

# **Scripting in Abaqus with abapys**

Usage of abapys v0.6 library with code examples

Dominik Zobel

version: January 26, 2021

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Scripting in Abaqus</b>	<b>3</b>
2.1	Scripting Resources in Abaqus . . . . .	3
2.2	Accessing Objects in Abaqus . . . . .	4
2.3	Useful Initial Commands when Accessing the Abaqus Object Model . . . .	6
2.4	Saving and Running a Script in Abaqus . . . . .	6
<b>3</b>	<b>Working with abapys</b>	<b>8</b>
3.1	How to Include abapys in a Script . . . . .	8
3.2	Overview of abapys Functions . . . . .	9
<b>4</b>	<b>Viewport Manipulation</b>	<b>11</b>
4.1	Opening an Output Database . . . . .	11
4.2	Adjusting the Viewport Size . . . . .	11
4.3	Saving the Current Viewport as an Image . . . . .	13
4.4	Adjusting the Viewport Style . . . . .	14
4.5	Adjusting the Model View . . . . .	16
4.6	Polishing a Viewport for Presentation . . . . .	16
4.7	Additional Image Saving Functions . . . . .	18
4.7.1	Printing High Resolution Images . . . . .	18
4.7.2	Creating Videos . . . . .	19
<b>5</b>	<b>Selecting Objects</b>	<b>21</b>
5.1	Elements and Sequences . . . . .	21
5.2	Labels and Indices . . . . .	22
5.3	Position and Coordinates . . . . .	24
5.4	Conditional Selection . . . . .	24
5.5	Sorting by Direction . . . . .	26
5.6	Direction Based Edge Selection . . . . .	26

<b>6</b>	<b>Model Creation Support</b>	<b>28</b>
6.1	Model Access and Initialisation . . . . .	28
6.2	Creating a Part . . . . .	28
6.2.1	Simple Predefined Parts . . . . .	28
6.2.2	Parts from Sketches . . . . .	29
6.2.3	Special Pile Parts . . . . .	32
6.3	Constraints with Connectors . . . . .	33
6.4	Soil Body . . . . .	35
6.4.1	Material Assignment . . . . .	37
6.4.2	Soil Body Creation Function . . . . .	38
6.4.3	Soil Stress . . . . .	40
<b>7</b>	<b>Setting and Transferring States</b>	<b>41</b>
7.1	Constant Initial Values for Solution Dependent Variables . . . . .	41
7.2	Transfer Values from an Output Database . . . . .	42
7.3	Assign Arbitrary Initial Values . . . . .	43
<b>8</b>	<b>Querying Meshed Element Data</b>	<b>46</b>
8.1	Volume of Elements . . . . .	46
8.2	Element Containing a Certain Point . . . . .	47
8.3	Weighting of Element Nodes on a Point . . . . .	48
<b>9</b>	<b>Output Processing</b>	<b>49</b>
9.1	Select Output Variables . . . . .	49
9.2	Plot Output . . . . .	49
9.3	Process Output Data . . . . .	52
<b>10</b>	<b>Model Creation Template</b>	<b>56</b>
10.1	Imports . . . . .	56
10.2	Parameter Definitions . . . . .	57
10.3	Material and Contact Processing . . . . .	60
10.4	Parts . . . . .	61
10.5	Assembly and Contact Applications . . . . .	62
10.6	Initial and Boundary Conditions . . . . .	62
10.7	Steps . . . . .	64
10.8	Postprocessing . . . . .	65
10.9	Job . . . . .	65
10.10	Keyword Adjustments . . . . .	66

<b>11 Closing Remark on a Graphical User Interface to Create Scripts</b>	<b>68</b>
--	-----------

# 1 Introduction

This document focuses on the Abaqus-Python library named `abapys`<sup>1</sup> and its usage. `abapys` was created to support and ease controlling of many Abaqus operations with Python, especially automation of output processing (odb-files) and model creation. Since it was developed alongside work at the Institute of Geotechnical Engineering at the Hamburg University of Technology, its focus is on soil mechanics and soil-structure interaction problems. With the current version of `abapys` it is easier to do tasks such as create simple geometric bodies and layered soil models, select elements or geometries based on arbitrary functions, transfer output states from one model to another and create or save plots and output values. Many functions encourage a more optimized scripting workflow, e. g. for model parametrization or postprocessing big parameter studies.

To get the most out of this documentation and understand how to use this library, basic knowledge in working with Abaqus and basic programming knowledge in Python is assumed. If you have no or very little knowledge about Python there are some very good tutorials available online. You may also want to have a look at the official Python homepage »<https://www.python.org/>. There are other introductions to scripting in Abaqus available online, but basic Python knowledge should suffice to understand the remainder of this document.

*hint » Abaqus uses Python 2 internally. Although compatibility with Python 3 has been a goal for `abapys` and the following example, some Python 2 specific code might still be used.*

*hint » In this document, all hints are highlighted like this.*

If you have no or very little knowledge about Abaqus, you may want to have a look at some tutorials (and the resulting Python code) to get started. In addition to many other tutorials provided in the web, a introductory tutorial named "Pile jacking example in Abaqus" is recommended.<sup>2</sup>

Interested readers may also want to look at the official »Abaqus Scripting User's Guide« from Dassault Systèmes. Especially for german speaking readers it is also recommended to have

---

<sup>1</sup>Composed of ABAqus PYthon Scripting. There is another project named `abapy` which has a similar scope. This library is not related to that project in any way.

<sup>2</sup>HTML-version: »<https://d-zo.github.io/abapys/pilejackingexample.html> (for a pdf-version change the suffix to .pdf).

a look at the abapys function documentation, e. g. at »<https://d-zo.github.io/abapys/abapys.html>.<sup>3</sup> Documentation and/or source code should always be consulted when questions about an abapys function and/or its function arguments need to be clarified.

**hint »** *The abapys function documentation should also be accompanying the abapys Python library.*

This document provides a short overview of scripting in Abaqus in general (chapter 2) and the basics to include and work with abapys (chapter 3). Afterwards the most relevant functions are discussed by topic: Chapter 4 deals with viewport manipulation and chapter 5 with selections. Functions and code snippets for model creation are presented in chapter 6. The transfer of output data to new models and the initialisation of states is discussed thereafter (chapter 7). Chapter 8 introduces some possibilities to query data from models. Chapter 9 is about processing and visualizing output database results. Although using abapys can only contribute to some extent when creating a full model creation script, a possible and general Python template for model creation is presented in chapter 10.

*Author's remark:*

I am grateful for all constructive feedback to this document, especially to my colleague Francisco.

---

<sup>3</sup>Currently abapys and its documentation are written in german.

## 2 Scripting in Abaqus

### 2.1 Scripting Resources in Abaqus

When working in Abaqus/CAE, almost all actions are logged in a replay-file (`abaqus.rpy`). Additionally, when a model file `<model>.cae` is saved, all actions relevant to the model are saved in an accompanying journal file `<model>.jnl`. Both files contain Python commands and can be run as script (partially or whole).<sup>1</sup>

*hint » A short note on notation: Text within pointy brackets like `.cae` resembles a placeholder. So a model with the name `cpt` would replace the placeholder to yield `cpt.cae`.*

In this document, scripts for model creation and for processing output databases will be differentiated since different dependencies (imports) are necessary. In general, when creating a custom script, it is recommended to start with a reference to the file encoding and an import section. Usually, only some of the imports shown in code 2.1 and 2.2 are needed to load all required functions, access modules and internal constants for model creation/manipulation. But there is usually no harm in loading all dependencies.

After loading the corresponding modules, all required dependencies, internal states and data should be available (and modifiable unless restricted by Abaqus). A comprehensive reference for all Abaqus-specific commands can be found in the official »Abaqus Scripting Reference Guide«.

In Abaqus, the internal data is saved in a nested structure called the Abaqus object model. A small excerpt of the structure is given by the following figure 2.1, but more in-depth knowledge can e.g. be found in the official »Abaqus Scripting User's Guide«.

---

<sup>1</sup>This statement has to be constrained for at least two reasons: Selection in Abaqus is depending on the amount/order of created elements and therefore the command has to be adjusted to be reliable. And since the exact same behaviour (and errors) will be reproduced when run again, some commands should also be adjusted or removed for a successful and uninterrupted execution.

**Code 2.1:** Imports for working with a model database.

```

1 # -*- coding: utf-8 -*-
2 from part import *
3 from material import *
4 from section import *
5 from assembly import *
6 from step import *
7 from interaction import *
8 from load import *
9 from mesh import *
10 from optimization import *
11 from job import *
12 from sketch import *
13 from visualization import *
14 from connectorBehavior import *

```

**Code 2.2:** Imports for working with an output database.

```

1 # -*- coding: utf-8 -*-
2 from odbAccess import *
3 from odbMaterial import *
4 from odbSection import *
5 from abaqusConstants import *
6 from visualization import *
7 import displayGroupOdbToolset as myodbtoolset

```

## 2.2 Accessing Objects in Abaqus

To query an object or its structure/children in Abaqus/CAE, simply type

```
1 print <object>
```

in the Abaqus command line at the bottom of the Abaqus/CAE window and press Enter. Child objects can be accessed by their parent name, a dot and their name. Elements of a list can be accessed by their index in square brackets. Members of a repository can be listed with the `keys()`-command and accessed by their name in square brackets. If the existence of a member of an repository object is not known, it can be checked with an `has_key()`-command (see e. g. code 2.3). Abaqus also has an auto-completion feature which allows browsing through members of the current repository or function arguments by pressing TAB.

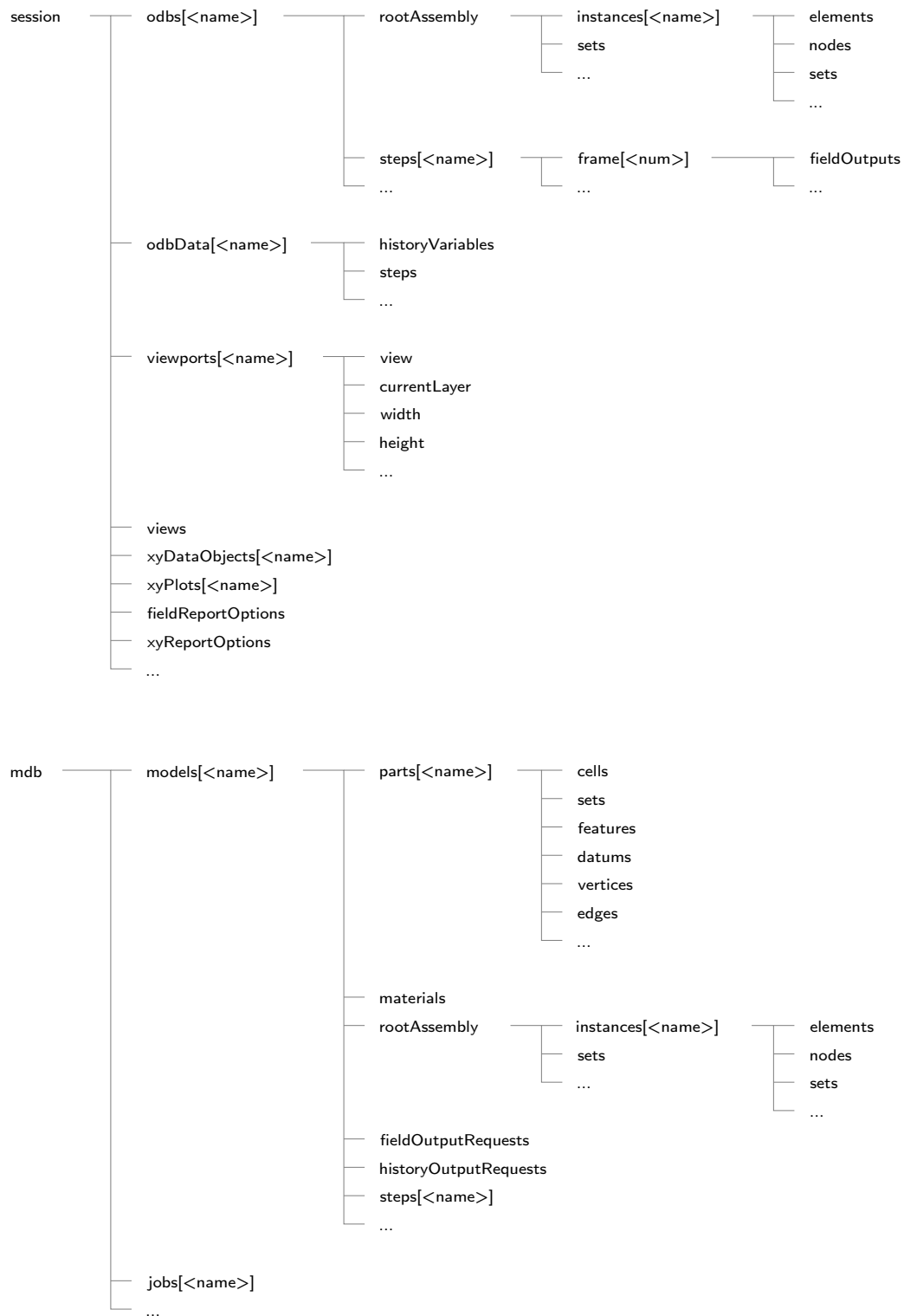
**Code 2.3:** Example commands to access the Abaqus object model.

```

1 print session.spectrums          # Show all available spectrum objects of this session
2 print session.spectrums.keys()   # Since spectrums is a Repository (object) of spectrum objects,
3                                 # the keys()-function is better suited to list all members
4 firstspectrum = session.spectrums.keys()[0] # Save the name of the first spectrum in a variable
5                                           # The result of keys() is always a list
6 print session.spectrums[firstspectrum]    # Use this variable to access the first spectrum
7                                           # without the need to use the following equivalent
8 print session.spectrums[session.spectrums.keys()[0]]
9
10 if (session.spectrums.has_key('Black to white')):
11     print('Yes');
12 else:
13     print('No');

```





**Figure 2.1:** Excerpt from the Abaqus object model structure.

## 2.3 Useful Initial Commands when Accessing the Abaqus Object Model

When processing models or postprocessing output databases, familiarity with the Abaqus object model is extremely helpful to get good results. Although the availability of some objects is problem dependent, some general commands are used more frequently than others, e.g. when accessing the current viewport, model or output database. Four basic commands to work with the Abaqus object model are shown in code 2.4.

**Code 2.4:** Useful commands when interacting with the Abaqus object model.

```
1 # Make a reference for the current (active) viewport
2 myviewport = session.viewports[session.currentViewportName];
3
4 # Print the currently displayed object (its name can be accessed by appending ".name")
5 print myviewport.displayedObject
6
7 # List of all available models
8 print mdb.models.keys();
9
10 # List of all available output databases
11 print session.odbs.keys();
```

*hint » To shorten subsequent access commands, variables can be assigned to create a reference to an object.*

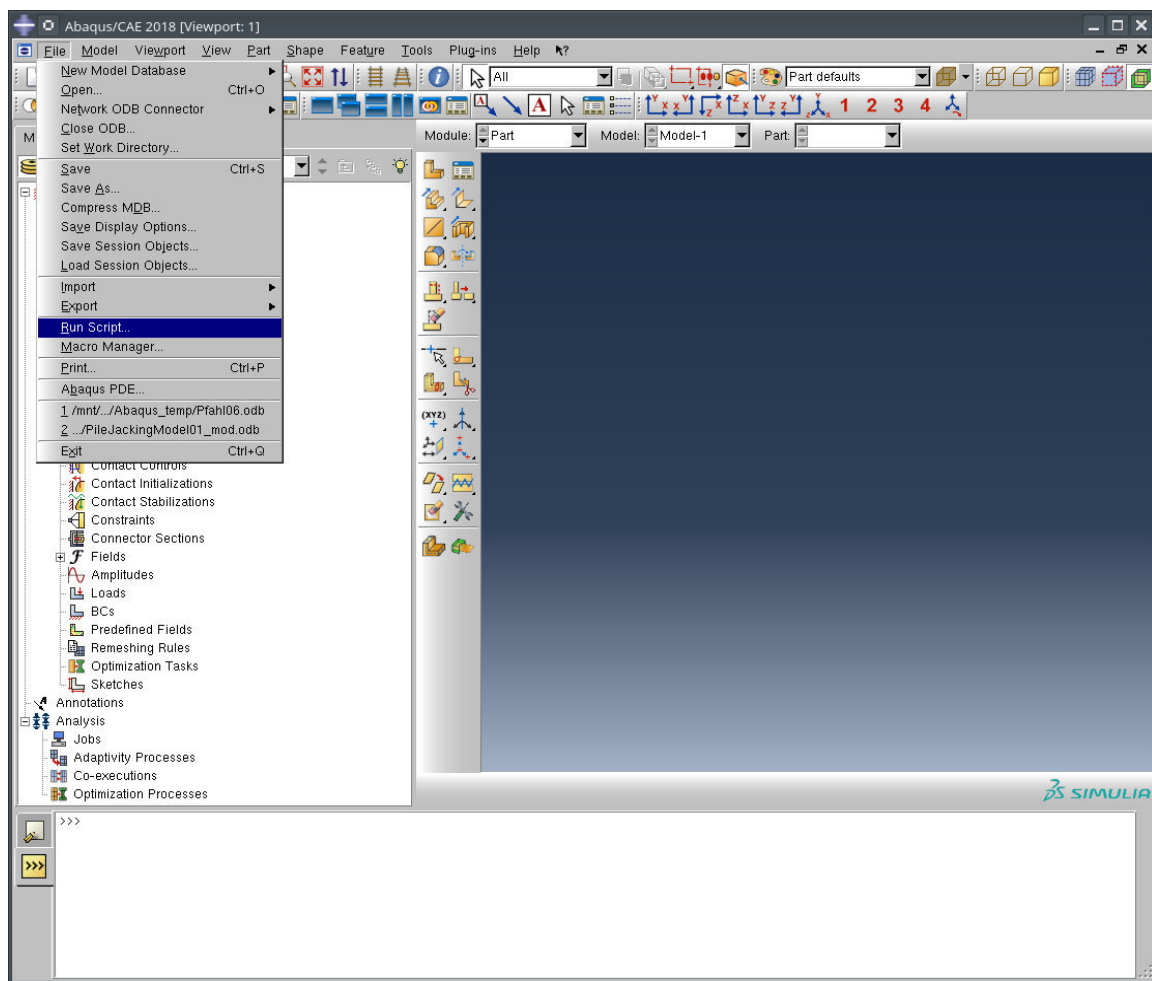
When an output database is opened, results are available in `session.odbs[<name>]` while some structural data is only available in `session.odbData[<name>]` (see also code 2.5).

**Code 2.5:** Commands to access output information and data.

```
1 stepnames = session.odbData[<name>].steps.keys();           # Name of steps
2 histnames = session.odbData[<name>].historyVariables.keys(); # Name of HistoryOutput variables
3
4 # Unlike HistoryOutputs, FieldOutputs are only available in requested frames
5 fieldnames = session.odbs[<name>].steps[stepnames[0]].frames[0].fieldOutputs;
```

## 2.4 Saving and Running a Script in Abaqus

Whenever a sequence of commands should be reused as a script, those commands can be pasted in a new text document (after the appropriate import header as described in section 2.1). The new script can be saved as `<filename>.py` since the ending `py` refers to a Python (script) file.



**Figure 2.2:** Abaqus main window with Python scripting console (>>>-tab in bottom area). To run a script, select File->Run Script as shown.

To run a script in Abaqus, choose File->Run Script from the main menu and select the Python script file (see figure 2.2). It is also possible to run a script without the Graphical User Interface (GUI). This might be handy when an input file should be generated or results should be read/saved without the need for examination or user interaction. To do so, open a shell (Linux) or command window (Windows), navigate to the desired working directory and execute the following command:<sup>2</sup>

```
abaqus cae noGUI=<scriptname.py>
```

<sup>2</sup>It might be necessary to adjust the path/command first (e.g. adding .exe and/or navigating to the right folder).

## 3 Working with abapys

### 3.1 How to Include abapys in a Script

First, the abapys library has to be available on the system. Since Abaqus uses its internal Python interpreter, it is sufficient to download the abapys archive, extract the files and put them in any folder accessible by Abaqus. One suitable folder is the temporary work directory of Abaqus.

In the Abaqus command line or any script using abapys, code 3.1 can be used to import the abapys functions and initialise them.

*hint » It is recommended to use InitialisiereAbapys() in every script after abapys import. Since some settings and commands changed for different Abaqus versions, abapys needs to know which version is used to achieve the intended results. If this function is not called, some functions might not behave as expected.*

**Code 3.1:** Initialisation of abapys.

```
1 abapys_dir = r'<X:\path\to\abapys>'; # Adjust path (Windows, starts with r before string) or
2 #abapys_dir = '</path/to/abapys>'; # adjust path (Linux)
3 sys.path.insert(0, abapys_dir);
4
5 from abapys import *
6 InitialisiereAbapys(session=session, version=version, pfad=abapys_dir);
```

InitialisiereAbapys() is tuning internal parameters of abapys with respect to the operating system and the Abaqus version (using the session and version variables provided by Abaqus). It has an additional pfad argument which should point to the directory of abapys<sup>1</sup> (two optional scaling arguments for InitialisiereAbapys() are introduced in section 4.2).

To get an overview of all available (default) arguments and detailed function information, also have a look at the abapys function documentation (in german). The most recent version is stored with the abapys module or available online at »<https://d-zo.github.io/abapys/abapys.html>. It is also possible to get information about any abapys function in Abaqus after the module

---

<sup>1</sup>Essentially this should be the directory containing the files gewichtung.dll, gewichtung.so and Materialdatenbank\_20####.xlsx.

is loaded. To query the doc-string of any abapys-function, enter the following command in the Abaqus command:

```
1 print <function_name>.__doc__
```

## 3.2 Overview of abapys Functions

Currently the abapys module provides more than 90 functions for model creation, output processing or both. Some of them might only be useful for very special purposes, others are needed in most scripts. The following list provides some categorization with the most important functions. All of the listed functions are covered in this document.

*Multi-purpose or auxiliary functions:*

- Initialise all scripts with: `InitialisiereAbapys()`
- Viewport adjustment: `ViewportGroesseAendern()` and `Modellansicht()`
- Viewport saving: `BildSpeichern()`, `MultiBildSpeichern()`, `VideobilderSpeichern()` and `MultiVideobilderSpeichern()`
- Element or geometry selection: `LabelAuswahl()`, `ElementAuswahl()`, `BedingteAuswahl()` and `ZweifachbedingteKantenAuswahl()`
- Positioning/Coloring: `PunktInElement()` and `BauteileEinfarben()`

*Functions to aid model creation:*

- Creation of simple bodies/parts: `Quader()`, `Zylinder()` and `Kugel()`
- Creation of special, more complex parts: `Bohrprofil_VVBP()`, `Bohrprofil_SOBP()` and `Bohrprofil_VBP()`
- Creation of a special soil body with simple geometry: `Boden()`
- Draw sketches: `Linie()`, `Linienzug()`, `Rechteck()`, `Kreis()`, `KreisbogenWinkel()` and `KreisbogenPunkte()`
- Soil pressure/materials: `BodenspannungErstellen()`, `BodenspannungDirektZuweisen()` and `BodenmaterialUndSectionErstellen()`
- Coupling conditions: `ReferenzpunktErstellenUndKoppeln()`, `AxiallagerErstellen()` and `DrehscharnierErstellen()`
- Transferring state information: `Zustandsuebertragung()` and `Knotentransformation()`

*Functions for output processing:*

- Print available steps/frames: `LetzterOutputStepUndFrame()`
- Plot creation: `PlotOutput()` and `PlotXYDaten()`
- Data processing: `SkalarenFieldOutputErstellen()`

- Save results (data files): `XYDatenSpeichern()` and `FieldOutputSpeichern()`

## **Note on Adjusting abapys Functions**

Abaqus loads all internal modules at startup and all other Python modules on their first import. If any changes are made to the files afterwards, this won't have any effect until the modules are actually read again (i. e. by restarting Abaqus). This does not only apply to abapys but to all module imports (and therefore all custom functions and used libraries).

## 4 Viewport Manipulation

### 4.1 Opening an Output Database

First load a model or an output database into your current session and display it. This can be done manually by File -> Open or in code. For a output database named *odbname* the code is simply `session.openOdb(name=odbname)`.

For many scripting scenarios it is recommended to check if the database is already open before trying to open it again (see code 4.1). It is useful to assign the resulting object to a variable and display it in the viewport afterwards (line 10).

**Code 4.1:** Accessing an output database.

```
1 odbname = <odbname>;                                # Adjust full name/path
2
3 if session.odbData.has_key(odbname):
4     myodb = session.odbs[odbname];
5 else:
6     myodb = session.openOdb(name=odbname);
7 #
8
9 myviewport = session.viewports[session.currentViewportName];
10 myviewport.setValues(displayedObject=myodb, displayMode=SINGLE);
```

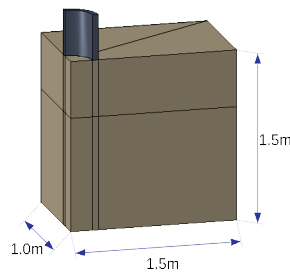
To show the effect of some commands, a simple model is used throughout this tutorial. The model consists of a quarter of an open-ended pile which is positioned above a soil volume (cf. figure 4.1). The soil instance is called *instBoden* and the pile instance is called *instPfahl*. There is nothing special about this model—any model can be used.

### 4.2 Adjusting the Viewport Size

The new viewport size in terms of pixels can be specified by `ViewportGroesseAendern()` with the required arguments `viewport` and the new size `bildgroesse`.

**Code 4.2:** Changing the viewport size.

```
1 ViewportGroesseAendern(viewport=myviewport, bildgroesse=[800, 800]);
```



**Figure 4.1:** Model used in multiple examples. A quarter of a cylindrical shell is positioned above a soil volume. The partitions of the soil volume and its dimension can also be seen.

*hint »* `ViewportGroesseAendern()` is a typical function which may yield different results with different Abaqus versions. Remember to always use `InitialisiereAbapys()` before any other `abapys` function calls to prevent this. It is strongly recommended to adjust code 3.1 to your needs and always use it before calling any other `abapys` function.

According the official documentation, Abaqus' internal values represent the width/height of the viewport in millimeters. With a density of 96 dpi (and 25.4 mm/inch), a factor can be calculated for transforming the internal length values into pixel values (and vice versa). But depending on the environment Abaqus is run in, Abaqus seems to use additional scaling factors for the resulting width/height. Therefore, if the output images do not match the intended pixel size in the current session, try passing two custom correction factors to the optional arguments `xSkalierung` and `ySkalierung` of `InitialisiereAbapys()`.

## Getting the Custom Correction Factors


One way to get the correction factors for `xSkalierung` and `ySkalierung` is to start Abaqus and restore the viewport, so it has a border and title bar (e.g. by `myviewport.restore()`). Then the current size can be obtained by

**Code 4.3:** Querying the current dimension of the viewport.

```
1 print([myviewport.currentWidth, myviewport.currentHeight])
```

Go to File->Print, select a rasterized file output like png and make sure to have *Show viewport decorations (if visible)* checked. For later output it can be checked/unchecked at will, but for obtaining the correction factors a windowed viewport in Abaqus with viewport



decorations is needed.<sup>1</sup> Either go to the png options () and read the current width and height in pixels or save the image to the hard disk and inspect its width and height. Use the Python command interpreter or any calculator to calculate

**Code 4.4:** Obtaining the current scaling factors.

```
1 ppi = 96.0/25.4;
2 xSkalierung = <image_width_in_pixels> / (myviewport.currentWidth*ppi);
3 ySkalierung = <image_height_in_pixels> / (myviewport.currentHeight*ppi);
```

Both factors should be close to 1 (usually between 0.998 and 1.001). The same factors (accurate to about six digits) should be obtainable, if the size of the viewport window is changed graphically.<sup>2</sup>

### 4.3 Saving the Current Viewport as an Image

To save the current viewport, you can either click on **File -> Print** or use the code

**Code 4.5:** Abaqus function for viewport printing.

```
1 session.printToFile(fileName=<dateiname>, format=PNG, canvasObjects=(myviewport, ));
```

Alternatively, the abapys function `BildSpeichern()` can be used. It calls the Abaqus function `session.printToFile()` internally but provides additional possibilities.

**Code 4.6:** Printing the viewport.

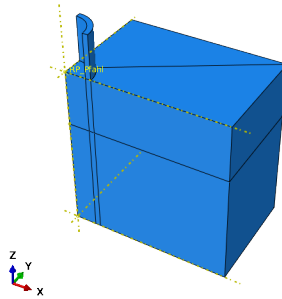
```
1 myviewport = session.viewports[session.currentViewportName];
2 ViewportGroesseAendern(viewport=myviewport, bildgroesse=[600, 600]);
3 BildSpeichern(dateiname='initialisieren', session=session);
```

The function has the following optional arguments with the given default values

- `dateityp='png'` saves the output as a *png* file. Alternatively, the output can be saved as an *eps* file,
- `bildgroesse=[]` allows to specify a viewport size (in pixels) for the image. It might be better to use `ViewportGroesseAendern()` explicitly, so that the actual changes can be seen (and the model view adjusted) before saving an image with `BildSpeichern()` and

<sup>1</sup>The windowed viewport will have a border and titlebar, adding 10 px horizontally and 31 px vertically (which will be subtracted internally).

<sup>2</sup>The advantage of doing it graphically is that the viewport window will be resized in steps of pixels. Therefore the scaling factor can be obtained with a good precision in the described manner. If the size is changed directly (e.g. by `myviewport.setValues(width=..., height=...)`), the accuracy of the scaling factor will be poor, if the values do not represent an exact pixel position.



**Figure 4.2:** After loading the model into the viewport.

- `hintergrund=False` to save the image with a transparent background instead of the viewport background.

The result for a simple model is shown in figure 4.2.

## 4.4 Adjusting the Viewport Style

In the default view, there are a text blocks and various symbols in the viewport. These viewport annotations can either be modified (e. g. larger font, different line width) or deactivated. Either go to the menu (View -> Viewport Annotation Option) or inspect the following structure to change the default settings.

**Code 4.7:** Displaying the viewport annotation options.

```
1 print myviewport.viewportAnnotationOptions
```

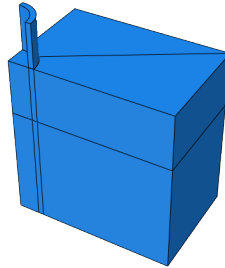
Additionally, the viewport annotation options as well as many other visual settings can be changed with the general purpose beautifying function `ViewportVerschoenern()`. This function applies many subjective improvements and can be influenced by its optional parameters. The basic call is

**Code 4.8:** Applying abapys default viewport settings.





```
1 ViewportVerschoenern(session=session);
```

The function has the following optional arguments with the given default values

- `neuerHintergrund=True` to specify a brighter color gradient as background (if `True`),
- `saubererViewport=True` to remove all previously mentioned viewport annotations (if `True`),



**Figure 4.3:** Adjusted viewport style.

- `saubereLegende=True` to adjust the appearance of the legend (if `True`),
- `evfSchnitt=False` to create a view cut on `EVF_VOID` if an Eulerian element is present (if `True`),
- `minimaleKanten=True` to change the visible edges to `FEATURE` (if `True`),
- `diskreteFarben=True` to have 10 discrete colors (if `True`) or a continuous color change (if `False`),
- `farbspektrum='Viridis'` changes the spectrum to one of the following internally defined spectra:
  - Viridis (default)  » <https://bids.github.io/colormap/>
  - CubeHelix  » <http://www.mrao.cam.ac.uk/~dag/CUBEHELIX/>
  - Moreland  » <https://www.kennethmoreland.com/color-maps/>
  - UniformRainbow  » <https://colorcet.com> defined as *CET-R2*
  - One of the above with `INV` at the end for the inverse of the spectrum (e. g. `VidirisINV`)
- `ausgabeVerschmieren=True` to switch off averaging the outputs (if `False`),
- `zeigeMarkierungen=False` to switch off the visibility of point elements (if `False`),
- `exportiereHintergrund=False` to switch off the background visibility for saving images of the viewport (if `False`) and
- `Standardansicht=False` to adjust the view on a displayed object to a given position, angle and zoom (if `True`).

Calling this function and using an additional command results in figure 4.3.

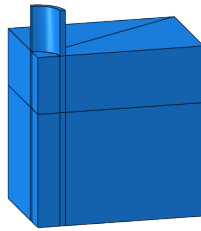


Figure 4.4: Adjusted model view.

## 4.5 Adjusting the Model View

The view on the model can be adjusted by different built-in views like `Top`, `Right` and `Front` as well as rotating, zooming and panning (cf. figure 4.4). Adjusting the basic view orientation can also be achieved by using `Modellansicht()`.

This function accepts a default view name passed to the `ansicht` argument. Even more control is possible by explicitly defining which axis is pointing to the right/top by using the arguments `rechts` and `oben` respectively. The principal axes are defined as 1 ( $x$ ), 2 ( $y$ ) and 3 ( $z$ ). So if the  $x$ -axis should point to the right and the  $z$ -axis to the top, `rechts=1` and `oben=3` can be used as shown in the following code (negative numbers can be used to have an axis point to the left or bottom respectively).

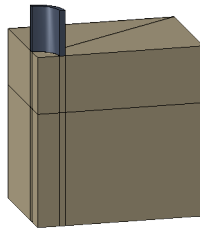
Afterwards, further viewport manipulation can be done with `view.rotate()`, `view.zoom()` and `view.pan()` (also shown in code 4.9).

Code 4.9: Adjusting the view on the model.

```
1 myviewport.view.setValues(projection=PARALLEL);          # Use orthographic instead of perspective view
2 Modellansicht(session=session, rechts=1, oben=3);
3 myviewport.view.rotate(xAngle=15, yAngle=0, zAngle=15, mode=MODEL);
4 myviewport.view.zoom(zoomFactor=0.9, mode=ABSOLUTE);
5 myviewport.view.pan(xFraction=0.0, yFraction=0.1);
```

## 4.6 Polishing a Viewport for Presentation

Some situations might benefit from more polished viewports, i. e. for presenting models. One simple method to improve the visualization of the model is to assign a different color to distinct instances/sets (cf. figure 4.5). For this example, the two instances *instPfahl* and



**Figure 4.5:** Colored model.

*instBoden* of a model database are adjusted (the instances and the assigned colors have to be passed as a list to the argument *zuweisungen*).

**Code 4.10:** Colorizing instances of the model.

```
1 zuweisungen = [['instPfahl', '#495366'], ['instBoden', '#A79B80']];
2 BauteileEinfarben(viewport=myviewport, zuweisungen=zuweisungen, odb=False);
```

The function has the following optional arguments with the given default values

- `standardfarbe='#000000'` as the default color (black) to assign to all not instances/sets not mentioned and
- `odb=True` as a necessary switch to hint if the model belongs to an output database (if True) or a model database (False).

Internally the color definitions of `viewport.colorMappings['Set']` (if working with a model database/*mdb*) or `viewport.colorMappings['Internal Set']` (if working with an output database/*odb*) respectively are modified with the Abaqus function `viewport.setColor()`.

Also, measurements can be added to the viewport as annotations with `ViewportBemassung()`. This function requires either the model database object *mdb* or the the output database user data `session.odbs[<name>].userData` passed to the argument *db* and the current viewport object. The measurement itself has to have a label (*bezeichnung*) for internal reference and two points (*punkt1* and *punkt2*), between which the distance should be measured. Using a non-zero *offset* in one or two dimensions can improve the visibility of the annotation from different angles.

**Code 4.11:** Adding three measurements to the viewport.

```
1 marker1 = ViewportBemassung(db=mdb, viewport=myviewport, bezeichnung='Bodenlaenge',
2   offset=(0.0, -0.15, -0.15), punkt1=(0.0, 0.0, -1.5), punkt2=(1.5, 0.0, -1.5));
3 marker2 = ViewportBemassung(db=mdb, viewport=myviewport, bezeichnung='Bodenbreite',
```

```

4 offset=(-0.15, 0.0, -0.15), punkt1=(0.0, 0.0, -1.5), punkt2=(0.0, 1.0, -1.5));
5 marker3 = ViewportBemassung(db=mdb, viewport=myviewport, bezeichnung='Bodenhoehe',
6 offset=(0.15, -0.15, 0.0), punkt1=(1.5, 0.0, -1.5), punkt2=(1.5, 0.0, 0.0));

```

The function has the following optional argument with the given default value

- `groesse=180` to specify the font size of the annotation. Only certain numbers (like 120, 140, 180, 240, 280, 320, ...) are supported by the default font.

The resulting model is shown at the beginning of this chapter in figure 4.1.

## 4.7 Additional Image Saving Functions

abapys has three special purpose functions enhancing the functionality of `BildSpeichern()`. They can be used to either save a high resolution image, a video sequence or both.

### 4.7.1 Printing High Resolution Images

There are at least two restrictions when saving the current viewport as a rasterized image: The image dimensions usually depend on the viewport size on screen and the maximum viewport size is restricted by Abaqus. One way to get high resolution images is to save multiple images by zooming and panning the viewport and fuse the seamless tiled images together afterwards. This can be done with `MultiBildSpeichern()` by adjusting the view of the resulting image in the viewport and specifying the number of tiles (images) in horizontal and vertical direction.

**Code 4.12:** Saving the viewport with tiled images.

```

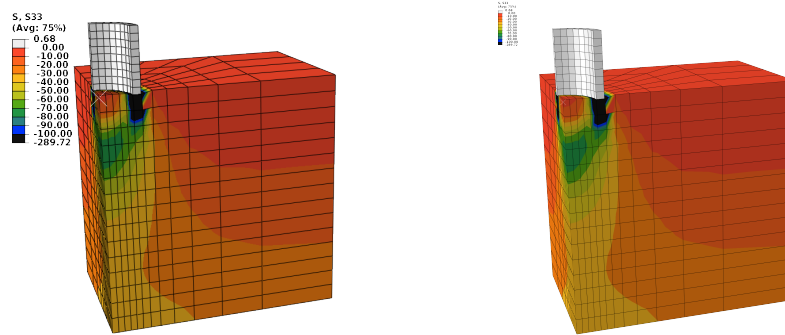
1 MultiBildSpeichern(dateiname='spannung_ende_mul', session=session, anordnung=[3, 3], autozoom=True);

```

This function has the following optional arguments with the given default values

- `anordnung=[1, 1]` to specify the amount of horizontal and vertical (sub-)images,
- `zeitstempel=None` a reference to a user data object to show arbitrary text (e.g. the return value of `myodb.userData.Text()`<sup>3</sup>) or `None`,
- `zeitstempelkanal=0` specifies the (sub-)image to show user data defined in `zeitstempel` (if `zeitstempel` is not `None`). The numbering starts at the top left image and is increased rowwise from left to right,

<sup>3</sup>A possible timestamp could be (assuming `myodb` and `myviewport` to be valid references):  
`zeitstempel = myodb.userData.Text(referencePoint=TOP_RIGHT, anchor=TOP_RIGHT, color='#000000',  
text='2020-01-05', name='date', font='*-liberation sans l-bold-r-normal-*-180-*-p-*-*)',  
offset=(0.0, 0.0)); myviewport.plotAnnotation(annotation=zeitstempel);`



**Figure 4.6:** Saving the viewport with `BildSpeichern()` (left hand side) and `MultiBildSpeichern()` (right hand side). Note that since the images are fused together on the right hand side, the legend is too small and should also be adjusted.

- `legendenkanal=0` if the viewports contains an active legend, it is only shown on the given (sub-)image (same numbering as for `zeitstempelkanal`),
- `mehrfachmodus=False` for display mode `SINGLE` (`True` for display mode `OVERLAY`) and
- `autozoom=False` assumes the view is adjusted to the first (sub-)image and no more zooming is necessary (if `False`) or that the whole visible screen should be used and tiled automatically (if `True`).

The images will be saved rowwise with an appended letter starting from a. Fusing a  $3 \times 3$  grid can be done e. g. with ImageMagick (example from Bash in Linux). The resulting image compared to a normal viewport saving with `BildSpeichern()` can be seen in figure 4.6.

```
convert \( spannung_ende_mula.png spannung_ende_mulb.png spannung_ende_mulc.png +append \( \
  \( spannung_ende_muld.png spannung_ende_mule.png spannung_ende_mulf.png +append \( \
    \( spannung_ende_mulg.png spannung_ende_mulh.png spannung_ende_muli.png +append \( \
      -append spannung_ende.png
```

### 4.7.2 Creating Videos

One of the simplest ways to create a video is by saving multiple (subsequent) images. If an output database is loaded to the current viewport containing multiple result frames, each frame can be saved in increasing order with `VideobilderSpeichern()`.

**Code 4.13:** Saving a sequence of images.

```
1 VideobilderSpeichern(dateiname='video_spannung', session=session, odbname='Pfah103std.odt');
```

This function has the following optional arguments with the given default values

- `dateityp='png'` (same as for `BildSpeichern()`),

- numzahlen=3 defines the number of digits appended to the file name,
- zeitstempel=None (same as for MultiBildSpeichern()),
- erstedateinummer=None can be set to a positive integer to define the starting number of the first saved image and
- startstep=0, startframe=0, stopstep=-1 and stopframe=-1 selects all frames in all available steps as default. The values can be adjusted to specify which range should be saved as image files (first frame in the first step of the selected interval as well as the last frame in the last step).

Additional programs can be used to create a video out of the images, e. g. FFmpeg (example from Bash in Linux).

```
ffmpeg -framerate 10 -f image2 -i video_spannung%03d.png -c:v libx264 -vf "fps=24,format=yuv420p" \
-y video_spannung.mp4;
```

There is also a function called MultiVideobilderSpeichern() combining the features (and some options) of both MultiBildSpeichern() and VideobilderSpeichern().



## 5 Selecting Objects

A common task is selecting objects (like nodes, elements or cells) and either creating a set of this selection (for easier reference later on) or directly applying loads, boundary conditions or other constraints. Whereas selecting objects in the Graphical User Interface (GUI) of Abaqus is straightforward, using the command line is even more powerful but requires some understanding about how Abaqus stores and retrieves objects.

Additionally, (using the default settings) selections in Abaqus are stored in the journal/replay file as masked sequences (commands using `getSequenceFromMask()`). Those masked sequences might not help understanding the selection process, but essentially represent a sequence of all selected objects by their current indices.<sup>1</sup>

### 5.1 Elements and Sequences

Selecting elements usually means creating a sequence of certain elements. Using a simple Python example (cf. code 5.1), *liste* represents a certain object (here a list) and *elem* a single element of that object. In general, *elem* has a different type than *liste* but the sequence *sequenz* has the same type as *liste* and contains a subset of its elements (`liste[0:1]` would only contain *elem*). Sequences share their behaviour with the objects they are derived from.

**Code 5.1:** Extracting elements and sequences.

```
1 liste = ['1', 'second', ['c', 4]];
2 elem = liste[0];
3 sequenz = liste[0:2];
```

Many Abaqus objects support the concatenation of sequences with a simple `+`. Some loops might benefit from using an empty sequence like `liste[0:0]` and iteratively adding other sequences to it. Although this approach seems to work as intended, Abaqus sometimes issues the following warning when empty sequences are used.

```
RuntimeWarning: tp_compare didn't return -1, 0 or 1
```

---

<sup>1</sup>Execute `session.journalOptions.setValues(recoverGeometry=COORDINATE, replayGeometry=COORDINATE)` at startup to log the actual sequences instead of the masked sequences.

## 5.2 Labels and Indices

Most geometric objects have an index and a label represented by integers.<sup>2</sup> The label is a unique identifier for that object while the index represents the current position of that object in the collection of objects of that type.

**hint »** *The index of geometric objects might change when objects of that type are added or removed. (e. g. by partitioning or changing the order of operations in a script). This is the main reason to never use them explicitly for element selection.*

In a model database, geometric elements like nodes and elements have a direct relation between label and index: `label = index + 1`. Since Abaqus is usually rearranging the order of objects defined in the model database to optimize access during the simulation procedure, the indices of geometric objects in the output database are most probably different.

**hint »** *Geometric objects like nodes and elements have the same label in the model database and the output database, but usually a different index.*

It is therefore useful to work with labels rather than indices. To find objects by their label and access them as easily as with indices, the function `ErstelleLabelsortierteGeomlist()` can be used. The function has one required argument `geomliste` which can be any object containing elements with a *label* attribute.

### Example: Selection by Label

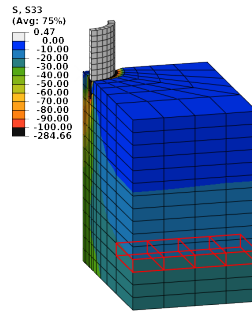
The following examples use a simple model (introduced in section 4.1). It is assumed that the model is loaded and displayed in the viewport. The elements of the model can be accessed by *elemente* and the nodes by *punkte* (similar to code 5.2, also cf. figure 2.1).

**Code 5.2:** Accessing nodes and elements.

```
1 instBoden = session.odbs[<odbname>].rootAssembly.instances[<instname>];
2 punkte = instBoden.nodes;
3 elemente = instBoden.elements;
4 labelliste = [265, 266, 267, 268]; # Some (arbitrary) element labels used in the selection example
```

Additionally, it is assumed that the labels of the elements to be selected are stored in a variable named *labelliste*. A straightforward approach to select elements by their label is to iterate over all elements. In code 5.3 all "selected" elements will be visualized with the `highlight()` function (a simple visualisation can be seen in figure 5.1). For quick checks, element information can be queried by using `Tools -> Query` and choosing `Element`.

<sup>2</sup>Some objects like odb result data have `nodeLabel` or an `elementLabel` attributes (instead of a normal label).



**Figure 5.1:** Selected elements of the example model with code presented in section 5.2.

**Code 5.3:** Highlighting elements by their label.

```
1 for elem in elemente:
2     if (elem.label in labelliste):
3         highlight(elem);
```

However, each time the code is used in a script/command line, it will iterate over all the elements (again) and become inefficient. A better approach is using the function `ErstelleLabelsortierteGeomlist()` introduced in the last section. Only one iteration over all elements is needed to create a dictionary (labels to indices) of these elements. Afterwards, simply input the requested label to the dictionary to get the index of the corresponding element.

**Code 5.4:** Highlighting elements with a label-index mapping.

```
1 label_zu_idx_elemente = ErstelleLabelsortierteGeomlist(geomliste=elemente);
2
3 for label in labelliste:
4     highlight(elemente[label_zu_idx_elemente[label]]);
```

There is also an abapys function `LabelAuswahl()` which essentially does the same thing. However, when using `LabelAuswahl()` the output of `ErstelleLabelsortierteGeomlist()` can be passed as an optional argument `elementhilfsliste` to be more efficient.

**Code 5.5:** Highlighting elements after selecting them by label first.

```
1 label_zu_idx_elemente = ErstelleLabelsortierteGeomlist(geomliste=elemente);
2 elems = LabelAuswahl(elemente=elemente, labelliste=labelliste,
3     elementhilfsliste=label_zu_idx_elemente);
4 highlight(elems);
```

## 5.3 Position and Coordinates

Usually, a selection is based on a geometric constraint like "beneath a certain height" or "in a certain coordinate range". Thus a selection can be done by extracting coordinates of all applicable objects and check if they satisfy the constraint.

Nodes simply have a *coordinate* attribute, which can be used. (Mesh) elements do not have a coordinate attribute, but a *connectivity* attribute listing references to all the nodes defining them. So by accessing its nodes, the coordinates of elements can also be obtained.

**hint »** *In a model database, the connectivity attribute of an element refers to the indices of the nodes, in an output database the connectivity attribute refers to the labels of the nodes!*

Different from nodes, construction elements created in the part module (*vertices*, *edges*, *faces* and *cells*) have a *pointOn* attribute to retrieve their position. The return value of a *pointOn* only matches the exact position for *vertices*. For all other elements, the return values is just somewhere on the edge/face/within the cell (corners and boundary edges included). Abaqus provides a `findAt()` function to query which geometry an entity at the given location belongs to (a simple example is given in section 6.2.2).

## 5.4 Conditional Selection

One of the most important conditional selection functions is `BedingteAuswahl()`. It needs a collection of objects passed to `elemente` and a condition passed as a logical expression (*bedingung*). Additional variables can be passed as a list to the optional argument `var` and might be referenced in the conditions by `var[0]`, `var[1]`, ...

Only certain keywords are allowed as condition and each single member of `elemente` is referenced as `elem`.<sup>3</sup> Code 5.6 can be used to select all nodes from *punkte* in a given volume  $x < 0.3$ ,  $y < 0.3$  and  $z > -0.25$  (cf. figure 5.2).

**Code 5.6:** Conditional selection of nodes.

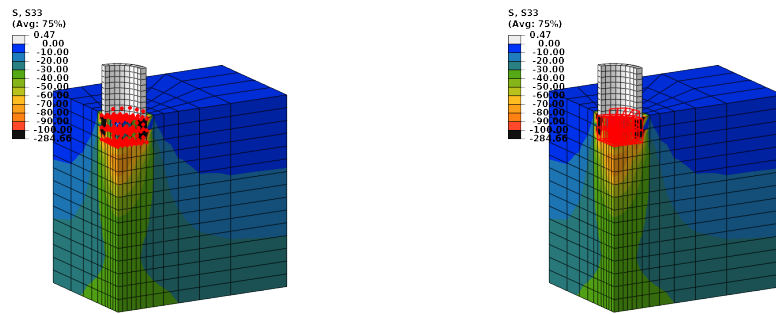
```

1 ausgewaehlte_punkte = BedingteAuswahl(elemente=punkte,
2   bedingung='(elem.coordinates[0] < 0.3) and (elem.coordinates[1] < 0.3) and \
3   (elem.coordinates[2] > -0.25)');
4 highlight(ausgewaehlte_punkte);

```

Although `BedingteAuswahl()` can obtain direct attributes, elements have no coordinates attached to them. Therefore a special function `ElementAuswahl()` can be used to select

<sup>3</sup>`BedingteAuswahl()` also supports a small set of mathematical expressions (if needed): `sqrt()`, `sin()`, `cos()`, `tan()`, `asin()`, `acos()` and `atan()`.



**Figure 5.2:** Conditional selection of a certain volume (here  $x < 0.3$ ,  $y < 0.3$  and  $z > -0.25$ ). Left hand side: selection of nodes with `BedingteAuswahl()`. Right hand side: selection of elements with `ElementAuswahl()`.

elements of a part/instance based on coordinates attached to their corresponding nodes. Consequently, the function needs a list of the nodes for those elements passed to `punktliste`.<sup>4</sup> Elements will be selected only if all nodes of that element fulfill the given condition. The same selection as above but for elements instead of nodes inside the given volume can be done with code 5.7.

**Code 5.7:** Conditional selection of elements by geometric constraints.

```
1 ausgewaehlte_elemente = ElementAuswahl(elemente=elemente, punktliste=punkte,
2   bedingung='(punkt.coordinates[0] < 0.3) and (punkt.coordinates[1] < 0.3) and \
3   (punkt.coordinates[2] > -0.25)');
4 highlight(ausgewaehlte_elemente);
```

The selection of labels as discussed in section 5.2 can also be done with `BedingteAuswahl()` and a suitable condition.

**Code 5.8:** Conditional selection of elements by label.

```
1 elems = BedingteAuswahl(elemente=elemente, bedingung='elem.label in var', var=labelliste);
2 highlight(elems);
```

Another possible use of a variable list is shown in code 5.9 with a tolerance for selection (in this case  $abapys\_tol = 10^{-6}$  is used).

**Code 5.9:** Another conditional selection using variables.

```
1 zellen = BedingteAuswahl(elemente=partBoden.cells, bedingung='(elem.pointOn[0][1] < var[0]+var[1])',
2   var=[sand_hoehe, abapys_tol]);
```

<sup>4</sup>The nodes just have to be in the list, so it is perfectly fine to pass all nodes of the instance to `punktliste`.

## 5.5 Sorting by Direction

Some tasks might require a list of elements (or their labels/indices) sorted by a coordinate direction. By default, elements and other objects are not sorted (in any direction). abapys provides two functions `KnotenAuswahlLabelliste()` and `ElementAuswahlLabelliste()` for sorting labels of nodes or elements respectively based on a given coordinate direction.

Both functions have the following optional arguments with the given default values

- `sortierung=0` defines the axis/coordinate direction (0:  $x$ , 1:  $y$  and 2:  $z$ ) and
- `aufsteigend=True` sorts from lowest to highest (if `True`) or highest to lowest (if `False`).

`KnotenAuswahlLabelliste()` requires a sequence of nodes passed as `knoten` whereas the function `ElementAuswahlLabelliste()` requires elements and nodes (passed as `elemente` and `punktliste` respectively).

In the following example a column of elements is selected with `ElementAuswahl()` and the selected elements are sorted in descending  $z$ -order with `ElementAuswahlLabelliste()` (see also figure 5.3). The result is a list of the element labels `elem_sortiert` sorted by vertical position of their elements.

**Code 5.10:** Selecting elements and sorting by coordinate direction.

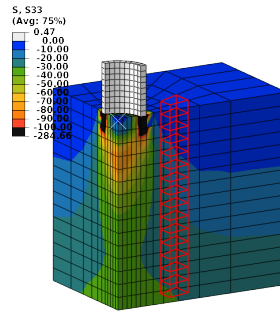
```

1 elems = ElementAuswahl(elemente=elemente, punktliste=punktliste,
2   bedingung='(punkt.coordinates[0] > 0.5) and (punkt.coordinates[0] < 0.75) and ' \
3   '(punkt.coordinates[1] > 0.1) and (punkt.coordinates[1] < 0.4)');
4
5 label_zu_idx_elemente = ErstelleLabelsSortierteGeomlist(geomliste=elemente);
6 elem_sortiert = ElementAuswahlLabelliste(elemente=elems, punktliste=punktliste, sortierung=2,
7   aufsteigend=False);
8 for einzelelem in elem_sortiert:
9     highlight(elemente[label_zu_idx_elemente[einzelelem]]);
10    print(einzelelem);

```

## 5.6 Direction Based Edge Selection

When meshing the model, some areas might need a finer mesh than others. Therefore, it may be useful to define an increasing/decreasing amount of seeds along certain edges. So the selected edges should also be differentiated based on their direction (defined by the order of the end nodes). abapys provides a function called `ZweifachbedingteKantenAuswahl()` to select edges and sort them. In addition to `elemente` (which should be a collection of edges) and an optional `var=[]` (same as in `BedingteAuswahl()`) three conditions can be provided:



**Figure 5.3:** Example of a column of elements which might need to be sorted from top to bottom. The labels of the elements selected by `ElementAuswahl()` are in increasing label order by default. `ElementAuswahlLabelliste()` can be used to sort them by their (vertical) position.

- `bedingung1=True` can be used to define a condition for edge selection. Each single edge can be referenced by `edge`, so `edge.pointOn[0][0]` accesses the  $x$ -coordinate assigned to the current edge,
- `bedingung2=True` additionally allows to refine the edge selection by accessing both end nodes with `vert1` and `vert2`, e. g. to compare coordinates and only select vertical edges,
- `bedingung3=True` can be used to check all remaining edges and assign them to the first list of return values (if the condition is met) or to the second list (if the condition is not met). The auxiliary variables `edge`, `vert1` and `vert2` are also available here.

All three conditions can access variables passed in `var` by their index (as the condition in `BedingteAuswahl()`). In the following example all edges along the  $y$ -direction on a part referenced by `partBoden` are selected, if their  $x$  coordinate is more than *entfernung* away from the origin. All selected edges pointing inward ( $x$ -coordinate of the start node is greater than the  $x$ -coordinate of the end node) are returned in the first list, all pointing outward in the second list.

**Code 5.11:** Selecting edges and sorting them by direction.

```

1 kanten_innen, kanten_aussen = ZweifachbedingteKantenAuswahl(elemente=partBoden,
2   bedingung1='(edge.pointOn[0][0] > var[0]-var[1])',
3   bedingung2='(abs(vert1.pointOn[0][1]-vert2.pointOn[0][1]) < var[1])',
4   bedingung3='(vert1.pointOn[0][0] > vert2.pointOn[0][0])',
5   var=[entfernung, abapys_tol]);

```

# 6 Model Creation Support

## 6.1 Model Access and Initialisation

Similar to opening and accessing an output database (see section 4.1), model databases can be created and accessed. Accessing an existing model database can be done by entering

**Code 6.1:** Assigning model database to variable.

```
1 mymodel = mdb.models[<modelname>];
```

To create a new model, choose a suitable model name and calling the Abaqus function `mdb.Model()`. Afterwards, the model can be referenced by its name as shown in code 6.2.

**Code 6.2:** Create and access a new model database.

```
1 modelname = '<modelname>';  
2 mdb.Model(name=modelname, modelType=STANDARD_EXPLICIT);  
3 mymodel = mdb.models[modelname];
```

## 6.2 Creating a Part

abapys provides functions to create simple geometric parts, (screw) piles and a soil body.

### 6.2.1 Simple Predefined Parts

abapys has three functions to create very simple parts, namely

- a cuboid with `Quader()`,
- a cylinder with `Zylinder()` and
- a sphere with `Kugel()`.

All three functions require a model reference `modell`, a `name` and geometric values like `laenge`, `breite`, `hoehe` and `radius`. They also need a basic mesh size `gittergroesse` and a type of material (usually `materialtyp=DEFORMABLE_BODY`). The functions return a reference on the created part and instance. Three example calls are shown in code 6.3.



**Code 6.3:** Creating simple geometric parts.

```

1 [partRad, instRad] = Zylinder(modell=mymodel, name='Rad', radius=0.6, hoehe=0.3,
2   materialtyp=DEFORMABLE_BODY, gittergroesse=0.2);
3 [partBasis, instBasis] = Quader(modell=mymodel, name='Basis', laenge=4.0, breite=2.0, hoehe=1.5,
4   materialtyp=DEFORMABLE_BODY, gittergroesse=0.2);
5 [partGelenk, instGelenk] = Kugel(modell=mymodel, name='Gelenk', radius=0.3,
6   materialtyp=DEFORMABLE_BODY, gittergroesse=0.2);

```

Quader() has the following optional arguments with the given default values

- rp=False to create a reference point for that part (if True),
- extrasets=False to create more predefined sets (if True) like outer faces normal to  $x/y$ -direction, lateral faces and top/bottom faces,
- xypartition=[True, True] for additional partitions normal to  $x$ -direction and  $y$ -direction (if True respectively) and
- viertel=4 if the whole model should be created (four quarters, i. e. 4), half of it (2) or one quarter (1).

The Zylinder() function has an additional optional argument r\_innen=[]. A filled cylinder will be created by default, but if a value is given to r\_innen (instead of an empty list) a hollow cylinder with a that inner radius is created.

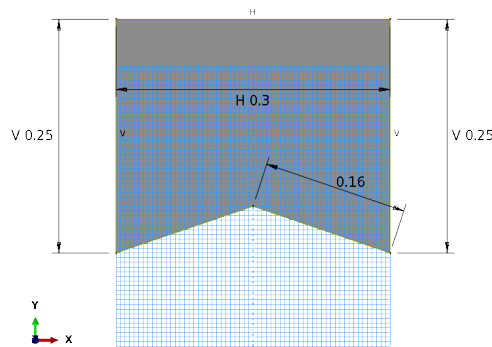
Kugel() has the optional arguments rp=False, extrasets=False and r\_innen=[]. While both first arguments have the same effect es described for Quader(), the last argument is similar to the optional argument of Zylinder() but would create a hollow sphere instead.

## 6.2.2 Parts from Sketches

Usually, it is necessary to create more complicated parts. If a part can be described by a simple drawing/sketch, Abaqus can extrude or revolve a part based on the geometries defined in that sketch.

In code 6.4 the creation of the sketch is done in line 6 and the extrusion to a part in line 14 (see also figure 6.1). Everything in between is defining the sketch geometry. As shown, it might be preferable to define all necessary parameters beforehand (here line 1–4) to build a fully parametrized model.

Although Abaqus has some rather straightforward commands to draw a single straight or curved line segment, abapys provides some improved commands for easier sketching (and the possibility to automatically add measurements): Linie(), Linienzug(), Rechteck(), Kreis(), KreisbogenWinkel() and KreisbogenPunkte(). They can be combined in any meaningful way to create a closed sketch.



**Figure 6.1:** The sketch resulting from code 6.4.

**Code 6.4:** Creating a part based on a custom sketch.

```

1  laenge = 0.4; # [m]
2  breite = 0.3; # [m]
3  hoehe = 0.25; # [m]
4  h_aussparung = 0.05; # [m]
5
6  zeichnung = mymodel.ConstrainedSketch(name='_profil_', sheetSize=max(laenge, breite));
7  Linienzug(zeichnung=zeichnung, punkte=[
8      (0.0, 0.0),
9      (breite/2.0, -h_aussparung),
10     (breite/2.0, hoehe-h_aussparung),
11     (-breite/2.0, hoehe-h_aussparung),
12     (-breite/2.0, -h_aussparung)], geschlossen=True);
13  partBauteil = mymodel.Part(dimensionality=THREE_D, name='Bauteil', type=DEFORMABLE_BODY);
14
15  partBauteil.BaseSolidExtrude(depth=laenge, sketch=zeichnung);
16  del zeichnung;

```

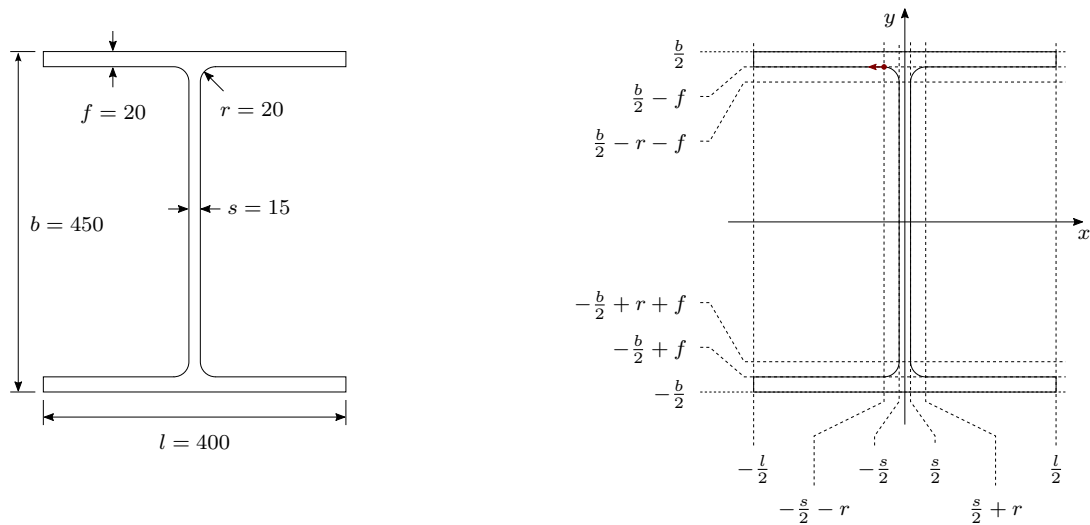
All of abapys' sketch commands have an optional argument `bemasst=True` to add a measurement to the drawing. `Linienzug()` also has an optional argument `geschlossen=False` which can be set to `True` to connect the first given point in `punkte` to the last. `KreisbogenWinkel()` and `KreisbogenPunkte()` require a direction argument for `richtung` which can either be `CLOCKWISE` or `COUNTERCLOCKWISE`. If the sketch should be revolved instead of extruded, the revolution axis has to be included with a fixed constraint (line 1 and 2 of code 6.5) and the extrusion command has to be replaced with a revolution command.

**Code 6.5:** Changes to create a part by revolution instead of extrusion.

```

1  zeichnung.ConstructionLine(point1=(0.0, -1.0), point2=(0.0, 1.0));
2  zeichnung.FixedConstraint(entity=zeichnung.geometry.findAt((0, 0),));
3  # Draw commands for the relevant geometry
4  partBauteil.BaseSolidRevolve(angle=360.0, flipRevolveDirection=OFF, sketch=zeichnung);

```



**Figure 6.2:** Example of an HEA beam. Left hand side: Construction sketch with units in millimetres. Right hand side: Same sketch adjusted for a parametrized model.

*hint » Whenever geometries need to be selected, use the `findAt()` function instead of trying to access objects by their index number. Indices might change, but positions stay the same.*

### Example: Parametrised Sketch

As demonstrated in code 6.4, it is recommended to use variables instead of fixed values within the draw commands. Variables with meaningful names or relatable symbols<sup>1</sup> are easier to understand than numbers and have to be defined only once in a script (e.g. near the beginning). This can be very helpful when dealing with more complex geometries.

As an example consider the HEA beam in figure 6.2 (left hand side), which is still a rather simple geometry. The geometry can be described by five parameters (as shown in the right hand side of the same figure) and code 6.6. It is assumed that the sketch is referenced by *zeichnung* as before.

**Code 6.6:** Parametrised sketch for HEA beam.

```

1 l = 0.4; # [m]
2 b = 0.45; # [m]
3 f = 0.02; # [m]
4 s = 0.015; # [m]
5 r = 0.02; # [m]
```

<sup>1</sup>Make sure that the meaning of each variable is easy to understand, if only single characters and no descriptive names are chosen.

```

6
7 Linienzug(zeichnung=zeichnung, punkte=[
8     (-s/2.0-r,      -b/2.0+f),
9     (-1/2.0,       -b/2.0+f),
10    (-1/2.0,       -b/2.0),
11    (1/2.0,        -b/2.0),
12    (1/2.0,        -b/2.0+f),
13    (s/2.0+r,      -b/2.0+f)]);
14 KreisbogenPunkte(zeichnung=zeichnung,
15     mittelpunkt=(s/2.0+r, -b/2.0+r+f),
16     punkt1=(s/2.0+r, -b/2.0+f),
17     punkt2=(s/2.0, -b/2.0+r+f), richtung=CLOCKWISE);
18 Linie(zeichnung=zeichnung, punkt1=(s/2.0, -b/2.0+r+f),
19     punkt2=(s/2.0, b/2.0-r-f));
20 KreisbogenPunkte(zeichnung=zeichnung,
21     mittelpunkt=(s/2.0+r, b/2.0-r-f),
22     punkt1=(s/2.0, b/2.0-r-f),
23     punkt2=(s/2.0+r, b/2.0-f), richtung=CLOCKWISE);
24 Linienzug(zeichnung=zeichnung, punkte=[
25     (s/2.0+r,      b/2.0-f),
26     (1/2.0,       b/2.0-f),
27     (1/2.0,       b/2.0),
28     (-1/2.0,      b/2.0),
29     (-1/2.0,      b/2.0-f),
30     (-s/2.0-r,    b/2.0-f)]);
31 KreisbogenPunkte(zeichnung=zeichnung,
32     mittelpunkt=(-s/2.0-r, b/2.0-r-f),
33     punkt1=(-s/2.0-r, b/2.0-f),
34     punkt2=(-s/2.0, b/2.0-r-f), richtung=CLOCKWISE);
35 Linie(zeichnung=zeichnung, punkt1=(-s/2.0, b/2.0-r-f),
36     punkt2=(-s/2.0, -b/2.0+r+f));
37 KreisbogenPunkte(zeichnung=zeichnung,
38     mittelpunkt=(-s/2.0-r, -b/2.0+r+f),
39     punkt1=(-s/2.0, -b/2.0+r+f),
40     punkt2=(-s/2.0-r, -b/2.0+f), richtung=CLOCKWISE);

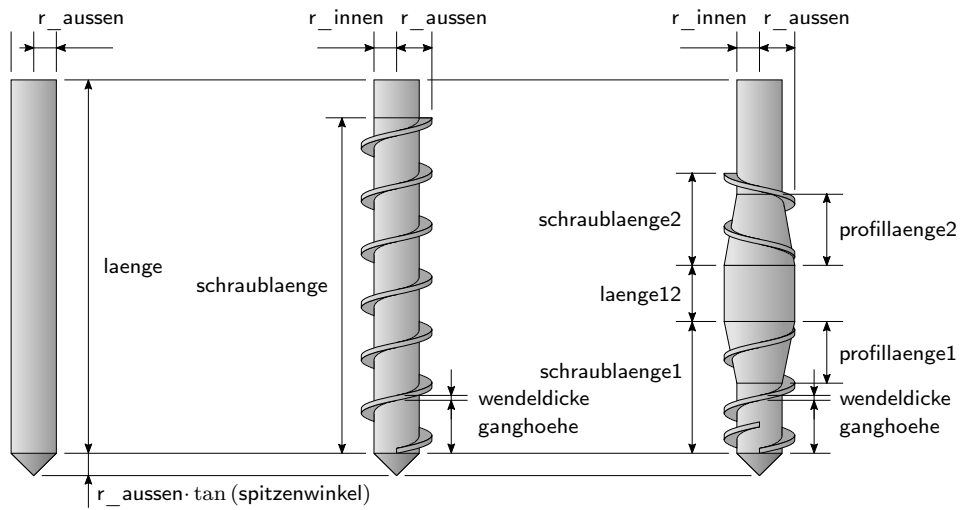
```

### 6.2.3 Special Pile Parts

abapys also includes three functions to create predefined pile parts:

- displacement piles with Bohrprofil\_VBP(),
- screw piles with Bohrprofil\_SOBP() and
- full displacement piles with Bohrprofil\_VVBP().

Like the basic part generation functions described in section 6.2.1, all three functions return a reference on the created part and instance. The (geometric) arguments required for these functions are described visually in figure 6.3 (except *rundwinkel*). Other required arguments (like *modell* and *name*) are the same as for the other basic part generation functions (*gitter\_werkzeug* corresponds to *gittergroesse*).



**Figure 6.3:** The values needed for the pile creation functions. Left hand side: Bohrprofil\_VBP(). Center: Bohrprofil\_SOBP(). Right hand side: Bohrprofil\_VVBP().

**Code 6.7:** Example of creating special pile parts.

```

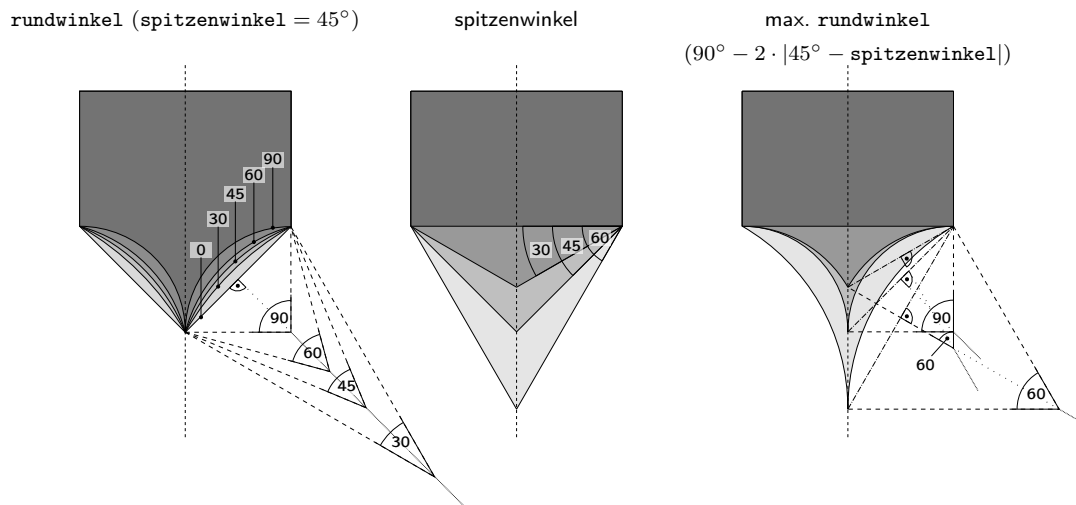
1 [partVBP, instVBP] = Bohrprofil_VBP(modell=mymodel, name='Verdraengungsprofil',
2   laenge=10.0, r_aussen=0.2, spitzenwinkel=45.0, rundwinkel=0.0, gitter_werkzeug=0.1);
3
4 [partSOBP, instSOBP] = Bohrprofil_SOBP(modell=mymodel, name='Schneckenprofil', laenge=10.0,
5   r_aussen=0.2, r_innen=0.12, spitzenwinkel=45.0, rundwinkel=0.0, schraublaenge=7.0, ganghoehe=0.3,
6   wendeldicke=0.05, gitter_werkzeug=0.1);
7
8 [partVVBP, instVVBP] = Bohrprofil_VVBP(modell=mymodel, name='Vollverdraengungsbohrprofil',
9   laenge=10.0, r_aussen=0.2, r_innen=0.12, spitzenwinkel=45.0, rundwinkel=0.0, schraublaenge1=0.45,
10  profillaenge1=0.2, schraublaenge2=0.25, profillaenge2=0.2, laenge12=0.16, ganghoehe=0.16,
11  wendeldicke=0.015, gitter_werkzeug=0.1);

```

Additionally  $0^\circ \leq \text{spitzenwinkel} < 90^\circ$  and  $0^\circ \leq \text{rundwinkel} \leq 90^\circ - 2 \cdot |45^\circ - \text{spitzenwinkel}|$  to have a coherent geometry. The `rundwinkel` argument can be used to create a concave pile tip (see also figure 6.4) or set to zero for a linear slope to the tip (`rundwinkel=0.0`).

## 6.3 Constraints with Connectors

Abaqus provides connectors and constraints to control/restrict the movement of certain instances relative to each other. For example a wheel can be restricted to only rotate around an axis by defining a hinge connector on the wheel with regard to this axis with `DrehscharnierErstellen()`. Therefore, next to a model reference (`modell`) and a name, a



**Figure 6.4:** Effect of angles `spitzenwinkel` and `rundwinkel` on the shape of the pile tip. Left hand side: different `rundwinkel` for `spitzenwinkel` equal to  $45^\circ$ . Center: different `spitzenwinkel` (no tip for  $0^\circ$ ). Right hand side: maximum values of `rundwinkel` for the four values of `spitzenwinkel` shown in the center image.

coordinate system `kos` defining the axis around which rotation is allowed ( $x$ -axis of coordinate system) and two reference points `punkt1` and `punkt2` are needed. Both reference points should be attached to instances/sets rotating relative to each other.

In the following example the reference point of a part/instance called `partBasis/instBasis` and a reference point of the assembly (`RP_Rad`) are only allowed to rotate around the  $x$ -axis of a given coordinate system (called `KoordSystemRotation` and attached to the same part/instance).

**Code 6.8:** Constraining geometry to only rotate around a given axis.

```
1 DrehscharnierErstellen(modell=mymodel, name='Wire00',
2   punkt1=mymodel.rootAssembly.referencePoints[mymodel.rootAssembly.features['RP_Rad'].id],
3   punkt2=instBasis.referencePoints[partBasis.features['RP'].id],
4   kos=instBasis.datums[partBasis.features['KoordSystemRotation'].id]);
```

Similarly a translate connector can be defined to allow only movement along one axis ( $x$ -axis) with `AxiallagerErstellen()`.

**Code 6.9:** Constraining geometry to only move along a given axis.

```
1 AxiallagerErstellen(modell=mymodel, name='WireAxiallager',
2   punkt1=instBasis.referencePoints[partBasis.features['RP'].id],
3   punkt2=instBohrprofil.referencePoints[partBohrprofil.features['RP'].id],
4   kos=instBasis.datums[partBasis.features['KoordSystemRotation'].id]);
```

Note that *RP\_Rad* for *DrehscharnierErstellen()* in code 6.8 is defined in the assembly while the reference point for the example to *AxiallagerErstellen()* (code 6.9) is defined in the part itself (both options are available for both functions/arguments).

To define new reference points in the assembly and fix them to any geometry, the function *ReferenzpunktErstellenUndKoppeln()* can be used. It needs the coordinates of the reference point (*punkt*), its name and geometry to connect to (*flaeche*).

**Code 6.10:** Creating and coupling a reference point.

```
1 ReferenzpunktErstellenUndKoppeln(modell=mymodel, punkt=(1.5, 1.0, 0.0), name='Rad',
2   flaeche=instRad.sets['setAll']);
```

In general, the Abaqus commands to simply create a reference point or coordinate system are similar to the following.

**Code 6.11:** Creating a reference point and a coordinate system.

```
1 partBasis.ReferencePoint(point=(0.0, 0.0, 0.0));
2 partBasis.DatumCsysByThreePoints(coordSysType=CARTESIAN, name='KoordSystemRotation',
3   origin=partBasis.referencePoints[partBasis.features['RP'].id],
4   point1=(0.0, 1.0, 0.0), point2=(1.0, 0.0, 0.0));
```

## 6.4 Soil Body

Most geotechnical simulations are done to investigate a soil-structure interaction. Therefore, a soil body with the most relevant properties has to be modelled at some point. With *abapys* it is easy to create a simple (cuboid or cylindrical) soil body with different layers of materials and automatically calculate and assign the initial earth pressure for each layer. To achieve a good result, it is necessary to have all used materials present in the material database (the *Materialdatenbank\_20####.xlsx*-file in the *abapys* directory) and properly define the soil body geometry.

The main function to create a soil body is introduced in section 6.4.2 but most relevant parameters are already listed in code 6.12 (see also figure 6.5) and introduced here:

- *bodentiefe* is the height of the soil body. The soil instance will be translated in the assembly so that the top is located at a vertical position of 0 and the bottom at *-bodentiefe* in *z*-direction,
- *voidhoehe* is only used if the soil body is an Eulerian volume. In that case, the height of the soil body will be increased by this additional height. The top of an Eulerian volume will be at a vertical position of *+voidhoehe* (the position of all soil layers and the bottom of the soil body are not influenced by this argument),

- `gitter_vertikal` the average height of an element (mesh size) for the soil volume,
- `bodenbereich` contains lists with one or two entries (with decreasing values) describing the soil structure in top view (cf. right hand side of figure 6.5 and line 13 of code 6.12). The first list represents either the radius of the soil volume cylinder (if it contains one value) or half of the width and half of the height of the soil volume cuboid (if it contains two values). All other lists represent increasingly smaller partitions (circular for one value and rectangular for two values) structuring the part,
- `gittergroessen` contains mesh size recommendations for the structure described in `bodenbereich`. Therefore, the amount of lists have to be the same in both. Lists may contain one value for a constant mesh size or two values for a linear increasing/decreasing mesh size within the corresponding partition,
- `schichten` can be used to create horizontal soil layers (cf. left hand side of figure 6.5 and line 21 of code 6.12). Each entry in `schichten` represents the bottom (vertical position) of a soil layer below ground level (positive values in increasing order). The vertical position cannot be greater than `bodentiefe` (but might be smaller),
- `schichtmaterial` assigns a material name to each layer defined in `schichten`. Thus it must have the same amount of entries. The mapping of the material names to the actual materials (also for `restmaterial`) is shown in code 6.13 and
- `restmaterial` is used to fill up the bottom volume if `bodentiefe` is greater than the last entry of `schichten`.

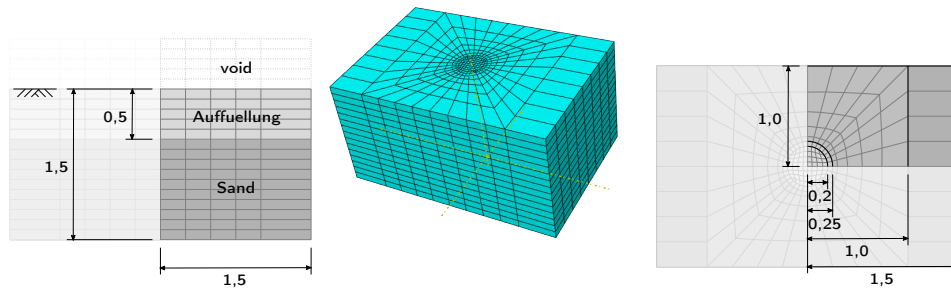
**Code 6.12:** Example of how soil body parameters could be defined.

```

1 bodentiefe, voidhoehe, gitter_vertikal = [
2 #>0          >0          >0
3 #[m]         [m]         [m]
4 #-----|-----|-----|
5 1.500      , 0.500      , 0.1
6 #-----|-----|-----|
7 ];
8
9 bodenbereich, gittergroessen = [
10 #[radius] oder [laenge/2, breite/2]      [konstant] oder [gross, klein]
11 #[m] (aussein->innen)                    [m] (aussein->innen)
12 #-----|-----|-----|
13 [[1.5, 1.0], [1.0, 1.0], [0.25], [0.2]] , [[0.4], [0.4, 0.1], [0.05], [0.05]]
14 #-----|-----|-----|
15 ];
16
17 schichten, schichtmaterial, restmaterial = [
18 #>[0]      >['']      >''
19 #[m]
20 #-----|-----|-----|
21 [0.5, 1.5] , ['Auffuellung', 'Sand'] , 'Sand' #
22 #-----|-----|-----|
23 ];

```





**Figure 6.5:** Left hand side: Side view on soil body. The void volume is only created for an Eulerian soil body and ignored otherwise (like here). Center: Soil body example. Right hand side: Top view on soil body.

### 6.4.1 Material Assignment

All materials used in *schichtmaterial* and *restmaterial* shown in code 6.12 have to be defined and reference to materials in the material database. They link internally chosen names with database entries and take density, saturation with water (concerning density) and used constitutive model into account when calculating the initial earth pressure. Code 6.13 illustrates how two materials can be defined.

**Code 6.13:** Defining two soil materials.

```

1 materialien_boden = [
2 #   Abaqus-Bez.   Datenbankname   Parameter-Bez.   Saettigung   Lagerungsd.   Stoffgesetz
3 #   >'           >'           ''              [0-1]        [0-1]        >'
4 #   |-----|-----|-----|-----|-----|-----|
5   ['Auffuellung', 'Standardparameter', '', 0.0, 0.4, 'Mohr-Coulomb'],
6   ['Sand', 'Hamburger Sand', '', 1.0, 0.5, 'Mohr-Coulomb'],
7 #   |-----|-----|-----|-----|-----|-----|
8 # Die Lagerungsdichte/Verdichtungsgrad kann bestimmt werden aus
9 # a) Anfangsdichte rho_0:   I_r = (rho_0 - rho_min)/(rho_max - rho_min)
10 # b) Anfangsporenzahl e_0: I_e = (e_max - e_0)/(e_max - e_min)
11 ];

```

The definitions in code 6.12 and 6.13 can be used with the parameters from the material database to define materials with `BodenmaterialUndSectionErstellen()` (see code 6.14).

**Code 6.14:** Calling `BodenmaterialUndSectionErstellen()` loads all relevant material parameters.

```

1 verwendeteMaterialien = sorted(set(schichtmaterial + [restmaterial]));
2 benoetigtUseroutine, verwendeteBodenwerte = BodenmaterialUndSectionErstellen(modell=mymodel,
3   verwendeteMaterialien=verwendeteMaterialien, verfuegbareMaterialien=materialien_boden,
4   euler=False);

```

`BodenmaterialUndSectionErstellen()` has the following optional arguments with the given default values

- `userroutine=''` if a user routine is to be used for at least one material, the name of the routine should be provided. Currently only some special (visco-)hypoplastic routines are supported,
- `numDepVar=0` to specify the number of internal state variables of the user routine (if any) and
- `euler=True` if the soil is an Eulerian volume (**True**) or a Lagrangian volume (**False**).

### 6.4.2 Soil Body Creation Function

After defining all parameters, they can be passed to the `Boden()` function to actually create the soil body (see code 6.15). This function also has the following optional arguments:

- `euler=True` to create a Lagrangian soil body (if **False**) or an Eulerian soil body (if **True**),
- `xypartition=[True, True]` to create a partition normal to  $x$ - and  $y$ -direction through the origin (if **True** respectively),
- `partition_durchziehen=False` to cut partitions through the whole part (if **True**) or just within the surrounding neighborhood (if **False**). This can be important for certain cases rectangular partitions (and auxiliary partitions) combined with mesh refinement.
- `viertel=4` if the whole model should be created (four quarters, i. e. 4), half of if (2) or one quarter (1) and
- `rotationsprofilpunkte=None` can be used to pass a reference to a sketch for an axisymmetric volume to subtract from a Lagrangian soil body (this can be used to simulate certain wished-in-place simulations for `euler=False`). This is not needed for an Eulerian soil body (see below).

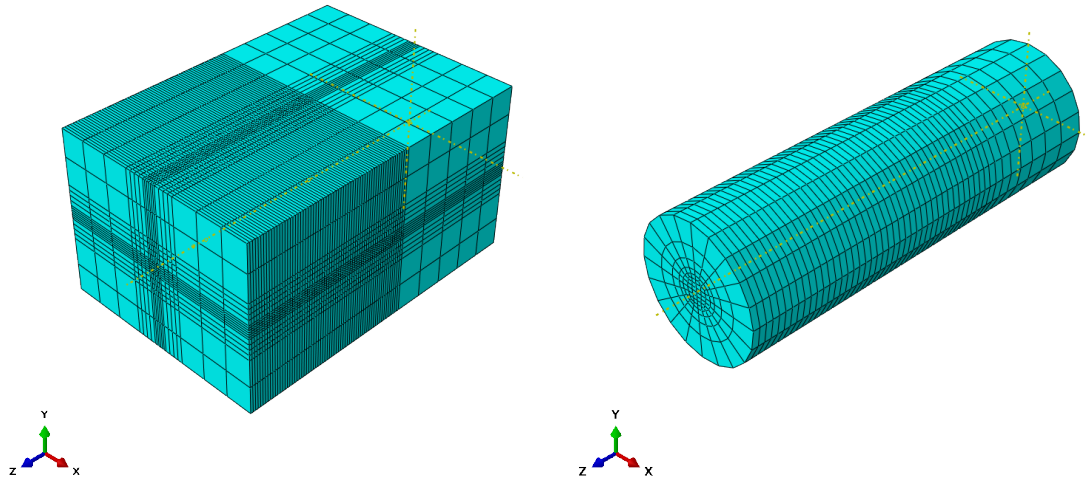
**Code 6.15:** Calling function `Boden()` to create the soil body with custom parameters.

```

1 [partBoden, instBoden] = Boden(modell=mymodel, name='Boden', bodentiefe=bodentiefe,
2   voidhoehe=voidhoehe, bodenbereich=bodenbereich, gittergroessen=gittergroessen,
3   gitter_boden_vertikal=gitter_vertikal, schichten=schichten, schichtmaterial=schichtmaterial,
4   restmaterial=restmaterial, euler=False, xypartition=[True, True], partition_durchziehen=True,
5   viertel=4);

```

When changing the parameters, a variety of soil bodies can be produced. Two examples are shown in figure 6.6 with the geometric parameters shown in code 6.16 and 6.17 respectively.



**Figure 6.6:** Two variations of soil bodies created with `Boden()` function. The multiple adjustable parameters allow for many possibilities. Left hand side: Created with parameters shown in code 6.16. Right hand side: Created with parameters shown in code 6.17.

**Code 6.16:** Parameters for soil body creation (left hand side of figure).

```

1 bodentiefe = 10.0;
2 voidhoehe = 1.5;
3 bodenbereich = [[4.0, 3.0], [1.0, 1.0],
4                 [0.25, 0.25]];
5 gittergroessen = [[1.0], [0.2], [0.1]];
6 gitter_vertikal = 0.1;
7 schichten = [0.5, 5.0];
8 # euler = True;
9 # partition_durchziehen = True;

```

**Code 6.17:** Parameters for soil body creation (right hand side of figure).

```

1 bodentiefe = 1.2;
2 voidhoehe = 0.0;
3 bodenbereich = [[0.2], [0.05]];
4 gittergroessen = [[0.08, 0.015], [0.015]];
5 gitter_vertikal = 0.02;
6 schichten = [0.8];
7 # euler = False;

```

After a soil body has been created, each a material has to be assigned to each section. For Lagrange soil bodies, this can be done with code 6.18, for Eulerian soil bodies the section assignment does not require any loops and looks similar to code 6.19.

**Code 6.18:** Assigning sections on a Lagrangian body.

```

1 for materialdaten in materialien_boden:
2     partBoden.SectionAssignment(offset=0.0, offsetField='', offsetType=MIDDLE_SURFACE,
3     region=partBoden.sets['set' + materialdaten[0]], sectionName='sec' + materialdaten[0],
4     thicknessAssignment=FROM_SECTION);

```

**Code 6.19:** Assigning sections on a Eulerian body.

```

1 partBoden.SectionAssignment(offset=0.0, offsetField='', offsetType=MIDDLE_SURFACE,
2   region=partBoden.sets['setAll'], sectionName='secEuler', thicknessAssignment=FROM_SECTION);

```

If a structure interacting with the soil body is positioned above the soil material, there is no restriction on its geometry. If the structure is already positioned within the soil body (i. e. wished-in-place), either an Eulerian soil volume (with Abaqus/CEL) or an axisymmetric structure with a Lagrangian soil volume (and an appropriate sketch passed to `rotationsprofilpunkte`) is strongly recommended when using `Boden()`.

To automatically create a discrete field of all elements of an Eulerian body containing no/partial material due to the presence of another geometry, the Eulerian Volume Fraction tool can be used. Either this field or any sets of elements can be used to create empty elements by assigning only material to where it should be.

Wished-in-place non-axisymmetric structures within a Lagrangian soil volume are not supported with `Boden()` and should be created manually.

### 6.4.3 Soil Stress

After a soil body is created (e. g. with `Boden()`), the earth pressure can be calculated automatically for each soil layer depending on the properties of each material, the vertical position and the weight of all layers above this layer. Simply invoke `BodenspannungErstellen()` as shown in code 6.20. `BodenspannungErstellen()` automatically creates a stress distribution with the Abaqus function `GeostaticStress()` for each soil layer. The optional argument `verbose=False` can be set to `True` to output a table with the soil parameters.

**Code 6.20:** Creating soil stress for each layer with `BodenspannungErstellen()`.

```

1 BodenspannungErstellen(modell=mymodel, bodenname='Boden', nullspannung=0.0, voidhoehe=voidhoehe,
2   schichten=schichten, bodentiefe=bodentiefe, materialschichten=Materialschichtliste,
3   verwendeteBodenwerte=verwendeteBodenwerte, verwendeteMaterialien=verwendeteMaterialien);

```

Although this should cover most situations, if stresses are to be assigned explicitly, a function `BodenspannungDirektZuweisen()` can be used. This function does not support `materialschichten`, `verwendeteBodenwerte` and `verwendeteMaterialien` but expects a custom list of densities of all soil layers passed to `bodendichten` and earth pressure coefficients passed to `k0Werte`.

Both functions are designed to work with the presented workflow and output of `Boden()`. If they are to be used separately, certain sets are expected to be present in the soil part.

## 7 Setting and Transferring States

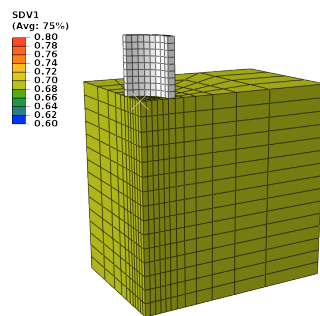
In some cases it might be necessary to set an initial solution for a set of nodes/elements ahead of the simulation. Since initial solutions are not supported in the Abaqus/CAE environment (except in the keyword editor), this is usually done directly in the input file after model creation. The following functions were developed for triangles and quadrilateral elements (2D) as well as three-dimensional elements like tetrahedrons and hexahedrons (other elements should not be expected to work). The following examples use the model described in section 4.1.

### 7.1 Constant Initial Values for Solution Dependent Variables

With `KonstanteAnfangsloesungFuerSet()` it is possible to set initial values for solution dependent variables (SDV) before creating an input file. Constant values are applied to the specified set (`setname`) of the given instance (`instname`). The following code sets the first SDV to 0.7 and the next nine SDVs to zero for all elements in *setAll* in instance *instBoden*.

**Code 7.1:** Initialising SDVs in a set with constant values.

```
1 KonstanteAnfangsloesungFuerSet(modell=mymodel, instname='instBoden', setname='setAll',  
2   ausgabewerte=[0.7] + [0.0 for x in range(9)]);
```



**Figure 7.1:** Constant initial values for the SDV within all elements of the soil volume.

*hint »* `KonstanteAnfangsloesungFuerSet()` only works for solution dependent variables (SDV) of user routines.

## 7.2 Transfer Values from an Output Database

It is more common to transfer selected results of an existing output database as initial solutions to a new model. Abaqus provides functionalities to transfer results if the same model is used.

If a different model with comparable geometry is used (e.g. coarser model, submodel or different meshing) and the geometric positions and dimensions are similar enough, the function `Zustandsuebertragung()` can be called directly. The following code can be used to transfer the stresses from an instance `INSTBODEN` in the output database named `example.odb` to an instance `instBoden` in the new model.

**Code 7.2:** Transferring the state from an output database to a model database.

```
1 Zustandsuebertragung(session=session, odbname='example.odb', odbinstname='INSTBODEN',
2   variablenliste=['S'], modell=mymodel, mdbinstname='instBoden');
```

By default, the results from the last frame in the last step are used in `Zustandsuebertragung()`. This can be changed by defining the number of another step/frame with the optional arguments `step` and `frame`. The results to transfer are defined in `variablenliste` (S, SVAVG, U and SDV were tested).

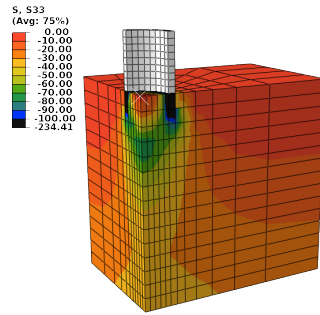
*hint »* Transformations with `Zustandsuebertragung()` are not optimized for accuracy. Excessive repeated usage with different meshes might degrade the results.

Occasionally the output database and the similar new model might have a different absolute position/offset. To transfer states between such models a coordinate transformation has to be applied (i.e. by using `Koordinatentransformation()` and passing the target coordinates to the optional `mdbknoten`-argument of `Zustandsuebertragung()`<sup>1</sup>). Code 7.3 shows an example transformation for the model described above (shifted by 7 units in *x*-direction and -2 units in *y*-direction).

**Code 7.3:** Transforming the reference coordinates for the state transfer.

```
1 mdbknoten = Knotentransformation(xneu='x+7', yneu='y-2',
2   punktliste=mdb.models[<modelName>].rootAssembly.instances['instBoden'].nodes);
3 Zustandsuebertragung(session=session, odbname='<odb-file>', odbinstname='INSTBODEN',
4   variablenliste=['S'], modell=mymodel, mdbinstname='instBoden', mdbknoten=mdbknoten);
```

<sup>1</sup>It is also possible to transform the source coordinates with the optional `odbknoten`-argument, but the default way is to use `mdbknoten`.



**Figure 7.2:** Initial state transferred from the output database of a similar model.

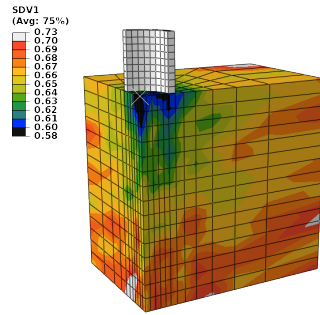
*hint » Although it is possible to scale/rotate coordinates with `Knotentransformation()`, this will only affect the position (which results are assigned to which element/node). The values at those positions will NOT be changed in any way which is especially important for tensor values.*

### 7.3 Assign Arbitrary Initial Values

It is possible to define initial solutions on a 3D grid (regularly meshed hexahedron) and transfer those values to any model. Therefore, three variables `zielkoordinaten`, `zielelemente` and `zielwerte` have to be provided to assign an arbitrary state with `Zustandszuweisung()` (most of the other arguments are similar to `Zustandsuebertragung()`). `zielkoordinaten` contains all point coordinates and each list in `zielelemente` refers to the indices of the coordinates for the corresponding element. `zielwerte` represents the initial values which can be either for elements (if it has the same amount of values as `zielelemente`) or for the nodes (it must have the same amount of values as `zielkoordinaten`).

*hint » Node values like U or S have to be defined per node and element values like SDV per element (theoretically per integration point but a simplified approach is used here).*

An example is shown in code 7.4. Like `Zustandsuebertragung()`, `Zustandszuweisung()` also allows for a coordinate transformation (e.g. with `Koordinatentransformation()`) and passing the transformed target coordinates to the optional `mdbknoten`-argument. The default variables are SDV as defined in the optional argument `variablentyp=['SDV']`. The values for `zielkoordinaten`, `zielelemente` and `zielwerte` are expected to be saved in a csv file (with ascending coordinate directions). The csv file is read and parsed with `ZielwertquaderEinlesen()`.



**Figure 7.3:** Example of assigning initial values from a 3D grid saved in a csv file.

**Code 7.4:** Assigning an initial state as defined in file.

```

1 [knoten, elemente, zielwerte] = ZielwertquaderEinlesen(dateiname='zielwertzuweisung.csv',
2   numKoordinaten=3, numVar=20, knotenwerte=False);
3 Zustandszuweisung(session=session, modell=mymodel, zielkoordinaten=knoten, zielelemente=elemente,
4   zielwerte=zielwerte, mdbinstname=instBoden.name);

```

`ZielwertquaderEinlesen()` expects `numKoordinaten=2` for 2D elements with 4 nodes or `numKoordinaten=3` for 3D elements with 8 nodes. `numVar` represents the amount of initial values (which might be greater than the amount of values stored in the file, but not smaller. Missing values will be initialised as 0). The optional argument `knotenwerte=False` assumes values defined for elements (i. e. at integration points) or at the nodes (if `True`).

The csv file for `Zustandszuweisung()` can be created with any program. `abapys` provides a function called `ZielwertquaderErstellen()` to handle all situations where a generator function for all initial values can be stated explicitly. A list of increasing values has to be specified for each coordinate direction (and passed to `xwerte`, `ywerte` and `zwerte` respectively). `ZielwertquaderEinlesen()` also expects a generator function to be passed to `ergebnisfunktion`.

**Code 7.5:** Creating initial values and saving them to a file.

```

1 xwerte = [round(5*x)/100.0 for x in range(11)] + [0.56, 0.67, 0.75, 0.84, 0.94] \
2   + [round(105 + 10*x)/100.0 for x in range(6)];
3 ywerte = [round(5*x)/100.0 for x in range(11)] + [0.56, 0.67, 0.75, 0.84, 0.94, 1.05];
4 zwerte = [-round(5*x)/100.0 for x in range(11)] + [-0.56, -0.67, -0.75, -0.84, -0.94] \
5   + [-round(105 + 10*x)/100.0 for x in range(6)];
6
7 ZielwertquaderErstellen(dateiname='zielwertzuweisung.csv', xwerte=xwerte, ywerte=ywerte,
8   zwerte=zwerte, ergebnisfunktion=Anfangswertverteilung);

```

For this example the following user defined function `Anfangswertverteilung()` (shown in code 7.6) is used.



**Code 7.6:** Example function to calculate initial values based on coordinates.

```
1 def Anfangswertverteilung(punktkoordinaten):
2     """Erwartet drei Koordinaten, um daraus einen Funktionswert zu berechnen. In dieser oder einer
3     gleichwertigen Funktion kann die gewünschte Verteilung von beliebigen Anfangszuständen fuer
4     ein Modell definiert werden, die mit ZielwertquaderErstellen() in eine Datei geschrieben werden.
5     """
6     from math import sqrt, exp
7     import random
8     #
9     ausgangswert = 0.7;
10    verdichtung = -0.1;
11    schwankung = 0.05;
12    #
13    referenz_punkt = [0.0, 0.0, 0.0];
14    referenz_entfernung = 1.0;
15    entfernung = sqrt(sum([(punktkoordinaten[idx] - referenz_punkt[idx])**2 \
16        for idx in range(len(punktkoordinaten))]));
17    #
18    funktionswert = random.gauss(mu=ausgangswert + verdichtung*exp(-entfernung/referenz_entfernung),
19        sigma=(3.0*schwankung)**2);
20    return [round(funktionswert*1000.0)/1000.0];
21 #
```

## 8 Querying Meshed Element Data

Sometimes specific information should be obtained like the volume of elements/parts/instances, which element contains a specified point coordinate and how a point within an element is weighted by the element nodes. Those tasks concerning meshed elements can also be accomplished with abapys. Since it can be of interest for model databases as well as output databases, most of the following functions work for both situations.

### 8.1 Volume of Elements

The volume of single elements<sup>1</sup> can be obtained with `ElementVolumen()` by passing the point coordinates to the `punkte` argument (and `dimensionen=3` for 3D elements or `dimensionen=2` for 2D elements). By iterating over all elements (and getting their node coordinates with `PunktkoordinatenVonElement()`) the cumulative volume of all elements can be determined (as shown in code 8.1).

`PunktkoordinatenVonElement()` requires a list of all nodes passed to `knoten`, the investigated element containing its node labels (`element`) and a label-index mapping either with `ErstelleLabelsortierteGeomlist()` (for output databases) or with a simple generated list for model databases passed to `listenhilfe`. *instBoden* is an instance reference of either a model database or an output database.<sup>2</sup>

**Code 8.1:** Calculating the volume of an instance.

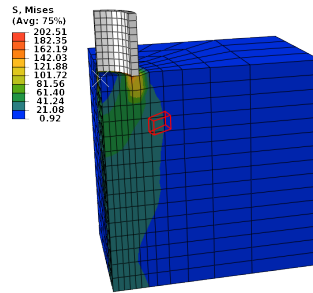
```
1 listenhilfe = [idx for idx in range(len(instBoden.nodes))]; # mdb (indices instead of labels)
2 listenhilfe = ErstelleLabelsortierteGeomlist(geomlist=punkte) # odb
3 vol = 0.0;
4 for element in instBoden.elements:
5     punkte = PunktkoordinatenVonElement(element=element, knoten=instBoden.nodes,
6         listenhilfe=listenhilfe);
7     vol += ElementVolumen(punkte=punkte, dimensionen=3);
```

For parts in a model database the whole volume can easily be determined with `PartVolumen()` as shown in code 8.2.

---

<sup>1</sup>The introductory note in section 7 on supported/tested element types also applies here.

<sup>2</sup>See also code 5.2 and figure 2.1 for accessing instances in a model database or output database.



**Figure 8.1:** Highlighted element containing the given reference coordinates.

**Code 8.2:** Calculating the volume of a part.

```
1 volBoden = PartVolumen(part=partBoden);
```

## 8.2 Element Containing a Certain Point

Sometimes it might be handy to find out which element contains a certain point. `abapys` provides a function called `PunktInElement()`, which returns the label of the element. All relevant elements have to be passed as argument to `elemente`, their nodes to `knoten` and the coordinates of the point of interest to `referenzpunkt`.

A simple example is shown in code 8.3 and the result in figure 8.1. If `PunktInElement()` is called more than once, a label-index mapping should be provided to the optional argument `listenhilfe=[]`.

**Code 8.3:** Finding and highlighting an element containing a given point.

```
1 refpunkt = (0.4, 0.4, -0.4);
2 elemente = session.odbs[<odbname>].rootAssembly.instances[<instname>].elements;
3 knoten = session.odbs[<odbname>].rootAssembly.instances[<instname>].nodes;
4
5 label_zu_idx_elemente = ErstelleLabelsorتيerteGeomlist(geomliste=elemente);
6 label_zielelement = PunktInElement(elemente=elemente, knoten=knoten, referenzpunkt=refpunkt);
7 ziel_element = element[label_zu_idx_elemente[label_zielelement]];
8 highlight(ziel_element);
```

*hint » If the reference coordinate is exactly on the surface/edge/node between two or more elements, only one of the matching elements is returned.*

### 8.3 Weighting of Element Nodes on a Point

The weighting of a point can be determined by `KnotengewichtungInElement()` if the element containing the point is known. The function returns the nodes of the given element and the weighting (0 to 1) i.e. how much node is contributing to this point. The center point has a weighting of  $1/(\text{number of nodes})$  for each node, e.g. eight times 0.125 for hexahedral elements with eight nodes. The following example uses the variables/results from the last code snippet (if called more than once, label-index mapping should be provided to the optional argument `listenhilfe=[]`.)

**Code 8.4:** Determining the weighting of element nodes on a point.

```
1 zielknoten, zielgewichtung = KnotengewichtungInElement(element=ziel_element, referenzpunkt=refpunkt,  
2   knoten=knoten);
```

If coordinates of the nodes are already known/given, it is also possible to use a similar function `KnotengewichtungPunktInPunktkoordinaten()`. Neither element nor node list is required since all coordinates are given as a list to `punkte`. Continuing from the previous example, the coordinates can be extracted as shown in code 8.5.

**Code 8.5:** Extracting the coordinates of an elements' nodes.

```
1 label_zu_idx_knoten = ErstelleLabelsSortierteGeomlist(geomliste=knoten);  
2 punktkoordinaten = PunktkoordinatenVonElement(element=ziel_element, knoten=knoten,  
3   listenhilfe=label_zu_idx_knoten);
```

Now the weighting can be determined with `KnotengewichtungPunktInPunktkoordinaten()` as shown in code 8.6.

**Code 8.6:** Determining the weighting of point coordinates on a point.

```
1 zielgewichtung = KnotengewichtungPunktInPunktkoordinaten(punkte=punktkoordinaten,  
2   referenzpunkt=refpunkt, dimensionen=3);
```

## 9 Output Processing

Output processing is a common task after each simulation and abapys provides functions for visualizing and exporting simulation results. Opening an output database is already described in section 4.1 and basic output printing in section 4.

### 9.1 Select Output Variables

Field Output variables can be selected by using the appropriate Abaqus commands. Code 9.1 can be used to display the output variable SVAVG33 (line 2–3) as contours on the deformed geometry (see also figure 9.1). The values of interest are set to the interval  $[-500.0, 0.0]$ . Since those Abaqus commands are simple and easy to adjust there are no substitute abapys functions.

**Code 9.1:** Selecting output variables.

```
1 myviewport.oddbDisplay.display.setValues(plotState=(CONTOURS_ON_DEF, ));
2 myviewport.oddbDisplay.setPrimaryVariable(variableLabel='SVAVG', refinement=(COMPONENT, 'SVAVG33'),
3     outputPosition=INTEGRATION_POINT, );
4 myviewport.oddbDisplay.contourOptions.setValues(minAutoCompute=OFF, minValue=-500.0,
5     maxAutoCompute=OFF, maxValue=0.0);
```

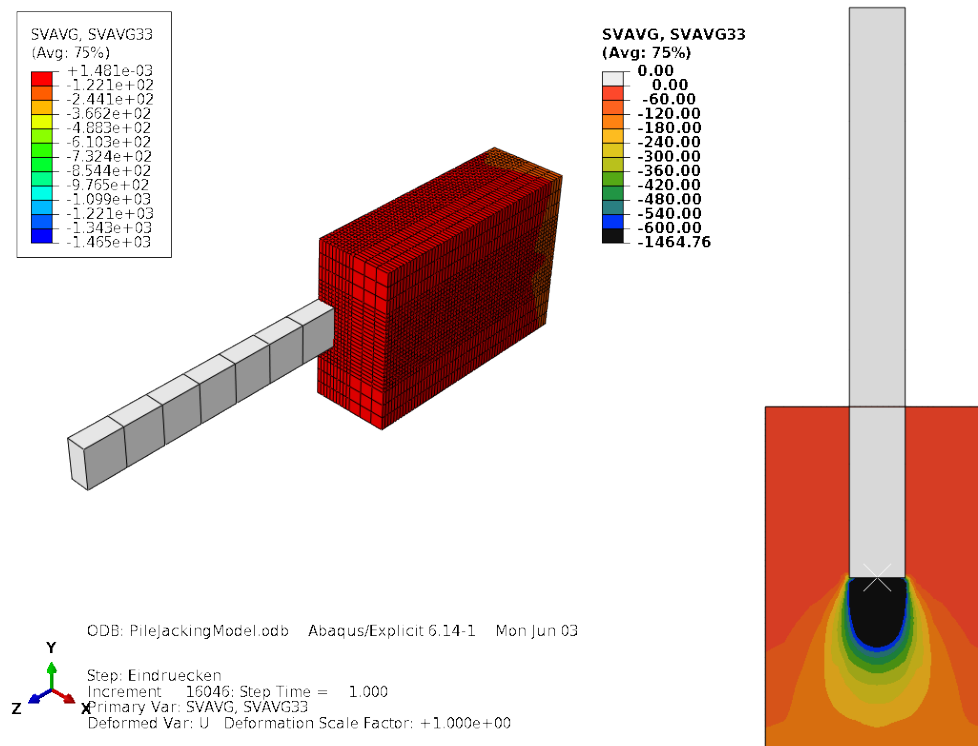
### 9.2 Plot Output

In contrast to variable selection, plotting FieldOutput or HistoryOutput variables can be more cumbersome. There are some restrictions as hinted below, but much can already be achieved with the PlotOutput() function.

Plots can either be created for distinct elements by specifying their label(s) or whole element sets. When only defining  $y$ -values, the variable is plotted over simulation time. An example for plotting the HistoryOutput is shown in code 9.2 and in figure 9.2.<sup>1</sup> It is also possible to plot one OutputVariable over another (see code 9.3).

---

<sup>1</sup>When saving plots with BildSpeichern(), it is recommended to use hintergrund=True.



**Figure 9.1:** Left hand side: Viewport after opening an odb file and setting the displayed output variable SVAVG33. Right hand side: Additional adjustments like ViewportVerschoenern() (described in section 4) and code 9.1.

**Code 9.2:** Plotting HistoryOutput over time.

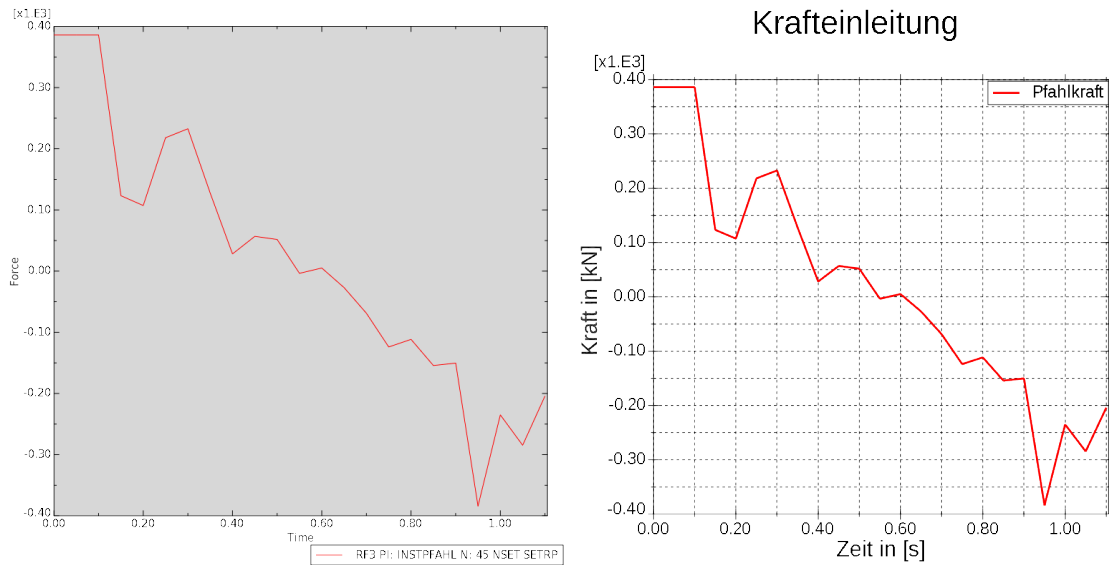
```
1 PlotOutput(session=session, odbname=odbname, yvar=(['instPfahl', 'RF3']),
2   ylabel='Kraft in [kN]', xlabel='Zeit in [s]', titel='Krafteinleitung',
3   legendeneintraege=['Pfahlkraft']);
```

*hint » Although multiple data points can be chosen as y-values, only one data set for x-values is allowed.*

**Code 9.3:** Plotting FieldOutput over another FieldOutput.

```
1 PlotOutput(session=session, odbname=odbname, yvar=(['instPfahl', 'RF3']),
2   xvar=('U', 'U3'), xvarposition=('INSTPFAHL.SETRP'), xlabel='Weg in [m]',
3   titel='Kraftverlauf', posxdir=False, legendeneintraege=['Pfahlkraft']);
```

*hint » Abaqus requires the x-values to be monotonic increasing when plotting one OutputVariable over another.*



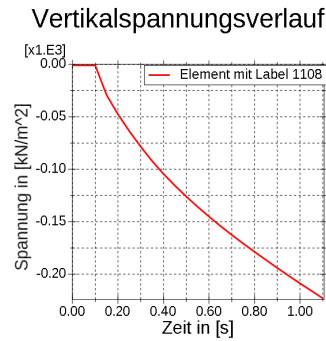
**Figure 9.2:** Plotting an OutputVariable does the job (left hand side) but is visually not as appealing as the results produced with `PlotOutput()`. It is also straightforward to plot one or multiple curves over each other with `PlotOutput()`.

It is possible to beautify the plot by specifying label descriptions, title and legend entries (to `xlabel/ylabel`, `titel` and `legendeneintraege` respectively). Defining custom legend entries internally calls `PlotLegendeFormatieren()`, which in turn applies additional modifications (line thickness and legend position). So it is recommended to always use custom legend entries for readability and consistent style.

`PlotOutput()` requires a reference to the current `session`, the name of the output database (`odbname`) and a definition of the variable to plot passed to `yvar`.

To plot `FieldOutput` data `yvar` has to be defined as (`<variable_name>`, `<component>`) and either `yvarposition=(<odb_set_name>)` (like in code 9.3) to plot the `FieldOutput` for all set elements or `yvarposition=(<instance_name>`, `[<label(s)>]`) (like in code 9.4) have to be provided. If `xvar` is also used as a `FieldOutput`, `xvarposition` has to be defined similar to `yvarposition`.

To plot `HistoryOutput` data, either use `yvar=( [<instance_name>`, `<variable_name>]`) if the correct `HistoryOutput` can be identified or use `yvar=(<output_name>)` by specifying the complete output name. No position should be passed when using `HistoryOutput` data (i.e. `yvarposition=[]` if `yvar` represents a `HistoryOutput`). This also applies to `xvar` if used as a `HistoryOutput`.



**Figure 9.3:** Stress over time on element labelled 1108.

`PlotOutput()` also has the following optional arguments with the given default values

- `ylabel=''` and `xlabel=''` to add a label to the  $y$ -axis or  $x$ -axis respectively,
- `posydir=True` and `posxdir=True` to increase values upwards/to the right (if `True`) or downwards/to the left with inverted axes (if `False`) respectively,
- `ylimit=[]` and `xlimit=[]` to define the limits of the  $y$ -axis and  $x$ -axis respectively,
- `titel=''` to give a title to the current plot,
- `legendeneintraege=[]` to add a legend entry for each data set to be plotted and
- `plothinzufuegen=False` to clear the current plot and create a new one (if `False`) or add the data to the current plot (if `True`).

### Example: Plotting by Label

To plot the vertical stresses `S33` of an element (with label 1108) of instance `instBoden`, the following `PlotOutput()` call can be used. Additionally, a label to the  $x$ -axis and the  $y$ -axis, a title and a legend entry are added. The result is shown in figure 9.3.

**Code 9.4:** Plot vertical stress at one element over time.

```

1 PlotOutput(session=session, odbname=odbname, yvar=('S', 'S33'), yvarposition=('instBoden', [1108]),
2   ylabel='Spannung in [kN/m^2]', xlabel='Zeit in [s]', titel='Vertikalspannungsverlauf',
3   legendeneintraege=['Element mit Label 1108']);

```

## 9.3 Process Output Data

This section focuses on two different kinds of output data: `FieldOutput/HistoryOutput` data and `XY` data. `FieldOutput/HistoryOutput` data is created and saved in the output database if requested in the model definition. `XY` data can be generated by one or more



FieldOutput/HistoryOutput data sets present in the output data base (and may contain less points in time and/or space).

Often FieldOutput/HistoryOutput data and/or XY data should be gathered and exported to continue processing it in another program (like OCTAVE or MATLAB). There are two different functions implemented to save FieldOutputs: FieldOutputSpeichern() directly takes a FieldOutput chunk for all defined output nodes/elements in a specified step and frame (see code 9.5).

**Code 9.5:** Save FieldOutput for all elements at one step/frame.

```
1 FieldOutputSpeichern(dateiname='spannung_eindruecken_10', session=session, odbname=odbname,
2   fieldOutput='SVAVG', step=1, frame=10);
```

Alternatively a subset of the FieldOutput can be saved by generating XY data first. Using FieldOutputVorbereiten(), data from a FieldOutput can be collected for all steps and frames for certain points of interest (this is usually the right choice when the behaviour during the whole simulation is of interest). XYDatenSpeichern() can be used to save existing XY data. It requires the name of the XY data (e.g. output of FieldOutputVorbereiten()) passed to xydatenname as shown in code 9.6.

**Code 9.6:** Save FieldOutput at certain elements/sets for all steps/frames.

```
1 spannung_element = FieldOutputVorbereiten(session=session, odbname=odbname,
2   var=['SVAVG', 'SVAVG33'], varposition=('INSTBODEN', [9951]));
3 XYDatenSpeichern(dateiname='spannung_9951', session=session, xydatenname=spannung_element[0].name);
```

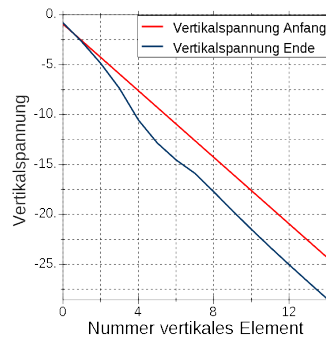
There is an additional function XYDatenAnElementen() to extract XY data from certain elements. As a preparation, elements are selected by a coordinate range, sorted by *z*-direction and their labels saved in *elem\_sortiert* as described in code 5.10.

Now the relevant XY data can be extracted for one point in time with XYDatenAnElementen(). In code 9.7 the vertical stresses S33 at the beginning (zeitpunkt=0.0) and after 1.1 s are extracted. The second extracted XY data is saved in a csv file with XYDatenSpeichern().

**Code 9.7:** Extracting and saving XY data.

```
1 xydaten_anfang = XYDatenAnElementen(session=session, odbname=odbname, odbinstname=instname,
2   labelliste=labelliste, zeitpunkt=0.0, var=['S', 'S33'], name='Vertikalspannung Anfang');
3 xydaten_ende = XYDatenAnElementen(session=session, odbname=odbname, odbinstname=instname,
4   labelliste=labelliste, zeitpunkt=1.1, var=['S', 'S33'], name='Vertikalspannung Ende');
5 XYDatenSpeichern(dateiname='Ausgabe_S33', session=session, xydatenname=xydaten_ende.name);
```

Both extracted XY data sets (representing a spatial distribution of stresses at one point in time) are plotted with PlotXYDaten() as shown in code 9.8 and figure 9.4.



**Figure 9.4:** XY data for vertical stresses S33 for a selection of elements extracted at two points in time.

**Code 9.8:** Plotting XY data.

```
1 PlotXYDaten(session=session, xyListe=[xydaten_anfang, xydaten_ende],
2     xlabel='Nummer vertikales Element', ylabel='Vertikalspannung',
3     legendeneintraege=[xydaten_anfang.name, xydaten_ende.name]);
```

## Example: Working with XY Data

Two examples of generating XY data from FieldOutputs at one point in time for a selection of nodes or elements can be seen in code 9.9 and 9.10. The resulting XY data can also be saved with `XYDatenSpeichern()` or plotted with `PlotXYDaten()` as shown before.

**Code 9.9:** Creating XY data for predefined nodes in ascending  $z$ -direction.

```
1 instBoden = myodb.rootAssembly.instances['INSTBODEN'];
2 zielKnoten = BedingteAuswahl(elemente=instBoden.nodes,
3     bedingung='(elem.coordinates[0] > var[0.2])', var=[0.2]);
4 sortierte_knotenlabels = KnotenAuswahlLabelliste(knoten=zielKnoten, sortierung=2, aufsteigend=True);
5 xydaten_knoten = XYDatenAnElementen(session=session, odbname=odbname, odbinstname='INSTBODEN',
6     labelliste=sortierte_knotenlabels, zeitpunkt=0.05, var=('U', 'U3'), name='U3-Label');
```

**Code 9.10:** Creating XY data for predefined elements in descending  $y$ -direction.

```
1 listenhilfe = ErstelleLabelsSortierteGeomlist(geomliste=instBoden.nodes);
2 zielElemente = ElementAuswahl(elemente=instBoden.elements, punktliste=instBoden.nodes,
3     listenhilfe=listenhilfe, bedingung='(punkt.coordinates[0] > punkt.coordinates[1])');
4 sortierte_elemlabels = ElementAuswahlLabelliste(elemente=zielElemente, punktliste=instBoden.nodes,
5     sortierung=1, aufsteigend=False, listenhilfe=listenhilfe);
6 xydaten_elemente = XYDatenAnElementen(session=session, odbname=odbname, odbinstname='INSTBODEN',
7     labelliste=sortierte_elemlabels, zeitpunkt=0.05, var=('SVAVG', 'SVAVG33'), name='SVAVG33-Label');
```

In some cases it might even be of use to create new FieldOutput data by combining existing ones. There is a special function called `SkalarenFieldOutputErstellen()` which needs a

FieldOutput and operates on its data. As an example code 9.11 shows how to save the trace of the stress tensor as a FieldOutput variable.<sup>2</sup>

**Code 9.11:** Create a new FieldOutput by combining existing results.

```
1 SkalarenFieldOutputErstellen(name='Spur des Spannungstensors', session=session, odbname=odbname,  
2   referenzAusgabe='SVAVG', bedingung='data11+data22+data33');
```

---

<sup>2</sup>While many variations are possible with this functions, the trace of a stress tensor has its own function `SpannungsSpurAlsFieldOutput()`.

# 10 Model Creation Template

In this chapter, a model similar to the one introduced in the »Abaqus Pile Jacking Tutorial« will be created. In contrast to the other tutorial, the focus here will be on starting with a template, i. e. general purpose structure for scripting, using exemplary code snippets when possible and adjusting them afterwards for a specific model. The resulting template is intended to be usable for various soil-structure interaction problems using a Coupled Eulerian Lagrangian approach in Abaqus/CAE.

This procedure might look burdensome when applied to a specific model. But the benefit of this structure will be more obvious when applied to more than one model.

Each part of the template is presented in detail in the following ten sections, which are divided like the structure of the template:

1. *Imports* – might always be chosen as suggested in code 10.1,
2. *Parameter Definitions* – should contain all modifiable parameters in one place to control the model. Usually, this will be the only part where changes are necessary in the script later on,
3. *Material and Contact Processing* – for all used materials and contacts,
4. *Parts* – Geometry, sets, meshes and section assignments for all parts,
5. *Assembly and Contact Applications* – Positioning of the whole model,
6. *Initial and Boundary Conditions* – Definition of the complete initial situation,
7. *Steps* – Change in loads/movement divided in single steps,
8. *Postprocessing* – Late adjustments of the model definition,
9. *Job* – Job creation and
10. *Keyword Adjustments* – Manual editing input file keywords.

## 10.1 Imports

Since the imports were already discussed in section 2.1 and 3.1, the beginning of the model creation template in code 10.1 should already look familiar.

**Code 10.1:** Imports and loading abapys for model creation.

```

1 # -*- coding: utf-8 -*-
2 from part import *
3 from material import *
4 from section import *
5 from assembly import *
6 from step import *
7 from interaction import *
8 from load import *
9 from mesh import *
10 from optimization import *
11 from job import *
12 from sketch import *
13 from visualization import *
14 from connectorBehavior import *
15
16 abapys_dir = r'<X:\path\to\abapys>';
17
18 sys.path.insert(0, abapys_dir);
19 from abapys import *
20
21 InitialisiereAbapys(session=session, version=version, pfad=abapys_dir);

```

## 10.2 Parameter Definitions

The next part should contain all modifiable parameters for this model. Usually, all geometric variables, soil and material parameters, simulation parameters and version numbers can be found here. Since a model similar to the pile jacking model should be created, a soil part and a pile part are needed.

The relevant parameters/geometries for both parts are defined in code 10.2–10.4.<sup>1</sup> For convenience, parameter unpacking can be used to easily change multiple values. If the hash sign in line 35 of code 10.2 would be removed (uncommented) and line 34 would be commented, all values for the variables in line 29 would be changed at once.<sup>2</sup>

**Code 10.2:** Soil geometry definition block (first part).

```

29 bodentiefe, voidhoehe, bodenbereich, gittergroessen, gitter_vertikal = [
30 #>0 >0 [radius] oder [konstant] oder >0
31 # [laenge/2, breite/2] [gross, klein]
32 # [m] [m] [m] (aussein->innen) [m] (aussein->innen) [m]
33 #-----|-----|-----|-----|-----|
34 4.0 , 2.0 , [[2.0, 2.0], [1.0, 1.0]], [[0.4], [0.1]] , 0.1 #
35 #5.0 , 3.0 , [[2.5, 2.5], [1.5, 1.5]], [[0.5], [0.1]] , 0.1 #
36 ];

```

<sup>1</sup>When combining all code snippets in this chapter, some lines might seem to be missing (like line 23–28 of modelcreation.py). Those lines are either empty or have structural comments.

<sup>2</sup>This approach can be very useful if a parameter study is to be conducted.

All values in this code regarding the soil geometry are used later on by a function `Boden()` (see code 10.11) to create the soil body. Since this function is designed to create cuboid or cylindric soil bodies/partitions, all governing parameters can and should already be adjusted here. Each partition/entry in *bodenbereich* needs a corresponding entry in *gittergroessen* to define its mesh size (see also section 6.4 for an explanation of the parameters).

A model can have multiple soil layers. They can be defined as shown in code 10.3. Every depth entry in *schichten* must have a corresponding material entry in *schichtmaterial*. If the lowest layer is not equal to the height of the model (*bodentiefe*), the remaining material is filled with *restmaterial*.

**Code 10.3:** Soil geometry definition block (second part).

```

38 schichten, schichtmaterial, restmaterial = [
39 #>[0]      >[' ' ]      >' '
40 #[m]
41 #-----|-----|-----|
42 [4.0]    , ['Sand']    , 'Sand'    #
43 #-----|-----|-----|
44 ];

```

In code 10.4 all pile geometry parameters are defined. All materials (used) are defined in code 10.5. After using a general purpose elastic steel material, all material names used in the soil layer definitions in line 42 must be declared in *materialien\_boden*.

**Code 10.4:** Pile geometry definition block.

```

46 pfahllaenge,   pfahlbreite,   pfahlhoehe,   gitter_pfahl = [
47 #>0 oder      >0 oder      >0 oder      >0 oder [radius, hoehe]
48 #[start, ende] [start, ende] [start, ende] oder [xwert, ywert, zwert]
49 #[m]          [m]          [m]          [m]
50 #-----|-----|-----|-----|
51 1.0           , 1.0           , 10.0           , 1.0           #
52 #-----|-----|-----|-----|
53 ];

```

**hint »** The database names of all used materials have to be present `Materialdatenbank_20####.xlsx` (material database file), which will be read automatically later on. This file should be in the *abapys* directory or the directory specified with the *pfad* argument in `InitialisiereAbapys()`.

**Code 10.5:** Material definitions block.

```

60 stahl_dichte = 7.87; # [kN/m^3]
61 #                E-Modul   Querdehnz.
62 #                E [kPa]   nu [-]
63 #                |-----|-----|
64 stahl_elastisch = [ 210e6    , 0.3          ];
65

```

```

66 materialien_boden = [
67 #   Abaqus-Bez.   Datenbankname   Parameter-Bez.   Saettigung   Lagerungsd.   Stoffgesetz
68 #   >'           >'           ''           [0-1]       [0-1]       >'
69 # |-----|-----|-----|-----|-----|-----|
70 # [ 'Sand'      , 'Standardparameter', ''      , 0.0      , 0.6      , 'Mohr-Coulomb' ],
71 # |-----|-----|-----|-----|-----|-----|
72 # Die Lagerungsdichte/Verdichtungsgrad kann bestimmt werden aus
73 # a) Anfangsdichte rho_0:      I_r = (rho_0 - rho_min)/(rho_max - rho_min)
74 # b) Anfangsporenzahl e_0:      I_e = (e_max - e_0)/(e_max - e_min)
75 ];

```

Independent variables like the simulation parameters are also defined here (code 10.6).

**Code 10.6:** Definition of simulation parameters.

```

82 viertel = 2;
83
84 # Reibung
85 kontakt_mit_reibung = False;
86 reibungskoeffizient = 0.22;
87
88 # Bewegung
89 pfahlgeschwindigkeit = -1.0;
90
91 # Simulationszeiten und -ausgabefrequenz
92 secondsperoutput = 0.05; # [s]

```

The step parameters concerning step duration and scaling factors for each step are defined next (explicit analysis). As shown in code 10.7 multiple parameters are defined in a list. In case parameters between different steps are not subject to change, individual parameter assignment might also be reasonable.

**Code 10.7:** Explicit steps parameter block.

```

95 schritt_schwerkraft = [
96 #   timePeriod   scaleFactor   linearBulkViscosity   quadBulkViscosity
97 #   [s]          [-]          ? -?                  ? -?
98 # |-----|-----|-----|-----|
99 # 0.1          , 1.0          , 0.48                  , 1.2          #
100 # |-----|-----|-----|-----|
101 ];
102
103 schritt_eindruecken = [
104 #   timePeriod   scaleFactor   linearBulkViscosity   quadBulkViscosity
105 #   [s]          [-]          ? -?                  ? -?
106 # |-----|-----|-----|-----|
107 # 1.0          , 1.0          , 0.48                  , 1.2          #
108 # |-----|-----|-----|-----|
109 ];

```

At the end of the parameter definitions it is recommended to use a version history (see code 10.8). Changes can be documented here and with the use of a designated variable saved

within the model name. It is also reasonable to define a suitable model name here, possibly with more adjustments by previously defined variables. Line 122 of code 10.8 does the actual creation of the model.

**Code 10.8:** Additional variable definitions.

```

116 modellversion = 1;
117 #
118 # 1: Erstellung und erste Tests
119
120
121 modelname = 'PileJackingModel' + str(modellversion).zfill(2);
122 mdb.Model(name=modelname, modelType=STANDARD_EXPLICIT);
123 mymodel = mdb.models[modelname];

```

### 10.3 Material and Contact Processing

After all parameters are defined, the material processing can commence. An elastic steel material is created in code 10.9 with the previously defined parameters. All materials of the soil body (as defined above) are processed in `MaterialUndBodensectionErstellen()`. This abapys function calculates earth pressure and earth pressure coefficients of all soil layers by using the data of a material database file `Materialdatenbank_20####.xlsx`. The database name as defined in line 70 (code 10.5) must match the name of the corresponding line/entry in the material database to successfully load the material parameters. Line 136 is used to remove duplicate entries, if the same material is used for different layers (since the material parameters for the same material must only be read once).

**Code 10.9:** Material processing.

```

130 g = 9.81;
131 mymodel.Material(name='Stahl');
132 mymodel.materials['Stahl'].Density(table=((stahl_dichte, ), ));
133 mymodel.materials['Stahl'].Elastic(table=((stahl_elastisch[0], stahl_elastisch[1]), ));
134 mymodel.HomogeneousSolidSection(material='Stahl', name='secStahl', thickness=None);
135
136 verwendeteMaterialien = sorted(set(schichtmaterial + [restmaterial]));
137 benoetigtUserroutine, verwendeteBodenwerte = BodenmaterialUndSectionErstellen(modell=mymodel,
138     verwendeteMaterialien=verwendeteMaterialien, verfuegbareMaterialien=materialien_boden);

```

Although surface-to-surface contact can be more efficient, using the general contact formulation for interactions in a model is easy to set up and can provide a realistic contact behaviour. The template uses normal contact by default but allows to decide, if friction behaviour should be used as well. Code 10.10 specifies all necessary commands for normal and tangential contact behaviour (in the whole model), which is applied later in code 10.14.



**Code 10.10:** Contact initialisation.

```

145 mymodel.ContactProperty('Kontakt');
146 mymodel.interactionProperties['Kontakt'].NormalBehavior(allowSeparation=ON,
147     constraintEnforcementMethod=DEFAULT, pressureOverclosure=HARD);
148 if (kontakt_mit_reibung):
149     mymodel.interactionProperties['Kontakt'].TangentialBehavior(
150         dependencies=0, directionality=ISOTROPIC, elasticSlipStiffness=None,
151         formulation=PENALTY, fraction=0.005, maximumElasticSlip=FRACTION,
152         pressureDependency=OFF, shearStressLimit=None, slipRateDependency=OFF,
153         table=((reibungskoeffizient, ), ), temperatureDependency=OFF);
154 else:
155     mymodel.interactionProperties['Kontakt'].TangentialBehavior(formulation=FRICTIONLESS);

```

## 10.4 Parts

Next, the soil and pile part are to be created. For generating the soil body, the `Boden()` function is used in line 162–165 of code 10.11. Usually, all parameters defined in section 10.2 are just inserted in the function as shown here. Afterwards the part is referenced by the variable *partBoden* in line 166 and a section is assigned to it.

**Code 10.11:** Soil part.

```

162 Boden(modell=mymodel, name='Boden', bodentiefe=bodentiefe, voidhoehe=voidhoehe,
163     bodenbereich=bodenbereich, gittergroessen=gittergroessen, gitter_boden_vertikal=gitter_vertikal,
164     schichten=schichten, schichtmaterial=schichtmaterial, restmaterial=restmaterial,
165     viertel=viertel, xypartition=[False, False], partition_durchziehen=True);
166 partBoden = mymodel.parts['Boden'];
167 partBoden.SectionAssignment(offset=0.0, offsetField='', offsetType=MIDDLE_SURFACE,
168     region=partBoden.sets['setAll'], sectionName='secEuler', thicknessAssignment=FROM_SECTION);

```

For the pile part a cuboid is used in this example (`Quader()` function in line 175–177). `abapys` provides other functions to create simple geometries, (screw) pile profiles or extruding/revolving a sketch (see section 6.2 for a list of those functions and their usage).

The pile part is also referenced by a variable (*partPfahl* in line 178) and a section is assigned.

**Code 10.12:** Pile part.

```

175 Quader(modell=mymodel, name='Pfahl', laenge=pfahllaenge, breite=pfahlbreite,
176     hoehe=pfahlhoehe, materialtyp=DEFORMABLE_BODY, gittergroesse=gitter_pfahl, rp=True,
177     xypartition=[False, False], viertel=viertel);
178 partPfahl = mymodel.parts['Pfahl'];
179 partPfahl.SectionAssignment(offset=0.0, offsetField='', offsetType=MIDDLE_SURFACE,
180     region=partPfahl.sets['setAll'], sectionName='secStahl', thicknessAssignment=FROM_SECTION);

```

## 10.5 Assembly and Contact Applications

If a model consists of rather simple parts, they can all be created and instantiated as shown in the previous section and then assembled as shown in code 10.13. The regeneration command in line 187 may be not needed in this example, but is necessary whenever parts or their properties are changed after they have been instantiated.<sup>3</sup> For assigning boundary conditions later on, *instBoden* and *instPfahl* are referenced in line 183 and 184 and the pile geometry is translated.

**Code 10.13:** Assembly of the model.

```
187 mymodel.rootAssembly.regenerate();
188 instBoden = mymodel.rootAssembly.instances['instBoden'];
189 instPfahl = mymodel.rootAssembly.instances['instPfahl'];
190 mymodel.rootAssembly.translate(vector=(0.0, 0.0, -bodentiefe), instanceList=('instBoden', ));
```

As we started working in the assembly, we apply the contact formulation defined previously as shown in code 10.14.

**Code 10.14:** Explicit General Contact application.

```
192 mymodel.ContactExp(createStepName='Initial', name='Allgemeinkontakt');
193 mymodel.interactions['Allgemeinkontakt'].includedPairs.setValuesInStep(
194     stepName='Initial', useAllstar=ON);
195 mymodel.interactions['Allgemeinkontakt'].contactPropertyAssignments.appendInStep(
196     assignments=((GLOBAL, SELF, 'Kontakt'), ), stepName='Initial');
```

## 10.6 Initial and Boundary Conditions

In almost any soil-structure interaction problem, the soil has to have an initial soil pressure. The different material dependent earth pressures and earth pressure coefficients were already calculated in the *MaterialUndBodensectionErstellen()* function used in code 10.9. In code 10.15 a geostatic stress condition is applied within *BodenspannungErstellen()* for every soil layer defined.

**Code 10.15:** Initial soil pressure distribution.

```
203 Materialschichtliste = schichtmaterial + [restmaterial];
204 BodenspannungErstellen(modell=mymodel, bodenname='Boden', nullspannung=0.0, voidhoehe=voidhoehe,
205     schichten=schichten, bodentiefe=bodentiefe, materialschichten=Materialschichtliste,
206     verwendeteBodenwerte=verwendeteBodenwerte, verwendeteMaterialien=verwendeteMaterialien);
```

<sup>3</sup>The regeneration command is also necessary when a new part is created and instantiated from the assembly of multiple simpler parts.

Next, all materials within an Eulerian body have to be assigned to it. Code 10.16 first assigns all materials used (line 209–211) and then assigns the list to the soil instance in line 213–214.

**Code 10.16:** Assign Eulerian materials.

```

208 assignmentList = [];
209 for idxMaterial, tempMaterial in enumerate(verwendeteMaterialien):
210     assignmentList += ((instBoden.sets['set' + tempMaterial],
211         Einheitsvektor(len(verwendeteMaterialien), idxMaterial)), );
212
213 mymodel.MaterialAssignment(assignmentList=assignmentList,
214     instanceList=(instBoden, ), name='Materialzuweisung', useFields=False);

```

Any other conditions (like rigid body for the steel pile) are defined here as shown in code 10.17.

**Code 10.17:** Rigid body definition for pile.

```

216 mymodel.RigidBody(bodyRegion=instPfahl.sets['setAll'], name='StarrerPfahl',
217     refPointRegion=instPfahl.sets['setRP']);

```

Now all boundary conditions have to be created and assigned to properly defined geometry sets. If a cylindric soil body should be generated, the `Boden()` function automatically creates a set `setMantelflaeche` for the curved lateral faces. Also when either a cuboid body should be generated or just half/a quarter of a cylinder, sets will be created of all outer faces (named by the normal of those faces). Therefore, the sets `setXFlaeche` and/or `setYFlaeche` might also be present. The bottom set `setUnterseite` is always created by `Boden()`. As shown in code 10.18 from line 219 onward, this template checks the existence of those sets and applies all eligible boundary conditions for the soil body. The boundary condition controlling the pile (by using the reference point or more precisely its set) is defined in line 235–237.

**Code 10.18:** Boundary conditions for soil body and pile.

```

219 if (instBoden.sets.has_key('setMantelflaeche')):
220     mymodel.VelocityBC(amplitude=UNSET, createStepName='Initial', distributionType=UNIFORM,
221         fieldName='', name='bcBodenMantel', region=instBoden.sets['setMantelflaeche'],
222         localCsys=None, v1=0.0, v2=0.0, v3=UNSET, vr1=UNSET, vr2=UNSET, vr3=UNSET);
223 if (instBoden.sets.has_key('setXFlaeche')):
224     mymodel.VelocityBC(amplitude=UNSET, createStepName='Initial', distributionType=UNIFORM,
225         fieldName='', name='bcBodenX', region=instBoden.sets['setXFlaeche'],
226         localCsys=None, v1=0.0, v2=UNSET, v3=UNSET, vr1=UNSET, vr2=UNSET, vr3=UNSET);
227 mymodel.VelocityBC(amplitude=UNSET, createStepName='Initial', distributionType=UNIFORM,
228     fieldName='', name='bcBodenY', region=instBoden.sets['setYFlaeche'],
229     localCsys=None, v1=UNSET, v2=0.0, v3=UNSET, vr1=UNSET, vr2=UNSET, vr3=UNSET);
230
231 mymodel.VelocityBC(amplitude=UNSET, createStepName='Initial', distributionType=UNIFORM,
232     fieldName='', name='bcBodenUnten', region=instBoden.sets['setUnterseite'],
233     localCsys=None, v1=UNSET, v2=UNSET, v3=0.0, vr1=UNSET, vr2=UNSET, vr3=UNSET);
234

```

```

235 mymodel.VelocityBC(amplitude=UNSET, createStepName='Initial',
236     distributionType=UNIFORM, fieldName='', localCsys=None, name='bcPfahlSteuerung',
237     region=instPfahl.sets['setRP'], v1=0.0, v2=0.0, v3=0.0, vr1=0.0, vr2=0.0, vr3=0.0);

```

## 10.7 Steps

Now all initial and boundary conditions are set and the simulation steps can be defined. Code 10.19 shows a typical example of an explicit dynamics step with parameters from the list defined in the parameter definition part in code 10.6.

**Code 10.19:** Definition of explicit step "Schwerkraft".

```

244 mymodel.ExplicitDynamicsStep(name='Schwerkraft', previous='Initial',
245     timePeriod=schritt_schwerkraft[0], scaleFactor=schritt_schwerkraft[1],
246     linearBulkViscosity=schritt_schwerkraft[2], quadBulkViscosity=schritt_schwerkraft[3]);

```

In the first custom defined step, all output requests can be set up. Here the standard output requests are renamed and their time interval is adjusted (line 248–252 of code 10.20). Additionally two new field output requests are created in line 261–267 and a history output request afterwards. Depending on the usage of an external subroutine for materials, SDV is also added to the list of requested output variables in line 254–259.

**Code 10.20:** FieldOutput and HistoryOutput requests.

```

248 mymodel.fieldOutputRequests.changeKey(fromName='F-Output-1', toName='FieldOutAll');
249 mymodel.historyOutputRequests.changeKey(fromName='H-Output-1', toName='HistoryOutAll');
250 mymodel.fieldOutputRequests['FieldOutAll'].setValues(variables=('U', ),
251     timeInterval=secondsperoutput);
252 mymodel.historyOutputRequests['HistoryOutAll'].setValues(timeInterval=secondsperoutput);
253
254 if (benoetigtUserroutine):
255     userroutineDatei = userroutine;
256     variables = ('A', 'SVAVG', 'EVF', 'SDV');
257 else:
258     userroutineDatei = '';
259     variables = ('A', 'SVAVG', 'EVF');
260
261 mymodel.FieldOutputRequest(createStepName='Schwerkraft', name='FieldOutBodenIn',
262     timeInterval=secondsperoutput, rebar=EXCLUDE, sectionPoints=DEFAULT, variables=variables,
263     region=instBoden.sets['setAll']);
264
265 mymodel.FieldOutputRequest(createStepName='Schwerkraft', name='FieldOutPfahl',
266     timeInterval=secondsperoutput, rebar=EXCLUDE, sectionPoints=DEFAULT, variables=('S', ),
267     region=instPfahl.sets['setAll']);
268
269 mymodel.HistoryOutputRequest(createStepName='Schwerkraft', name='HistoryOutPfahl', rebar=EXCLUDE,
270     region=instPfahl.sets['setRP'], sectionPoints=DEFAULT, timeInterval=secondsperoutput,
271     variables=('RF1', 'RF2', 'RF3', 'RM1', 'RM2', 'RM3'));

```

Loads like gravity can be applied in any step after the initial conditions (see code 10.21).

**Code 10.21:** Assignment of gravity force.

```
273 mymodel.Gravity(comp3=-g, createStepName='Schwerkraft', distributionType=UNIFORM,
274   field='', name='Schwerkraft');
```

Similar to code 10.19 other steps can be defined as shown in code 10.22.

**Code 10.22:** Definition of explicit step "Eindruecken".

```
281 mymodel.ExplicitDynamicsStep(name='Eindruecken', previous='Schwerkraft',
282   timePeriod=schritt_eindruecken[0], scaleFactor=schritt_eindruecken[1],
283   linearBulkViscosity=schritt_eindruecken[2], quadBulkViscosity=schritt_eindruecken[3]);
```

Loads and boundary conditions can be changed in steps after their definition. In this example we are interested in the soil stresses and pile reaction forces when driving a pile with constant speed into the soil body. The constant speed is defined as a (change of the) boundary condition as can be seen in code 10.23.

**Code 10.23:** Modifications of boundary conditions.

```
285 mymodel.boundaryConditions['bcPfahlSteuerung'].setValuesInStep(stepName='Eindruecken',
286   v3=pfahlgeschwindigkeit);
```

## 10.8 Postprocessing

This section becomes important when transferring states from an output database to a new model. In general everything needing a fully created model will but not yet a written out job should be handled here. An example for transferring states with `Zustandsuebertragung()` or applying initial solutions with one of the other available `abapys` functions can be seen in section 7.

## 10.9 Job

At this point the model is almost finished. To conduct the simulation, a job has to be created. The commands shown in code 10.24 can be adjusted (e. g. adjust *numCpus* or *numDomains* if more CPUs or domains are to be used respectively) but should already provide reasonable default values for small models.

**Code 10.24:** Job creation.

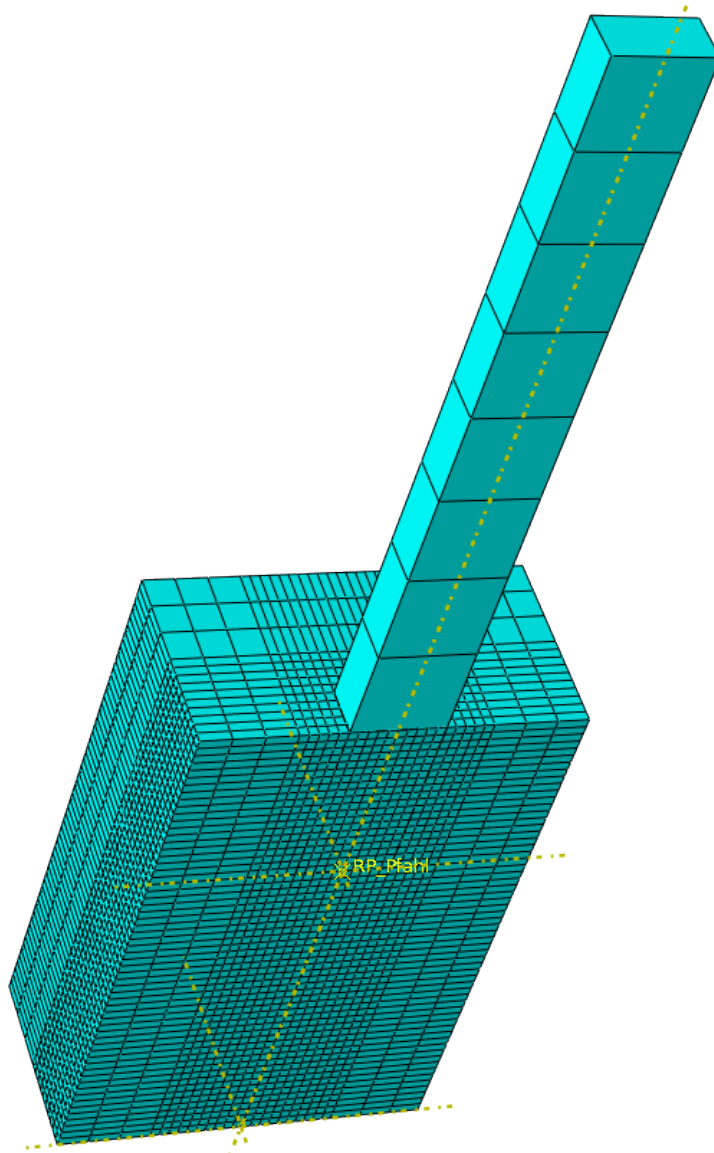
```
293 mdb.Job(activateLoadBalancing=False, atTime=None, contactPrint=OFF,  
294         description='', echoPrint=OFF, explicitPrecision=DOUBLE_PLUS_PACK,  
295         historyPrint=OFF, memory=90, memoryUnits=PERCENTAGE, model=modelname,  
296         modelPrint=OFF, multiprocessingMode=DEFAULT, name=modelname,  
297         nodalOutputPrecision=FULL, numCpus=2, numDomains=2,  
298         parallelizationMethodExplicit=DOMAIN, queue=None, resultsFormat=ODB,  
299         scratch='', type=ANALYSIS, userSubroutine=userroutineDatei, waitHours=0, waitMinutes=0);  
300  
301 myjob = mdb.jobs[modelname];
```

## 10.10 Keyword Adjustments

There are some cases, when manual intervention after model and job creation is still necessary, because Abaqus/CAE doesn't provide the desired functionality (yet). By accessing the keyword block, all input file entries can be read, modified and extended. To find a certain pattern, all keyword entries are investigated. A match with a certain phrase like in line 307 of code 10.25 allows modification of the matched block. Figure 10.1 shows the model created with the template and its default parameters presented in this chapter.

**Code 10.25:** Keyword changes.

```
305 mymodel.keywordBlock.synchVersions(storeNodesAndElements=False);  
306 for idx, text in enumerate(mymodel.keywordBlock.sieBlocks):  
307     if (text == '*Contact, op=NEW'):  
308         mymodel.keywordBlock.replace(idx, '*Contact');  
309  
310 myjob.writeInput(consistencyChecking=OFF);
```



**Figure 10.1:** Model created with code and parameters described in this chapter.

## 11 Closing Remark on a Graphical User Interface to Create Scripts

This document does not cover all functionalities of abapys but it provides an overview over its most important functions and contains some ideas on model creation and output processing. Using and adapting the presented functions in a different context requires some knowledge about Python scripting in Abaqus (i. e. a basic understanding of Python and functions/structures in Abaqus). So it is recommended to create your own scripts, experiment with the given examples and learn when and how to use them.

An additional way to get started with the basics of generating scripts for model creation or output processing (using some abapys function) is provided by the *abapys\_front* template for the SimpleScriptGenerator (found at »<https://github.com/d-zo/SimpleScriptGenerator>).

SimpleScriptGenerator is a graphical frontend to interactively create scripts based on modifiable code templates. The *abapys\_front* template allows to create ready-to-use Python scripts for Abaqus (e. g. the model presented in section 10 can be reproduced graphically). The resulting scripts can be run in Abaqus to see their effect and investigated in a text editor to understand the code. Adjusting parameters in the graphical frontend and inspecting its effects hopefully makes it easier to understand scripting in Abaqus and to start exploring what abapys has to offer.