DELPHIN 6 Scripting Tutorial

Andreas Nicolai
andreas.nicolai@tu-dresden.de

Version 1.0.0, June 2020

Table of Contents

Download of scripts and examples:

- DELPHIN6_scripting_example_1.7z
- DELPHIN6_scripting_example_2.7z

1. Scripting overview

The possibility to run the DELPHIN solver as command line executable with plain ASCII text and XML input files allows for very flexibly automatization of simulation variant studies. The tutorial shows you a pragmatic and simple approach to use Python for scripted execution. In the tutorial we'll cover:

- reading/modifying/writing DELPHIN project files
- regenerating/adjusting the discretization grid
- modifying material properties and climatic data
- running the simulation sequential/parallel (several jobs, each possibly parallized)
- reading simulation output files

2. Some thoughts before we start

When running scripted variant studies there are a couple of things to consider:

- Will the hard drive space be enough to store results of all simulation runs?
- Are there many fast simulations, or rather slow simulations (that would benefit from parallel execution)?
- Do variations depend on previous runs (e.g. optimization calculations)?
- Are the simulations output-heavy (e.g. much simulation time is spent on writing/reading output data)?

Some general recommendations:

2.1. Step-wise approach

When hard-drive space is not an issue, and variations do not depend on each others outputs (example: parameter screening, no optimization runs), do the scripting in distinct steps:

1. generate simulation projects first,
2. run all the jobs (in parallel),
3. read all results and evaluate data

Hereby, the individual project files should be placed in a directory with different project names, e.g. `var_01.d6p`, `var_02.d6p`. The results will end up in directories with same name, making analysis easy.

This approach is also the most robust. You can stop each processing step at will and continue later on, without redoing work.

2.2. Sequential approach

Either when hard-drive space is limited, or results of one run impact generation of the next, you need to perform a sequential operation:

1. generate project file
2. run
3. evaluate
4. next iteration → (1)

Hereby, it is recommended to use the same project file, but *cache* intermittend states to be able to continue simulation. For example, when a temperature sensor output is obtained, you may copy the file `temperature_sensor.d6o` to `temperature_sensor_XXX.d6o` where XXX is a running iteration counter. Then, if you need to re-start the simulation, you take the latest sensor data (i.e. file with largest number) as initial variant.

3. Example 1 - a simple variation script for modifying initial conditions

To get started, we create a simple 1D simulation project, modify the initial moisture content over a range of values and generate a data set with moisture content after 2 years.

3.1. Preparation steps and the basic algorithm

Step 1

create the template project

You set up a project file as usual, test-simulate it and save it somewhere. Then you create a copy of the project file, for example in some subdirectory `variations` called `template.d6p`.

Step 2

replace project file parts to be modified with placeholders

You now open the template project file in a text editor, and replace the parts of the file to be modified with a unique placeholder text. In our example, I chose `${INITIAL_MOISTURE_CONTENT}`:

Initial condition sections before

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

Initial condition sections after inserting placeholder

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

The modified project file will now serve as template for all our variants.

Step 3

generate project files for all variants

Now we use Python for the first time. The script to write is rather simple:

- read template file content into string
- replace placeholder text with value of current variant
- write project file with unique variant name to directory
- remember project file name in a list of simulation jobs

Step 4

run all jobs

We call the solver for each registered job and wait for all the jobs to finish.

Step 5

collect results

We now read all the result files for each variant, extract the interesting values and store them into some data container.

Step 6

save results into some meaningful file

Finally, we dump the data into some file for further processing (we might also generate diagrams with Python-Matplotlib right away, but *PostProc 2* does a better job at that).

The script below does all that. The functions `read_file()` and `write_file()` are some handy functions we will use quite often, later on. But let's go through the script step-by-step:

3.2. The variation script

The script starts the typical import statements and constants:

Header of the file, import statements and constants

```
#!/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)
```

Important here is only the directory for the generated project files, as well as the path to the DELPHIN command line solver. It is good practice to use a full file path to the solver executable, but instead of hard-coding an absolute path, we generated from the current working directory.

First functionality in the script comes with the declaration of two utility functions:

Functions for reading and writing simple text files to/from string variables, including some basic error handling

```
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))
```

The actual script starts with reading of the template project file we created earlier. Also, the target directory is created in step 2, if not existing already.

Reading of project template file into a 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)
```

Now the project files for the different variants are created:

In 20-steps the initial moisture content is variied and project files `var_00.d6p ... var_19.d6p` *are created*

```
# 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])
```

Besides creating and writing the project files, we also generate a list with simulation jobs. Each job is basically defined as command line, stored as list in the list of jobs.

Now we can run all jobs one after another (option 1):

Running all jobs sequentially

```
# 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))
```

Here, we spawn a subprocess for each job and wait until it has finished. Then we take the next, until all simulation jobs have finished.

Alternatively, we can run several jobs at the same time (thanks to multi-core CPUs). The thread-handling is wrapped in the utility class `IBK.JobRunner`.

Running all jobs in parallel (here, 4 parallel jobs are spawned, each using 2 CPUs, so we use all our 8 CPU cores)

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

For larger simulation projects it is often worthwhile to run the simulations sequentially, yet give each simulation the maximum number of CPUs with the `-p=<cpu-count>` command line argument. Then, the limiting memory bandwidth is available to each simulation job and multiple jobs won't compete over the memory bus.

Finally, after we are done with simulating, we collect the results:

Reading of all result files and storing data for each variation into a list of results

```
# 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]) )
```

The utility classes `Delphin6GeoFileManager` and `Delphin6OutputFile` handle the actual parsing of ASCII-Format output files. The geometry file manager basically stores all geometry files referenced from `d6o` files so that it is not necessary to read the same geometry file multiple times. In this example, there is not much use for geometry files, since we only look at integral values. But if profiles were to be inspected, the geometry file provided by the geometry file manager can be queried for the required coordinates.

Finally we write the data into a tsv-file (tab-separated values), which can be easily read into LibreOffice-Calc/Excel and the like, or be readily visualized with *PostProc 2*:

Dumping the results into some file suitable for post-processing.

```
# 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'.")
```

That's it already for the first tutorial. Source code and example project can be found inside the archive DELPHIN6_scripting_example_1.7z.

4. Example 2 - a parameter variation study including grid adjustment

In this tutorial we'll do a more complex variation study where we change 3 things:

- climatic data / location
- insulation material properties (thermal conductivity)
- insulation width

We want to visualize the increase in interstitial condensate depending on these locations. The goal is to create diagrams with *PostProc 2* that show the course of condensate forming in the construction for the different variants. To keep diagramm generation work simple, we'll construct our variant project file names such, that they will be already meaningful for PostProc-analysis.

Changing a material layer width requires grid adjustment. For this purpose we use the command line discretization tool `CmdDiscretize` installed in the DELPHIN 6 install directory.

Let's look at the script, step-by-step.

4.1. Preparation steps - template files

As in the first tutorial, we need to generate template files first. We set up a base project and test-simulate it (see subdirectory `BaseProject` in the `example_2` directory).

We create a copy of the project and name it `template.d6p`, and `materials/Calcium silicate.m6` to `materials/template.m6`.

The original directory structure looks as follows is obtained:

```
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_Partenkirchen__Jahr_00000K0_00719m.c6b
├─── materials
│    ├─── Calcium silicate.m6
│    ├─── Clinker.m6
│    ├─── Glue mortar.m6
│    ├─── Historical Brick.m6
│    ├─── Lime cement plaster.m6
│    └─── template.m6
└─── template.d6p
```

Pretty early in the script we will copy this entire directory structure to the target variations directory.

It is generally a good idea to separate original data and automatically generated data, preferably in a completely separate directory structure. In case you want to clean up and delete the generated data, you can just remove the directory structure without risking loss of original data.

Now we edit the material template file `materials/template.m6` and insert the placeholder `${LAMBDA}`:

Material data file with inserted placeholder text

```
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

...
```

Next we edit the project template file `template.d6p`:

Project template with inserted placeholders

```
<?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>

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

...
```

We have 3 placeholders to replace: `${LOCATION}`, `${INSULATION_MATERIAL}`, `${INSULATION_LAYER_WIDTH}`

> Obviously, we need to remove the discretization grid in the `template.d6p` file before editing it. The material layer we want to modify should be a single column (assignment) only.

4.2. Variation script

We start with the usual imports and constant definitions:

```
#!/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]
```

The variants are defined through different lists with the individual parameters to use. For the climate data files we store both the filename of the `c6b`-file and a meaningful name to be used in the project file name later.

Also, we now have two command line tools - the DELPHIN solver executable and the command line discretization tool.

Next we have our utility functions again (Hint: you may want to place them into some separate Python file):

```
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))
```

First of all, we read template files, create the target directory an copy the base data structure.

Initial setup

```
# *** 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)
```

Now we generate our material files:

Generating material files

```
# 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)
```

Note, we store the relative path to the generated material file in the list `mat_files`, so that we do not need to regenerate the filename later on (if we choose to adjust the file name later, we only need to do this at one place in the code).

Step 5 is the generation of the project files:

Project file creation and automated grid generation

```
# 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_WIDTHS:

            # 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 + "/{}_{:.2f}_{:.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])
```

We have three independent variation types, so there are three nested loops. The project template is configured as in example 1 by replacing the placeholder strings.

Once the project file is stored on the harddrive, we run the command line discretization tool `CmdDiscretize` (see also example 1 for the use of subprocess to run an external tool). The argument `-l=1.7` says "use stretch factor of 1.7", which is ok for this 1D geometry. The argument `'-o=' + d6pfile` defines the output file to be the same as the generated project file (in-place operation). This project file will now have a meaningful discretization grid. The project file and a list of arguments is added to the job list, as we did already in example 1.

Finally, we run all the simulation jobs:

Run simulation projects in parallel

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

And we are done. We can now add the `variations`-directory to *PostProc 2* and create a diagram of all the condensation mass time series to find out the worst/best-case.