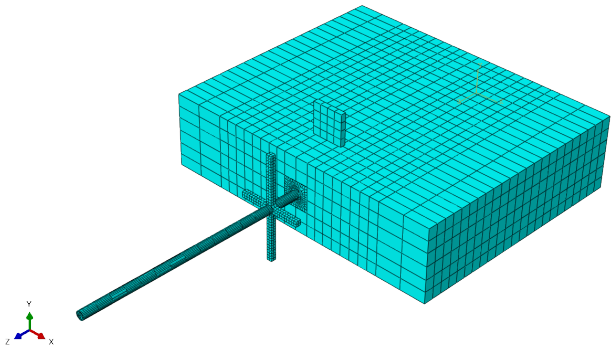
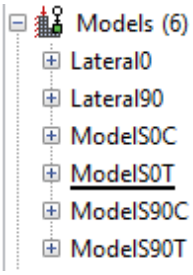
How to calibrate a model

Let us take Resch's master thesis as an example on how to use scripting in Abaqus.



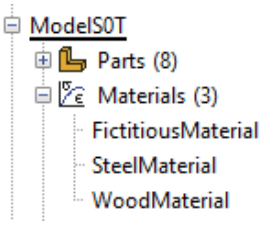
Figur 1: A snapshot of the configuration for tension parallel to grain.

In my CAE file I have four different configurations (ignore *Lateral*0 and *Lateral*90) of the experiment from Resch's master thesis, shown in figure 2. The number 0 or 90 determines if the applied force is parallel or perpendicular to the grain, and C or T determines whether it is in compression or tension.



Figur 2: Model tree.

The materials are all saved in the file, as shown in figure 3, in the same manner as usual. The steel and wood material are constant and are not included in the Python script. The fictitious material is included in the script, as figuring out the correct coefficients is part of the calibration.



Figur 3: Materials.

When making the model in Abaqus, I recorded everything from start to finish using the *Macro Manager*, resulting in a macro that has all the interesting parts saved in a Python file. From there on it was a matter of just figuring out what does what, and remove everything that was unnecessary. Figure 4, for example, shows the Python code that changes the engineering constants for the fictitious material. **modelAbaqus** is a variable used to tell Abaqus which model from the model tree in figure 2 to choose. Similarly, such a variable could be used instead of **'FictitiousMaterial'**, but since all the models in the model tree have the same materials, this can be left hard-coded in the script. Notice also the variable **modifier** which will be discussed later.

```
# Specify material coefficients for fictitious material
mdb.models[modelAbaqus].materials['FictitiousMaterial'].Elastic(type=ENGINEERING_CONSTANTS,
table=((E1, E2, E3, Nu12, Nu13, Nu23, modifier*G12, modifier*G13, modifier*G23),) )
```

Figur 4: A snapshot code specifying the coefficients for the fictitious material.

Using Python to run Abaqus files

So how do we use Python to run a simulation in Abaqus? There are probably many ways to do this, but I have chosen the following. First, I have one script with all the various functions you want your calibration model to have. I have called this script **abaqusCommands.py**, and you can see a snapshot of it in figure 5. This script is made in such a way that you can run it in Abaqus, using the **File > Run Script...** function, with all the constraints that entails. For example, it needs to import the same libraries as your Abaqus macro's, and it is very easy to get errors.

The **openJob** function instructs Abaqus to open the .cae file. In this case, I have all models saved in one file, and there is no need to parameterise this. Next, the **specifyCoefficients** function is used to specify *all* parameters. Notice that the function takes in five arguments, **modelAbaqus, modifier, embeddedLength, stiffness,** and **redInt**.

```
def openJob(): openMdb(pathName='C:/temp/AlisaExperimentDB.cae')

def specifyCoefficients(modelAbaqus, modifier, embeddedLength, stiffness, redInt):


    # Specify material coefficients for fictitious material
    mdb.models[modelAbaqus].materials['FictitiousMaterial'].Elastic(type=ENGINEERING_CONSTANTS,
    table=((E1, E2, E3, Nu12, Nu13, Nu23, modifier*G12, modifier*G13, modifier*G23),) )


    # Specify embedded length (default = 110)
    a = mdb.models[modelAbaqus].rootAssembly
    a.translate(instanceList=('RodCoreOuter-1', ), vector=(0.0, 0.0, -(embeddedLength-110)))
    a = mdb.models[modelAbaqus].rootAssembly
    a.translate(instanceList=('RodCoreInner-1', ), vector=(0.0, 0.0, -(embeddedLength-110)))

    # Specify cohesive zone stiffness property
    mdb.models[modelAbaqus].interactionProperties['Cohesive'].cohesiveBehavior.setValues(table=((0.1, stiffness, stiffness), ))
```

Figur 5: Important functions from abaqusCommands.py

- **modelAbaqus** is a string that specifies which model to use,
- **modifier** is a float that is multiplied with the shear moduli of the ficitious material, usually between 0 and 1,
- **embeddedLength** is an integer that specifies how deep you want the rod to be embedded in the wood,
- **stiffness** is an integer used to specify the cohesive stiffness property of the cohesive zone, and
- **redInt** is a boolean (True/False) that specifies whether to use full or reduced integration.

There are not that many variables that need paramterising in this thesis. Apart from the ability to specify engineering constants (shown in figure 4), you also see the ability to specify the embedded length of the rod. This is used by translating the two parts that make up the rod: **'RodCoreOuter-1'** and **'RodCoreInner-1'**. Notice here that the model already has the rod embedded by 110 mm. At the bottom you can see the functionality for specifying cohesive stiffness properties, K_{nn}, K_{ss} and K_{tt}.

```
def runJob(filename, modelAbaqus):

    # Creating, submitting and waiting for completion of job
    mdb.Job(name=filename, model=modelAbaqus, description='', type=ANALYSIS,
    atTime=None, waitMinutes=0, waitHours=0, queue=None, memory=90,
    memoryUnits=PERCENTAGE, getMemoryFromAnalysis=True,
    explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE, echoPrint=OFF,
    modelPrint=OFF, contactPrint=OFF, historyPrint=OFF, userSubroutine='',
    scratch='', resultsFormat=ODB, multiprocessingMode=DEFAULT, numCpus=1,
    numGPUs=0)
    mdb.jobs[filename].submit(consistencyChecking=OFF)
    mdb.jobs[filename].waitForCompletion()
```

Figur 6: Running the simulation

Figure 6 shows the bit of code that instructs Abaqus to create and run a simulation, or a *job* as Abaqus calls it. It only needs to know what to call file, the **filename**, and of course it needs to know which model to run, **modelAbaqus**. Notice that the function **waitForCompletion()** is included in the last line. This is important for the next bit of code, which is concerned with printing the results to a file. Before this can happen, the job needs to be completed to avoid errors. *Add part about what to do if you want to add functions*

Fetching results

The next part is all about getting the relevant results from the simulation. When you are in the **Result** section of Abaqus, you are inspecting an ODB file, or the output database, where you can basically get all kinds of interesting information about stresses, forces, strains and displacements of every single node in your model. For this thesis, we were only interested in

the concentrated force on a single node, in addition to the relative displacement between four pairs of nodes. The **openResults** function is partially displayed in figure 7. Other than the two first lines, concerned with opening the ODB file, the function is a messy copy from the recorded macro. Basically, it takes the XY data from the selected nodes (in this case, hard-coded into the script for convenience), and writes them to a file using the **writeXYReport** function. The file is saved as **filename.rpt** (bears a resemblance to .txt file) in the chosen directory, for later processing. The results printed to file can be seen in figure 8. This is Abaqus' default way of reporting. How to process it will be covered later.

```
def openResults(filename, modelAbaqus):
    o3 = session.openOdb(name='C:/Users/Johannes/'+ filename +'.odb')
    odb = session.odbs['C:/Users/Johannes/'+ filename +'.odb']

    # Kind of a black box and not very elegant, but opens results, creates XY data,
    # prints the data to a .rpt file in a specified folder and deletes the data to
    # avoid conflicts with a later simulation

    session.xyDataListFromField(odb=odb, outputPosition=NODAL, variable=(('CF',
            NODAL, ((COMPONENT, 'CF3'), )), ), nodeSets=(
            "RODCOREINNER-1.LOADREFERENCENODE", ))
```

Figur 7: Code used to open the results from the simulation.

X	CF:CF3 PI: RODCOR EINNER-1 N: 12	U:U3 PI: MEASUREP LATE-1 N: 23	U:U3 PI: MEASUREP LATE-2 N: 23	U:U3 PI: MEASURIN GDEVICE-1 N: 2	U:U3 PI: MEASURIN GDEVICE-1 N: 7	U:U3 PI: MEASURIN GDEVICE-1 N: 26	U:U3 PI: MEASURIN GDEVICE-1 N: 31	U:U3 PI: WOODBLOC KCOARSE-1 N: 112	U:U3 PI: WOODBLOC KCOARSE-1 N: 116
0.	0.	0.	0.	0.	0.	0.	0.	0.	0.
1.	6.56	1.73807E-03	1.7385E-03	4.23412E-03	4.22324E-03	4.20604E-03	4.22192E-03	1.53319E-03	1.53322E-03
2.	13.12	3.47613E-03	3.477E-03	8.46824E-03	8.44647E-03	8.41207E-03	8.44384E-03	3.06639E-03	3.06643E-03
3.	19.68	5.2142E-03	5.2155E-03	12.7024E-03	12.6697E-03	12.6181E-03	12.6658E-03	4.59958E-03	4.59965E-03
4.	26.24	6.9522GE-03	6.95401E-03	16.9365E-03	16.8929E-03	16.8241E-03	16.8877E-03	6.13277E-03	6.13287E-03
5.	32.8	8.69033E-03	8.69251E-03	21.1706E-03	21.1162E-03	21.0302E-03	21.1096E-03	7.66597E-03	7.66608E-03
6.	39.36	10.4284E-03	10.431E-03	25.4047E-03	25.3394E-03	25.2362E-03	25.3315E-03	9.19916E-03	9.1993E-03
7.	45.92	12.1665E-03	12.1695E-03	29.6388E-03	29.5627E-03	29.4422E-03	29.5534E-03	10.7324E-03	10.7325E-03
8.	52.48	13.9045E-03	13.908E-03	33.873E-03	33.7859E-03	33.6483E-03	33.7754E-03	12.2655E-03	12.2657E-03
9.	59.04	15.6426E-03	15.6465E-03	38.1071E-03	38.0091E-03	37.8543E-03	37.9973E-03	13.7987E-03	13.7989E-03
10.	65.6	17.3807E-03	17.385E-03	42.3412E-03	42.2324E-03	42.0604E-03	42.2192E-03	15.3319E-03	15.3322E-03
11.	72.16	19.1187E-03	19.1235E-03	46.5753E-03	46.4556E-03	46.2664E-03	46.4411E-03	16.8651E-03	16.8654E-03
12.	78.72	20.8568E-03	20.862E-03	50.8094E-03	50.6788E-03	50.4724E-03	50.663E-03	18.3983E-03	18.3986E-03
13.	85.28	22.5949E-03	22.6005E-03	55.0436E-03	54.9021E-03	54.6785E-03	54.8849E-03	19.9315E-03	19.9318E-03
14.	91.84	24.3329E-03	24.339E-03	59.2777E-03	59.1253E-03	58.8845E-03	59.1069E-03	21.4647E-03	21.465E-03
15.	98.4	26.071E-03	26.0775E-03	63.5118E-03	63.3485E-03	63.0905E-03	63.3288E-03	22.9979E-03	22.9982E-03
16.	104.96	27.8091E-03	27.816E-03	67.7459E-03	67.5718E-03	67.2966E-03	67.5507E-03	24.5311E-03	24.5315E-03
17.	111.52	29.5471E-03	29.5545E-03	71.98E-03	71.795E-03	71.5026E-03	71.7726E-03	26.0643E-03	26.0647E-03
18.	118.08	31.2852E-03	31.293E-03	76.2141E-03	76.0182E-03	75.7086E-03	75.9945E-03	27.5975E-03	27.5979E-03
19.	124.64	33.0233E-03	33.0315E-03	80.4483E-03	80.2415E-03	79.9147E-03	80.2165E-03	29.1307E-03	29.1311E-03
20.	131.2	34.7613E-03	34.77E-03	84.6824E-03	84.4647E-03	84.1207E-03	84.4384E-03	30.6639E-03	30.6643E-03
21.	137.76	36.4994E-03	36.5085E-03	88.9165E-03	88.688E-03	88.3267E-03	88.6603E-03	32.1971E-03	32.1975E-03
22.	144.32	38.2375E-03	38.247E-03	93.1506E-03	92.9112E-03	92.5328E-03	92.8822E-03	33.7302E-03	33.7308E-03
23.	150.88	39.9755E-03	39.9855E-03	97.3848E-03	97.1344E-03	96.7388E-03	97.1041E-03	35.2634E-03	35.264E-03
24.	157.44	41.7136E-03	41.724E-03	101.619E-03	101.358E-03	100.945E-03	101.326E-03	36.7966E-03	36.7972E-03
25.	164.	43.4516E-03	43.4625E-03	105.853E-03	105.581E-03	105.151E-03	105.548E-03	38.3298E-03	38.3304E-03

Figur 8: The output in .rpt file.

Running Abaqus in the terminal

Figure 9 shows all the previously discussed functions neatly collected in a single function, called **quickJob**, which is meant to do everything from start to finish. If you were to call this function in your Python script, nothing would happen. That's because these are all instructions that must be run in Abaqus. You could use **File > Run Script...**, select, and run this Python file, but there would be no way to provide the arguments needed to run the various functions. It would also mean that you would have to open Abaqus every time you wanted to run a script, which is not very efficient.

The solution is to exploit the fact that Abaqus can run through the terminal, or **Command Promt (CMD)** as it is called on Windows. Let's say your file is called **script.py**, and you

```
def quickJob(filename, modelAbaqus, modifier, embeddedLength, stiffness, redInt):
    openJob()
    specifyCoefficients(modelAbaqus, modifier, embeddedLength, stiffness, redInt)
    runJob(filename, modelAbaqus)
    openResults(filename, modelAbaqus)
```

Figur 9: Pulling it all together.

wanted to run the script without opening Abaqus first. You can open **CMD** and type the following:

```
abaqus cae script="script.py"
```

This would be the exact same as pressing **File > Run Script...** and selecting the script. It would, however, still open the Abaqus window, which gets in the way of automating the process. The solution is to use the **noGUI** functionality:

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

This would run the same script, but without opening the Abaqus window. Unfortunately, we still have not solved the problem of passing the arguments to the functions, and still involves a bit of manual labour. Luckily, you put all that in a Python script.

```
import os
os.system('cmd /c abaqus cae noGUI="script.py"')
```

These two lines in the Python script will allow you to run Abaqus from while only having your IDE open. In figure 10 you can see my **runCommand** function, used to perform this task. All the arguments have been discussed before, except for **noGUI=True**, which is included so that I have the ability to decide if I want to open the Abaqus GUI or not, with the default setting being that it does not open.

```
def runCommand(filename, modelAbaqus, modifier, embeddedLength, stiffness, redInt, noGUI=True):
    app = " -- " + filename + " " + str(modelAbaqus) + " " + str(modifier) + " " + str(
        embeddedLength) + " " + str(000) + " " + str(stiffness)+ " " + str("perp") + " " + str(redInt)
    if noGUI: command = r'abaqus cae noGUI="C:\Users\Johannes\Documents\Masteroppgave\PythonScripts\abaqusCommands.py"'
    else: command = r'abaqus cae script="C:\Users\Johannes\Documents\Masteroppgave\PythonScripts\abaqusCommands.py"'
    print("\nModel: " + str(modelAbaqus) + "\nEmbedded length: " + str(embeddedLength) +  "\nMod: " + str(
        modifier) + "\nStiffness: " + str(stiffness) + "\nReduced integration: " + str(redInt) + '\n')
    os.system('cmd /c ' + command + app)
```

Figur 10: The function used to run Abaqus.

Notice also that all the arguments are saved in the variable **app**. These are all appended to the command used to run Abaqus. Here is an attempt to explain the process. Let's say you have a Python script called runCommand.py, containing one function called runCommand(). At the end of the system command, you add three arguments: arg1, arg2, and arg3. These will be used in the script called script.py.

```
def runCommand():
    os.system('cmd /c abaqus cae noGUI="script.py" arg1 arg2 arg3')
```

In the called script.py, let's say you have a function called runJob(), which takes in three arguments. The command line arguments added at the end of runCommand() can then be used in runJob().

```
import sys

def runJob(filename, modelAbaqus, stiffness):
    arg1 = sys.argv[-3]
    arg2 = sys.argv[-2]
    arg3 = sys.argv[-1]

runJob(arg1, arg2, arg3)
```

This is how I manage to run Abaqus in my IDE. In figure 11 you see the end of the script that runs in Abaqus. Six arguments are saved in appropriate variables, and lastly, the function quickJob, also shown in figure 9, is called. Done this way, the script abaqusCommands.py is fit to run in Abaqus without producing errors, as all arguments are passed to it from the script in figure 10.

```
# Arguments passed through runAbaqus.py
argFilename = str(sys.argv[-6])
argModelAbaqus = str(sys.argv[-5])
argModifier = float(sys.argv[-4])
argEmbeddedLength = float(sys.argv[-3])
argStiffness = float(sys.argv[-2])
argRedInt = bool(sys.argv[-1])

# Running the job
quickJob(argFilename, argModelAbaqus, argModifier, argEmbeddedLength, argStiffness, argRedInt)
```

Figur 11: The last lines of abaqusCommands.py

Processing results

So now we know how to run Abaqus without any manual labour. Let us now turn our attention to how you could automate the process even further, or just produce informative plots. Figure 8 shows the RPT file. In this case, the model has 25 frames, resulting in the leftmost column showing 25 steps. Then there is the concentrated force, which is gradually increased to $164N$ per node, and the rest are the displacements. I have created a Python script that reads the data from this file, cleans it up and calculates the relative displacements, and finally saves that in a CSV file. It is very possible to calculate, for example, the stiffness, compare that to some predetermined acceptable value, and write code to increase or decrease some variable based

on that – thereby automating the calibration process. I decided it was easier test intervals, and just plot them.