Efficient Finite Element Modelling

Automated model generation & evaluation using Simulia Abaqus©

Martin Pletz

2nd edition

Leoben, 2021

www.martinpletz.com/fe-scripting

martin.pletz@unileoben.ac.at

Chair of Designing Plastics and Composite Materials, Department of Polymer Engineering and Science, Montanuniversitaet Leoben, Austria

Foreword

Have you already spent dozens or hundreds of hours on clicking in Abaqus CAE to build up your model, noticing how unpractical that is? And hoping in discussions and meetings that nobody asks you for changing the model because you know that will take at least two days of clicking?

My name is Martin Pletz, and I've been using Simulia Abaqus© since 2007. Over the years, I found it more and more annoying to build up every model manually in Abaqus CAE. For my Master's thesis, I started using ANSYS and its scripting language APDL. Since then, nearly every FEA model that I build, I build in a fully scripted way. I mostly use Abaqus CAE with the scripting language Python. As far as I know, there are no good resources for starting to script Abaqus models: The Abaqus Documentation is ok, but there are some tricks they do not tell you there and that I had to find out from internet forums, colleagues and by trying stuff in Abaqus.

This course is me trying to provide a brief and clear introduction into scripting Abaqus, defining a workflow for building your model scripts, and explaining the parts of the scripting that are not so intuitive.

This course is also held as a seminar at the Montanuniversitaet Leoben, starting in October 2021. If you have any comments or suggestions, please send me an email (martin.pletz@unileoben.ac.at). This script is a short version of the online script you can find at www.martinpletz.com/fe-scripting.

Table of Contents

1) Introduction

Wouldn't it be nice to have an Abaqus model that does not have to be build again (including hundreds of clicks and dozens of swear-words) when some parameters have to be changed? Including solving and evaluating the model? Here, you will learn how to do that. On the way, we will discuss how to set up models on paper because a good plan and a structured approach is at least as important as being able to automize the model setup and evaluation.

1.1) Prerequesites

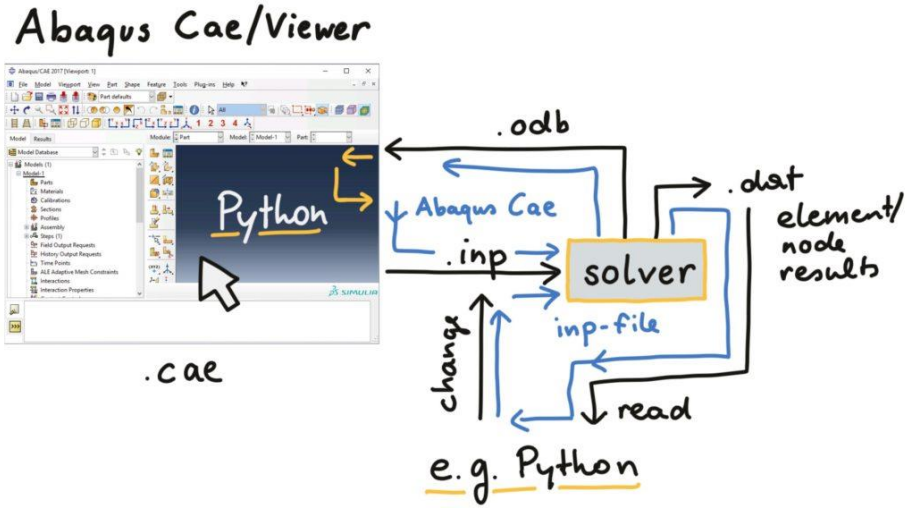
You should already be familiar with the Abaqus CAE interface. If not, you can find some introductory Abaqus CAE videos in the youtube chanel AbaqusPython. You should also know the syntax of the scripting language Python which you can learn with SoloLearn or a huge number of other online resources. Everything we do in this course can be done with the Abaqus Student Edition, which is freely available to students but is limited to 1000 nodes. For playing with Python, I recommend getting Anaconda, which is a Python installation that comes with the most important Python modules and Jupyter Notebooks, which I recommend using when you start with Python.

1.2) Scripting Abaqus

Abaqus consists of the Computer-Aided Engineering (CAE) and Viewer interface (for setting up models and visualizing Output Database (odb) files) and the solver. The model is sent to the solver in plain-text formatted input (inp) files that contain the full model definition with all the nodes, elements, boundaries, loads, etc. In the 90s, you would write those input files by hand. Today, you would rather use Abaqus CAE or other software to build your model and get the inp file. In the input file, you can also request node and element output written to a dat file. That gives us two basic options for automizing the model generation and evaluation:

1. *inp-file*: Manipulating an existing input file and reading results from the dat-file using a programming language. This is fast but limited to simple changes of material or loads but not changes in geometry (because the geometry would have to be re-meshed)
2. *Abaqus CAE*: Using Abaqus-specific Python commands to automize everything that can be done in the CAE and viewer interface. Anything goes, but you have to deal with the way this is implemented in Abaqus CAE, which can be ~~horrible~~ – I mean tricky at times.

In this course, we will only mainly with the second option: Scripting Abaqus CAE using Python.



1.3) Further chapters

In chapter 2, we will look at how scripting the Abaqus CAE interface with Python works and what you need to understand before you can build your scripted models. A big part of that chapter will deal with selecting things like edges in Abaqus because this is the most complicated part of scripting Abaqus CAE models.

Chapter 3 will deal with evaluating Abaqus models in Abaqus CAE or the Abaqus Viewer to automatically generate png images and text files that contain the result data we are interested in.

All tools from chapter 3 and 4 will be applied to the example model in chapter 4, creating a Python script that completely builds, runs and evaluates a model from a set of geometry, material and load parameters.

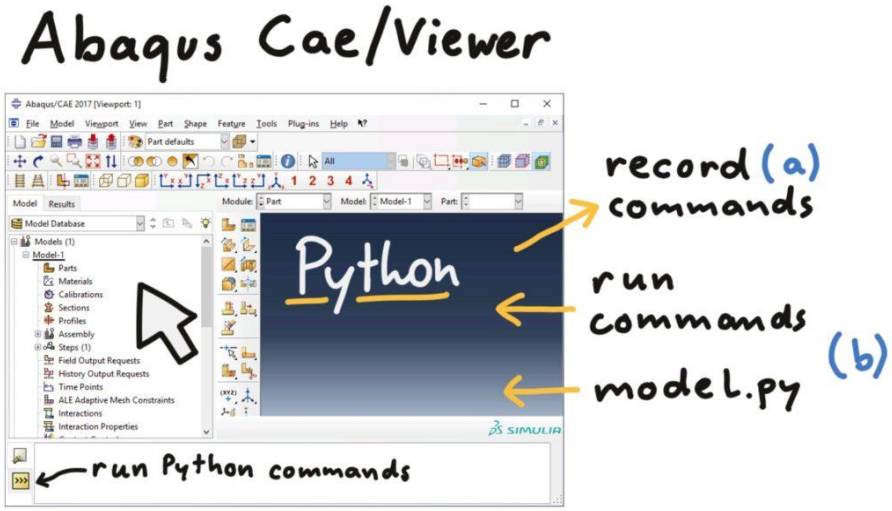
Finally, chapter 5 will give some advice on keeping order with all the automatically created models, which means deleting unwanted result files and creating a folder structure for the job files and result files. A small introduction into plotting diagrams using Python (the `matplotlib` module) is also given.

1.3) Exercises

1. Set up Abaqus: Install the Abaqus student version or get access to a computer with Abaqus installed.
2. Set up Anaconda: Download and install it. We will need that at the end for creating diagrams in Python.
3. If you are not already familiar with the Python syntax, learn that online using SoloLearn or something similar. If you do not know much about Abaqus CAE, you can find lots of introduction videos on youtube.

2) Abaqus CAE

Now let's look at how to script a model in Abaqus CAE, where we have all the options like changing geometry and remeshing at our fingertips. All your clicks in Abaqus CAE are regarded as Python commands in the background. You can also write such commands yourself and run them in Abaqus CAE, building up your model automatically.



We can obtain those Python commands from the reference guide in the Abaqus documentation. Since we already know how to use CAE and do not know how Abaqus calls all its Python commands, it is more convenient to record commands of what we are doing in Abaqus CAE. And then look into their detailed options up in the documentation to use them to build up our scripted Abaqus model.

2.1) Recording and Running Python Commands

How to record Python commands from clicking in Abaqus CAE and how to run Python commands in Abaqus?

2.1.1) Recording Python commands from Abaqus CAE

When clicking in Abaqus CAE, the corresponding Python commands are automatically written to the `abaqus.rpy` file in the active directory. Everything is recorded in this file, even if you scroll or zoom into the model or do evaluations.

At the time you save your Abaqus model, two files are saved: First, the actual CAE (`.cae`) file that you can open with Abaqus CAE, and second, the journal file (`.jnl`) where commands crucial to the model generation are recorded. So, actions like zooming and evaluations in the Viewer are not recorded.

Furthermore, you can start recording a macro in Abaqus with *File/Macro Manager >> create, Directory=work*. This starts writing commands to the `abaqusMacros.py` file until you click *Stop Recording*.

> In the Student Edition of Abaqus, both the *rpy* and the *jnl* recording are disabled, which leaves you with the Macro Manager. If you have a full version of Abaqus, I suggest using the *jnl* file for building the model and the *rpy* file for the evaluation.

2.1.2) Running Python code in Abaqus CAE

How can we now run Python commands in Abaqus CAE? The first option is clicking on *File/Run Script*. There, you have to select a python file to be run in Abaqus.

The second option to run Python commands in Abaqus is the so-called *Kernel Command Line Interface*. This is the frame at the bottom of the CAE window that you can reach by clicking on the `>>>`-symbol. Here, you can run single commands or multiple lines of Python code. It is best for testing new commands and playing with Python. This *command line* also helps you with writing code: If you press the UP key, you get to the last executed command and pressing the TAB key auto-completes your commands.

The third option is starting Abaqus CAE from the Windows or Linux command line and stating a Python script that is then run in CAE. This can be done either by opening the Abaqus CAE window (`script=...`) or just opening CAE in the background (`noGUI=...`):

```
abaqus cae script=model.py
abaqus cae noGUI=model.py
```

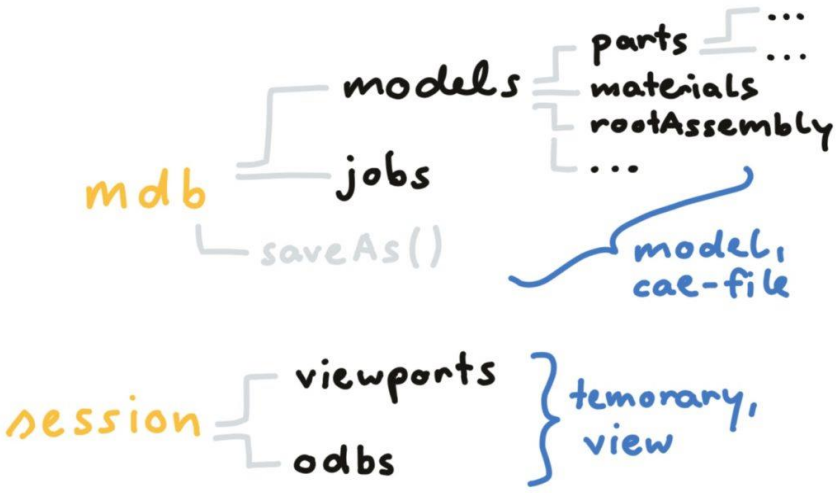
From recorded commands a script for the model can be developed. In many cases, you can just copy your model together from those commands, but for some of them, you can't. The trickiest commands concern selecting entities such as edges, faces, and cells. For such selections, the most convenient commands are not recorded. Also, they make it necessary to understand how the Abaqus Python commands are organized, so we look into that now.

2.2) Python in Abaqus CAE

As already mentioned, Abaqus CAE has a Python-kernel running in the background, with Abaqus objects similar to the modules you already know from Abaqus CAE (models, parts, materials, etc.). To work with those Abaqus objects, you have to import them:

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

There is a hierarchy of Abaqus objects: Everything builds on the two repositories `mdb` (Model Database) and `session` (CAE window and view). You have imported `mdb` and `session` by writing `from abaqus import *`. The model database `mdb` has the sub-object `models`, that itself contains the `parts`, `materials`, etc. which itself contain geometry, nodes, and so on.



Let's look at the object `models`. You can create a new model with the command `mdb.Model()`. Such commands for creating models, parts, etc. are called constructors and are written with a capital letter. In the brackets of a constructor, you can pass along more information, in this case, the name of the new model:

```
mdb.Model(name='test-model')
```

This constructor creates an instance of the class models with the name *test-model*. We can access this instance by writing:

```
mdb.models['test-model']
```

> When writing commands in the Abaqus Command Line, you can use TAB to auto-complete your lines. If you have already written *`mdb.models[`* and you press TAB, Abaqus suggests a model name. If you press TAB again, it suggests the next option. Also if you write *`mdb.Model(`* and then TAB, you get the options of this function.

When creating a model, you can write `m = mdb.Model['test-model']` to use the variable `m` instead of writing `mdb.models['test-model']` each time you need it. The command mdb.models also works as a Python dictionary, so you can use the `.keys()` and `.values()` functions to see what models exist:

```
>>> mdb.models.keys()
['Model-1', 'test-model']
```

Then you can create a part in this model and then a reference point in this part and so on. To list all the methods of your model `m`, `mdb`, `session`, or an instance, you can use the `dir`-command:

```
>>> dir(mdb)
```

I do not list the output here, because it is too long. Look for yourself in Abaqus! These methods can change the object or create a new instance. There are not only constructors and sub-classes, but also functions that do something. For `mdb`, there is the function `saveAs()` that saves your model under the path that you state in the brackets:

```
>>> mdb.saveAs('test.cae')
```

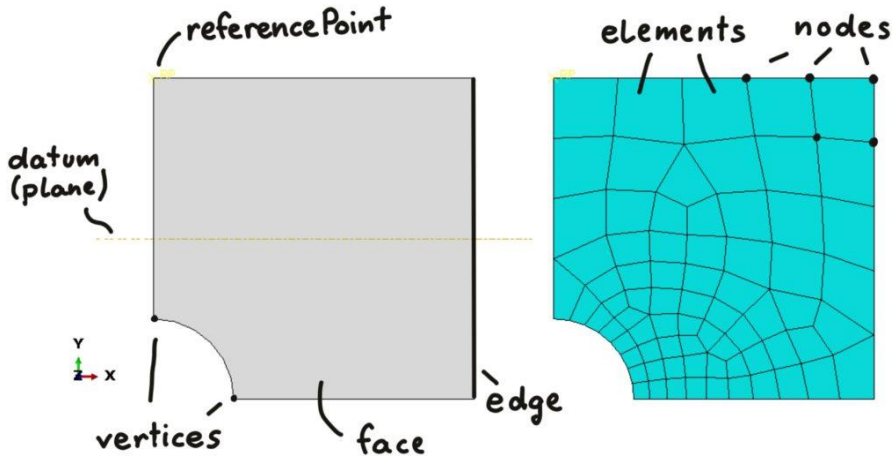
Now you know that there is a hierarchy of objects in Abaqus Python code, and those objects are parts, surfaces and so on. This will help us now, when we look into creating sets and surfaces that contain such objects.

2.3) Sets and Surfaces

For a nice and clean Abaqus CAE model, it is important to create sets and surfaces first using meaningful names and then apply constraints, boundary conditions, and loads on those.

Sets can contain anything (vertices, edges, faces, cells, datums, referencePoints, nodes or elements)

Surfaces can contain edges in 2D models and faces in 3D models. They are used to define contact and surface-based loads or constraints.

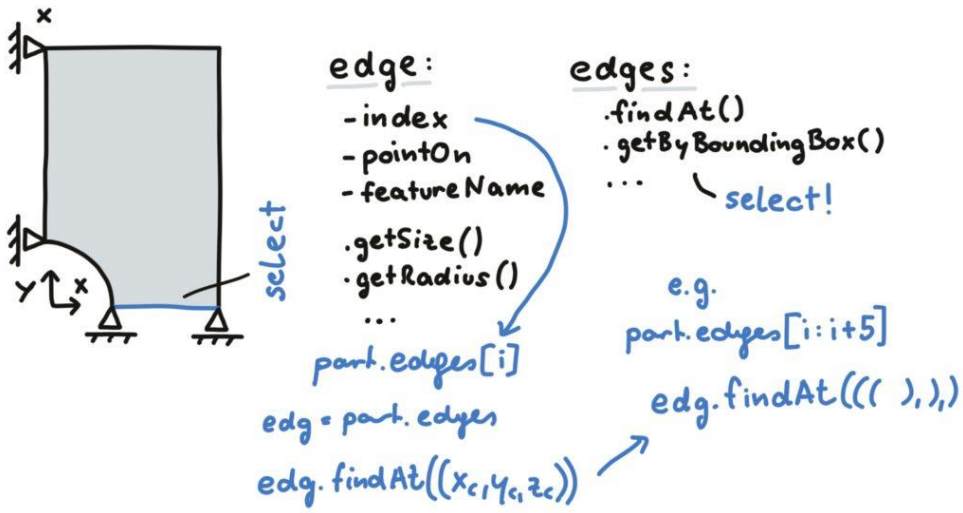


So let's look at how we can create sets and surfaces in a scripted way. First, we look into edges to understand the basic principles of automated selection. Then, we see how that can be extended to faces, cells and other entities.

> There are boolean operations (union, difference, intersection) for existing sets that are handy: With p defined as a part we can write (replace the keyword UNION by DIFFERENCE or INTERSECTION for the other operations): `p.SetByBoolean(name='Set3', sets=(p.sets['Set1'],p.sets['Set2'],), operation=UNION)`

2.3.1) Selecting Edges

We first look at how to handle edges in Abaqus CAE using Python commands and then look at differences to selecting other geometry, model, or mesh entities like vertices, faces, cells, datums, referencePoints, nodes, and elements. In the figure below, we see that there are two data types for edges: A single edge (Edge) or edges (Sequence of Edges). Every edge has an index and every edge is also uniquely identified by a point on it (`pointOn`).



Recording commands for edge selection

If you have a part in Abaqus CAE and you record the commands for creating a set that contains one edge of the part, you get a line like that:

```
p.Set(edges=p.edges.getSequenceFromMask(('[#400]',),),
      name='Set-1')
```

So we create a set from edges that we provide during the set creation. More details on what the set needs are stated in the Abaqus Documentation. The edges are provided in a weird way with a mask command and a certain number which is 400 in this case:

```
p.edges.getSequenceFromMask(('[#400]',),)
```

That does not help us much for our automated model: We do not know what the number 400 means and by using this number we might select something different of the model once we scripted the model.

Selecting edges using `findAt`

There is also no easy way for knowing what number to use for what edge. This is why we already had a certain command in the *model.py* template, where we adapted the `journalOptions` of Abaqus CAE:

```
session.journalOptions.setValues(replayGeometry=COORDINATE,
                                 recoverGeometry=COORDINATE)
```

After running that command, recording set creation in Abaqus CAE will give something like this for the edges:

```
p.edges.findAt(((20.0, 10.0, 5.0), ))
```

Now that looks better for scripting! We obviously just need to write a coordinate that lies on the edge in three brackets. Just make sure that the coordinate does not have multiple edges on it because then you will just get one of them, and you do not know which one upfront. Multiple edges work similarly:

```
p.edges.findAt(((20.0,3.4375,20.0),), ((20.0,-16.25,5.0),),
               ((20.0,-9.6875,0.0),), ((20.0,10.0,5.0), ))
```

Do we need three brackets or would two also work? Well, for that question, we need to look at the data types in Abaqus. When we write a `findAt` command like above, we get a sequence of edges (that we can also get by using a range of indices in square brackets) and not just a single edge. Ans in a set or surface we do need a Sequence of edges:

```
# multiple edges
>>> type(part.edges[1:5)
<type 'Sequence'>
# one single edge
>>> type(part.edges[0])
<type 'Edge'>
```

Writing *`p.edges`* gives you all the edges in the part in a sequence.

If you have a list of coordinates that you want to use in findAt, you need to pass the list as coordinates:

```
list_edge_pos = [[20.0, 3.4375, 20.0], [20.0, -16.25, 5.0],
                 [20.0, -9.6875, 0.0], [20.0, 10.0, 5.0]]

p.edges.findAt(coordinates=list_edge_pos)
```

A list like that could have been created automatically from positions of edges that you know from your model parameters, so this command could be something to use in a scripted model.

Selecting edges using `getByBoundingBox`

Sometimes we want to create sets from multiple edges that lie at specific coordinate ranges in the model. If we have a symmetry plane it thus would be nice to just select all edges that lie in this plane, no matter how many of them there are. This can be done with the function `getByBoundingBox()`. You can use `xMin`, `xMax`, `yMin`, etc. to specify a region. All edges that completely lie in this region are selected. Due to the tolerances in Abaqus, using `xMax=0` might not give you all edges that lie at x=0, so you should define a very small tolerance value and use that:

```
TOL = 1e-5

# all edges that lie at x=10
p.edges.getByBoundingBox(xMin=10.-TOL, xMax=10.+TOL)

# all edges that lie at x=0 (no edges at x < 0)
p.edges.getByBundingBox(xMax=TOL)
```

Now that is much more elegant than `findAt`, isn't it? I use it for all simple selections and in most cases, the combination of `getByBoundingBox` and boolean set operations is sufficient. There are similar functions that use a cylinder or a sphere instead of the box for selection. You can check them out in the Reference Guide of the Abaqus Documentation.

Selecting edges by edge properties

For some edges, it would be rather complicated to calculate a coordinate that lies on the edge. With defining a bounding box, we often would also select other edges that we do not want. So it would be nice to select an edge by other properties. In the above figure, we saw that each

edge has a `featureName`(a string that defines how it was created) and functions like `getSize` and `getRadius`. Maybe you have some edges that you know exactly the size of but not where they are: So let's look at how to select those! You can check out all the properties of single edges by looking at one edge in your part:

```
>>> dir(p.edges[0])
```

There is no direct way of creating edge sequences we need for sets by edge properties. We can, however, get the edges we want by those features, then create a list of edge positions (with the function `pointOn` for each edge), and use those positions in a `findAt` command. If you use a for loop, that could look like that:

```
edge_coord = []

for edge in p.edges:
    if edge.getSize() > 5.-TOL and edge.getSize() < 20.+TOL:
        edge_coord.append(edge.pointOn)

p.Set(name='e1', edges=p.edges.findAt(coordinates=edge_coord))
```

Or if you use list comprehension like me, you can write the selection in one line. See which one you prefer and then stick to it:

```
edge_coord = [edge.pointOn for edge in p.edges if
              edge.getSize()>5.-TOL and edge.getSize()>20.+TOL]

p.Set(name='e1', edges=p.edges.findAt(coordinates=edge_coord))
```

By all means, do not call your edge sets *e1*, this is just to not make the code to long in these blocks. Use meaningful names, it will save you a lot of worries!

2.3.2) Selecting other model entities

Everything that we have seen for selecting edges also works for vertices, faces, cells, nodes, and elements. Just note that the properties would be different: A vertice, for example, wouldn't have a size! So use `findAt` and `getByBoundingBox`, and if you want to select something by its properties, take one of those and use `dir()` to see the properties.

For datums (planes, axes, coordinate systems, etc.) and referencePoints, it actually is different. When you pass a referencePoint to a set, Abaqus wants you to use the following syntax:

```
# create referencePoint in part at coordinate (0,0,0)
p.ReferencePoint(point=(0,0,0))

# create set out of referencePoint
p.Set(name='RP', referencePoints=(p.referencePoints[3],))
```

with 3 being the `id` of the reference Point that might also change when we change our model in the script. There might be some reasons why this has to be so complicated that I not get, but this is that rather inconvenient way you have to use referencePoints in sets:

```
# create referencePoint in part at coordinate (0,0,0)
rp = p.ReferencePoint(point=(0,0,0))

# create set out of referencePoint
p.Set(name='RP', referencePoints=(p.referencePoints[rp.id],))
```

A typical application of a datum is using a datum plane to cut a geometry. This would look like that:

```
# create a datum plane at x=5
plane = p.DatumPlaneByPrincipalPlane(offset=5, principalPlane=YZPLANE)

# partition all faces by the datum plane
p.PartitionFaceByDatumPlane(faces=p.faces[:],
                            datumPlane=p.datums[plane.id)
```

2.4) Exercises

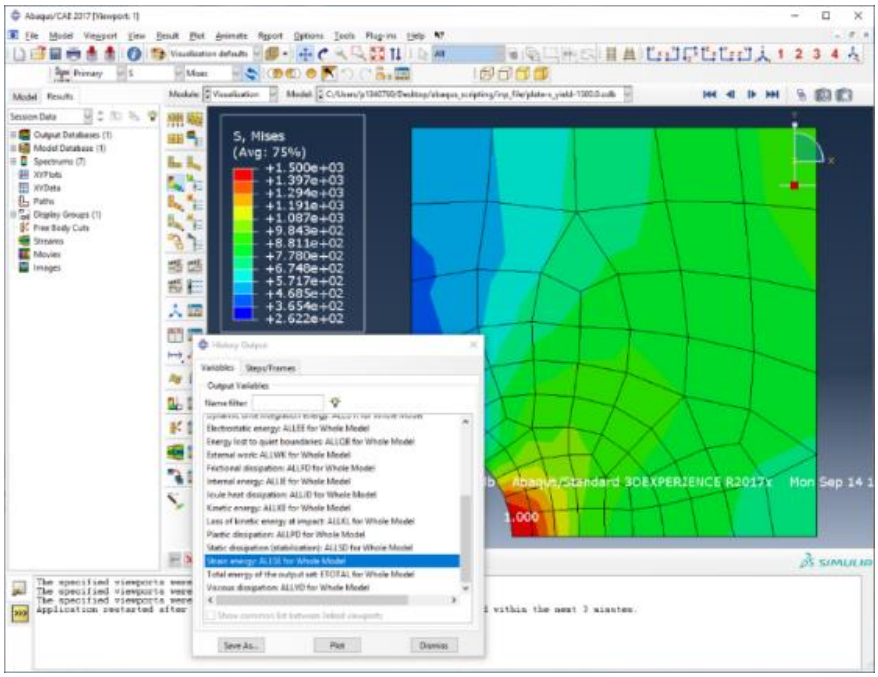
1. Create a small model that contains a 2D part by hand in Abaqus CAE and look at all the functions the model and the part have.
2. Try all the ways for selecting edges that were described above. See for yourself what happens when you select an edge with `findAt` by giving a coordinate that belongs to two edges!
3. Create a reference point in your part and make a set out of it. Look at the recorded Python commands and see what id your referencePoint was.

3) Abaqus Viewer

As we looked into scripting the model in the last section, we now deal with the evaluation of the model in a scripted way. This can be done in the Abaqus Viewer, which is also included in Abaqus CAE. Everything starts now with the `session` repository. We can get the recorded commands from either the .rpy file or the *abaqusMacors.py* file.

There are two kinds of output in Abaqus:

1. Field Output (result fields like displacement or stress)
2. History Output (lists of result values like total strain energy for all time frames)



3.1) Stuff we want to evaluate

We would like to use the field output to export the data, view the result fields to write images, and look for maximum or mean result values in the model. We need the history output to export the data and create time-output curves.

Furthermore, in the Abaqus Viewer, you can create a path and evaluate field output along this path. The path points can be nodes selected by the user or coordinates that might or might not lie on the mesh. If you use a coordinate-based path, Abaqus has to interpolate the field output and returns nothing for points that lie outside the mesh.

There is some default output that is generated in each CAE model. If you need something that is not in the default output, you can define that in your model by creating an additional field or history output in the step module of Abaqus CAE (with the model defined as `m`)

```
# additional field output
m.FieldOutputRequest(name='F-Output-2',
       createStepName='Step-1',variables=('IVOL',))

# additional history output: U1 for nodes in set
```

```
m.HistoryOutputRequest(name='H-Output-2',
        createStepName='Step-1', variables=('U1', ),
        region=inst.sets['Set-1'])
```

To plot field output outside Abaqus, you need the coordinates of the nodes (node-based results like displacement) or the integration points (integration-point based results like stress). The easiest way to get those coordinates is defining coordinates as an additional field output (COORD). For the nodes, this can be done in Abaqus CAE and thus directly with Python; for the integration points, this has to written to the inp file, which can be done in Abaqus CAE using the following commands (we also output the integration point volumes and node coordinates):

```
model.keywordBlock.synchVersions()
block_list = model.keywordBlock.sieBlocks
pos_i = [i for i, j in enumerate(block_list)
         if 'Output, field' in j]
# coordinates of IP (COORD) & volume of IP (IVOL)
str_out = "**\n*Element Output, directions=YES\n\
           COORD,IVOL,\n**\n*Node Output\nCOORD,"
# insert the str_out string
model.keywordBlock.insert(pos_i[0], str_out)
```

> Writing lines to the input file in Abaqus CAE is something you should do last before submitting the job because changing something in your model later usually messes up your model.

The first decision to make for the evaluation is whether to use Abaqus CAE or the Abaqus Viewer, which can be opened separately. I suggest to take Abaqus CAE as well, and if you build, run and evaluate your model in the same script, you need to do that anyway. With our usual import statements from the model, we have everything imported that we need for evaluation in Abaqus CAE.

Our first command is for loading the odb that we can view or get data from:

```
# open the odb with the odb_name
odb = session.openOdb(name=odb_name+'.odb')
```

Note that having many odb files open at the same time is not a good idea, so after you finished your evaluation, you should close the odb:

```
# close the odb
odb.close()
```

Usually, you would open an odb file and look at filed outputs and maybe create some rather ugly-looking diagrams from history output. We now look into how to access field and history output with Python commands. Only some of it can be done by recording commands from Abaqus, so we need to take a closer look:

3.2) Obtaining field ouptut data

Let‘s access field output data: In a frame, there are `fieldOutputs` with their values that you can access by its index like `frame.fieldOutputs.values`. Those values have a node or element number, other information, and most importantly, our result data as a list. Here we take the last step and the last frame of it (an index of -1 returns the last element of a list) and get the stress for this last frame:

```
# last frame
frame_end = odb.steps.values()[0].frames[-1]
field_out = frame_end.fieldOutputs['S']

# getting field output directly (slow, but robust)
s_out = np.array([i.data for i in field_out.values])
```

There is also the possibility of using so-called `BulkDataBlocks` to access the result data. This method is very fast but has some bugs. Different kinds of elements are output in different data blocks. So for a simple evaluation, especially if you do not care about how long reading the data needs, I do not suggest to use BulkDataBlocks The command `np.copy()` is needed there because of a bug:

```
# getting field output with BulkDataBlocks (fast, but with some bugs)
s_bulk = field_out.bulkDataBlocks
s_out = np.concatenate([np.copy(i.data) for i in s_bulk])
```

The evaluation of field output data seems rather complicated: We do not directly know what node or element is behind `values`, but we know that all kinds of output will be in there (node- and integration point data, different element types, etc.). Usually, it is sufficient to have the coordinates of the output points and the output data, so we just need to make sure we get the same kinds of coordinates (nodes or integration points) as we have in our other field output. This can be done with the `getSubset` function applied to a `fieldOutput`:

```
field_out = frame_end.fieldOutputs['COORD']

# position can be NODAL or INTEGRATION_POINT
field_out = field_out.getSubset(position=NODAL)
```

We can get the nodal and integration point coordinates separately. Displacements are nodal results; stress and strain are integration point results — We can write one of those NumPy arrays to a text file like that:

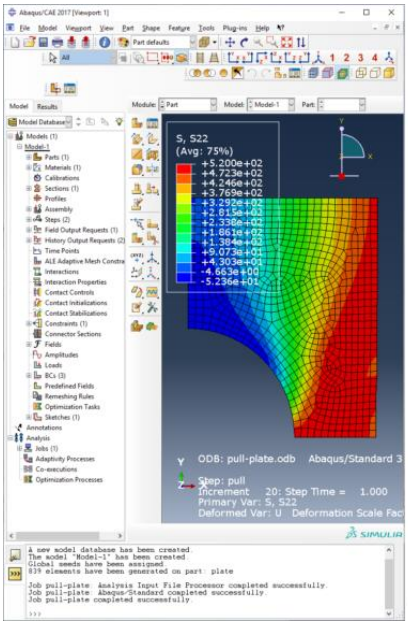
```
# write to .dat file
np.savetxt(odb_name+'_s.dat', s_out)
```

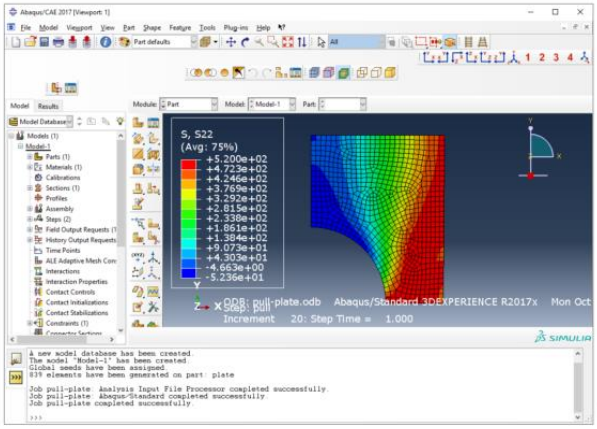
> Using the name of the odb file in the result files and keeping a similar syntax (‘_s_out’ for stress, ‘u_out’ for displacement, etc.) is a good idea! If you do not want to have too many separate files for your output, you can think about merging the arrays that you get for nodal coordinates and displacements using NumPy array operations.

3.3) Printing field output images

One of the first things you probably do after an Abaqus job is finished is opening the odb file and looking at the von Mises stress field. There are many options for displaying the filed output: Different kinds of field output, zoom or rotate the view, cut your model, change the legend and so on. Once you have your view ready, you can print an image from Abaqus (using *File/Print*), where png images work best (they even have a transparent background). Let’s look at how to automate that, assuming you already know at what angle you want to view what result.

When we want to print nice-looking images, we face two problems in Abaqus: First, depending on the Abaqus window, the aspect ratio of the image will change and second, the legend is usually too small and there is lots of text in the window that we do not want to have in our image:


(a)


(b)

3.3.1) Saving the view

So, what can we do concerning the changing aspect ratios of the view window? We give the view window a fixed width and height that we will work with. Run those commands and run them in Abaqus:

```
vp = session.viewports['Viewport: 1']

# Change size of viewport (e.g. 300x200 pixel)
vp.restore()
# position of the viwport
vp.setValues(origin=(50,-100))
vp.setValues(width=300, height=200)
```

Now you have a window with a fixed size and thus aspect ratio. Start recording a macro or use the rpy file in the following: Load your odb and display the results in the way you want. Zoom, drag and rotate your model in the viewport. Once you are happy with what you see, go to *View/Save* and save your view e.g. as *'User-1'* with the option *Save current*.

Now look into your macro or rpy file and copy the commands for loading your odb, showing the results, and saving your view. Since all information about your view is now in the 'User-1' view, you can delete all zooming, rotating and dragging commands. Also, add a line for loading the saved view by `vp.view.setValues()`. After cleaning the code up and adding this line, it should look something like that:

```
# load the odb
odb = session.openOdb(odb_name+'.odb')

# display the results
vp.setValues(displayedObject=odb)
# show field output (INVARIANT/COMPONENT)
vp.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF,))
vp.odbDisplay.setPrimaryVariable(variableLabel='S',
     outputPosition=INTEGRATION_POINT,
     refinement=(INVARIANT, 'Max. Principal'),)

# save view 'User-1'
session.View(name='User-1', nearPlane=150,farPlane=240,width=63,
    height=40,projection=PERSPECTIVE,cameraPosition=(15,0,20),
```

```
    cameraUpVector=(0,1,0),cameraTarget=(15,0,0),
    viewOffsetX=-3.5, viewOffsetY=0.02, autoFit=OFF)

# load the defined view
vp.view.setValues(session.views['User-1'])
```

If the size of your model geometry can change a lot, using a fixed view is not such a good idea: Then, you can use the default view, save your view with the option *Auto-fit* (Abaqus automatically zooms so that the hole geometry is in the window) or just use one of the main views like that (access them with *View/Toolbars/Views*):

```
vp.view.setValues(session.views['Front'])
```

3.3.2) Formatting the legend and hiding text boxes

In Abaqus, you can go to *Viewport/Viewport Annotation Options* to change the legend formatting and also select what kinds of text boxes and coordinate systems should be viewed in the viewport. Just do that in Abaqus and record it. My favourite formatting looks like that (hide everything except the legend, white legend background and bigger font size):

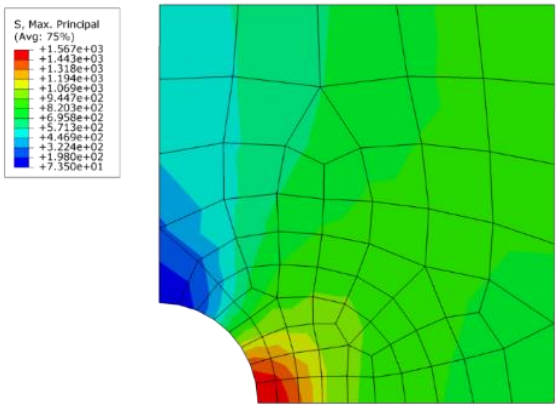
```
# change the legend and what is displayed
vp.viewportAnnotationOptions.setValues(legendFont=
       '-*-verdana-medium-r-normal-*-*-140-*-*-p-*-*-*')
vp.viewportAnnotationOptions.setValues(triad=OFF, state=OFF,
    legendBackgroundStyle=MATCH,annotations=OFF,compass=OFF,
    title=OFF)
```

You might need to adapt the view of your model because the legend became bigger. Just do it and copy the new command for saving the view to your script. Finally, we create a png image of the viewport. You can go to *File/Print*, set everything as you like, record the commands, and copy them to your script. After cleaning them up they might look something like that:

```
# output image
session.printOptions.setValues(reduceColors=False, vpDecorations=OFF)
# set a bigger image size: 3000 x 2000 pixels
session.pngOptions.setValues(imageSize=(3000, 2000))
session.printToFile(fileName=odb_name+'_max_princ',
    format=PNG, canvasObjects=(vp,))
```

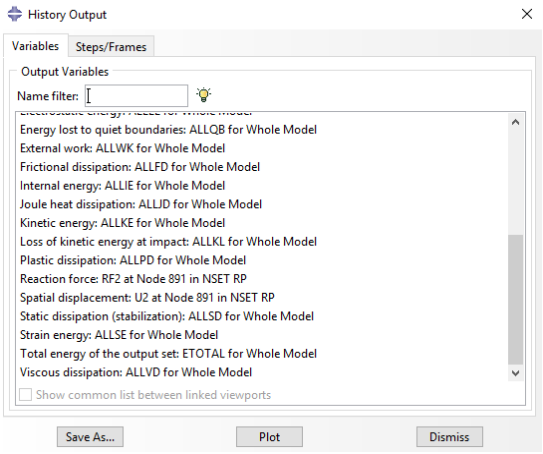
This way, you automatically export an image like that. And can do so also for hundreds of different load cases in the model. Even with this model, you could think about how to export an image for each frame to see how the stresses become more and more uniform with plastic deformation.

> A common error for saving images is that you set the size of the viewport bigger than the screen resolution. So rather take a smaller size of the window but then use imageSize in the pngOptions to increase the image size.



3.4) Getting history output data

History output is part of the step object. Between the step and the history output, there is something called `historyRegion`, which defines what the history output is for (the whole assembly, a contact definition, single nodes, etc.). Let's look at how to access history output and deal with those regions in more detail.



Once you know the history region where your desired histroyOutput is in, you can get the time/result list as I do it here for the whole strain energy of the model *ALLSE*, which is in the history region with the name *'Assembly ASSEMBLY'*:

```
step1 = odb.steps['load']

# get time/ALLAE list
ae_data = np.array(step1.historyRegions['Assembly ASSEMBLY'].
                   historyOutputs['ALLSE'].data)
```

You can then save your NumPy array containing the history output as we saw it for the filed output.

> To evaluate multiple steps, you have to obtain the history output for each step separately and then merge them.

But how do we get history output that is not just the default energy of the hole model but for example the displacement of a single node? Writing your step and then `.historyRegions[`, and pressing TAB, you can browse through your historyRegions. In one of my models, I find a node that I used for history output:

```
>>> step.historyRegions['Node PLATE-1.7814']
```

Maybe you ask yourself: “How do I know what number my history output node has?”. You don’t, at least not directly! However, you can access the node sets of your instances and find out the node label `n_hr` in this way:

```
>>> inst = odb.rootAssembly.instances['PLATE-1']
>>> n_hr = inst.nodeSets['RP'].nodes[0].label
```

You can then create a string `str_hr_node` for the historyRegion:

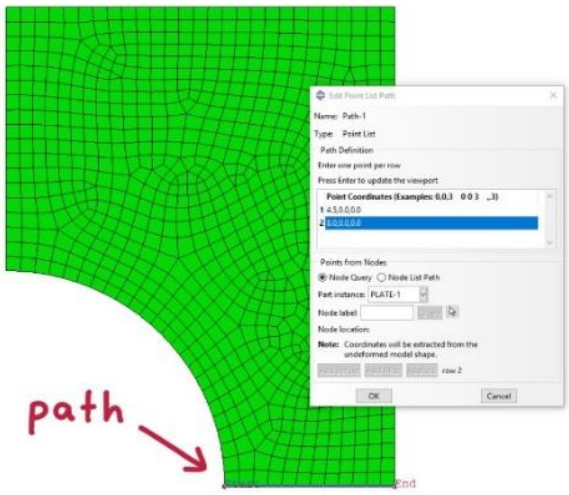
```
# put together a string to select history region
str_hr_node = 'Node '+inst.name+'.'+str(n_hr)

# get the history region for the node
hr = step.historyRegions
```

If you know that there is only one node in the history regions, you can also select this history region by its name like that (using the so-called list comprehension of Python):

```
hr_node = [hr for hr in step.historyRegions
           if 'Node' in hr.name][0]
```

3.5) Evaluating results along a path

Sometimes, we want to evaluate field output along a path. Abaqus lets us create paths in the Viewer with *Tools/Path/…* and can evaluate results along those paths. The basic option is using a list of nodes to define the path.



However, it is also possible to define coordinate-based paths. This, for example, creates a path with three points from the point coordinates:

```
# create path from coordinates
pth = session.Path(name='Path-1', type=POINT_LIST,
          expression=((0,0,0), (20,0,0), (50,0,0)))
```

For the path evaluation, you first have to display the desired field output variable before extracting path results. The first column of the path output can be set by the statement `labelType`: It can be the distance along the path (`TRUE_DISTANCE`), the distance projected onto an axis (`X_DISTANCE` for x-axis) or a coordinate (e.g. `X_COORDINATE`):

```
# field output to be evaluated along path
vp.odbDisplay.setPrimaryVariable(variableLabel='S',
    outputPosition=INTEGRATION_POINT,refinement=(COMPONENT, 'S22'))

# output for 20 eqiuidistant points over path
# shape: path over DEFORMED or UNDEFORMED geometry

xy = session.XYDataFromPath(name='XYData-1', path=pth,
    includeIntersections=False, shape=DEFORMED,
    projectOntoMesh=False, pathStyle=UNIFORM_SPACING,
    numIntervals=20, projectionTolerance=0,
    labelType=TRUE_DISTANCE)
```

Then you can save your path results `xy` in a a text file.

> Abaqus needs to interpolate result fields if you evaluate paths: You get a value for any coordinate or nothing if the coordinate lies outside your elements. However, this interpolation is rather slow.

3.6) Exercises

1. Run the model script from appendix A and copy the odb file to a separate folder.
2. Evaluate the field Output of the displacement `U` and the coordinates `COORD` for all model nodes and write them to separate dat files. You will need to use the above `getSubset` for the coordinates.
3. Write a script that automatically creates a nice-looking png image of the vertical stress.
4. Evaluate the history output of the vertical reaction force RF2 and vertical displacement U2 of the reference point and write those two columns into a dat file.
5. Create a path that goes from the notch tip to the right and evaluate the vertical stress S22 over the x-coordinate: a) for undeformed geometry and b) for deformed geometry.
6. Load and plot all the dat files in the program of your choice (Origin, Excel, MatLab, Python, etc.). How do the results correspond to the expectations and analytical calculations in chapter 2?

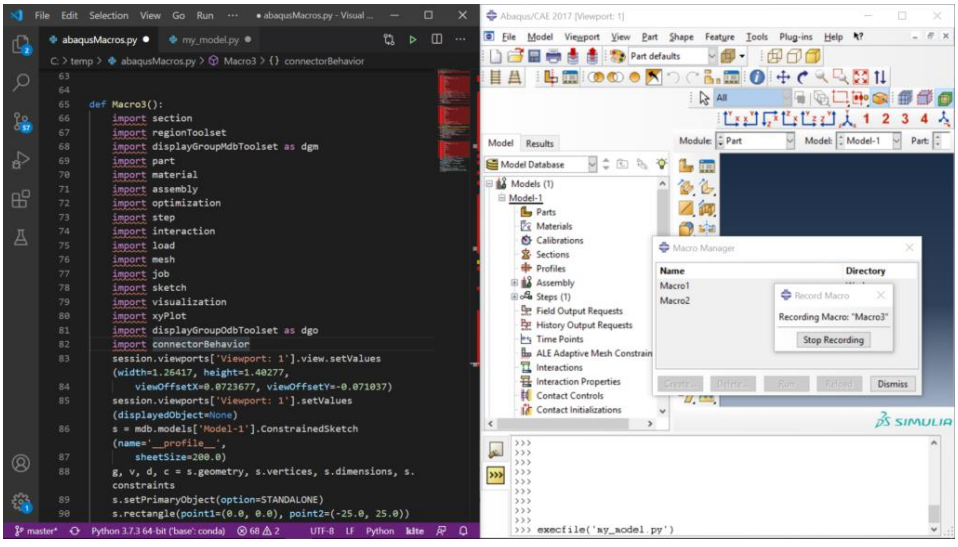
4) Example Model

4.1) Developing a scripted model

Now you know how to record and run Python commands in Abaqus and what commands you can directly adapt from the recordings. You also have learned how to create sets and surfaces, and do evaluations that could not be directly taken from the recorded commands. So you are prepared to use these skills to build a complete Abaqus model that is scripted in Python.

You could just sit down and start developing your code. However, there is a workflow that evolved for me over the years that I will state in the following. Some of it is based on my personal preferences so once you get a hang of it, of course, you can and should adopt it. But for starting to script Abaqus, I advise you to use the following workflow, particularly so you do not forget about some necessary steps that cause you troubles later. Also, be aware that you will spend dozens of hours on developing models like that, most of it looking for errors and correcting them: So if you work as structured and clean as possible, this saves you a lot of time (and worries).

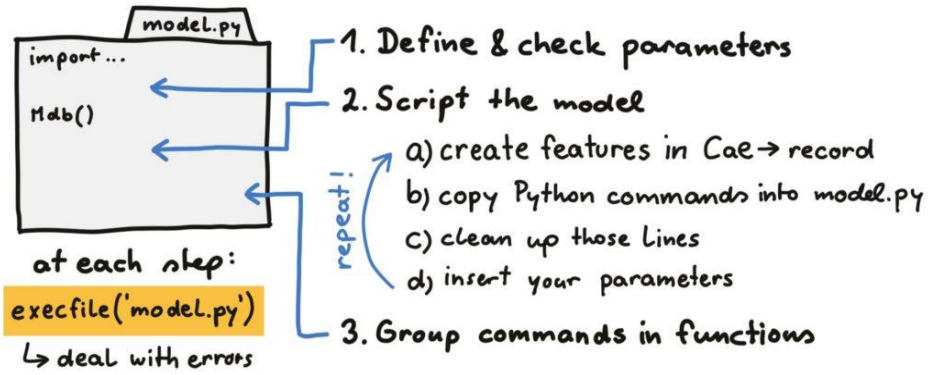
As you record commands and build the scripted model (which we will call *model.py*), have an Abaqus CAE window open and a text editor (I prefer editors that support syntax highlighting like VisualStudio Code or PyCharm) open side by side. If you use the student version, you can only obtain the Python commands using the Macro manager. Otherwise, you can also look at the jnl file for Abaqus CAE commands and the rpy file for viewer commands. In this setup, we will create the model feature by feature in Abaqus CAE, and stop after each step to copy the new Python code to out *model.py*, adapt them, test them by running `execfile('model.py')` in the Abaqus command window (>>>), and continue.



Development environments like Visual Studio Code or PyCharm help you a lot while developing your Python code: They can do code completion, highlight possible errors and so on. They do, however, not know all the Abaqus-Python commands. Tools like kite, however, can learn the Abaqus commands, so once you typed an Abaqus-Python command, it will suggest it in the future when you use code completion.

In the following figure, the important steps of the workflow for developing your *model.py* file are listed. You start from a template that already has al the necessary import statements. As the first step, you define all your model parameters and run some sanity checks on them, so you won't spend hours looking for an error in your scripted model until you find out that your parameters didn't make sense. The second step is actually building up your model, which you should do step by step: a) add one or two features in Abaqus CAE, b) copy the recorded lines into your *model.py* file, c) clean up those lines, and d) insert your model parameters. Then you run your new *model.py* file in Abaqus CAE either by writing `execfile('model.py')` in the Abaqus command window (>>>) or by clicking *File/Run Script*. If everything runs ok, you can continue, otherwise, you have to look for errors in your code, try correcting them, rerun the model and so on.

Once you have completed scripting the model comes the third step: Group all commands in functions to make your code easier to read and more structured. I even suggest having one function in the end that builds, runs, and evaluates your hole model.



Now we look into these steps in more detail, starting from the model header. It contains all necessary import statements (so you can ignore the import statements in the recorded files). We need the `TOL` variable and the `journalOptions` command for selecting stuff as stated in chapter 4. Having our working directory defined as `DIR0` will help us in the next chapter. Do not forget to describe your model in the first line comment. I also like to add a date, and when I might share it with somebody, my initials *MP*.

```
# insert meaningful definition of the model, MP, 09-2020
from abaqus import *
from abaqusConstants import *
from caeModules import *
import os
import numpy as np
session.journalOptions.setValues(replayGeometry=COORDINATE,
                                 recoverGeometry=COORDINATE)
TOL = 1e-6
DIR0 = os.path.abspath('')
```

4.1.1) Define and check parameters

When you got this far, you should already have a sketch of your model with all its parameters defined. At this point, you write them into your *model.py* file. o not forget to state the dimension system you are in (e.g. N-mm-s) and add comments for all the variables. This is so you can be sure that when you continue developing your code, you do not have to deal with errors you made here, because you mixed up variables or stated some of them in cm and others in mm.

```
# parameters for geometry, material, load and mesh (N-mm-s)
# ----------------------------------------------------------
b = 10.               # width of the plate (mm)
h = 20.               # height of the plate (mm)
# ...
```

Because we really don't want to look for problems later that had to do with parameters that did not make sense, I suggest adding some sanity checks to your model script after the parameter definitions. Particularly for geometric parameters, I suggest doing so! You can choose for yourself how detailed you want to do that: Everybody knows, for example, that a Young's modulus needs to be positive: So that would be an error you could directly see when you look at your parameters. This is the syntax for checking a variable and raising an error:

```
if b < 0:
    raise ValueError('width b must be positive')
```

4.1.2) Script the model

Now everything is ready to build the model. Take these lines to reset the model and defining the main model as `model`.

```
# create the model
# -------------------------------------------------------
Mdb()                 # reset the model
model = mdb.models['Model-1']
```

Run the existing Python model in Abaqus CAE with `execfile('model.py')` in the Python command line. Add one or two features in the Abaqus CAE window. Copy the relevant lines from the rpy, jnl or macro file to your *model.py* file. Check if this runs. For drawing a rectangle in a sketch, this might look like that:

```
mdb.models['Model-1'].ConstrainedSketch(name='__profile__',
    sheetSize=200.0)
mdb.models['Model-1'].sketches['__profile__'].rectangle(
    point1=(0.0, 0.0), point2=(35.0, 25.0))
```

A big part of the recorded commands are the import statements. After importing everything from `abaqus`, `abaqusConstants` and `caeModules`, as done in the above header, you do not need those individual import statements.

Introduce shorter variables for the model, sketch, etc. and replace numbers by your variables. Add comments to the code:

```
# create a sketch
s = model.ConstrainedSketch(name='plate', sheetSize=200.0)
# draw a rectangle
s.rectangle(point1=(0,0), point2=(b,h))
```

Run the whole script that now contains some new lines. Does it work? Does it do as it should? You will spend most of your time at this step, looking for errors. Abaqus usually tells you where the error occurred. In this case, for example, the sketch was not defined before drawing a rectangle in it:

```
>>> execfile('model.py')
A new model database has been created.
The model "Model-1" has been created.
File "model_plate.py", line 38, in <module>
```

```
    s.rectangle(point1=(0.0, 0.0), point2=(35.0, 25.0))
NameError: name 's' is not defined
```

If you do not instantly get what is wrong with your code, go back to the last running version and then continue in very small steps (adding and running the script) to find out where exactly the problem was.

Once your newest commands work, continue building your model by recording the next steps of your model creation, job creation and model evaluation.

To create, run, and evaluate a model in one script, Python should wait for the job to complete before doing the evaluation of the odb. This can be done by `job.waitForCompletion()` if the job has been defined as `job`.

4.1.3) Group commands in functions

Once you made sure that parts of the code really work, you can start collecting them in functions. Think of all the variables that you have to pass to the function and what the function should return. In most cases, you will pass the model and some other parameters to the functions that will change the model. You do not need to return the changed model.

```
def draw_sketch(model,b,h):
    # create a sketch
    s = model.ConstrainedSketch(name='plate', sheetSize=200.0)
    # draw a rectangle
    s.rectangle(point1=(0,0), point2=(b,h))
    return
```

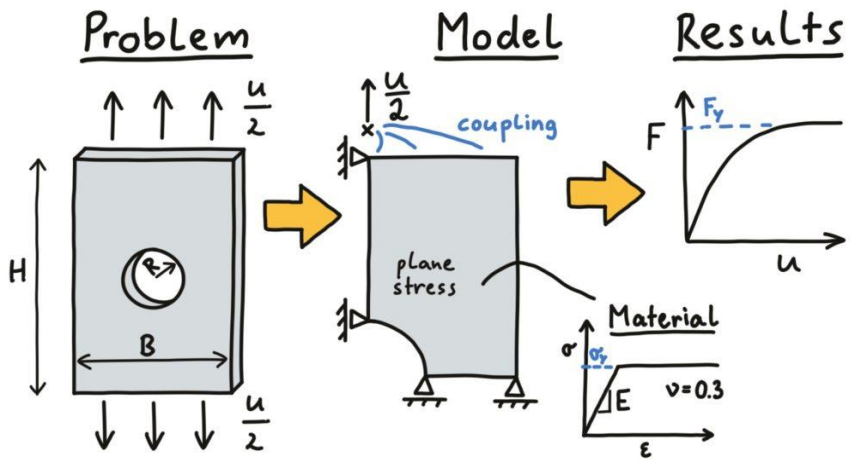
Introduce one function that builds, runs and evaluates your hole model by calling all the other functions. If you want, you can keep some order by grouping parameters that belong together in the function arguments (geometry, material, load, etc.):

```
# function for the whole model
def make_model((b,h),(E,nu),u_y,mesh_size):
    # reset model
    Mdb()
    model = mdb.models['Model-1']
    # call other functions to build the model
    # ...
    return
```

This might seem unnecessary to you, and introducing lots of arguments of functions where we input variables that we defined in a fixed way above seems useless. But keep in mind what we want to do ultimately: Varying some of our parameters in an automated way, and this is much easier once we have to call one function of the model and not make a loop over more than hundred lines of code.

4.2) Scripting the example model

The figure below shows the model we introduced in chapter 2. All the parameters except the plate thickness *T* are stated in the figure. The boundary conditions look good (no rigid body motion expected) so we can start scripting the model. We start by creating an Abaqus Model by clicking in Abaqus CAE. This is the time to look if the model runs, not during scripting. Once everything is working (as in this cae file), you can start building your scripted model.



4.2.1) Header and variables

We take the header of the script from above and insert a meaningful name, my initials and a date:

```
# Plate with hole model for Abaqus/Python course, MP, 10-2020

from abaqus import *
from abaqusConstants import *
from caeModules import *
import os
import numpy as np
session.journalOptions.setValues(replayGeometry=COORDINATE,
                                 recoverGeometry=COORDINATE)
DIR0 = os.path.abspath('')
TOL = 1e-6
```

We state that the parameters are given in the N-mm-s system and define all of them. Taking time to write comments for all those parameters and writing down their dimensions helps you avoid errors. In this case, we check if the hole is really inside the plate: If not, we return an error.

```
# parameters for geometry, material, load and mesh (N-mm-s)
# ---------------------------------------------------------
b = 16.               # width of the plate (mm)
h = 20.               # height of the plate (mm)
radius = 4.5          # radius of the hole (mm)
uy = 0.08             # vertical displacement of the top (mm)
E_steel = 210000.     # Young's modulus of steel (MPa)
nu_steel = 0.3        # Poisson's ratio of steel (1)
sy_steel = 500.       # yield stress of steel (MPa)
mesh_size = 0.3       # edge length of the mesh (mm)

if radius >= b/2:
    raise ValueError('width b too small for the hole radius')
if radius >= h/2:
    raise ValueError('height h too small for the hole radius')
```

Already structuring the code by grouping commands and writing headers makes it easier to find problems in your code and later helps you for creating functions.

4.2.2) Model setup

We next take the template for resetting and redefining the model as `model` and add it to the *model.py* script:

```
# create the model
# -------------------------------------------------------
Mdb()                  # reset the model

model = mdb.models['Model-1']
```

Now its time for recording commands in Abaqus CAE and copying them into the *model.py* file. I created a sketch and drew the geometry of the plate in it, which consisted of four lines and one arc. After cleaning up the code and removing lots of unnecessary constraint commands, the sketch can be built like this:

```
# draw the sketch
s = model.ConstrainedSketch(name='plate', sheetSize=200.0)
s.ArcByCenterEnds(center=(0,0), point1=(0,radius),
                  point2=(radius,0), direction=CLOCKWISE)
s.Line(point1=(radius,0), point2=(b/2.,0))
s.Line(point1=(b/2.,0), point2=(b/2.,h/2.))
s.Line(point1=(b/2.,h/2.), point2=(0,h/2.))
s.Line(point1=(0,radius), point2=(0,h/2.))
```

As described above, this yielded from iteratively changing and running the code and checking what it did in Abaqus CAE. Creating the part is fairly simple, and was adapted to those lines:

```
# create the part
p = model.Part(name='plate', dimensionality=TWO_D_PLANAR,
               type=DEFORMABLE_BODY)
p.BaseShell(sketch=s)
```

As we discussed in chapter 3, the trickiest part of scripting the model is creating sets and surfaces. I did not record any commands for that but used the commands that were introduced in chapter 3 for selecting all faces, edges by their bounding box, and a reference point in this weird (but necessary) syntax:

```
# create sets, surfaces, and reference point
p.Set(name='all', faces=p.faces[:])
p.Set(name='x_sym', edges=p.edges.getByBoundingBox(xMax=TOL))
p.Set(name='y_sym', edges=p.edges.getByBoundingBox(yMax=TOL))
p.Set(name='top',edges=p.edges.getByBoundingBox(
      yMin=h/2.-TOL))

rp = p.ReferencePoint(point=(0,h/2,0))
p.Set(name='RP', referencePoints=(p.referencePoints[rp.id],))
```

The commands for creating materials, creating sections and assigning sections is straightforward again. Just record the commands and clean them up.

In the recorded commands, you will have lots of additional options like the `thickness` in `HomogeneousSolidSection`: You can just keep all of them, which makes your code bulky and confusing — Or you try removing some that seem irrelevant and check if your script still works.

```
# create material, create and assign sketchOptions
mat = model.Material(name='steel')
mat.Elastic(table=((E_steel, nu_steel), ))
mat.Plastic(table=((sy_steel, 0.0), ))

model.HomogeneousSolidSection(name='steel', material='steel',
```

```
                                thickness=None)

p.SectionAssignment(region=p.sets['all'], sectionName='steel',
                    thicknessAssignment=FROM_SECTION)
```

The next step is defining the assembly as `a` and creating an instance of your part:

```
# assembly
a = model.rootAssembly
inst = a.Instance(name='plate-1', part=p, dependent=ON)
```

Then we record and adapt the commands for creating a static load step and create a history output for the reference point. For the history output, as for boundary conditions and loads, we have to use the instance and not the part (`inst.sets`):

```
# step and hostory output
model.StaticStep(name='pull', previous='Initial',
    maxNumInc=1000, initialInc=0.1, minInc=1e-08, maxInc=0.1,
    nlgeom=ON)

model.HistoryOutputRequest(name='H-Output-2',
    createStepName='pull', variables=('U2', 'RF2'),
    region=inst.sets['RP'])
```

In the interaction module of Abaqus CAE, we defined a coupling of the upperr nodes and the reference point, which corresponds to this recorded command:

```
# constraint
model.Coupling(name='couple_top', couplingType=KINEMATIC,
    controlPoint=inst.sets['RP'], surface=inst.sets['top'],
    influenceRadius=WHOLE_SURFACE, u1=OFF, u2=ON, ur3=OFF)
```

We need three boundary conditions in the model: Two symmetries and one vertical displacement at the top. You can record creating one of these boundary conditions, and adapt it for the three of them. Just adapt the createStepName, the region and what displacements are constrained:

```
# boundaries and load
model.DisplacementBC(name='x_sym', createStepName='Initial',
                     region=inst.sets['x_sym'], u1=0)
model.DisplacementBC(name='y_sym', createStepName='Initial',
                     region=inst.sets['y_sym'], u2=0)
model.DisplacementBC(name='pull', createStepName='pull',
                region=inst.sets['RP'], u1=0, u2=uy, ur3=0)
```

For creating boundary conditions in Abaqus CAE, you can use the option *Symmetry/Antisymmetry/Encastre*: This yields separate commands for each type of boundary condition. I suggest to always use *Displacement/Rotation* and then set relevant displacements to zero. This always uses the same command `model.DisplacementBC()` and is thus easier for scripting the model.

Creating a mesh can get quite complicated. In this case, however, we only want to seed the whole part with a edge length of size and generate the mesh of the plate. For using edge-based seeds or changing the element types (plane stress is default for 2D elements), you can record and insert additinal commands:

```
# meshing
p.seedPart(size=mesh_size)
```

```
p.generateMesh()
```

So the model is completed! For running it, we create and run a job. If we want to evaluate the results in the same model, we can use the `waitForCompletion` function so the script waits until the job finishes before it continues:

```
job_name = 'pull-plate'

# create and run job
job = mdb.Job(name=job_name, model='Model-1', type=ANALYSIS,
              resultsFormat=ODB)
job.submit(consistencyChecking=OFF)
job.waitForCompletion()
```

Before submitting the job, you can save the model using `mdb.saveAs('model-name.cae')`. If the job crashes, you then have the latest cae file to check for errors.

4.2.3) Model evaluation

Evaluating the model using Python commands is usually quite different from what you get when you click in the Abaqus Viewer and record commands. Scripting those evaluations is described in chapter 4. We now apply those scripts to the plate model, starting with a small header that loads our odb file, defines the viewport as `vp` and displays the odb file:

```
# open the odb
odb = session.openOdb(job_name+'.odb')

vp = session.viewports['Viewport: 1']
vp.setValues(displayedObject=odb)
```

4.2.3.1) Print field output to an image

In the plate model, we do not want to output the full stress, strain and displacement fields. Otherwise, we would have needed to define additional field output of the coordinates like described in chapter 4. We want, however, create images of the stress fields. For that, we add the commands for setting the size of the viewport to the model script, run it and then rotate, zoom, and drag until we like the view of the model.

```
# print a field output image
# -------------------------------------------------------

# Change size of viewport (e.g. 300x200 pixel)
vp.restore()
# position of the viwport
vp.setValues(origin=(50,-50))
vp.setValues(width=150, height=200)
```

Then, we save the view using *View/Save* with the option *Save current*. The recording of this command is then added to our script. We only need to use the command `vp.view.setValues` to load this saved view and we are good to go. If we intend to change the geometry of our model a lot, this fixed view might be ok for some of the options but not for the others. Then, you can set the view to one of the default views (in the code below, it is the *Front* view) or use the option *Auto-fit* when saving the view.

```
# set up in Abaqus Viewer and copied here
session.View(name='User-1', nearPlane=24.242, farPlane=39.003,
    width=7.2278, height=9.3665, projection=PERSPECTIVE,
```

```
    cameraPosition=(2.5, 5, 31.623), cameraUpVector=(0, 1, 0),
    cameraTarget=(2.5, 5, 0), viewOffsetX=-1.3504,
    viewOffsetY=0.25069, autoFit=OFF)

# load the defined view
vp.view.setValues(session.views['User-1'])

# for a varied plate size, take an automatic front view
# vp.view.setValues(session.views)
```

Then, we need to define what kind of field output we want to display and for what frame we want to display the results. Those commands can be recorded and copied into the model script.

```
# view the right field output
vp.odbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ))
vp.odbDisplay.setPrimaryVariable(variableLabel='S',
     outputPosition=INTEGRATION_POINT,
     refinement=(INVARIANT, 'Mises'),)
# set the displayed frame
vp.odbDisplay.setFrame(step=0,frame=-1)
```

There are some view options regarding state blocks and the legend that I like to set. If you would like to keep some of the elements or prefer a different font in the legend, record similar commands as described in chapter 4:

```
# change the legend and what is displayed
vp.viewportAnnotationOptions.setValues(legendFont=
       '-*-verdana-medium-r-normal-*-*-140-*-*-p-*-*-*')
vp.viewportAnnotationOptions.setValues(triad=OFF, state=OFF,
    legendBackgroundStyle=MATCH,annotations=OFF,compass=OFF,
    title=OFF)
```

Finally, we can print the view of the field output to a png file. With the `imageSize` argument, you can change the resolution of the picture. Here, we create a png image that uses the model name with the ending *_mises*:

```
# print viewport to png file
session.printOptions.setValues(reduceColors=False, vpDecorations=OFF)
session.pngOptions.setValues(imageSize=(1500, 2000))
session.printToFile(fileName=job_name+'_mises', format=PNG,
                    canvasObjects=(vp,))
```

4.2.3.2) Evaluating history output

Following the description in chapter 4 on evaluation history output, we here obtain the history region by its name (must contain the string *'Node'*):

```
# evaluate the history output
# -------------------------------------------------------
step = odb.steps['pull']

# select the history region: name starts with 'Node '
hr_rp = [hr for hr in step.historyRegions
         if 'Node' in hr.name][0]
```

The history region then contains the history output, that we access here. We get the data for the vertical displacement (`U2`) and the vertical reaction force (`RF2`). Using the syntax of NumPy arrays, we obtain the second column of both displacement and reaction force with the index and stack those two columns into a new array `res_u_f`, that we save in a dat file:

```
# get vertical dispacement and reaction force data
# ((t,u),...), ((t,rf),...)
res_u = np.array(hr_rp.historyOutputs['U2'].data)
res_rf = np.array(hr_rp.historyOutputs['RF2'].data)

# only take the second colum of data and stack those columns
res_u_f = np.column_stack((res_u,res_rf))

# write u-rf data to file
np.savetxt(job_name+'_res_u_f.dat',res_u_f)
```

4.2.3.3) Evaluating field output along a path

Furthermore, we are interested in the stress distribution in the remaining cross-section of the plate. As described in chapter 4, we create a coordinate-based path:

```
# evaluate the stresses along path (y=0)
# -----------------------------------------------------

# create path from coordinates
pth = session.Path(name='Path-1', type=POINT_LIST,
                   expression=((radius,0,0), (b/2.,0,0)))
```

To evaluate field output along a path, we must first display the desired field output and set the frame for which we want to get the results:

```
# field output to be evaluated along path
vp.odbDisplay.setPrimaryVariable(variableLabel='S',
    outputPosition=INTEGRATION_POINT,refinement=(COMPONENT, 'S22'))

# setting the frame for path evaluation
vp.odbDisplay.setFrame(step=0,frame=-1)
```

And we evaluate the vertical stresses along 20 equidistant points over this path. This means that Abaqus interpolates the stress fields. Stating shape=UNDEFORMED and labelType=X_COORDINATE returns a list of undeformed x-coordinates / vertical stress pairs that we save to a dat file:

```
# output for 20 eqiuidistant points over path
# shape: path over DEFORMED or UNDEFORMED geometry

n_points = 20

# get vertical stress S22 over X_COORDINATE
xy = session.XYDataFromPath(name='XYData-1', path=pth,
    includeIntersections=False, shape=UNDEFORMED,
    projectOntoMesh=False, pathStyle=UNIFORM_SPACING,
    numIntervals=n_points, projectionTolerance=0,
    labelType=X_COORDINATE)

# write results to file
np.savetxt(job_name+'_res_s22_path.dat',xy.data)
```

4.2.4) Organizing the model using functions

Once you have your model fully scripted and tested some parameter options, it is time to tidy the script up! Collect all the commands that belong together in a function to get a function for geometry, sections, the assembly, boundaries, running the model and your evaluations. Think about what is needed as an argument in each function: The geometry function, for example,

would need the model and the geometry parameters as arguments. When a function creates something (like a part) that is needed in later functions, return this variable.

Create one function for the whole model creation, run, and evaluation that calls all the other functions. Check the model parameters in this function (or write a separate function for checking model parameters and call that function). This function could look like that:

```
# one funtion for the whole plate model(create, run, evaluate)

def plate_model(job_name,(b,h,radius),(E,nu,sy),uy,mesh_size):
    # check parameters for errors
    if radius >= b/2:
        raise ValueError('width b too small for hole radius')
    if radius >= h/2:
        raise ValueError('height h too small for hole radius')
    # reset model
    Mdb()
    model = mdb.models
    # make the parts, mesh it and return it
    p = make_geometry(model,(b,h,radius),mesh_size)
    # make materials and sections
    make_sections(model,(E,nu,sy))
    # make the assembly & constraint
    inst = make_assembly(model,p)
    # create step and boundaries
    make_boundaries(model,inst,uy)
    # save the model
    mdb.saveAs(job_name)
    # run the model
    run_model(model,job_name)
    # evaluate: create image
    evaluate_image(job_name)
    # evaluate: history output and path
    evaluate_ho_path(job_name)
    return model
```

Note that by writing `inst = make_assembly(model,p)` you take the returned argument from the function and pass it to the variable inst to be used later. The individual functions are not listed here but can be found in the Appendix.

You might ask yourself why we do not define the set and step names as variables: For a simple script like that, this would only complicate things. Just make sure that you have all your sets and surfaces defined in the geometry function that you will need in the other functions.

You can call the model function just by stating values or variables for the parameters:

```
# call the model function
plate_model('small-plate',(20.,15.,7.),(1200.,0.3,40),0.05,1.)
```

If you want to do that for 3 different parameter combinations, just write multiple function calls. If you want to vary a parameter in a certain rage and look at model results, you can write a for loop that does that (note that in some way you should remember what job name corresponds to what parameters: Here, we just write the height h into the job name):

```
# define a list for the height
h_list =

# a for loop to vary the height in the model
for h in h_list:
    # writing the h value into the job name
    job_name = 'plate-h-'+str(int(h))
```

```
plate_model(job_name,(20.,h,7.),(1200.,0.3,40),0.05,1.)
```

Now, this is what we all have been waiting for! You have all the parameters at your fingertips and can vary whatever you want. The ~~sky~~ computation time is the limit!

4.3) Exercises

Now the time has come to script your own model. As you have pracitced in the last two chapters on small model details, you will now do for your whole model.

1. Collect all parameters of your model and write the part of your model script where you define all those parameters and do some sanity checks.
2. Build up your scripted model feature by feature as described above. Test this after each step.
3. Build up your evaluation as described above and with help from the previous chapter. Test some different kinds of parameter sets and look at the results. Did you expect that?

5) Tidying up

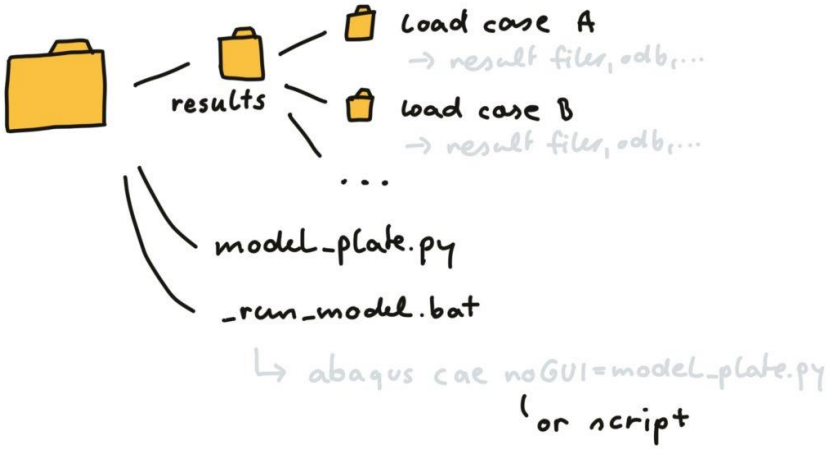
You now know how to build up a scripted Abaqus model and automatically evaluate its results. You might already have noticed that your very valuable Python file gets lost between all the result files. In this chapter, you learn to create a neat folder structure for your result files and delete result files that you do not need anymore. For your day-to-day work, this will be at least as important as being able to script your models.

Between all the copying of Abaqus commands, writing into a Python file, testing the script, and looking at results, one could get quite confused. Confused enough to delete the model file or to make some changes that break the model file: To stay safe, I advise you to look into version control, which can be done with Git. An introduction to this is given here, and you can also synch your model with the GitLab server of the University of Leoben (if you are either studying at or employed by the Montanuniversitaet) or sites like BitBucket or GitHub.

5.1) Run models in sub-folders

Changing the working directory of Abaqus (where your models are run and the *rpy* file is saved) is quite simple: You can just write `os.chdir()` and state the new path for the working directory in the brackets. This lets you work in a more structured way: Change into a results directory where you run your model and where you can delete files without worrying too much. So why didn't we introduce this command two chapters ago? Because as you build your model, you will have lots of errors to deal with, as you are in your subfolder: Then, you would have to manually change back to your initial folder to make the `execfile('model.py')` command work again.

This is how a neat folder structure could look like. There is a resuls folder, containing a folders for each load case. Furthermore, we start running the Abaqus model by running a batch file (if you use Windows).



As shown in the figure, it is a good idea to introduce a parameter file model_par.py that contains the parameter definition of your model and is loaded into the model Python script by:

```
execfile('model_par.py')
```

This parameter file can then be copied in the folder where the script is run, so you can later look up what parameters you used in that model. In Python, copying files is pretty easy, and if you are already in your results folder, you can copy the parameter file to the results folder in the following way:

```
import shutil
# copy files from 2 folders up (../../) to current folder
shutil.copyfile('../../model_par.py', 'model.par.py')
```

For your model, I suggest setting up a function that first clears the folder you want to work in (or create it), then changes into it, then does all your building and evaluating the model stuff, and then returns back to your initial directory:

```
import os, shutil

# absolut working path
DIR0 = os.path.abspath('')

# insert more parameters after run_folder
def create_model(run_folder):
    # delete folder if it exists
    if os.path.exists(dir_name):
        shutil.rmtree(dir_name)
    shutil.copyfile('../../model_par.py', 'model.par.py')
    # create folder
    os.mkdir(dir_name)
    # go into folder
    os.chdir(dir_name)
    # model setup, run and evaluation
    # ...
    # go back to initial folder
    os.chdir(DIR0)
    return
```

You can adapt this function to build up, run and evaluate your model, but do so in a sub-folder that you can pass to the function as `run_folder`.

Note that directories in Python can be denoted by either `\\` or `/` in Windows, but if you use `/`, this works both for Windows and Linux. So in the above example, you might pass the string `'results/load_case_a'` as the `run_folder`.

5.2) Deleting unwanted result files

With `os.remove()`, you can delete a file that you state in the brackets. For us, it can be interesting to delete files that have certain extensions like *.stt* and *.mdl* after the job has finished. We can use a `for` loop to find all the files with such endings and delete them. This is already a function with a default list of file types type_list:

```
def remove_files(dir0,type_list=('com','sim','prt','msg',
                                 'reg')):
    # get all files and folders in dir0
    file_list = os.listdir(dir0)
    # go through the files and delete some of them
    for file in file_list:
        if file.split('.')[-1] in type_list:
            try:
                os.remove(dir0+'/'+file)
            except:
                print('file '+file+' could not be deleted!')
    return
```

You can directly copy this function into your model file. But be careful: `os.remove()` can be rather dangerous and delete stuff that you would like to keep. If you want to remove a folder and all its content, `os.remove()` does not work. For that, there is the `rmtree` function in the `shutil` module. If you try to delete a folder in that way, this will raise an error. So it is best to first check if the path exists with the command `os.path.exists()`:

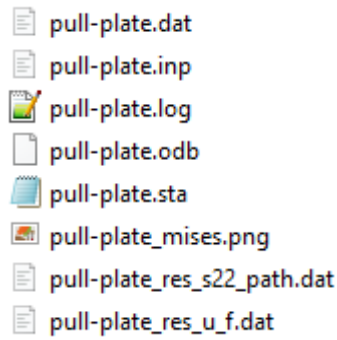
```
import shutil, os

# create folder (delete existing folder)
if os.path.exists(dir_name):
    shutil.rmtree(dir_name)
```

5.3) Exporting and Importing Data

If you work with data science and Python, you use the module NumPy a lot. Numpy also has some neat functions for loading and saving NumPy-arrays (those arrays need to have the same data type — string, float, integer, etc. — in all entries) with some very simple functions:

```
import numpy as np

# write the numpy array f_u_arr to a .dat ASCI file
np.savetxt('output.dat',f_u_arr)

# load a numpy array from a ASCI .dat file
f_u_arr_loaded = np.loadtxt('output.dat')
```

These `savetxt()` and `loadtxt()` commands use ASCI files that can also be opened with a text editor. If you only want to store the data, not look at it directly and load it again with Python, you can use `save()` and `load()` instead of those commands to write to and read from binary files.



In some cases, it would be good to write something more complex than an array of similar strings, floats, or integers to a file. Maybe you have created a dictionary with all the step names as keys, and the maximum occurring stresses as the values. You can write this dictionary to a file or read a dictionary from such a file with the json module (note that such a dictionary cannot contain NumPy arrays):

```
import json

# write dictionary to dat-file
with file('output_dict.json','w') as f:
    json.dump(result_dict,f)

# read a dictionary from a json-formatted dat-file
with file('output_dict.json','r') as f:
    result_dict_loaded = json.load(f)
```

Exercises

1. Use your model from chapter 4 and adapt it so that it is run in a subfolder and all the unwanted Abaqus files like .prt, .msg, etc. are deleted afterwards.
2. Create a separate Python file, Jupyter Notebook, or plotting sofware of your choice to plot the model results. Create plots that compare different sets of load parameters.

Appendix A: *model_plate.py*

Get the file here: www.martinpletz.com/fe-scripting-8

```
# Plate with hole model for Abaqus/Python course, MP, 09-2020

from abaqus import *
from abaqusConstants import *
from caeModules import *
import os
import numpy as np
session.journalOptions.setValues(replayGeometry=COORDINATE,
                                 recoverGeometry=COORDINATE)
DIR0 = os.path.abspath('')
TOL = 1e-6

# Model functions
# -------------------------------------------------------

def make_geometry(model,(b,h,radius),mesh_size):
    # draw the sketch
    s = model.ConstrainedSketch(name='plate', sheetSize=200.0)
    s.ArcByCenterEnds(center=(0,0), point1=(0,radius),
                      point2=(radius,0), direction=CLOCKWISE)
    s.Line(point1=(radius,0), point2=(b/2.,0))
    s.Line(point1=(b/2.,0), point2=(b/2.,h/2.))
    s.Line(point1=(b/2.,h/2.), point2=(0,h/2.))
    s.Line(point1=(0,radius), point2=(0,h/2.))

    # create the part
    p = model.Part(name='plate', dimensionality=TWO_D_PLANAR,
                   type=DEFORMABLE_BODY)
    p.BaseShell(sketch=s)

    # create sets, surfaces, and reference point
    p.Set(name='all', faces=p.faces[:])
    p.Set(name='x_sym',
          edges=p.edges.getByBoundingBox(xMax=TOL))
    p.Set(name='y_sym',
          edges=p.edges.getByBoundingBox(yMax=TOL))
    p.Set(name='top',
          edges=p.edges.getByBoundingBox(yMin=h/2.-TOL))

    rp = p.ReferencePoint(point=(0,h/2,0))
    p.Set(name='RP',
          referencePoints=(p.referencePoints[rp.id],))
    # meshing
    p.seedPart(size=mesh_size)
    p.generateMesh()
    return p

def make_sections(model, p, (E,nu,sy)):
    # create material, create and assign sketchOptions
    mat = model.Material(name='steel')
```

```
    mat.Elastic(table=((E_steel, nu_steel), ))
    mat.Plastic(table=((sy_steel, 0.0), ))

    model.HomogeneousSolidSection(name='steel',
        material='steel', thickness=None)

    p.SectionAssignment(region=p.sets['all'],
        sectionName='steel',thicknessAssignment=FROM_SECTION)
    return

def make_assembly(model,p):
    # assembly
    a = model.rootAssembly
    inst = a.Instance(name='plate-1', part=p, dependent=ON)
    # constraint
    model.Coupling(name='couple_top',surface=inst.sets['top'],
        controlPoint=inst.sets['RP'], couplingType=KINEMATIC,
        influenceRadius=WHOLE_SURFACE, u1=OFF, u2=ON, ur3=OFF)
    return inst

def make_boundaries(model,inst,uy):
    # step and hostory output
    step = model.StaticStep(name='pull', previous='Initial',
               maxNumInc=1000, initialInc=0.1, minInc=1e-08,
               maxInc=0.1, nlgeom=ON)

    model.HistoryOutputRequest(name='H-Output-2',
        createStepName='pull', variables=('U2', 'RF2'),
        region=inst.sets['RP'])

    # boundaries and load
    model.DisplacementBC(name='x_sym', u1=0,
        createStepName='Initial', region=inst.sets['x_sym'])
    model.DisplacementBC(name='y_sym', u2=0,
        createStepName='Initial', region=inst.sets['y_sym'])
    model.DisplacementBC(name='pull', createStepName='pull',
        region=inst.sets['RP'], u1=0, u2=uy, ur3=0)
    return

def run_model(model,job_name):
    # create and run job
    job = mdb.Job(name=job_name, model='Model-1',
              type=ANALYSIS,
                resultsFormat=ODB)
    job.submit(consistencyChecking=OFF)
    job.waitForCompletion()
    return

def evaluate_image(job_name):
    # open the odb
    odb = session.openOdb(job_name+'.odb')
    vp = session.viewports['Viewport: 1']
    vp.setValues(displayedObject=odb)
    # Change size of viewport (e.g. 300x200 pixel)
    vp.restore()
    # position of the viwport
    vp.setValues(origin=(50,-50))
    vp.setValues(width=150, height=200)
    # set up in Abaqus Viewer, recorded, and copied here
    """
    session.View(name='User-1', nearPlane=24.242,
        farPlane=39.003, width=7.2278, height=9.3665,
        projection=PERSPECTIVE, cameraUpVector=(0, 1, 0),
        cameraPosition=(2.5, 5, 31.623),
        cameraTarget=(2.5, 5, 0), viewOffsetX=-1.3504,
        viewOffsetY=0.25069, autoFit=OFF)
    # load the defined view
    vp.view.setValues(session.views['User-1'])
    """
```

```
    # for a changing plate size, take an automatic front view
    vp.view.setValues(session.views['Front'])
    # view the right field output
    vp.odbDisplay.display.setValues(plotState=
        (CONTOURS_ON_DEF, ))
    vp.odbDisplay.setPrimaryVariable(variableLabel='S',
        outputPosition=INTEGRATION_POINT,
        refinement=(INVARIANT, 'Mises'),)
    # change the legend and what is displayed
    vp.viewportAnnotationOptions.setValues(legendFont=
        '-*-verdana-medium-r-normal-*-*-140-*-*-p-*-*-*')
    vp.viewportAnnotationOptions.setValues(triad=OFF,
        state=OFF, legendBackgroundStyle=MATCH,
        annotations=OFF, compass=OFF, title=OFF)
    # print viewport to png file
    session.printOptions.setValues(reduceColors=False,
        vpDecorations=OFF)
    session.pngOptions.setValues(imageSize=(1500, 2000))
    session.printToFile(fileName=job_name+'_mises', format=PNG,
                canvasObjects=(vp,))
    odb.close()
    return

def evaluate_ho_path(job_name):
    # open the odb
    odb = session.openOdb(job_name+'.odb')
    vp = session.viewports['Viewport: 1']
    vp.setValues(displayedObject=odb)
    # evaluate the history output
    # -------------------------------------------------------
    step = odb.steps['pull']
    # select the history region: starts with 'Node '
    hr_rp = [i for i in step.historyRegions.values()
            if i.name.startswith('Node ')][0]
    # get vertical dispacement and reaction force data
    # ((t,u),...), ((t,rf),...)
    res_u = np.array(hr_rp.historyOutputs['U2'].data)
    res_rf = np.array(hr_rp.historyOutputs['RF2'].data)
    # stack second columns
    res_u_f = np.column_stack((res_u[:,1],res_rf[:,1]))
    # write u-rf data to file
    np.savetxt(job_name+'_res_u_f.dat',res_u_f)
    # evaluate the stresses along path (y=0)
    # -------------------------------------------------------
    # create path from coordinates
    pth = session.Path(name='Path-1', type=POINT_LIST,
                    expression=((radius,0,0), (b/2.,0,0)))
    # field output to be evaluated along path
    vp.odbDisplay.setPrimaryVariable(variableLabel='S',
        outputPosition=INTEGRATION_POINT,
        refinement=(COMPONENT, 'S22'))
    # output for 20 eqiuidistant points over path
    # shape: path over DEFORMED or UNDEFORMED geometry
    n_points = 20
    # get vertical stress S22 over X_COORDINATE
    xy = session.XYDataFromPath(name='XYData-1', path=pth,
        includeIntersections=False, shape=UNDEFORMED,
        projectOntoMesh=False, pathStyle=UNIFORM_SPACING,
        numIntervals=n_points, projectionTolerance=0,
        labelType=X_COORDINATE)
    # write results to file
    np.savetxt(job_name+'_res_s22_path.dat',xy.data)
    odb.close()
    return

# one function for the plate model (create, run, evaluate)
def plate_model(job_name,(b,h,radius),(E,nu,sy),uy,mesh_size):
    # check parameters for errors
    if radius >= b/2:
```

```
        raise ValueError('width b too small for hole radius')
    if radius >= h/2:
        raise ValueError('height h too small for hole radius')
    # reset model
    Mdb()
    model = mdb.models['Model-1']
    # make the parts, mesh it and return it
    p = make_geometry(model,(b,h,radius),mesh_size)
    # make materials and sections
    make_sections(model,p,(E,nu,sy))
    # make the assembly & constraint
    inst = make_assembly(model,p)
    # create step and boundaries
    make_boundaries(model,inst,uy)
    # save the model
    mdb.saveAs(job_name)
    # run the model
    run_model(model,job_name)
    # evaluate: create image
    evaluate_image(job_name)
    # evaluate: history output and path
    evaluate_ho_path(job_name)
    return model

# parameters for geometry, material, load, and mesh (N-mm-s)
# ----------------------------------------------------
b = 16.               # width of the plate (mm)
h = 20.               # height of the plate (mm)
radius = 4.5          # radius of the hole (mm)
uy = 0.08             # vertical displacement of the top (mm)
E_steel = 210000.     # Young's modulus of steel (MPa)
nu_steel = 0.3        # Poisson's ratio of steel (1)
sy_steel = 500.       # yield stress of steel (MPa)
mesh_size = 0.3       # edge length of the mesh (mm)


# run the model function
# ----------------------------------------------------

plate_model('plate-test',(b,h,radius),
            (E_steel,nu_steel,sy_steel), uy,mesh_size)
```