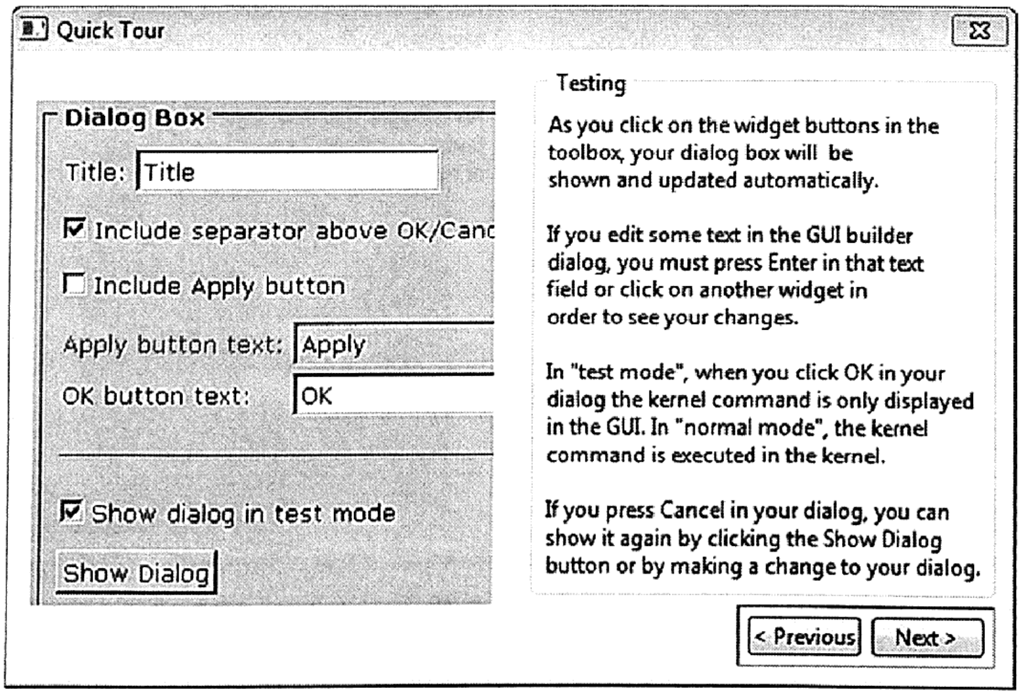
Keywords associated with string text fields have string values; keywords associated with integer or float text fields have integer or float values, respectively.

Keywords associated with lists or combo boxes have string values.

Keywords associated with check or radio buttons have Boolean values (True or False).

An example of what a command sent to the kernel might look like is shown to the left.

You associate keywords with each widget of the dialog box and also define the type of data it accepts. Here the text fields for name is given the keyword ‘name’ and accepts Strings. The other two fields are assigned the keywords ‘w’ and ‘h’ and accept floats. The checkbox's keyword is 'rigid' and it always returns a Boolean. These keywords and their values are passed to the function associated with the dialog box as parameters.



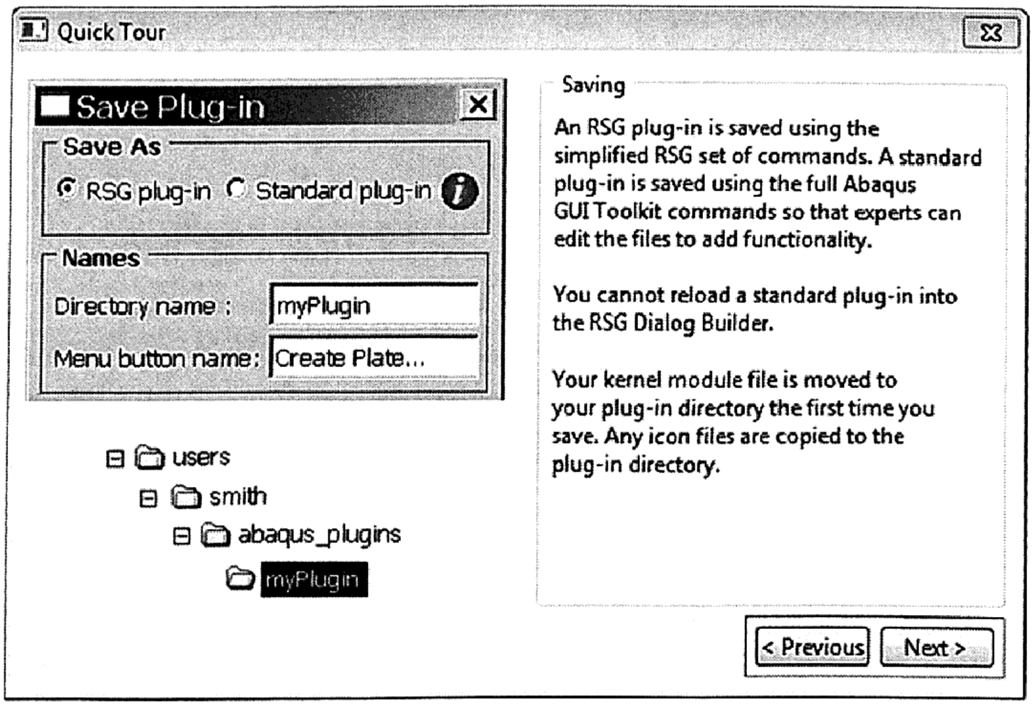
Testing

As you click on the widget buttons in the toolbox, your dialog box will be shown and updated automatically.

If you edit some text in the GUI builder dialog, you must press Enter in that text field or click on another widget in order to see your changes.

In "test mode", when you click OK in your dialog the kernel command is only displayed in the GUI. In "normal mode", the kernel command is executed in the kernel.

If you press Cancel in your dialog, you can show it again by clicking the Show Dialog button or by making a change to your dialog.

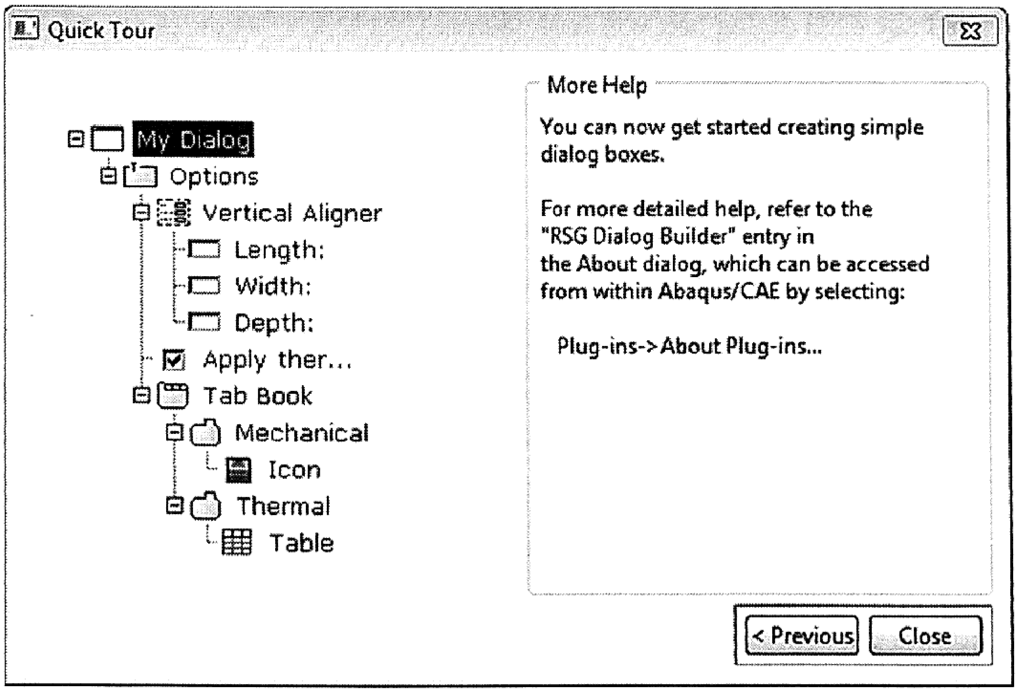


Saving

An RSG plug-in is saved using the simplified RSG set of commands. A standard plug-in is saved using the full Abaqus GUI Toolkit commands so that experts can edit the files to add functionality.

You cannot reload a standard plug-in into the RSG Dialog Builder.

Your kernel module file is moved to your plug-in directory the first time you save. Any icon files are copied to the plug-in directory.



You can now get started creating simple dialog boxes.

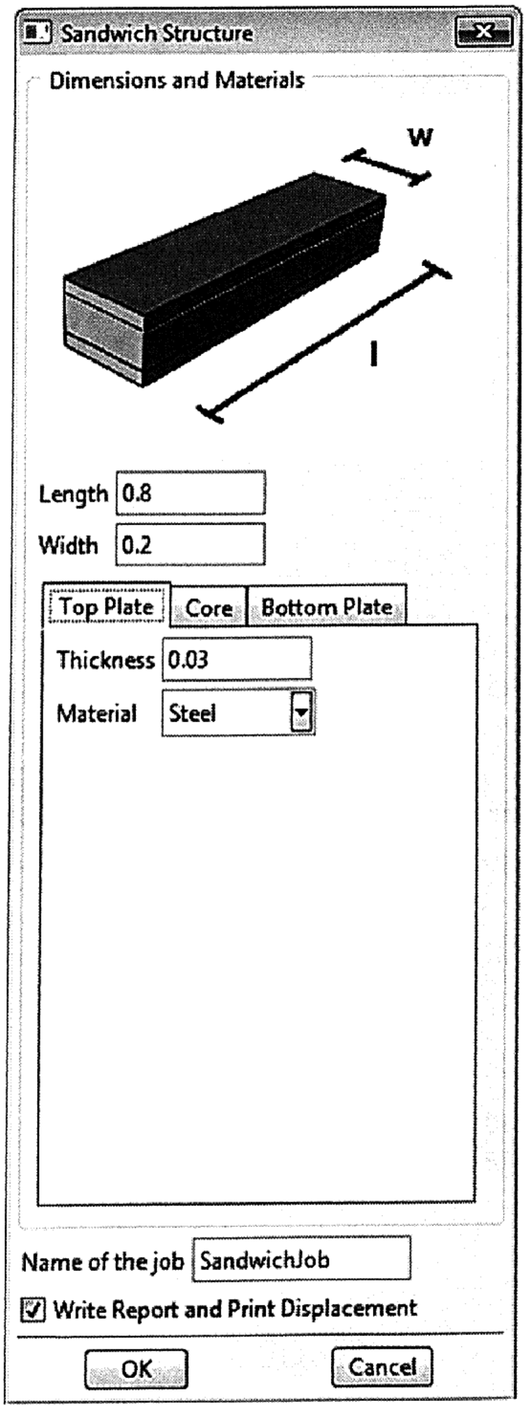
For more detailed help, refer to the "RSG Dialog Builder" entry in the About dialog, which can be accessed from within Abaqus/CAE by selecting:

Plug-ins->About Plug-ins...

19.4 Create an RSG for Sandwich Structure Analysis

Now that you've run through the 5 minute tutorial and got an idea of how RSG works, let's work through our example. I have already gone ahead and created a GUI dialog box. Laying the widgets out onto the canvas is simple enough but you should try it once and obtain the same layout that I have here.

Here is what our RSG dialog box will look like:



Sandwich Structure

Dimensions and Materials

Length 0.8

Width 0.2

Top Plate | Core | Bottom Plate

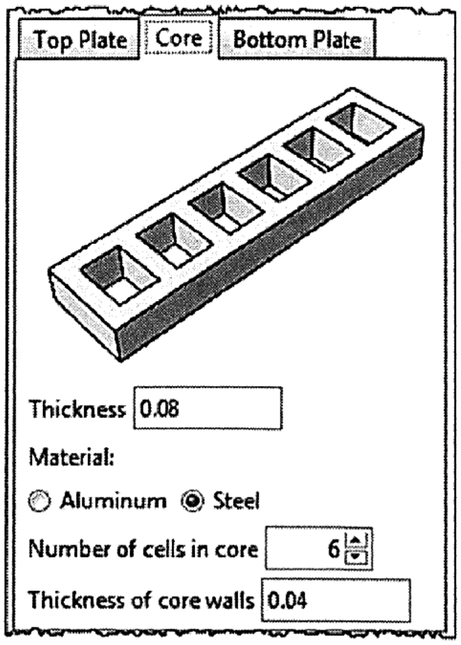
Thickness 0.03

Material Steel

Name of the job SandwichJob

☑ Write Report and Print Displacement

OK Cancel



Top Plate | Core | Bottom Plate

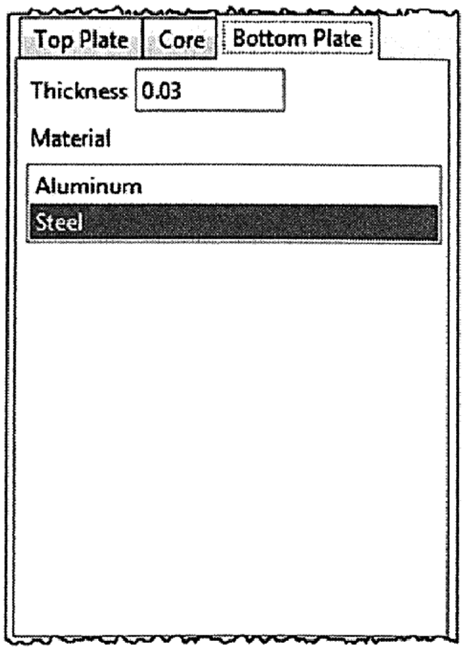
Thickness 0.08

Material:

○ Aluminum ● Steel

Number of cells in core 6

Thickness of core walls 0.04



Top Plate | Core | Bottom Plate

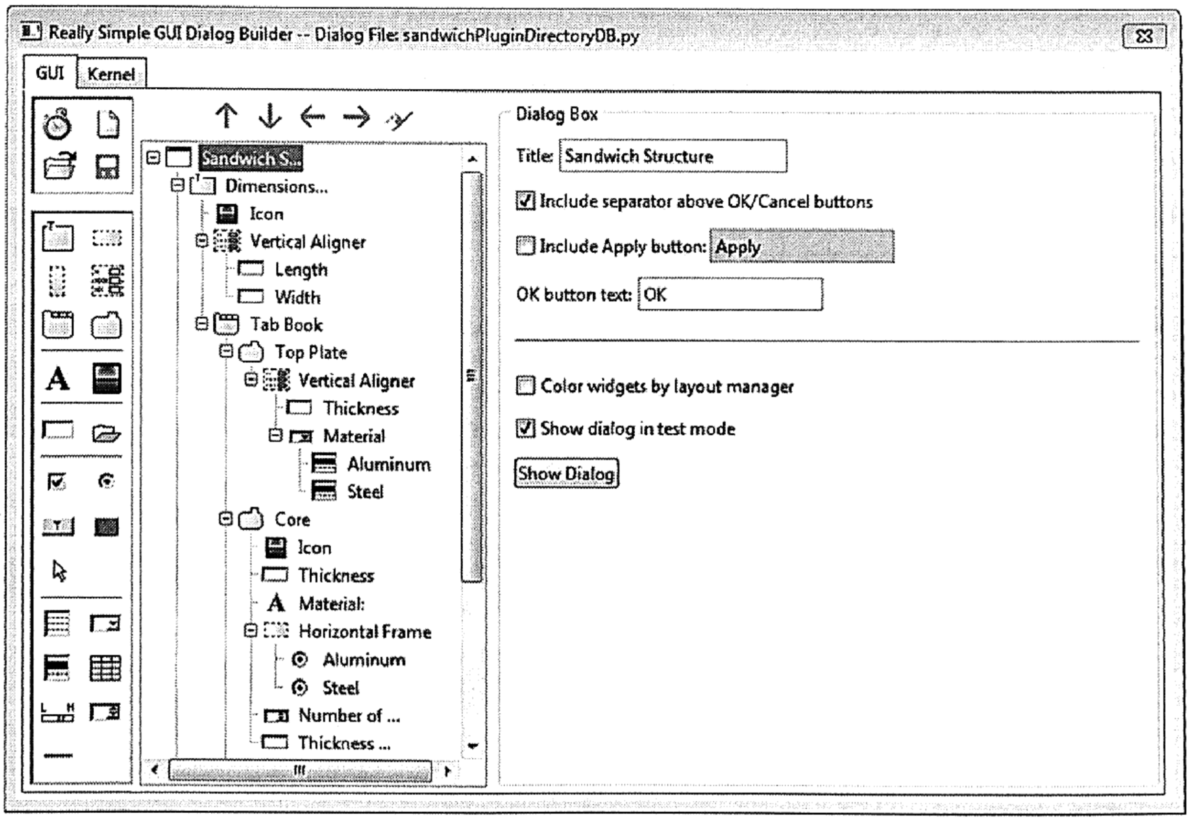
Thickness 0.03

Material

Aluminum

Steel

Lets focus on the parameters used to create this.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

GUI | Kernel

```
Sandwich S...
  Dimensions...
    Icon
    Vertical Aligner
      Length
      Width
    Tab Book
      Top Plate
        Vertical Aligner
          Thickness
          Material
            Aluminum
            Steel
      Core
        Icon
        Thickness
        Material:
        Horizontal Frame
          Aluminum
          Steel
        Number of ...
        Thickness ...
```

Dialog Box

Title: Sandwich Structure

[✓] Include separator above OK/Cancel buttons

[ ] Include Apply button: Apply

OK button text: OK

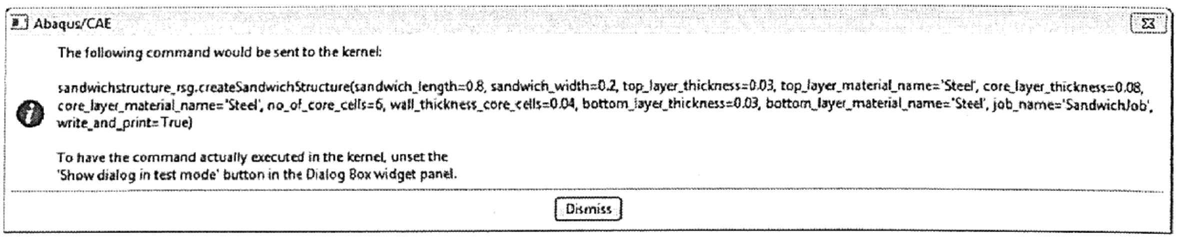
[ ] Color widgets by layout manager

[✓] Show dialog in test mode

Show Dialog

Here you see the settings for the plugin. The title ‘Sandwich Structure’ will appear in the title bar of the dialog box. We are including a separator, which is a horizontal bar, above the OK and Cancel buttons by checking the option. We have set the OK button text to the default of “OK” although you can change it to something else if you prefer.

If you click the ‘Show Dialog’ button, you will see the dialog box. ‘Show dialog in test mode’ is currently checked for testing purposes. This means that when you click OK Abaqus will not actually run the script. Instead it will display a message:


Abaqus/CAE

The following command would be sent to the kernel:

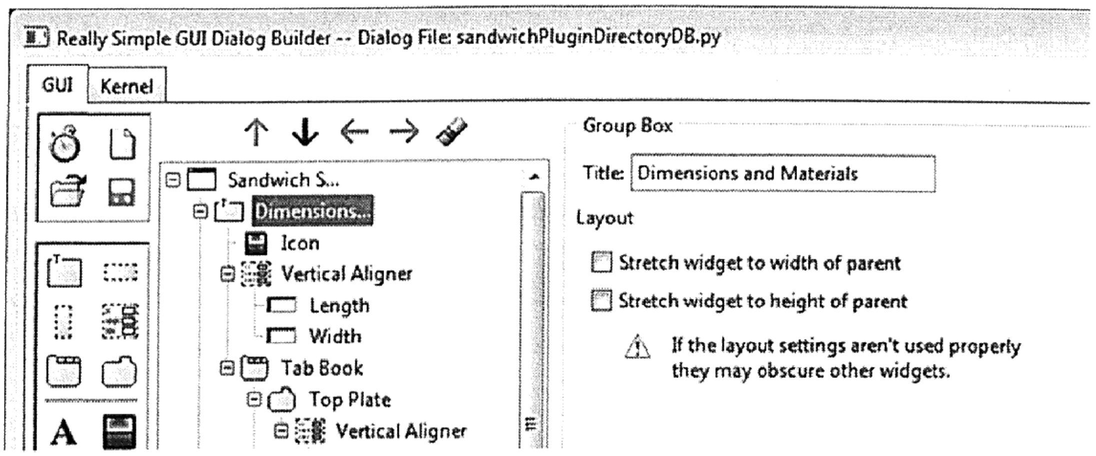
sandwichstructure_rsg.createSandwichStructure(sandwich_length=0.8, sandwich_width=0.2, top_layer_thickness=0.03, top_layer_material_name='Steel', core_layer_thickness=0.08, core_layer_material_name='Steel', no_of_core_cells=6, wall_thickness_core_cells=0.04, bottom_layer_thickness=0.03, bottom_layer_material_name='Steel', job_name='SandwichJob', write_and_print=True)

To have the command actually executed in the kernel, unset the 'Show dialog in test mode' button in the Dialog Box widget panel.

Abaqus indicates that it will call the **createSandwichStructure()** method in the Sandwichstructure_rsg.py file with the statement

```
Sandwichstructure_rsg.createSandwichStructure(sandwich_length=0.8,
sandwich_width=0.2, width=0.2, top_layer_thicker=0.03,
top_layer_material_name='Steel', core_layer_thickness=0.08,
core_layer_material_name='Steel', no_of_core_cells=6, wall_thickness_core_cell=0.04,
bottom_layer_thickness=0.03, bottom_layer_material_name='Steel',
job_name='SandwichJob', write_and_print=True).
```

All the widgets are placed inside a group box which we have given the title 'Dimensions and Materials'.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

Title: Dimensions and Materials

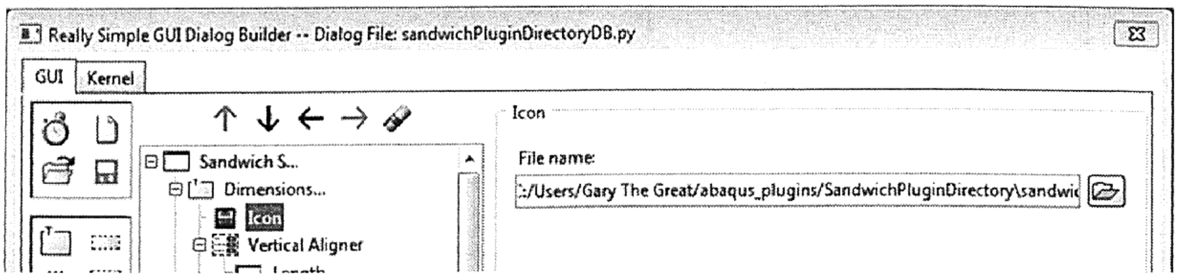
Layout

Stretch widget to width of parent

Stretch widget to height of parent

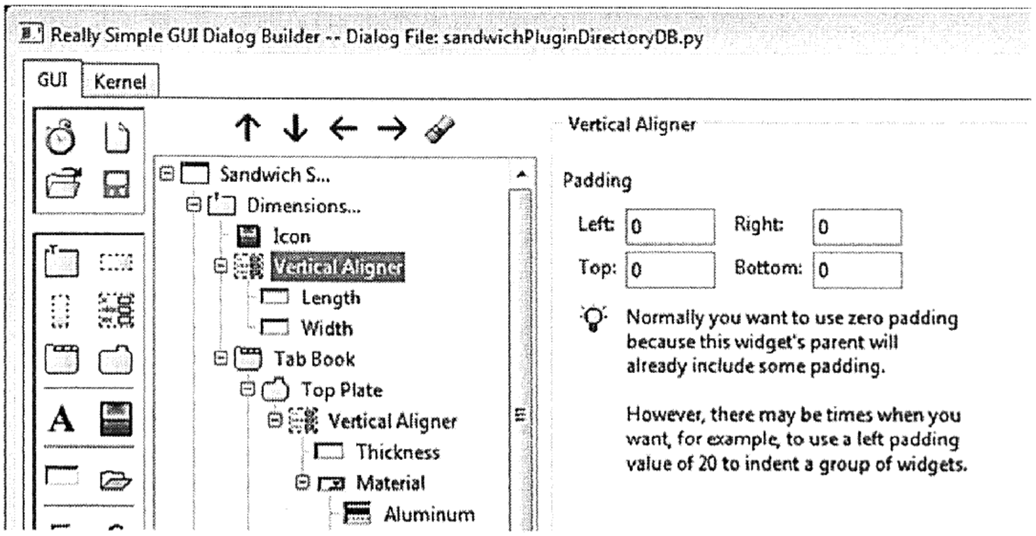
If the layout settings aren't used properly they may obscure other widgets.

An icon widget is used to add the image. The path to the image is specified here.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

File name: :/Users/Gary The Great/abaqus_plugins/SandwichPluginDirectory\sandwic

We create a vertical aligner widget to position the length and width text fields vertically. Any items placed inside a vertical aligner are automatically positioned vertically. We will not apply any padding to this vertical aligner.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

Vertical Aligner

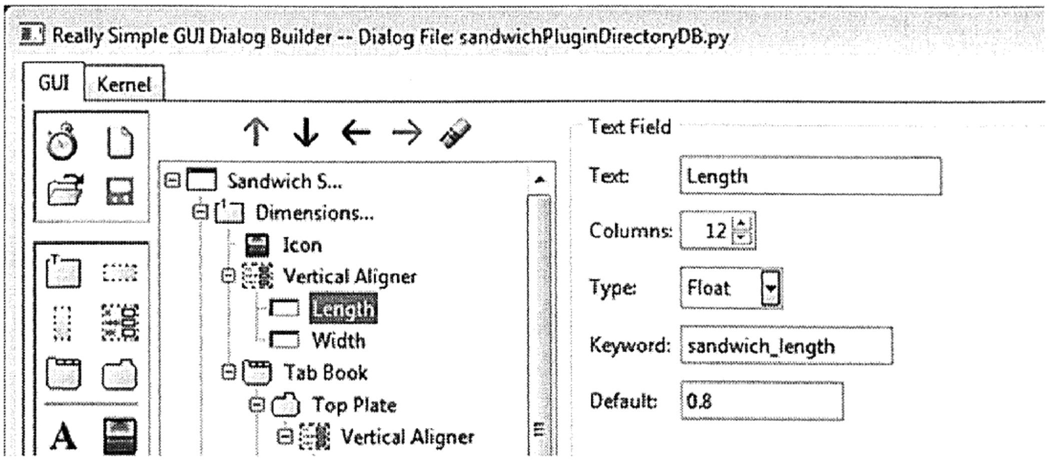
Padding

Left: 0 Right: 0

Top: 0 Bottom: 0

Normally you want to use zero padding because this widget's parent will already include some padding.

However, there may be times when you want, for example, to use a left padding value of 20 to indent a group of widgets.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

Text Field

Text: Length

Columns: 12

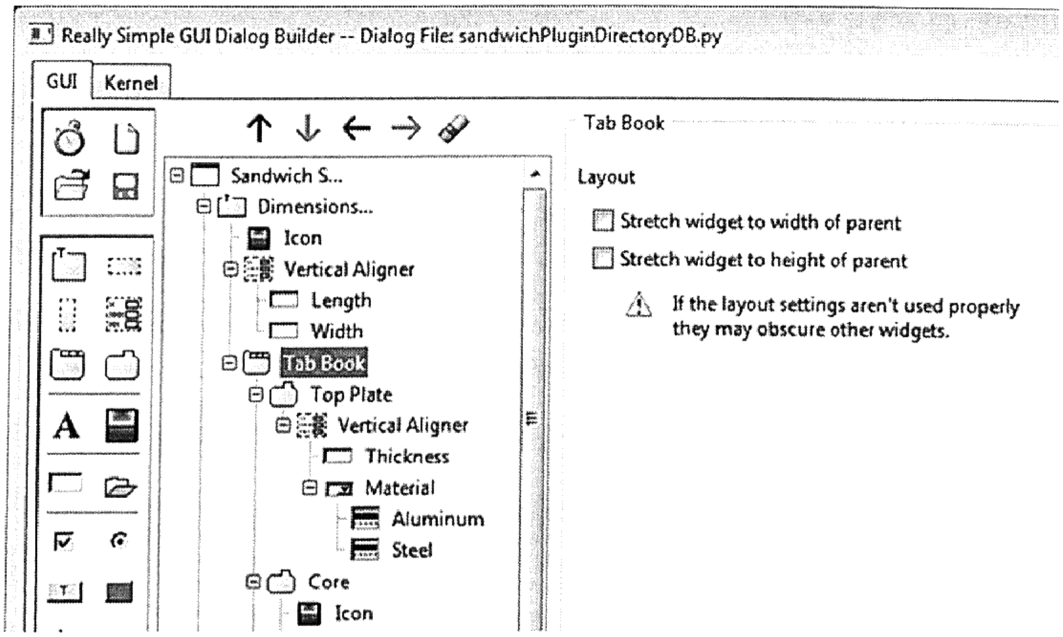
Type: Float

Keyword: sandwich_length

Default: 0.8

The length text field is defined here. The text is set to ‘Length’ hence the word ‘Length’ will appear next to the text field on the canvas. The number of columns is set to 12 meaning that 12 characters will be visible in the text field. You can actually type more characters, but the whole line will shift left as you type more and you will only be able to see 12 characters/digits. This is more than enough room for our purposes. The type is set to ‘Float’ indicating that a float value is expected here and this will be passed to a float variable. The keyword **sandwich_length** is associated with this text field, hence when the **OK** button of the dialog box is pressed the function **createSandwichStructure()** will be passed the parameter **sandwich_length=xyz** where **xyz** is the float entered by the user. The default is set to 0.8.

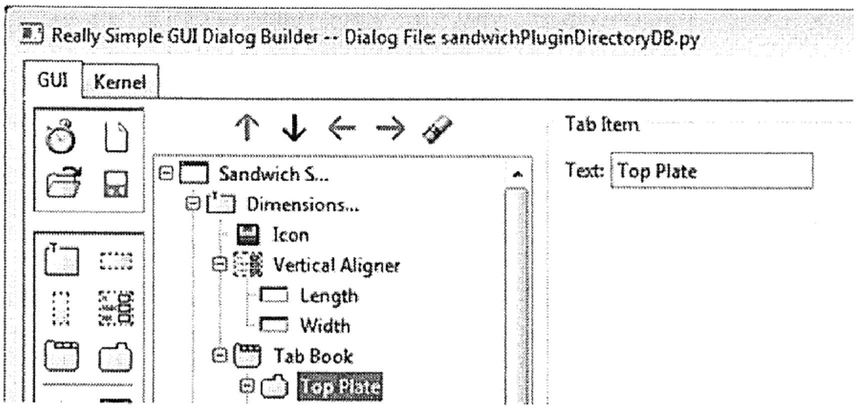
The definition of the width text field is similar. It is assigned the text 'Width', the keyword associated with it is **sandwich_width** and the default value is 0.2.



Stretch widget to width of parent
Stretch widget to height of parent
If the layout settings aren't used properly they may obscure other widgets.

A tab book widget is used to create a tabbed section. Each of the tabs – Top Plate, Core and Bottom Plate will be individual containers nested within the tab book container.

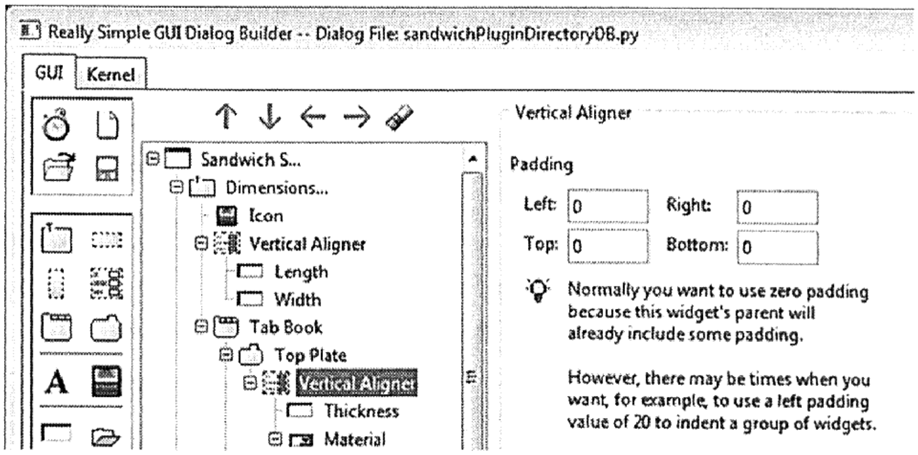
The Top Plate container will accept settings for the top plate. We give it the title 'Top Plate' which appears as the name of the tab in the tab book.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

Tab Item

Text: Top Plate

A vertical aligner is used to position the widgets inside the top plate tab.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

Vertical Aligner

Padding

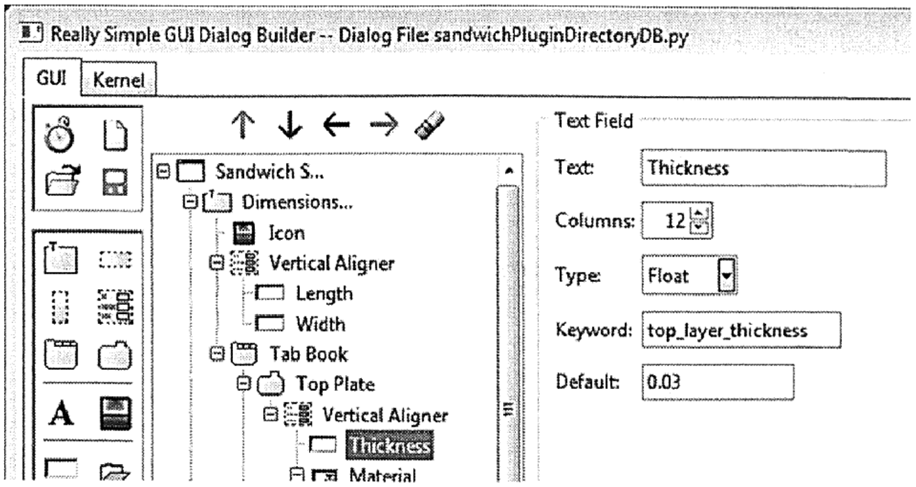
Left: 0 Right: 0

Top: 0 Bottom: 0

Normally you want to use zero padding because this widget's parent will already include some padding.

However, there may be times when you want, for example, to use a left padding value of 20 to indent a group of widgets.

The text field 'Thickness' specifies the thickness of the top plate of the sandwich structure and is assigned the keyword **top_layer_thickness**.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

Text Field

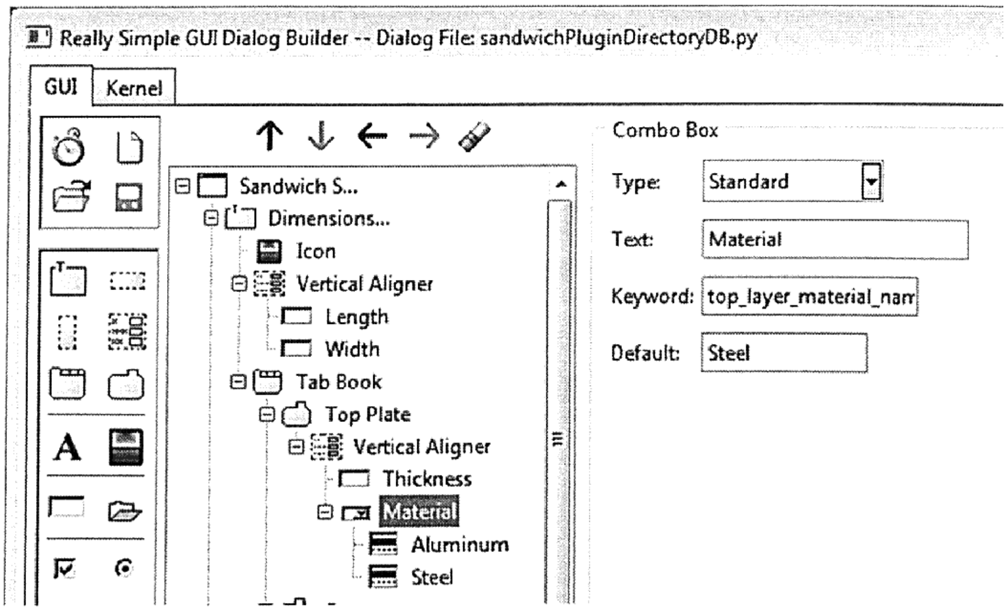
Text: Thickness

Columns: 12

Type: Float

Keyword: top_layer_thickness

Default: 0.03


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

Combo Box

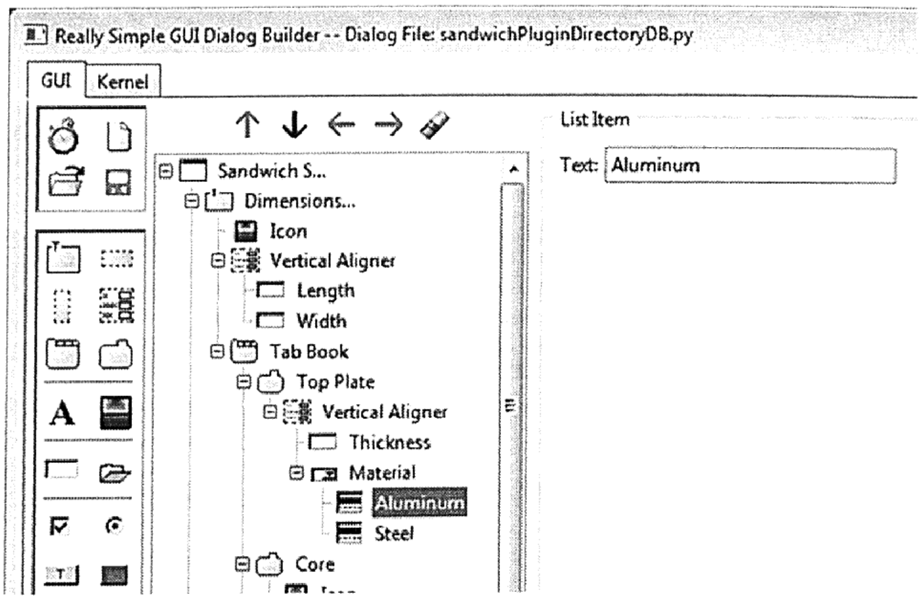
Type: Standard

Text: Material

Keyword: top_layer_material_nam

Default: Steel

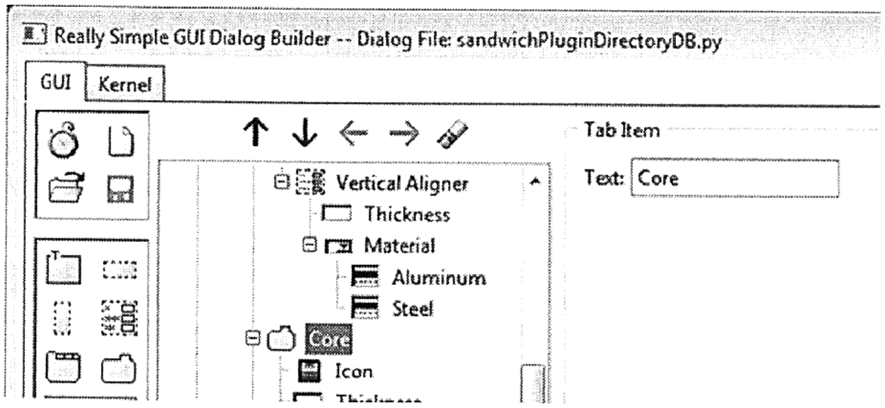
A standard combo box named ‘Material’ is created here. It is assigned the keyword **top_layer_material_name**. The default value has been set to ‘Steel’ which is one of the combo box items. Notice that the default value has been spelt exactly as the name of the combo box item ‘Steel’. If you were to type anything other than ‘Aluminum’ or ‘Steel’ in the default field, it would be meaningless to Abaqus.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

List Item

Text: Aluminum

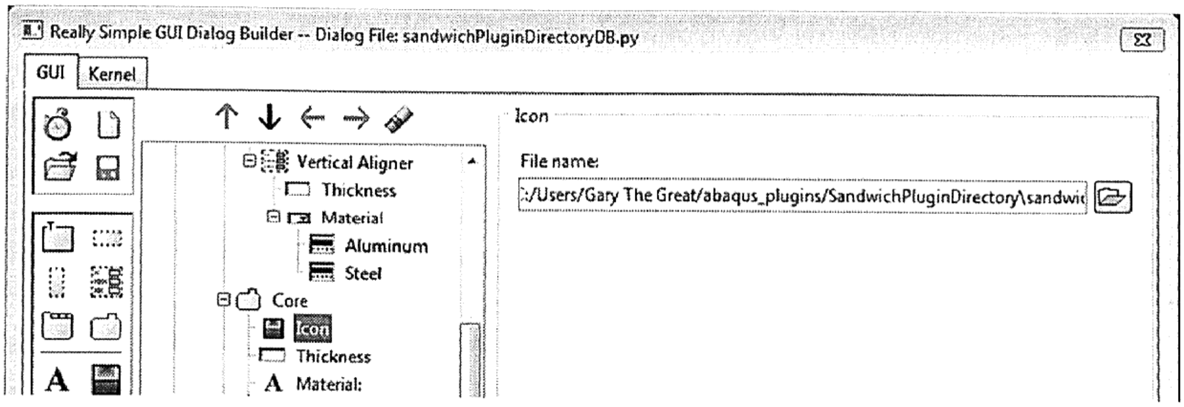
A combo box item ‘Aluminum’ is added here, followed by one named ‘Steel’.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

Tab Item

Text: Core

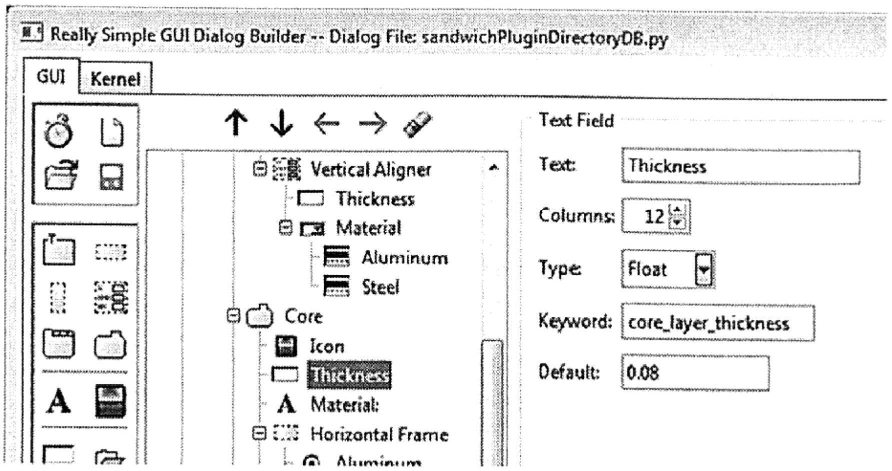
The second tab is named 'Core' and the user will define the properties of the core here.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

Icon

File name: :/Users/Gary The Great/abaqus_plugins/SandwichPluginDirectory\sandwi

The icon widget is used to place an image of the core in the core tab.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

Text Field

Text: Thickness

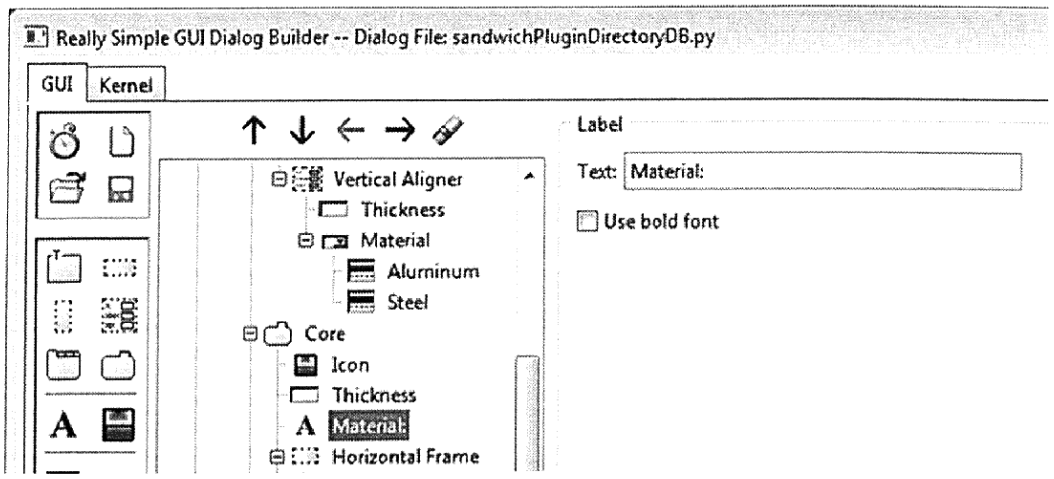
Columns: 12

Type: Float

Keyword: core_layer_thickness

Default: 0.08

A text field labeled thickness is created and assigned the keyword **core_layer_thickness** and a default value of 0.08.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

GUI | Kernel

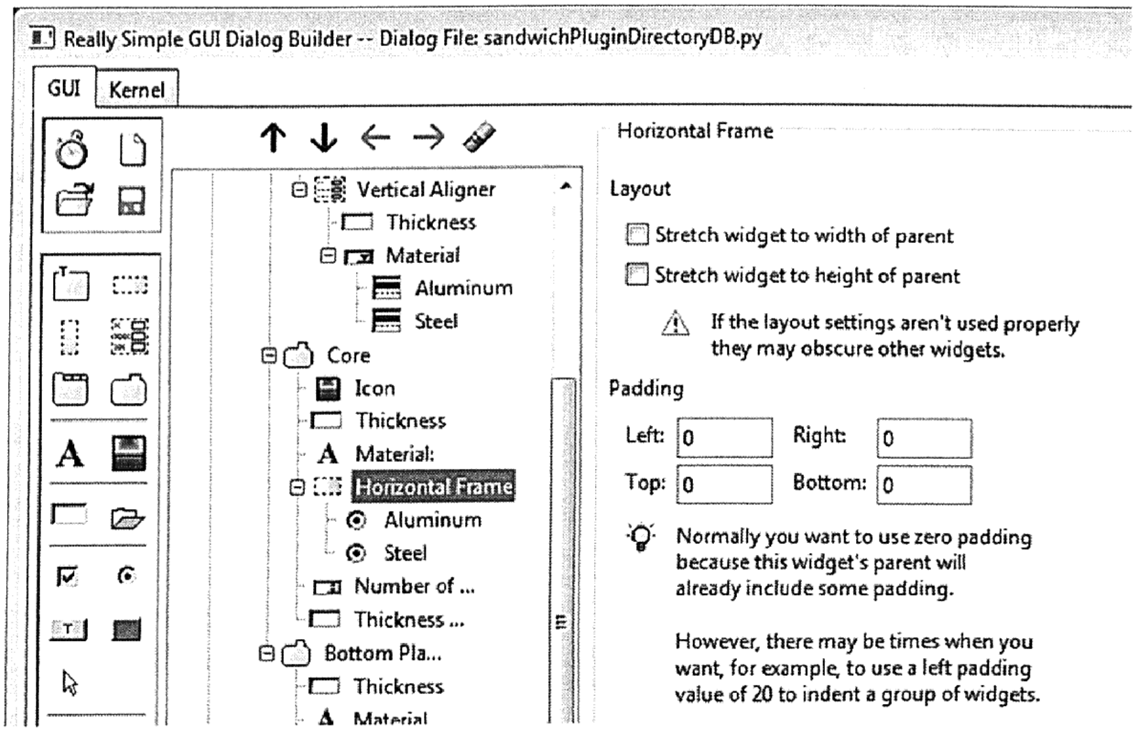
Vertical Aligner
Thickness
Material
Aluminum
Steel
Core
Icon
Thickness
Material:
Horizontal Frame

Label

Text: Material:

Use bold font

A read only text label with the text 'Material' is added to the core tab.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

GUI | Kernel

Vertical Aligner
Thickness
Material
Aluminum
Steel
Core
Icon
Thickness
Material:
Horizontal Frame
Aluminum
Steel
Number of ...
Thickness ...
Bottom Pla...
Thickness
Material

Horizontal Frame

Layout

Stretch widget to width of parent

Stretch widget to height of parent

If the layout settings aren't used properly they may obscure other widgets.

Padding

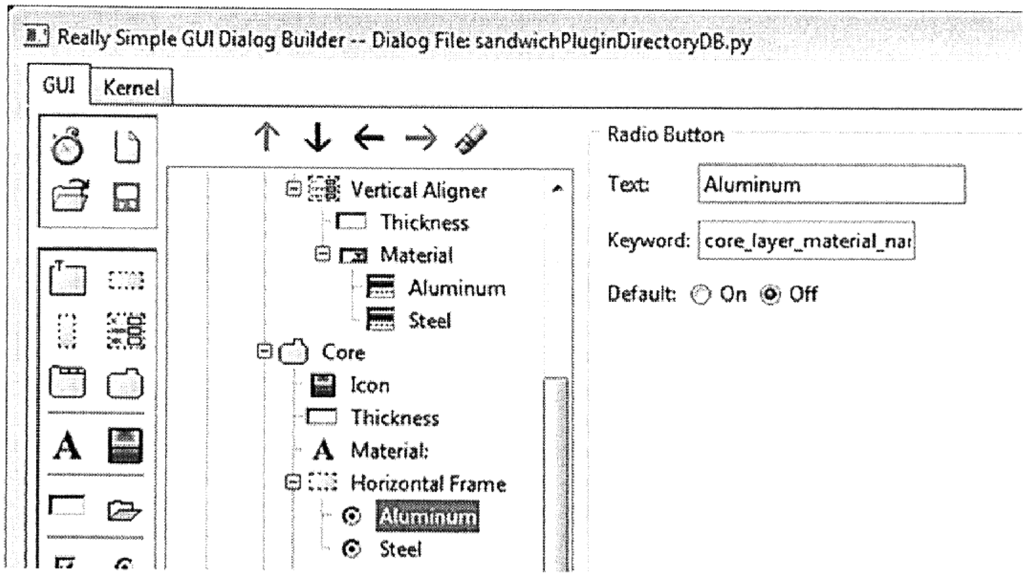
Left: 0 Right: 0

Top: 0 Bottom: 0

Normally you want to use zero padding because this widget's parent will already include some padding.

However, there may be times when you want, for example, to use a left padding value of 20 to indent a group of widgets.

A horizontal frame is created in which we will place the radio buttons for the two materials. This will make them appear side by side.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

GUI | Kernel

Vertical Aligner
Thickness
Material
Aluminum
Steel
Core
Icon
Thickness
Material:
Horizontal Frame
Aluminum
Steel

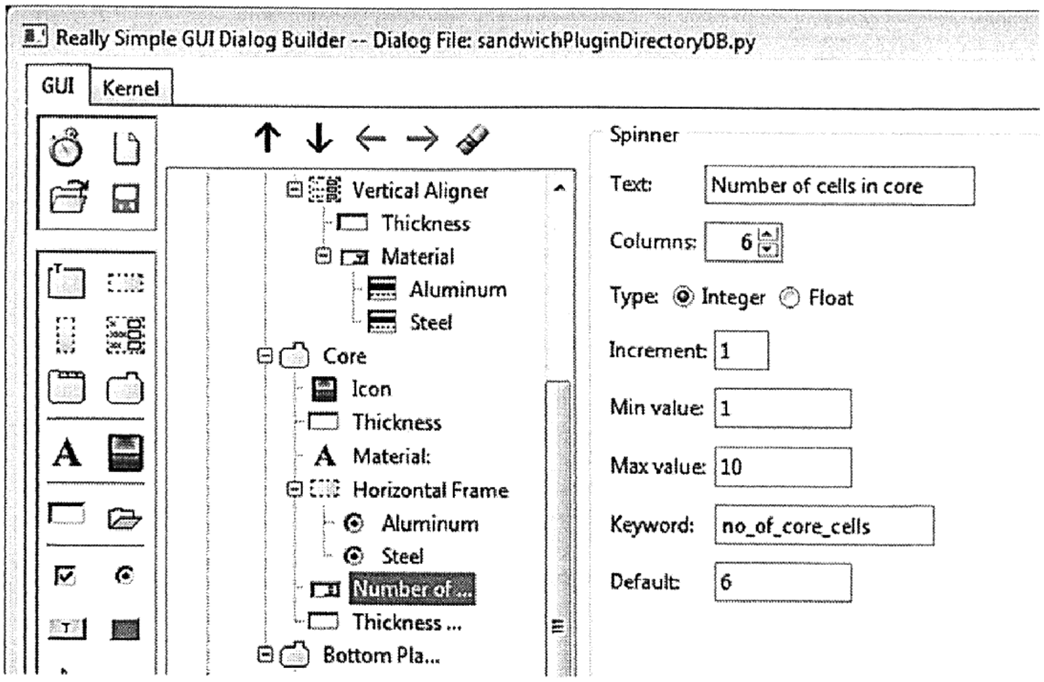
Radio Button

Text: Aluminum

Keyword: core_layer_material_nai

Default: ( ) On [✓ Off]

Radio buttons are created for ‘Aluminum’ and ‘Steel’. Radio buttons allow you to select just one out of a set of options. If you select one radio button, the other will get deselected. In order to enforce this behavior, both radio buttons must be given the same keyword **core_layer_material_name.** If they are given different keywords they will not be part of the same radio group and will operate independently, meaning that you will be able to select both of them at the same time which will be quite meaningless.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

GUI | Kernel

Vertical Aligner
Thickness
Material
Aluminum
Steel
Core
Icon
Thickness
Material:
Horizontal Frame
Aluminum
Steel
Number of ...
Thickness ...
Bottom Pla...

Spinner

Text: Number of cells in core

Columns: 6

Type: [✓ Integer] ( ) Float

Increment: 1

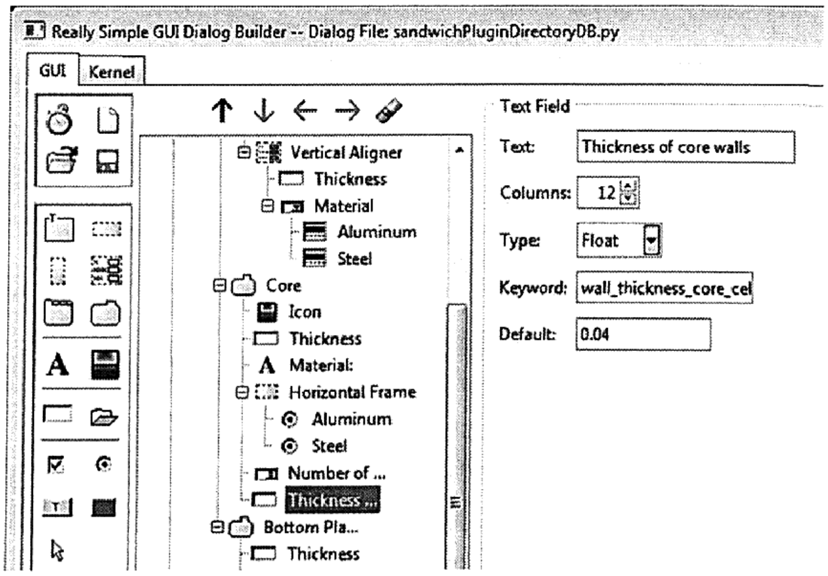
Min value: 1

Max value: 10

Keyword: no_of_core_cells

Default: 6

A spinner is used to allow the user to select the number of cells in the core. It is given the label text 'Number of cells in core' which will appear next to it in the dialog box. It will allow the user to select a value between the specified minimum of 1 and the specified maximum which is 10. The default has been set to 6. The selected value will be passed to the parameter **no_of_core_cells**.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

Text Field

Text: Thickness of core walls

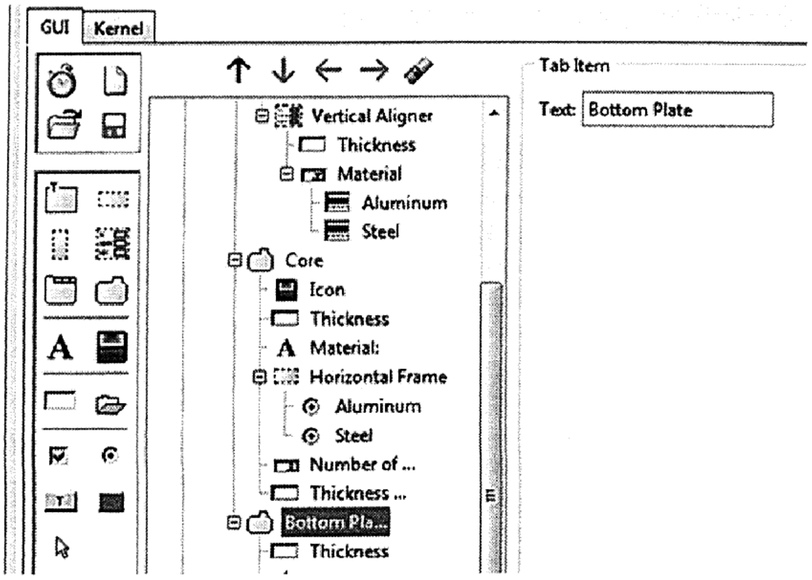
Columns: 12

Type: Float

Keyword: wall_thickness_core_cel

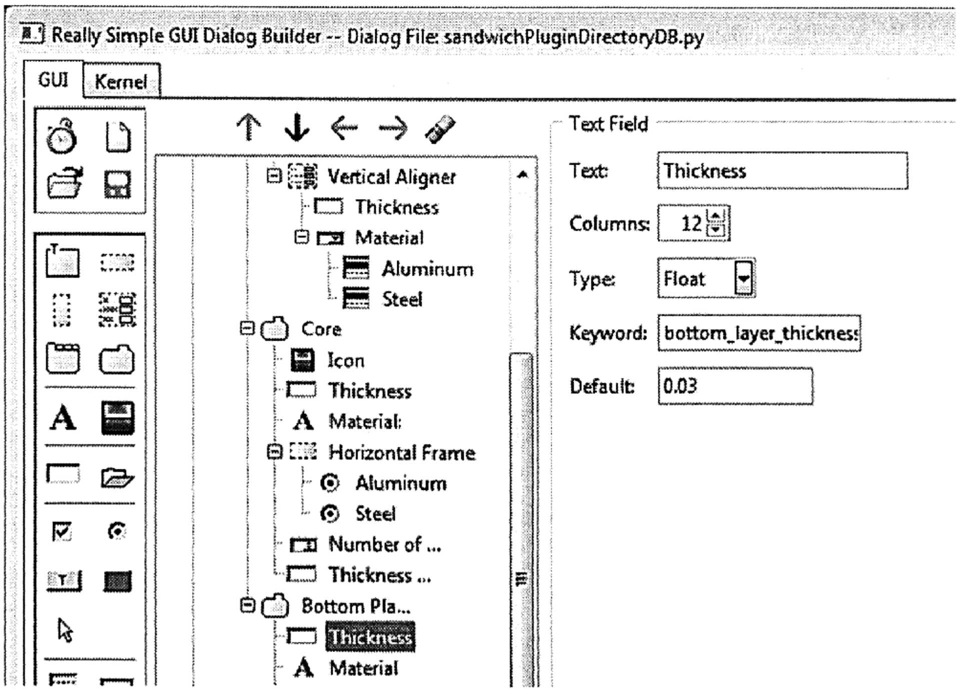
Default: 0.04

A text field is supplied for the user to enter the thickness of the walls of the core cells.


Tab Item

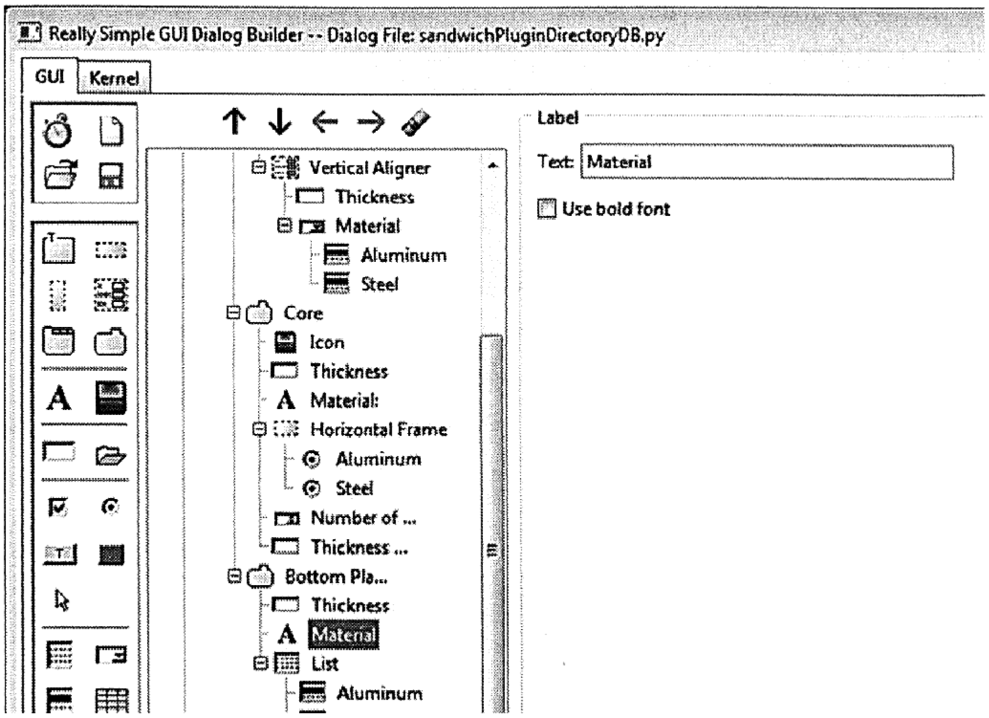
Text: Bottom Plate

The third tab is named 'Bottom Plate'.



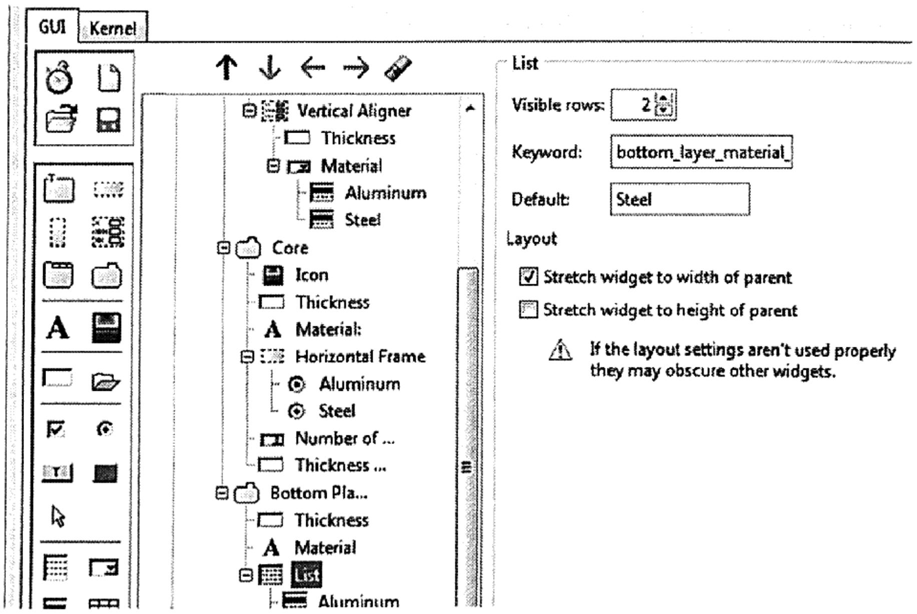
Text: Thickness
Columns: 12
Type: Float
Keyword: bottom_layer_thickness
Default: 0.03

A text field is supplied for the user to enter the thickness of the bottom layer.



Text: Material
Use bold font

A text label 'Material' is inserted on the canvas.



List

Visible rows: 2

Keyword: bottom_layer_material_

Default: Steel

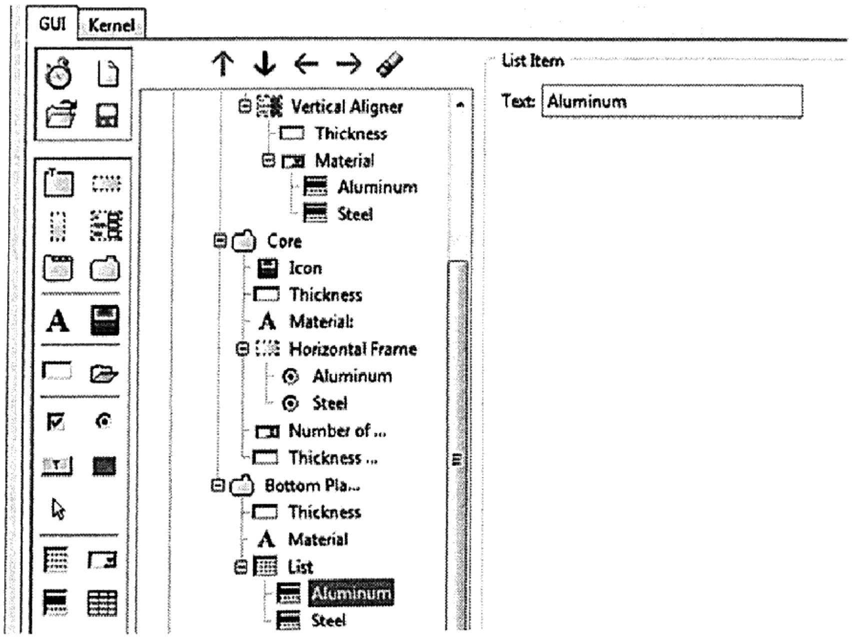
Layout

[✓] Stretch widget to width of parent

[ ] Stretch widget to height of parent

If the layout settings aren't used properly they may obscure other widgets.

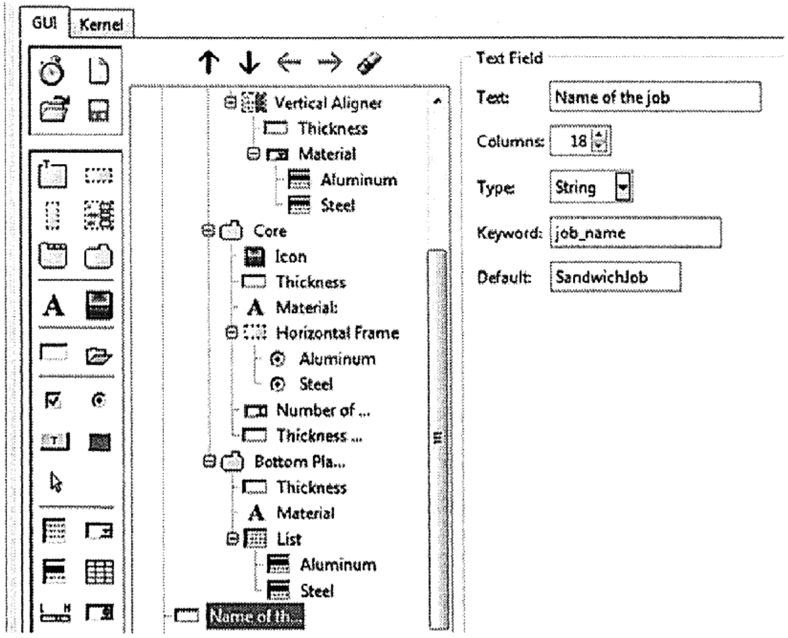
A list is used to provide the user with material options. The keyword **bottom_layer_material_name** is applied to the list container itself rather than individual list items. The default is set to 'Steel' which is one of the list items. Note that the default must be a name of one of the list items, in this case 'Aluminum' or 'Steel' otherwise it would be meaningless.



List Item

Text: Aluminum

List items 'Aluminum' and 'Steel' are added to the list container.



Text Field

Text: Name of the job

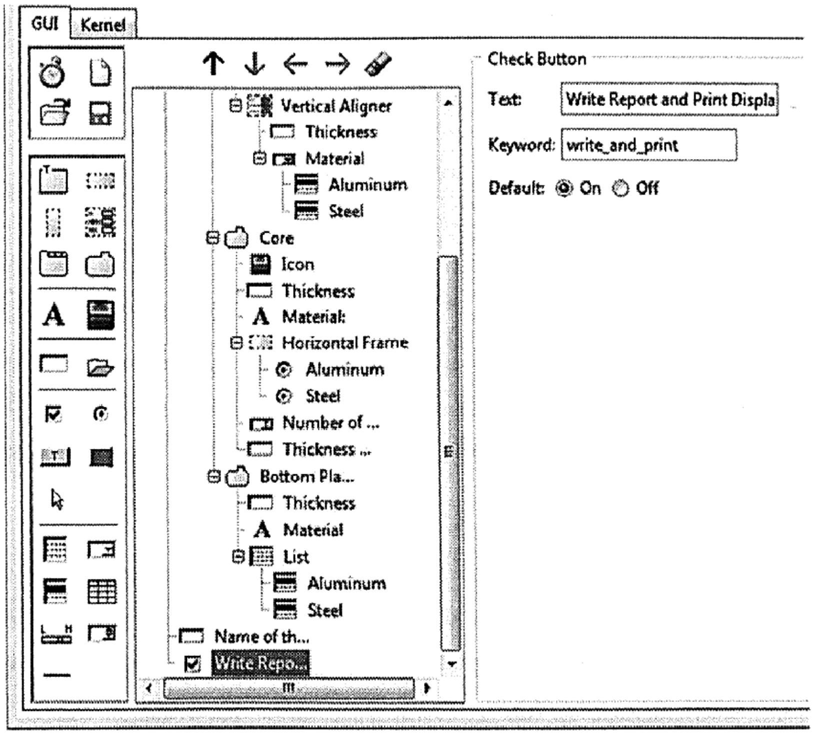
Columns: 18

Type: String

Keyword: job_name

Default: SandwichJob

A text field is provided for the user to supply the job name.



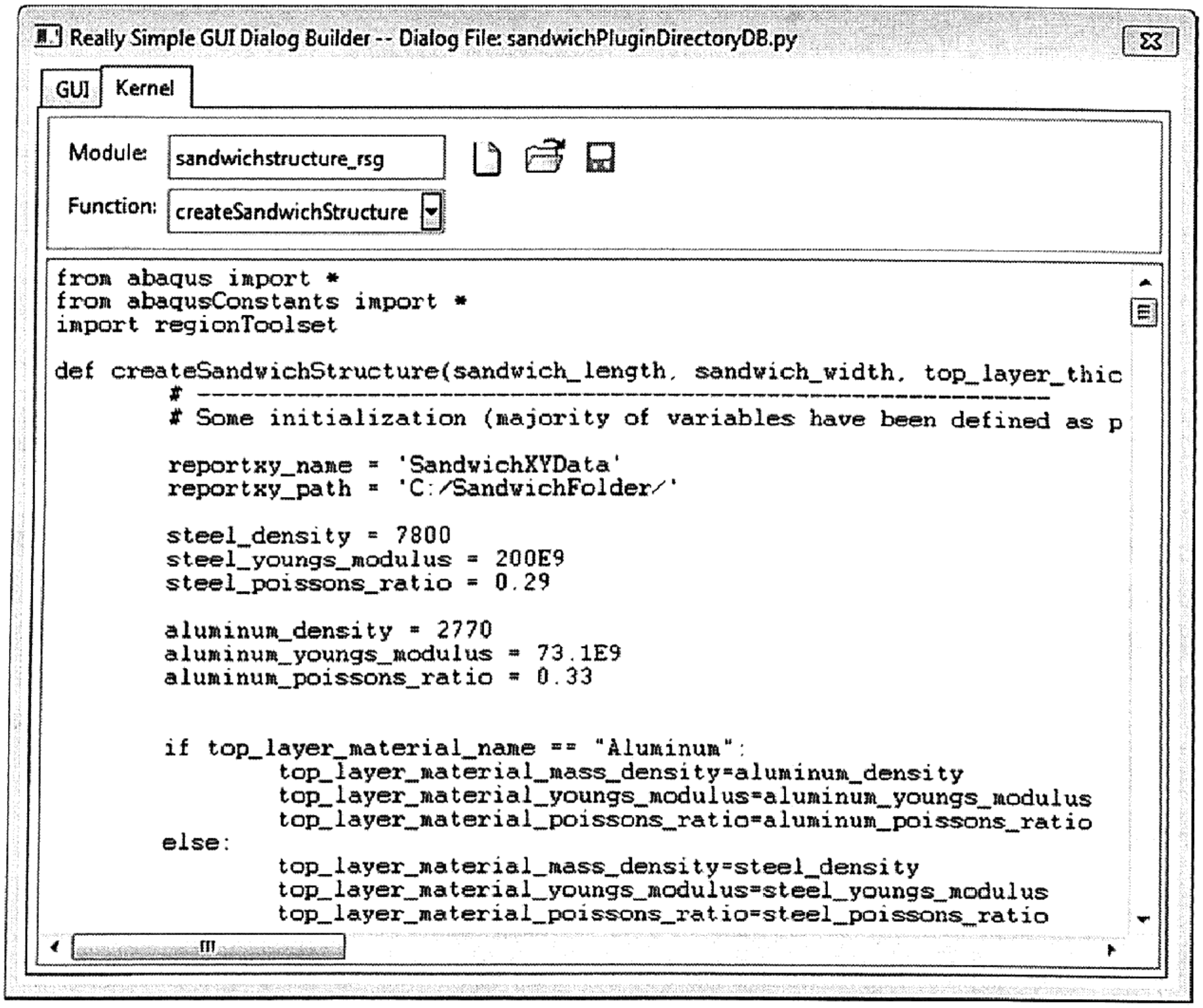
Check Button

Text: Write Report and Print Displa

Keyword: write_and_print

Default: [✓ On] Off

A checkbox allows the user to specify whether or not the XY report should be written and the displacement subsequently printed to the message area.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py
GUI
Kernel

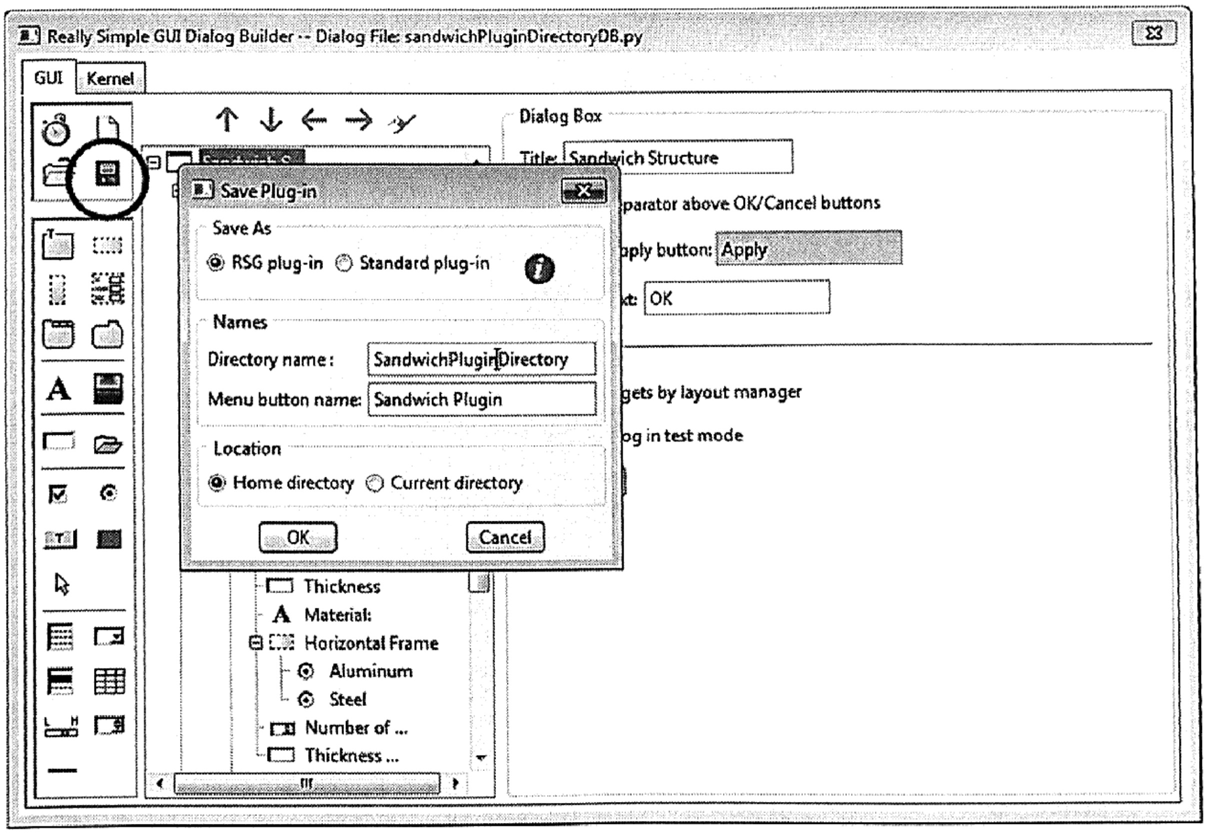
Module: sandwichstructure_rsg

Function: createSandwichStructure

```
from abaqus import *
from abaqusConstants import *
import regionToolset

def createSandwichStructure(sandwich_length, sandwich_width, top_layer_thic
        # ---------------------------------------------------------------
        # Some initialization (majority of variables have been defined as p

        reportxy_name = 'SandwichXYData'
        reportxy_path = 'C:/SandwichFolder/'

        steel_density = 7800
        steel_youngs_modulus = 200E9
        steel_poissons_ratio = 0.29

        aluminum_density = 2770
        aluminum_youngs_modulus = 73.1E9
        aluminum_poissons_ratio = 0.33


        if top_layer_material_name == "Aluminum":
                top_layer_material_mass_density=aluminum_density
                top_layer_material_youngs_modulus=aluminum_youngs_modulus
                top_layer_material_poissons_ratio=aluminum_poissons_ratio
        else:
                top_layer_material_mass_density=steel_density
                top_layer_material_youngs_modulus=steel_youngs_modulus
                top_layer_material_poissons_ratio=steel_poissons_ratio
```

In the **Kernel** tab, we set the module to 'sandwichstructure_rsg' and the function to 'createSandwichStructure'. This means our script will be in the file **sandwichstructure_rsg.py** and will contain a function called **createSandwichStructure()**.

We now save the RSG Dialog Box as a plug-in by clicking the 'Save your dialog box as a plug-in' button. We shall save it as an RSG plug-in, which means internally Abaqus will use RSG commands to construct it. If we were to save it as a standard plug-in, Abaqus would use the GUI toolkit commands instead. You will learn about those in the next two chapters. We set the location to 'Home directory' which tells Abaqus to save the plug-in in the default plug-ins folder. On my Windows 7 system this is C:\users\(username)\abaqus_plugins\. The directory name is the name of the directory in which the scripts will be stored – these scripts include the RSG plug-in startup, and RSG dialog construction scripts generated by Abaqus, as well as the kernel script written by us. The menu button name specified by you will be the name of the plug-in in the Plug-ins menu in Abaqus/CAE. Note that it will only be visible in the Plug-ins menu after you restart Abaqus/CAE.


Really Simple GUI Dialog Builder -- Dialog File: sandwichPluginDirectoryDB.py

Save Plug-in

Save As: [✓ RSG plug-in] Standard plug-in

Names

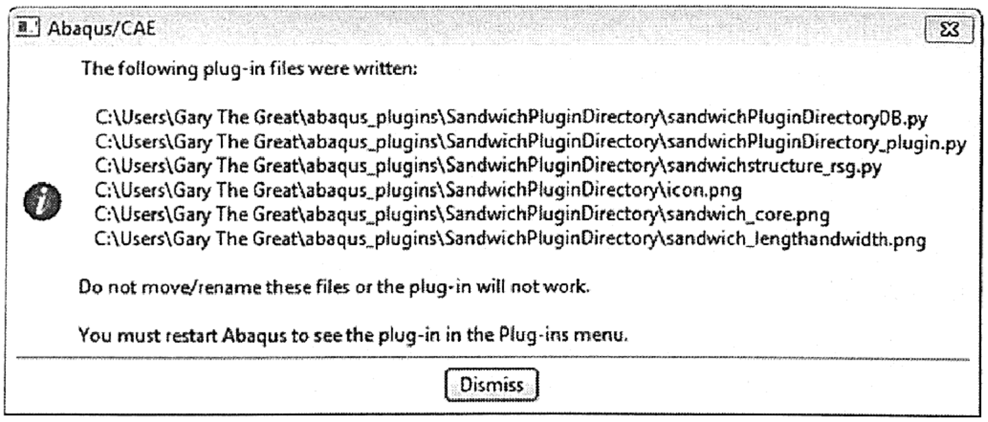
Directory name: SandwichPluginDirectory

Menu button name: Sandwich Plugin

Location: [✓ Home directory] Current directory

Title: Sandwich Structure

When you click OK Abaqus will inform you of which files were saved and where. Since we selected 'Home directory' these are saved in the 'abaqus_plugins' folder.


Abaqus/CAE

The following plug-in files were written:

```
C:\Users\Gary The Great\abaqus_plugins\SandwichPluginDirectory\sandwichPluginDirectoryDB.py
C:\Users\Gary The Great\abaqus_plugins\SandwichPluginDirectory\sandwichPluginDirectory_plugin.py
C:\Users\Gary The Great\abaqus_plugins\SandwichPluginDirectory\sandwichstructure_rsg.py
C:\Users\Gary The Great\abaqus_plugins\SandwichPluginDirectory\icon.png
C:\Users\Gary The Great\abaqus_plugins\SandwichPluginDirectory\sandwich_core.png
C:\Users\Gary The Great\abaqus_plugins\SandwichPluginDirectory\sandwich_lengthandwidth.png
```

Do not move/rename these files or the plug-in will not work.

You must restart Abaqus to see the plug-in in the Plug-ins menu.

19.5 Python Script to respond to the GUI dialog inputs

When the **OK** button is pressed, the **createSandwichStructure()** method will be called. The following script contains its definition.

```
from abaqus import *
from abaqusConstants import *
import regionToolset

def createSandwichStructure(sandwich_length, sandwich_width, top_layer_thickness,
                            core_layer_thickness, bottom_layer_thickness,
                            no_of_core_cells, wall_thickness_core_cells,
                            top_layer_material_name, bottom_layer_material_name,
                            core_layer_material_name, job_name, write_and_print):
    # -------------------------------------------------------------------
    # Some initialization (majority of variables have been defined as parameters
    # to the function)

    reportxy_name = 'SandwichXYData'
    reportxy_path = 'C:/SandwichFolder/'

    steel_density = 7800
    steel_youngs_modulus = 200E9
    steel_poissons_ratio = 0.29

    aluminum_density = 2770
    aluminum_youngs_modulus = 73.1E9
    aluminum_poissons_ratio = 0.33


    if top_layer_material_name == "Aluminum":
        top_layer_material_mass_density=aluminum_density
        top_layer_material_youngs_modulus=aluminum_youngs_modulus
        top_layer_material_poissons_ratio=aluminum_poissons_ratio
    else:
        top_layer_material_mass_density=steel_density
    top_layer_material_youngs_modulus=steel_youngs_modulus
    top_layer_material_poissons_ratio=steel_poissons_ratio

if core_layer_material_name == "Aluminum":
    core_layer_material_mass_density=aluminum_density
    core_layer_material_youngs_modulus=aluminum_youngs_modulus
    core_layer_material_poissons_ratio=aluminum_poissons_ratio
else:
    core_layer_material_mass_density=steel_density
    core_layer_material_youngs_modulus=steel_youngs_modulus
    core_layer_material_poissons_ratio=steel_poissons_ratio

if bottom_layer_material_name == "Aluminum":
    bottom_layer_material_mass_density=aluminum_density
    bottom_layer_material_youngs_modulus=aluminum_youngs_modulus
    bottom_layer_material_poissons_ratio=aluminum_poissons_ratio
else:
    bottom_layer_material_mass_density=steel_density
    bottom_layer_material_youngs_modulus=steel_youngs_modulus
    bottom_layer_material_poissons_ratio=steel_poissons_ratio



# ----------------------------------------------------------------------
# Initial calculations

# We will draw the cells in the core by making square cutouts
core_layer_cell_cutout_length = (sandwich_length - \
       ((no_of_core_cells + 1)*wall_thickness_core_cells)) / no_of_core_cells

# Point used to find the top surface of the core
# We will be using the findAt command to find the top surface of the core
# As an argument we need to pass the coordinates of a point on this surface
# For an even number of cells, the exact center point of the top surface of
# the core can be used. However for an odd number of cells, this point will
# lie over one of the holes
# In that case we'll pick a point offset a little way from it

if no_of_core_cells % 2 == 0:
    coreLayer_top_face_point = (sandwich_width/2, core_layer_thickness,
                                                 sandwich_length/2)
    coreLayer_bottom_face_point = (sandwich_width/2, 0.0, sandwich_length/2)
    coreLayer_inside_coord = (sandwich_width/2, core_layer_thickness/2,
                                               sandwich_length/2)
else:
    coreLayer_top_face_point = (sandwich_width/2, core_layer_thickness,
                            sandwich_length/2 + \
                                    core_layer_cell_cutout_length/2 + \
                                    wall_thickness_core_cells/2)
    coreLayer_bottom_face_point = (sandwich_width/2, 0.0,
                               sandwich_length/2 + \
                                    core_layer_cell_cutout_length/2 + \
                                                 wall_thickness_core_cells/2)
    coreLayer_inside_coord = (sandwich_width/2, core_layer_thickness/2,
                              sandwich_length/2 + \
                                       core_layer_cell_cutout_length/2 + \
                                       wall_thickness_core_cells/2)

# -----------------------------------------------------------------------
# Abaqus statements start here

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# -----------------------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Sandwich Structure')
sandwichModel = mdb.models['Sandwich Structure']


# -----------------------------------------------------------------------
# Create the parts

import sketch
import part

# a) Top layer

# i) Sketch the top layer cross section using rectangle tool
topLayerProfileSketch = sandwichModel \
                               .ConstrainedSketch(name='Top Layer Sketch',
                                                   sheetSize=40)
topLayerProfileSketch.rectangle(point1=(0.0,0.0),
                               point2=(sandwich_width,top_layer_thickness))

# ii) Create a 3D deformable part named "Top Layer" by extruding the sketch
topLayerPart=sandwichModel.Part(name='Top Layer', dimensionality=THREE_D,
                                                  type=DEFORMABLE_BODY)
topLayerPart.BaseSolidExtrude(sketch=topLayerProfileSketch,
                              depth=sandwich_length)


# b) Bottom layer

# i) Sketch the bottom layer cross section using rectangle tool
bottomLayerProfileSketch = sandwichModel \
                              .ConstrainedSketch(name='Bottom Layer Sketch',
                                                  sheetSize=40)
bottomLayerProfileSketch.rectangle(point1=(0.0,0.0),
                              point2=(sandwich_width,bottom_layer_thickness))

# ii) Create a 3D deformable part named "Bottom Layer" by extruding the
#     sketch
bottomLayerPart=sandwichModel.Part(name='Bottom Layer',
                                    dimensionality=THREE_D,
                                    type=DEFORMABLE_BODY)
bottomLayerPart.BaseSolidExtrude(sketch=bottomLayerProfileSketch,
                                  depth=sandwich_length)

# c) Core layer

# i) Sketch the core layer cross section using rectangle tool
coreLayerProfileSketch = sandwichModel \
                    .ConstrainedSketch(name='Core Layer Sketch', sheetSize=40)
coreLayerProfileSketch.rectangle(point1=(0.0,0.0),
                                  point2=(sandwich_width,core_layer_thickness))

# ii) Create a 3D deformable part named "Core Layer" by extruding the sketch
coreLayerPart=sandwichModel.Part(name='Core Layer', dimensionality=THREE_D,
                                                    type=DEFORMABLE_BODY)
coreLayerPart.BaseSolidExtrude(sketch=coreLayerProfileSketch,
                                depth=sandwich_length)

# iii) Cut out square holes to create the core
# We need to sketch out the squares onto the surface
# In order to enter the sketcher we need to select the surface and an axis
# that will appear vertical and left (or any orientation we choose)

# Select the top surface of core layer block
sketch_core_top_face_point = (sandwich_width/2, core_layer_thickness,
                                                sandwich_length/2)
sketch_core_top_face = coreLayerPart.faces.findAt(sketch_core_top_face_point,)

# Select the left edge of core layer block
sketch_core_left_edge_point = (sandwich_width/2, core_layer_thickness, 0)
sketch_core_left_edge = coreLayerPart.edges \
                                        .findAt(sketch_core_left_edge_point,)

sketch_transform = coreLayerPart \
                      .MakeSketchTransform(sketchPlane=sketch_core_top_face,
                                           sketchUpEdge=sketch_core_left_edge,
                                           sketchPlaneSide=SIDE1,
                                           sketchOrientation=LEFT,
                                           origin=(0.0,0.0,0.0))

coreLayerCutoutSketch = sandwichModel \
                          .ConstrainedSketch(name='core layer cutout sketch',
                                             sheetSize = 50,
                                             transform=sketch_transform)

rect_pt_1_y = wall_thickness_core_cells
rect_pt_2_y = sandwich_width - wall_thickness_core_cells

rect_pt_1_x = wall_thickness_core_cells

for i in range(0,no_of_core_cells):
        rect_pt_2_x =rect_pt_1_x + core_layer_cell_cutout_length
        coreLayerCutoutSketch.rectangle(point1=(rect_pt_1_x,rect_pt_1_y),
                                        point2=(rect_pt_2_x,rect_pt_2_y))
        rect_pt_1_x = rect_pt_2_x + wall_thickness_core_cells

    coreLayerPart.CutExtrude(sketchPlane=sketch_core_top_face,
                             sketchUpEdge=sketch_core_left_edge,
                             sketchPlaneSide=SIDE1,
                             sketchOrientation=LEFT,
                             sketch=coreLayerCutoutSketch,
                             flipExtrudeDirection=OFF)


    # -------------------------------------------------------------------
    # Create material

    import material

    # Create materials for the top, bottom and core plate by assigning mass
    # density, youngs modulus and poissons ratio
    topPlateMaterial = sandwichModel.Material(name='Top Plate Material')
    topPlateMaterial.Density(table=((top_layer_material_mass_density, ), ))
    topPlateMaterial.Elastic(table=((top_layer_material_youngs_modulus,
                                     top_layer_material_poissons_ratio), ))

    coreMaterial = sandwichModel.Material(name='Core Material')
    coreMaterial.Density(table=((core_layer_material_mass_density, ), ))
    coreMaterial.Elastic(table=((core_layer_material_youngs_modulus,
                                 core_layer_material_poissons_ratio), ))

    bottomPlateMaterial = sandwichModel.Material(name='Bottom Plate Material')
    bottomPlateMaterial.Density(table=((bottom_layer_material_mass_density, ), ))
    bottomPlateMaterial.Elastic(table=((bottom_layer_material_youngs_modulus,
                                        bottom_layer_material_poissons_ratio), ))


    # -------------------------------------------------------------------
    # Create solid section and assign the beam to it

    import section

    # Create a section to assign to the top layer
    topLayerSection = sandwichModel \
                      .HomogeneousSolidSection(name='Top Layer Section',
                                               material='Top Plate Material')

    # Assign the top layer to this section
    top_layer_region = (topLayerPart.cells,)
    topLayerPart.SectionAssignment(region=top_layer_region,
                                   sectionName='Top Layer Section')
```

```
# Create a section to assign to the bottom layer
bottomLayerSection = sandwichModel \
                        .HomogeneousSolidSection(name='Bottom Layer Section',
                                                  material='Bottom Plate Material')

# Assign the bottom layer to this section
bottom_layer_region = (bottomLayerPart.cells,)
bottomLayerPart.SectionAssignment(region=bottom_layer_region,
                                  sectionName='Bottom Layer Section')

# Create a section to assign to the core layer
coreLayerSection = sandwichModel \
                        .HomogeneousSolidSection(name='Core Layer Section',
                                                  material='Core Material')

# Assign the core layer to this section
core_layer_region = (coreLayerPart.cells,)
coreLayerPart.SectionAssignment(region=core_layer_region,
                                sectionName='Core Layer Section')


# --------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instances
sandwichAssembly = sandwichModel.rootAssembly
topLayerInstance = sandwichAssembly.Instance(name='Top Layer Instance',
                                             part=topLayerPart, dependent=ON)
bottomLayerInstance = sandwichAssembly.Instance(name='Bottom Layer Instance',
                                                part=bottomLayerPart,
                                                dependent=ON)
coreLayerInstance = sandwichAssembly.Instance(name='Core Layer Instance',
                                              part=coreLayerPart,
                                              dependent=ON)

# ++++++++++++++++++++++++++++++++++++++
# Identify all the faces used to constrain the assembly

topLayer_front_face_point = (sandwich_width/2, top_layer_thickness/2, 0.0)
topLayer_front_face = topLayerInstance.faces \
                                        .findAt(topLayer_front_face_point,)

topLayer_side_face_point = (0.0, top_layer_thickness/2, sandwich_length/2)
topLayer_side_face = topLayerInstance.faces.findAt(topLayer_side_face_point,)

topLayer_bottom_face_point = (sandwich_width/2, 0.0, sandwich_length/2)
topLayer_bottom_face = topLayerInstance.faces \
                                        .findAt(topLayer_bottom_face_point,)

bottomLayer_front_face_point = (sandwich_width/2, bottom_layer_thickness/2,
                                                                        0.0)
    bottomLayer_front_face = bottomLayerInstance.faces \
                                        .findAt(bottomLayer_front_face_point,)

    bottomLayer_side_face_point = (0.0, bottom_layer_thickness/2,
                                                   sandwich_length/2)
    bottomLayer_side_face = bottomLayerInstance.faces \
                                        .findAt(bottomLayer_side_face_point,)

    bottomLayer_top_face_point = (sandwich_width/2,bottom_layer_thickness,
                                                          sandwich_length/2)
    bottomLayer_top_face = bottomLayerInstance.faces \
                                        .findAt(bottomLayer_top_face_point,)

    coreLayer_front_face_point = (sandwich_width/2, core_layer_thickness/2, 0.0)
    coreLayer_front_face = coreLayerInstance.faces \
                                        .findAt(coreLayer_front_face_point,)

    coreLayer_side_face_point = (0.0, core_layer_thickness/2, sandwich_length/2)
    coreLayer_side_face = coreLayerInstance.faces \
                                        .findAt(coreLayer_side_face_point,)

    coreLayer_top_face = coreLayerInstance.faces.findAt(coreLayer_top_face_point,)

    coreLayer_bottom_face = coreLayerInstance.faces \
                                        .findAt(coreLayer_bottom_face_point,)

    # ++++++++++++++++++++++++++++++++++++++++
    # Identify all the faces used for boundary conditions

    topLayer_fix_front_face_point = (sandwich_width/2, top_layer_thickness/2, 0.0)
    topLayer_fix_front_face = topLayerInstance.faces \
                                   .findAt((topLayer_fix_front_face_point,))
    topLayer_fix_front_region = regionToolset \
                                   .Region(faces=(topLayer_fix_front_face))

    bottomLayer_fix_front_face_point = (sandwich_width/2,
                                        bottom_layer_thickness/2,
                                        0.0)
    bottomLayer_fix_front_face = bottomLayerInstance.faces \
                                 .findAt((bottomLayer_fix_front_face_point,))
    bottomLayer_fix_front_region = regionToolset \
                                 .Region(faces=(bottomLayer_fix_front_face))

    coreLayer_fix_front_face_point = (sandwich_width/2, core_layer_thickness/2,
                                                                        0.0)
    coreLayer_fix_front_face = coreLayerInstance.faces \
                                  .findAt((coreLayer_fix_front_face_point,))
    coreLayer_fix_front_region = regionToolset \
                                  .Region(faces=(coreLayer_fix_front_face))

    # ++++++++++++++++++++++++++++++++++++++++
# Identify all the faces used for loads

topLayer_load_top_face_point = (sandwich_width/2, top_layer_thickness,
                                                    sandwich_length/2)
topLayer_load_top_face = topLayerInstance.faces \
                               .findAt((topLayer_load_top_face_point,))

# ++++++++++++++++++++++++++++++++++++++
# Create a set to measure displacement history output

# For the first history point we use one of the upper corners of the core \
# layer inside one of its holes
vertex_coords_for_displacement_1 = (wall_thickness_core_cells, 0.0,
                                    sandwich_length-wall_thickness_core_cells)
vertex_for_displacement_1 = coreLayerInstance.vertices \
                                 .findAt((vertex_coords_for_displacement_1,))
sandwichAssembly.Set(vertices=vertex_for_displacement_1,
                     name='displacement point set 1')

# For the second history point we use one of the lower corners at the end of
# the bottom layer
vertex_coords_for_displacement_2 = (sandwich_width, 0.0, sandwich_length)
vertex_for_displacement_2 = bottomLayerInstance.vertices \
                                 .findAt((vertex_coords_for_displacement_2,))
sandwichAssembly.Set(vertices=vertex_for_displacement_2,
                         name='displacement point set 2')


# ++++++++++++++++++++++++++++++++++++++
# Identify faces used to define Surfaces in the assembly. These will later be
# used for tie constraints.

topLayer_bottom_surface_point = (sandwich_width/2, 0.0, sandwich_length/2)
topLayer_bottom_surface = topLayerInstance.faces \
                                    .findAt((topLayer_bottom_surface_point,))

bottomLayer_top_surface_point = (sandwich_width/2, bottom_layer_thickness,
                                                        sandwich_length/2)
bottomLayer_top_surface = bottomLayerInstance.faces \
                                    .findAt((bottomLayer_top_surface_point,))

coreLayer_top_surface_point = coreLayer_top_face_point
coreLayer_top_surface = coreLayerInstance.faces \
                                      .findAt((coreLayer_top_surface_point,))

coreLayer_bottom_surface_point = coreLayer_bottom_face_point
coreLayer_bottom_surface = coreLayerInstance.faces \
                                   .findAt((coreLayer_bottom_surface_point,))


# ++++++++++++++++++++++++++++++++++++++
# Assemble the parts using face to face relationships
# Establish face to face relationships between top layer and core layer
sandwichAssembly.FaceToFace(movablePlane=topLayer_front_face,
                            fixedPlane=coreLayer_front_face,
                            flip=OFF, clearance=0.0)
sandwichAssembly.FaceToFace(movablePlane=topLayer_side_face,
                            fixedPlane=coreLayer_side_face,
                            flip=OFF, clearance=0.0)
sandwichAssembly.FaceToFace(movablePlane=topLayer_bottom_face,
                            fixedPlane=coreLayer_top_face,
                            flip=ON, clearance=0.0)

# establish face to face relationships between bottom layer and core layer
sandwichAssembly.FaceToFace(movablePlane=bottomLayer_front_face,
                            fixedPlane=coreLayer_front_face,
                            flip=OFF, clearance=0.0)
sandwichAssembly.FaceToFace(movablePlane=bottomLayer_side_face,
                            fixedPlane=coreLayer_side_face,
                            flip=OFF, clearance=0.0)
sandwichAssembly.FaceToFace(movablePlane=bottomLayer_top_face,
                            fixedPlane=coreLayer_bottom_face,
                            flip=ON, clearance=0.0)


# -------------------------------------------------------------------
# Create the steps

import step

# Create the loading step
sandwichModel.StaticStep(name='Apply Load', previous='Initial',
   description='Apply the pressure load on top surface of sandwich structure')

# -------------------------------------------------------------------
# Create the field output request
# Leave at defaults

# -------------------------------------------------------------------
# Create the history output request

displacement_point_region_1 = \
                            sandwichAssembly.sets['displacement point set 1']
sandwichModel.historyOutputRequests.changeKey(fromName='H-Output-1',
                                         toName='Displacement output 1')
sandwichModel.historyOutputRequests['Displacement output 1'] \
                            .setValues(variables=('UT',),
                                       frequency=1,
                                       region=displacement_point_region_1,
                                       sectionPoints=DEFAULT, rebar=EXCLUDE)

displacement_point_region_2 = sandwichAssembly \
                                        .sets['displacement point set 2']
```

```
sandwichModel.HistoryOutputRequest(name='Displacement output 2',
                                   createStepName='Apply Load',
                                   variables=('UT',),
                                   region=displacement_point_region_2,
                                   sectionPoints=DEFAULT, rebar=EXCLUDE)

# -------------------------------------------------------------------
# Apply boundary conditions

sandwichModel.EncastreBC(name='Fix Top Layer Front',
                         createStepName='Apply Load',
                         region=topLayer_fix_front_region)
sandwichModel.EncastreBC(name='Fix Bottom Layer Front',
                         createStepName='Apply Load',
                         region=bottomLayer_fix_front_region)
sandwichModel.EncastreBC(name='Fix Core Layer Front',
                         createStepName='Apply Load',
                         region=coreLayer_fix_front_region)

# -------------------------------------------------------------------
# Apply loads

# We extract the region of the face choosing which direction its normal
# points in
topLayer_top_face_region= \
                      regionToolset.Region(side1Faces=topLayer_load_top_face)

# Apply the pressure load on this region in the 'Apply Load' step
sandwichModel.Pressure(name='Uniform Applied Pressure',
                       createStepName='Apply Load',
                       region=topLayer_top_face_region,
                       distributionType=UNIFORM, magnitude=10,
                       amplitude=UNSET)


# -------------------------------------------------------------------
# Define surfaces to use in tie constraints

sandwichAssembly.Surface(side1Faces=topLayer_bottom_surface,
                         name='Top Layer Bottom')
sandwichAssembly.Surface(side1Faces=bottomLayer_top_surface,
                         name='Bottom Layer Top')
sandwichAssembly.Surface(side1Faces=coreLayer_bottom_surface,
                         name='Core Layer Bottom')
sandwichAssembly.Surface(side1Faces=coreLayer_top_surface,
                         name='Core Layer Top')

# -------------------------------------------------------------------
# Create tie constraints

import interaction
```

```
region1=sandwichAssembly.surfaces['Core Layer Top']
region2=sandwichAssembly.surfaces['Top Layer Bottom']

sandwichModel.Tie(name='Constraint-1', master=region1, slave=region2,
                  positionToleranceMethod=COMPUTED, adjust=ON,
                  tieRotations=ON, constraintEnforcement=SURFACE_TO_SURFACE,
                  thickness=ON)

region1=sandwichAssembly.surfaces['Core Layer Bottom']
region2=sandwichAssembly.surfaces['Bottom Layer Top']

sandwichModel.Tie(name='Constraint-2', master=region1, slave=region2,
                  positionToleranceMethod=COMPUTED, adjust=ON,
                  tieRotations=ON, constraintEnforcement=SURFACE_TO_SURFACE,
                  thickness=ON)

# ---------------------------------------------------------------------
# Create the mesh

import mesh

# +++++++++++++++++++++++++++++++++++++++
# Mesh the top layer
# We place a point somewhere inside it based on our knowledge of the geometry
topLayer_inside_coord=(sandwich_width/2,top_layer_thickness/2,
                                          sandwich_length/2)

elemType1 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN,
                          secondOrderAccuracy=OFF, hourglassControl=DEFAULT,
                          distortionControl=DEFAULT)

topLayerCells=topLayerPart.cells
selectedTopLayerCells=topLayerCells.findAt(topLayer_inside_coord,)
topLayerMeshRegion=(selectedTopLayerCells,)
topLayerPart.setElementType(regions=topLayerMeshRegion,
                            elemTypes=(elemType1,))

topLayerPart.seedPart(size=0.04, deviationFactor=0.1)

topLayerPart.generateMesh()

# +++++++++++++++++++++++++++++++++++++++
# Mesh the bottom layer
# We place a point somewhere inside it based on our knowledge of the geometry
bottomLayer_inside_coord=(sandwich_width/2, bottom_layer_thickness/2,
                                              sandwich_length/2)

elemType2 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN,
                          secondOrderAccuracy=OFF,
                          hourglassControl=DEFAULT, distortionControl=DEFAULT)
```

```
bottomLayerCells=bottomLayerPart.cells
selectedBottomLayerCells=bottomLayerCells.findAt(bottomLayer_inside_coord,)
bottomLayerMeshRegion=(selectedBottomLayerCells,)
bottomLayerPart.setElementType(regions=bottomLayerMeshRegion,
                               elemTypes=(elemType2,))

bottomLayerPart.seedPart(size=0.04, deviationFactor=0.1)

bottomLayerPart.generateMesh()

# ++++++++++++++++++++++++++++++++++++++
# Mesh the core layer
# We place a point somewhere inside it based on our knowledge of the geometry
# This point has already been defined in the initial calculations section as
# coreLayer_inside_coord

elemType3 = mesh.ElemType(elemCode=C3D8R, elemLibrary=STANDARD,
                          kinematicSplit=AVERAGE_STRAIN,
                          secondOrderAccuracy=OFF, hourglassControl=DEFAULT,
                          distortionControl=DEFAULT)

coreLayerCells=coreLayerPart.cells
selectedCoreLayerCells=coreLayerCells.findAt(coreLayer_inside_coord,)
coreLayerMeshRegion=(selectedCoreLayerCells,)
coreLayerPart.setElementType(regions=coreLayerMeshRegion,
                             elemTypes=(elemType3,))

coreLayerPart.seedPart(size=0.04, deviationFactor=0.1)

coreLayerPart.generateMesh()


# -------------------------------------------------------------------
# Create and run the job

import job

# Create the job
mdb.Job(name=job_name, model='Sandwich Structure', type=ANALYSIS,
        explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE,
        description='Run the contact simulation',
        parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
        numDomains=1, userSubroutine='', numCpus=1, memory=50,
        memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
        contactPrint=OFF, historyPrint=OFF)

# Run the job
mdb.jobs[job_name].submit(consistencyChecking=OFF)

# Do not return control till job is finished running
mdb.jobs[job_name].waitForCompletion()
```

```
# End of run job
# ---------------------------------------------------------------------


# =====================================================================
# ---------------------------------------------------------------------
# Post processing
# ---------------------------------------------------------------------
# =====================================================================

if write_and_print:

    # ---------------------------------------------------------------
    # Send XY Data for U3 displacement of bottom and top points to an output
    # file

    import odbAccess
    import visualization

    sandwich_odb_path = job_name + '.odb'
    sandwich_odb_object = session.openOdb(name=sandwich_odb_path)

    # The main session viewport must be set to the odb object using the
    # following line. If not you might receive an error message that states
    # "The current viewport is not associated with an output database file.
    # Request operation cancelled."
    session.viewports['Viewport: 1'] \
                              .setValues(displayedObject=sandwich_odb_object)


    keyarray=session.odbData[sandwich_odb_path].historyVariables.keys()

    theoutputvariablename=[]

    for x in keyarray:
        if (x.find('U2')>-1):
            theoutputvariablename.append(x)

    # You may enter an entire path if you wish to have the report stored in
    # a particular location.
    # One way to do it is using the following syntax.

    reportxy_name_and_path = reportxy_path + reportxy_name + '.txt'

    # Note however that the folder 'MyNewFolder' must exist otherwise you
    # will likely get the following error
    # "IOError:C/MyNewFolder: Directory not found"
    # You must either create the folder in Windows before running the script
    # Or if you wish to create it using Python commands you must use the
    # os.makedir() or os.makedirs() function
    # os.makedirs() is preferable because you can create multiple nested
# directories in one statent if you wish
# Note that this function returns an exception if the directory already
# exists hence it is a good idea to use a try block

try:
    os.makedirs(reportxy_path)
except:
    print "Directory exists hence no need to recreate it. " + \
          "Move on to next statement"


session.XYDataFromHistory(name='sandwichXYData-1',
                          odb=sandwich_odb_object,
                          outputVariableName=theoutputvariablename[0])
sandwich_xydata_object = session.xyDataObjects['sandwichXYData-1']
session.xyReportOptions.setValues(totals=ON, minMax=ON)
session.writeXYReport(fileName=reportxy_name_and_path,
                      xyData=(sandwich_xydata_object, ),
                      appendMode=OFF)

session.XYDataFromHistory(name='sandwichXYData-2',
                          odb=sandwich_odb_object,
                          outputVariableName=theoutputvariablename[1])
sandwich_xydata_object = session.xyDataObjects['sandwichXYData-2']
session.xyReportOptions.setValues(totals=ON, minMax=ON)
session.writeXYReport(fileName=reportxy_name_and_path,
                      xyData=(sandwich_xydata_object, ),
                      appendMode=ON)

sandwich_odb_object.close()

# Read the displacement from the report

extracted_line = ''
# Need a boolean variable to state whether we are reading the correct
# section of the file
file_xydata_section = 0

f=open(reportxy_name_and_path)
for line in f:
    str=line
    if 'sandwichXYData-2' in str:
        file_xydata_section = 1
    if 'MINIMUM' in str and file_xydata_section == 1:
        extracted_line = str
        extracted_list = extracted_line.split()
        max_displacement = extracted_list[2]
        print "The displacement of the node at end of beam is " + \
                                    repr(-1 * float(max_displacement))

f.close()
```

19.6 Examining the Script

This script is a modified version of the parameterized sandwich structure script from Chapter 15. Since it was already parameterized it is easy to link the user entered values to variables that affect the script. We shall only consider the newer bits here.

19.6.1 Function definition

All of the processing will be done by our function **createSandwichStructure()**

```
def createSandwichStructure(sandwich_length, sandwich_width, top_layer_thickness,
                            core_layer_thickness, bottom_layer_thickness,
                            no_of_core_cells, wall_thickness_core_cells,
                            top_layer_material_name, bottom_layer_material_name,
                            core_layer_material_name, job_name, write_and_print):
```

The function is called by the RSG dialog box when the **OK** button is pressed. All the keywords assigned to widgets are passed to this function along with the data values entered by the users, whether they are Strings, Floats, Integers or Boolean.

Notice that many of the variable assignment statements that were present at the top of the script in parameterized sandwich structure example are now missing since all their values are obtained as parameters when **createSandwichStructure()** is called.

19.6.2 Material variable assignments

This block deals with assigning material properties to variables.

```
    steel_density = 7800
    steel_youngs_modulus = 200E9
    steel_poissons_ratio = 0.29

    aluminum_density = 2770
    aluminum_youngs_modulus = 73.1E9
    aluminum_poissons_ratio = 0.33


    if top_layer_material_name == "Aluminum":
        top_layer_material_mass_density=aluminum_density
        top_layer_material_youngs_modulus=aluminum_youngs_modulus
        top_layer_material_poissons_ratio=aluminum_poissons_ratio
    else:
        top_layer_material_mass_density=steel_density
        top_layer_material_youngs_modulus=steel_youngs_modulus
        top_layer_material_poissons_ratio=steel_poissons_ratio
    if core_layer_material_name == "Aluminum":
        core_layer_material_mass_density=aluminum_density
        core_layer_material_youngs_modulus=aluminum_youngs_modulus
        core_layer_material_poissons_ratio=aluminum_poissons_ratio
    else:
        core_layer_material_mass_density=steel_density
        core_layer_material_youngs_modulus=steel_youngs_modulus
        core_layer_material_poissons_ratio=steel_poissons_ratio

    if bottom_layer_material_name == "Aluminum":
        bottom_layer_material_mass_density=aluminum_density
        bottom_layer_material_youngs_modulus=aluminum_youngs_modulus
        bottom_layer_material_poissons_ratio=aluminum_poissons_ratio
    else:
        bottom_layer_material_mass_density=steel_density
        bottom_layer_material_youngs_modulus=steel_youngs_modulus
        bottom_layer_material_poissons_ratio=steel_poissons_ratio
```

Recall that the user selects the material of the top layer using a combo box with the options 'Steel' and 'Aluminum' and the keyword **top_layer_material_name**. The core layer is selected using one of the radio buttons both of which share the same keyword **core_layer_material_name**. The bottom layer is selected using a list with the keyword **bottom_layer_material_name**. In all 3 cases, the variables will either have the String "Aluminum" or "Steel" stored in them depending on the user's choice. The above statements then proceed to assign the appropriate material properties to the layers (or to variables that will later be assigned to the layers).

19.6.3 Create the materials

Here we create the materials themselves using the property variables created earlier.

```
    # -------------------------------------------------------------------
    # Create material

    import material

    # Create materials for the top, bottom and core plate by assigning mass
    # density, youngs modulus and poissons ratio
    topPlateMaterial = sandwichModel.Material(name='Top Plate Material')
    topPlateMaterial.Density(table=((top_layer_material_mass_density, ), ))
    topPlateMaterial.Elastic(table=((top_layer_material_youngs_modulus,
                                     top_layer_material_poissons_ratio), ))

    coreMaterial = sandwichModel.Material(name='Core Material')
    coreMaterial.Density(table=((core_layer_material_mass_density, ), ))
    coreMaterial.Elastic(table=((core_layer_material_youngs_modulus,
                                 core_layer_material_poissons_ratio), ))
    bottomPlateMaterial = sandwichModel.Material(name='Bottom Plate Material')
    bottomPlateMaterial.Density(table=((bottom_layer_material_mass_density, ), ))
    bottomPlateMaterial.Elastic(table=((bottom_layer_material_youngs_modulus,
                                        bottom_layer_material_poissons_ratio), ))
        bottomPlateMaterial.Elastic(table=((bottom_layer_material_youngs_modulus,
bottom_layer_material_poissons_ratio), ))
```

The block of code that creates the materials has been modified to work with the changes to the material variable names, and the fact that each part can be assigned a material independent of the others.

19.6.4 Create the sections

This block creates the sections.

```
    # -----------------------------------------------------------------
    # Create solid section and assign the beam to it

    import section

    # Create a section to assign to the top layer
    topLayerSection = sandwichModel \
                         .HomogeneousSolidSection(name='Top Layer Section',
                                                  material='Top Plate Material')

    # Assign the top layer to this section
    top_layer_region = (topLayerPart.cells,)
    topLayerPart.SectionAssignment(region=top_layer_region,
                                   sectionName='Top Layer Section')

    # Create a section to assign to the bottom layer
    bottomLayerSection = sandwichModel \
                         .HomogeneousSolidSection(name='Bottom Layer Section',
                                                  material='Bottom Plate Material')

    # Assign the bottom layer to this section
    bottom_layer_region = (bottomLayerPart.cells,)
    bottomLayerPart.SectionAssignment(region=bottom_layer_region,
                                      sectionName='Bottom Layer Section')

    # Create a section to assign to the core layer
    coreLayerSection = sandwichModel \
                         .HomogeneousSolidSection(name='Core Layer Section',
                                                  material='Core Material')

    # Assign the core layer to this section
    core_layer_region = (coreLayerPart.cells,)
    coreLayerPart.SectionAssignment(region=core_layer_region,
                                   sectionName='Core Layer Section')
```

We have only modified this block to point to use the new material variables.

19.6.5 To write (or not write) XY report and print displacement

This block looks to see if the checkbox has been checked or not, and if it has it writes an XY report and also prints out the displacement.

```
    # ====================================================================
    # --------------------------------------------------------------------
    # Post processing
    # --------------------------------------------------------------------
    # ====================================================================

    if write_and_print:

        # ----------------------------------------------------------------
        # Send XY Data for U3 displacement of bottom and top points to an output
        # file

        import odbAccess
        import visualization

        sandwich_odb_path = job_name + '.odb'
        sandwich_odb_object = session.openOdb(name=sandwich_odb_path)
                ...
                ...
                ...
                ...
```

The variable **write_and_print** contains a Boolean value indicating whether the checkbox "Write Report and Print Displacement" in the RSG dialog box was checked or not. The **if** condition decides whether or not to run the post processing script based on this.

19.7 Summary

In this chapter, you discovered that the RSG is, as its name suggests, "really simple". You can rapidly create a dialog box with useful widgets, and hook it up to a kernel script. This script needs to have a function that accepts the data from the widgets as inputs. The RSG is suitable for a simple GUI interfaces, and the fact that it gets stored as a Plug-in makes it accessible within all instances of Abaqus/CAE.

20

Create a Custom GUI Application Template

20.1 Introduction

GUI Customization allows Abaqus users to modify or customize the Abaqus/CAE Interface. The analyst can change the look and feel of Abaqus/CAE to a great extent, creating his own modules, menus, toolbars, tool buttons and dialog boxes. He can also remove existing Abaqus/CAE modules and toolsets.

This technology has many uses. Think of a company or research institute that, for the most part, runs a handful of analyses on a regular basis with minor changes to these. A vertical application can be built with much of the repetitive tasks automated with scripts, giving the analyst the ability to make only certain allowed changes, and automating the rest of the process. This type of automation of in-house processes is of great use to some organizations.

This may be compounded by the fact that a lot of the personnel working on a project are not very proficient at using Abaqus, but need to harness its functionality and run simulations within a narrow framework. An application can be created which guides them through the process step by step, prompting them for inputs and hiding most of complexity of the Abaqus interface from them.

GUI Customization does not require an entire automated application to be built, it can be used to create plug-ins which accomplish a single specific task and have a well designed interactive interface suited to this.

You need to understand the fundamentals of Abaqus GUI development before we attempt to write a script. It is important that you read the following sections and understand them before we get into our GUI example.

20.2 What is the Abaqus GUI Toolkit

Abaqus extends the functionality of a 3rd party open source GUI toolkit called the FOX toolkit. FOX is a cross platform C++ based toolkit for creating GUIs. If you wish to learn more about this toolkit you can visit their website at http://www.fox-toolkit.org/.

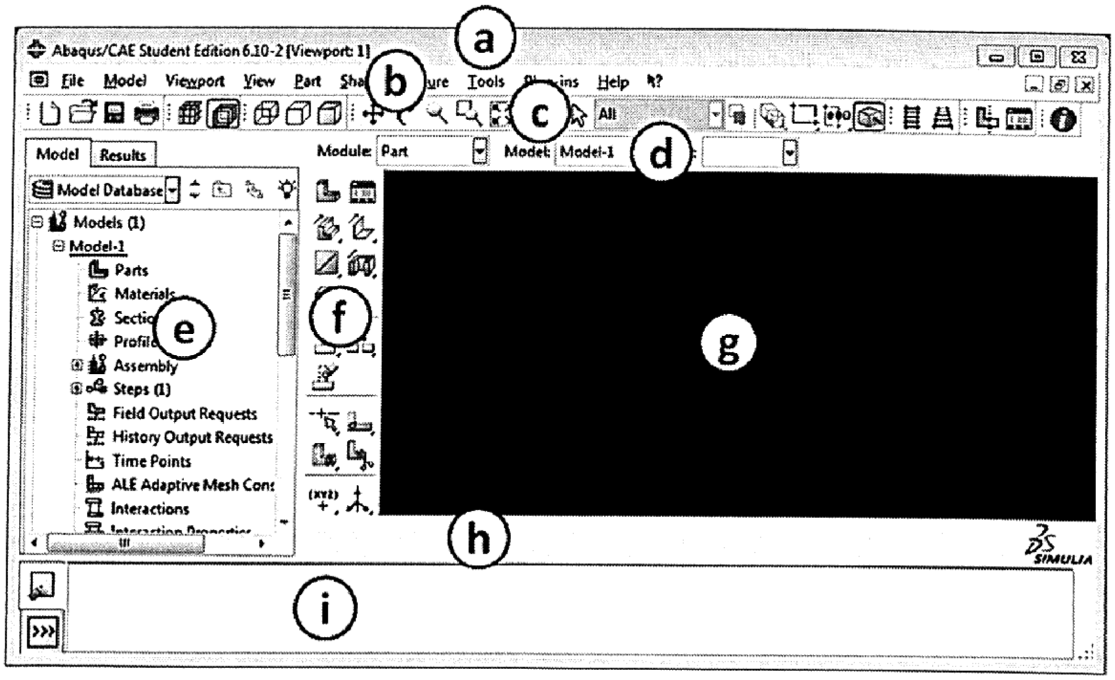
Abaqus provides a Python interface to the Abaqus/CAE C++ GUI toolkit. This interface, or toolkit, is called the Abaqus GUI Toolkit.

20.3 Components of a GUI Application

In order to design an Abaqus GUI Application it is very important that you understand the GUI infrastructure - the components that constitute the GUI, and how they work together.

1. The top most component is the application object itself. This is an object of type **AFXApp** which you will learn more about in a little bit.
2. The application consists of a window with the GUI infrastructure. All custom Abaqus applications have this basic look. The window consists of
   a) a title bar,
   b) a menu bar,
   c) one or more toolbars,
   d) a context bar which consists of the module control and context controls
   e) a tree area which displays the model tree or output database tree
   f) a module toolbox with tool buttons
   g) a canvas area where the parts, assemblies, renderings and so on are displayed
   h) a prompt area below the window
   i) and a message area (which can be switched with the command line interface)

   These are marked in the figure. The main window itself is an object of type **AFXMainWindow**.



3. Within the main window you have modules and toolsets. Modules are clearly marked in Abaqus/CAE with the word "Module:" and a combo box (drop down menu) listing the different modules such as Part, Property, Assembly, Step, Visualization and so on. This combo box is visible in the context bar (d) in the figure. Modules are of type **AFXModuleGui**. Toolsets on the other hand are the buttons displayed right next to the canvas in the same area as module toolboxes (f). However they are different from module toolboxes in that module toolboxes change depending on which module you are in whereas toolsets remain there no matter which module you are in. Toolsets are of type **AFXToolsetGui**.
4. Within the modules you have menus, toolbars and module toolboxes. As you switch modules, these change. Menus have panes which are of type **AFXMenuPane**, and within these you have the menu title **AFXMenuTitle** and menu items **AFXMenuCommand**. Toolbars exist as groups of type **AFXToolbarGroup** and they are made up of toolbar buttons of type **AFXToolButton**. Toolboxes also exist as groups of type **AFXToolboxGroup** and these consist of toolbox buttons **AFXToolButton** similar to toolbars.
5. The menus, toolbar buttons and toolboxes launch modes. Modes get input from the user and issue a command. There are two types of modes – form modes and procedure modes.

Form modes create a dialog box where the user can type in inputs or select options using checkboxes, radio buttons, lists and so on. For example, when you click on **View > Part Display Options**, you see the **Part Display Options** dialog box. You can select your options here and when you click **Apply** a command is issued to the kernel. Form modes do not allow the user to pick anything in the viewport. Form modes are of type **AFXForm**.

Procedure modes on the other hand prompt users to make selections in the viewport and then use this information to execute a kernel command. So for example, if you try to define a concentrated force in the loads module, Abaqus prompts you to select the nodes on which to apply it and you pick the nodes in the viewport window. This is a procedure mode. Procedure modes can have multiple steps. They can also be used to launch dialog boxes. Procedure modes are of type **AFXProcedure**. It is also possible for menu items, toolbar buttons or toolbox buttons to launch a dialog box that is not associated with a form or procedure. This type of dialog will not communicate with the kernel, only with the GUI (more on this later). Such a dialog box will be of type **AFXDialog**.

6. Form modes launch dialog boxes of type **AFXDataDialog**. These are different from the previously mentioned **AFXDialog** because **AFXDataDialog** dialog boxes send commands to the kernel for processing. Procedure modes create objects of type **AFXPickStep** and can also launch dialog boxes of type **AFXDataDialog**.
7. Dialog boxes are made up of layout managers such as **AFXVerticalAligner** which creates a vertical layout, and many others which we shall discuss later.
8. The layout managers contain within them the widgets such as labels (**FXLabel**), text fields (**AFXTextField**), radio buttons (**FXRadioButton**) and so on.

It is important that you understand the above structure and recognize the names of the classes. Scripts written to target the Abaqus GUI Toolkit usually span multiple .py files and it can get a little confusing to keep track of what goes where if you don't fully understand the structure.

20.4 GUI and Kernel Processes

In the previous section we mentioned **AFXDialog** and **AFXDataDialog**, and briefly spoke of how one (the second one) sends commands to the kernel while the other (the first one) does not. It is important to understand that when you create a custom Abaqus GUI, you have two types of processes running simultaneously – GUI processes and kernel processes. GUI processes execute GUI commands and kernel processes execute kernel commands.

You've already seen kernel commands. All of the scripts written up until this point were kernel scripts. They interacted with the Abaqus kernel in order to set up your model, send it to the solver, and post process it. To elaborate further, only a kernel script can have a statement such as

```
mdb.Model(name=My Model, modelType=STANDARD_EXPLICIT)
```

or

```
myPart = myModel.Part(name='Plate', dimensionality=THREE_D, type = DEFORMABLE_BODY)
```

Model() and **Part()** are commands that are executed by the Abaqus kernel. Kernel scripts usually have the following import statements at the top

```
from abaqus import *
from abaqusConstants import *
```

GUI scripts on the other hand only deal with GUI processing. They create the GUI, and can issue Python commands, but not commands that target the Abaqus kernel. They usually have the import statement

```
from abaqusGui import *
```

at the top.

GUI and kernel scripts must be kept separate. You cannot have "from abaqus import *" and "from abaqusGui import *" in the same script as a script must either be purely GUI or purely kernel.

Since the GUI must eventually issue commands to the kernel, a link must be established between GUI and kernel scripts. This is usually done using a mode. For example, a form mode (**AFXForm**) launches a dialog (**AFXDialog**) which contains the GUI commands necessary to display widgets (checkboxes, text fields, labels etc), and when the **OK** button is pressed in the dialog box the form calls a command in a separate kernel script. This way the GUI and kernel scripts are kept separate and one calls the other through the use of a mode. Another method is to use **sendCommand()** method. You will see both of this demonstrated in the next chapter, but it is essential that you learn these concepts right now.

20.5 Methodology

In this example we create a basic GUI application. As such it does not execute any kernel scripts; it is just a GUI with no real functionality. However it is a complete framework, and we will be using it for the example in the next chapter. More importantly, this code framework can be reused by you in all GUI scripts you write in the future, as it serves as a stable base off which you can build.

The GUI application is created using a number of scripts. We will examine each of these scripts in turn, but first an overview so that you see the bigger picture.

- **customCaeApp.py** is the application startup script. It creates the application (**AFXApp**) and calls the main window
- **customCaeMainWindow.py** creates the main window (**AFXMainWidnow**). It registers the toolsets and modules that will be part of the application. These toolsets and modules include standard ones as well as custom ones made by us.
- **modifiedCanvasToolsetGui.py** creates a modified version of the **Viewport** menu which you see when you open Abaqus/CAE. It will adds a few new menu items to the **Viewport** menu, removes others that exist by default, adds a couple of horizontal separators in the menu pane, and changes the name of the **Viewport** menu to 'Viewport Modified'.

  When menu items or toolbar buttons are clicked in this modified viewport toolset, the form mode, defined in **demoForm.py**, is called to post the dialog box which is defined in **demoDB.py**
- **customToolboxButtonsGui.py** creates a new toolset (**AFXToolsetGui**). The toolset buttons which appear to the left of the canvas (along with module toolboxes) will be visible in all modules.

  When buttons in this toolbox are clicked, the form mode defined in **demoForm.py** is called to post the dialog box defined in **demoDB.py**
- **customModuleGui.py** creates a new module (**AFXModuleGui**) which appears in the module combobox as 'Custom Module'. This module has a menu (**AFXMenuPane**) called 'Custom Menu' associated with it, a toolbar (**AFXToolbarGroup**) called 'Arrow Toolbar' and a toolbox group (**AFXToolboxGroup**). All of these are only visible when the user is in the custom module.

When most of the menu items, toolbar buttons or toolbox buttons are clicked in this custom module, the form mode defined in **demoForm.py** is called to post the dialog box defined in **demoDB.py**. However to change things up, one of the menu items instead posts a modeless dialog defined in **demoDBwoForm.py** without calling any form mode. This is to demonstrate how you launch a modeless dialog box.

- **demoForm.py** creates a form mode (**AFXForm**) which will post the dialog created in **demoDB.py** and will issue a command when the **OK** button is clicked in that dialog.
- **demoDB.py** creates the modal dialog box (**AFXDataDialog**) that will be posted by the form mode of **demoForm.py**
- **demoDBwoForm.py** creates a modeless dialog box – one that is posted without any form.

20.6 Python Script

We shall now look at each of the script files in turn. Remember that these must all exist together in the same folder for the application to work.

20.6.1 Application Startup Script

This script is contained in **customCaeApp.py**. It is the application startup script - it creates the application (**AFXApp**) - and calls for the creation of the main window

```
# ***********************************************************************************
# Startup Script - This script creates and launches a custom Abaqus/CAE application
# ***********************************************************************************

from abaqusGui import AFXApp
import sys
# import class that will create the main window
from customCaeMainWindow import CustomCaeMainWindow

# Initialize application object
# In AFXApp, appName and vendorName are displayed if productName is set to ''
# otherwise productName is displayed.
app = AFXApp(appName='ABAQUS/CAE',
             vendorName='ABAQUS, Inc.',
             productName='Custom GUI Application',
             majorNumber=1,
             minorNumber=1,
             updateNumber=1,
             prerelease=False)
app.init(sys.argv)
```

```
# Construct main window
CustomCaeMainWindow(app)

# Create application
app.create()

# Run application
app.run()
```

This script is called a startup script. All applications made using the Abaqus GUI toolkit must be launched using a startup script. You are unlikely to make any changes to this script in your own applications except to the name of the main window script.

```
from abaqusGui import AFXApp
```

This statement imports the **AFXApp** constructor from the **abaqusGui** module. **abaqusGui** allows you to access the Abaqus GUI Toolkit.

You could instead have written

```
from abaqusGui import *
```

The statement

```
import sys
```

imports the **sys** module. This module is required to pass arguments to the **init()** function in a subsequent statement.

```
from customCaeMainWindow import CustomCaeMainWindow
```

This statement imports the main window constructor which we define in the **CustomCaeMainWindow** class in the script **customCaeMainWindow.py**

```
# Initialize application object
# In AFXApp, appName and vendorName are displayed if productName is set to ''
# otherwise productName is displayed.
app = AFXApp(appName='ABAQUS/CAE',
             vendorName='ABAQUS, Inc.',
             productName='Custom GUI Application',
             majorNumber=1,
             minorNumber=1,
             updateNumber=1,
             prerelease=False)
app.init(sys.argv)
```

AFXApp() is the constructor for creating an instance of the **AFXApp** class. Among the arguments passed to it **appName** is the application registry key, **vendorName** is the vendor registry key, **productName** is the name of the product, **majorNumber** is the version number, **minorNumber** is the release number, **updateNumber** (not used here) is the update number, and **prerelease** is a boolean indicating if it is official or prerelease. The registry keys **appName** and **vendorName** are not actually used by Abaqus in the current version (6.10), however they may provide capabilities in future versions of the software. The title of the main window is created using the remaining arguments. It will appear as "productName + "Version" + major + "." + minor + "-" + update. If **prerelease** = TRUE then "PRE" will appear before the update number. Note however that if a title is supplied to **AFXMainWindow** (in our example this would be in the file **customCaeMainWindow.py**) that title will override this one.

init() is used to initialize the application object represented by the **AFXApp** instance.

The application object, which is an instance of **AFXApp**, is responsible for updating the GUI, managing the message queue and other related tasks. Its constructor creates the data structures needed for the application to function. One application object must be created for each application.

```
CustomCaeMainWindow(app)
```

This statement creates an instance of the main window which is defined in the **CustomCaeMainWindow** class in **customCaeMainWindow.py**. It passes the **AFXApp** object to the **__init__** method of **CustomCaeMainWindow**. In plain English this statement creates the main window.

```
app.create()
```

This statement creates the application. More specifically it causes the creation of required GUI windows.

```
app.run()
```

This statement runs the application. The application is displayed and it enters an event loop. An event loop is a state in which the application waits for "events", usually actions performed by a user, and reacts accordingly.

20.6.2 Main Window

This script is contained in **customCaeMainWindow.py**. It creates the main window (**AFXMainWidnow**), and registers the toolsets and modules that will be part of the application, including the custom ones made by us.

```
# ********************************************************************************
# This script defines the main window of the custom Abaqus/CAE application
# ********************************************************************************

from abaqusGui import *
from sessionGui import *
from canvasGui import CanvasToolsetGui
from viewManipGui import ViewManipToolsetGui
from modifiedCanvasToolsetGui import ModifiedCanvasToolsetGui
from customToolboxButtonsGui import CustomToolboxButtonsGui

# Define the class
class CustomCaeMainWindow(AFXMainWindow):

    def __init__(self, app, windowTitle=''):

        AFXMainWindow.__init__(self, app, windowTitle)

        # Register toolsets
        self.registerToolset(FileToolsetGui(), GUI_IN_MENUBAR|GUI_IN_TOOLBAR)
        self.registerToolset(ModelToolsetGui(), GUI_IN_MENUBAR)
        self.registerToolset(ViewManipToolsetGui(), GUI_IN_MENUBAR|GUI_IN_TOOLBAR)

        # The following statement would normally be used to register the viewport
        # menu in its original form
        # self.registerToolset(CanvasToolsetGui(), GUI_IN_MENUBAR)
        # We will instead replace the viewport menu with a modified version of it
        self.registerToolset(ModifiedCanvasToolsetGui(), GUI_IN_MENUBAR)

        # Register the custom toolset for adding custom toolbox buttons
        self.registerToolset(CustomToolboxButtonsGui(), GUI_IN_TOOLBOX)

        # The following statement would normally be used to register the help
        # menu/toolset in its original form
        # self.registerHelpToolset(HelpToolsetGui(),
        #                          GUI_IN_MENUBAR|GUI_IN_TOOLBAR)
        # We will replace this with a custom help toolset with our own copyright
        # info hence we comment this out
        # Modify Help > About Abaqus... dialog to include custom copyright
        # information
        customHelpToolsetGui = HelpToolsetGui()
        abaqus_product_name = getAFXApp().getProductName()
        major_version_no, minor_version_no, update_no = \
                                            getAFXApp().getVersionNumbers()
        is_this_prerelease = getAFXApp().getPrerelease()
        if is_this_prerelease:
            version = '%s Version %s.%s-PRE%s' % (abaqus_product_name,
                                                  major_version_no,
                                                  minor_version_no,
                                                  update_no)
        else:
            version = '%s Version %s.%s-%s' % (abaqus_product_name,
                                               major_version_no,
                                               minor_version_no,
                                               update_no)
        custom_title = 'Custom GUI Framework that you can modify and reuse'
        custom_information = 'Python Scripts for Abaqus - Learn by Example ' + \
                             '\n Gautam Puri \n Copyright 2011' + \
                             '\n Running Abaqus ' + version
        customHelpToolsetGui.setCustomCopyrightStrings(custom_title,
                                                       custom_information)
        custom_icon=afxCreateIcon('icon_bookcover.png')
        customHelpToolsetGui.setCustomLogoIcon(custom_icon)
        self.registerHelpToolset(customHelpToolsetGui,
                                 GUI_IN_MENUBAR|GUI_IN_TOOLBAR)


        self.registerToolset(AnnotationToolsetGui(),
                             GUI_IN_MENUBAR|GUI_IN_TOOLBAR)
        self.registerToolset(DatumToolsetGui(), GUI_IN_TOOLBOX)
        self.registerToolset(EditMeshToolsetGui(), GUI_IN_TOOLBOX)
        self.registerToolset(PartitionToolsetGui(), GUI_IN_TOOLBOX)
        self.registerToolset(QueryToolsetGui(), GUI_IN_TOOLBOX)
        self.registerToolset(RepairToolsetGui(), GUI_IN_TOOLBOX)
        self.registerToolset(SelectionToolsetGui(), GUI_IN_TOOLBAR)
        self.registerToolset(TreeToolsetGui(), GUI_IN_TOOLBOX|GUI_IN_MENUBAR)


        # Register the modules. These will appear in the Modules combo box in
        # Abaqus/CAE in the order they are registered here
        self.registerModule('Part',          'Part')
        self.registerModule('Property',      'Property')
        self.registerModule('Assembly',      'Assembly')
        self.registerModule('Step',          'Step')
        self.registerModule('Interaction',   'Interaction')
        self.registerModule('Load',          'Load')
        self.registerModule('Mesh',          'Mesh')
        self.registerModule('Job',           'Job')
        self.registerModule('Visualization', 'Visualization')
        self.registerModule('Sketch',        'Sketch')
        # Register our custom module which resides in the script file
        # customModuleGui.py
        self.registerModule('Custom Module',        'customModuleGui')
```

```
from abaqusGui import *
from sessionGui import *
from canvasGui import CanvasToolsetGui
from viewManipGui import ViewManipToolsetGui
```

The script begins by importing required modules. **abaqusGUI** allows the script to work with the Abaqus GUI Toolkit. It also defines the existing modules and toolsets.

```
from modifiedCanvasToolsetGui import ModifiedCanvasToolsetGui
```

This statement imports the **ModifiedCanvasToolsetGui** class from the script **modifiedCanvasToolsetGui.py**. **ModifiedCanvasToolsetGui** contains our instructions on how the canvas toolset (which is the toolset that displays the **Viewport** menu) should be modified in our custom application. We will register it a few statements later.

```
from customToolboxButtonsGui import CustomToolboxButtonsGui
```

This statement imports the **CustomToolboxButtonsGui** class from the script **customToolboxButtonsGui.py**. **CustomToolboxButtonsGui** contains our instructions on creating custom toolbox buttons that appear just to the left of the viewport. We will register it a few statements later.

```
class CustomCaeMainWindow(AFXMainWindow):
```

Our class **CustomCaeMainWindow** is derived from **AFXMainWindow**. This must always be done when creating the main window.

```
    def __init__(self, app, windowTitle=''):
```

The **__init__** method can be considered (for the sake of simplification) to be the constructor of the class. When **CustomCaeMainWindow()** is called in **customCaeApp.py** it is this method that is executed. The **__init__** method must register toolsets and modules. All modules and toolsets registered within it are "persistent" meaning that they are visible the moment the custom GUI application starts, and remain visible no matter which module the user switches to. It accepts two arguments, an application object (of type **AFXApp**) and a String that will appear in the title bar.

```
    AFXMainWindow.__init__(self, app, windowTitle)
```

Ths statement calls the **__init__** method of the base class (**AFXMainWindow**) which, to put things simply, takes care of a lot of stuff required for creating the main window that we don't need to know about.

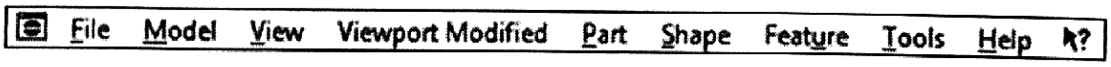
```
        # Register toolsets
        self.registerToolset(FileToolsetGui(), GUI_IN_MENUBAR|GUI_IN_TOOLBAR)
        self.registerToolset(ModelToolsetGui(), GUI_IN_MENUBAR)
        self.registerToolset(ViewManipToolsetGui(), GUI_IN_MENUBAR|GUI_IN_TOOLBAR)
```

The **registerToolset()** method is used to register persistent toolsets (toolsets that are always visible no matter which module you are in). Here the file, model and view manipulation toolsets are registered. **FileToolsetGui** displays the **File** menu and the **File** toolbar. **ModelToolsetGui** displays the **Model** menu. **ViewManipToolset** displays the **View** menu. It is required that every application register **viewManipToolsetGui**.

The **registerToolset()** method takes care of the actual registering. It is called when modules are constructed. Its first argument is the toolset (of type **AFXToolsetGui**) and the second one is a set of location options separated with a '|' for creating the toolset components. The available options are **GUI_IN_NONE** (no components), **GUI_IN_MENUBAR** (components in menubar), **GUI_IN_TOOLBAR** (components in toolbar) and **GUI_IN_TOOLBOX** (components in toolbox).

```
self.registerToolset(ModifiedCanvasToolsetGui(), GUI_IN_MENUBAR)
```

This statement registers the modified version of the canvas toolset we create in **modifiedCanvasToolsetGui.py**. **CanvasToolsetGui** displays the **Viewport** menu. We will dissect that script a later in this chapter, but for now just know that it changes the title of the **Viewport** menu to 'Viewport Modified'. Hence the menu bar in our custom application will look as shown in the following figure. Notice the menu titled 'Viewport Modified'.



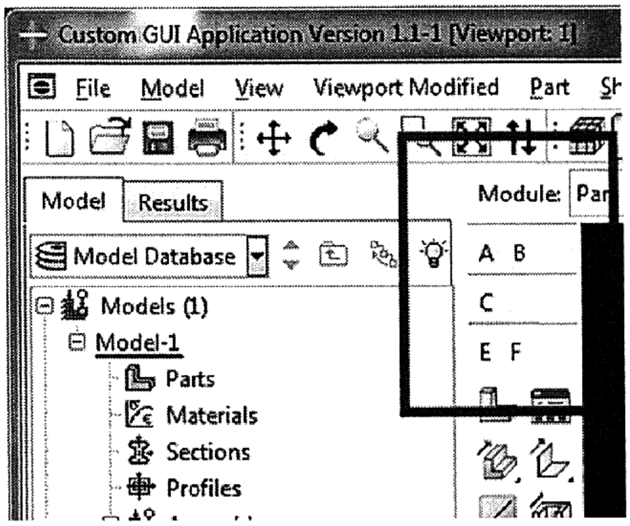
If you did not wish to modify this menu as we have done here, you would instead have used:

```
self.registerToolset(CanvasToolsetGui(), GUI_IN_MENUBAR)
```

The statement

```
self.registerToolset(CustomToolboxButtonsGui(), GUI_IN_TOOLBOX)
```

registers the custom toolset we create in **customToolboxButtonsGui.py**. We will examine this script later in this chapter, for now just know that it will create 5 toolset buttons labeled 'A', 'B', 'C', 'D', 'E' that will be displayed in the toolbox to the left of the viewport as displayed in the following figure.

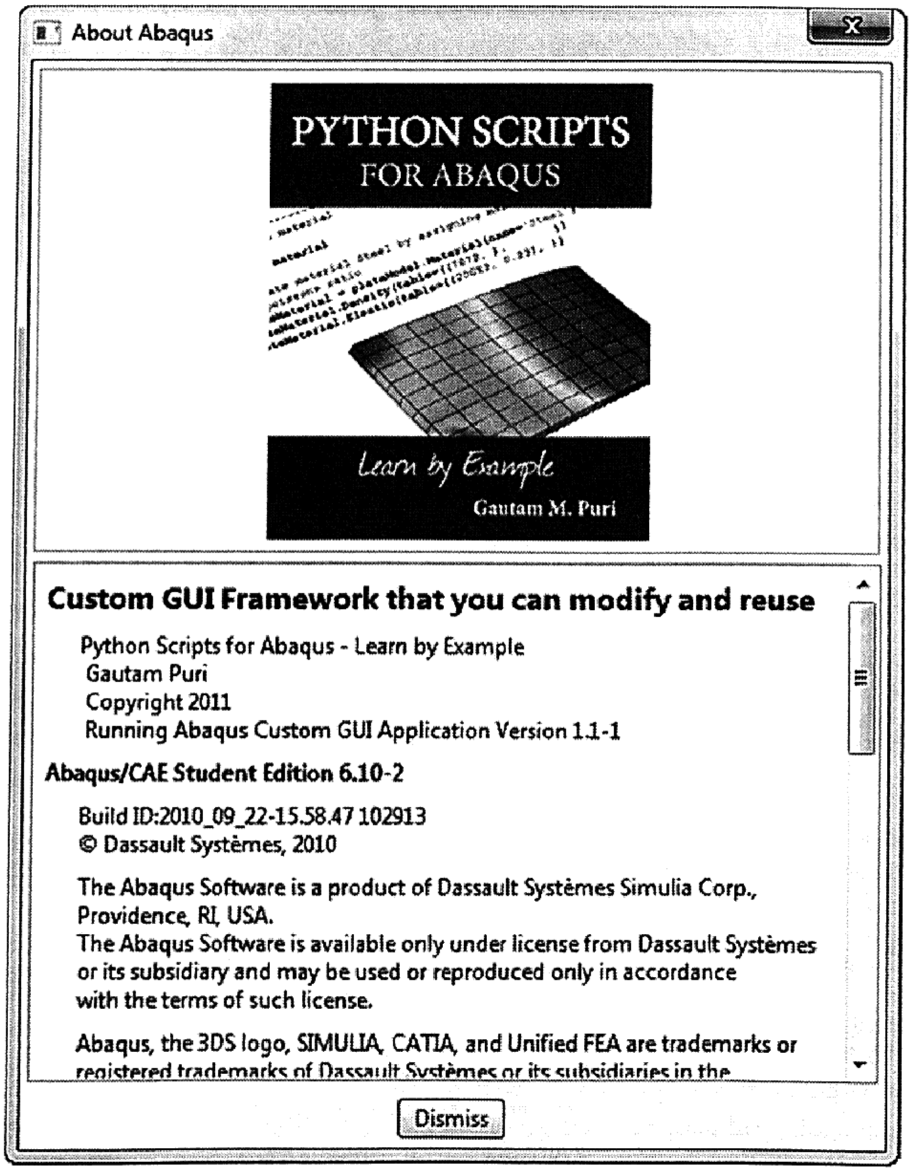


```
        abaqus_product_name = getAFXApp().getProductName()
        major_version_no, minor_version_no, update_no = \
                                            getAFXApp().getVersionNumbers()
        is_this_prerelease = getAFXApp().getPrerelease()
        if is_this_prerelease:
            version = '%s Version %s.%s-PRE%s' % (abaqus_product_name,
                                                  major_version_no,
                                                  minor_version_no,
                                                  update_no)
        else:
            version = '%s Version %s.%s-%s' % (abaqus_product_name,
                                               major_version_no,
                                               minor_version_no,
                                               update_no)
        custom_title = 'Custom GUI Framework that you can modify and reuse'
        custom_information = 'Python Scripts for Abaqus - Learn by Example ' + \
                             '\n Gautam Puri \n Copyright 2011' + \
                             '\n Running Abaqus ' + version
        customHelpToolsetGui.setCustomCopyrightStrings(custom_title,
                                                       custom_information)
        custom_icon=afxCreateIcon('icon_bookcover.png')
        customHelpToolsetGui.setCustomLogoIcon(custom_icon)
        self.registerHelpToolset(customHelpToolsetGui,
                                 GUI_IN_MENUBAR|GUI_IN_TOOLBAR)
```

This block modifies the **About Abaqus** dialog box displayed through **Help > About Abaqus...** Normally (if you wished to leave the help menu alone) you would use the statement:

```
self.registerHelpToolset(HelpToolsetGui(), GUI_IN_MENUBAR|GUI_IN_TOOLBAR)
```

However in this example we demonstrate how to modify the help dialog box with our own image and custom copyright information. This block makes the **About Abaqus** dialog box appear as displayed in the following figure:



Custom GUI Framework that you can modify and reuse

Python Scripts for Abaqus - Learn by Example
Gautam Puri
Copyright 2011
Running Abaqus Custom GUI Application Version 1.1-1

Abaqus/CAE Student Edition 6.10-2

Build ID:2010_09_22-15.58.47 102913
© Dassault Systèmes, 2010

The Abaqus Software is a product of Dassault Systèmes Simulia Corp., Providence, RI, USA.
The Abaqus Software is available only under license from Dassault Systèmes or its subsidiary and may be used or reproduced only in accordance with the terms of such license.

Abaqus, the 3DS logo, SIMULIA, CATIA, and Unified FEA are trademarks or registered trademarks of Dassault Systèmes or its subsidiaries in the

```
customHelpToolsetGui = HelpToolsetGui()
```

We create **customHelpToolsetGui** as an instance of the help toolset.

```
abaqus_product_name = getAFXApp().getProductName()
```

The **getAFXApp()** method returns the application object. The application object has a number of methods. A few of the commonly used ones are **getAFXMainWindow()** which returns a handle to the main window object, **getProductName()** which returns the product name, **getVersionNumbers()** which returns a tuple containing the major version number (**majorNumber**), minor version number (**minorNumber**) and update number (**updateNumber**), and **getPrerelease()** which returns a Boolean (True if the application is a prerelease, False otherwise).

The **getProductName()** method of the application object is used here to get the product name.

```
major_version_no, minor_version_no, update_no = \
                                        getAFXApp().getVersionNumbers()
```

Here the **getVersionNumbers()** method is used to obtain the major, minor and update numbers.

```
is_this_prerelease = getAFXApp().getPrerelease()
```

Here **getPrerelease()** is used to find out if the version of Abaqus is a prerelease version.

```
if is_this_prerelease:
    version = '%s Version %s.%s-PRE%s' % (abaqus_product_name,
                                          major_version_no,
                                          minor_version_no,
                                          update_no)
else:
    version = '%s Version %s.%s-%s' % (abaqus_product_name,
                                       major_version_no,
                                       minor_version_no,
                                       update_no)
```

Depending on whether the version is prerelease or not, a String is created with the product name, major, minor and update numbers, and the letters ‘PRE’.

```
custom_title = 'Custom GUI Framework that you can modify and reuse'
custom_information = 'Python Scripts for Abaqus - Learn by Example ' + \
                     '\n Gautam Puri \n Copyright 2011' + \
                     '\n Running Abaqus ' + version
customHelpToolsetGui.setCustomCopyrightStrings(custom_title,
                                               custom_information)
```

The **setCustomCopyrightStrings()** method allows you to set the title and the information in the **About Abaqus** dialog box. These strings are provided as arguments to the method.

```
custom_icon=afxCreateIcon('icon_bookcover.png')
```

The **afxCreateIcon()** method creates an icon using a provided image file and returns a handle to it. The file can be in the following formats – bmp, gif, png and xpm. The method automatically determines which of these file formats is being used from the extension and does the needful.

```
customHelpToolsetGui.setCustomLogoIcon(custom_icon)
```

The **setCustomLogoIcon()** method creates a logo or image using the icon provided as an argument and displays it in the **About Abaqus** dialog box as displayed in the figure.

```
self.registerHelpToolset(customHelpToolsetGui,
                         GUI_IN_MENUBAR|GUI_IN_TOOLBAR)
```

We now register our modified help toolset with the **registerHelpToolset()** method. Note that this is different from **registerToolset()** which we used for all the other toolsets.

```
self.registerToolset(AnnotationToolsetGui(),
                     GUI_IN_MENUBAR|GUI_IN_TOOLBAR)
self.registerToolset(DatumToolsetGui(), GUI_IN_TOOLBOX)
self.registerToolset(EditMeshToolsetGui(), GUI_IN_TOOLBOX)
self.registerToolset(PartitionToolsetGui(), GUI_IN_TOOLBOX)
self.registerToolset(QueryToolsetGui(), GUI_IN_TOOLBOX)
self.registerToolset(RepairToolsetGui(), GUI_IN_TOOLBOX)
self.registerToolset(SelectionToolsetGui(), GUI_IN_TOOLBAR)
self.registerToolset(TreeToolsetGui(), GUI_IN_TOOLBOX|GUI_IN_MENUBAR)
```

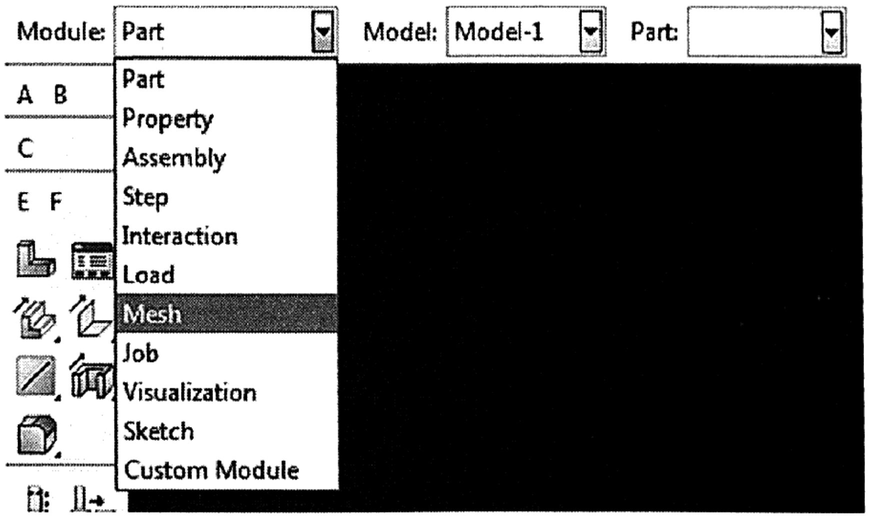
These statements register a number of other default toolsets available in Abaqus/CAE.

```
# Register the modules. These will appear in the Modules combo box in
# Abaqus/CAE in the order they are registered here
self.registerModule('Part',          'Part')
self.registerModule('Property',      'Property')
self.registerModule('Assembly',      'Assembly')
self.registerModule('Step',          'Step')
self.registerModule('Interaction',   'Interaction')
self.registerModule('Load',          'Load')
self.registerModule('Mesh',          'Mesh')
self.registerModule('Job',           'Job')
self.registerModule('Visualization', 'Visualization')
self.registerModule('Sketch',        'Sketch')
```

The **registerModule()** method registers modules. These become available in the **Module** combo box above the viewport.

```
# Register our custom module which resides in the script file
# customModuleGui.py
self.registerModule('Custom Module',       'customModuleGui')
```

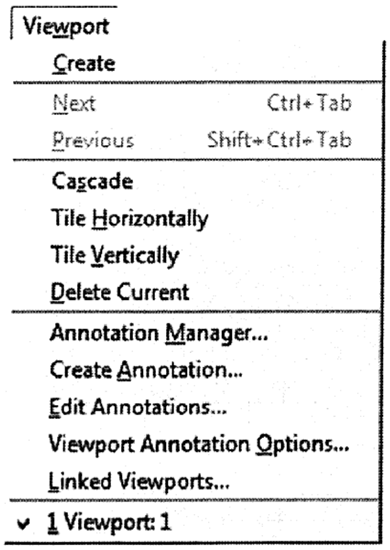
This statement registers the toolset 'Custom Module' which we create in **customModuleGui.py** and makes it available in the **Module** combo box. Note that we did not use an **import** statement at the top of the script to import **customModuleGui.py** because **registerModule** will take care of that for us. The **Module** combo box will now appear as shown in the figure.

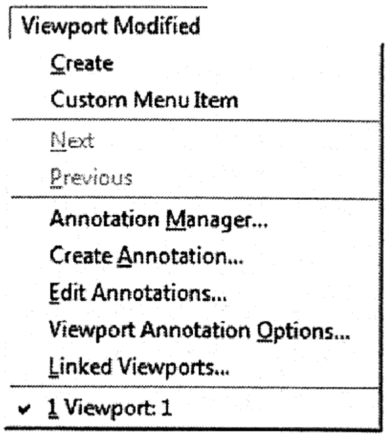


20.6.3 Modified Canvas Toolset (modified 'Viewport' menu)

This script is contained **in modifiedCanvasToolsetGui.py**. In it we modify the **Viewport** menu that is normally displayed in Abaqus/CAE. We add a menu item called 'Custom Menu Item'. When 'Custom Menu' is clicked it will launch a dialog box. This dialog box is an instance of **DemoDb** created by use in **demoDB.py**. Being a modal dialog box it is launched by a form mode **DemoForm** which we create in **demoForm.py**. (You will see how it is created when we focus on that script). We also remove the menu items **Cascade, Tile Horizontally, Tile Vertically** and **Delete Current**.

The following figure displays the usual **Viewport** menu as well as our modified version of it – **Viewport Modified** – for easy comparison.





```
# *********************************************************************************
# This script modifies the viewport menu which is part of the canvas toolset
# (CanvasToolsetGui)
# *********************************************************************************

from abaqusGui import *
from sessionGui import CanvasToolsetGui
from demoForm import DemoForm


# Class definition

class ModifiedCanvasToolsetGui(CanvasToolsetGui):

    def __init__(self):

        # Construct a modified canvas toolset (viewport menu)
        CanvasToolsetGui.__init__(self)

        # Remove 4 items from the Viewport menu
        menubar = getAFXApp().getAFXMainWindow().getMenubar()
        viewport_menu_with_contents = \
                                 getWidgetFromText(menubar, 'Viewport').getMenu()
        getWidgetFromText(viewport_menu_with_contents, 'Cascade').hide()
        getWidgetFromText(viewport_menu_with_contents, 'Tile Horizontally').hide()
        getWidgetFromText(viewport_menu_with_contents, 'Tile Vertically').hide()
        getWidgetFromText(viewport_menu_with_contents, 'Delete Current').hide()

        # Remove the 2nd horizontal separator
        getSeparator(viewport_menu_with_contents, 2).hide()
        # Add a new item to the menu after the 'Create' item
        # We use AFXMode.ID_ACTIVATE to cause the mode DemoForm in demoForm.py to
        # activate, which causes it to display its form DemoDB which is
        # in demoDB.py
        new_viewport_menu_item = AFXMenuCommand(self,
                                                viewport_menu_with_contents,
                                                'Custom Menu Item',
                                                None,
                                                DemoForm(self),
                                                AFXMode.ID_ACTIVATE)
        create_widget = getWidgetFromText(viewport_menu_with_contents, 'Create')
        new_viewport_menu_item.linkAfter(create_widget)

        # Modify the name of the Viewport menu
        viewport_menu = getWidgetFromText(menubar, 'Viewport')
        viewport_menu.setText('Viewport Modified')

        # Remove 3 items from the Viewport toolbar
        toolbar = self.getToolbarGroup('Viewport')
        # We find the widgets using their names which can be obtained by hovering
        # your mouse over them in CAE.
        # Since the names themselves are spread over 2 lines, we must use the
        # newline character '\n' in the appropriate places
        getWidgetFromText(toolbar, 'Cascade\nViewports').hide()
        getWidgetFromText(toolbar, 'Tile Viewports\nHorizontally').hide()
        getWidgetFromText(toolbar, 'Tile Viewports\nVertically').hide()
```

```
from abaqusGui import *
from sessionGui import CanvasToolsetGui
```

These statements give us access to the necessary GUI classes and modules.

```
from demoForm import DemoForm
```

In our custom viewport menu, we have a menu item that launches a dialog box **DemoDB** created by us in **demoDB.py**. Being a modal dialog it is launched by a form **DemoForm** which we create in **demoForm.py**. You will see how it is created when we dissect and study that script. In this script we are only importing it and calling it to launch the dialog box. For this reason we need to import the **DemoForm** class from **demoForm.py**.

```
class ModifiedCanvasToolsetGui(CanvasToolsetGui):
```

We derive our class **ModifiedCanvasToolsetGui** from **CanvasToolsetGui**. This is done because **CanvasToolsetGui** provides the standard menu items and basic functionality we need, and we only make modifications to this as opposed to creating an entirely custom menu (which we will do for **Custom Module** in **customModuleGui.py**).

```
def __init__(self):
```

The **__init__()** method, which you've already encountered in the previous section, is basically the constructor of the class and is called when an instance of the class is created. Just as in the case of the main window, the **__init__()** method in this class is where we do most of our scripting.

```
        # Construct a modified canvas toolset (viewport menu)
        CanvasToolsetGui.__init__(self)
```

The statement calls the **__init__()** method of the base class (**CanvasToolsetGui**) which does whatever is required to create the default **Viewport** menu and add functionality to it.

```
        menubar = getAFXApp().getAFXMainWindow().getMenubar()
```

getAFXApp(), which you have seen before, gives us a handle to the application. **getAFXMainWindow()** gives us a handle to the main window. And **getMenubbar()** returns a handle to the entire menu pane. We store this handle in the variable **menubar** for use in the next statement.

```
        viewport_menu_with_contents = \
                                    getWidgetFromText(menubar, 'Viewport').getMenu()
```

The **getWidgetFromText(parent, text)** method always returns the widget whose label or tip text matches the text specified as the **text** argument and is a child of the **parent** widget. In our case the parent widget is the menu and the text is 'Viewport' which is the title of the default **Viewport** menu. The **getWidgetFromText()** toolbar will return a handle to the title of our menu, which is a widget of type **AFXMenuTitle**, and not a handle to the entire menu.

Since we want the entire menu and its contents, we use **getMenu()**. The **getMenu()** method returns a handle to the popup menu associated with the widget. And the menu associated with our menu title is the one we want. I know this seems like a long winded way to do things but that's just how it is. We store the menu in the variable **viewport_menu_with_contents** for use in the next few statements.

```
        getWidgetFromText(viewport_menu_with_contents, 'Cascade').hide()
        getWidgetFromText(viewport_menu_with_contents, 'Tile Horizontally').hide()
        getWidgetFromText(viewport_menu_with_contents, 'Tile Vertically').hide()
        getWidgetFromText(viewport_menu_with_contents, 'Delete Current').hide()
```

In these statements we once again use **getWidgetsFromText()**. This time the viewport menu is the parent widget, and we use the text associated with each menu item (which is of type **AFXMenuCommand**) to get a handle to it.

We then use the **hide()** method to hide these so that they will no longer be displayed as menu items when the **Viewport** menu is clicked. These statements will therefore remove **Cascade**, **Tile Horizontally, Tile Vertically** and **Delete Current** from the menu.

```
# Remove the 2nd horizontal separator
getSeparator(viewport_menu_with_contents, 2).hide()
```

Here we use the **getSeparator()** method to obtain a handle to a separator - the horizontal lines that appears between some menu items. We wish to remove the horizontal rule between **Previous** and **Cascade** (or where cascade would have been if we hadn't hidden it in the previous statements). We provide the handle to the **Viewport** menu as the first argument. The second argument identifies which separator we wish to remove, but counting them from 1 upward. Since the separator we wish to remove is the second one (there is one between **Create** and **Next**) we pass '2' as a parameter.

Once we have a handle to the separator, we use the **hide()** method to remove it.

```
# Add a new item to the menu after the 'Create' item
# We use AFXMode.ID_ACTIVATE to cause the mode DemoForm in demoForm.py to
# activate, which causes it to display its form DemoDB which is
# in demoDB.py
new_viewport_menu_item = AFXMenuCommand(self,
                                        viewport_menu_with_contents,
                                        'Custom Menu Item',
                                        None,
                                        DemoForm(self),
                                        AFXMode.ID_ACTIVATE)
```

AFXMenuCommand() creates an instance of an enhanced version of the **FXMenuCommand** class. To put things briefly, it creates a menu item. The first argument is **owner**, which refers to the creator of the menu command, the second is **p**, which is the parent widget – in our case the **Viewport** menu. The third argument is **label**, which is the label for the menu button. The fourth is **ic**, which is the menu button icon. The fifth is **tgt**, the message target. We set the message target to **DemoForm(self)** which means it will create an instance of the **DemoForm** class which can launch the modal dialog **DemoDB** associated with it. So when the user clicks this menu item, he will see a dialog box. The sixth argument is **sel** which is the message ID. In our case we send an

ID_ACTIVATE message to **DemoForm** which causes it to launch **DemoDB**. Note that this statement does not display the new menu item in the menu, it only creates it internally.

```
create_widget = getWidgetFromText(viewport_menu_with_contents, 'Create')
new_viewport_menu_item.linkAfter(create_widget)
```

We use the **getWidgetFromText()** method to get a handle to **Create** so that we can use it to position our new menu item in the subsequent statement. **linkAfter()** is what actually displays our menu item in the menu. As its name suggests, it places the menu item after the widget specified as its parameter. The opposite of it is **linkBefore()**.

```
# Modify the name of the Viewport menu
viewport_menu = getWidgetFromText(menubar, 'Viewport')
viewport_menu.setText('Viewport Modified')
```

getWidgetFromText() is used again, this time only to get a handle on the title of the menu rather than the entire menu. **setText()** is used to change the text of the **Viewport** menu title to 'Viewport Modified'.

```
# Remove 3 items from the Viewport toolbar
toolbar = self.getToolbarGroup('Viewport')
```

This statement is used to get a handle to the **Viewport** toolbar. You can use this to get a handle for any of the toolbars by using their name. If you need to know the name of a toolbar, look in **View > Toolbars** in Abaqus/CAE.

```
# We find the widgets using their names which can be obtained by hovering
# your mouse over them in CAE.
# Since the names themselves are spread over 2 lines, we must use the
# newline character '\n' in the appropriate places
getWidgetFromText(toolbar, 'Cascade\nViewports').hide()
getWidgetFromText(toolbar, 'Tile Viewports\nHorizontally').hide()
getWidgetFromText(toolbar, 'Tile Viewports\nVertically').hide()
```

The **getWidgetFromText()** function allows us to grab the toolbar buttons using their tool tips. You can see these tool tips by hovering your mouse over those buttons in Abaqus/CAE. Note that newline characters have also been included in the text string to make the tooltips span multiple lines. The **hide()** method removes these toolbar buttons from the toolbar.

20.6.4 Custom Persistant toolset

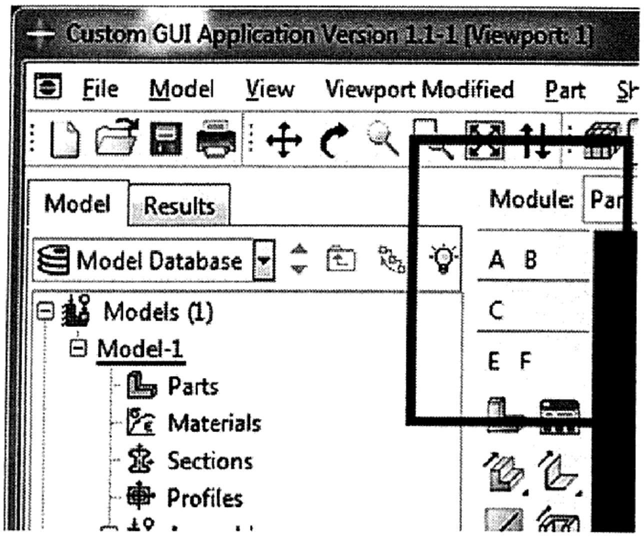
This script is contained in **customToolboxButtonsGui.py**. In it we create 5 new toolset buttons labeled 'A', 'B', 'C', 'E', 'F' that will be displayed in the module toolset area except that these will be persistent i.e., visible, no matter which module you are in. This is because we registered this toolset in the main window (**customCaeMainWindow.py**) as opposed to registering it within a custom module (we will see an example of that in customModuleGui.py hence it is a persistent toolset rather than a module toolset which would only display when the user is in that module.

This script demonstrates a very important task in GUI scripts, that of capturing events. We will assign unique identifiers – **ID_A, ID_B** and **ID_C** – to buttons 'A', 'B' and 'C'. We will map these IDs to functions within our custom toolset class using **FXMAPFUNC()**. When any of these buttons are clicked, **FXMAPFUNC()** will map the button click to the appropriate function depending on the ID of the button clicked.

To make things a little more interesting, we will store some functionality in a separate script **mainProgram.py** and the functions will call the appropriate method of the other script. This will help demonstrate not only how to store functionality in a separate file, but also how to create STATIC variables (I will explain these when we get there).

As for buttons 'E' and 'F' they will launch the dialog box (**DemoDB** in **demoDB.py**) by sending an **ID_ACTIVATE** instruction to the form (**DemoForm** in **demoForm.py**) just as was done when **Custom Menu Item** was clicked in our **Viewport Modified** menu in the previous section.

We will also divide the buttons into 3 toolbox groups which are automatically separated by horizontal separators as shown in the figure.



```
# ******************************************************************************
# This script creates a persistent toolset (toolbox buttons next to the viewport)
# which will remain visible no matter which module t in
# ******************************************************************************

from abaqusGui import *
from sessionGui import CanvasToolsetGui
from demoForm import DemoForm
from mainProgram import MainProgram

class CustomToolboxButtonsGui(AFXToolsetGui):

    [
        ID_A,
        ID_B,
        ID_C,
        ID_LAST
    ] = range(AFXToolsetGui.ID_LAST, AFXToolsetGui.ID_LAST+4)

    #This could be written as
    # ID_A = 1001
    # ID_B = 10022
    # ID_C = 1003

    print 'ID_A is %i' % ID_A
    print 'ID_B is %i' % ID_B
    print 'ID_C is %i' % ID_C


    def __init__(self):

        print 'CustomToolboxButtonsGui initialization method called.'
        # Construct base class.
        AFXToolsetGui.__init__(self, 'Test Toolset')


        FXMAPFUNC(self, SEL_COMMAND, self.ID_A, CustomToolboxButtonsGui.onCmdA)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_B, CustomToolboxButtonsGui.onCmdB)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_C, CustomToolboxButtonsGui.onCmdC)

        # Toolbox buttons
        toolbox_group_1 = AFXToolboxGroup(self)
        AFXToolButton(p=toolbox_group_1, label=' A \t Initialize the \n ' + \
                          'static variable x', icon=None, tgt=self, sel=self.ID_A)
        AFXToolButton(p=toolbox_group_1, label=' B \t Increment the static ' +\
                          '\n varible x by 5', icon=None, tgt=self, sel=self.ID_B)

        toolbox_group_2 = AFXToolboxGroup(self)
        AFXToolButton(p=toolbox_group_2, label=' C \t Increment the static ' + \
                                     '\n variable x by 5 \n (create an instance)',
                                               icon=None, tgt=self, sel=self.ID_C)

        toolbox_group_3 = AFXToolboxGroup(self)
        AFXToolButton(p=toolbox_group_3,
                      label=' E \t Display dialog',
                      icon=None,
                      tgt=DemoForm(self),
                      sel=AFXMode.ID_ACTIVATE)
        AFXToolButton(p=toolbox_group_3,
                      label=' F \t Display dialog',
                      icon=None, tgt=DemoForm(self),
                      sel=AFXMode.ID_ACTIVATE)


    def onCmdA(self, sender, sel, ptr):

        # Print a message to the command prompt
        print 'Toolbox button A was clicked'

        # Call method of MainProgram without using an instance variable
        MainProgram().createStaticVarX()

    def onCmdB(self, sender, sel, ptr):

        # Print a message to the command prompt
        print 'Toolbox button B was clicked'

        # Call method of MainProgram using an instance variable
        MainProgram().incrementXBy5()

    def onCmdC(self, sender, sel, ptr):

        # Print a message to the command prompt
        print 'Toolbox button C was clicked'
        # Print a message to the Abaqus/CAE message area
        getAFXApp().getAFXMainWindow() \
                            .writeToMessageArea('Toolbox button C was clicked.')

        # Call a meethod of MainProgram after creating an instance
        mp = MainProgram()
        mp.incrementXBy5()
```

```
from abaqusGui import *
from sessionGui import CanvasToolsetGui
from demoForm import DemoForm
from mainProgram import MainProgram
```

Here we import the standard modules and toolsets, as well as others created by us. The **DemoForm** class in **demoForm.py** is the form mode that launches our dialog box **DemoDB** defined in **demoDB.py**. We want this dialog box to be launched when 'E' or 'F' are clicked hence the script needs to be imported here. **mainProgram.py** on the other hand contains functionality which we will use when 'A', 'B' or 'C' are clicked, hence that script too needs to be imported here.

```
class CustomToolboxButtonsGui(AFXToolsetGui):
```

We derive our class **CustomToolboxButtonsGui** from **AFXToolsetGui**. If you recall, in the previous section (script **modifiedCanvasToolsetGui**) we derived our class from **CanvasToolsetGui** because we only wished to modify the **Viewport** menu that it supplies and retain the rest of the functionality. This time we wish to create entirely new functionality (menus/toolboxes/toolbars etc) hence we derive our class directly from **AFXToolsetGui**.

```
    [
        ID_A,
        ID_B,
        ID_C,
        ID_LAST
    ] = range(AFXToolsetGui.ID_LAST, AFXToolsetGui.ID_LAST+4)
```

Let me begin by explaining the syntax used here before we talk about what it does.

Range(x,y), which you have seen previously returns a list of numbers between **x** and **y** including **x** but not including **y**. For example, range(1,5) returns [1,2,3,4]. Here we are using the range function to return a list of numbers and assigning these to a list **[ID_A, ID_B, ID_C]**.

Moments ago we discussed the **ID_A, ID_B** and **ID_C** are identifiers which we will associate with buttons 'A', 'B' and 'C'. When any of these 3 buttons are pressed the **FXMAPFUNC()** method will be able to tell which button was pressed based on its ID, and will call the appropriate function.

This requires that each of the buttons in this script (or more technically each widget in the target class and its base classes) has a unique identifiers otherwise **FXMAPFUNC()** will not be able to differentiate between them. If we know for a fact exactly how many IDs are being used and we wish to manually assign these, we could instead write

```
ID_A = 1001
ID_B = 1002
ID_C = 1003
```

assigning 1001, 1002 and 1003 as the IDs of the 3 buttons. We would have to be sure that these IDs have not already been used in this class or in its base classes. We would also have to make sure we do not reuse these same IDs by mistake for any other buttons.

A much easier way of doing this is to let Abaqus assign the IDs itself so that they are all unique. We use the variable **ID_LAST** which is defined by Abaqus for the base class (**AFXToolsetGui**) that we derived our class from. The value of **ID_LAST** is one more than the last ID used in the base class **AFXToolsetGui**. Using the range function we accomplish what could instead be written as

```
ID_A = ID_LAST
ID_B = ID_LAST+1
ID_C = ID_LAST+2
```

This way **ID_A, ID_B** and **ID_C** are unique identities that have not already been used by our class **CustomToolboxButtonsGui** or the base class **AFXToolsetGui** (or any of its base classes).

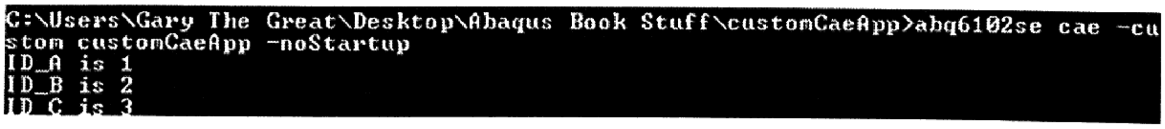
While not necessary in our case, for good programming practice we also go ahead and define an **ID_LAST** for our class which works out to

```
ID_LAST = ID_LAST+3
```

This may not be useful to us in this example, but if we were to derive another class from the class we have created, and then use **ID_LAST** in that class as we have done here, then Abaqus would use this value of **ID_LAST** specified by us. Abaqus does not itself update the value of **ID_LAST** on its own. This is something to be careful of if you derive classes from your own classes.

```
print 'ID_A is %i' % ID_A
print 'ID_B is %i' % ID_B
print 'ID_C is %i' % ID_C
```

For the sake of demonstration we print the values of **ID_A, ID_B** and **ID_C** to the console. The following figure displays some of what you see when you run the program.



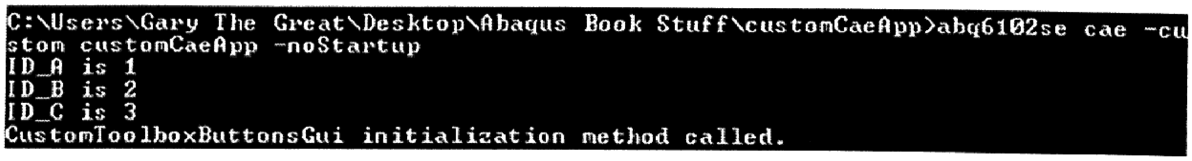
```
C:\Users\Gary The Great\Desktop\Abaqus Book Stuff\customCaeApp>abq6102se cae -cu
stom customCaeApp -noStartup
ID_A is 1
ID_B is 2
ID_C is 3
```

```
def __init__(self):
```

As usual we will put the important stuff in the **__init__()** method which will be called when an instance of the class is created.

```
        print 'CustomToolboxButtonsGui initialization method called.'
```

This **print** statement is for demonstration and debugging purposes. The following figure displays part of what you see (thanks to this **print** statement) when you run the program.



```
C:\Users\Gary The Great\Desktop\Abaqus Book Stuff\customCaeApp>abq6102se cae -cu
stom customCaeApp -noStartup
ID_A is 1
ID_B is 2
ID_C is 3
CustomToolboxButtonsGui initialization method called.
```

```
        # Construct base class.
        AFXToolsetGui.__init__(self, 'Test Toolset')
```

This statement calls the **__init__()** method of the base class (**AFXToolsetGui**) which provides the basic functionality to the toolset.

```
        FXMAPFUNC(self, SEL_COMMAND, self.ID_A, CustomToolboxButtonsGui.onCmdA)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_B, CustomToolboxButtonsGui.onCmdB)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_C, CustomToolboxButtonsGui.onCmdC)
```

We have already discussed the general working of the script and the **FXMAPFUNC()**. This is where we actually implement it.

FXMAPFUNC() takes 4 arguments – **self, messageType**, which is an integer specifying the type of message to send (in this case **SEL_COMMAND**), **messageId**, which is an integer specifying the message ID, and **method**, which is the function to be called, and which must be written in the format *classname.methodname*.

Here we have mapped the widget associated with **ID_A** (which is button 'A') to **onCmdA()**. Similarly button 'B' calls **onCmdB()** and 'C' calls **onCmdC()**. **SEL_COMMAND** is the most common message type you will encounter (and in fact the only one we use in this book) and it indicates that the widget has been clicked or selected by the user.

```
        # Toolbox buttons
        toolbox_group_1 = AFXToolboxGroup(self)
        AFXToolButton(p=toolbox_group_1, label=' A \t Initialize the \n ' + \
                      'static variable x', icon=None, tgt=self, sel=self.ID_A)
        AFXToolButton(p=toolbox_group_1, label=' B \t Increment the static ' +\
                      '\n varible x by 5', icon=None, tgt=self, sel=self.ID_B)

        toolbox_group_2 = AFXToolboxGroup(self)
        AFXToolButton(p=toolbox_group_2, label=' C \t Increment the static ' + \
                                  '\n variable x by 5 \n (create an instance)',
                                              icon=None, tgt=self, sel=self.ID_C)

        toolbox_group_3 = AFXToolboxGroup(self)
        AFXToolButton(p=toolbox_group_3,
                   label=' E \t Display dialog',
                   icon=None,
                   tgt=DemoForm(self),
                   sel=AFXMode.ID_ACTIVATE)
        AFXToolButton(p=toolbox_group_3,
                   label=' F \t Display dialog',
                   icon=None, tgt=DemoForm(self),
                   sel=AFXMode.ID_ACTIVATE)
```

This block creates all the toolbox buttons.

```
        toolbox_group_1 = AFXToolboxGroup(self)
```

AFXToolboxGroup is a class that creates a container for the groups of the toolbox. **AFXToolboxGroup()** is its constructor which we use here to create the first toolbox group. It accepts 2 arguments **owner** and **parent**. We set **owner**, which is the creator of the toolbox group, to **self**, which points to our class **CustomToolboxButtonsGui**. Since we don't specify a parent widget **parent** will default to **None**. We store our toolbox group in the variable **toolbox_group_1**.

```
        AFXToolButton(p=toolbox_group_1, label=' A \t Initialize the \n ' + \
                      'static variable x', icon=None, tgt=self, sel=self.ID_A)
```

```
AFXToolButton(p=toolbox_group_1, label=' B \t Increment the static ' +\
                    '\n varible x by 5', icon=None, tgt=self, sel=self.ID_B)
```

The **AFXToolButton** class creates buttons for both toolbars and toolboxes. Its constructor **AFXToolButton()** takes 6 arguments. **p** is the parent widget, which in this case is the toolbox group we created in the previous statement. **label** is the label for the button. If in the label you use a **\t**, anything following the **\t** is a tooltip which appears as you hover your mouse over the button. Anything before **\t** is what appears on the button. **icon** is an icon for the button. In this case we have no icon, but rather text ('A' and 'B'). For button 'A', as you hover your mouse over it, you will see the message 'Initialize the static variable x' and this will be spaced over 2 lines because of our use of **\n**. **tgt** is the message target which is the object/instance that will handle the message. We set it to self because the **CustomToolboxButtonsGui** will handle the message itself using its **FXMAPFUNC()** method. **sel** is the message ID which is passed to the message target, which in the case of button 'A' is **ID_A** and button 'B' is **ID_B**.

```
toolbox_group_2 = AFXToolboxGroup(self)
AFXToolButton(p=toolbox_group_2, label=' C \t Increment the static ' + \
                              '\n variable x by 5 \n (create an instance)',
                                     icon=None, tgt=self, sel=self.ID_C)
```

Similarly we put button 'C' in a toolbox group of its own. Abaqus will automatically place a horizontal separator between toolbox groups 1 and 2.

```
toolbox_group_3 = AFXToolboxGroup(self)
```

Similarly we create a third toolbox group for buttons 'E' and 'F'.

```
AFXToolButton(p=toolbox_group_3,
              label=' E \t Display dialog',
              icon=None,
              tgt=DemoForm(self),
              sel=AFXMode.ID_ACTIVATE)
AFXToolButton(p=toolbox_group_3,
              label=' F \t Display dialog',
              icon=None, tgt=DemoForm(self),
              sel=AFXMode.ID_ACTIVATE)
```

Buttons 'E' and 'F' are supposed to launch the dialog box **DemoDB** defined in **demoDB.py**. In order to do this they need to activate the form mode **DemoForm** in **demoForm.py**. Hence we do not set the target **tgt** to **self** but instead set it to **DemoForm(self)**. In addition we set the message ID **sel** to **AFXMode.ID_ACTIVATE**. Thus when buttons 'E' and 'F' are clicked, they send a message

AFXModel.ID_ACTIVATE to **DemoForm**. This causes the **activate()** method of the form to be called which launches the dialog box. You will learn more about the form, dialog box and the **activate()** method when we reach that section later in this chapter.

```
    def onCmdA(self, sender, sel, ptr):

        # Print a message to the command prompt
        print 'Toolbox button A was clicked'

        # Call method of MainProgram without using an instance variable
        MainProgram().createStaticVarX()
```

onCmdA() is called when button ‘A’ is clicked, sending an **ID_A** message to the **CustomToolboxButtonsGUI** instance, which **FXMAPFUNC()** maps to **onCmdA()**. Here we are defining the function itself. Notice that we have defined it inside the **CustomToolboxButtonsGui** class.

onCmdA() prints a message to the console indicating that ‘Toolbox button A was clicked’. It then calls the **createStaticVarX()** method of the **MainProgram** class (defined by us in **mainProgram.py**) without creating an instance of the class. We will look at **mainProgram.py** more closely in the next section, for now you only need to know that it has a class called **MainProgram** with a method called **createStaticVarX()** which we are calling here.

Note that by using the statement **MainProgram().createStaticVarX()** we do not create an instance of the **MainProgram** class, we are calling its method directly. This is different from creating an instance which we will do in **onCmdC()**.

```
    def onCmdB(self, sender, sel, ptr):

        # Print a message to the command prompt
        print 'Toolbox button B was clicked'

        # Call method of MainProgram using an instance variable
        MainProgram().incrementXBy5()
```

onCmdB() is called when button ‘B’ is clicked. It is similar to **onCmdA()** except that it calls the **incrementXBy5()** method of the **MainProgram** class.

```
    def onCmdC(self, sender, sel, ptr):

        # Print a message to the command prompt
        print 'Toolbox button C was clicked'
        # Print a message to the Abaqus/CAE message area
        getAFXApp().getAFXMainWindow() \
                             .writeToMessageArea('Toolbox button C was clicked.')

        # Call a meethod of MainProgram after creating an instance
        mp = MainProgram()
        mp.incrementXBy5()
```

onCmdC() is called when button 'C' is clicked. It is similar to **onCmdB()** and **onCmdA()**. However there are a couple of differences. **onCmdC()** uses the **print** command to print a message to the console, just like **onCmdA()** and **onCmdB()** do. As you may have noticed, in GUI applications **print** always prints to the console rather than the Abaqus/CAE message area. However **onCmdC()** also prints a message to the Abaqus/CAE message area using the statement

```
        # Print a message to the Abaqus/CAE message area
        getAFXApp().getAFXMainWindow() \
                             .writeToMessageArea('Toolbox button C was clicked.')
```

You've already encountered **getAFXApp()** and **getAFXMainWindow()** which return handles to the application and the main window. We use **writeToMessageArea()** to print a message to the Abaqus/CAE message area.

```
        # Call a meethod of MainProgram after creating an instance
        mp = MainProgram()
        mp.incrementXBy5()
```

The other difference in **onCmdC()** compared with **onCmdA()** or **onCmdB()** is that we assign an instance of the **MainProgram** class to a variable here named **mp**. We then call the **incrementXBy5()** method of this instance using the variable name.

Both methods of calling **incrementXBy5()** i.e., with and without creating a variable for the instance of **MainProgram()**, are used to demonstrate two ways of doing things.

20.6.5 Adding some functionality with a 'main' program

This script is contained in **mainProgram.py**. In this script we add some very basic functionality to our program for demonstration purposes. It contains a class called **MainProgram**. When our custom toolbox buttons 'A', 'B' and 'C' are clicked, the methods **onCmdA()**, **onCmdB()** and **onCmdC()** call corresponding methods of the **MainProgram** class. So indirectly, clicking button 'A' calls **createStaticVarX()** of the

MainProgram class, and clicking button 'B' or 'C' calls **incrementXBy5()** of the **MainProgram** class.

createStaticVarX() creates a static variable **static_x** with the value 0. **incrementXBy5()** increases the current value of **static_x** by 5 and prints it to the console window.

What is a static variable? In general, static variables are variables that are assigned a value once during programs execution, and these variables continue to exist and hold the value all the way till the end of the program. The value itself may be modified by the program, but the variable itself will continue to exist. One use of static variables is to hold, or share, a value across different objects of the same class.

Let me elaborate further. Let's assume you have a **class myClass**. It has a variable **myVar**. If you create an instance of that class called **myInstance1**, this instance will have that variable **myInstance1.myVar** and you can store a value in it. If you create another instance of **myClass** called **myInstance2**, it will have a variable **myInstance2.myVar**. However this variable will not necessarily have the same value as **myInstance1.myVar**. Unless... you make **myVar** a static variable. If **myVar** is a static variable, then **myInstance1.myVar** = **myInstance2.myVar**. Changing the value of either of them will also be reflected in the other.

In languages such as C or C++, static variables are explicitly defined. So in a C++ program you would write **static int x = 0** and a static integer variable x would be created an initialized with the value 0. In this manner you can create a static variable that is local or global in scope. Languages such as Pascal and Python on the other hand do not let you explicitly create a static variable, but they implicitly make all global variables static. So in order to make a variable static in Python we will need to declare it as a global variable.

In our program we wish to make the variable **static_x** static so that its value can be initialized by pressing button 'A' and the incremented by either 'B' or 'C'.

```
# ************************************************************************************
# This script contains the main functionality of the program.
# While the other scripts deal with GUI construction, this one contains the actual
# simulation code
# (in this example we don't do any actual model building but only build a framework
# for future use, hence we have very little useful functionality here)

# Note however that this script is also essentially a GUI script and not a kernel
# script as indicated by the import statement 'from abaqusGUI import *'
# ******************************************************************************


from abaqusGui import *

class MainProgram():

    def __init__(self):
        print 'MainProgram has been initialized.'


    # Function to create a static (global) variable 'static_x' and assign it the
    # value 0
    def createStaticVarX(self):

        global static_x
        static_x = 0
        print 'The value of x is now %d' % static_x

        # Display an information dialog box
        mainWindow = getAFXApp().getAFXMainWindow()
        showAFXInformationDialog(owner=mainWindow,
                        message='The variable static_x has been created/initialized')


    # Function to incremement the value of variable 'static_x' by 5
    def incrementXBy5(self):
        # We need to make sure 'static_x' exists (the user may not have clicked
        # button A, in which case createStaticVarX() was not called)
        try:
            global static_x
            static_x = static_x + 5
            print 'The value of x is now %d' % static_x
        except:
            # Display an error dialog box
            mainWindow = getAFXApp().getAFXMainWindow()
            showAFXErrorDialog(owner=mainWindow,
                               message='First define static_x by clicking on A')
```

```
from abaqusGui import *
```

Since this is a GUI script, it is necessary to include this **import** statement. In fact in the current state of the program it is necessary for this script to be a GUI script because a function of a kernel script cannot be called by menu items, toolbar buttons or toolbox buttons without using a mode or the **sendCommand()** method. If you recall, this is because of the separation of GUI and kernel scripts discussed at the beginning of this chapter. Realistically the only GUI task this script undertakes is displaying information and error dialog boxes (we will look at those in a minute), other than that it increments a variable and print its value in the console window.

```
class MainProgram():
```

We create a class called MainProgram in which to place all the functionality. It is not derived from any other class.

```
    def __init__(self):
        print 'MainProgram has been initialized.'
```

Like all classes we've encountered before, the **__initi__()** method of this class is called first as the constructor. Here we do not place any crucial code in this method except a **print** statement indicating that the **MainProgram** class has been instanced.

```
    # Function to create a static (global) variable 'static_x' and assign it the
    # value 0
    def createStaticVarX(self):

        global static_x
        static_x = 0
        print 'The value of x is now %d' % static_x

        # Display an information dialog box
        mainWindow = getAFXApp().getAFXMainWindow()
        showAFXInformationDialog(owner=mainWindow,
                       message='The variable static_x has been created/initialized')
```

This method is called when button 'A' is clicked.

```
        global static_x
```

As discussed earlier, in Python there is no way to explicitly declare a static variable. However making a variable global in scope makes it behave like a static variable. Hence the keyword **global** is used when declaring the variable.

```
        static_x = 0
```

We then assign it the value 0 to initialize it.

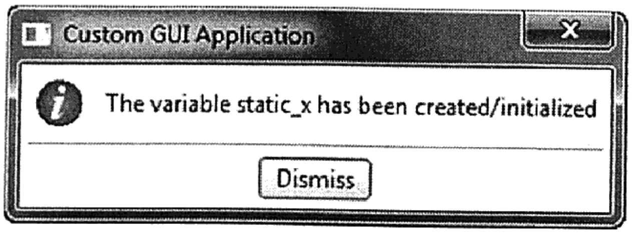
```
        mainWindow = getAFXApp().getAFXMainWindow()
```

This statement obtains a handle to the main window and places it in the variable **mainWindow**.

```
showAFXInformationDialog(owner=mainWindow,
                message='The variable static_x has been created/initialized')
```

The **showAFXInformationDialog()** is used to display the information dialog shown in the figure. Information dialog boxes are used to provide information to the user and have the information symbol ('i' symbol in a blue circle). They only contain a **Dismiss** button. Their title bar contains the application name. They are a good way to provide information to users in format they are accustomed to.

showAFXInformationDialog() accepts 4 arguments – **owner**, **message**, **tgt** and **sel**. **owner** is the window over which to center the dialog box, **message** is the text String to display as the message. **tgt** and **sel** are the message target and message ID neither of which are supplied here.



The variable static_x has been created/initialized

```
    # Function to incremement the value of variable 'static_x' by 5
    def incrementXBy5(self):
        # We need to make sure 'static_x' exists (the user may not have clicked
        # button A, in which case createStaticVarX() was not called)
        try:
            global static_x
            static_x = static_x + 5
            print 'The value of x is now %d' % static_x
        except:
            # Display an error dialog box
            mainWindow = getAFXApp().getAFXMainWindow()
            showAFXErrorDialog(owner=mainWindow,
                               message='First define static_x by clicking on A')
```

This method is called when buttons 'B' or 'C' are clicked. Its purpose is to increment the value of the variable **static_x**. Since it is possible the user did not click 'A' first, **static_x** has not yet been created and initialized, hence trying to increment it will cause an error that will crash the program. To prevent this we have used a **try-except** block, so that the program can respond to and recover from such an occurrence.

```
try:
    global static_x
    static_x = static_x + 5
```

```
        print 'The value of x is now %d' % static_x
```

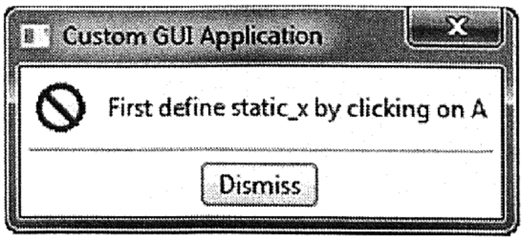
In the **try** block we attempt to achieve the purpose of this method. The statement **global static_x** is used to tell Python that we are using the previously defined global (and therefore also static) variable called **static_x**. If we left out the keyword **global**, Python would assume we are trying to create a local variable **static_x** since it is possible to have global and local variables with the same name. Needless to say, the local variable would not have the value of the global one, making it of no use to us.

```
    except:
        # Display an error dialog box
        mainWindow = getAFXApp().getAFXMainWindow()
        showAFXErrorDialog(owner=mainWindow,
                           message='First define static_x by clicking on A')
```

If the global variable **static_x** was not assigned a value earlier (by the user clicking 'A') then the statement **static_x = static_x + 5** in the **try** block will throw an exception, and the statements in the **except** block will be executed.

The **showAFXErrorDialog()** method is used to display the error dialog box shown in the figure. Error dialog boxes are used to inform the user of an error and they have the error symbol (red circle with a diagonal line). Just like information dialogs they only contain a **Dismiss** button. Their title bar contains the application name.

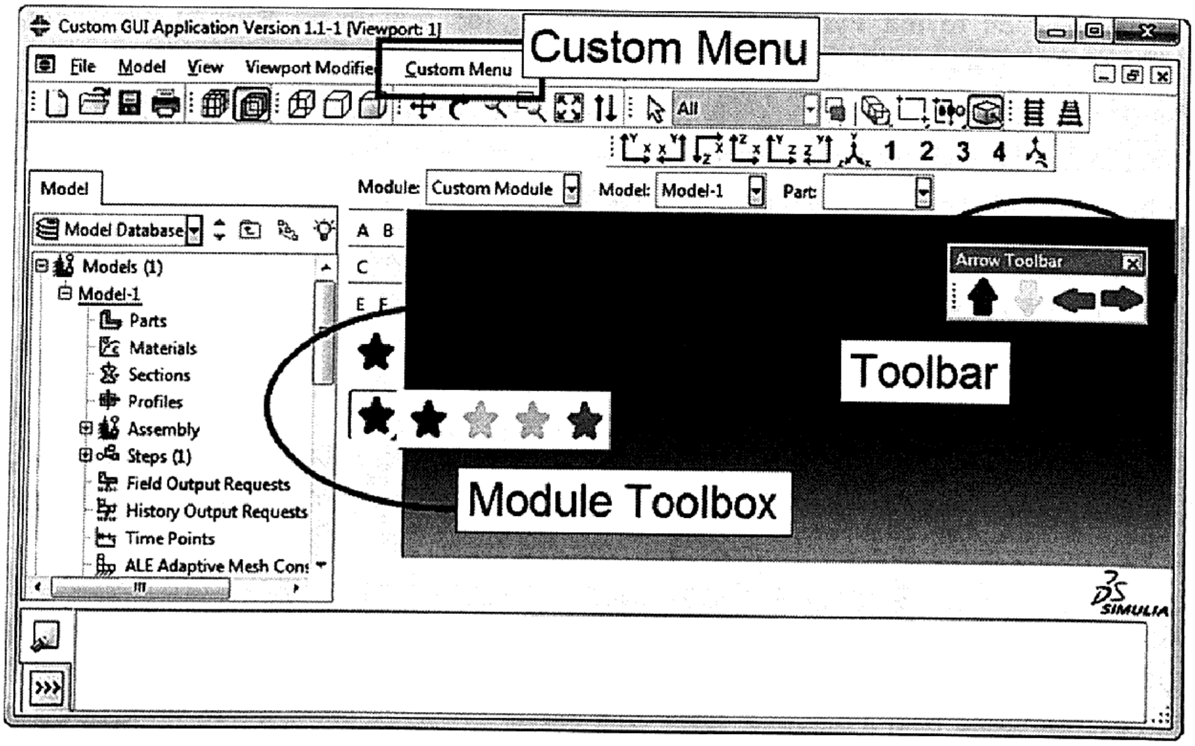
showAFXErrorDialog() takes 4 arguments – **owner**, **message**, **tgt** and **sel**. **owner** is the window over which to center the dialog box, **message** is the text string to display as the message. **tgt** and **sel** are the message target and message ID, neither of which are supplied here.


Custom GUI Application

First define static_x by clicking on A

20.6.6 Custom Module

This script is contained in **customModuleGui.py**. In this script we create a custom module. It will appear in the **Module** combo box above the viewport in the Abaqus/CAE interface. This custom module consists of a menu, a toolbar, and a toolbox. The menu contains a nested menu as well. Most menu items launch a modal dialog box but one of them launches a modeless dialog box. The toolbox includes flyout buttons.



```
# ******************************************************************************
# This script defines a custom module
# As part of this custom module we create a custom menu, toolbar and toolbox.
# The custom menubar launches dialog boxes posted through a form as well as a
# dialog box without a form.
# ******************************************************************************

from abaqusGui import *

from demoForm import DemoForm
from demoDBwoForm import DemoDBwoForm

# Class definition

class CustomModuleGui(AFXModuleGui):

    def __init__(self):

        print 'CustomModuleGui initialization method called.'

        # Construct base class.
        AFXModuleGui.__init__(self, 'Custom Module', AFXModuleGui.PART)
        # By default when you switch to a custom module the model tree disappears
        # and the region of the Abaqus/CAE window where the model tree was present
        # becomes blank.
        # The following 3 statements prevent this from happening. If you do
        # in fact desire that effect then comment/delete the next 3 lines.
        main_window=getAFXApp().getAFXMainWindow()
        main_window.appendApplicableModuleForTreeTab('Model',
                                               self.getModuleName() )
        main_window.appendVisibleModuleForTreeTab('Model', self.getModuleName() )

        # Create menu bar and add 2 items that post a dialog box using a form
        custom_menu = AFXMenuPane(self)
        AFXMenuTitle(self, '&Custom Menu', None, custom_menu)
        AFXMenuCommand(self, custom_menu, 'Custom Item 1 (Form)', None,
                                    DemoForm(self), AFXMode.ID_ACTIVATE)
        AFXMenuCommand(self, custom_menu, 'Custom Item 2 (Form)', None,
                                    DemoForm(self), AFXMode.ID_ACTIVATE)

        # Create a submenu within this menubar
        custom_sub_menu = AFXMenuPane(self)
        AFXMenuCascade(self, custom_menu, 'Custom Submenu', None, custom_sub_menu)
        AFXMenuCommand(self, custom_sub_menu, 'Custom Item 3 (Form)', None,
                                      DemoForm(self), AFXMode.ID_ACTIVATE)
        AFXMenuCommand(self, custom_sub_menu, 'Custom Item 3 (Form)', None,
                                      DemoForm(self), AFXMode.ID_ACTIVATE)

        # Add an item to the menu that posts a dialog box directly without using
        # a form
        dialog_box_without_form = DemoDBwoForm(self)
        dialog_box_without_form.create()
        AFXMenuCommand(self, custom_menu, 'Custom Item 4 (DialogBox)', None,
                                dialog_box_without_form, FXWindow.ID_SHOW)

        # Toolbar items
        toolbar_group = AFXToolbarGroup(owner=self, title='Arrow Toolbar')
        toolbar_icon_up = afxCreateIcon('icon_arrow_up.bmp')
        toolbar_icon_down = afxCreateIcon('icon_arrow_down.bmp')
        toolbar_icon_left = afxCreateIcon('icon_arrow_left.bmp')
        toolbar_icon_right = afxCreateIcon('icon_arrow_right.bmp')
        AFXToolButton(toolbar_group, '\tUp Arrow', toolbar_icon_up,
                                DemoForm(self), AFXMode.ID_ACTIVATE)
        AFXToolButton(toolbar_group, '\tDown Arrow', toolbar_icon_down,
                                DemoForm(self), AFXMode.ID_ACTIVATE)
        AFXToolButton(toolbar_group, '\tLeft Arrow', toolbar_icon_left,
                                DemoForm(self), AFXMode.ID_ACTIVATE)
        AFXToolButton(toolbar_group, '\tRight Arrow', toolbar_icon_right,
                                DemoForm(self), AFXMode.ID_ACTIVATE)

        # Toolbox button
        toolbox_group = AFXToolboxGroup(self)
        toolbox_icon = afxCreateIcon('icon_star_black.bmp')
        AFXToolButton(toolbox_group, '\tThis is a Tool Tip', toolbox_icon,
                                               DemoForm(self), AFXMode.ID_ACTIVATE)

         # Toolbox Flyout buttons
         toolbox_flyout_group = AFXToolboxGroup(self)
         toolbox_popup = FXPopup(getAFXApp().getAFXMainWindow())

         toolbox_flyout_icon_1 = afxCreateIcon('icon_star_red.bmp')
         toolbox_flyout_icon_2 = afxCreateIcon('icon_star_green.bmp')
         toolbox_flyout_icon_3 = afxCreateIcon('icon_star_orange.bmp')
         toolbox_flyout_icon_4 = afxCreateIcon('icon_star_blue.bmp')

         AFXFlyoutItem(toolbox_popup, '\tFlyout Button Red',
                       toolbox_flyout_icon_1, DemoForm(self), AFXMode.ID_ACTIVATE)
         AFXFlyoutItem(toolbox_popup, '\tFlyout Button Green',
                       toolbox_flyout_icon_2, DemoForm(self), AFXMode.ID_ACTIVATE)
         AFXFlyoutItem(toolbox_popup, '\tFlyout Button Orange',
                       toolbox_flyout_icon_3, DemoForm(self), AFXMode.ID_ACTIVATE)
         AFXFlyoutItem(toolbox_popup, '\tFlyout Button Blue',
                       toolbox_flyout_icon_4, DemoForm(self), AFXMode.ID_ACTIVATE)

         AFXFlyoutButton(toolbox_flyout_group, toolbox_popup)


# Actually create the custom module GUI.
CustomModuleGui()
```

```
from abaqusGui import *

from demoForm import DemoForm
from demoDBwoForm import DemoDBwoForm
```

As usual we begin with **import** statements. We import the form mode **DemoForm** from **demoForm.py**. We will use this to launch the modal dialog box **DemoDB** in **demoDB.py**. We also import the modeless dialog box **DemoDBwoForm** defined in **demoDBwoForm**. This dialog box will be launched directly without using a form.

```
class CustomModuleGui(AFXModuleGui):
```

We derive our class **CustomModuleGui** from **AFXModuleGui** which defines the basic module functionality, such as keeping track of the module's menus, toolbars and toolbox icons, and allowing them to be swapped in and out when the user switches into and out of the module.

```
    def __init__(self):
```

The code for the module will be placed in the **__init__()** method as expected.

```
        print 'CustomModuleGui initialization method called.'
```

We print this message for observation and debugging purposes.

```
        # Construct base class.
        AFXModuleGui.__init__(self, 'Custom Module', AFXModuleGui.PART)
```

As always we construct the base class. Note that this constructor accepts 3 arguments. The first argument is **self**, as usual. The second one is **moduleName**, which is the name that will be displayed in the module combo box. The third one is **displayTypes**, which specifies what type of object will be displayed in this module. Possible values are **AFXModuleGui.PART**, **AFXModuleGui.ASSEMBLY**, **AFXModuleGui.ODB**, **AFXModuleGui.XY_PLOT** and **AFXModuleGui.SKETCH**. Assuming we wish to display a part here, we will use **AFXModuleGui.PART**. Note that if you wished to display an assembly and specified **AFXModuleGui.ASSEMBLY**, you would be required to also import the assembly kernel module to initialize some assembly display options.

```
        # By default when you switch to a custom module the model tree disappears
        # and the region of the Abaqus/CAE window where the model tree was present
        # becomes blank.
        # The following 3 statements prevent this from happening. If you do
        # in fact desire that effect then comment/delete the next 3 lines.
        main_window=getAFXApp().getAFXMainWindow()
        main_window.appendApplicableModuleForTreeTab('Model',
                                                      self.getModuleName() )
        main_window.appendVisibleModuleForTreeTab('Model', self.getModuleName() )
```

In Abaqus/CAE you are used to seeing the model tree or the results tree on the left hand side. This is represented by **TreeToolsetGui**. The tabs (model/results) in this tree are not visible by default in custom modules. If you wish to make them visible you must use the **appendApplicableModuleForTreeTab()** method to create the tab applicable to that module, and the **appendVisibleModuleForTreeTab()** method to make it visible. The first argument supplied is the tab. In this case we make the **Model** tab applicable and visible in our custom module. Since we have not made the **Results** tab applicable and visible, you will not see the results tab when in the custom module.

```
        # Create menu bar and add 2 items that post a dialog box using a form
        custom_menu = AFXMenuPane(self)
        AFXMenuTitle(self, '&Custom Menu', None, custom_menu)
        AFXMenuCommand(self, custom_menu, 'Custom Item 1 (Form)', None,
                                          DemoForm(self), AFXMode.ID_ACTIVATE)
        AFXMenuCommand(self, custom_menu, 'Custom Item 2 (Form)', None,
                                          DemoForm(self), AFXMode.ID_ACTIVATE)
```

A menu is created as an **AFXMenuPane** object using the **AFXMenuPane()** constructor. It has a title which is an **AFXMenuTitle** object created with the **AFXMenuTitle()** constructor. It takes 3 parameters – **owner**, which can be a module or toolset GUI, **label**, which is the displayed title, **ic**, which is an icon (in our case None) and **popup**, which is the name of the pull down menu – we shall call it **custom_menu** and use it when creating the menu items. Note the use of the ampersand '&' in the label. It causes the letter it precedes to become a keyboard shortcut and underlines it. Since the letter 'c' has been made the shortcut, the user can press **Alt** + **C** on the keyboard instead of clicking on the menu item with the mouse, and the menu will drop open.

The menu items are **AFXMenuCommand** objects and are created using the **AFXMenuCommand()** constructor which takes 6 parameters. **owner** is the creator of the menu command. The second parameter **p** is the parent widget, in our case **custom_menu**, which was set as the pulldown in **AFXMenuTitle()**. The third is **label** which is the label for the menu button. **ic** is the menu button icon, which we set to **None** since we don't wish to give it an icon. The last two are **tgt** and **sel** which are the message target and message ID. We set the target to **DemoForm(self)** which is the form mode responsible for launching the dialogbox DemoDB defined in demoDB.py. We set the message to **AFXMode.ID_ACTIVATE** which will call the **activate()** method of the form mode. You have seen this target and message combination used previously to launch the dialog box.

```
        # Create a submenu within this menubar
        custom_sub_menu = AFXMenuPane(self)
        AFXMenuCascade(self, custom_menu, 'Custom Submenu', None, custom_sub_menu)
        AFXMenuCommand(self, custom_sub_menu, 'Custom Item 3 (Form)', None,
                                          DemoForm(self), AFXMode.ID_ACTIVATE)
        AFXMenuCommand(self, custom_sub_menu, 'Custom Item 3 (Form)', None,
                                          DemoForm(self), AFXMode.ID_ACTIVATE)
```

We now create a submenu. It too is an **AFXMenuPane** object created with the **AFXMenuPane()** constructor. However we then use an **AFXMenuCascade** object to identify it as a menu item with a submenu as opposed to an **AFXMenuCommand** object. This is done using the **AFXMenuCascade()** constructor which takes 6 arguments. These are **owner**, **p**, **label**, **ic** and **popup**, all of which were described a moment ago. This sub menu has **AFXMenuCommand** objects forming its menu items.

```
# Add an item to the menu that posts a dialog box directly without using
# a form
dialog_box_without_form = DemoDBwoForm(self)
dialog_box_without_form.create()
AFXMenuCommand(self, custom_menu, 'Custom Item 4 (DialogBox)', None,
                               dialog_box_without_form, FXWindow.ID_SHOW)
```

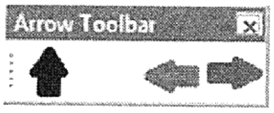
The last menu item we create has a slightly different function. While the rest of the menu and submenu items launch a modal dialog box **DemoDB** by calling the **activate()** method of its form mode **DemoForm**, this menu item will launch a modeless dialog box **DemoDBwoForm** direcly without using a form mode.

In order to do this we first create an instance of the **DemoDBwoForm** class that we create in **demoDBwoForm.py**. We call **the create()** method associated with it. When posting a dialog box using a form or procedure mode, the **create()** method is called automatically by the mode. However since we are launching the dialog box without using a mode we must call **create()** ourselves before we can call **show()**.

In order to call **show()** we send an **FXWindow.ID_SHOW** message to the dialog box when the **AFXMenuCommand** object (the menu item) is clicked.

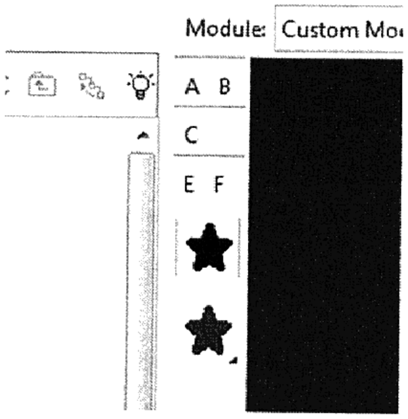
```
# Toolbar items
toolbar_group = AFXToolbarGroup(owner=self, title='Arrow Toolbar')
toolbar_icon_up = afxCreateIcon('icon_arrow_up.bmp')
toolbar_icon_down = afxCreateIcon('icon_arrow_down.bmp')
toolbar_icon_left = afxCreateIcon('icon_arrow_left.bmp')
toolbar_icon_right = afxCreateIcon('icon_arrow_right.bmp')
AFXToolButton(toolbar_group, '\tUp Arrow', toolbar_icon_up,
                             DemoForm(self), AFXMode.ID_ACTIVATE)
AFXToolButton(toolbar_group, '\tDown Arrow', toolbar_icon_down,
                             DemoForm(self), AFXMode.ID_ACTIVATE)
AFXToolButton(toolbar_group, '\tLeft Arrow', toolbar_icon_left,
                             DemoForm(self), AFXMode.ID_ACTIVATE)
AFXToolButton(toolbar_group, '\tRight Arrow', toolbar_icon_right,
                             DemoForm(self), AFXMode.ID_ACTIVATE)
```

The **AFXToolbarGroup** class creates a container to be used for groups in the toolbar. It creates vertical separators between toolbar groups. It accepts 3 arguments – **owner** (the creator of the group), **name** (the toolst name) and **title** (name that appears in the title bar of the menu when it is floating) – two of which are used here. You've previously seen **afxCreateIcon()** which creates an icon using a BMP, GIF, PNG or XPM file. Here 4 icons are created using 4 bitmap images. You've also seen the **AFXToolButton** class used previously to create a button for the custom toolbox in section 20.6.4.



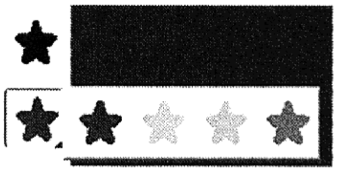
```
# Toolbox button
toolbox_group = AFXToolboxGroup(self)
toolbox_icon = afxCreateIcon('icon_star_black.bmp')
AFXToolButton(toolbox_group, '\tThis is a Tool Tip', toolbox_icon,
                             DemoForm(self), AFXMode.ID_ACTIVATE)
```

Here you see a toolbox being created as was done in **customToolboxButtonsGui.py** described in section 20.6.4. One button is added to the toolbox group with a black star shaped icon.



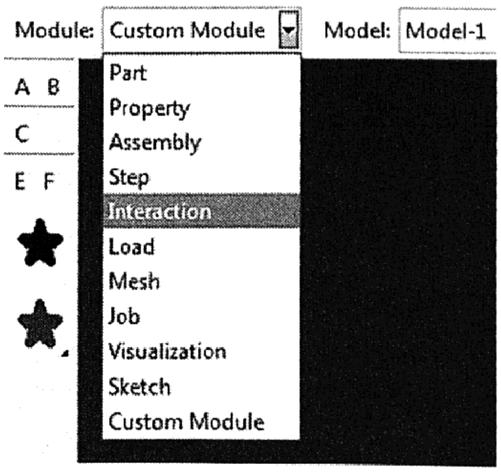
```
# Toolbox Flyout buttons
toolbox_flyout_group = AFXToolboxGroup(self)
toolbox_popup = FXPopup(getAFXApp().getAFXMainWindow())

toolbox_flyout_icon_1 = afxCreateIcon('icon_star_red.bmp')
toolbox_flyout_icon_2 = afxCreateIcon('icon_star_green.bmp')
toolbox_flyout_icon_3 = afxCreateIcon('icon_star_orange.bmp')
toolbox_flyout_icon_4 = afxCreateIcon('icon_star_blue.bmp')

AFXFlyoutItem(toolbox_popup, '\tFlyout Button Red',
              toolbox_flyout_icon_1, DemoForm(self), AFXMode.ID_ACTIVATE)
AFXFlyoutItem(toolbox_popup, '\tFlyout Button Green',
              toolbox_flyout_icon_2, DemoForm(self), AFXMode.ID_ACTIVATE)
AFXFlyoutItem(toolbox_popup, '\tFlyout Button Orange',
              toolbox_flyout_icon_3, DemoForm(self), AFXMode.ID_ACTIVATE)
AFXFlyoutItem(toolbox_popup, '\tFlyout Button Blue',
              toolbox_flyout_icon_4, DemoForm(self), AFXMode.ID_ACTIVATE)

AFXFlyoutButton(toolbox_flyout_group, toolbox_popup)
```

Another toolbox group is created here, except this toolbox group consists of a flyout button. When the user keeps his left mouse button pressed over this toolbox button, 3 other buttons are displayed. On the other hand if the user clicks without holding the mouse button down for a certain time duration, it is the same as clicking on the first out of the four buttons. Read the above statements and observe the use of **AFXFlyoutButton** widget which creates a flyout popup window and **AFXFlyoutItem** widget which creates the individual buttons within the flyout.



```
# Actually create the custom module GUI.
CustomModuleGui()
```

While the classes we define in other scripts are called by Abaqus while drawing the GUI or when responding to user activity, our custom module must be explicitly created by us otherwise it will not be created at all and will not appear in the module combo box. We accomplish this by calling the constructor of the class which contains all the instructions.



20.6.7 Form Mode

This script is contained in **demoForm.py**. In this script we create the form mode that launches the dialog box (**DemoDB** in **demoDB.py**).

```
# ***************************************************************************
# This script defines a form that will post a dialog box (DemoDB in demoDB.py)
# ***************************************************************************

from abaqusGui import *
import demoDB

# Class definition

class DemoForm(AFXForm):

    #-----------------------------------------------------------------------
    def __init__(self, owner):

        # Construct base class.
        AFXForm.__init__(self, owner)

        # Command to execute when OK is clicked
        # We have not created a method to deal with the user clicking OK hence we
        # shall put in kernalCommandMethod and kernelCommandObject for now
        self.cmd = AFXGuiCommand(self, 'kernalCommandMethod',
                                       'kernelCommandObject')

    #-----------------------------------------------------------------------
    # A getFirstDialog() method MUST be written for a mode (a Form mode)
    # It should return the first dialog box of the mode
    def getFirstDialog(self):

        # Reload the dialog module so that any changes to the dialog are updated.
        #
        reload(demoDB)
        return demoDB.DemoDB(self)

    #-----------------------------------------------------------------------
    def issueCommands(self):

        # Get the command string that will be sent to the kernel for processing
        cmds = self.getCommandString()

        # Since we have not actually created commands that will execute, let's
        # instead write this command string to the message area so we know it
        # executed
        getAFXApp().getAFXMainWindow().writeToMessageArea(cmds)

        self.deactivateIfNeeded()
        return TRUE

    #-----------------------------------------------------------------------
    # A mode is usually activated by sending it a message with its ID set
    # to ID_ACTIVATE
    # This message causes activate() to be called
    # Note that it is NOT necessary to define this activate() method unless you
    # wish to change the default behavior
    # Here we would like it to print a message to the screen before proceeding
    def activate(self):
        print 'Mode has been activated'
        # Now must call the base method
        AFXForm.activate(self)
```

```
from abaqusGui import *
import demoDB
```

We need to import **demoDb.py** which contains the class **DemoDB** – the dialog box.

```
class DemoForm(AFXForm):
```

We derive our class **DemoForm** from **AFXForm** which defines the basic form mode functionality.

```
    #-------------------------------------------------------------------
    def __init__(self, owner):

        # Construct base class.
        AFXForm.__init__(self, owner)

        # Command to execute when OK is clicked
        # We have not created a method to deal with the user clicking OK hence we
        # shall put in kernalCommandMethod and kernelCommandObject for now
        self.cmd = AFXGuiCommand(self, 'kernalCommandMethod',
                                       'kernelCommandObject')
```

The code for the mode is placed in the **__init__()** method as always. The base class is constructed to acquire the necessary functionality of a form mode before adding our customizations.

The **AFXGuiCommand** class is used to construct a GUI command that will be executed when the **OK** button is clicked. It accepts the arguments **mode**, **method**, **objectName** and **registerQuery**. **mode** can be a form mode or a procedure mode; we set it to **self**. **method** is a String specifying the method to execute in the kernel script, **objectName** is the name of the kernel script, and **registerQuery** is a Boolean specifying whether or not to register a query on the object so that the commands keyword values can be updated based on the kernel state.

In this example we are creating an application framework or an application prototype that you can build upon when creating your own GUI programs (which you will do in the next chapter). Hence we do not actually have any functionality in this application, and nothing is supposed to happen when the **OK** button is clicked. So we set **method** to **kernalCommandMethod** and **object** to **kernelCommandObject** as placeholders for when we actually have methods and scripts to use.

```
    #-----------------------------------------------------------------
    # A getFirstDialog() method MUST be written for a mode (a Form mode)
    # It should return the first dialog box of the mode
    def getFirstDialog(self):

        # Reload the dialog module so that any changes to the dialog are updated.
        #
        reload(demoDB)
        return demoDB.DemoDB(self)
```

As the comment states, every form mode must have a **getFirstDialog()** method. The method must return the first dialog box of the mode. We reload the form using **reload()**so that any changes to it are updated. The function then returns an instance of the dialog box **DemoDB**.

```
    #-----------------------------------------------------------------
    def issueCommands(self):

        # Get the command string that will be sent to the kernel for processing
        cmds = self.getCommandString()

        # Since we have not actually created commands that will execute, let's
        # instead write this command string to the message area so we know it
        # executed
        getAFXApp().getAFXMainWindow().writeToMessageArea(cmds)

        self.deactivateIfNeeded()
        return TRUE
```

The **issueCommands** method constructs the command String and issues it to the kernel. It also handles any exceptions thrown. It will deactivate the mode if the user pressed the **OK** button. The **issueCommands** method calls **getCommandString()**, **sendCommandString()** and **doCustomTask()**.

getCommandString() returns a String representing the command associated with the mode. By default it includes the required keywords in the order in which they were constructed in the mode. **sendCommandString()** takes the command String from **getCommandString()** and sends it to the kernel for processing. This method should not be overwritten. **doCustomTasks()** can be redefined to perform additional tasks after the command is processed by the kernel.

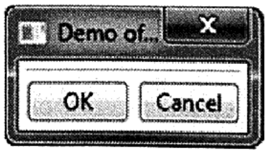
issueCommands() does not usually need to be called since the Abaqus GUI infrastructure will do this automatically. It is only necessary to call it manually if you interrupt the processing of the mode. Or, as in our case, if you wish to print the command String to the screen for debugging purposes. We obtain the String using **getCommandString()**. However we do not call **sendCommandString()**, which we must call manually to submit the command to the kernel now that we have modified **issueCommands()**. Hence no command will actually be passed when you click OK in this example application and therefore nothing happens.

```
    #-------------------------------------------------------------------
    # A mode is usually activated by sending it a message with its ID set
    # to ID_ACTIVATE
    # This message causes activate() to be called
    # Note that it is NOT necessary to define this activate() method unless you
    # wish to change the default behavior
    # Here we would like it to print a message to the screen before proceeding
    def activate(self):
        print 'Mode has been activated'
        # Now must call the base method
        AFXForm.activate(self)
```

The comments preceding this block summarize what you need to know. **activate()** is the method called whenever we pass an **ID_ACTIVATE** message to a dialog box such as **DemoDB**. This entire block of code does not need to be present in our script, since the **activate()** method is called by default. However since we wish to print a message to the screen when it is called, we have to define **activate()** manually. And since we have done that, the base **activate()** method must also be manually called using **AFXForm.activate(self)**.

20.6.8 Modal Dialog box

This script is contained in **demoDB.py**. In this script we create the dialog box (displayed in the figure) that will be posted by a number of menu items and toolbar and toolbox buttons.



```
# ******************************************************************************
# This script defines a dialog box that will be posted by a form (DemoForm in
# demoForm.py)

# Created for the book "Python Scripts for Abaqus - Learn by Example"
# Author: Gautam Puri
# **********************************************************************************

from abaqusGui import *

# Class definition

class DemoDB(AFXDataDialog):

    #-----------------------------------------------------------------------
    def __init__(self, form):

        # DIALOG_ACTIONS_SEPARATOR places a horizontal line/separator between the
        # contents of the dialog box and the OK/CANCEL buttons at the bottom
        # DIALOG_BAILOUT displays a message "Save changes made in the xyz dialog?"
        # if the user clicks Cancel after changing some values in the dialog box
        AFXDataDialog.__init__(self,
                               form,
                               'Demo of Dialog Box posted using Form',
                               self.OK|self.CANCEL,
                               DIALOG_ACTIONS_SEPARATOR|DATADIALOG_BAILOUT)

    #-----------------------------------------------------------------------
    # The show() method is called by default whenever the dialog box is to be
    # displayed.
    # For example in modifiedCanvasToolsetGui.py you have the statement
    # AFXMenuCommand(self, viewport_menu_with_contents, 'Custom Menu Item',  None,
    #                               DemoForm(self), AFXMode.ID_ACTIVATE)
    # So whenever the custom menu item is clicked, the activate() method of the
    # mode is called
    # This in turn tries to create the dialog box and the infrastructure calls the
    # create() and show() method of the dialog box.
    # Note that this method does NOT need to be defined here if we wish to leave
    # the behavior at default
    # Here however we wish to modify the show method to also print a message to
    # the screen (aside from opening the window which it does by default).
    def show(self):
        print 'Dialog box will be displayed'
        # Now must call the base show() command
        AFXDataDialog.show(self)


    #-----------------------------------------------------------------------
    # The hide() method is the opposite of show(). It is called by default
    # whenever the dialog box is to be hidden
    # Note that this method does NOT need to be defined here if we wish to leave
    # the behavior at default
    # Here however we wish to modify the hide method to also print a message to
    # the screen (aside from closing the window which it does by default).
    def hide(self):
        print 'Dialog box will be hidden'
        # Now must call the base hide() command
        AFXDataDialog.hide(self)
```

```
from abaqusGui import *
```

As usual, this **import** statement features at the top of the script.

```
class DemoDB(AFXDataDialog):
```

We derive our class **DemoDB** from **AFXDataDialog** which defines the basic dialog box functionality. A data dialog box is one in which data is collected from the user. If we did not wish to collect any information from the user and only wanted to display some information, we could instead use the base class **AFXDialog** from which **AFXDataDialog** is derived. **AFXDataDialog** is designed to be used with a form mode to gather data from the user. It is also best to use a data dialog in a module or any non-persistent toolset so that the infrastructure can manage the dialog box when the user switches modules. We will implement data collection and keyword usage in the next chapter, in this example we merely create an empty data dialog box so you understand the fundamentals.

```
    #-------------------------------------------------------------------
    def __init__(self, form):

        # DIALOG_ACTIONS_SEPARATOR places a horizontal line/separator between the
        # contents of the dialog box and the OK/CANCEL buttons at the bottom
        # DIALOG_BAILOUT displays a message "Save changes made in the xyz dialog?"
        # if the user clicks Cancel after changing some values in the dialog box
        AFXDataDialog.__init__(self,
                               form,
                               'Demo of Dialog Box posted using Form',
                               self.OK|self.CANCEL,
                               DIALOG_ACTIONS_SEPARATOR|DATADIALOG_BAILOUT)
```

The code for the dialog box, if any, is placed in the **__init__()** method. In our case the dialog box has no widgets inside it. Note that the **__init__()** method of the class has 2 parameters – **self** and **form**. As you are already aware, all methods inside of a class have **self** as the first parameter in the method declaration. The second parameter here is **form**. This placeholder will store the form mode. This is because in **DemoForm** we call the dialog box using **DemoDB(self)** in the **getFirstDialog()** method. Here **self** refers to the form **DemoForm** which is calling the constructor of **DemoDB**. It is this reference to **self** that is passed to **form** in the constructor of **DemoDB**.

The **__init__()** method of **DemoDB** calls the **__init__()** method (the constructor) of the base class **AFXDataDialog**. The constructor of **AFXDataDialog** accepts a number of parameters – **mode**, **title**, **actionButtonIds**, **opts**, **x**, **y**, **w** and **h**. We set **mode** to **form**, which, as stated before, refers to the form mode **DemoForm** that launches **DemoDB**. **title** is the title of the dialog box which will appear in its title bar.

actionButtonIds contains the ID's of the action buttons to be created at the bottom of the form. **self.OK** and **self.CANCEL** create **OK** and **Cancel** buttons. You can also have other buttons such as **APPLY**, **DISMISS** and so on. These action buttons have default behavior. For example **AFXDataDialog.OK** sends an **(ID_COMMIT, SEL_COMMAND)** message to the form mode with its button ID before hiding the dialog box. **AFXDataDialog.CANCEL** on the other hand checks for a bailout, and then sends an **(ID_DEACTIVATE, SEL_COMMAND)** message to the form mode with its button ID before hiding the dialog box. Other available buttons are **Apply**, **Continue**, **Defaults** and **x** in the title bar – refer to the documentation for more information on these.

opts provides some additional options – **DIALOG_ACTIONS_SEPARATOR** places a separator between the action buttons and the rest of the dialog box contents. **DATADIALOG_BAILOUT** jumps into action when the **Cancel** button is clicked – if the user has made changes to the contents of the dialog box before hitting **Cancel**, the application will post a standard warning dialog box. You can find other options in the documentation. **x** and **y**, if specified, are the X and Y coordinates of the origin, and **w** and **h** if specified are the width and height.

```
    #-------------------------------------------------------------------------
    # The show() method is called by default whenever the dialog box is to be
    # displayed.
    # For example in modifiedCanvasToolsetGui.py you have the statement
    # AFXMenuCommand(self, viewport_menu_with_contents, 'Custom Menu Item',  None,
    #                                         DemoForm(self), AFXMode.ID_ACTIVATE)
    # So whenever the custom menu item is clicked, the activate() method of the
    # mode is called
    # This in turn tries to create the dialog box and the infrastructure calls the
    # create() and show() method of the dialog box.
    # Note that this method does NOT need to be defined here if we wish to leave
    # the behavior at default
    # Here however we wish to modify the show method to also print a message to
    # the screen (aside from opening the window which it does by default).
    def show(self):
        print 'Dialog box will be displayed'
        # Now must call the base show() command
        AFXDataDialog.show(self)
```

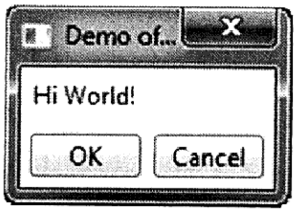
The comments in the above code describe what is going on. Whenever a dialog box is to be displayed, the **show()** method must be called. So when a form mode or a procedure mode launches a dialog box, it is calling the **show()** method. The infrastructure does this behind the scenes so you don't need to call **show()** yourself. In fact you don't need to define a **show()** method at all in your script. In this example we wish to print a message to the screen when **show()** is called, in order to help you understand how the infrastructure works, and for this reason we had to manually define **show()**. Since we have done this, we now have to make sure that the base **show()** command is called, hence we use **AFXDataDialog.show(self)**.

```
    #-----------------------------------------------------------------
    # The hide() method is the opposite of show(). It is called by default
    # whenever the dialog box is to be hidden
    # Note that this method does NOT need to be defined here if we wish to leave
    # the behavior at default
    # Here however we wish to modify the hide method to also print a message to
    # the screen (aside from closing the window which it does by default).
    def hide(self):
        print 'Dialog box will be hidden'
        # Now must call the base hide() command
        AFXDataDialog.hide(self)
```

The Abaqus GUI infrastructure calls the **hide()** method of a dialog box when it is supposed to close. For example, although the **OK** and **Cancel** buttons mean slightly different things (and presumably clicking **OK** will cause something to happen), both buttons need to close the dialog box when they are pressed. They both achieve this by calling the **hide()** method. Since the infrastructure takes care of this by default, you do not need to call **hide()**, nor do you need to define **hide()** at all in your scripts. The reason we define a **hide()** method here is so we can print a message to the screen when it is called, and therefore demonstrate how the infrastructure is working internally. Since we have defined a **hide()** method, we now need to call the base **hide()** command manually to do the actual hiding using **AFXDataDialog.hide(self)**.

20.6.9 Modeless Dialog box

This script is contained in demoDBwoForm.py. In this script we create the dialog box that will be directly posted without the use of a form mode by the last menu item in Custom menu (displayed in the Custom mode).



```
# ****************************************************************************
# This script describes a dialog box that will be posted directly without using a
# form
# ****************************************************************************

from abaqusGui import *

# Class definition
class DemoDBwoForm(AFXDialog):

    #-------------------------------------------------------------------
    def __init__(self, form):

        AFXDialog.__init__(self,
                           'Demo of Dialog Box posted without Form',
                           AFXDialog.OK|AFXDialog.CANCEL,
                           DECOR_RESIZE)

        FXLabel(self, 'Hi World!')
```

```
from abaqusGui import *
```

As usual, this **import** statement features at the top of the script.

```
class DemoDBwoForm(AFXDialog):
```

We derive our class **DemoDBwoForm** from **AFXDialog** which defines the basic dialog box functionality. In the previous section, we derived our modal dialog box from **AFXDataDialog** since it needed to gather data from the user (not that we implemented keyword usage in this example, but you'll see how that's done in the next chapter). In this dialog box on the other hand, let's assume we only wish to display something without accepting input from the user. The **AFXDialog** class suits our needs.

```
    #-------------------------------------------------------------------
    def __init__(self, form):

        AFXDialog.__init__(self,
                           'Demo of Dialog Box posted without Form',
                        AFXDialog.OK|AFXDialog.CANCEL,
                        DECOR_RESIZE)

        FXLabel(self, 'Hi World!')
```

The code for the dialog box, if any, is placed in the **__init__()** method. In our case the dialog box has one widgets inside it – a label created using **FXLabel()**. Note that the **__init__()** method of the class accepts a number of parameters – **owner**, **title**, **actionButtonIds**, **opts**, **x**, **y**, **w** and **h**. We've spoken about these parameters in previous sections. Here we set **owner** to **self**, indicating that the dialog box is not owned by any other object. We set the **title** to 'Demo of Dialog Box posted without Form', hence this will appear in the title bar of the dialog box. We wish to have **OK** and **Cancel** buttons so we provide IDs of **AFXDialog.OK** and **AFXDialog.CANCEL**. Aside from **OK** and **CANCEL**, other possible options are **APPLY**, **CONTINUE**, **DEFAULTS**, **DISMISS**, **NO**, **YES** and **YES_TO_ALL**. If we did not wish to have any action buttons we could instead use a 0 here. For **option** we specify **DECOR_RESIZE** which makes it possible for the user to resize the dialog box.

We did not define the **show()** method for this dialog box, but that does not mean it is not present. It exists by default, just as it would have even if we hadn't created it for **DemoDB**. The infrastructure calls it behind the scenes in order to show the dialog box, and it also calls **hide()** when it is time to close the dialog box.

20.7 Summary

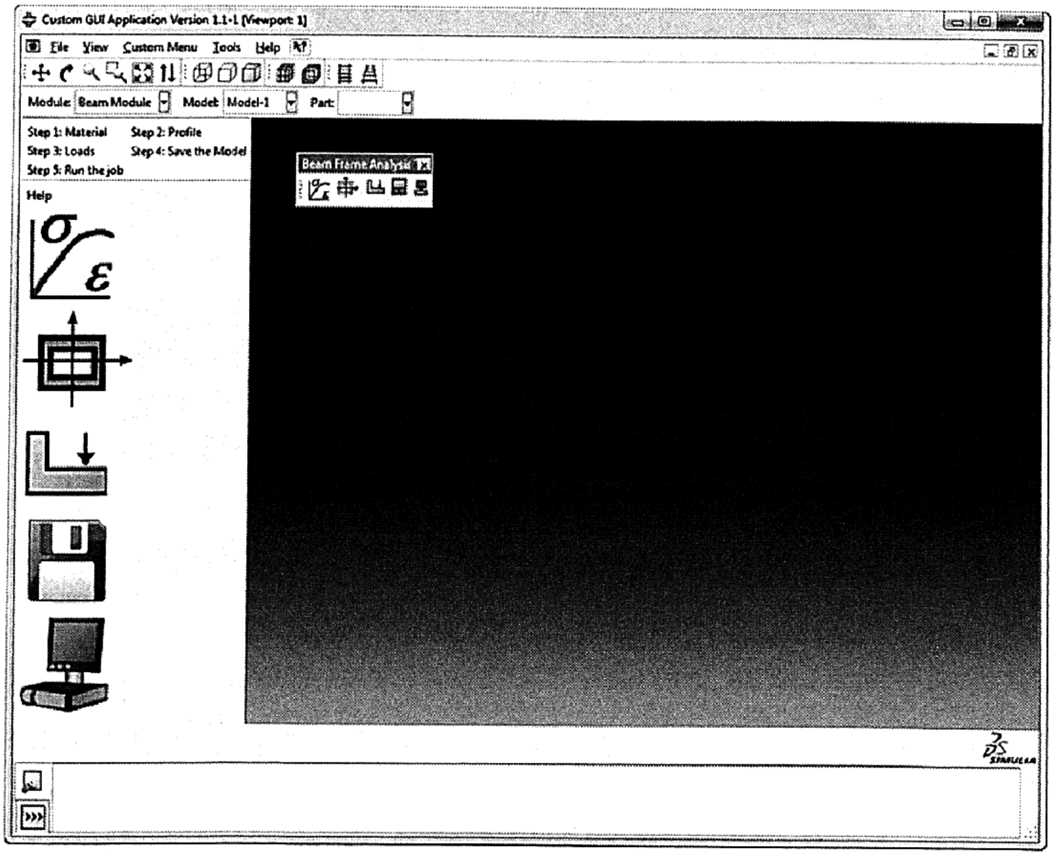
We created a working GUI framework in this chapter in order to explain the process of writing the scripts, and also to understand the inner workings of the Abaqus GUI infrastructure. The application created here does not do anything useful on its own, however the basic framework has been created, and it is one you can reuse when creating your own GUI applications. In fact we shall reuse it in the next chapter.

21

Custom GUI Application for Beam Frame Analysis

21.1 Introduction

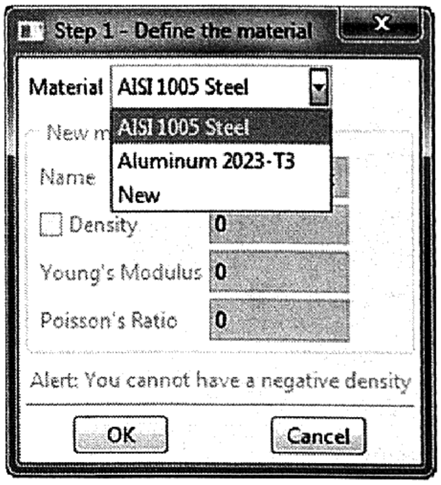
In the previous chapter we created a framework that can be reused for any GUI application. It included a persistent toolset, a custom module with menus, toolboxes, toolbuttons and a toolbar, and other customizations to the standard GUI interface.


Custom GUI Application Version 1.1-1 [Viewport: 1]
File View Custom Menu Tools Help
Module: Beam Module Model: Model-1 Part:
Step 1: Material Step 2: Profile
Step 3: Loads Step 4: Save the Model
Step 5: Run the job
Help
Beam Frame Analysis

In this chapter we will create a functional application that demonstrates project automation. We will use the beam frame model from Chapter 9. The application will create this same beam frame simulation, but prompt the user for inputs along the way. It will create a custom interface where the user can only perform certain actions, and only when prompted to do so, just as you would expect from a vertical application.

The figure displays our custom GUI application. It will not have a model tree on the left. The majority of menus and toolbars are removed leaving only a few barebones items. There is a persistent toolset with buttons 'Step 1' thru 'Step 5'. All the modules are removed as well leaving only a custom module called 'Beam Module'. This module has a module toolset which consists of 5 large buttons (with large icons on them). A custom toolbar is available with buttons and small icons. There is also a menu called 'Custom menu' with 5 menu items. The persistent toolset, beam module toolset (with the big icons), the toolbar, and the custom menu all have 5 buttons/items and provide the exact same functionality.

When 'Step 1' is initiated using any of the buttons or menus, the user is prompted for material properties. He can select 'Steel' or 'Aluminum' or define a new material. When the user clicks **OK**, Abaqus proceeds to create the model, beam parts (frame and crossbracing) and materials (using the users input).


Step 1 - Define the material

Material: AISI 1005 Steel

AISI 1005 Steel
Aluminum 2023-T3
New

New m

Name

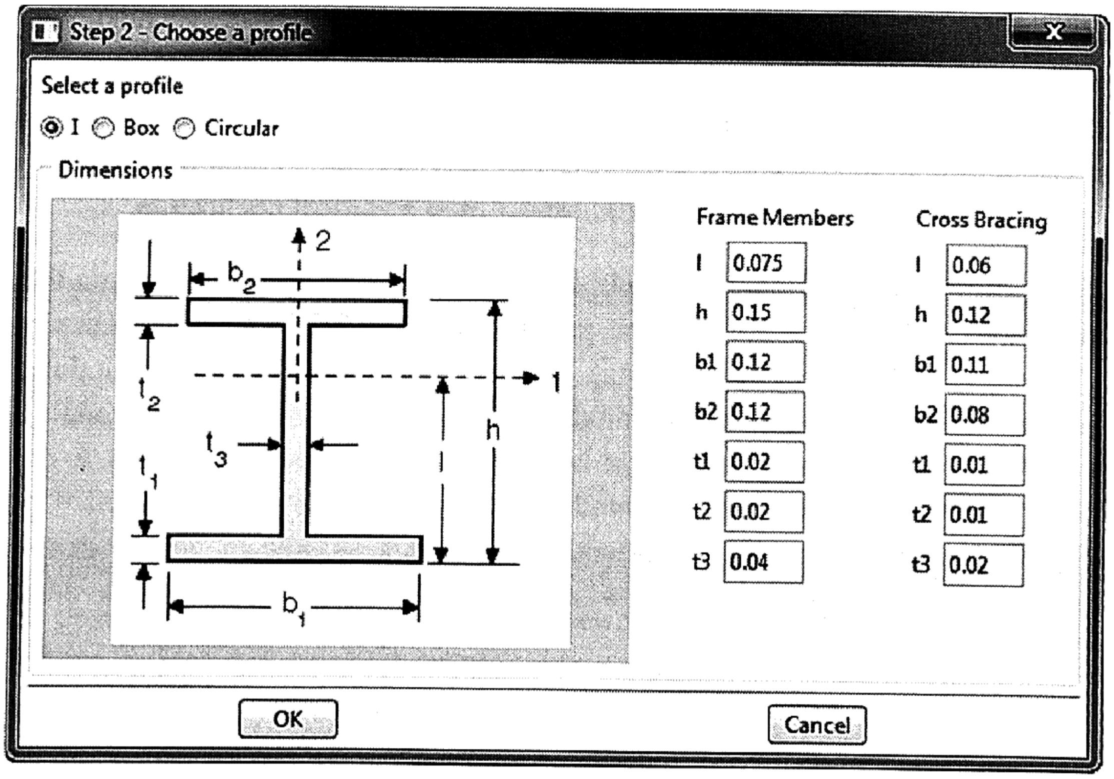
☐ Density 0

Young's Modulus 0

Poisson's Ratio 0

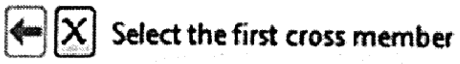
Alert: You cannot have a negative density

When 'Step 2' is initiated, the user is prompted to create the profile of the beam with options of 'I', 'Box' and 'Circular'. A number of default values are filled into the fields which the user can alter. When the use clicks **OK** the profiles are created. The application also proceeds to create the sections and assembly.

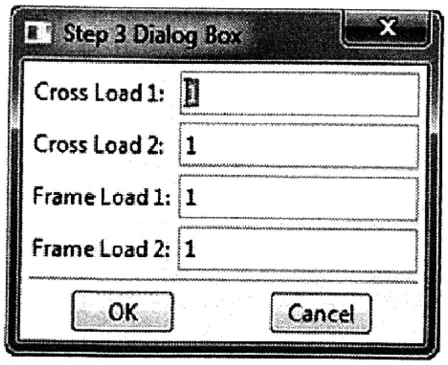


	Frame Members	Cross Bracing
I	0.075	0.06
h	0.15	0.12
b1	0.12	0.11
b2	0.12	0.08
t1	0.02	0.01
t2	0.02	0.01
t3	0.04	0.02

When 'Step 3' is initiated, the user is prompted to select a cross member, then a second, and then two frame members. The user will be able to pick these in the viewport.


Select the first cross member

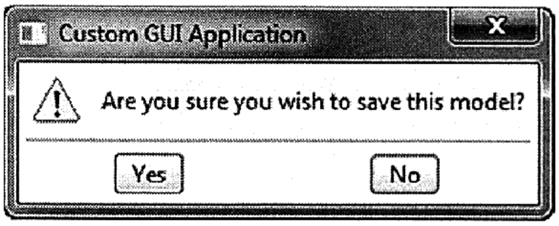
The application will then prompt the user to enter loads for each of the members selected.



Cross Load 1:	1
Cross Load 2:	1
Frame Load 1:	1
Frame Load 2:	1

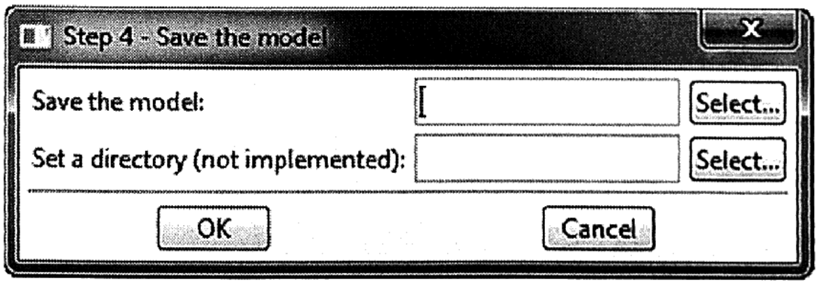
On accepting these inputs, the application will create the loads and display the assembly with loads in the viewport.

'Step 4' asks the user if he wishes to save the model'.


Custom GUI Application

Are you sure you wish to save this model?

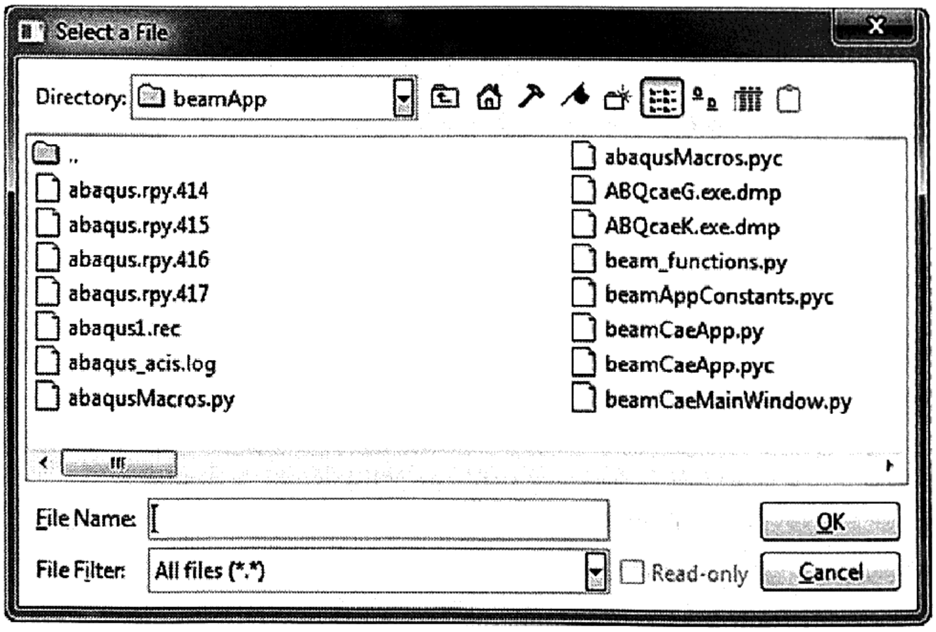
If he clicks **Yes** he is asked to provide a path at which to save the model.


Step 4 - Save the model

Save the model:

Set a directory (not implemented):

If he clicks the **Select...** button, he will be provided a file selection window


Select a File

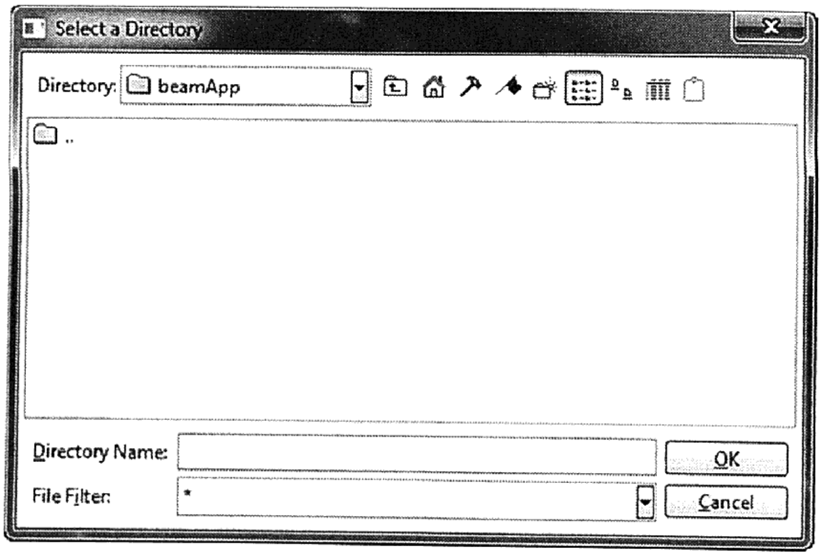
Directory: beamApp

..
abaqus.rpy.414
abaqus.rpy.415
abaqus.rpy.416
abaqus.rpy.417
abaqus1.rec
abaqus_acis.log
abaqusMacros.py
abaqusMacros.pyc
ABQcaeG.exe.dmp
ABQcaeK.exe.dmp
beam_functions.py
beamAppConstants.pyc
beamCaeApp.py
beamCaeApp.pyc
beamCaeMainWindow.py

File Name:

File Filter: All files (*.*)

The directory selection on the other hand is not actually implemented in this application, but is provided to show you how to present the user with a directory selection window if you need to do so in one of your own scripts. If the user clicks **Select...** next to 'set a directory', he will see the directory selection window.



When the **OK** button is finally clicked, the entire model is saved at the specified file location.

Finally 'Step 5' runs the analysis.

21.2 Layout Managers and Widgets

In the custom CAE example of the previous chapter, our dialog boxes were mostly empty. This time they will be populated with useful text fields, check boxes, radio buttons and combo boxes. All of these are known as widgets. In fact regular buttons, toolbar and toolbox buttons, flyout buttons and menu buttons are also widgets, so you have in fact used widgets before. Widget is a generic term for GUI controls, and these widgets allow a user to interact with the program.

Layout managers are containers used to arrange widgets in a dialog box. You place the widgets within the layout manager, and depending on the type of layout manager those widgets will be placed in an ordered manner in the dialog box. For example, a vertical alignment layout manager will cause all widgets inside it to be placed one below the other. A tab book layout manager on the other hand will allow you to have multiple tabs, and different widgets in each tab which will be displayed only when the user is in that tab.

You'll use layout managers and widgets in the dialog boxes for 'Step 1' through 'Step 4' so you'll have a good understanding of them by the end of the chapter.

21.3 Transitions and Process Updates

Transitions allow you to detect changes in the state of widgets. The program can then change the GUI state in a dialog box based on the detected activity. For example, in the dialog box for 'Step 1', the user is presented with 3 material choices – 'AISI 1005 Steel', 'Aluminum 2024-T3' and 'New'. A transition is added to the application to detect whether the user has clicked 'New' or not, and if he has, a number of text fields are enabled allowing him to provide a name and material properties for this material. On the other hand if 'Steel' or 'Aluminum' are selected, these material property fields will be disabled or grayed out.

The transition allows the program to detect the change in state of the combo box widget and execute the appropriate method to enable or disable the text fields. Transitions do this by comparing the value of the keyword associated with the widget with a specified value and doing a simple comparison such as EQ (equals), GT (greater than) or LT (less than). However sometimes you may need to perform a more complicated comparison, or meet some more complex condition that cannot be represented using simple comparisions such as EQ, GT and LT. In that case you will need to use process updates.

The **processUpdates()** method is called during every GUI update cycle. You can place your own code in this method to test for some condition, and if some condition is met then you can execute the relevant methods. Needless to say this should be used with caution since it is called at every GUI update, and if you have a lot of time consuming code here you can slow your program down considerably.

We will demonstrate how to use transitions in the dialog box for 'Step 1', and **processUpdates()** in the dialog box for 'Step 2'.

21.4 Exploring the scripts

A number of scripts are associated with this example and must exist together in the same folder to make the application work. We shall explore each of these in detail.

21.4.1 Beam Application Kernel Script

This following script is the kernel script responsible for creating the beam model. It is the backbone of the application. While the rest of the scripts deal with constructing the GUI and responding to events (and are therefore GUI scripts), this script is called at the end of each 'step' and executes the necessary kernel commands to create parts, materials, sections and so on.

The script is contained in **beamKernel.py**

```
# ********************************************************************************
# Custom Beam Frame Analysis GUI Application
# This script is the heart of the program. It sends commands to the kernel
# to set up the model and run the analysis job
# Hence it is a kernel script, it does not participate in GUI construction

# Created for the book "Python Scripts for Abaqus - Learn by Example"
# Author: Gautam Puri
# ********************************************************************************

from abaqus import *
from abaqusConstants import *
import regionToolset

step_counter = 0

class BeamKernel():

    def createModelPartMaterial(self, mat_selected, mat_name, mat_density,
                                                            mat_youngs, mat_poissons):

        global material_name

        # The user may have selected steel or aluminum in which case the rest of
        # the fields will be left blank and the parameters will have default
        # values
        if mat_selected == 1 :
            material_name = 'AISI 1005 Steel'
            mat_density = 7800.0
            mat_youngs = 200E9
            mat_poissons = 0.3
        elif mat_selected == 2 :
            material_name = 'Aluminum 2024-T3'
            mat_density = 2770.0
            mat_youngs = 73.1E9
            mat_poissons = 0.33
        else:
            material_name = mat_name

        # if you try to provide a youngs modulus and poissons ratio both equal to
        # zero Abaqus throws an error because in the elasticity table you've
        # given it a row of zeros
        if mat_youngs==0 and mat_poissons==0 :
            print 'Young\s modulus and Poisson\s ratio cant both be zero'
    # Since we exit Step 1 we reset the global counter so the user can
    # perform Step 1 again
    return False


session.viewports['Viewport: 1'].setValues(displayedObject=None)

# ---------------------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Beam Frame')

# Make beamModel a global variable so it can be accessed by the other
# methods without being passed to them as an argument
# This is similar to having a 'static' variable in a language such as
# C++ ie the variable exists without creating an instance of the class
global beamModel

beamModel = mdb.models['Beam Frame']

# ---------------------------------------------------------------------
# Create the parts

import sketch
import part

# ---------------------------------------------------------------------
# Create the frame

# Start with a 3D Point Deformable Body
global framePart
framePart = beamModel.Part(name='Frame', dimensionality=THREE_D,
                                         type=DEFORMABLE_BODY)
framePart.ReferencePoint(point=(0.0, 0.0, 0.0))

# ---------------------------------------------------------------------
# a) Create one side of the frame

# Create other datum points by offsetting from the reference point
the_reference_point = framePart.referencePoints[1]
framePart.DatumPointByOffset(point=the_reference_point,
                             vector=(13.0,0.0,0.0))
framePart.DatumPointByOffset(point=the_reference_point,
                             vector=(4.0,-3.0,0.0))
framePart.DatumPointByOffset(point=the_reference_point,
                             vector=(1.0,0.0,0.0))

# There are 3 items in the datums repository, the 3 datum points that
# have been created
# Extract the keys of the datums repository, these will be used to get
# the datum points
# The keys will be numbers but might be in random order as dictionaries
```

```
    # are unordered
    # Sort them to get them in ascending order as abaqus assigns keys in
    # ascending order as a datum point is created
    # Once the points are available a datum plane is created using the 3
    #points
    framePart_datums_keys = framePart.datums.keys()
    framePart_datums_keys.sort()
    frame_datum_point_1 = framePart.datums[framePart_datums_keys[2]]
    frame_datum_point_2 = framePart.datums[framePart_datums_keys[1]]
    frame_datum_point_3 = framePart.datums[framePart_datums_keys[0]]
    framePart.DatumPlaneByThreePoints(point1=frame_datum_point_1,
                                      point2=frame_datum_point_2,
                                      point3=frame_datum_point_3)

    # Create a datum axis
    framePart.DatumAxisByPrincipalAxis(principalAxis=YAXIS)

    # There are 5 objects in the datums repository, 3 datum points, a datum
    # plane, and the datum axis
    # The datum plane will be the one whose key is the second to highest
    # number
    # The datum axis will be the one whose key is the highest number
    framePart_datums_keys = framePart.datums.keys()
    framePart_datums_keys.sort()
    index_of_plane = (len(framePart_datums_keys) - 2)
    index_of_axis = (len(framePart_datums_keys) - 1)
    frame_datum_plane = \
                 framePart.datums[framePart_datums_keys[index_of_plane]]
    frame_datum_axis = \
                 framePart.datums[framePart_datums_keys[index_of_axis]]

    # Create the sketch
    sketch_transform1 = \
             framePart.MakeSketchTransform(sketchPlane=frame_datum_plane,
                                           sketchUpEdge=frame_datum_axis,
                                           sketchPlaneSide=SIDE1,
                                           sketchOrientation=LEFT,
                                           origin=(0.0, 0.0, 0.0))
    framePart_sketch = mdb.models['Beam Frame'] \
                          .ConstrainedSketch(name='frame sketch 1',
                                             sheetSize=20,
                                             gridSpacing=1,
                                             transform=sketch_transform1)

    framePart_sketch.Line(point1=(1.0,0.0), point2=(4.0,-3.0))
    framePart_sketch.Line(point1=(4.0,-3.0), point2=(6.0,-3.0))
    framePart_sketch.Line(point1=(6.0,-3.0), point2=(8.0,-3.0))
    framePart_sketch.Line(point1=(8.0,-3.0), point2=(10.0,-3.0))
    framePart_sketch.Line(point1=(10.0,-3.0), point2=(13.0,0.0))
    framePart_sketch.Line(point1=(13.0,0.0), point2=(8.0,0.0))
    framePart_sketch.Line(point1=(8.0,0.0), point2=(6.0,0.0))
    framePart_sketch.Line(point1=(6.0,0.0), point2=(1.0,0.0))
```

```
framePart_sketch.Line(point1=(6.0,0.0), point2=(6.0,-3.0))
framePart_sketch.Line(point1=(8.0,0.0), point2=(8.0,-3.0))

# Use the sketch to create a wire
framePart.Wire(sketchPlane=frame_datum_plane,
               sketchUpEdge=frame_datum_axis, sketchPlaneSide=SIDE1,
               sketchOrientation=LEFT, sketch=framePart_sketch)

# -------------------------------------------------------------------
# b) Create other side of the frame

# Create a datum plane by offsetting from existing one
framePart.DatumPlaneByOffset(plane=frame_datum_plane, flip=SIDE1,
                                                      offset=1.5)

framePart_datums_keys = framePart.datums.keys()
framePart_datums_keys.sort()
index_of_plane2 = (len(framePart_datums_keys) - 1)
frame_datum_plane2=framePart \
                          .datums[framePart_datums_keys[index_of_plane2]]

framePart.DatumPointByCoordinate(coords=(1.0, 0.0, 1.5))
framePart.DatumPointByCoordinate(coords=(13.0, 0.0, 1.5))
framePart.DatumPointByCoordinate(coords=(4.0, -3.0, 1.5))

framePart.DatumAxisByTwoPoint(point1=(0.0,0.0,1.5), point2=(0.0,5.0,1.5))

framePart_datums_keys = framePart.datums.keys()
framePart_datums_keys.sort()
index_of_axis2 = (len(framePart_datums_keys) -1)
frame_datum_axis2 = \
                  framePart.datums[framePart_datums_keys[index_of_axis2]]

# Make the sektch
sketch_transform2 = \
           framePart.MakeSketchTransform(sketchPlane=frame_datum_plane2,
                                         sketchUpEdge=frame_datum_axis2,
                                         sketchPlaneSide=SIDE1,
                                         sketchOrientation=LEFT,
                                         origin=(0.0, 0.0, 1.5))
framePart_sketch2 = \
  mdb.models['Beam Frame'].ConstrainedSketch(name='frame sketch 2',
                                             sheetSize=20,
                                             gridSpacing=1,
                                             transform=sketch_transform2)

framePart_sketch2.Line(point1=(1.0,0.0), point2=(4.0,-3.0))
framePart_sketch2.Line(point1=(4.0,-3.0), point2=(6.0,-3.0))
framePart_sketch2.Line(point1=(6.0,-3.0), point2=(8.0,-3.0))
framePart_sketch2.Line(point1=(8.0,-3.0), point2=(10.0,-3.0))
framePart_sketch2.Line(point1=(10.0,-3.0), point2=(13.0,0.0))
framePart_sketch2.Line(point1=(13.0,0.0), point2=(8.0,0.0))
```

```
    framePart_sketch2.Line(point1=(8.0,0.0), point2=(6.0,0.0))
    framePart_sketch2.Line(point1=(6.0,0.0), point2=(1.0,0.0))
    framePart_sketch2.Line(point1=(6.0,0.0), point2=(6.0,-3.0))
    framePart_sketch2.Line(point1=(8.0,0.0), point2=(8.0,-3.0))


    # Use the sketch to create a wire
    framePart.Wire(sketchPlane=frame_datum_plane2,
                   sketchUpEdge=frame_datum_axis2, sketchPlaneSide=SIDE1,
                   sketchOrientation=LEFT, sketch=framePart_sketch2)

    # ------------------------------------------------------------------
    # Create the cross bracing

    # Start with a 3D Point Deformable Body
    global crossPart
    crossPart = beamModel.Part(name='CrossBracing', dimensionality=THREE_D,
                                                    type=DEFORMABLE_BODY)
    crossPart.ReferencePoint(point=(0.0, 0.0, 0.0))

    crossPart.DatumPointByCoordinate(coords=(1.0, 0.0, 0.0))
    crossPart.DatumPointByCoordinate(coords=(1.0, 0.0, 1.5))
    crossPart.DatumPointByCoordinate(coords=(4.0, -3.0, 0.0))
    crossPart.DatumPointByCoordinate(coords=(4.0, -3.0, 1.5))
    crossPart.DatumPointByCoordinate(coords=(6.0, 0.0, 0.0))
    crossPart.DatumPointByCoordinate(coords=(6.0, 0.0, 1.5))
    crossPart.DatumPointByCoordinate(coords=(6.0, -3.0, 0.0))
    crossPart.DatumPointByCoordinate(coords=(6.0, -3.0, 1.5))
    crossPart.DatumPointByCoordinate(coords=(8.0, 0.0, 0.0))
    crossPart.DatumPointByCoordinate(coords=(8.0, 0.0, 1.5))
    crossPart.DatumPointByCoordinate(coords=(8.0, -3.0, 0.0))
    crossPart.DatumPointByCoordinate(coords=(8.0, -3.0, 1.5))
    crossPart.DatumPointByCoordinate(coords=(10.0, -3.0, 0.0))
    crossPart.DatumPointByCoordinate(coords=(10.0, -3.0, 1.5))
    crossPart.DatumPointByCoordinate(coords=(13.0, 0.0, 0.0))
    crossPart.DatumPointByCoordinate(coords=(13.0, 0.0, 1.5))


    crossPart_datums_keys = crossPart.datums.keys()
    crossPart_datums_keys.sort()

    datum_points = crossPart.datums
    crossPart.WirePolyLine(points=((datum_points[crossPart_datums_keys[0]],
                                    datum_points[crossPart_datums_keys[1]]),
                                   (datum_points[crossPart_datums_keys[2]],
                                    datum_points[crossPart_datums_keys[3]]),
                                   (datum_points[crossPart_datums_keys[4]],
                                    datum_points[crossPart_datums_keys[5]]),
                                   (datum_points[crossPart_datums_keys[6]],
                                    datum_points[crossPart_datums_keys[7]]),
                                   (datum_points[crossPart_datums_keys[8]],
                                    datum_points[crossPart_datums_keys[9]]),
                                    (datum_points[crossPart_datums_keys[10]],
                                     datum_points[crossPart_datums_keys[11]]),
                                    (datum_points[crossPart_datums_keys[12]],
                                     datum_points[crossPart_datums_keys[13]]),
                                    (datum_points[crossPart_datums_keys[14]],
                                     datum_points[crossPart_datums_keys[15]])),
                                     mergeWire=OFF, meshable=ON)

        # -------------------------------------------------------------------
        # Create material

        import material

        # Create material by assigning mass density, youngs modulus and poissons
        # ratio
        beamMaterial = beamModel.Material(name=material_name)

        # If material density is 0 (meaning user did not enter a value and it
        # defaulted to 0), we will not assign a density
        if mat_density !=0 :
            beamMaterial.Density(table=((mat_density, ),       ))

        beamMaterial.Elastic(table=((mat_youngs, mat_poissons), ))

        return True





    def createProfiles(self, cs_profile, cs_l_1, cs_l_2, cs_h_1, cs_h_2, cs_b1_1,
                       cs_b1_2, cs_b2_1, cs_b2_2, cs_t1_1, cs_t1_2, cs_t2_1,
                       cs_t2_2, cs_t3_1, cs_t3_2, cs_a_1, cs_a_2, cs_b_1, cs_b_2,
                       cs_t_1, cs_t_2, cs_r_1, cs_r_2):
    # ----------------------------------------------------------------------
    # Create profiles

        if cs_profile==1:
            beamModel.IProfile(name='FrameProfile', l=cs_l_1, h=cs_h_1,
                               b1=cs_b1_1, b2=cs_b2_1, t1=cs_t1_1, t2=cs_t2_1,
                               t3=cs_t3_1)
            beamModel.IProfile(name='CrossProfile', l=cs_l_2, h=cs_h_2,
                               b1=cs_b1_2, b2=cs_b2_2, t1=cs_t1_2, t2=cs_t2_2,
                               t3=cs_t3_2)
        elif cs_profile == 2:
            beamModel.BoxProfile(name='FrameProfile', b=cs_b_1, a=cs_a_1,
                                             uniformThickness=ON, t1=cs_t_1)
            beamModel.BoxProfile(name='CrossProfile', b=cs_b_2, a=cs_a_2,
                                             uniformThickness=ON, t1=cs_t_2)
        else :
            beamModel.CircularProfile(name='FrameProfile', r=cs_r_1)
        beamModel.CircularProfile(name='CrossProfile', r=cs_r_2)



    def createSectionAssemblySteps(self):
        # ------------------------------------------------------------------
        # Create sections and assign frame and crosbracing to them

        global beamModel
        global framePart
        global crossPart
        global material_name

        import section

        frameSection = beamModel.BeamSection(name='Frame Section',
                                             profile='FrameProfile',
                                             integration=DURING_ANALYSIS,
                                             material=material_name)
        crossSection = beamModel.BeamSection(name='Cross Section',
                                             profile='CrossProfile',
                                             integration=DURING_ANALYSIS,
                                             material=material_name)

        global edges_for_frame_section_assignment
        edges_for_frame_section_assignment = framePart.edges \
                                                .findAt(((3.5, 0.0, 0.0), ),
                                                        ((7.0, 0.0, 0.0), ),
                                                        ((10.5, 0.0, 0.0), ),
                                                        ((2.5, -1.5, 0.0), ),
                                                        ((5.0, -3.0, 0.0), ),
                                                        ((7.0, -3.0, 0.0), ),
                                                        ((9.0, -3.0, 0.0), ),
                                                        ((11.5, -1.5, 0.0), ),
                                                        ((6.0, -1.5, 0.0), ),
                                                        ((8.0, -1.5, 0.0), ),
                                                        ((3.5, 0.0, 1.5), ),
                                                        ((7.0, 0.0, 1.5), ),
                                                        ((10.5, 0.0, 1.5), ),
                                                        ((2.5, -1.5, 1.5), ),
                                                        ((5.0, -3.0, 1.5), ),
                                                        ((7.0, -3.0, 1.5), ),
                                                        ((9.0, -3.0, 1.5), ),
                                                        ((11.5, -1.5, 1.5), ),
                                                        ((6.0, -1.5, 1.5), ),
                                                        ((8.0, -1.5, 1.5), ),)

        global frame_region
        frame_region = \
                regionToolset.Region(edges=edges_for_frame_section_assignment)
```

```
framePart.SectionAssignment(region=frame_region,
                            sectionName='Frame Section')

global edges_for_cross_section_assignment
edges_for_cross_section_assignment = crossPart.edges \
                                     .findAt(((1.0, 0.0, 0.75), ),
                                             ((6.0, 0.0, 0.75), ),
                                             ((8.0, 0.0, 0.75), ),
                                             ((13.0, 0.0, 0.75), ),
                                             ((4.0, -3.0, 0.75), ),
                                             ((6.0, -3.0, 0.75), ),
                                             ((8.0, -3.0, 0.75), ),
                                             ((10, -3.0, 0.75), ),)


global cross_region
cross_region = \
           regionToolset.Region(edges=edges_for_cross_section_assignment)
crossPart.SectionAssignment(region=cross_region,
                            sectionName='Cross Section')


# ------------------------------------------------------------------
# Assign beam orientations

framePart.assignBeamSectionOrientation(region=frame_region,
                                       method=N1_COSINES,
                                       n1=(0.0, 0.0, 1.0))

crossPart.assignBeamSectionOrientation(region=cross_region,
                                       method=N1_COSINES,
                                       n1=(1.0, 0.0, 0.0))

# ------------------------------------------------------------------
# Create the assembly

import assembly

# Create the part instance
beamAssembly = beamModel.rootAssembly

global frameInstance
frameInstance = beamAssembly.Instance(name='Frame Instance',
                                      part=framePart, dependent=ON)

global crossInstance
crossInstance = beamAssembly.Instance(name='Cross Instance',
                                      part=crossPart, dependent=ON)

# ------------------------------------------------------------------
# Create the wire features
```

```
    vertices_for_frame_side_1 = frameInstance.vertices \
                                               .findAt(((1.0, 0.0, 0.0),),
                                                       ((4.0, -3.0, 0.0),),
                                                       ((6.0, -3.0, 0.0),),
                                                       ((10.0, -3.0, 0.0),),
                                                       ((13.0, 0.0, 0.0),),
                                                       ((6.0, 0.0, 0.0),),)

    vertices_for_frame_side_2 = frameInstance.vertices \
                                               .findAt(((1.0, 0.0, 1.5),),
                                                       ((4.0, -3.0, 1.5),),
                                                       ((6.0, -3.0, 1.5),),
                                                       ((10.0, -3.0, 1.5),),
                                                       ((13.0, 0.0, 1.5),),
                                                       ((6.0, 0.0, 1.5),),)

    vertices_for_crossbars_side_1 =  crossInstance.vertices \
                                               .findAt(((1.0, 0.0, 0.0),),
                                                       ((4.0, -3.0, 0.0),),
                                                       ((6.0, -3.0, 0.0),),
                                                       ((10.0, -3.0, 0.0),),
                                                       ((13.0, 0.0, 0.0),),
                                                       ((6.0, 0.0, 0.0),),)

    vertices_for_crossbars_side_2 = crossInstance.vertices\
                                               .findAt(((1.0, 0.0, 1.5),),
                                                       ((4.0, -3.0, 1.5),),
                                                       ((6.0, -3.0, 1.5),),
                                                       ((10.0, -3.0, 1.5),),
                                                       ((13.0, 0.0, 1.5),),
                                                       ((6.0, 0.0, 1.5),),)

    # We need to create the connectors using these points
    # the format of the WirePolyLine function is
    # WirePolyLine(points=((v1[1], v2[1]), (v1[2], v2[2]), mergeWire=OFF,
    #                                                        meshable=OFF)
    # Notice that the points argument is a tuple of point pairs, in other
    # words a tuple of tuples
    # We first create the tuples of the point pairs specifying the
    # individual connections such as (v1[1], v2[1]) and put them in a list
    # And then create a tuple of those tuples from that list using the
    # "tuple()" function

    list_of_point_tuples = []
    for i in range(len(vertices_for_frame_side_1)):
        list_of_point_tuples.append((vertices_for_frame_side_1[i],
                                     vertices_for_crossbars_side_1[i]))
        list_of_point_tuples.append((vertices_for_frame_side_2[i],
                                     vertices_for_crossbars_side_2[i]))

    tuple_of_point_tuples = tuple(list_of_point_tuples)
```

```
beamAssembly.WirePolyLine(points=tuple_of_point_tuples, mergeWire=OFF,
                                                        meshable=OFF)

# ---------------------------------
# Assign these wire features/connectors to a set that can be used later
# This is the equivalent of checking off the "Create set of wires"
# checkbox in the "Create Wire Feature" window

# This of course requires that we first find the edges that are the wire
# connectors

edges_for_connector_set = beamAssembly.edges.findAt(((1.0, 0.0, 0.0),),
                                                     ((4.0, -3.0, 0.0),),
                                                     ((6.0, -3.0, 0.0),),
                                                     ((10.0, -3.0, 0.0),),
                                                     ((13.0, 0.0, 0.0),),
                                                     ((6.0, 0.0, 0.0),),
                                                     ((1.0, 0.0, 1.5),),
                                                     ((4.0, -3.0, 1.5),),
                                                     ((6.0, -3.0, 1.5),),
                                                     ((10.0, -3.0, 1.5),),
                                                     ((13.0, 0.0, 1.5),),
                                                     ((6.0, 0.0, 1.5),),)

# Now assign them to a set
beamAssembly.Set(edges=edges_for_connector_set,
                 name='Set of connector wires')


# ---------------------------------------------------------------
# Create a connector section

beamModel.ConnectorSection(name='FrameCrossConnSect',
                           translationalType=JOIN)

# ---------------------------------
# Assign this connector section to the wire features using the set
# created earlier
conn_wire_region = beamAssembly.sets['Set of connector wires']
beamAssembly.SectionAssignment(sectionName='FrameCrossConnSect',
                               region=conn_wire_region)


# ---------------------------------
# Use constraint equations on the other two nodes

# We did not apply the JOIN condition to four of the nodes
# We will instead use an equation constraint to achieve the same effect
# on the top two
# However we wont use the equation constraint on the lower two because
# we are going to fix that edge anyway
# and having an equation constraint as well as a fixed boundary condition
```

```
    # might give an error

    # First we need to assign the nodes, both on the frame part and on the
    # crossbars, to sets

    vertex_for_framenode_1 = frameInstance.vertices \
                                            .findAt(((8.0, 0.0, 0.0),),)
    beamAssembly.Set(vertices=vertex_for_framenode_1, name='framenode1')
    vertex_for_framenode_2 = frameInstance.vertices \
                                            .findAt(((8.0, 0.0, 1.5),),)
    beamAssembly.Set(vertices=vertex_for_framenode_2, name='framenode2')
    vertex_for_crossnode_1 = crossInstance.vertices \
                                            .findAt(((8.0, 0.0, 0.0),),)
    beamAssembly.Set(vertices=vertex_for_crossnode_1, name='crossnode1')
    vertex_for_crossnode_2 = crossInstance.vertices \
                                            .findAt(((8.0, 0.0, 1.5),),)
    beamAssembly.Set(vertices=vertex_for_crossnode_2, name='crossnode2')

    # Create the equation constraints
    beamModel.Equation(name='JoinConstraint1',
                  terms=((1.0, 'crossnode1', 1), (-1.0, 'framenode1', 1)))
    beamModel.Equation(name='JoinConstraint2',
                  terms=((1.0, 'crossnode1', 2), (-1.0, 'framenode1', 2)))
    beamModel.Equation(name='JoinConstraint3',
                  terms=((1.0, 'crossnode1', 3), (-1.0, 'framenode1', 3)))
    beamModel.Equation(name='JoinConstraint4',
                  terms=((1.0, 'crossnode2', 1), (-1.0, 'framenode2', 1)))
    beamModel.Equation(name='JoinConstraint5',
                  terms=((1.0, 'crossnode2', 2), (-1.0, 'framenode2', 2)))
    beamModel.Equation(name='JoinConstraint6',
                  terms=((1.0, 'crossnode2', 3), (-1.0, 'framenode2', 3)))

    # -----------------------------------------------------------------
    # Create the step

    import step

    # Create a static general step
    beamModel.StaticStep(name='Apply Loads', previous='Initial',
                         description='Loads are applied in this step')

    # -----------------------------------------------------------------
    # Field output requests
    # Leave at defaults


    # -----------------------------------------------------------------
    # History output request
    # Leave at defaults

    return frameInstance, crossInstance, frame_region, cross_region, \
        edges_for_frame_section_assignment, edges_for_cross_section_assignment
```

```
def createLoads(self, load_crossloadedge1, load_crossloadedge2,
                      load_frameloadedge1, load_frameloadedge2,
                      load_crossload1, load_crossload2,
                      load_frameload1, load_frameload2):
    # -------------------------------------------------------------------
    # Apply loads

    beamModel.LineLoad(name='CrossLoad1', createStepName='Apply Loads',
                       region=(load_crossloadedge1,), comp2=-load_crossload1)
    beamModel.LineLoad(name='CrossLoad2', createStepName='Apply Loads',
                       region=(load_crossloadedge2,), comp2=-load_crossload2)
    beamModel.LineLoad(name='FrameLoad1', createStepName='Apply Loads',
                       region=(load_frameloadedge1,), comp2=-load_frameload1)
    beamModel.LineLoad(name='FrameLoad2', createStepName='Apply Loads',
                       region=(load_frameloadedge2,), comp2=-load_frameload2)



def createBoundaryMeshJob(self):
    # -------------------------------------------------------------------
    # Apply boundary conditions

    global beamModel
    global framePart
    global crossPart
    global frameInstance
    global crossInstance
    global frame_region
    global cross_region
    global edges_for_frame_section_assignment
    global edges_for_cross_section_assignment

    frame_edges_for_bc = frameInstance.edges.findAt(((5.0,-3.0,0.0),),
                                                    ((7.0,-3.0,0.0),),
                                                    ((9.0,-3.0,0.0),),
                                                    ((5.0,-3.0,1.5),),
                                                    ((7.0,-3.0,1.5),),
                                                    ((9.0,-3.0,1.5),),)

    cross_edges_for_bc = crossInstance.edges.findAt(((4.0,-3.0,0.75),),
                                                    ((6.0,-3.0,0.75),),
                                                    ((8.0,-3.0,0.75),),
                                                    ((10.0,-3.0,0.75),),)

    edges_for_bc = frame_edges_for_bc + cross_edges_for_bc
    bc_region = regionToolset.Region(edges=edges_for_bc)
```

```
        beamModel.DisplacementBC(name='FixBottom', createStepName='Initial',
                                 region=bc_region, u1=SET, u2=SET, u3=SET,
                                 ur1=UNSET, ur2= UNSET, ur3=UNSET,
                                 amplitude=UNSET, distributionType=UNIFORM,
                                 fieldName='', localCsys=None)

        # -----------------------------------------------------------------
        # Create the mesh

        import mesh

        frame_mesh_region = frame_region
        frame_edges_for_meshing = edges_for_frame_section_assignment
        frame_mesh_element_type=mesh.ElemType(elemCode=B31, elemLibrary=STANDARD)
        framePart.setElementType(regions=frame_mesh_region,
                                 elemTypes=(frame_mesh_element_type, ))
        framePart.seedEdgeByNumber(edges=frame_edges_for_meshing, number=4)
        framePart.generateMesh()


        cross_mesh_region = cross_region
        cross_edges_for_meshing = edges_for_cross_section_assignment
        cross_mesh_element_type=mesh.ElemType(elemCode=B31, elemLibrary=STANDARD)
        crossPart.setElementType(regions=cross_mesh_region,
                                 elemTypes=(cross_mesh_element_type, ))
        crossPart.seedEdgeByNumber(edges=cross_edges_for_meshing, number=4)
        crossPart.generateMesh()

        # -----------------------------------------------------------------
        # Create the job

        import job

        mdb.Job(name='BeamFrameAnalysisJob', model='Beam Frame', type=ANALYSIS,
                explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE,
                description='Bending of loaded beam frame',
                parallelizationMethodExplicit=DOMAIN, multiprocessingMode=DEFAULT,
                numDomains=1, userSubroutine='', numCpus=1, memory=50,
                memoryUnits=PERCENTAGE, scratch='', echoPrint=OFF, modelPrint=OFF,
                contactPrint=OFF, historyPrint=OFF)



    def saveModel(self, save_fileName, save_fileReadOnlyBool, save_dirName):
        # -----------------------------------------------------------------
        # Save the model
        # First check to see if the user chose an actual path to save it
        if save_fileName != '':
            mdb.saveAs(pathName=save_fileName)
        # We wont actually do anything with the read only boolean or the directory
        # in this example, but the fields were included for demonstration
        # purposes.

        print 'The filename chosen is: ' + save_fileName

        if save_fileReadOnlyBool:
            print 'Read Only is ON'
        else:
            print 'Read Only is OFF'

        print 'The directory chosen is:' + save_dirName


    def runJob(self):
        #-----------------------------------------------------------------
        # Run the job

        mdb.jobs['BeamFrameAnalysisJob'].submit(consistencyChecking=OFF)

        # do not return control till job is finished running
        mdb.jobs['BeamFrameAnalysisJob'].waitForCompletion()



def callStep1(selected_material,name,density,youngs,poissons):

    global step_counter

    if step_counter !=0 :
        print '!Error - You must call steps in sequence. The next step ' +\
              'is Step ' + str(step_counter+1)
    else:
        callStep1_successful = \
               BeamKernel().createModelPartMaterial(selected_material, name,
                                                     density,youngs,poissons)

        if callStep1_successful:
            step_counter = 1
        else:
            step_counter = 0


def callStep2(profile=1, l_1=0, l_2=0, h_1=0, h_2=0, b1_1=0, b1_2=0, b2_1=0,
              b2_2=0, t1_1=0, t1_2=0, t2_1=0, t2_2=0, t3_1=0, t3_2=0, a_1=0,
              a_2=0, b_1=0, b_2=0, t_1=0, t_2=0, r_1=0, r_2=0):

    global step_counter

    if step_counter !=1 :
        print '!Error - You must call steps in sequence. The next step ' + \
            'is Step ' + str(step_counter+1)
    else:
        BeamKernel().createProfiles(profile, l_1, l_2, h_1, h_2, b1_1, b1_2,
                                    b2_1, b2_2, t1_1, t1_2, t2_1, t2_2, t3_1,
                                    t3_2, a_1, a_2, b_1, b_2, t_1, t_2, r_1, r_2)
        BeamKernel().createSectionAssemblySteps()
        step_counter = 2


def callStep3(crossloadedge1, crossloadedge2, frameloadedge1, frameloadedge2,
              crossload1, crossload2, frameload1, frameload2):

    global step_counter

    if step_counter !=2 :
        print '!Error - You must call steps in sequence. The next step is ' + \
              'Step ' + str(step_counter+1)
    else:
        BeamKernel().createLoads(crossloadedge1, crossloadedge2, frameloadedge1,
                                 frameloadedge2, crossload1, crossload2,
                                 frameload1, frameload2)
        BeamKernel().createBoundaryMeshJob()
        step_counter = 3

def callStep4(fileName, fileReadOnlyBool, dirName):

    global step_counter

    if step_counter !=3 :
        print '!Error - You must call steps in sequence. The next step is ' + \
              'Step ' + str(step_counter+1)
    else:
        BeamKernel().saveModel(fileName, fileReadOnlyBool, dirName)
        step_counter = 4

def callStep5():

    global step_counter

    if step_counter !=4 :
        print '!Error - You must call steps in sequence. The next step is ' + \
              'Step ' + str(step_counter+1)
    else:
        BeamKernel().runJob()
        step_counter = 5
        print 'Analysis Complete !!!'
```

A number of global variables exist in this script. These variables point to objects in the model. They are created inside the methods, but we need to be able to access them from other methods as well hence they have been made global in scope.

The entire script is interspersed with comments. These are descriptive, and combined with the fact that a lot of this script was copied from the beam frame analysis chapter, you will easily figure out much of this script. We shall only focus on the new or different bits.

```
from abaqus import *
from abaqusConstants import *
import regionToolset
```

The first thing you notice is that there is no "*from abaqusGui import* *" statement. This is because this script is a kernel script, not a GUI script. Instead we have "*from abaqus import* *" which we include in all of our kernel scripts. You are already aware that if you include both import statements in the same script Abaqus will throw an error.

```
step_counter = 0
```

A variable **step_counter** is initialized to zero. This variable will keep track of which step of the vertical application the user is on. If he tries to execute steps out of sequence, the program will detect this immediately and inform him of his mistake. By placing **step_counter** outside of the class we make it a global variable which can be accessed by other methods of the script outside of the **BeamKernel()** class.

```
class BeamKernel():
```

All of the functionality copied from the beam frame analysis script of Chapter 9 is included in the **BeamKernel** class with some minor modifications. The script has been split up into many functions such as **createModelPartMaterial()**, **createProfiles()**, **createSectionAssemblySteps()** and so on, which are all part of the **BeamKernel** class.

```
    def createModelPartMaterial(self, mat_selected, mat_name, mat_density,
                                                            mat_youngs, mat_poissons):

        global material_name

        # The user may have selected steel or aluminum in which case the rest of
        # the fields will be left blank and the parameters will have default
        # values
        if mat_selected == 1 :
            material_name = 'AISI 1005 Steel'
            mat_density = 7800.0
            mat_youngs = 200E9
            mat_poissons = 0.3
        elif mat_selected == 2 :
            material_name = 'Aluminum 2024-T3'
            mat_density = 2770.0
            mat_youngs = 73.1E9
            mat_poissons = 0.33
        else:
            material_name = mat_name

        # if you try to provide a youngs modulus and poissons ratio both equal to
        # zero Abaqus throws an error because in the elasticity table you've
        # given it a row of zeros
        if mat_youngs==0 and mat_poissons==0 :
            print 'Young\s modulus and Poisson\s ratio cant both be zero'
            # Since we exit Step 1 we reset the global counter so the user can
            # perform Step 1 again
            return False
```

The **createModelPartMaterial()** method accepts a number of arguments which will be obtained from the 'Step 1' dialog box of the GUI. These are **mat_selected, mat_name**, **mat_density**, **mat_youngs** and **mat_poissons**. **mat_selected** can be 1, 2 or 3, representing steel, aluminum, and a new user defined material. **mat_name, mat_density**, **mat_youngs** and **mat_poissons** are the name, density, Young's modulus and Poisson's ratio of the material.

A global variable **material_name** is created. It is made global so that it can be accessed by other methods in the script such as **createSectionAssemblySteps()**.

```
session.viewports['Viewport: 1'].setValues(displayedObject=None)

       # -----------------------------------------------------------------
       # Create the model

                              ...

       # -----------------------------------------------------------------
       # Create the parts

       import sketch
       import part

       # -----------------------------------------------------------------
       # Create the frame

                              ...

       # -----------------------------------------------------------------
       # a) Create one side of the frame

                              ...
```

```
        # -----------------------------------------------------------------
        # b) Create other side of the frame

                                  ...
```

These statements are copied from the beam frame analysis script which was created and explained in great detail in Chapter 9.

```
        # -----------------------------------------------------------------
        # Create material

        import material

        # Create material by assigning mass density, youngs modulus and poissons
        # ratio
        beamMaterial = beamModel.Material(name=material_name)

        # If material density is 0 (meaning user did not enter a value and it
        # defaulted to 0), we will not assign a density
        if mat_density !=0 :
            beamMaterial.Density(table=((mat_density, ),))

        beamMaterial.Elastic(table=((mat_youngs, mat_poissons), ))

        return True
```

These statements are also copied from the beam frame analysis script and modified. The only difference is that they check to see if a density value was assigned by the user, and if not then no density will be specified when creating the material. The method returns True once it is complete.

```
    def createProfiles(self, cs_profile, cs_l_1, cs_l_2, cs_h_1, cs_h_2, cs_b1_1,
                       cs_b1_2, cs_b2_1, cs_b2_2, cs_t1_1, cs_t1_2, cs_t2_1,
                       cs_t2_2, cs_t3_1, cs_t3_2, cs_a_1, cs_a_2, cs_b_1, cs_b_2,
                       cs_t_1, cs_t_2, cs_r_1, cs_r_2):
    # ---------------------------------------------------------------------
    # Create profiles

        if cs_profile==1:
            beamModel.IProfile(name='FrameProfile', l=cs_l_1, h=cs_h_1,
                               b1=cs_b1_1, b2=cs_b2_1, t1=cs_t1_1, t2=cs_t2_1,
                               t3=cs_t3_1)
            beamModel.IProfile(name='CrossProfile', l=cs_l_2, h=cs_h_2,
                               b1=cs_b1_2, b2=cs_b2_2, t1=cs_t1_2, t2=cs_t2_2,
                               t3=cs_t3_2)
        elif cs_profile == 2:
            beamModel.BoxProfile(name='FrameProfile', b=cs_b_1, a=cs_a_1,
                                                uniformThickness=ON, t1=cs_t_1)
            beamModel.BoxProfile(name='CrossProfile', b=cs_b_2, a=cs_a_2,
                                                uniformThickness=ON, t1=cs_t_2)
        else :
            beamModel.CircularProfile(name='FrameProfile', r=cs_r_1)
            beamModel.CircularProfile(name='CrossProfile', r=cs_r_2)
```

The **createProfiles()** method creates the profiles. It creates an 'I', 'Box' or 'Circular' profile depending on the users choice using the **IProfile()**, **BoxProfile()** and **CircularProfile()** methods provided by the Abaqus Scripting Interface. The method accepts a large number of parameters since some of these profiles, especially the 'I', require you to define many dimensions.

```
    def createSectionAssemblySteps(self):
        # ----------------------------------------------------------------
        # Create sections and assign frame and crosbracing to them

                          ...

        # ----------------------------------------------------------------
        # Assign beam orientations

                          ...

        # ----------------------------------------------------------------
        # Create the assembly
                          ...

        # ----------------------------------------------------------------
        # Create the wire features

                          ...

        # ---------------------------------
        # Assign these wire features/connectors to a set that can be used later

                          ...

        # ----------------------------------------------------------------
        # Create a connector section

                          ...

        # ---------------------------------
        # Use constraint equations on the other two nodes

                          ...
        # ------------------------------------------------------------------
        # Create the step

        import step

                                   ...

        # ------------------------------------------------------------------
        # Field output requests
        # Leave at defaults


        # ------------------------------------------------------------------
        # History output request
        # Leave at defaults

        return frameInstance, crossInstance, frame_region, cross_region, \
            edges_for_frame_section_assignment, edges_for_cross_section_assignment
```

createSectionAssemblySteps() creates the sections, assembles the parts and creates the steps of the simulation.

```
    def createLoads(self, load_crossloadedge1, load_crossloadedge2,
                          load_frameloadedge1, load_frameloadedge2,
                          load_crossload1, load_crossload2,
                          load_frameload1, load_frameload2):
        # ------------------------------------------------------------------
        # Apply loads

        beamModel.LineLoad(name='CrossLoad1', createStepName='Apply Loads',
                           region=(load_crossloadedge1,), comp2=-load_crossload1)
        beamModel.LineLoad(name='CrossLoad2', createStepName='Apply Loads',
                           region=(load_crossloadedge2,), comp2=-load_crossload2)
        beamModel.LineLoad(name='FrameLoad1', createStepName='Apply Loads',
                           region=(load_frameloadedge1,), comp2=-load_frameload1)
        beamModel.LineLoad(name='FrameLoad2', createStepName='Apply Loads',
                           region=(load_frameloadedge2,), comp2=-load_frameload2)
```

createLoads() creates line loads on the 4 beam members selected by the user during ‘Step 3’.

```
    def createBoundaryMeshJob(self):
        # ------------------------------------------------------------------
        # Apply boundary conditions
                      ...


        # ------------------------------------------------------------------
        # Create the mesh
```

```
                ...


        # ----------------------------------------------------------------
        # Create the job

                ...
```

createBoundaryMeshJob() creates the boundary conditions, mesh and analysis job. It does not however run the job.

```
    def saveModel(self, save_fileName, save_fileReadOnlyBool, save_dirName):
        # ----------------------------------------------------------------
        # Save the model
        # First check to see if the user chose an actual path to save it
        if save_fileName != '':
            mdb.saveAs(pathName=save_fileName)

        # We wont actually do anything with the read only boolean or the directory
        # in this example, but the fields were included for demonstration
        # purposes.

        print 'The filename chosen is: ' + save_fileName

        if save_fileReadOnlyBool:
            print 'Read Only is ON'
        else:
            print 'Read Only is OFF'

        print 'The directory chosen is:' + save_dirName
```

saveModel() saves the file using **mdb.saveAs()**. It obtains the file name from 'Step 4'. While it detects the presence of the 'read on' Boolean indicating whether a file is to be treated as read only or not, it does not actually do anything with this information. I only wished to demonstrate that it is possible to obtain this information. Feel free to tinker with the code to enhance it.

```
    def runJob(self):
        #----------------------------------------------------------------
        # Run the job

        mdb.jobs['BeamFrameAnalysisJob'].submit(consistencyChecking=OFF)

        # do not return control till job is finished running
        mdb.jobs['BeamFrameAnalysisJob'].waitForCompletion()
```

runJob() submits the model to the solver to run the analysis.

```
def callStep1(selected_material,name,density,youngs,poissons):

    global step_counter

    if step_counter !=0 :
        print '!Error - You must call steps in sequence. The next step ' +\
              'is Step ' + str(step_counter+1)
    else:
        callStep1_successful = \
               BeamKernel().createModelPartMaterial(selected_material, name,
                                                           density,youngs,poissons)

        if callStep1_successful:
            step_counter = 1
        else:
            step_counter = 0
```

This method is called by **Step1Form** when the user presses **OK** in **Step1DB** (the dialog box for 'Step 1'). The entered values and the options checked by the user are passed to it as parameters. The global variable **step_counter** is examined, and if its value is not zero, meaning that other steps have already been executed, it tells the user to follow the steps in sequence. If not, it calls the **createModelPartMaterial()** method of the **BeamKernel** class we saw a little earlier, after which it sets **step_counter** to 1.

```
def callStep2(profile=1, l_1=0, l_2=0, h_1=0, h_2=0, b1_1=0, b1_2=0, b2_1=0,
              b2_2=0, t1_1=0, t1_2=0, t2_1=0, t2_2=0, t3_1=0, t3_2=0, a_1=0,
              a_2=0, b_1=0, b_2=0, t_1=0, t_2=0, r_1=0, r_2=0):

    global step_counter

    if step_counter !=1 :
        print '!Error - You must call steps in sequence. The next step ' + \
              'is Step ' + str(step_counter+1)
    else:
        BeamKernel().createProfiles(profile, l_1, l_2, h_1, h_2, b1_1, b1_2,
                                    b2_1, b2_2, t1_1, t1_2, t2_1, t2_2, t3_1,
                                    t3_2, a_1, a_2, b_1, b_2, t_1, t_2, r_1, r_2)
        BeamKernel().createSectionAssemblySteps()
        step_counter = 2
```

This method is called by **Step2Form** when the user presses **OK** in **Step2DB**. The values input and the options checked by the user are passed to it as parameters. After examining the global variable **step_counter** and ensuring that the step has been executed at the correct point in the sequence it calls the method **createProfiles()** of the **BeamKernel** class followed by **createSectionAssemblySteps()**.

```
def callStep3(crossloadedge1, crossloadedge2, frameloadedge1, frameloadedge2,
              crossload1, crossload2, frameload1, frameload2):

    global step_counter

    if step_counter !=2 :
        print '!Error - You must call steps in sequence. The next step is ' + \
              'Step ' + str(step_counter+1)
    else:
        BeamKernel().createLoads(crossloadedge1, crossloadedge2, frameloadedge1,
                                 frameloadedge2, crossload1, crossload2,
                                 frameload1, frameload2)
        BeamKernel().createBoundaryMeshJob()
        step_counter = 3
```

This method is called by **Step3Procedure** when the user presses **OK** in **Step3DB**. The beam members selected by the user in the viewport and the values input in the dialog box are passed to it as parameters. After examining the global variable **step_counter**, and ensuring that the step has been executed in sequence, it calls the **createBoundaryMeshJob()** method of the **BeamKernel** class.

```
def callStep4(fileName, fileReadOnlyBool, dirName):

    global step_counter

    if step_counter !=3 :
        print '!Error - You must call steps in sequence. The next step is ' + \
              'Step ' + str(step_counter+1)
    else:
        BeamKernel().saveModel(fileName, fileReadOnlyBool, dirName)
        step_counter = 4
```

This method is called by **Step4Form** when the user specifies a location to save the model in **Step4DB**. After examining the global variable **step_counter** and ensuring that the step has been executed in sequence, it calls the **saveModel()** method of the **BeamKernel** class.

```
def callStep5():

    global step_counter

    if step_counter !=4 :
        print '!Error - You must call steps in sequence. The next step is ' + \
                'Step ' + str(step_counter+1)
    else:
        BeamKernel().runJob()
        step_counter = 5
        print 'Analysis Complete !!!'
```

This method is called by **Step5Form** when the user initiates Step5. After examining the global variable **step_counter** and ensuring that the step has been executed in sequence, it calls the **runJob()** method of the **BeamKernel** class. Once the analysis is complete it prints a message to the message area.

21.4.2 Beam Application Startup Script

This script is contained in **beamCaeApp.py**. It is the application startup script - it creates the application (**AFXApp**) - and calls for the creation of the main window.

```
# ********************************************************************************
# Custom Beam Frame Analysis GUI Application
# This script creates and launches the Beam GUI application
# ********************************************************************************

from abaqusGui import AFXApp
import sys
# import class that will create the main window
from beamCaeMainWindow import BeamCaeMainWindow

# Initialize application object
# In AFXApp, appName and vendorName are displayed if productName is set to ''
# otherwise productName is displayed.
app = AFXApp(appName='ABAQUS/CAE',
             vendorName='ABAQUS, Inc.',
             productName='Custom GUI Application',
             majorNumber=1,
             minorNumber=1,
             updateNumber=1,
             prerelease=False)
app.init(sys.argv)

# Construct main window
BeamCaeMainWindow(app)

# Create application
app.create()

# Run application
app.run()
```

This script is a slightly modified version of the custom CAE application startup script written in the previous chapter, and should require no further explanation. Refer to section 20.6.1 of the previous chapter for information on writing a CAE application startup script.

21.4.3 Beam Application Main Window

This script is contained in **beamCaeMainWindow.py**. It creates the main window (**AFXMainWidnow**), and removes the modules that will not be part of our application while registering the one created by us. It also registers our custom toolsets and removes the model tree.

```
# ***********************************************************************************
# Custom Beam Frame Analysis GUI Application
# This script defines the main window of the Beam GUI application
# ***********************************************************************************

from abaqusGui import *
from sessionGui import *
from canvasGui import CanvasToolsetGui
from viewManipGui import ViewManipToolsetGui
from customToolboxButtonsGui import CustomToolboxButtonsGui

# Define the class
class BeamCaeMainWindow(AFXMainWindow):

    def __init__(self, app, windowTitle=''):

        AFXMainWindow.__init__(self, app, windowTitle)

        # Register toolsets
        self.registerToolset(ViewManipToolsetGui(), GUI_IN_MENUBAR|GUI_IN_TOOLBAR)

        # Register the custom toolset for adding custom toolbox buttons
        self.registerToolset(CustomToolboxButtonsGui(), GUI_IN_TOOLBOX)

        # Create and register our custom help toolset with our own copyright info
        # (Modify Help > About Abaqus... dialog will include custom copyright
        # information)
        customHelpToolsetGui = HelpToolsetGui()
        abaqus_product_name = getAFXApp().getProductName()
        major_version_no, minor_version_no, update_no = \
                                                  getAFXApp().getVersionNumbers()
        is_this_prerelease = getAFXApp().getPrerelease()
        if is_this_prerelease:
            version = '%s Version %s.%s-PRE%s' % (abaqus_product_name,
                                                  major_version_no,
                                                  minor_version_no,
                                                      update_no)
        else:
            version = '%s Version %s.%s-%s' % (abaqus_product_name,
                                               major_version_no,
                                               minor_version_no,
                                               update_no)
        custom_title = 'Beam Frame GUI Application'
        custom_information = 'Python Scripts for Abaqus - Learn by Example ' + \
               '\n Gautam Puri \n Copyright 2011' + '\n Running Abaqus ' + version
        customHelpToolsetGui.setCustomCopyrightStrings(custom_title,
                                                       custom_information)
        custom_icon=afxCreateIcon('icon_bookcover.png')
        customHelpToolsetGui.setCustomLogoIcon(custom_icon)
        self.registerHelpToolset(customHelpToolsetGui,
                                 GUI_IN_MENUBAR|GUI_IN_TOOLBAR)

        # Register our custom module which resides in the script
        # file beamModuleGui.py
        self.registerModule('Beam Module',       'beamModuleGui')
```

This script is a modified version of the main window script written in the previous chapter and should require no further explanation. All of the standard modules, and most of the standard toolsets, have been removed. Hence the application window has very few toolbars and even lacks a model tree. Refer to section 20.6.2 of the previous chapter for information on registering modules and toolsets.

21.4.4 Custom Persistant toolset

This script is contained in **customToolboxButtonsGui.py**. In it we create 5 new persistent toolset buttons labeled 'Step 1', 'Step 2', 'Step 3', 'Step 4' and 'Step 5'. There is also a 6th button called 'Help'.

Some of these buttons launch dialog boxes, others have IDs that map to functions thanks to **FXMAPFUNC()**, using the syntax you learned in the previous chapter.

```
# ********************************************************************************
# Custom Beam Frame Analysis GUI Application
# This script adds buttons to the viewport
# ********************************************************************************


from abaqusGui import *
from sessionGui import CanvasToolsetGui
from step1Form import Step1Form
from step2Form import Step2Form
from step3Procedure import Step3Procedure
from step4Form import Step4Form


class CustomToolboxButtonsGui(AFXToolsetGui):

    [
        ID_SAVE,
        ID_RUN,
        ID_HELP,
        ID_WARNING,
        ID_LAST
    ] = range(AFXToolsetGui.ID_LAST, AFXToolsetGui.ID_LAST+5)


    def __init__(self):

        print 'CustomToolboxButtonsGui initialization method called.'

        # Construct base class.
        AFXToolsetGui.__init__(self, 'Test Toolset')

        FXMAPFUNC(self, SEL_COMMAND, self.ID_SAVE, CustomToolboxButtonsGui.onSave)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_RUN, CustomToolboxButtonsGui.onRun)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_HELP, CustomToolboxButtonsGui.onHelp)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_WARNING,
                                     CustomToolboxButtonsGui.onWarning)

        # Toolbox buttons
        toolbox_group_1 = AFXToolboxGroup(self)
        AFXToolButton(toolbox_group_1,
                      ' Step 1: Material \t Create Model, Part and define ' + \
                                                              'the Material',
                      None, Step1Form(self),
                       AFXMode.ID_ACTIVATE)
        AFXToolButton(toolbox_group_1,
                      ' Step 2: Profile \t Select the profile and create ' + \
                                                          'Assembly and Steps',
                      None,
                      Step2Form(self),
                      AFXMode.ID_ACTIVATE)
        AFXToolButton(toolbox_group_1,
                      ' Step 3: Loads \t Define the loads, and create ' + \
                                        'boundary conditions, mesh and job',
                      None,
                      Step3Procedure(self),
                      AFXMode.ID_ACTIVATE)
        AFXToolButton(toolbox_group_1,
                      ' Step 4: Save the Model',
                      None,
                      self,
                  self.ID_SAVE)
    AFXToolButton(toolbox_group_1,
                  ' Step 5: Run the job',
                  None,
                  self,
                  self.ID_RUN)

    toolbox_group_2 = AFXToolboxGroup(self)
    AFXToolButton(toolbox_group_2, ' Help \t Display help', None, self,
                                                          self.ID_HELP)


# Initialize the necessary module or toolset in the kernel using
# getKernelInitializationCommand
# This method is called by the module manager when a GUI module is loaded for
# the first time.
# This method is empty in the base class implementation
# The analyst may write a method that returns the proper command that will
# import the appropriate modules on the kernel side.
# Basically this allows you to 'import' a kernel module into the program
# which you would not otherwise be able to do in this file since it has GUI
# commands
# We need to do this because in step1Form.py we use the statement
# self.cmd = AFXGuiCommand(self, 'callStep1', 'beamKernel')
# which calls callStep1() in beamKernel.py, hence beamKernel needs to be
# defined/imported
def getKernelInitializationCommand(self):

    return 'import beamKernel'



def onHelp(self, sender, sel, ptr):
    mainWindow = getAFXApp().getAFXMainWindow()
    showAFXInformationDialog(mainWindow,
                             'Click on each Step in sequence using the ' + \
                                           'buttons in the toolbox. \n' + \
                             'Step 1 : Supply material properties \n' + \
                             'Step 2: Provide profile dimensions \n' + \
                             'Step 3: Define Loading \n' + \
                             'Step 4: Save the model \n' + \
                             'Step 5: Run the Analysis')
    getAFXApp().getAFXMainWindow().writeToMessageArea('Script created by ' + \
        'Gautam Puri, author of Python Scripts for Abaqus - Learn by Example')



def onSave(self, sender, sel, ptr):
    # Ask user if he is sure he wishes to save
    warning_owner = getAFXApp().getAFXMainWindow()
    showAFXWarningDialog(owner=warning_owner,
                         message='Are you sure you wish to save this model?',
                            buttonIds=AFXDialog.YES | AFXDialog.NO,
                            tgt=self,
                            sel=self.ID_WARNING)


    def onWarning(self, sender, sel, ptr):
        if sender.getPressedButtonId() == AFXDialog.ID_CLICKED_YES:
            # Launch Step 4 dialog box
            # To do this we must launch the form mode Step4Form
            # Note that this time we are not launching the form mode (that
            # displays the dialog box) using a GUI button as we have done for
            # Step 1, 2 and 3
            # Instead we are launching it manually here. Hence we will first
            # create a form mode object, and then call the activate() method
            fm = Step4Form(self)
            Step4Form.activate(fm)


        elif sender.getPressedButtonId() == AFXDialog.ID_CLICKED_NO:
            self.deactivate()

        return True

    def onRun(self, sender, sel, ptr):
        # To issue a kernel command directly from the GUI we can use the
        # sendCommand() method
        # To the sendCommand() method we pass the statements that should be
        # executed
        # We separate multiple statements with a \n as is done here. Note that
        # you cannot leave a space after the \n because that means there is a
        # space in front of the next statement, and Python being will object to
        # this wrong indentation
        sendCommand('beamKernel.callStep5()')
```

This script is very similar to the one written for creating custom toolbox buttons in the previous chapter. Let's focus on the new concepts here, refer to section 20.6.4 for more details.

```
from abaqusGui import *
from sessionGui import CanvasToolsetGui
from step1Form import Step1Form
from step2Form import Step2Form
from step3Procedure import Step3Procedure
from step4Form import Step4Form

class CustomToolboxButtonsGui(AFXToolsetGui):
```

Aside from the usual **import** statements (**abaqusGUI** and **sessionGui**), the form and procedure modes that will be launched by clicking on the toolbox buttons are also imported. The class is derived from **AFXToolsetGui** as expected.

```
    [
        ID_SAVE,
        ID_RUN,
        ID_HELP,
        ID_WARNING,
        ID_LAST
    ] = range(AFXToolsetGui.ID_LAST, AFXToolsetGui.ID_LAST+5)



    def __init__(self):

        print 'CustomToolboxButtonsGui initialization method called.'

        # Construct base class.
        AFXToolsetGui.__init__(self, 'Test Toolset')

        FXMAPFUNC(self, SEL_COMMAND, self.ID_SAVE, CustomToolboxButtonsGui.onSave)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_RUN, CustomToolboxButtonsGui.onRun)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_HELP, CustomToolboxButtonsGui.onHelp)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_WARNING,
                                     CustomToolboxButtonsGui.onWarning)

        # Toolbox buttons
        toolbox_group_1 = AFXToolboxGroup(self)
        AFXToolButton(toolbox_group_1,
                      ' Step 1: Material \t Create Model, Part and define ' + \
                                                              'the Material',
                      None, Step1Form(self),
                       AFXMode.ID_ACTIVATE)
        AFXToolButton(toolbox_group_1,
                      ' Step 2: Profile \t Select the profile and create ' + \
                                                          'Assembly and Steps',
                      None,
                      Step2Form(self),
                      AFXMode.ID_ACTIVATE)
        AFXToolButton(toolbox_group_1,
                      ' Step 3: Loads \t Define the loads, and create ' + \
                                    'boundary conditions, mesh and job',
                      None,
                      Step3Procedure(self),
                      AFXMode.ID_ACTIVATE)
        AFXToolButton(toolbox_group_1,
                      ' Step 4: Save the Model',
                      None,
                      self,
                     self.ID_SAVE)
        AFXToolButton(toolbox_group_1,
                     ' Step 5: Run the job',
                     None,
                     self,
                     self.ID_RUN)

        toolbox_group_2 = AFXToolboxGroup(self)
        AFXToolButton(toolbox_group_2, ' Help \t Display help', None, self,
                                                                  self.ID_HELP)
```

All this should also look familiar to you from the previous chapter. A number of 'IDs' are established which are associated with the buttons for 'Step 4', 'Step 5' and 'Help', and are used by **FXMAPFUNC()** to execute the appropriate method. 'Step 1', 'Step2' and 'Step 3' on the other hand use **AFXMode.ID_ACTIVATE** to call the **activate()** method of the corresponding form or procedure mode with syntax you've seen before.

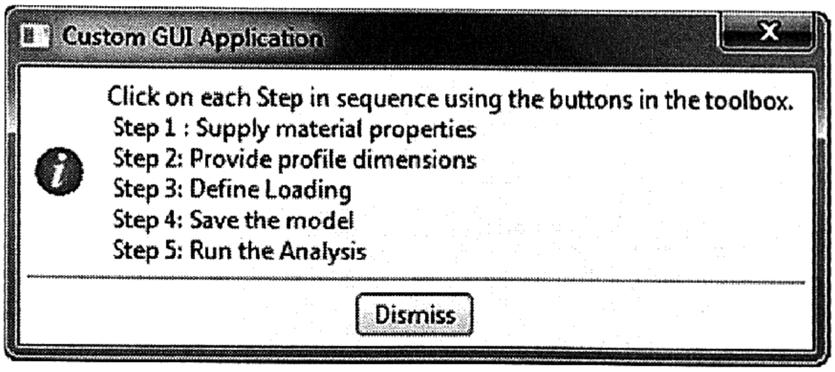
The only ID that does not appear to be assigned to a button here is **ID_WARNING**. This is in fact used later in the **onSave()** method, which launches a warning dialog box, and uses this ID to map to the correct function when the user clicks **OK**. We'll revisit it when we get to **onSave()**.

```
    # Initialize the necessary module or toolset in the kernel using
    # getKernelInitializationCommand
    # This method is called by the module manager when a GUI module is loaded for
    # the first time.
    # This method is empty in the base class implementation
    # The analyst may write a method that returns the proper command that will
    # import the appropriate modules on the kernel side.
    # Basically this allows you to 'import' a kernel module into the program
    # which you would not otherwise be able to do in this file since it has GUI
    # commands
    # We need to do this because in step1Form.py we use the statement
    # self.cmd = AFXGuiCommand(self, 'callStep1', 'beamKernel')
    # which calls callStep1() in beamKernel.py, hence beamKernel needs to be
    # defined/imported
    def getKernelInitializationCommand(self):

        return 'import beamKernel'
```

Quite frankly, the paragraph of comments preceding these two statements describe precisely what they accomplish. To make it perfectly clear, the method **getKernelInitializationCommand()** does not need to be defined in a GUI script, which is why we didn't use it in the previous chapter. However this time we would like our form modes to send actual commands to the kernel script (**beamKernel.py**) therefore the kernel module needs to be imported. It is not possible to import a kernel module into a GUI script (just as it is not possible to have "*from abaqus import* *" and "*from abaqusGui import* *" statements in the same script). **getKernelInitializationCommand()** helps us get around this issue.

```
    def onHelp(self, sender, sel, ptr):
        mainWindow = getAFXApp().getAFXMainWindow()
        showAFXInformationDialog(mainWindow,
                                 'Click on each Step in sequence using the ' + \
                                            'buttons in the toolbox. \n' + \
                                 'Step 1 : Supply material properties \n' + \
                                 'Step 2: Provide profile dimensions \n' + \
                                 'Step 3: Define Loading \n' + \
                                 'Step 4: Save the model \n' + \
                                 'Step 5: Run the Analysis')
        getAFXApp().getAFXMainWindow().writeToMessageArea('Script created by ' + \
            'Gautam Puri, author of Python Scripts for Abaqus - Learn by Example')
```

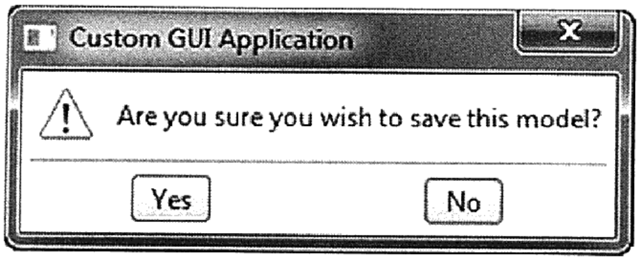
onHelp() is called when the user clicks the 'Help' toolbox button. It uses **showAFXInformationDialog()** to launch an information dialog box. This technique was described in the previous chapter in section 20.6.5.



Click on each Step in sequence using the buttons in the toolbox.
Step 1 : Supply material properties
Step 2: Provide profile dimensions
Step 3: Define Loading
Step 4: Save the model
Step 5: Run the Analysis

```
    def onSave(self, sender, sel, ptr):
        # Ask user if he is sure he wishes to save
        warning_owner = getAFXApp().getAFXMainWindow()
        showAFXWarningDialog(owner=warning_owner,
                             message='Are you sure you wish to save this model?',
                             buttonIds=AFXDialog.YES | AFXDialog.NO,
                             tgt=self,
                             sel=self.ID_WARNING)
```

showAFXWarningDialog() is another type of message dialog available for use in GUI scripts. It is used to obtain user assistance in response to some condition. Warning dialog boxes have a warning symbol (yellow triangle with an exclamation point). They can contain **Yes**, **No** and **Cancel** buttons. Their title bar contains the application name.



showAFXWarningDialog() accepts 5 arguments – **owner**, **message**, **buttonIds**, **tgt** and **sel**. **owner** is the window over which to center the dialog box, **message** is the text String to display as the message, and **buttonIds** is a series of ID's of action area buttons to be created (the options are **YES**, **NO** and **CANCEL**). Here we have used **YES** and **NO**. Note the use of the word **AFXDialog** preceding **YES** and **NO**. **tgt** and **sel** are the message target and message ID. Here we set the toolset itself as the target, and **ID_WARNING** as the messageID. Recall that we had created a unique identifier, **ID_WARNING**, near the beginning of the script, and **FXMAPFUNC()** maps it to the **onWarning()** function. So if the user clicks either button, the **onWarning()** method will be executed.

```
    def onWarning(self, sender, sel, ptr):
        if sender.getPressedButtonId() == AFXDialog.ID_CLICKED_YES:
            # Launch Step 4 dialog box
            # To do this we must launch the form mode Step4Form
            # Note that this time we are not launching the form mode (that
            # displays the dialog box) using a GUI button as we have done for
            # Step 1, 2 and 3
            # Instead we are launching it manually here. Hence we will first
            # create a form mode object, and then call the activate() method
            fm = Step4Form(self)
            Step4Form.activate(fm)


        elif sender.getPressedButtonId() == AFXDialog.ID_CLICKED_NO:
            self.deactivate()

        return True
```

One of the parameters automatically passed to **onWarning()** by **FXMAPFUNC()** (when the user clicks **Yes** in the warning dialog box) is **sender**, which is the warning dialog box. Having a handle to the warning dialog box, we can use one of its methods –

getPressedButtonId(). This method allows us to query which button was clicked by the user. Since **onWarning()** is called no matter which button is clicked, we need to determine if the user clicked **Yes**, and if not we must not save the model and should call the **deactivate()** method.

If the user clicked **Yes** we wish to launch the form mode **Step4Form** which will display the dialog box **Step4DB**. Up until this point whenever we have launched a form mode and its dialog, it has been by associating the form mode with a button, and having it automatically called using **AFXMode.ID_ACTIVATE**. This was done for 'Step 1', 'Step 2' and 'Step 3'.

However for 'Step 4' we used **ID_WARNING** to call **onWarning()**. In order to activate a form mode from without using a button, we need to call its **activate()** method manually. Hence we create an instance of the form mode class, **fm**, and call its **activate()** method.

```
    def onRun(self, sender, sel, ptr):
        # To issue a kernel command directly from the GUI we can use the
        # sendCommand() method
        # To the sendCommand() method we pass the statements that should be
        # executed
        # We separate multiple statements with a \n as is done here. Note that
        # you cannot leave a space after the \n because that means there is a
        # space in front of the next statement, and Python being will object to
        # this wrong indentation
        sendCommand('beamKernel.callStep5()')
```

We've already spoken about how GUI and kernel scripts must be kept separate and cannot call each other using direct means. One way to make them communicate is to use form and procedure modes. This is the most common way, and you'll see it in practice in subsequent sections of this chapter. Another way is to use the **sendCommand()** function. **sendCommand()** takes three arguments, a String argument (required) specifying the command to be executed in the kernel, and two Boolean arguments (optional) **writeToReplay** and **writeToJournal**, which indicate whether the command should be recorded in the replay and journal files.

Here we use **sendCommand()** to execute **beamKernel.callStep5()**. Note that if you wished to execute multiple commands, you would separate them with **\n**. You will see this in later in this chapter as well.

21.4.5 Custom Beam Module

This script is contained in **beamModuleGui.py**. In this script we create a custom module for setting up and analyzing the beam frame problem. It will appear in the **Module** combo box above the viewport in the Abaqus/CAE interface. In fact, since we have not registered any other modules in **beamCaeMainWindow.py**, it is the only module available and will be displayed by default. The custom module consists of a menu with 5 items, a toolbar with 5 buttons, and a toolbox with 5 large buttons. All of these share the exact same functionality as the persistent toolset with ‘Step 1’ through ‘Step 5’.

```
# ********************************************************************************
# Custom Beam Frame Analysis GUI Application
# This script defines a custom beam frame analysis module

# Created for the book "Python Scripts for Abaqus - Learn by Example"
# Author: Gautam Puri
# ********************************************************************************

from abaqusGui import *

from step1Form import Step1Form
from step2Form import Step2Form
from step3Procedure import Step3Procedure
from step4Form import Step4Form

# Class definition

class BeamModuleGui(AFXModuleGui):

    [
        ID_SAVE,
        ID_RUN,
        ID_HELP,
        ID_WARNING
    ] = range(AFXToolsetGui.ID_LAST, AFXToolsetGui.ID_LAST+4)

    def __init__(self):

        print 'BeamModuleGui initialization method called.'

        # Construct base class.
        AFXModuleGui.__init__(self, 'Custom Module', AFXModuleGui.PART)

        FXMAPFUNC(self, SEL_COMMAND, self.ID_SAVE, BeamModuleGui.onSave)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_RUN, BeamModuleGui.onRun)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_HELP, BeamModuleGui.onHelp)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_WARNING, BeamModuleGui.onWarning)

        # Create menu bar and add 2 items that post a dialog box using a form
        custom_menu = AFXMenuPane(self)
        AFXMenuTitle(self, '&Custom Menu', None, custom_menu)
        AFXMenuCommand(self, custom_menu, 'Step 1: Material', None,
                       Step1Form(self), AFXMode.ID_ACTIVATE)
        AFXMenuCommand(self, custom_menu, 'Step 2: Profile', None,
                       Step2Form(self), AFXMode.ID_ACTIVATE)
        AFXMenuCommand(self, custom_menu, 'Step 3: Loads', None,
                       Step3Procedure(self), AFXMode.ID_ACTIVATE)
        AFXMenuCommand(self, custom_menu, 'Step 4: Save the Model', None,
                                                    self, self.ID_SAVE)
        AFXMenuCommand(self, custom_menu, 'Step 5: Run the job', None,
                                                    self, self.ID_RUN)



        # Toolbar items
        toolbar_group = AFXToolbarGroup(owner=self,
                                        title='Beam Frame Analysis Toolbar')
        toolbar_icon_material = afxCreateIcon('icon_material_small.png')
        toolbar_icon_profile = afxCreateIcon('icon_profile_small.png')
        toolbar_icon_load = afxCreateIcon('icon_loads_small.png')
        toolbar_icon_save = afxCreateIcon('icon_save_small.png')
        toolbar_icon_runjob = afxCreateIcon('icon_runjob_small.png')
        AFXToolButton(toolbar_group,
                      '\tCreate Model, Part and define the Material',
                      toolbar_icon_material, Step1Form(self),
                      AFXMode.ID_ACTIVATE)
        AFXToolButton(toolbar_group,
                      '\tSelect the profile and create Assembly and Steps',
                      toolbar_icon_profile,
                      Step2Form(self),
                      AFXMode.ID_ACTIVATE)
        AFXToolButton(toolbar_group,
                      '\tDefine the loads, and create boundary conditions,' + \
                                                          'mesh and job',
                      toolbar_icon_load,
                      Step3Procedure(self),
                      AFXMode.ID_ACTIVATE)
        AFXToolButton(toolbar_group,
                      '\tSave the Model',
                      toolbar_icon_save,
                      self,
                      self.ID_SAVE)
        AFXToolButton(toolbar_group,
                     '\tRun the job',
                     toolbar_icon_runjob,
                     self,
                     self.ID_RUN)


        # Toolbox buttons
        toolbox_icon_material = afxCreateIcon('icon_material_big.png')
        toolbox_icon_profile = afxCreateIcon('icon_profile_big.png')
        toolbox_icon_load = afxCreateIcon('icon_loads_big.png')
        toolbox_icon_save = afxCreateIcon('icon_save_big.png')
        toolbox_icon_runjob = afxCreateIcon('icon_runjob_big.png')

        toolbox_group = AFXToolboxGroup(self)
        AFXToolButton(toolbox_group,
                      '\tCreate Model, Part and define the Material',
                      toolbox_icon_material,
                      Step1Form(self),
                      AFXMode.ID_ACTIVATE)
        toolbox_group = AFXToolboxGroup(self)
        AFXToolButton(toolbox_group,
                      '\tSelect the profile and create Assembly and Steps',
                      toolbox_icon_profile,
                      Step2Form(self),
                      AFXMode.ID_ACTIVATE)
        toolbox_group = AFXToolboxGroup(self)
        AFXToolButton(toolbox_group,
                      '\tDefine the loads, and create boundary conditions, ' + \
                                                              'mesh and job',
                      toolbox_icon_load,
                      Step3Procedure(self),
                      AFXMode.ID_ACTIVATE)
        toolbox_group = AFXToolboxGroup(self)
        AFXToolButton(toolbox_group,
                      '\tSave the Model',
                      toolbox_icon_save,
                      self,
                      self.ID_SAVE)
        toolbox_group = AFXToolboxGroup(self)
        AFXToolButton(toolbox_group,
                      '\tRun the job',
                      toolbox_icon_runjob,
                      self,
                      self.ID_RUN)


    def onHelp(self, sender, sel, ptr):
        mainWindow = getAFXApp().getAFXMainWindow()
        showAFXInformationDialog(mainWindow,
         'Click on each Step in sequence using the buttons in the toolbox. \n' + \
         'Step 1 : Supply material properties \n' + \
         'Step 2: Provide profile dimensions \n' + \
         'Step 3: Define Loading \n' + \
         'Step 4: Save the model \n' + \
         'Step 5: Run the Analysis')
        getAFXApp().getAFXMainWindow().writeToMessageArea('Script created by ' + \
            'Gautam Puri, author of Python Scripts for Abaqus - Learn by Example')
    def onSave(self, sender, sel, ptr):
        # Ask user if he is sure he wishes to save
        warning_owner = getAFXApp().getAFXMainWindow()
        showAFXWarningDialog(owner=warning_owner,
                             message='Are you sure you wish to save this model?',
                             buttonIds=AFXDialog.YES | AFXDialog.NO,
                             tgt=self, sel=self.ID_WARNING)


    def onWarning(self, sender, sel, ptr):
        if sender.getPressedButtonId() == AFXDialog.ID_CLICKED_YES:
            # Launch Step 4 dialog box
            # To do this we must launch the form mode Step4Form
            # Note that this time we are not launching the form mode (that
            # displays the dialog box) using a GUI button as we have done for
            # Step 1, 2 and 3
            # Instead we are launching it manually here. Hence we will first
            # create a form mode object, and then call the activate() method
            fm = Step4Form(self)
            Step4Form.activate(fm)


        elif sender.getPressedButtonId() == AFXDialog.ID_CLICKED_NO:
            self.deactivate()

        return True

    def onRun(self, sender, sel, ptr):
        # To issue a kernel command directly from the GUI we can use the
        # sendCommand() method
        # To the sendCommand() method we pass the statements that should be
        # executed
        # We separate multiple statements with a \n as is done here. Note that
        # you cannot leave a space after the \n because that means there is a
        # space in front of the next statement, and Python being will object to
        # this wrong indentation
        sendCommand('beamKernel.callStep5()')



# Actually create the custom module GUI.
BeamModuleGui()
```

```
from abaqusGui import *

from step1Form import Step1Form
from step2Form import Step2Form
from step3Procedure import Step3Procedure
from step4Form import Step4Form

# Class definition

class BeamModuleGui(AFXModuleGui):

    [
        ID_SAVE,
        ID_RUN,
        ID_HELP,
        ID_WARNING
    ] = range(AFXToolsetGui.ID_LAST, AFXToolsetGui.ID_LAST+4)

    def __init__(self):

        print 'BeamModuleGui initialization method called.'

        # Construct base class.

                        ...
                        ...

        # Create menu bar and add 2 items that post a dialog box using a form

                        ...
                        ...

        # Toolbar items

                        ...
                        ...

        # Toolbox buttons

                        ...
                        ...
```

By this point in the book all of this content should be quite familiar to you. In any case, the code is well commented and quite readable.

```
    def onHelp(self, sender, sel, ptr):

                        ...
                        ...
```

```
   def onSave(self, sender, sel, ptr):

                    ...
                    ...

    def onWarning(self, sender, sel, ptr):

                    ...
                    ...


    def onRun(self, sender, sel, ptr):

                    ...
                    ...
```

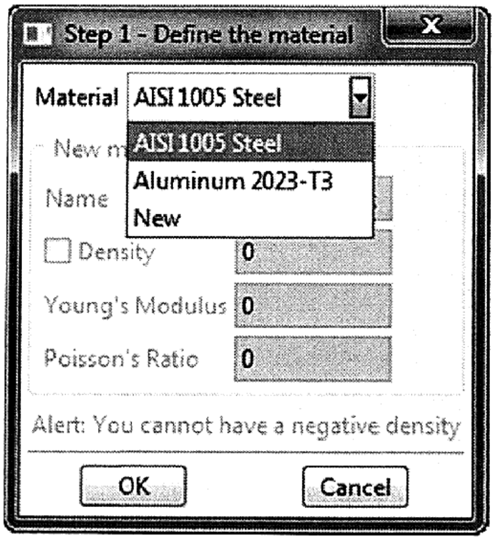
These methods are identical to their namesake methods in **customToolboxButtonsGui.py**. You could consider putting the contents of each method definition in a separate script, and call the methods from both scripts. But in this case it seemed easier to just copy and paste them from **customToolboxButtonsGui.py** to beamModuleGui.py. Also I did not want to complicate this long example by adding one more script file.

```
# Actually create the custom module GUI.
BeamModuleGui()
```

Don't forget to create an instance of the class at the end of your script, otherwise your custom GUI application will not contain this module.

21.4.6 Step 1 Dialog Form and Dialog Box

The form is defined in the script **step1Form.py** and the dialog box in **step1DB.py**. These two scripts together create the dialog box that is displayed when the user clicks on 'Step 1'. They associate the fields in the dialog box with variables, and call the appropriate kernel command when the user clicks **OK** in the dialog box.



The dialog box has a 'Material' combo box widget (**AFXComboBox**) a.k.a. drop-down menu, providing the user with the material options 'AISI 1005 Steel', 'Aluminum 2024-T3' and 'New'. Below the combo-box is a group box layout manager (**FXGroupBox**). This layout manager has within it 4 text fields (**AFXTextField**) for 'Name', 'Density', 'Young's Modulus' and 'Poisson's Ratio'. 'Density' also has a check box next to it which enables and disables (grays out) the field. The entire group box is disabled unless the user selects 'New' from the combo box. Below the group box layout manager is a label (**FXLabel**) which displays a message 'Alert: You cannot have a negative density'). This label is disabled except when the user enters a negative value in the density field.

Two transitions are used – one to detect whether the user has selected 'New' in the combo box, and the other to detect whether the density typed by the user in the 'density' field is positive or negative.

Let's first look at **step1DB.py**

```
# ************************************************************************************
# Custom Beam Frame Analysis GUI Application
# This script defines a dialog box that will be posted by a form (Step1Form in
# step1Form.py)
# ************************************************************************************

from abaqusGui import *

# Class definition

class Step1DB(AFXDataDialog):
```

```
[
    ID_Density,
    ID_NegativeDensity,
    ID_NewMaterialComboSelection,
    ID_ExistingMaterialComboSelection
] = range(AFXToolsetGui.ID_LAST, AFXToolsetGui.ID_LAST+4)

#----------------------------------------------------------------------
def __init__(self, form):

    # DIALOG_ACTIONS_SEPARATOR places a horizontal line/separator between the
    # contents of the dialog box and the OK/CANCEL buttons at the bottom
    # DIALOG_BAILOUT displays a message "Save changes made in the xyz dialog?"
    # if the user clicks Cancel after changing some values in the dialog box
    AFXDataDialog.__init__(self, form, 'Step 1 - Define the material',
                           self.OK|self.CANCEL, DIALOG_ACTIONS_SEPARATOR)

    # Save the form variable for use in the processUpdates() method
    self.form = form

    okBtn = self.getActionButton(self.ID_CLICKED_OK)
    okBtn.setText('OK')

    ComboBox = AFXComboBox(p=self, ncols=0, nvis=1, text='Material',
                           tgt=form.materialselectionKw, sel=0)
    ComboBox.setMaxVisible(3)
    # When creating the combobox items, use the 'sel' argument to assign an
    # integer value to each so that we can use addTransition() whcih requires
    # an integer or a float.
    # In step1Form.py use AFXIntKeyword() and since the combobox items
    # themselves are not integers use 'evalExpression = False'
    ComboBox.appendItem(text='AISI 1005 Steel', sel=1)
    ComboBox.appendItem(text='Aluminum 2023-T3', sel=2)
    ComboBox.appendItem(text='New', sel=3)

    GroupBox = FXGroupBox(p=self, text='New material properties',
                          opts=FRAME_GROOVE|LAYOUT_FILL_X)
    self.groupbox = GroupBox
    VerticalAligner = AFXVerticalAligner(p=GroupBox, opts=0, x=0, y=0, w=0,
                                         h=0, pl=0, pr=0, pt=0, pb=0)
    self.name_textfield = AFXTextField(p=VerticalAligner,
                                       ncols=12,
                                       labelText='Name',
                                       tgt=form.nameKw,
                                       sel=0)
    self.density_textfield = AFXTextField(p=VerticalAligner,
                                          ncols=12,
                                          labelText='Density',
                                          tgt=form.densityKw,
                                          sel=0,
                                          opts=AFXTEXTFIELD_CHECKBUTTON)
```

```
        self.youngs_textfield = AFXTextField(p=VerticalAligner,
                                             ncols=12,
                                             labelText='Young\'s Modulus',
                                             tgt=form.youngsKw,
                                             sel=0)
        self.poissons_textfield = AFXTextField(p=VerticalAligner,
                                               ncols=12,
                                               labelText='Poisson\'s Ratio',
                                               tgt=form.poissonsKw,
                                               sel=0)

        self.density_label = FXLabel(self,
                                  'Alert: You cannot have a negative density')
        self.density_label.disable()


        # Need to route commands from addTransition() to required functions
        FXMAPFUNC(self, SEL_COMMAND, self.ID_Density, Step1DB.onDensity)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_NegativeDensity,
                                     Step1DB.onNegativeDensity)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_NewMaterialComboSelection,
                                     Step1DB.onNewMaterialComboSelection)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_ExistingMaterialComboSelection,
                                     Step1DB.onExistingMaterialComboSelection)

        # Transitions
        self.addTransition(form.materialselectionKw,
                    AFXTransition.EQ,
                    1,
                    self,
                    MKUINT(self.ID_ExistingMaterialComboSelection, SEL_COMMAND),
                    None)
        self.addTransition(form.materialselectionKw,
                    AFXTransition.EQ,
                    2,
                    self,
                    MKUINT(self.ID_ExistingMaterialComboSelection, SEL_COMMAND),
                    None)
        self.addTransition(form.materialselectionKw,
                    AFXTransition.EQ,
                    3,
                    self,
                    MKUINT(self.ID_NewMaterialComboSelection, SEL_COMMAND),
                    None)

        self.addTransition(form.densityKw,
                    AFXTransition.LT,
                    0,
                    self,
                    MKUINT(self.ID_NegativeDensity,SEL_COMMAND),
                    None)
        self.addTransition(form.densityKw,
                    AFXTransition.EQ,
                    0,
                    self,
                    MKUINT(self.ID_Density,SEL_COMMAND),
                    None)
        self.addTransition(form.densityKw,
                    AFXTransition.GT,
                    0,
                    self,
                    MKUINT(self.ID_Density,SEL_COMMAND),
                    None)

    def onNegativeDensity(self, sender, sel, ptr):
        print 'Negative density detected'
        self.density_label.enable()

    def onDensity(self, sender, sel, ptr):
        print 'Density not negative'
        self.density_label.disable()

    def onNewMaterialComboSelection(self, sender, sel, ptr):
        print 'Transition - User has opted to define a new material in combo box'
        self.groupbox.enable()
        self.name_textfield.enable()
        self.density_textfield.enable()
        self.youngs_textfield.enable()
        self.poissons_textfield.enable()


    def onExistingMaterialComboSelection(self, sender, sel, ptr):
        print 'Transition - User has selected existing material in combo box'
        # Grey out the title of the groupbox and make all fields uneditable
        # (however those fields do not get greyed out)
        self.groupbox.disable()
        # Grey out the name textfield
        self.name_textfield.disable()
        # Grey out the density textfield
        self.density_textfield.disable()
        # Grey out the Young's modulus textfield
        self.youngs_textfield.disable()
        # Grey out the Poisson's ratio textfield
        self.poissons_textfield.disable()

    #--------------------------------------------------------------------------
    # The show() method is called by default whenever the dialog box is to be
    # displayed.
    # For example in modifiedCanvasToolsetGui.py you have the statement
    # AFXMenuCommand(self, viewport_menu_with_contents, 'Custom Menu Item',
    #                         None, DemoForm(self), AFXMode.ID_ACTIVATE)
    # So whenever the custom menu item is clicked, the activate() method of the
    # mode is called, and this in turn calls the show() method of the dialog box.
    # Note that this method does NOT need to be defined here if we wish to leave
    # the behavior at default
    # Here however we wish to modify the show method to also print a message to
    # the screen (aside from opening the window which it does by default).
    def show(self):
        print 'Step 1 Dialog box will be displayed'
        # Now must call the base show() command
        AFXDataDialog.show(self)


    #-----------------------------------------------------------------
    # The hide() method is the opposite of show(). It is called by default
    # whenever the dialog box is to be hidden
    # Note that this method does NOT need to be defined here if we wish to leave
    # the behavior at default
    # Here however we wish to modify the hide method to also print a message to
    # the screen (aside from closing the window which it does by default).
    def hide(self):
        print 'Step 1 Dialog box will be hidden'
        # Now must call the base hide() command
        AFXDataDialog.hide(self)
```

You are already familiar with creating dialog boxes, and much of this script is similar to what you saw in section 20.6.8 of the previous chapter. We shall focus on what is new here.

```
class Step1DB(AFXDataDialog):

    [
        ID_Density,
        ID_NegativeDensity,
        ID_NewMaterialComboSelection,
        ID_ExistingMaterialComboSelection
    ] = range(AFXToolsetGui.ID_LAST, AFXToolsetGui.ID_LAST+4)
```

We create IDs that are associated with the fields of the dialog box. Previously you've associated IDs with toolbox buttons, as was done in **customToolboxButtonsGui.py** and **beamModuleGui.py**. However they can also be associated with transitions. To detect the transition of the 'density' field we define two IDs, **ID_Density** and **ID_NegativeDensity**. To detect if the user selects 'New' from the combo box we have **ID_NewMaterialComboSelection** and **ID_ExistingMaterialComboSelection**.

```
    #-----------------------------------------------------------------
    def __init__(self, form):

        # DIALOG_ACTIONS_SEPARATOR places a horizontal line/separator between the
        # contents of the dialog box and the OK/CANCEL buttons at the bottom
        # DIALOG_BAILOUT displays a message "Save changes made in the xyz dialog?"
        # if the user clicks Cancel after changing some values in the dialog box
        AFXDataDialog.__init__(self, form, 'Step 1 - Define the material',
                               self.OK|self.CANCEL, DIALOG_ACTIONS_SEPARATOR)

        # Save the form variable for use in the processUpdates() method
        self.form = form
```

The only statement here you haven't seen before is the last one. **form**, which is passed to the dialog box by the form mode as a reference to itself, is a local variable within the **__init__()** method. We will however need to refer to the form in the **processUpdates()** method a little further down in the script. We therefore need to make a global variable. We refer to it as **self.form** indicating that this is the **form** variable of the class **Step1DB**, and is different from the **form** variable passed to **__init__()** which becomes local to that method.

```
        okBtn = self.getActionButton(self.ID_CLICKED_OK)
        okBtn.setText('OK')
```

getActionButtons() returns the action button with the **messageID** specified as a parameter (or 0 if it is not found). **OK** buttons have a message ID of **ID_CLICKED_OK**. So our variable **okBtn** refers to the **OK** button of the dialog box. The **setText()** method can be used to set/change the text that appears on the **OK** button. We set it to 'OK' here.

```
        ComboBox = AFXComboBox(p=self, ncols=0, nvis=1, text='Material',
                               tgt=form.materialselectionKw, sel=0)
        ComboBox.setMaxVisible(3)
        # When creating the combobox items, use the 'sel' argument to assign an
        # integer value to each so that we can use addTransition() whcih requires
        # an integer or a float.
        # In step1Form.py use AFXIntKeyword() and since the combobox items
        # themselves are not integers use 'evalExpression = False'
        ComboBox.appendItem(text='AISI 1005 Steel', sel=1)
        ComboBox.appendItem(text='Aluminum 2023-T3', sel=2)
        ComboBox.appendItem(text='New', sel=3)
```

A combo box widget is created for selecting the material. **AFXComboBox** accepts a number of parameters. **p** is the parent widget, **ncols** is the number of columns which we have set to 0 for auto-sizing, **nvis** is the number of visible items when the combo box is not expanded, **text** is the label String that appears next to the combo box, **tgt** and **sel** are the message target and message ID. We set these to **form.materialselectionKw** and '0' so the variable **materialselectionKw** in **Step1Form** gets updated with the option selected by the user from the combo box.

For the combo box items created with the **appendItem()** method, we specify **text**, which is the text String, and **sel**, which is a selector. We have assigned selector values of 1, 2 and 3 to the combo box items. This means that if the user selects the first option, **form.materialselectionKw** will hold the integer value 1 and so on.

```
        GroupBox = FXGroupBox(p=self, text='New material properties',
                              opts=FRAME_GROOVE|LAYOUT_FILL_X)
        self.groupbox = GroupBox
```

The group box layout manager is a general purpose layout manager which creates a labeled border around the collection of widgets placed inside it. It has many available parameters; we use 3 of them here. **p** is the parent, **text** is the String that is the label/title of the group box, and **opts** is where you list the various options – we specify **FRAME_GROOVE** for a groove or etched-in border, and **LAYOUT_FILL_X** which causes the group box to stretch or fill the available space in the X direction.

```
        VerticalAligner = AFXVerticalAligner(p=GroupBox, opts=0, x=0, y=0, w=0,
                                             h=0, pl=0, pr=0, pt=0, pb=0)
```

A vertical aligner is another type of layout manager which lays out its children widgets in a vertical fashion one under the other. From each of its children widgets it finds the maximum width of the first child. It then sets the width of the first children of all its children widgets to this same value. This means if you have an **AFXTextField** widget which places a label and a text field on the canvas, the width of all the labels will be the same in order to maintain a neat aligned look.

The parameters accepted by **AFXVerticalAligner** are **p** (parent widget), **opts** (options), **x** (X coordinate of origin), **y** (Y coordinate of origin), **w** (width), **h** (height), **pl** (left padding), **pr** (right padding), **pt** (top padding), **pb** (bottom padding), **hs** (horizontal spacing) and **vs** (vertical spacing).

```
        self.name_textfield = AFXTextField(p=VerticalAligner,
                                           ncols=12,
                                           labelText='Name',
                                           tgt=form.nameKw,
                                           sel=0)
        self.density_textfield = AFXTextField(p=VerticalAligner,
                                              ncols=12,
                                              labelText='Density',
                                              tgt=form.densityKw,
                                              sel=0,
                                              opts=AFXTEXTFIELD_CHECKBUTTON)
        self.youngs_textfield = AFXTextField(p=VerticalAligner,
                                             ncols=12,
                                             labelText='Young\'s Modulus',
                                             tgt=form.youngsKw,
                                             sel=0)
        self.poissons_textfield = AFXTextField(p=VerticalAligner,
                                               ncols=12,
                                               labelText='Poisson\'s Ratio',
                                               tgt=form.poissonsKw,
                                               sel=0)
```

4 text fields are created here as children of the vertical aligner layout manager using **AFXTextField()**. It accepts a number of parameters but we've only used a few here. **p** is the parent widget, which we set to the variable **VerticalAligner** (which refers to our vertical aligner). **ncols** is the number of columns in the text field. **labelText** is the text that appears in the label next to the text field. **tgt** and **sel** are the message target and message ID. We set the message target to the variable in **Step1Form** that will hold the value of the widget.

For the text field for density we also specify an option using **opts**. We set it to **AFXTEXTFIELD_CHECKBUTTON** which places a check button next to the label. If the button is checked, the text field is enabled, and if not, it is disabled or grayed out.

```
        self.density_label = FXLabel(self,
                                     'Alert: You cannot have a negative density')
        self.density_label.disable()
```

FXLabel() creates a text label on the canvas. We set the parent **p** to **self** and the text label **labelText** to an alert. The label is then grayed out using the **disable()** method. It will later be enabled when a transition detects the user inputting a negative density in the density text field.

```
        # Need to route commands from addTransition() to required functions
        FXMAPFUNC(self, SEL_COMMAND, self.ID_Density, Step1DB.onDensity)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_NegativeDensity,
                                     Step1DB.onNegativeDensity)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_NewMaterialComboSelection,
                                     Step1DB.onNewMaterialComboSelection)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_ExistingMaterialComboSelection,
                                     Step1DB.onExistingMaterialComboSelection)
```

FXMAPFUNC() maps the IDs used by the transitions to the correct functions.

```
        # Transitions
        self.addTransition(form.materialselectionKw,
                           AFXTransition.EQ,
                           1,
                           self,
                           MKUINT(self.ID_ExistingMaterialComboSelection, SEL_COMMAND),
                           None)
        self.addTransition(form.materialselectionKw,
                           AFXTransition.EQ,
                           2,
                           self,
                           MKUINT(self.ID_ExistingMaterialComboSelection, SEL_COMMAND),
                           None)
        self.addTransition(form.materialselectionKw,
                           AFXTransition.EQ,
                           3,
                           self,
                           MKUINT(self.ID_NewMaterialComboSelection, SEL_COMMAND),
                           None)

        self.addTransition(form.densityKw,
                           AFXTransition.LT,
                           0,
                           self,
                           MKUINT(self.ID_NegativeDensity,SEL_COMMAND),
                           None)
        self.addTransition(form.densityKw,
                           AFXTransition.EQ,
                           0,
                           self,
                           MKUINT(self.ID_Density,SEL_COMMAND),
                           None)
        self.addTransition(form.densityKw,
                           AFXTransition.GT,
                           0,
                           self,
                           MKUINT(self.ID_Density,SEL_COMMAND),
                           None)
```

The **addTransition()** method adds the transition detection spoken off earlier to the application. It accepts 6 parameters. **target** is the keyword in the form which holds the value of the widget, **op** is the operator type (such as **AFXTransition.EQ)**, **value** is the reference value that you are comparing with, **tgt** is the message target, **sel** is the message selector which will be made up of a message ID and a message type, and the last parameter is **ptr** - the message data – which we set to **None**.

The first 3 transitions are used to detect the selection of item 1, 2 or 3 in the materials combo box. If either 1 ('AISI 1005 Steel') or 2 ('Aluminum 2024-T3) is selected, **ID_ExistingMaterialComboSelection** is used, which **FXMAPFUNC()** maps to the method **Step1DB.onExistingMaterialComboSelection**. Whereas if 3 ('New') is selected **ID_NewMaterialComboSelection** is used, which **FXMAPFUNC()** maps to the method **Step1DB.onNewMaterialComboSelection()**. For the message selector we use **SEL_COMMAND** since the user selects an item from the combo box.

The next 3 transitions are used to detect whether the value typed into the density text field is positive, negative or zero using **AFXTransition.LT**, **EQ** and **GT**. For the message selector we use **MKUINT()** which is a function to create a selector using an identity and a message type.

```
    def onNegativeDensity(self, sender, sel, ptr):
        print 'Negative density detected'
        self.density_label.enable()
```

onNegativeDensity() is called by a transition using **FXMAPFUNC()** when the user enters a negative density value in the text field. It prints a message to the console, and enables the alert label on the canvas (which is grayed out by default) using the **enable()** method. Since the transition repeatedly detects the presence of a negative density as long as it is present in the text field, you will see the message being printed to the console repeatedly. This has been intentionally done so you can observe the mechanism of transitions in action.

```
    def onDensity(self, sender, sel, ptr):
        print 'Density not negative'
        self.density_label.disable()
```

onDensity() is the opposite of **onNegativeDensity()**. It is called by a transition using **FXMAPFUNC()** when the density text field has a positive value or zero. It prints a message to the console, and disables the alert label on the canvas if it is currently enabled by using the **disable()** method. Since the transition repeatedly detects the presence of a positive or zero density as long as it is present in the text field, you will see the message being printed to the console repeatedly.

```
    def onNewMaterialComboSelection(self, sender, sel, ptr):
        print 'Transition - User has opted to define a new material in combo box'
        self.groupbox.enable()
        self.name_textfield.enable()
        self.density_textfield.enable()
        self.youngs_textfield.enable()
        self.poissons_textfield.enable()
```

onNewMaterialComboSelection() is called by a transition using **FXMAPFUNC()** when the user selects 'New' in the material combo box. It prints a message to the console. It then enables the group box, which means the group box title and border are not grayed out, and then enables all of the widgets inside it one by one. This is done using the **enable()** method. Since the transition repeatedly detects the presence of 'New' as long as it is selected, you will see the message being printed to the console repeatedly.

```
    def onExistingMaterialComboSelection(self, sender, sel, ptr):
        print 'Transition - User has selected existing material in combo box'
        # Grey out the title of the groupbox and make all fields uneditable
        # (however those fields do not get greyed out)
        self.groupbox.disable()
        # Grey out the name textfield
        self.name_textfield.disable()
        # Grey out the density textfield
        self.density_textfield.disable()
        # Grey out the Young's modulus textfield
        self.youngs_textfield.disable()
        # Grey out the Poisson's ratio textfield
        self.poissons_textfield.disable()
```

onExistingMaterialComboSelection() is called by a transition using **FXMAPFUNC()** when the user selects 'AISI 1005 Steel' or 'Aluminum 2024-T3' from the material combo box. It prints a message to the console. It then disables the group box, which means the group box title and border are grayed out, and then disables all of the widgets inside it one by one. This is done using the **disable()** method. Since the transition repeatedly detects the presence of the first and second materials as long as they are selected, you will see the message being printed to the console repeatedly.

```
    #-----------------------------------------------------------------------
    # The show() method is called by default whenever the dialog box is to be
    # displayed.
    # For example in modifiedCanvasToolsetGui.py you have the statement
    # AFXMenuCommand(self, viewport_menu_with_contents, 'Custom Menu Item',
    #                                None, DemoForm(self), AFXMode.ID_ACTIVATE)
    # So whenever the custom menu item is clicked, the activate() method of the
    # mode is called, and this in turn calls the show() method of the dialog box.
    # Note that this method does NOT need to be defined here if we wish to leave
    # the behavior at default
    # Here however we wish to modify the show method to also print a message to
    # the screen (aside from opening the window which it does by default).
    def show(self):
        print 'Step 1 Dialog box will be displayed'
        # Now must call the base show() command
        AFXDataDialog.show(self)

    #------------------------------------------------------------------------
    # The hide() method is the opposite of show(). It is called by default
    # whenever the dialog box is to be hidden
    # Note that this method does NOT need to be defined here if we wish to leave
    # the behavior at default
    # Here however we wish to modify the hide method to also print a message to
    # the screen (aside from closing the window which it does by default).
    def hide(self):
        print 'Step 1 Dialog box will be hidden'
        # Now must call the base hide() command
        AFXDataDialog.hide(self)
```

You learnt about **show()** and **hide()** in the previous chapter. The code has been reused here.

Let's now look **step1Form.py**, which defines the form associated with the dialog box.

```
# ********************************************************************************
# Custom Beam Frame Analysis GUI Application
# This script is the form for step 1. It will post the step 1 dialog box
# (Step1 in step1DB.py)
# ********************************************************************************

from abaqusGui import *
import step1DB

# Class definition

class Step1Form(AFXForm):

    #------------------------------------------------------------------------
    def __init__(self, owner):

        # Construct base class.
        AFXForm.__init__(self, owner)

        # Command to execute when OK is clicked
        self.cmd = AFXGuiCommand(self, 'callStep1', 'beamKernel')

        # Material selection is done with a combo box.
        # For the items in the combo box we will specify integers using the 'sel'
        # argument of appendItem().
        # This integer will be returned and stored in 'materialselectionKw' hence
        # we use AFXIntKeyword instead of AFXStringKeyword which could otherwise
        # be used with a combobox
        # Since the choices shown in the combobox do not themselves represent
        # numerical values, expression evaluation must be disabled with
        # 'evalExpression = False'
        self.materialselectionKw = AFXIntKeyword(command=self.cmd,
                                                 name='selected_material',
                                                 isRequired=TRUE,
                                                 defaultValue=1,
                                                 evalExpression=FALSE)
        self.nameKw = AFXStringKeyword(command=self.cmd, name='name',
                                       isRequired=TRUE, defaultValue='NewUserMat')
        self.densityKw = AFXFloatKeyword(command=self.cmd, name='density',
                                         isRequired=TRUE, defaultValue=0)
        self.youngsKw = AFXFloatKeyword(command=self.cmd, name='youngs',
                                        isRequired=TRUE, defaultValue=0)
        self.poissonsKw = AFXFloatKeyword(command=self.cmd, name='poissons',
                                          isRequired=TRUE, defaultValue=0)


    #-----------------------------------------------------------------------
    # A getFirstDialog() method MUST be written for a mode (a Form mode)
    # It should return the first dialog box of the mode
    def getFirstDialog(self):

        # Reload the dialog module so that any changes to the dialog are updated.
        reload(step1DB)
        return step1DB.Step1DB(self)


    #-----------------------------------------------------------------------
    # A mode is usually activated by sending it a message with its ID set to
    # ID_ACTIVATE
    # This message causes activate() to be called
    # Note that it is NOT necessary to define this activate() method unless you
    # wish to change the default behavior
    # Here we would like it to print a message to the screen before proceeding
    def activate(self):
        print 'Step 1 Mode (form) has been activated'
        # Now must call the base method
        AFXForm.activate(self)

    #-----------------------------------------------------------------------
    # issueCommands() is called by default and you do not need to define it
    # We define it here in order to get the command string and print it to the
    # message area to help with debugging
    def issueCommands(self):

        # Get the command string that will be sent to the kernel for processing
        cmdstr = self.getCommandString()

        # Write this command string to the message area so we know it executed
        getAFXApp().getAFXMainWindow().writeToMessageArea(cmdstr)
        # The sendCommandString() method is usually called by default.
        # However since we are defining issueCommands() we now need to manually
        # call this method
        self.sendCommandString(cmdstr)

        # Deactivate the form if the user presses the OK button of the dialog box
        self.deactivateIfNeeded()
        return TRUE
```

We shall focus on what is new in this script.

```
        # Command to execute when OK is clicked
        self.cmd = AFXGuiCommand(self, 'callStep1', 'beamKernel')
```

Here we define the kernel command that should be called when **OK** is clicked. **beamKernel.py** is the script, and **callStep1()** is the method in the script that should be called.

```
        # Material selection is done with a combo box.
        # For the items in the combo box we will specify integers using the 'sel'
        # argument of appendItem().
        # This integer will be returned and stored in 'materialselectionKw' hence
        # we use AFXIntKeyword instead of AFXStringKeyword which could otherwise
        # be used with a combobox
        # Since the choices shown in the combobox do not themselves represent
        # numerical values, expression evaluation must be disabled with
        # 'evalExpression = False'
        self.materialselectionKw = AFXIntKeyword(command=self.cmd,
                                                 name='selected_material',
                                                 isRequired=TRUE,
                                                 defaultValue=1,
                                                 evalExpression=FALSE)
        self.nameKw = AFXStringKeyword(command=self.cmd, name='name',
                                       isRequired=TRUE, defaultValue='NewUserMat')
        self.densityKw = AFXFloatKeyword(command=self.cmd, name='density',
                                         isRequired=TRUE, defaultValue=0)
        self.youngsKw = AFXFloatKeyword(command=self.cmd, name='youngs',
                                        isRequired=TRUE, defaultValue=0)
        self.poissonsKw = AFXFloatKeyword(command=self.cmd, name='poissons',
                                          isRequired=TRUE, defaultValue=0)
```

materialselectionKw, **nameKw**, **densityKw**, **youngsKw** and **poissonsKw** are the keywords or variables associated with the widgets of the dialog box. These variables were set as the target **(tgt)** of the widgets in the dialog box. All the data the user types, and all the options he selects, get stored in the target variables corresponding to each widget.

We use **AFXIntKeyword()**, **AFXFloatKeyword()** and **AFXStringKeyword()** depending on whether the value will be an Int, a Float or a String. Needless to say, the name of the material, **nameKw**, is a string, while the density **densityKw**, Young's Modulus **youngsKw** and Poisson's Ratio **poissonsKw** are floats. The reason the selected item of the combo box **materialselectionKw** is an integer is because we used the **sel** parameter in the **appendItem()** method while creating the combo box, and set it to 1, 2 and 3 respectively for each of the 3 items. This integer value is what will get stored in **materialselectionKw**.

AFXIntKeyword() accepts 5 parameters. **command** is the host command, which in this case is set to **self.cmd** defined earlier. **name** is the keyword name which must be identical to the parameter name in the function definition in the kernel script. **isRequired** is a Boolean stating whether or not the variable is required. By setting it to 'True' we can force the application to include this parameter in the command string it sends to the kernel, whether or not the user has defined a value for it. **defaultValue** is the default value of the keyword/variable. Note that the default value for the keyword is the default text in a text field, or default option selected in a combo box widget, and so on. **evalExpression** indicates whether or not the keyword supports expression evaluation. In the case of the material name selection from the combo box, the text in the combo box does not represent the numerical values, hence we must turn expression evaluation off.

AFXFloatKeyword() and **AFXStringKeyword()** accept similar parameters. If you wish to learn more refer to the documentation for details.

```
#------------------------------------------------------------------
# A getFirstDialog() method MUST be written for a mode (a Form mode)
# It should return the first dialog box of the mode
def getFirstDialog(self):

    # Reload the dialog module so that any changes to the dialog are updated.
    reload(step1DB)
    return step1DB.Step1DB(self)


#------------------------------------------------------------------
# A mode is usually activated by sending it a message with its ID set to
# ID_ACTIVATE
# This message causes activate() to be called
# Note that it is NOT necessary to define this activate() method unless you
# wish to change the default behavior
# Here we would like it to print a message to the screen before proceeding
def activate(self):
        print 'Step 1 Mode (form) has been activated'
        # Now must call the base method
        AFXForm.activate(self)

    #--------------------------------------------------------------------------
    # issueCommands() is called by default and you do not need to define it
    # We define it here in order to get the command string and print it to the
    # message area to help with debugging
    def issueCommands(self):

        # Get the command string that will be sent to the kernel for processing
        cmdstr = self.getCommandString()

        # Write this command string to the message area so we know it executed
        getAFXApp().getAFXMainWindow().writeToMessageArea(cmdstr)

        # The sendCommandString() method is usually called by default.
        # However since we are defining issueCommands() we now need to manually
        # call this method
        self.sendCommandString(cmdstr)

        # Deactivate the form if the user presses the OK button of the dialog box
        self.deactivateIfNeeded()
        return TRUE
```

getFirstDialog(), **activate()** and **issueCommands()** were all used in **DemoForm** in the previous chapter. The only difference here is in **issueCommands()**. While in the previous chapter **DemoDB** did not actually do anything, here the statement

```
        self.sendCommandString(cmdstr)
```

is used to send the command to the kernel script.

cmdstr contains the command string obtained using **getCommandString()**. If 'AISI 1005 Steel' is selected (and therefore no values entered into the text fields), the command string looks something like this

```
beamKernel.callStep1(selected_material=1, name='NewUserMat', density=0, youngs=0,
poissons=0)
```

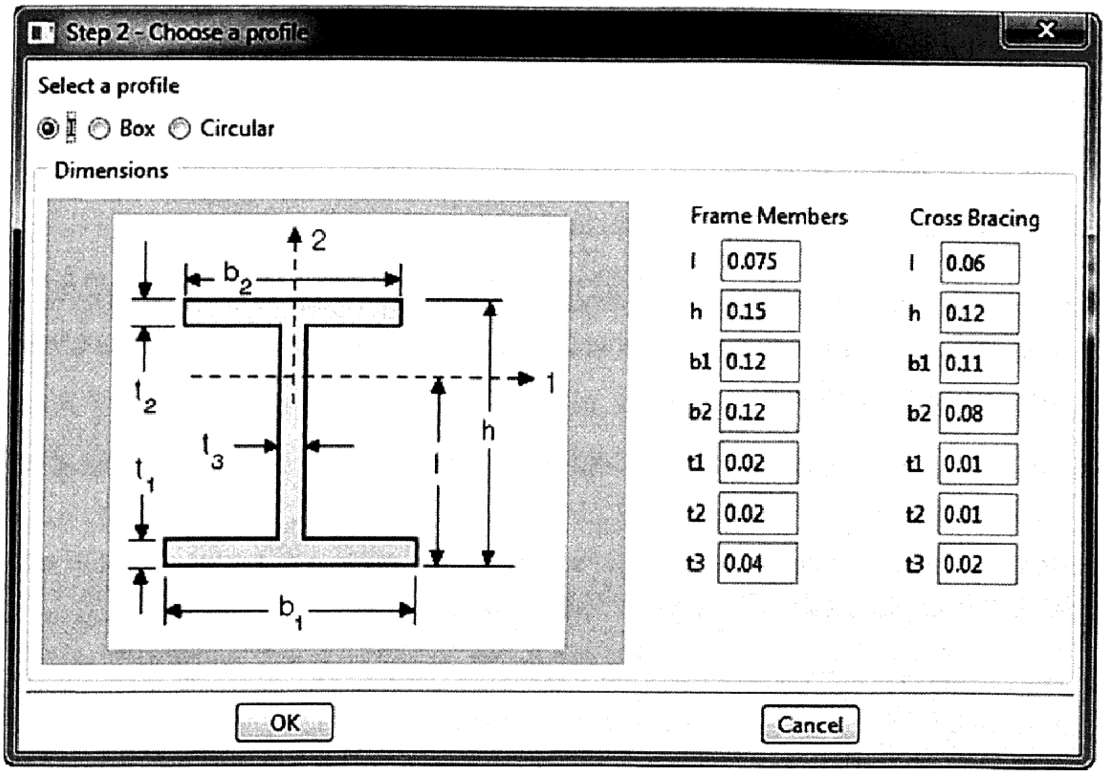
Notice that the order of the arguments passed in the function call are exactly the same as the order in which the keywords were defined a moment ago.

21.4.7 Step 2 Dialog Form and Dialog Box

This form is defined in the script **step2Form.py** and the dialog box in **step2DB.py**. These two scripts together create the dialog box that is displayed when the user clicks on 'Step 2'. They associate the fields in the dialog box with variables, and call the appropriate kernel command when the user clicks **OK** in the dialog box.

The dialog box has a label 'Select a profile'. Below this is a horizontal frame layout manager (**FXHorizontalFrame**). Within this horizontal frame are 3 radio buttons for 'I', 'Box' and 'Circular' beam profiles.

Below this is a group box (**FXGroupBox**) to hold all of the other widgets in the dialog box. Inside this group box a switcher (**FXSwitcher**) is placed. This switcher will "switch" depending on which of the 3 radio buttons are selected. 3 sets of widgets are placed inside this switcher. All of them consist of a horizontal frame (**FXHorizontalFrame**) and within this horizontal frame are an icon (**FXLabel** with **afxCreateIcon**) and two vertical frames (**FXVerticalFrame**). These two vertical frames consist of a label and text fields so the user can enter dimensions for frame members and cross bracing members.

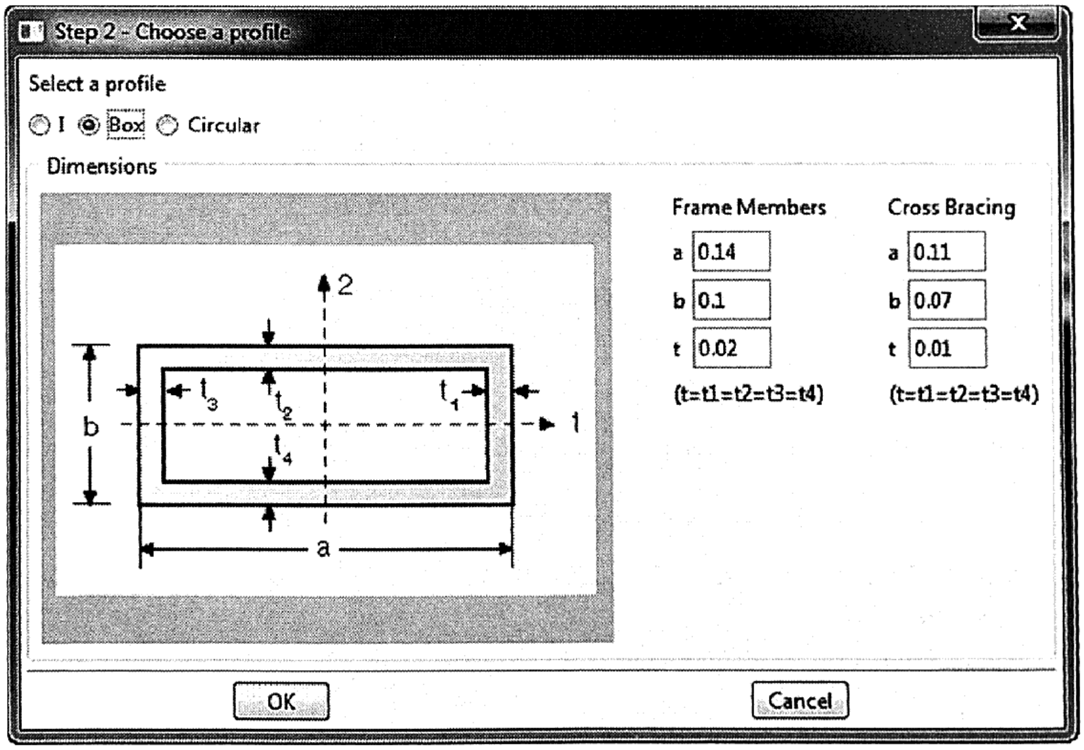


Select a profile

[✓ I] Box Circular

Dimensions

	Frame Members		Cross Bracing
I	0.075	I	0.06
h	0.15	h	0.12
b1	0.12	b1	0.11
b2	0.12	b2	0.08
t1	0.02	t1	0.01
t2	0.02	t2	0.01
t3	0.04	t3	0.02

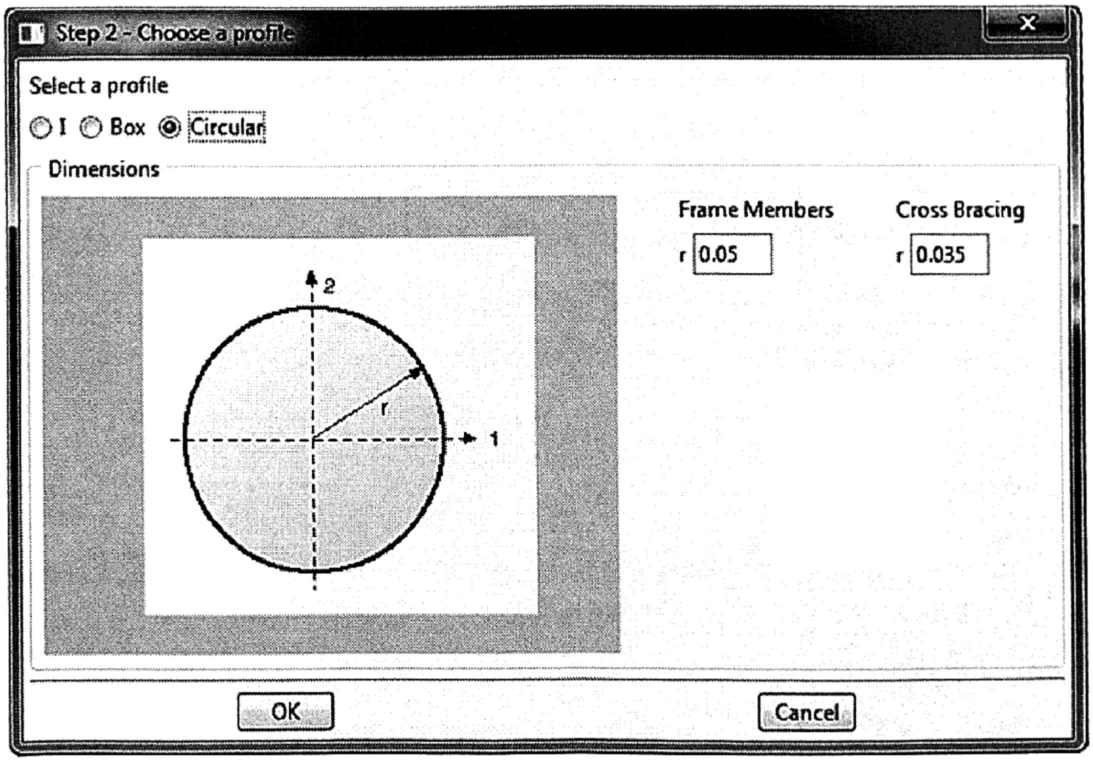


Select a profile

( ) I [✓ Box] ( ) Circular

Dimensions

	Frame Members	Cross Bracing
a	0.14	0.11
b	0.1	0.07
t	0.02	0.01
	(t=t1=t2=t3=t4)	(t=t1=t2=t3=t4)



Select a profile

( ) I ( ) Box [✓ Circular]

Dimensions

	Frame Members	Cross Bracing
r	0.05	0.035

Let's first examine **step2DB.py**.

```
# ******************************************************************************
# Custom Beam Frame Analysis GUI Application
# This script defines a dialog box that will be posted by a form (Step2Form in
# step2Form.py)
# ******************************************************************************

from abaqusGui import *

# Class definition

class Step2DB(AFXDataDialog):

    #-----------------------------------------------------------------------
    def __init__(self, form):

        # DIALOG_ACTIONS_SEPARATOR places a horizontal line/separator between the
        # contents of the dialog box and the OK/CANCEL buttons at the bottom
        # DIALOG_BAILOUT displays a message "Save changes made in the xyz dialog?"
        # if the user clicks Cancel after changing some values in the dialog box
        AFXDataDialog.__init__(self, form, 'Step 2 - Choose a profile',
                               self.OK|self.CANCEL, DIALOG_ACTIONS_SEPARATOR)

        # Save the form variable for use in the processUpdates() method
        self.form = form

        okBtn = self.getActionButton(self.ID_CLICKED_OK)
        okBtn.setText('OK')

        l = FXLabel(p=self, text='Select a profile', opts=JUSTIFY_LEFT)

        HFrame_1 = FXHorizontalFrame(p=self, opts=0, x=0, y=0, w=0, h=0, pl=0,
                                                          pr=0, pt=0, pb=0)
        FXRadioButton(p=HFrame_1, text='I', tgt=form.profile_Kw, sel=1)
        FXRadioButton(p=HFrame_1, text='Box', tgt=form.profile_Kw, sel=2)
        FXRadioButton(p=HFrame_1, text='Circular', tgt=form.profile_Kw, sel=3)


        # Groupbox 'Dimensions' to contain all widget
        GroupBox = FXGroupBox(p=self, text='Dimensions', opts=FRAME_GROOVE)

        # All contents of this groupbox are inside a switcher so they can change
        # for I, Box and Circular profiles
        self.groupbox_switcher = FXSwitcher(GroupBox, 0, 0,0,0,0, 0,0,0,0)


        # Switcher contents for I Profile ------------------------------
        HFrame = FXHorizontalFrame(p=self.groupbox_switcher, opts=0, x=0, y=0,
                                   w=0, h=0, pl=0, pr=0, pt=0, pb=0, hs=30, vs=0)
```

```
i_profile_icon = afxCreateIcon('icon_i_profile.png')
FXLabel(p=HFrame, text='', ic=i_profile_icon)

VFrame = FXVerticalFrame(p=HFrame, opts=0, x=0, y=0, w=0, h=0, pl=0, pr=0,
                                                              pt=0, pb=0)
FXLabel(p=VFrame, text='Frame Members')
VAligner_frame = AFXVerticalAligner(p=VFrame, opts=0, x=0, y=0, w=0, h=0,
                                                    pl=0, pr=0, pt=0, pb=0)
AFXTextField(p=VAligner_frame, ncols=6, labelText='l',
                               tgt=form.l_1_Kw, sel=0)
AFXTextField(p=VAligner_frame, ncols=6, labelText='h',
                               tgt=form.h_1_Kw, sel=0)
AFXTextField(p=VAligner_frame, ncols=6, labelText='b1',
                               tgt=form.b1_1_Kw, sel=0)
AFXTextField(p=VAligner_frame, ncols=6, labelText='b2',
                               tgt=form.b2_1_Kw, sel=0)
AFXTextField(p=VAligner_frame, ncols=6, labelText='t1',
                               tgt=form.t1_1_Kw, sel=0)
AFXTextField(p=VAligner_frame, ncols=6, labelText='t2',
                               tgt=form.t2_1_Kw, sel=0)
AFXTextField(p=VAligner_frame, ncols=6, labelText='t3',
                               tgt=form.t3_1_Kw, sel=0)


VFrame = FXVerticalFrame(p=HFrame, opts=0, x=0, y=0, w=0, h=0, pl=0,
                                                      pr=0, pt=0, pb=0)
FXLabel(p=VFrame, text='Cross Bracing')
VAligner_cross = AFXVerticalAligner(p=VFrame, opts=0, x=0, y=0, w=0, h=0,
                                                 pl=0, pr=0, pt=0, pb=0)
AFXTextField(p=VAligner_cross, ncols=6, labelText='l',
                               tgt=form.l_2_Kw, sel=0)
AFXTextField(p=VAligner_cross, ncols=6, labelText='h',
                               tgt=form.h_2_Kw, sel=0)
AFXTextField(p=VAligner_cross, ncols=6, labelText='b1',
                               tgt=form.b1_2_Kw, sel=0)
AFXTextField(p=VAligner_cross, ncols=6, labelText='b2',
                               tgt=form.b2_2_Kw, sel=0)
AFXTextField(p=VAligner_cross, ncols=6, labelText='t1',
                               tgt=form.t1_2_Kw, sel=0)
AFXTextField(p=VAligner_cross, ncols=6, labelText='t2',
                               tgt=form.t2_2_Kw, sel=0)
AFXTextField(p=VAligner_cross, ncols=6, labelText='t3',
                               tgt=form.t3_2_Kw, sel=0)


# Switcher contents for Box Profile ------------------------------
HFrame = FXHorizontalFrame(p=self.groupbox_switcher, opts=0, x=0, y=0,
                           w=0, h=0, pl=0, pr=0, pt=0, pb=0, hs=30, vs=0)

box_profile_icon = afxCreateIcon('icon_box_profile.png')
FXLabel(p=HFrame, text='', ic=box_profile_icon)
```

```
VFrame = FXVerticalFrame(p=HFrame, opts=0, x=0, y=0, w=0, h=0, pl=0,
                                                    pr=0, pt=0, pb=0)
FXLabel(p=VFrame, text='Frame Members')
VAligner_frame = AFXVerticalAligner(p=VFrame, opts=0, x=0, y=0, w=0, h=0,
                                                pl=0, pr=0, pt=0, pb=0)
AFXTextField(p=VAligner_frame, ncols=6, labelText='a',
                               tgt=form.a_1_Kw, sel=0)
AFXTextField(p=VAligner_frame, ncols=6, labelText='b',
                               tgt=form.b_1_Kw, sel=0)
AFXTextField(p=VAligner_frame, ncols=6, labelText='t',
                               tgt=form.t_1_Kw, sel=0)
FXLabel(p=VFrame, text='(t=t1=t2=t3=t4)')

VFrame = FXVerticalFrame(p=HFrame, opts=0, x=0, y=0, w=0, h=0, pl=0,
                                                    pr=0, pt=0, pb=0)
FXLabel(p=VFrame, text='Cross Bracing')
VAligner_cross = AFXVerticalAligner(p=VFrame, opts=0, x=0, y=0, w=0, h=0,
                                                pl=0, pr=0, pt=0, pb=0)
AFXTextField(p=VAligner_cross, ncols=6, labelText='a',
                               tgt=form.a_2_Kw, sel=0)
AFXTextField(p=VAligner_cross, ncols=6, labelText='b',
                               tgt=form.b_2_Kw, sel=0)
AFXTextField(p=VAligner_cross, ncols=6, labelText='t',
                               tgt=form.t_2_Kw, sel=0)
FXLabel(p=VFrame, text='(t=t1=t2=t3=t4)')


# Switcher contents for Circular Profile -------------------------
HFrame = FXHorizontalFrame(p=self.groupbox_switcher, opts=0, x=0, y=0,
                           w=0, h=0, pl=0, pr=0, pt=0, pb=0, hs=30, vs=0)

circular_profile_icon = afxCreateIcon('icon_circular_profile.png')
FXLabel(p=HFrame, text='', ic=circular_profile_icon)

VFrame = FXVerticalFrame(p=HFrame, opts=0, x=0, y=0, w=0, h=0, pl=0,
                                                    pr=0, pt=0, pb=0)
FXLabel(p=VFrame, text='Frame Members')
VAligner_frame = AFXVerticalAligner(p=VFrame, opts=0, x=0, y=0, w=0, h=0,
                                                pl=0, pr=0, pt=0, pb=0)
AFXTextField(p=VAligner_frame, ncols=6, labelText='r',
                               tgt=form.r_1_Kw, sel=0)

VFrame = FXVerticalFrame(p=HFrame, opts=0, x=0, y=0, w=0, h=0, pl=0,
                                                    pr=0, pt=0, pb=0)
FXLabel(p=VFrame, text='Cross Bracing')
VAligner_cross = AFXVerticalAligner(p=VFrame, opts=0, x=0, y=0, w=0, h=0,
                                                pl=0, pr=0, pt=0, pb=0)
AFXTextField(p=VAligner_cross, ncols=6, labelText='r',
                               tgt=form.r_2_Kw, sel=0)
```

```
# The processUpdates() method is called during each GUI update cycle
# We will use it to replace the groupbox contents as the user selects a
# different profile by changing the switcher
# While we can use transitions to achieve this same effect, for more complex
# tasks such as comparing different radiobutton or textfield values
# processUpdates() offers more control and you can call the appropriate
# methods based on complex decision logic
# Since processUpdate() is called at each GUI update cycle, it should not
# execute any time consuming code
def processUpdates(self):
    # unlike in the case of transitions, here we must use the getValues()
    # method to get the value of a radiobutton, textfield or other widget
    if self.form.profile_Kw.getValue() == 1:
        self.groupbox_switcher.setCurrent(0)
    elif self.form.profile_Kw.getValue() == 2:
        self.groupbox_switcher.setCurrent(1)
    else:
        self.groupbox_switcher.setCurrent(2)


#--------------------------------------------------------------------
# The show() method is called by default whenever the dialog box is to be
# displayed.
# For example in modifiedCanvasToolsetGui.py you have the statement
# AFXMenuCommand(self, viewport_menu_with_contents, 'Custom Menu Item', None,
#                                  DemoForm(self), AFXMode.ID_ACTIVATE)
# So whenever the custom menu item is clicked, the activate() method of the
# mode is called, and this in turn calls the show() method of the dialog box.
# Note that this method does NOT need to be defined here if we wish to leave
# the behavior at default
# Here however we wish to modify the show method to also print a message to
# the screen (aside from opening the window which it does by default).
def show(self):
    print 'Step 2 Dialog box will be displayed'
    # Now must call the base show() command
    AFXDataDialog.show(self)


#--------------------------------------------------------------------
# The hide() method is the opposite of show(). It is called by default
# whenever the dialog box is to be hidden
# Note that this method does NOT need to be defined here if we wish to leave
# the behavior at default
# Here however we wish to modify the hide method to also print a message to
# the screen (aside from closing the window which it does by default).
def hide(self):
    print 'Step 2 Dialog box will be hidden'
    # Now must call the base hide() command
    AFXDataDialog.hide(self)
```

Let's focus our attention on the new content here.

```
        HFrame_1 = FXHorizontalFrame(p=self, opts=0, x=0, y=0, w=0, h=0, pl=0,
                                                             pr=0, pt=0, pb=0)
        FXRadioButton(p=HFrame_1, text='I', tgt=form.profile_Kw, sel=1)
        FXRadioButton(p=HFrame_1, text='Box', tgt=form.profile_Kw, sel=2)
        FXRadioButton(p=HFrame_1, text='Circular', tgt=form.profile_Kw, sel=3)
```

A horizontal frame **FXHorizontalFrame** arranges its contents horizontally, as opposed to a vertical frame which you saw earlier. We use it so we can place our radio buttons side by side. It accepts a number of arguments. **p** is the parent, **opts** is options, **x** and **y** are the X and Y coordinates, **w** and **h** are width and height, **pl**, **pr**, **pt** and **pb** are the padding on all 4 sides, and **hs** and **vs** are the horizontal and vertical spacing between widgets inside the horizontal frame.

FXRadioButton() is used to create a radio button. It accepts all the same arguments as **FXHorizontalFrame**, and in addition accepts **text**, the text displayed next to the radio button, and **tgt** and **sel**, which are the message target and message ID. Note that we set the parent of the radio buttons to **HFrame_1** which is our horizontal frame. A very important point to note is that the target of all 3 radio buttons is the same keyword variable. This is why when you select one radio button the other gets unselected. This is the behavior you are most likely looking for, since the point of radio buttons is that you can only select one of them (if you wish for the user to select multiple options, it makes more sense to use checkboxes).

```
        # All contents of this groupbox are inside a switcher so they can change
        # for I, Box and Circular profiles
        self.groupbox_switcher = FXSwitcher(GroupBox, 0, 0,0,0,0, 0,0,0,0)
```

A switcher is created using **FXSwitcher()**. It swaps children that are located on top of each other. You can create a number of children for a switcher and then instruct it to display one instead of the others. This helps preserve real-estate since several panels can be placed one over another and switched out as needed. In addition it allows you to show the user GUI panels based on context, and hide what is not required. This is handy for us because we would like a different set of labels and text fields to be displayed depending on whether the user wishes to define an 'I' profile, or a 'box' or 'circular' profile. **FXSwitcher** accepts all the same arguments as **FXHorizontalFrame**.

```
        # Switcher contents for I Profile ------------------------------
        HFrame = FXHorizontalFrame(p=self.groupbox_switcher, opts=0, x=0, y=0,
                                        w=0, h=0, pl=0, pr=0, pt=0, pb=0, hs=30, vs=0)

                    ...
                    ...
                    ...



        # Switcher contents for Box Profile ------------------------------
        HFrame = FXHorizontalFrame(p=self.groupbox_switcher, opts=0, x=0, y=0,
                                        w=0, h=0, pl=0, pr=0, pt=0, pb=0, hs=30, vs=0)

                    ...
                    ...
                    ...



        # Switcher contents for Circular Profile -------------------------
        HFrame = FXHorizontalFrame(p=self.groupbox_switcher, opts=0, x=0, y=0,
                                        w=0, h=0, pl=0, pr=0, pt=0, pb=0, hs=30, vs=0)
                    ...
                    ...
                    ...
```

For all 3 profiles, we start by creating a horizontal frame since we wish to have the icon, the column for frame member profile dimensions, and the column for cross brace member profile dimensions side by side. The important thing to note here is that the parent **p** has been set the switcher. This is what makes the switching possible. The horizontal frames for the 3 profiles must all be made children of the switcher in order for it to function. You may also notice that **hs**, which is the horizontal spacing between the children of the horizontal frame, has been set to 30 so that they do not stick close to one another.

```
    # The processUpdates() method is called during each GUI update cycle
    # We will use it to replace the groupbox contents as the user selects a
    # different profile by changing the switcher
    # While we can use transitions to achieve this same effect, for more complex
    # tasks such as comparing different radiobutton or textfield values
    # processUpdates() offers more control and you can call the appropriate
    # methods based on complex decision logic
    # Since processUpdate() is called at each GUI update cycle, it should not
    # execute any time consuming code
    def processUpdates(self):
        # unlike in the case of transitions, here we must use the getValues()
        # method to get the value of a radiobutton, textfield or other widget
        if self.form.profile_Kw.getValue() == 1:
            self.groupbox_switcher.setCurrent(0)
        elif self.form.profile_Kw.getValue() == 2:
            self.groupbox_switcher.setCurrent(1)
        else:
            self.groupbox_switcher.setCurrent(2)
```

The **processUpdates()** method was discussed in section 21.3 of the previous chapter. You learned that this method will be called whenever the GUI is redrawn, and is an alternative to transitions which we used in the dialog box for step 1.

Here we use **processUpdates()** to find out which radio button is currently selected. Recall that the target of all 3 radio buttons was set to **profile_Kw**, and this variable exists in the form for the dialog box (**Step2Form** in **step2Form.py**). Hence we refer to it using the notation **self.form.profile_Kw**. Depending on its value we use the **setCurrent()** method of the **FXSwitcher** to display the correct GUI pane. Each of the 3 **FXHorizontalFrame** layout managers was made a child of the switcher, and they are numbered serially (0, 1, 2..) by default.

Let's now look **step2Form.py** which defines the form associated with the dialog box.

```
# ********************************************************************************
# Custom Beam Frame Analysis GUI Application
# This script is the form for step 2. It will post the step 2 dialog box
# (Step2 in step2DB.py)
# ********************************************************************************

from abaqusGui import *
import step2DB

# Class definition

class Step2Form(AFXForm):

    #-----------------------------------------------------------------------
    def __init__(self, owner):

        # Construct base class.
        AFXForm.__init__(self, owner)

        # Command to execute when OK is clicked
        self.cmd = AFXGuiCommand(self, 'callStep2', 'beamKernel')

        self.profile_Kw = AFXIntKeyword(command=self.cmd, name='profile',
                                        isRequired=TRUE, defaultValue=1)
        self.l_1_Kw = AFXFloatKeyword(command=self.cmd, name='l_1',
                                      isRequired=TRUE, defaultValue=0.075)
        self.l_2_Kw = AFXFloatKeyword(command=self.cmd, name='l_2',
                                      isRequired=TRUE, defaultValue=0.06)
        self.h_1_Kw = AFXFloatKeyword(command=self.cmd, name='h_1',
                                      isRequired=TRUE, defaultValue=0.15)
        self.h_2_Kw = AFXFloatKeyword(command=self.cmd, name='h_2',
                                      isRequired=TRUE, defaultValue=0.12)
        self.b1_1_Kw = AFXFloatKeyword(command=self.cmd, name='b1_1',
                                       isRequired=TRUE, defaultValue=0.12)
        self.b1_2_Kw = AFXFloatKeyword(command=self.cmd, name='b1_2',
                                       isRequired=TRUE, defaultValue=0.11)
        self.b2_1_Kw = AFXFloatKeyword(command=self.cmd, name='b2_1',
                                       isRequired=TRUE, defaultValue=0.12)
        self.b2_2_Kw = AFXFloatKeyword(command=self.cmd, name='b2_2',
                                       isRequired=TRUE, defaultValue=0.08)
        self.t1_1_Kw = AFXFloatKeyword(command=self.cmd, name='t1_1',
                                       isRequired=TRUE, defaultValue=0.02)
        self.t1_2_Kw = AFXFloatKeyword(command=self.cmd, name='t1_2',
                                       isRequired=TRUE, defaultValue=0.01)
        self.t2_1_Kw = AFXFloatKeyword(command=self.cmd, name='t2_1',
                                       isRequired=TRUE, defaultValue=0.02)
        self.t2_2_Kw = AFXFloatKeyword(command=self.cmd, name='t2_2',
                                       isRequired=TRUE, defaultValue=0.01)
        self.t3_1_Kw = AFXFloatKeyword(command=self.cmd, name='t3_1',
                                       isRequired=TRUE, defaultValue=0.04)
        self.t3_2_Kw = AFXFloatKeyword(command=self.cmd, name='t3_2',
                                       isRequired=TRUE, defaultValue=0.02)

        self.a_1_Kw = AFXFloatKeyword(command=self.cmd, name='a_1',
                                      isRequired=TRUE, defaultValue=0.14)
        self.a_2_Kw = AFXFloatKeyword(command=self.cmd, name='a_2',
                                      isRequired=TRUE, defaultValue=0.11)
        self.b_1_Kw = AFXFloatKeyword(command=self.cmd, name='b_1',
                                      isRequired=TRUE, defaultValue=0.1)
        self.b_2_Kw = AFXFloatKeyword(command=self.cmd, name='b_2',
                                      isRequired=TRUE, defaultValue=0.07)
        self.t_1_Kw = AFXFloatKeyword(command=self.cmd, name='t_1',
                                      isRequired=TRUE, defaultValue=0.02)
        self.t_2_Kw = AFXFloatKeyword(command=self.cmd, name='t_2',
                                      isRequired=TRUE, defaultValue=0.01)

        self.r_1_Kw = AFXFloatKeyword(command=self.cmd, name='r_1',
                                      isRequired=TRUE, defaultValue=0.05)
        self.r_2_Kw = AFXFloatKeyword(command=self.cmd, name='r_2',
                                      isRequired=TRUE, defaultValue=0.035)


    #-----------------------------------------------------------------------
    # A getFirstDialog() method MUST be written for a mode (a Form mode)
    # It should return the first dialog box of the mode
    def getFirstDialog(self):

        # Reload the dialog module so that any changes to the dialog are updated.
        reload(step2DB)
        return step2DB.Step2DB(self)


    #-------------------------------------------------------------------
    # A mode is usually activated by sending it a message with its ID set to
    # ID_ACTIVATE
    # This message causes activate() to be called
    # Note that it is NOT necessary to define this activate() method unless you
    # wish to change the default behavior
    # Here we would like it to print a message to the screen before proceeding
    def activate(self):
        print 'Step 2 Mode (form) has been activated'
        # Now must call the base method
        AFXForm.activate(self)


    #-------------------------------------------------------------------
    # doCustomChecks() is called by default right before issueCommands() when you
    # click OK in a dialog mode
    # you do not need to define it unless you wish to change the behavior
    # We define it here in order to check the values of some variables (and make
    # sure they are positive and non-zero)
    # The method must return TRUE if no errors were encountered, if FALSE is
    # returned then command processing will be terminated.
    def doCustomChecks(self):

        # If any of the dimensions for the selected profile are 0 show an error
        # messiage and return FALSE

        if  self.profile_Kw.getValue() == 1:
            if self.l_1_Kw.getValue() <=0 or \
              self.l_2_Kw.getValue() <=0 or \
              self.h_1_Kw.getValue() <=0 or \
              self.h_2_Kw.getValue() <=0 or \
              self.b1_1_Kw.getValue() <=0 or \
              self.b1_2_Kw.getValue() <=0 or \
              self.b2_1_Kw.getValue() <=0 or \
              self.b2_2_Kw.getValue() <=0 or \
              self.t1_1_Kw.getValue() <=0 or \
              self.t1_2_Kw.getValue() <=0 or \
              self.t2_1_Kw.getValue() <=0 or \
              self.t2_2_Kw.getValue() <=0 or \
              self.t3_1_Kw.getValue() <=0 or \
              self.t3_2_Kw.getValue() <=0 :

                showAFXErrorDialog(self.getCurrentDialog(),
                                   'Dimensions must be greater than zero 1')
                return FALSE

        elif self.profile_Kw.getValue() == 2:
            if self.a_1_Kw.getValue() <=0 or \
            self.a_2_Kw.getValue() <=0 or \
            self.b_1_Kw.getValue() <=0 or \
            self.b_2_Kw.getValue() <=0 or \
            self.t_1_Kw.getValue() <=0 or \
            self.t_2_Kw.getValue() <=0 :

              showAFXErrorDialog(self.getCurrentDialog(),
                                 'Dimensions must be greater than zero 2')
              return FALSE

        else :  # self.profile_Kw == 3
            if self.r_1_Kw.getValue() <=0 or \
              self.r_2_Kw.getValue() <=0 :

              showAFXErrorDialog(self.getCurrentDialog(),
                                 'Dimensions must be greater than zero 3')
              return FALSE

        # otherwise return TRUE
        return TRUE

    #----------------------------------------------------------------------
    # issueCommands() is called by default and you do not need to define it
    # We define it here in order to get the command string and print it to the
    # message area to help with debugging
    # By default the issueCommands() method calls the deactive() method to close
    # the dialog box if the user clicked the OK button
    def issueCommands(self):

        # Get the command string that will be sent to the kernel for processing
        cmdstr = self.getCommandString()

        # Write this command string to the message area so we know it executed
        getAFXApp().getAFXMainWindow().writeToMessageArea(cmdstr)

        # The sendCommandString() method is usually called by default.
        # However since we are defining issueCommands() we now need to manually
        # call this method
        self.sendCommandString(cmdstr)

        # Deactivate the form if the user presses the OK button of the dialog box
        self.deactivateIfNeeded()
        return TRUE
```

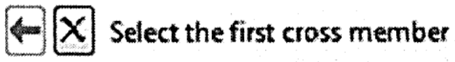
The only syntax here you haven't seen previously is the **doCustomChecks()** method. When you click **OK** in a dialog box, **doCustomChecks()** is called before **issueCommands()** so that last minute checks can be made before the keywords are sent to the kernel script. **doCustomChecks()** is optional which is why we have not used it previously. After the checks are performed, it must return True for the program.

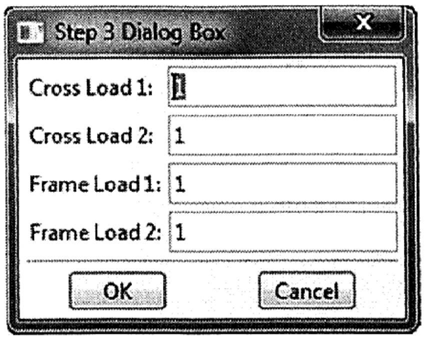
```
    def doCustomChecks(self):

        # If any of the dimensions for the selected profile are 0 show an error
        # messiage and return FALSE

        if  self.profile_Kw.getValue() == 1:
            if self.l_1_Kw.getValue() <=0 or \
              self.l_2_Kw.getValue() <=0 or \
              self.h_1_Kw.getValue() <=0 or \
              self.h_2_Kw.getValue() <=0 or \
              self.b1_1_Kw.getValue() <=0 or \
              self.b1_2_Kw.getValue() <=0 or \
              self.b2_1_Kw.getValue() <=0 or \
              self.b2_2_Kw.getValue() <=0 or \
              self.t1_1_Kw.getValue() <=0 or \
              self.t1_2_Kw.getValue() <=0 or \
              self.t2_1_Kw.getValue() <=0 or \
              self.t2_2_Kw.getValue() <=0 or \
              self.t3_1_Kw.getValue() <=0 or \
              self.t3_2_Kw.getValue() <=0 :

                showAFXErrorDialog(self.getCurrentDialog(),
                                   'Dimensions must be greater than zero 1')
                return FALSE

        elif self.profile_Kw.getValue() == 2:
            if self.a_1_Kw.getValue() <=0 or \
              self.a_2_Kw.getValue() <=0 or \
              self.b_1_Kw.getValue() <=0 or \
              self.b_2_Kw.getValue() <=0 or \
              self.t_1_Kw.getValue() <=0 or \
              self.t_2_Kw.getValue() <=0 :

                showAFXErrorDialog(self.getCurrentDialog(),
                                   'Dimensions must be greater than zero 2')
                return FALSE

        else :  # self.profile_Kw == 3
            if self.r_1_Kw.getValue() <=0 or \
              self.r_2_Kw.getValue() <=0 :

                showAFXErrorDialog(self.getCurrentDialog(),
                                   'Dimensions must be greater than zero 3')
                return FALSE

        # otherwise return TRUE
        return TRUE
```

Here we check the value of every dimension variable for the selected profile (we only care about the variables associated with the profile that is selected). If any of them is 0 or a negative number, we return False, which will prevent **issueCommands()** from being called. In addition we display an error message using **showAFXErrorDialog()**.

21.4.8 Step 3 Procedure and Dialog Box

The procedure mode is defined in the script **step3Procedure.py** and the dialog box in **step3DB.py**. These two scripts create a procedure which prompts the user to select two cross brace members followed by two frame members, and then displays a dialog box where the loads for each of these selected members can be specified. The scripts associate the members picked in the viewport with variables, and call the appropriate kernel command when the user clicks OK in the dialog box.


Select the first cross member



Cross Load 1: 1
Cross Load 2: 1
Frame Load 1: 1
Frame Load 2: 1

Let's briefly examine **step3DB.py**.

```
# ****************************************************************************
# Custom Beam Frame Analysis GUI Application
# This script defines a dialog box that will be posted by a procedure
# (Step3Procedure in step3Procedure.py)
# ****************************************************************************

from abaqusGui import *

# Class definition

class Step3DB(AFXDataDialog):

    #--------------------------------------------------------------------
    def __init__(self, form):

        # DIALOG_ACTIONS_SEPARATOR places a horizontal line/separator between the
        # contents of the dialog box and the OK/CANCEL buttons at the bottom
        # DIALOG_BAILOUT displays a message "Save changes made in the xyz dialog?"
        # if the user clicks Cancel after changing some values in the dialog box
        AFXDataDialog.__init__(self, form, 'Step 3 Dialog Box',
                               self.OK|self.CANCEL, DIALOG_ACTIONS_SEPARATOR)

        va = AFXVerticalAligner(self)
        AFXTextField(va, 20, 'Cross Load 1:', form.crossload1Kw, 0)
        AFXTextField(va, 20, 'Cross Load 2:', form.crossload2Kw, 0)
        AFXTextField(va, 20, 'Frame Load 1:', form.frameload1Kw, 0)
        AFXTextField(va, 20, 'Frame Load 2:', form.frameload2Kw, 0)


    #--------------------------------------------------------------------
    # The show() method is called by default whenever the dialog box is to be
    # displayed.
    # For example in modifiedCanvasToolsetGui.py you have the statement
    # AFXMenuCommand(self, viewport_menu_with_contents, 'Custom Menu Item',  None,
    # DemoForm(self), AFXMode.ID_ACTIVATE)
    # So whenever the custom menu item is clicked, the activate() method of the
    # mode is called, and this in turn calls the show() method of the dialog box.
    # Note that this method does NOT need to be defined here if we wish to leave
    # the behavior at default
    # Here however we wish to modify the show method to also print a message to
    # the screen (aside from opening the window which it does by default).
    def show(self):
        print 'Step 3 Dialog box will be displayed'
        # Now must call the base show() command
        AFXDataDialog.show(self)


    #--------------------------------------------------------------------
    # The hide() method is the opposite of show(). It is called by default
    # whenever the dialog box is to be hidden
    # Note that this method does NOT need to be defined here if we wish to leave
    # the behavior at default
    # Here however we wish to modify the hide method to also print a message to
    # the screen (aside from closing the window which it does by default).
    def hide(self):
        print 'Step 3 Dialog box will be hidden'

        # Now must call the base hide() command
        AFXDataDialog.hide(self)

        # For some reason Abaqus (version 6.10) shows a blank viewport at the end
        # of the procedure. Hence we shall force it to show the assembly again.
        sendCommand('root_assembly= mdb.models[\'Beam Frame\'].rootAssembly \n' +\
            'session.viewports[\'Viewport: 1\'].setValues(displayedObject=' + \
                                                      'root_assembly)  \n' + \
           'session.viewports[\'Viewport: 1\'].assemblyDisplay.setValues' + \
             '(loads=ON, bcs=ON, predefinedFields=ON, connectors=ON)  \n' + \
           'session.viewports[\'Viewport: 1\'].assemblyDisplay.setValues' + \
                                                   '(step=\'Apply Loads\')  \n' + \
          'root_assembly.regenerate()')
    print 'Procedure Complete'
```

This dialog box is very simple compared to the ones created for 'Step 1' and 'Step 2'. The only statement you might find a little confusing is the use of the **sendCommand()** method on the second to last line.

```
# For some reason Abaqus (version 6.10) shows a blank viewport at the end
# of the procedure. Hence we shall force it to show the assembly again.
sendCommand('root_assembly= mdb.models[\'Beam Frame\'].rootAssembly \n' +\
    'session.viewports[\'Viewport: 1\'].setValues(displayedObject=' + \
                                              'root_assembly)  \n' + \
    'session.viewports[\'Viewport: 1\'].assemblyDisplay.setValues' + \
      '(loads=ON, bcs=ON, predefinedFields=ON, connectors=ON)  \n' + \
    'session.viewports[\'Viewport: 1\'].assemblyDisplay.setValues' + \
                                     '(step=\'Apply Loads\')  \n' + \
    'root_assembly.regenerate()')
```

However the comments on the first two lines explain the reasoning behind it quite succinctly.

Now let's turn our attention to **step3Procedure.py**.

```
# ****************************************************************************
# Custom Beam Frame Analysis GUI Application
# This script is the procedure mode for step 3. It will prompt the user to
# select beams in the viewport and post the step 3 dialog box
# (Step3DB in step3DB.py)

# Created for the book "Python Scripts for Abaqus - Learn by Example"
# Author: Gautam Puri
# ****************************************************************************

from abaqusGui import *
from step3DB import Step3DB

# Class definition

class Step3Procedure(AFXProcedure):

    #-------------------------------------------------------------------
    def __init__(self, owner):

        # Construct base class.
        AFXProcedure.__init__(self, owner)

        # Command to execute when OK is clicked
        self.cmd = AFXGuiCommand(self, 'callStep3', 'beamKernel')
        self.crossloadedge1Kw = AFXObjectKeyword(command=self.cmd,
                                                 name='crossloadedge1',
                                                 isRequired=TRUE)
        self.crossloadedge2Kw = AFXObjectKeyword(command=self.cmd,
                                                 name='crossloadedge2',
                                                 isRequired=TRUE)
        self.frameloadedge1Kw = AFXObjectKeyword(command=self.cmd,
                                                 name='frameloadedge1',
                                                 isRequired=TRUE)
        self.frameloadedge2Kw = AFXObjectKeyword(command=self.cmd,
                                                 name='frameloadedge2',
                                                 isRequired=TRUE)

        self.crossload1Kw = AFXFloatKeyword(command=self.cmd, name='crossload1',
                                            isRequired=TRUE, defaultValue = 1)
        self.crossload2Kw = AFXFloatKeyword(command=self.cmd, name='crossload2',
                                            isRequired=TRUE, defaultValue = 1)
        self.frameload1Kw = AFXFloatKeyword(command=self.cmd, name='frameload1',
                                            isRequired=TRUE, defaultValue = 1)
        self.frameload2Kw = AFXFloatKeyword(command=self.cmd, name='frameload2',
                                            isRequired=TRUE, defaultValue = 1)


    #----------------------------------------------------------------------
    # A getFirstStep() method MUST be written for a mode (a Procedure mode)
    def getFirstStep(self):
        self.step1 = AFXPickStep(self, self.crossloadedge1Kw,
                                 'Select the first cross member', EDGES)
        # Add a refinement to pick only straight edges (we actually only have
        # straight edges in our model so this is more for demonstration purposes)
        #self.step1.setEdgeRefinements(AFXPickStep.STRAIGHT)
        return self.step1

    # ---------------------------------------------------------------------
    # A getNextStep() method MUST be written for modes containing more than one
    # step
    # The previous step is passed to getNextStep() so you can determine where the
    # user is in the sequence of steps
    # The next step in the sequence should be returned by getNextStep(), or
    # 'None' if there are no more steps
    def getNextStep(self, previousStep):

        if previousStep == self.step1:
            self.step2 = AFXPickStep(self, self.crossloadedge2Kw,
                                     'Select the second cross member', EDGES)
            return self.step2

        elif previousStep == self.step2:
            self.step3 = AFXPickStep(self, self.frameloadedge1Kw,
                                     'Select the first frame member', EDGES)
            return self.step3
        elif previousStep == self.step3:
            self.step4 = AFXPickStep(self, self.frameloadedge2Kw,
                                     'Select the second frame member', EDGES)
            return self.step4

        elif previousStep == self.step4:
            db = Step3DB(self)
            self.step5 = AFXDialogStep(self, db, 'Specify loads in dialog box')
            return self.step5

        else:
            return None




    #-----------------------------------------------------------------------
    # A mode is usually activated by sending it a message with its ID set to
    # ID_ACTIVATE
    # This message causes activate() to be called
    # Note that it is NOT necessary to define this activate() method unless you
    # wish to change the default behavior
    # Here we would like it to print a message to the screen before proceeding
    def activate(self):

        print 'Step 3 Mode (procedure) has been activated'

        # We must check to see if a part has been displayed otherwise the user
        # cannot pick anything
        if getDisplayedObjectType() != PART:

            showAFXErrorDialog(getAFXApp().getAFXMainWindow(),
                        'There is no part displayed in the current viewport. ' + \
                        'A part must be displayed to continue')
            return

        else:

            # The edges must be picked in the assembly module. The viewport
            # currently displays the part module. If the user picks the edges
            # here the LineLoad() method called in beamKernel will not function
            # correctly
            # To change the viewport to the assembly module we use the following
            # commands
            # root_assembly = mdb.models['Beam Frame'].rootAssembly
            # session.viewports['Viewport: 1'].setValues(displayedObject=
            #                                                    root_assembly)
            # However these commands cannot be executed in a GUI script
            # The session object can only be used in a Kernel script
            # To issue a kernel command directly from the GUI we can use the
            #                                                sendCommand() method
        # To the sendCommand() method we pass the statements that should be
        # executed
        # We separate multiple statements with a \n as is done here. Note
        # that you cannot leave a space after the \n because that means there
        # is a space in front of the next statement, and Python being will
        # object to this wrong indentation
        sendCommand('root_assembly = ' + \
                              'mdb.models[\'Beam Frame\'].rootAssembly \n' + \
                      'session.viewports[\'Viewport: 1\'].setValues(displa' + \
                                              'yedObject=root_assembly)')

        # Call the base method
        AFXProcedure.activate(self)
```

Let's examine what is new here.

```
    #-----------------------------------------------------------------
    # A getFirstStep() method MUST be written for a mode (a Procedure mode)
    def getFirstStep(self):
        self.step1 = AFXPickStep(self, self.crossloadedge1Kw,
                                 'Select the first cross member', EDGES)
        # Add a refinement to pick only straight edges (we actually only have
        # straight edges in our model so this is more for demonstration purposes)
        #self.step1.setEdgeRefinements(AFXPickStep.STRAIGHT)
        return self.step1
```

Just as you have a **getFirstDialog()** method in a form mode, you must define a **getFirstStep()** method in a procedure mode. In order to make a user pick something in the viewport an **AFXPickStep()** is used. This method accepts a number of parameters. The 4 we have used are **owner**, **keyword**, **prompt** and **entitiesToPick** (even though the parameter names are not explicitly typed here). **owner** is the procedure that creates the step. **keyword** is the variable associated with what is picked, which can be passed to the kernel as part of the **AFXGuiCommand()** method. **prompt** is the text string to display in the prompt area. **entitiesToPick** is the type of entity that can/should be picked. Another parameter which is not used here but might be useful is **numberToPick**, which determines how many entities may be picked – it defaults to 1.

The use of **AFXPickStep** here causes the prompt message 'Select the first cross member' to be displayed in the prompt area. The user is able to select edges thanks to **EDGES** being set in **entitiesToPick**.

```
    # -----------------------------------------------------------------
    # A getNextStep() method MUST be written for modes containing more than one
    # step
    # The previous step is passed to getNextStep() so you can determine where the
    # user is in the sequence of steps
    # The next step in the sequence should be returned by getNextStep(), or
    # 'None' if there are no more steps
    def getNextStep(self, previousStep):

        if previousStep == self.step1:
            self.step2 = AFXPickStep(self, self.crossloadedge2Kw,
                                     'Select the second cross member', EDGES)
            return self.step2

        elif previousStep == self.step2:
            self.step3 = AFXPickStep(self, self.frameloadedge1Kw,
                                     'Select the first frame member', EDGES)
            return self.step3

        elif previousStep == self.step3:
            self.step4 = AFXPickStep(self, self.frameloadedge2Kw,
                                     'Select the second frame member', EDGES)
            return self.step4

        elif previousStep == self.step4:
            db = Step3DB(self)
            self.step5 = AFXDialogStep(self, db, 'Specify loads in dialog box')
            return self.step5

        else:
            return None
```

Since our procedure involves more than one step we must define a **getNextStep()** method. Aside from **self** this method also receives **previousStep** which is exactly what the name implies. We use this variable to find out what the previous step was, and using a sequence of **if-elif** statements we call the subsequent step using **AFXPickStep()**. After 4 pick steps, we call an **AFXDialogStep()**. Recall that procedure modes can also be used to launch a dialog box (whereas form modes cannot launch pick steps) and **AFXDialogStep()** is the method used to accomplish this.

AFXDialogStep() accepts 3 parameters – **owner**, which is the procedure that created the step, **dialog**, which is the dialog box to post, and **prompt**, which is the text string to display.

Once all the steps are complete we return **None** indicating that there are no further steps.

```
    #-------------------------------------------------------------------
    # A mode is usually activated by sending it a message with its ID set to
    # ID_ACTIVATE
    # This message causes activate() to be called
    # Note that it is NOT necessary to define this activate() method unless you
    # wish to change the default behavior
    # Here we would like it to print a message to the screen before proceeding
    def activate(self):

        print 'Step 3 Mode (procedure) has been activated'

        # We must check to see if a part has been displayed otherwise the user
        # cannot pick anything
        if getDisplayedObjectType() != PART:

            showAFXErrorDialog(getAFXApp().getAFXMainWindow(),
                        'There is no part displayed in the current viewport. ' + \
                        'A part must be displayed to continue')
            return

        else:

            # The edges must be picked in the assembly module. The viewport
            # currently displays the part module. If the user picks the edges
            # here the LineLoad() method called in beamKernel will not function
            # correctly
            # To change the viewport to the assembly module we use the following
            # commands
            # root_assembly = mdb.models['Beam Frame'].rootAssembly
            # session.viewports['Viewport: 1'].setValues(displayedObject=
            #                                                  root_assembly)
            # However these commands cannot be executed in a GUI script
            # The session object can only be used in a Kernel script
            # To issue a kernel command directly from the GUI we can use the
            #                                            sendCommand() method
            # To the sendCommand() method we pass the statements that should be
            # executed
            # We separate multiple statements with a \n as is done here. Note
            # that you cannot leave a space after the \n because that means there
            # is a space in front of the next statement, and Python being will
            # object to this wrong indentation
            sendCommand('root_assembly = ' + \
                          'mdb.models[\'Beam Frame\'].rootAssembly \n' + \
                   'session.viewports[\'Viewport: 1\'].setValues(displa' + \
                                          'yedObject=root_assembly)')

            # Call the base method
            AFXProcedure.activate(self)
```

Just as with form modes, **activate()** is called when a procedure mode begins running. We can do a check here to make sure a part is actually displayed. If not, it probably means the user has not completed 'Step 1'. It is not possible to pick any edges if there is no part displayed, hence we need to make sure this situation does not occur. We display an error using **showAFXErrorDialog()** informing the user that no part is displayed.

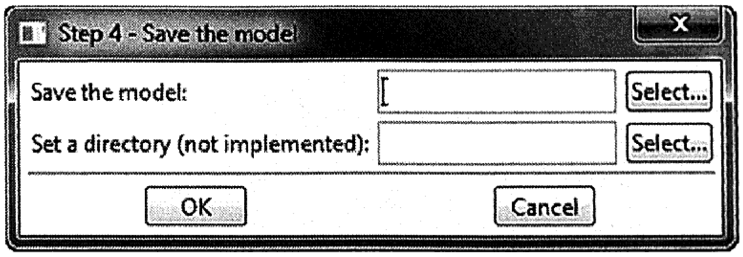
If on the other hand a part is displayed, we must switch to the assembly module. If you were using Abaqus/CAE you would select the edges for the load in the assembly module, hence even in this procedure we must switch out of the part module into the assembly module. The comments (there's an entire paragraph of them) help you understand how this is accomplished.

Finally the base **AFXProcedure.activate()** method must be called to activate the procedure mode.

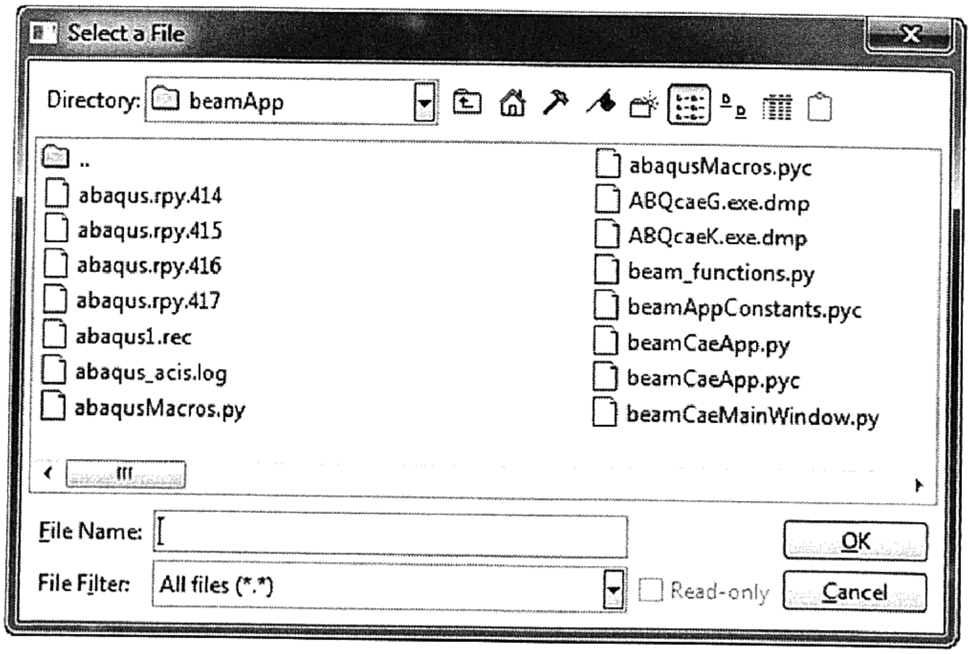
21.4.9 Step 4 Form and Dialog Box

The form mode is defined in the script **step4Form.py** and the dialog box in **step4DB.py**.

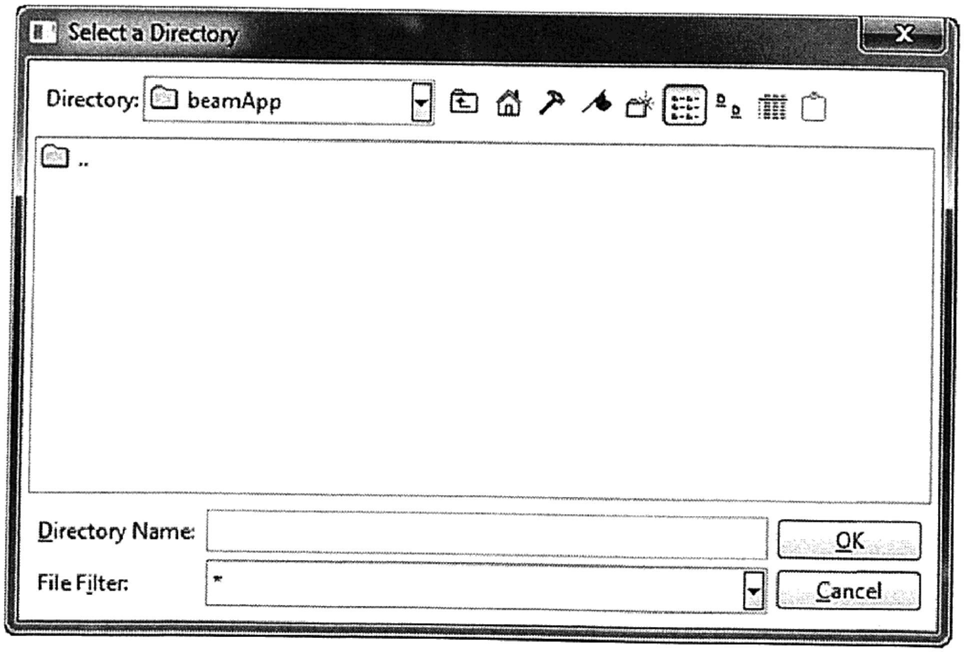
These two scripts create a dialog box which lets the user decide where to save the model. The **Select...** button next to it opens a file browser. The dialog box also lets you select a directory, but this feature is only included for demonstration purposes and is not actually implemented in the script.



If the user clicks the first **Select...** button he will be provided a file selection window



If the user clicks **Select...** next to 'set a directory' he will get the directory selection window.



When the **OK** button is finally clicked, the entire model is saved at the specified file location.

Let's first take a quick look at **step4DB.py**.

```
# ****************************************************************************
# Custom Beam Frame Analysis GUI Application
# This script defines a dialog box that will be posted by a form (Step4Form in
# step4Form.py)

# Created for the book "Python Scripts for Abaqus - Learn by Example"
# Author: Gautam Puri
# ****************************************************************************


from abaqusGui import *

# Class definition

class Step4DB(AFXDataDialog):

    [
        ID_FILE,
        ID_DIRECTORY,
    ] = range(AFXToolsetGui.ID_LAST, AFXToolsetGui.ID_LAST+2)

    #-----------------------------------------------------------------------
    def __init__(self, form):

        # DIALOG_ACTIONS_SEPARATOR places a horizontal line/separator between the
        # contents of the dialog box and the OK/CANCEL buttons at the bottom
        # DIALOG_BAILOUT displays a message "Save changes made in the xyz dialog?"
        # if the user clicks Cancel after changing some values in the dialog box
        AFXDataDialog.__init__(self, form, 'Step 4 - Save the model',
                               self.OK|self.CANCEL, DIALOG_ACTIONS_SEPARATOR)

        self.form = form
        self.fileDb = None
        self.dirDb = None

        FXMAPFUNC(self, SEL_COMMAND, self.ID_FILE, Step4DB.onCmdFile)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_DIRECTORY, Step4DB.onCmdDirectory)

        va = AFXVerticalAligner(self, LAYOUT_FILL_X)

        # File name
        hf = FXHorizontalFrame(va, LAYOUT_FILL_X, 0,0,0,0, 0,0,0,0)
        AFXTextField(hf, 20, 'Save the model:', form.fileNameKw, 0, LAYOUT_FILL_X)
        FXButton(hf, 'Select...', None, self, self.ID_FILE)

        # Directory name
        hf = FXHorizontalFrame(va, LAYOUT_FILL_X, 0,0,0,0, 0,0,0,0)
        AFXTextField(hf, 20, 'Set a directory (not implemented):', form.dirNameKw,
                                                                     0, LAYOUT_FILL_X)
        FXButton(hf, 'Select...', None, self, self.ID_DIRECTORY)


    def onCmdFile(self, sender, sel, ptr):

        if not self.fileDb:
            patterns = 'All files (*.*)\nOutput Database (*.cae)'
            self.fileDb = AFXFileSelectorDialog(owner=self, title='Select a File',
                                    pathNameKw=self.form.fileNameKw,
                                    readOnlyKw=self.form.fileReadOnlyBoolKw,
                                    mode=AFXSELECTFILE_ANY,
                                    patterns=patterns,
                                    patternIndexTgt=self.form.filePatternIndexTgt)
            self.fileDb.setReadOnlyPatterns('*.cae')
            self.fileDb.showReadOnly(show=True)
            self.fileDb.create()

        self.fileDb.showModal()

        return 1


    def onCmdDirectory(self, sender, sel, ptr):

        if not self.dirDb:
            self.dirDb = AFXFileSelectorDialog(owner=self,
                                               title='Select a Directory',
                                               pathNameKw=self.form.dirNameKw,
                                               readOnlyKw=None,
                                               mode=AFXSELECTFILE_DIRECTORY)
            self.dirDb.create()

        self.dirDb.showModal()

        return 1


    #-----------------------------------------------------------------------
    # The show() method is called by default whenever the dialog box is to be
    # displayed.
    # For example in modifiedCanvasToolsetGui.py you have the statement
    # AFXMenuCommand(self, viewport_menu_with_contents, 'Custom Menu Item',  None,
    # DemoForm(self), AFXMode.ID_ACTIVATE)
    # So whenever the custom menu item is clicked, the activate() method of the
    # mode is called, and this in turn calls the show() method of the dialog box.
    # Note that this method does NOT need to be defined here if we wish to leave
    # the behavior at default
    # Here however we wish to modify the show method to also print a message to
    # the screen (aside from opening the window which it does by default).
    def show(self):
        print 'Step 4 Dialog box will be displayed'
        # Now must call the base show() command
        AFXDataDialog.show(self)


    #-------------------------------------------------------------------
    # The hide() method is the opposite of show(). It is called by default
    # whenever the dialog box is to be hidden
    # Note that this method does NOT need to be defined here if we wish to leave
    # the behavior at default
    # Here however we wish to modify the hide method to also print a message to
    # the screen (aside from closing the window which it does by default).
    def hide(self):
        print 'Step 4 Dialog box will be hidden'
        # Now must call the base hide() command
        AFXDataDialog.hide(self)
```

The new things here are the methods **onCmdFile()** and **onCmdDirectory()** so let's look closely at those.

```
    def onCmdFile(self, sender, sel, ptr):

        if not self.fileDb:
            patterns = 'All files (*.*)\nOutput Database (*.cae)'
            self.fileDb = AFXFileSelectorDialog(owner=self, title='Select a File',
                                  pathNameKw=self.form.fileNameKw,
                                  readOnlyKw=self.form.fileReadOnlyBoolKw,
                                  mode=AFXSELECTFILE_ANY,
                                  patterns=patterns,
                                  patternIndexTgt=self.form.filePatternIndexTgt)
            self.fileDb.setReadOnlyPatterns('*.cae')
            self.fileDb.showReadOnly(show=True)
            self.fileDb.create()

        self.fileDb.showModal()

        return 1
```

AFXFileSelector dialog is used to ask the user for a file or directory name. It accepts the following arguments: **owner** is the owner of the browser dialog box, **title** is the dialog box title, **pathNameTgt** is the path name target i.e., the variable in the path will be stored, **readOnlyKw** is a Boolean specifying if it should be read only or not, **mode** is the file selection mode, **patterns** is the file filter patterns, and **patternIndexTgt** is the index used to select a file filter pattern when the dialog box is posted. The **mode** is set to **AFXSELECTFILE_ANY** to display a file selection dialog. Other options are **AFXSELECTFILE_EXISTING** which allows the selection of an existing file only,

AFXSELECTFILE_MULTIPLE which allows the selection of multiple existing files, **AFXSELECTFILE_DIRECTORY** (which allows the selection of an existing directory) and **AFXSELECTFILE_REMOTE_HOST** which allows opening of files on a remote host.

The statement **self.fileDb** checks to make sure the file selector dialog box is not already displayed. We set the patterns using the format shown in the variable **patterns**, separating the patterns with newline characters.

setReadOnlyPatters() sets the patterns that force the display of the read-only checkbox. **showReadOnly()** shows the read-only checkbox. Since we have set the pattern to '*.cae' the read-only checkbox will only be shown when the user sets the file filter to 'Output database (*.cae)'. We do not actually add any functionality to our program with this feature in this example, it has only been included for demonstration purposes.

The **create()** method must be called to create the **AFXFileSelectorDialog** object. The **showModal()** method must be called to actually display the file selector dialog box.

```
    def onCmdDirectory(self, sender, sel, ptr):

        if not self.dirDb:
            self.dirDb = AFXFileSelectorDialog(owner=self,
                                               title='Select a Directory',
                                               pathNameKw=self.form.dirNameKw,
                                               readOnlyKw=None,
                                               mode=AFXSELECTFILE_DIRECTORY)
            self.dirDb.create()

        self.dirDb.showModal()

        return 1
```

The procedure for creating a directory selection dialog box is quite similar to that for a file selector dialog box. The only difference is that the mode is set to **AFXSELECTFILE_DIRECTORY**.

The form mode is in the file **step4Form.py**.

```
# ***********************************************************************************
# Custom Beam Frame Analysis GUI Application
# This script is the form for step 4. It will post the step 4 dialog box
# (Step4DB in step4DB.py)
# Created for the book "Python Scripts for Abaqus - Learn by Example"
# Author: Gautam Puri
# ****************************************************************************


from abaqusGui import *
import step4DB

# Class definition

class Step4Form(AFXForm):

    #-------------------------------------------------------------------------
    def __init__(self, owner):

        # Construct base class.
        AFXForm.__init__(self, owner)

        # Command to execute when OK is clicked
        self.cmd = AFXGuiCommand(self, 'callStep4', 'beamKernel')

        self.fileNameKw = AFXStringKeyword(command=self.cmd, name='fileName',
                                                  isRequired=TRUE)
        self.fileReadOnlyBoolKw = AFXBoolKeyword(command=self.cmd,
                                                 name='fileReadOnlyBool',
                                                 isRequired=TRUE,
                                                 defaultValue=FALSE)
        self.dirNameKw = AFXStringKeyword(command=self.cmd, name='dirName',
                                                 isRequired=TRUE)

        # Target to remember dialog pattern
        self.filePatternIndexTgt = AFXIntTarget(0)

    #-------------------------------------------------------------------------
    # A getFirstDialog() method MUST be written for a mode (a Form mode)
    # It should return the first dialog box of the mode
    def getFirstDialog(self):

        # Reload the dialog module so that any changes to the dialog are updated.
        reload(step4DB)
        return step4DB.Step4DB(self)


    #-------------------------------------------------------------------------
    # A mode is usually activated by sending it a message with its ID set to
    # ID_ACTIVATE
    # This message causes activate() to be called
    # Note that it is NOT necessary to define this activate() method unless you
    # wish to change the default behavior
    # Here we would like it to print a message to the screen before proceeding
    def activate(self):
        print 'Step 4 Mode (form) has been activated'
        # Now must call the base method
        AFXForm.activate(self)
```

Once again, you've seen most of these statements and methods used before. The one difference is the statement

```
        self.filePatternIndexTgt = AFXIntTarget(0)
```

which remembers the pattern that should be selected by default when the file selection dialog is displayed.

21.5 Summary

You've now created a fully functional custom GUI application and have a good understanding of the steps involved in scripting one. GUI design is a fairly complicated subject and you'll probably spend a lot of time debugging code, but hopefully the scripts from this chapter and the previous one will give you a great starting point for any GUI applications you develop.

Abaqus offers a number of widgets and layout managers aside from the ones used in this example so it is recommended that you take a look at the 'Abaqus GUI Toolkit User's Manual' and the 'Abaqus GUI Toolkit Reference Manual' for further information.

22

Plug-ins

22.1 Introduction

In this chapter we will talk about creating plug-ins. Plug-ins are scripts available to a user in Abaqus/CAE through the Plug-ins menu. They help extend the functionality of Abaqus. A plug-in can be a simple kernel script that performs a routine task, the same sort of script you could run through **File > Run Script...** In this scenario the advantage is that of convenience - the script is easily accessible to everyone who is using Abaqus/CAE once it is packaged as a plug-in. On the other hand the plug-in can be a GUI script which displays a custom interface prompting the user to input data and select items in the viewport. If all you need is a little extra functionality, creating a plug-in requires less work than writing an entire custom GUI application. However a plug-in cannot modify or remove Abaqus/CAE modules and toolsets the way a custom application can.

22.2 Methodology

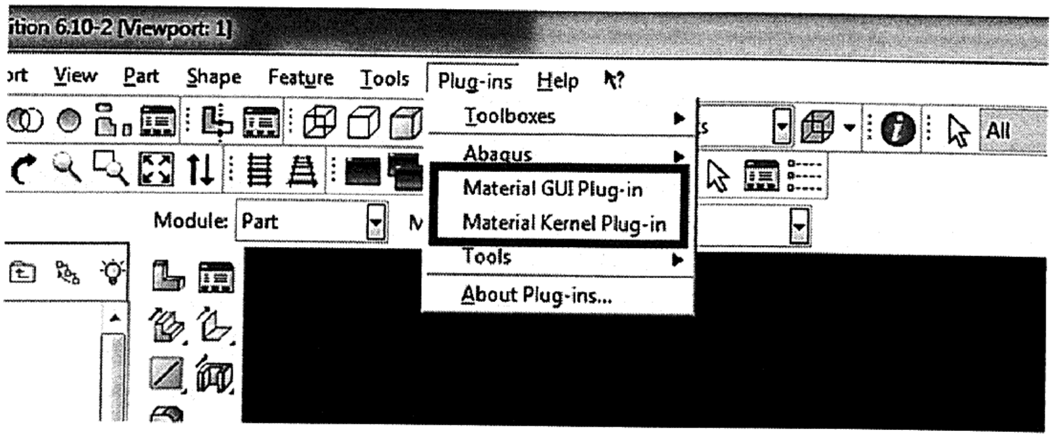
All plug-ins must follow the naming convention *_plugin.py. This helps Abaqus identify a script that is a plug-in. A plug-in may consist of more than one script; however the rest of the scripts do not need to follow this naming convention. Presumably your *_plugin.py script has **import** statements which will cause the other scripts to be imported as needed. Also, it is recommended that you store all these related scripts (and other files such as icons) in the same directory unless you wish to mess with the PYTHONPATH variable.

Abaqus/CAE automatically searches for plug-ins in certain directories while starting up. All plug-ins detected are added to the **Plug-ins** menu. Your plug-ins must be placed in one of these key locations. By default Abaqus searches for a folder called **abaqus_plugins**, first in the Abaqus directory **(abq_dir\cae\abaqus_plugins\)**, then the home directory **(home_dir\abaqus_plugins\)**, and finally the current directory **(cur_dir\abaqus_plugins\)**.

If a plug-in is a kernel plug-in, Abaqus/CAE sends commands of the form *module_name.function_name* to the kernel. If the plug-in is a GUI plug-in, Abaqus/CAE sends a command of the type **ID_ACTIVATE, SEL_COMMAND** to the GUI object created for the plug-in.

22.3 Learn by Example

Since kernel and GUI plug-ins operate slightly differently, we're going to create one of each. We shall call them 'Material Kernel Plug-in' and 'Material GUI Plug-in'. We won't write too much new code, we'll just reuse statements written in previous chapters and package them as plug-ins.



22.3.1 Kernel Plug-in Example

We will use the first script we wrote in this book, the one in Chapter 1, section 1.2. If you recall, all this script does is create 3 materials. We have placed it inside a function, **createMaterials()**, which our plug-in can call.

We place the contents in **materialkernelscript.py**. Here is the listing:

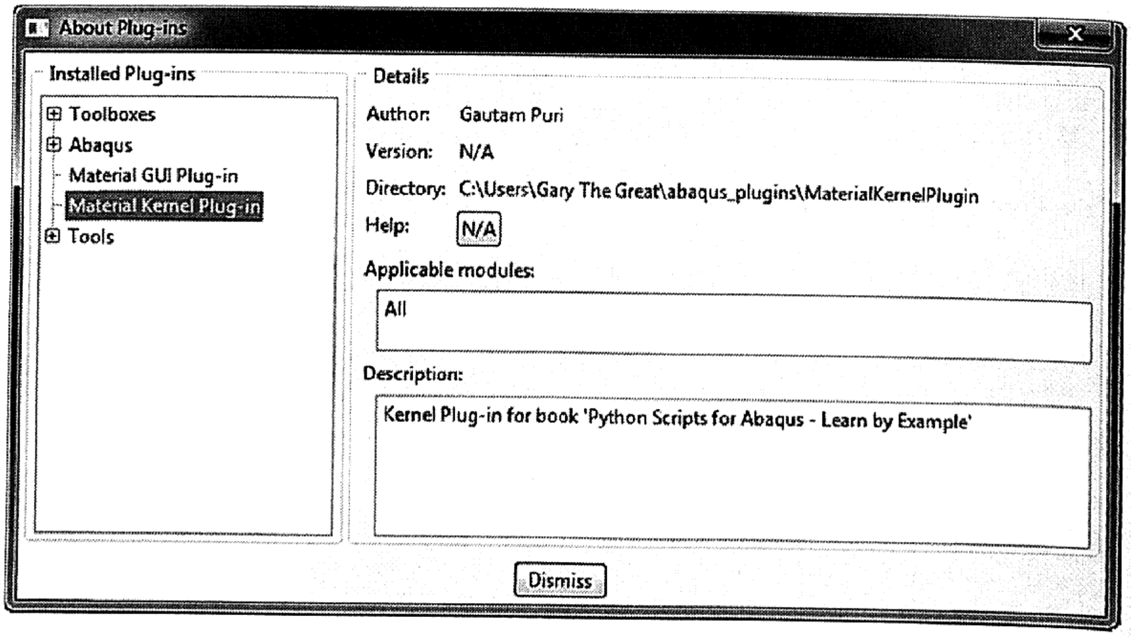
```
# ********************************************************************************
# Material Kernel Plug-in
# This script sends commands to the kernel to create the materials
# ********************************************************************************

from abaqus import *
from abaqusConstants import *

def createMaterials():

    mdb.models['Model-1'].Material('Titanium')
    mdb.models['Model-1'].materials['Titanium'].Density(table=((4500, ), ))
    mdb.models['Model-1'].materials['Titanium'].Elastic(table=((200E9, 0.3), ))

    mdb.models['Model-1'].Material('AISI 1005 Steel')
    mdb.models['Model-1'].materials['AISI 1005 Steel'].Density(table=((7872, ), ))
    mdb.models['Model-1'].materials['AISI 1005 Steel'].Elastic(table=((200E9, 0.29),
))

    mdb.models['Model-1'].Material('Gold')
    mdb.models['Model-1'].materials['Gold'].Density(table=((19320, ), ))
    mdb.models['Model-1'].materials['Gold'].Elastic(table=((77.2E9, 0.42), ))
```

We now create the plug-in. Here are the contents of 'materialkernel_plugin.py'

```
# ******************************************************************************
# Material Kernel Plug-in
# This script registers the material kernel plug-in
# ******************************************************************************

from abaqusGui import getAFXApp

toolset = getAFXApp().getAFXMainWindow().getPluginToolset()
toolset.registerKernelMenuButton(buttonText='Material Kernel Plug-in',
                    moduleName='materialkernelscript',
                    functionName='createMaterials()',
                    author = 'Gautam Puri',
                    description='Kernel Plug-in for book \'Python Scripts for ' + \
                                        'Abaqus - Learn by Example\'')
```

These are the statements required to create a kernel plug-in. The **registerKernelMenuButton()** method registers a kernel plug-in in the Plug-ins menu. It must be used on a toolset, which we obtain using **getAFXApp().getAFXMainWindow().getPluginToolset()**. Its required arguments are **moduleName**, **functionName** and **buttonText.**

buttonText is the text to be displayed as the name of the plug-in in the **Plug-ins** menu. Here we name it 'Material Kernel Plugin'. **moduleName** is the name of the module to import. In our case this is **materialkernelscript.py**, which contains the actual functionality. **functionName** is the name of the function to execute inside the module imported using **moduleName**. It is for this purpose that we placed all the statements inside a function – **createMaterials()**. There are also a number of optional arguments. The ones we've used here are **author** and **description**. The contents of these are displayed in the **Plug-ins > About Plug-ins** dialog box as displayed in the figure. Another optional parameter that might be useful to you (although we haven't used it here) is **applicableModules**. The plug-in will only be available in those modules specified in this parameter. You must give it a sequence of module names, of which the possible values are **Part**, **Property**, **Assembly**, **Step**, **Interaction**, **Load**, **Mesh**, **Job**, **Visualization** and **Sketch**. The default is the SymbolicConstant **ALL**.

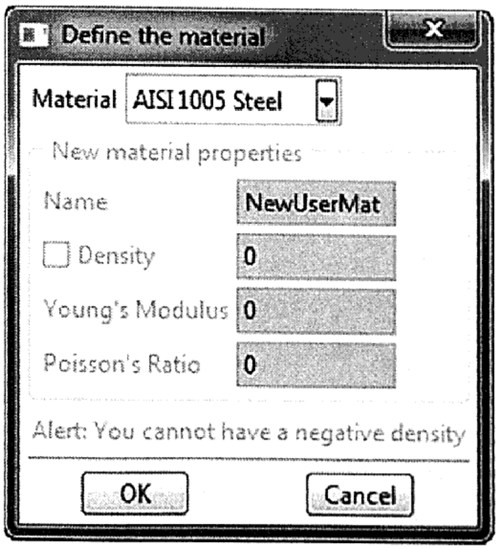


Author: Gautam Puri
Version: N/A
Directory: C:\Users\Gary The Great\abaqus_plugins\MaterialKernelPlugin
Applicable modules:
All
Description:
Kernel Plug-in for book 'Python Scripts for Abaqus - Learn by Example'

This is all it takes to turn your kernel script into a functional kernel plug-in.

22.3.2 GUI Plug-in Example

We will reuse the material selection dialog box we created for the beam frame custom application in the previous chapter. This time it will appear as a standalone add-on rather than part of a full-blown custom application.


Define the material

Material AISI 1005 Steel

New material properties

Name NewUserMat

Density 0

Young's Modulus 0

Poisson's Ratio 0

Alert: You cannot have a negative density

We reuse most of the code. **materialGuiDB.py** defines the dialog box, **materialGuiForm.py** defines the form mode that launches the dialog box, and **materialscript.py** is the associated kernel script.

The contents of **materialGuiDB.py** are the same as **step1DB.py** from the previous chapter.

```
from abaqusGui import *

# Class definition

class Step1DB(AFXDataDialog):

    [
                    ...
                    ...

    ] = range(AFXToolsetGui.ID_LAST, AFXToolsetGui.ID_LAST+4)

    #--------------------------------------------------------------------
    def __init__(self, form):

                    ...
                    ...

    def onNegativeDensity(self, sender, sel, ptr):

                    ...
                    ...
    def onDensity(self, sender, sel, ptr):

                    ...
                    ...

    def onNewMaterialComboSelection(self, sender, sel, ptr):

                    ...
                    ...

    def onExistingMaterialComboSelection(self, sender, sel, ptr):
                    ...
                    ...

    #-----------------------------------------------------------------------
    def show(self):
                    ...
                    ...

    #-----------------------------------------------------------------------
    def hide(self):
                    ...
                    ...
```

The contents of **materialGuiForm.py** are the same as **step1Form.py** from the previous chapter.

```
from abaqusGui import *
import step1DB

# Class definition

class Step1Form(AFXForm):

    #-----------------------------------------------------------------------
    def __init__(self, owner):
                    ...
                    ...
    #-----------------------------------------------------------------------
    def getFirstDialog(self):
                    ...
                    ...
    #-----------------------------------------------------------------------
    def activate(self):
                    ...
                    ...
    #-----------------------------------------------------------------------
    def issueCommands(self):
                    ...
                    ...
```

As for **materialscript.py**, it is similar to the corresponding function from **beamKernel.py** of the previous chapter.

```
# ******************************************************************************
# Material GUI Plug-in
# This script sends commands to the kernel to create the material
# ******************************************************************************

from abaqus import *
from abaqusConstants import *
import material

def createMaterial(selected_material, name, density, youngs, poissons):

    # The user may have selected steel or aluminum in which case the rest of the
    # fields will be left blank and the parameters will have default values
    if selected_material == 1 :
        material_name = 'AISI 1005 Steel'
        density = 7800.0
        youngs = 200E9
        poissons = 0.3
    elif selected_material == 2 :
        material_name = 'Aluminum 2024-T3'
        density = 2770.0
        youngs = 73.1E9
        poissons = 0.33
    else:
        material_name = name

    # if you try to provide a youngs modulus and poissons ratio both equal to zero
    # Abaqus throws an error because in the elasticity table you've given it a row
    # of zeros
    if youngs==0 and poissons==0 :
        print 'Young\s modulus and Poisson\s ratio cant both be zero'
        # Since we exit Step 1 we reset the global counter so the user can perform
        # Step 1 again
        return False


    # Create material by assigning mass density, youngs modulus and poissons ratio
    modelMaterial = mdb.models['Model-1'].Material(name=material_name)

    # if material density is 0 (meaning user did not enter a value and it defaulted
    # to 0), we will not assign a density
    if density !=0 :
        modelMaterial.Density(table=((density, ),       ))

    modelMaterial.Elastic(table=((youngs, poissons), ))
    return True
```

Here is the script that actually creates the plug-in. It is **materialGui_plugin.py**.

```
# ***************************************************************************
# Material GUI Plug-in
# This script registers the material GUI plug-in
# ***************************************************************************

from abaqusGui import getAFXApp
from materialGuiForm import MaterialGuiForm

# Register the plugin
toolset = getAFXApp().getAFXMainWindow().getPluginToolset()
toolset.registerGuiMenuButton(buttonText='Material GUI Plug-in',
                              object=MaterialGuiForm(toolset),
                              kernelInitString='import materialscript')
```

It has many similarities to the kernel plug-in script. First of all it is short and simple. These statements will likely be required for any GUI plug-in you create (of course you will import some module other than **materialGuiForm**). As for registering the plug-in, the function used is only slightly different. It is **registerGuiMenuButton()**, as opposed to **registerKernelMenuButton()** which was used for the kernel plug-in.

The required arguments for **registerGuiMenuButton()** are **buttonText**, which is the name to be displayed in the Plug-ins menu, and **object**, which is the GUI object to which a (**messageId, SEL_COMMAND**) message will be sent. We set this object to our form mode. Hence our plug-in launches a dialog box. There are a number of optional arguments as well; the one we have used here is **kernelInitString**. We need to import **materialscript.py** which is the kernel script, however this import statement cannot be placed in any of the other scripts because they are GUI scripts (and, as you know very well by this point, you cannot mix GUI and kernel scripts). By using **kernelInitString** we can tell Abaqus to import **materialscript.py** into the plug-in. In other words it serves the same function as **getKernelInitializationCommand()** does in custom applications.

22.4 Summary

Registering a plug-in is quite easy; you use the **registerKernelMenuButton()** and **registerGuiMenuButton()** methods depending on whether you are registering a kernel plug-in or a GUI plug-in. The real work goes into creating the kernel or GUI scripts that make up the plug-in. Once you have those, it's easy to package them into a plug-in for future use.