

DELPHIN 6 Scripting-Tutorial

Inhaltsverzeichnis

1. Download der Skripte und Beispiele für Python 3 als ZIP-Dateien:	1
2. Überblick zum Scripting	1
3. Einige Überlegungen vor dem Start.	1
3.1. Schrittweises Vorgehen	1
3.2. Sequenzielles Vorgehen	2
4. Beispiel 1 - Ein einfaches Variantenskript zum Ändern der Anfangsbedingungen	2
4.1. Vorbereitungsschritte und grundlegender Algorithmus	2
4.2. Das Variantenskript	4
5. Beispiel 2 - Parameterstudie mit Anpassung des Gitters	8
5.1. Vorbereitungsschritte - Vorlagendateien	8
5.2. Variantenskript	11

1. Download der Skripte und Beispiele für Python 3 als ZIP-Dateien:

[DELPHIN6 Scripting-Beispiel 1](#)

[DELPHIN6 Scripting-Beispiel 2](#)

2. Überblick zum Scripting

Die Möglichkeit, den DELPHIN-Solver als Kommandozeilenprogramm mit einfachen ASCII-Text- und XML-Eingabedateien auszuführen, erlaubt eine sehr flexible Automatisierung von Variantenstudien. Dieses Tutorial zeigt einen pragmatischen und einfachen Ansatz, Python zur skriptgesteuerten Ausführung zu verwenden. Behandelt werden:

- Lesen, Ändern und Schreiben von DELPHIN-Projektdateien
- Neuerzeugen und Anpassen des Diskretisierungsgitters
- Ändern von Materialeigenschaften und Klimadaten
- sequenzielle oder parallele Ausführung der Simulation (mehrere Jobs, die jeweils selbst parallelisiert sein können)
- Lesen von Simulationsausgabedateien

3. Einige Überlegungen vor dem Start

Bei skriptgesteuerten Variantenstudien sollten einige Punkte berücksichtigt werden:

- Reicht der Festplattenspeicher aus, um die Ergebnisse aller Simulationsläufe zu speichern?
- Gibt es viele schnelle Simulationen oder eher langsame Simulationen, die von einer parallelen Ausführung profitieren würden?
- Hängen Varianten von vorherigen Läufen ab (z. B. bei Optimierungsrechnungen)?
- Erzeugen die Simulationen große Ausgabemengen (z. B. wenn viel Simulationszeit für das Schreiben und Lesen von Ausgabedaten benötigt wird)?

Einige allgemeine Empfehlungen:

3.1. Schrittweises Vorgehen

Wenn der Festplattenspeicher keine Einschränkung darstellt und die Varianten nicht von den Ergebnissen anderer Varianten abhängen (z. B. Parameter-Screening ohne Optimierungsrechnungen), sollte das Scripting in getrennten Schritten erfolgen:

1. zuerst die Simulationsprojekte erzeugen,

2. alle Jobs ausführen (parallel),
3. alle Ergebnisse einlesen und die Daten auswerten

Die einzelnen Projektdateien sollten dabei mit unterschiedlichen Projektnamen in einem Verzeichnis abgelegt werden, z. B. `var_01.d6p`, `var_02.d6p`. Die Ergebnisse landen in Verzeichnissen mit denselben Namen, wodurch die Auswertung erleichtert wird.



Dieses Vorgehen ist zugleich das robusteste. Jeder Verarbeitungsschritt kann jederzeit unterbrochen und später fortgesetzt werden, ohne bereits erledigte Arbeit wiederholen zu müssen.

3.2. Sequenzielles Vorgehen

Wenn der Festplattenspeicher begrenzt ist oder die Ergebnisse eines Laufs die Erzeugung des nächsten beeinflussen, muss sequenziell vorgegangen werden:

1. Projektdatei erzeugen
2. ausführen
3. auswerten
4. nächste Iteration → (1)

Dabei empfiehlt es sich, dieselbe Projektdatei zu verwenden, Zwischenstände jedoch zwischenzuspeichern, damit die Simulation fortgesetzt werden kann. Wenn beispielsweise die Ausgabe eines Temperatursensors vorliegt, kann die Datei `temperature_sensor.d6o` nach `temperature_sensor_XXX.d6o` kopiert werden, wobei XXX ein fortlaufender Iterationszähler ist. Muss die Simulation neu gestartet werden, können die neuesten Sensordaten (also die Datei mit der höchsten Nummer) als Ausgangsvariante verwendet werden.

4. Beispiel 1 - Ein einfaches Variantenskript zum Ändern der Anfangsbedingungen

Zum Einstieg erstellen wir ein einfaches 1D-Simulationsprojekt, variieren den Anfangsfeuchtegehalt über einen Wertebereich und erzeugen einen Datensatz mit dem Feuchtegehalt nach zwei Jahren.

4.1. Vorbereitungsschritte und grundlegender Algorithmus

Schritt 1

Vorlagenprojekt erstellen

Erstellen Sie wie gewohnt eine Projektdatei, führen Sie eine Testsimulation durch und speichern Sie die Datei. Anschließend erstellen Sie eine Kopie der Projektdatei, beispielsweise im Unterverzeichnis `variations`, und nennen sie `template.d6p`.

Schritt 2

Zu ändernde Teile der Projektdatei durch Platzhalter ersetzen

Öffnen Sie nun die Projektvorlage in einem Texteditor und ersetzen Sie die zu ändernden Teile der Datei durch einen eindeutigen Platzhalter. In unserem Beispiel wird `${INITIAL_MOISTURE_CONTENT}` verwendet:

Abschnitte mit Anfangsbedingungen vor der Änderung

```
<InitialConditions>
  <InitialCondition name="Initial moisture content" type="MoistureContent">
    <IBK:Parameter name="Value" unit="m3/m3">0.01</IBK:Parameter>
  </InitialCondition>
</InitialConditions>
```

Abschnitte mit Anfangsbedingungen nach dem Einfügen des Platzhalters

```
<InitialConditions>
  <InitialCondition name="Initial moisture content" type="MoistureContent">
    <IBK:Parameter name="Value"
unit="m3/m3">${INITIAL_MOISTURE_CONTENT}</IBK:Parameter>
  </InitialCondition>
</InitialConditions>
```

Die geänderte Projektdatei dient nun als Vorlage für alle Varianten.

Schritt 3

Projektdateien für alle Varianten erzeugen

Nun verwenden wir erstmals Python. Das benötigte Skript ist recht einfach:

- Inhalt der Vorlagendatei in eine Zeichenkette einlesen
- Platzhalter durch den Wert der aktuellen Variante ersetzen
- Projektdatei mit einem eindeutigen Variantennamen in das Verzeichnis schreiben
- Namen der Projektdatei in einer Liste von Simulationsjobs speichern

Schritt 4

Alle Jobs ausführen

Für jeden registrierten Job rufen wir den Solver auf und warten, bis alle Jobs abgeschlossen sind.

Schritt 5

Ergebnisse sammeln

Nun lesen wir für jede Variante alle Ergebnisdateien ein, extrahieren die relevanten Werte und speichern sie in einem Datencontainer.

Schritt 6

Ergebnisse in einer geeigneten Datei speichern

Abschließend schreiben wir die Daten zur weiteren Verarbeitung in eine Datei. Diagramme könnten auch direkt mit Python-Matplotlib erzeugt werden, *PostProc 2* eignet sich dafür jedoch besser.

Das folgende Skript erledigt all diese Schritte. Die Funktionen `read_file()` und `write_file()` sind praktische Hilfsfunktionen, die wir später noch häufig verwenden werden. Gehen wir das Skript nun Schritt für Schritt durch:

4.2. Das Variantenskript

Das Skript beginnt mit den üblichen Import-Anweisungen und Konstanten:

Dateikopf, Import-Anweisungen und Konstanten

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import os, sys
import subprocess

# import IBK utility classes
from IBK import *

# global constants
VARIANTS_SUBDIR = "variants"
DELPHIN_EXECUTABLE = "DelphinSolver"

# get absolute file path to DELPHIN solver (assumed to be inside this directory)
DELPHIN_EXECUTABLE = os.path.join(os.getcwd(), DELPHIN_EXECUTABLE)
if not os.path.exists(DELPHIN_EXECUTABLE):
    print("Missing DELPHIN Solver executable '{}'.format(DELPHIN_EXECUTABLE))
    exit(1)
```

Wichtig sind hier lediglich das Verzeichnis für die erzeugten Projektdateien sowie der Pfad zum DELPHIN-Kommandozeilen-Solver. Es ist empfehlenswert, einen vollständigen Dateipfad zur ausführbaren Solver-Datei zu verwenden. Statt einen absoluten Pfad fest einzutragen, erzeugen wir ihn aus dem aktuellen Arbeitsverzeichnis.

Die erste Funktionalität des Skripts besteht in der Definition zweier Hilfsfunktionen:

Funktionen zum Lesen und Schreiben einfacher Textdateien aus bzw. in Zeichenkettenvariablen einschließlich einer grundlegenden Fehlerbehandlung

```
def read_file(fname):
    try:
        fobj = open(fname, 'r')
        content = fobj.read()
```

```

        fobj.close()
        del fobj # release file handle
        return content
    except IOError as e:
        print(str(e))
        raise RuntimeError("Error reading/opening file '{}'.format(fname))

def write_file(fname, contents):
    try:
        fobj = open(fname, 'w')
        fobj.write(contents)
        fobj.close()
        del fobj # release file handle
    except IOError as e:
        print(str(e))
        raise RuntimeError("Error opening file '{}' for writing".format(fname))

```

Das eigentliche Skript beginnt mit dem Einlesen der zuvor erstellten Projektvorlage. Außerdem wird in Schritt 2 das Zielverzeichnis angelegt, sofern es noch nicht existiert.

Einlesen der Projektvorlage in eine Variable

```

# *** main ***

# 1. read template project file
TEMPLATE = read_file('template.d6p')

# 2. create directory for project variants
if not os.path.exists(VARIANTS_SUBDIR):
    os.mkdir(VARIANTS_SUBDIR)

```

Nun werden die Projektdateien für die verschiedenen Varianten erzeugt:

In 20 Schritten wird der Anfangsfeuchtegehalt variiert und die Projektdateien var_00.d6p ... var_19.d6p werden erzeugt

```

# 3. loop over range of moisture contents - 0.01...0.21 and create project variations
jobs = []
for i in range(20):
    moisture_content = 0.01 + i*0.01

    # replace placeholder text
    variant =
    TEMPLATE.replace('${INITIAL_MOISTURE_CONTENT}', '{}'.format(moisture_content))

    # create variant file name, file name pattern = var_xx.d6p where xx is a zero-
    leading number
    d6pfile = VARIANTS_SUBDIR + "/var_{:02d}.d6p".format(i)

    # write the file, unless it exists already

```

```

if not os.path.exists(d6pfile):
    write_file(d6pfile, variant)

# add to list of jobs, unless project was already simulated, use command line
arguments -x for windows,
# and -p=2 for 2 CPUs per job
if not os.path.exists(d6pfile[:-4] + "/var/restart.bin"):
    jobs.append([DELPHIN_EXECUTABLE, '-x', '--verbosity-level=0', '-p=2',
d6pfile])

```

Neben dem Erzeugen und Schreiben der Projektdateien erstellen wir auch eine Liste mit Simulationsjobs. Jeder Job wird im Wesentlichen als Kommandozeilenaufruf definiert und als Liste in der Jobliste gespeichert.

Nun können wir alle Jobs nacheinander ausführen (Option 1):

Sequenzielle Ausführung aller Jobs

```

# 4. run all the jobs, one after another
for job in jobs:
    try:
        print(job[-1])
        solverProcess = subprocess.Popen(job)
        solverProcess.wait()
    except OSError as e:
        print(str(e))

```

Hier starten wir für jeden Job einen Unterprozess und warten, bis dieser beendet ist. Anschließend wird der nächste Job gestartet, bis alle Simulationsjobs abgeschlossen sind.

Alternativ können mehrere Jobs gleichzeitig ausgeführt werden, was durch Mehrkernprozessoren ermöglicht wird. Die Verwaltung der parallelen Ausführung ist in der Hilfsklasse **IBK.JobRunner** gekapselt.

Parallele Ausführung aller Jobs (hier werden 4 parallele Jobs gestartet, die jeweils 2 CPUs verwenden, sodass alle 8 CPU-Kerne genutzt werden)

```

# 4. run all the jobs in parallel
jobRunner = JobRunner(4) # 4 jobs at the same time
jobRunner.run(jobs)

```



Bei größeren Simulationsprojekten ist es häufig sinnvoll, die Simulationen sequenziell auszuführen und jeder Simulation mit dem Kommandozeilenargument **-p=<cpu-count>** die maximale Anzahl an CPUs zuzuweisen. Dadurch steht jedem Simulationsjob die begrenzte Speicherbandbreite zur Verfügung und mehrere Jobs konkurrieren nicht um den Speicherbus.

Nachdem die Simulationen abgeschlossen sind, sammeln wir die Ergebnisse:

Einlesen aller Ergebnisdateien und Speichern der Daten jeder Variante in einer Ergebnisliste

```
# 5. read all result files and collect results

# create manager for geometry files - actually not used here, but needed by
Delphin6OutputFile()
geoManager = Delphin6GeoFileManager()

resultData = []
for i in range(20):
    moisture_content = 0.01 + i*0.01
    d6pfile = VARIANTS_SUBDIR + "/var_{:02d}.d6p".format(i)

    # NOTE: instead of recreating filename and variant here, one could also store this
    # information
    #         in a list and use it here

    # path to result d6o file
    resultFile = d6pfile[:-4] + "/results/Moisture_content_integral.d6o"

    # read file
    d6o = Delphin6OutputFile()
    if not d6o.read(resultFile, geoManager):
        print("Error reading result file '{}'.format(resultFile))
        continue

    # get value for time index #2
    value = d6o.valuesAtIndex(2)

    # store in vector
    resultData.append( (moisture_content, value[0]) )
```

Die Hilfsklassen `Delphin6GeoFileManager` und `Delphin6OutputFile` übernehmen das eigentliche Einlesen der Ausgabedateien im ASCII-Format. Der Geometriedatei-Manager speichert im Wesentlichen alle Geometriedateien, auf die aus `d6o`-Dateien verwiesen wird, sodass dieselbe Geometriedatei nicht mehrfach eingelesen werden muss. In diesem Beispiel werden die Geometriedateien kaum benötigt, da nur Integralwerte betrachtet werden. Sollen jedoch Profile ausgewertet werden, kann die vom Geometriedatei-Manager bereitgestellte Geometriedatei nach den benötigten Koordinaten abgefragt werden.

Abschließend schreiben wir die Daten in eine TSV-Datei (tabulatorgetrennte Werte). Diese lässt sich problemlos in LibreOffice Calc, Excel oder ähnlichen Programmen einlesen oder direkt mit *PostProc 2* visualisieren:

Speichern der Ergebnisse in einer für die Nachbearbeitung geeigneten Datei.

```
# 6. create a tsv-file with initial moisture content vs. final content
```

```
tsv = "Moisture content [m3/m3]\tFinal moisture mass [kg/m]\n"
for val in resultData:
    tsv = tsv + "{}\t{}\n".format(val[0], val[1])

write_file("results.tsv", tsv)
print("Wrote 'results.tsv'.")
```

Damit ist das erste Beispiel abgeschlossen. Quellcode und Beispielprojekt befinden sich im Archiv [DELPHIN6_scripting_example_1.7z](#).

5. Beispiel 2 - Parameterstudie mit Anpassung des Gitters

In diesem Beispiel führen wir eine komplexere Variantenstudie durch, bei der drei Dinge verändert werden:

- Klimadaten / Standort
- Materialeigenschaften der Dämmung (Wärmeleitfähigkeit)
- Dämmstoffdicke

Wir möchten die Zunahme des Tauwassers im Bauteil in Abhängigkeit von diesen Standorten visualisieren. Ziel ist es, mit *PostProc 2* Diagramme zu erstellen, die den zeitlichen Verlauf der Tauwasserbildung in der Konstruktion für die verschiedenen Varianten zeigen. Um die Diagrammerstellung möglichst einfach zu halten, wählen wir die Dateinamen der Variantenprojekte so, dass sie bereits für die Auswertung in PostProc aussagekräftig sind.

Das Ändern der Dicke einer Materialschicht erfordert eine Anpassung des Gitters. Hierzu verwenden wir das Kommandozeilenwerkzeug `CmdDiscretize`, das im Installationsverzeichnis von DELPHIN 6 installiert ist.

Sehen wir uns das Skript Schritt für Schritt an.

5.1. Vorbereitungsschritte - Vorlagendateien

Wie im ersten Beispiel müssen zunächst Vorlagendateien erstellt werden. Dazu richten wir ein Basisprojekt ein und führen eine Testsimulation durch (siehe Unterverzeichnis `BaseProject` im Verzeichnis `example_2`).

Wir erstellen eine Kopie des Projekts mit dem Namen `template.d6p` und kopieren `materials/Calcium silicate.m6` nach `materials/template.m6`.

Damit ergibt sich folgende ursprüngliche Verzeichnisstruktur:

```
BaseProject/
|—— CaSili30_Brick600_1D.d6p
|—— DE-01-TRY-2010__Bremerhaven__Jahr_00000K0_00007m.c6b
```

```

|—— DE-04-TRY-2010__Potsdam__Jahr_00000K0_00081m.c6b
|—— DE-15-TRY-2010__Garmisch_Parttenkirchen__Jahr_00000K0_00719m.c6b
|—— materials
|   |—— Calcium silicate.m6
|   |—— Clinker.m6
|   |—— Glue mortar.m6
|   |—— Historical Brick.m6
|   |—— Lime cement plaster.m6
|   |—— template.m6
|—— template.d6p

```

Bereits zu Beginn des Skripts wird diese gesamte Verzeichnisstruktur in das Zielverzeichnis für die Varianten kopiert.



Grundsätzlich ist es sinnvoll, Originaldaten und automatisch erzeugte Daten voneinander zu trennen, vorzugsweise in vollständig getrennten Verzeichnisstrukturen. Sollen die erzeugten Daten später bereinigt und gelöscht werden, kann die entsprechende Verzeichnisstruktur einfach entfernt werden, ohne einen Verlust der Originaldaten zu riskieren.

Nun bearbeiten wir die Materialvorlage `materials/template.m6` und fügen den Platzhalter `${LAMBDA}` ein:

Materialdatendatei mit eingefügtem Platzhalter

```

D6MARLZ! 006.001

[IDENTIFICATION]
  NAME                      = EN: Calcium silicate

...

[TRANSPORT_BASE_PARAMETERS]
  LAMBDA                    = ${LAMBDA} W/mK
  AW                       = 1.24 kg/m2s05
  MEW                      = 5 -
  DLEFF                    = 1.3e-05 m2/s

...

```

Als Nächstes bearbeiten wir die Projektvorlage `template.d6p`:

Projektvorlage mit eingefügten Platzhaltern

```

<?xml version="1.0" encoding="UTF-8" ?>
<DelphinProject ... fileVersion="6.1">

...

```

```

<!--Model data, solver settings, general parameters-->
<Init>
  <SimulationParameter>
    <BalanceEquationModule>BEHeatMoisture</BalanceEquationModule>
    <Interval>
      <IBK:Parameter name="Duration" unit="a">1</IBK:Parameter>
    </Interval>
    <ClimateDataFilePath>${Project
Directory}/${LOCATION}</ClimateDataFilePath>
    <StartYear>2000</StartYear>
    <TimeZone>1</TimeZone>
  </SimulationParameter>
  <SolverParameter>
    <LESSolver>GMRES</LESSolver>
  </SolverParameter>
</Init>

<!--
~~~~~
~~~~~-->

<!--List of all materials used in this construction-->
<Materials>
  <MaterialReference name="Lime cement plaster" color="#ff80ff80">${Project
Directory}/materials/Lime cement plaster.m6</MaterialReference>
  <MaterialReference name="Historical Brick" color="#ffff8040">${Project
Directory}/materials/Historical Brick.m6</MaterialReference>
  <MaterialReference name="Clinker" color="#ffff8080">${Project
Directory}/materials/Clinker.m6</MaterialReference>
  <MaterialReference name="Glue mortar" color="#ff8080c0">${Project
Directory}/materials/Glue mortar.m6</MaterialReference>
  <MaterialReference name="Calcium silicate" color="#ff80ffff">${Project
Directory}/materials/${INSULATION_MATERIAL}</MaterialReference>
</Materials>

<!--
~~~~~
~~~~~-->

<!--Discretization data (grid and sketches)-->
<Discretization>
  <XSteps unit="m">0.01 ${INSULATION_LAYER_WIDTH} 0.004 0.012 0.24 0.01 0.24
0.012 0.12 </XSteps>
</Discretization>

<!--
~~~~~
~~~~~-->

...

```

Es sind drei Platzhalter zu ersetzen: `${LOCATION}`, `${INSULATION_MATERIAL}`, `${INSULATION_LAYER_WIDTH}`



Vor der Bearbeitung muss das Diskretisierungsgitter in der Datei `template.d6p` entfernt werden. Die Materialschicht, die geändert werden soll, darf nur aus einer einzelnen Spalte (Zuordnung) bestehen.

5.2. Variantenskript

Wir beginnen mit den üblichen Importen und Konstantendefinitionen:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import os, sys
import subprocess
from distutils.dir_util import copy_tree

from IBK import *

BASE_SUBDIR = "BaseProject"
VARIANTS_SUBDIR = "variations"

DELPHIN_EXECUTABLE = "DelphinSolver"
DISCTOOL_EXECUTABLE = "CmdDiscretise"

# get absolute file paths tools (assumed to be inside this directory)
DELPHIN_EXECUTABLE = os.path.join(os.getcwd(), DELPHIN_EXECUTABLE)
if not os.path.exists(DELPHIN_EXECUTABLE):
    print("Missing DELPHIN Solver executable '{}'.format(DELPHIN_EXECUTABLE))
    exit(1)

DISCTOOL_EXECUTABLE = os.path.join(os.getcwd(), DISCTOOL_EXECUTABLE)
if not os.path.exists(DISCTOOL_EXECUTABLE):
    print("Missing CmdDiscretize executable '{}'.format(DISCTOOL_EXECUTABLE))
    exit(1)

# list of climate locations to use; store as tuples (file, description)
CLIMATE_FILES = [
    ("DE-01-TRY-2010__Bremerhaven__Jahr_00000K0_00007m.c6b", "Bremerhaven"),
    ("DE-04-TRY-2010__Potsdam__Jahr_00000K0_00081m.c6b", "Potsdam"),
    ("DE-15-TRY-2010__Garmisch-Partenkirchen__Jahr_00000K0_00719m.c6b", "Garmisch")
]

# list of insulation layer thicknesses to use
LAYER_WIDTHS = [ 0.04, 0.05, 0.06, 0.08, 0.10 ]

# insulation properties
LAMBDA = [ 0.04, 0.044, 0.048, 0.053]
```

Die Varianten werden über verschiedene Listen mit den jeweils zu verwendenden Parametern definiert. Für die Klimadatendateien speichern wir sowohl den Dateinamen der **c6b**-Datei als auch einen aussagekräftigen Namen, der später im Namen der Projektdatei verwendet wird.

Außerdem verwenden wir nun zwei Kommandozeilenwerkzeuge: den ausführbaren DELPHIN-Solver und das Kommandozeilenwerkzeug zur Diskretisierung.

Als Nächstes verwenden wir wieder unsere Hilfsfunktionen. Hinweis: Es kann sinnvoll sein, diese in eine separate Python-Datei auszulagern.

```
def read_file(fname):
    try:
        fobj = open(fname, 'r')
        content = fobj.read()
        fobj.close()
        del fobj # release file handle
        return content
    except IOError as e:
        print(str(e))
        raise RuntimeError("Error reading/opening file '{}'.format(fname))

def write_file(fname, contents):
    try:
        fobj = open(fname, 'w')
        fobj.write(contents)
        fobj.close()
        del fobj # release file handle
    except IOError as e:
        print(str(e))
        raise RuntimeError("Error opening file '{}' for writing".format(fname))
```

Zunächst lesen wir die Vorlagendateien ein, erstellen das Zielverzeichnis und kopieren die grundlegende Datenstruktur.

Grundlegende Einrichtung

```
# *** main ***

# 1. read template files
PROJECT_TEMPLATE = read_file(BASE_SUBDIR + '/template.d6p')
MATERIAL_TEMPLATE = read_file(BASE_SUBDIR + '/materials/template.m6')

# 2. create directory for project variants
if not os.path.exists(VARIANTS_SUBDIR):
    os.mkdir(VARIANTS_SUBDIR)

# 3. copy base project directory to variations directory
copy_tree(BASE_SUBDIR, VARIANTS_SUBDIR)
```

Nun erzeugen wir die Materialdateien:

Erzeugen der Materialdateien

```
# 4. create material variants
mat_files = []
for lmd in LAMBDA:
    m6file = "casi_{:.3f}.m6".format(lmd) # file name format casi_0.xxx.m6
    mat_files.append(m6file)
    variant = MATERIAL_TEMPLATE.replace('${LAMBDA}', '{}'.format(lmd))
    # write the file inside the 'materials' subdirectory
    write_file(VARIANTS_SUBDIR + "/materials/" + m6file, variant)
```

Der relative Pfad zur erzeugten Materialdatei wird in der Liste `mat_files` gespeichert, sodass der Dateiname später nicht erneut erzeugt werden muss. Falls der Dateiname später angepasst werden soll, muss dies dadurch nur an einer Stelle im Code erfolgen.

Schritt 5 ist die Erzeugung der Projektdateien:

Erzeugen der Projektdateien und automatische Gittererzeugung

```
# 3. create project variants
jobs = []
for cli in CLIMATE_FILES:
    for lmdIdx in range(len(LAMBDA)):
        lmd = LAMBDA[lmdIdx]
        m6file = mat_files[lmdIdx]

        for w in LAYER_WIDTHTHS:

            # replace placeholder texts for climate data file and layer width
            variant = PROJECT_TEMPLATE.replace('${LOCATION}', '{}'.format(cli[0]))
            variant = variant.replace('${INSULATION_MATERIAL}', '{}'.format(m6file))
            variant = variant.replace('${INSULATION_LAYER_WIDTH}', '{}'.format(w))

            # create variant file name, file name pattern =
            <location>_<width>_<lambda>.d6p
            d6pfile = VARIANTS_SUBDIR +("/{_}{:0.2f}_{:0.3f}.d6p".format(cli[1], w, lmd))

            # write the file
            write_file(d6pfile, variant)

            # run cmd line discretization tool
            # > CmdDiscretize -m=auto -l=1.7 -o=<d6pfile> d6pfile
            solverProcess = subprocess.Popen([DISCTOOL_EXECUTABLE, '-m=auto', '-l=1.7', '-o="' + d6pfile + '"', d6pfile])
            solverProcess.wait()

            # add to list of jobs, unless project was already simulated, use command
            line arguments -x for windows,
            # and -p=2 for 2 CPUs per job
```

```
if not os.path.exists(d6pfile[:-4] + "/var/restart.bin"):
    jobs.append([DELPHIN_EXECUTABLE, '-x', '--verbosity-level=0', '-p=2',
d6pfile])
```

Es gibt drei unabhängige Variationstypen, daher werden drei verschachtelte Schleifen verwendet. Wie in Beispiel 1 wird die Projektvorlage durch Ersetzen der Platzhalter konfiguriert.

Sobald die Projektdatei auf der Festplatte gespeichert ist, führen wir das Kommandozeilenwerkzeug `CmdDiscretize` aus (siehe auch Beispiel 1 zur Verwendung von `subprocess` für den Aufruf eines externen Programms). Das Argument `-l=1.7` bedeutet, dass ein Streckungsfaktor von 1,7 verwendet wird, was für diese 1D-Geometrie geeignet ist. Das Argument `'-o=' + d6pfile` legt fest, dass die Ausgabedatei mit der erzeugten Projektdatei identisch ist (In-place-Operation). Die Projektdatei enthält anschließend ein sinnvolles Diskretisierungsgitter. Wie bereits in Beispiel 1 werden die Projektdatei und eine Liste der Argumente zur Jobliste hinzugefügt.

Abschließend führen wir alle Simulationsjobs aus:

Simulationsprojekte parallel ausführen

```
# 6. run all the jobs in parallel
jobRunner = JobRunner(4) # 4 jobs at the same time
jobRunner.run(jobs)
```

Damit sind wir fertig. Nun können wir das Verzeichnis `variations` zu *PostProc 2* hinzufügen und ein Diagramm aller Zeitreihen der Tauwassermasse erstellen, um den ungünstigsten bzw. günstigsten Fall zu ermitteln.