Automated Variation Studies with DELPHIN using Python

A DELPHIN Tutorial from www.bauklimatik-dresden.de

Andreas Nicolai

11/19/2008

Contents

1 Introduction **1**

1.1 Getting Started 2

2 Creating Project Variations **2**

2.1 The DELPHIN Project File 3

2.2 Making Property Variations with Python 4

2.3 Reading and Writing Project Files 4

3 Calling the Solver **6**

3.1 Running one Solver at a Time 6

3.2 Running Several Solver Tasks in Parallel 7

3.3 Monitoring the Outcome of a Simulation 8

4 Extracting Sensitivity Parameters **8**

5 Summary **12**

1 Introduction

This tutorial is meant for advanced users of DELPHIN, who want to extend the basic functionality of the DELPHIN project management towards variation and sensitivity studies, or even probability methods. Of course, the concepts explained in this tutorial can be used with other scripting languages or 'real' programming languages as well. However, the language Python has been chosen for its simplicity and because Python scripts can be written with simple text editors and don't require learning an integrated development environment (IDE) first, like it is often necessary for languages such as C++ and Java.

The tutorial also shows a method for running scheduled DELPHIN simulations, like it can be done with plain batch files or bash scripts. However, the method explained in section 3.2 shows actually a *Batch XXL* version, with parallel execution and simulation crash tolerance.

In order to follow the tutorial and the examples below, go to *www.python.org* and download a copy of Python. Once installed, the install path should be added to the system `PATH` variable, so that it can be run from the command line. The Python webpage contains an excellent tutorial of the language. It is advisable to read this at some point (preferably before reading on).

1.1 Getting Started

Let's first get started with a simple test. For this test you should know where on your system DELPHIN is installed and where the `delphin_solver.exe` is located. Now create a text file containing the content of listing 1 (don't type the line numbers) and save it as `delphin_run.py`.

Listing 1: Python test script

```
1  # specify the path to the DELPHIN solver executable
2  delphin_executable = r'C:\Program Files\IBK\Delphin 5.6\delphin_solver.exe'
3
4  # print out the path
5  print 'Using: ' + delphin_executable
6
7  # import the module for calling external programs (creating subprocesses)
8  import subprocess
9
10 # call the DELPHIN solver, yet without command line arguments
11 retcode = subprocess.call([delphin_executable])
```

Let's quickly look at the script line-by-line. The lines beginning with a hash (#) character are comments. In line 2 we define a new string variable `delphin_solver` which contains the full path to the DELPHIN solver executable. Of course, this needs to be adjusted based on your DELPHIN installation path. Mind the 'r' in front of the string literal, which tells Python to treat the string as raw string and does not interpret the backslash character as escape sequence. In line 8 we tell Python to import all functionality related to creating/spawning new child processes. This is required so that we can call other programs (i.e. the DELPHIN solver) from a Python script. Finally, in line 11 we call the solver program using the member function `call()` of the subprocess module. Note, that the argument to `subprocess.call` is a list, with the first item being the path to the executable and the following items being the arguments to be passed. Lists in Python are created using brackets `[]`. The return code of the program is stored in the variable *retcode*, which can be used to check if the simulation was completed successfully.

Now let's test the script. Open a console window and change into the directory of the new script. To start the script type

```
python delphin_run.py
```

which will start the script. If everything is set up correctly, you will see the command line option help of the DELPHIN solver.

2 Creating Project Variations

The key to creating automated simulation variations is the availability of the project file as plain text. Using a script language we can easily modify small sections of the project file and create

Listing 2: DELPHIN Project File

```
147
148     [BOUND_COND]
149       TYPE                    = VAPDIFF
150       NAME                    = Outside Vapdiff
151       KIND                    = EXCHANGE
152       EXCOEFF                 = 2e-07 s/m
153       TEMPER                  = Outside temperature (data)
154       VAPPRESS                = Outside vapor pressure (constant)
155
156     [BOUND_COND]
157       TYPE                    = WATCONTC
158       NAME                    = Outside Watcontc
159       KIND                    = IMPFLUX
160       TEMPER                  = Outside ambient temperature (constant)
161       WATFLUX                 = Outside imposed flux (data)
162
163     [BOUND_COND]
164       TYPE                    = HEATCOND
165       NAME                    = Outside Heatcond
166       KIND                    = EXCHANGE
167       EXCOEFF                 = 25 W/m2K
168       TEMPER                  = Outside temperature (data)
169
170     [BOUND_COND]
171       TYPE                    = HEATCOND
172       NAME                    = Inside Heatcond
173       KIND                    = EXCHANGE
174       EXCOEFF                 = 8 W/m2K
175       TEMPER                  = Inside temperature (constant)
176
177     [BOUND_COND]
178       TYPE                    = VAPDIFF
179       NAME                    = Inside Vapdiff
180       KIND                    = EXCHANGE
181       EXCOEFF                 = 3e-08 s/m
182       TEMPER                  = Inside temperature (constant)
183       VAPPRESS                = Inside vapor pressure (data)
```

variations. Together with automatic solver runs and extraction of results from output data we get a system that works essentially as a loop around DELPHIN and simplifies the manual editing of several project files with the user interface.

In this tutorial we will only look at making modifications to the project file itself. However, the same principles apply to climatic data files and material data files, since they are also just plain text files.

2.1 The DELPHIN Project File

Let's begin with an example from the DELPHIN installation and open the example project `hamstad_benchmark_4.dpj` in a text editor. You will find all input data normally specified in the user interface in this file, except for imported material data and referenced climate data files.

Consider the section of the project file dealing with boundary conditions (see listing 2. The listing only shows the lines 147 to 184 that contain the boundary condition definitions. The words shown in blue indicate keywords recognized by DELPHIN. In the next parts of the tutorial we will create a script that modifies some parts of these boundary conditions.

2.2 Making Property Variations with Python

Let's assume that the heat and vapor mass exchange coefficients are not exactly known and a variation study of these parameters is required. The task is to vary the coefficients α and β in the ranges shown below.

$$\alpha = 8 \ldots 20 \ \text{W/m2K}$$
$$\beta = 0.3 \times 10^{-7} \ldots 1.6 \times 10^{-7} \ \text{s/m}$$

The relation between the transfer coefficients is actually not linear, but for simplicity (and to keep this tutorial short) the following variation rules are defined.

$$\alpha\,[i] = 8 + (20 - 8)\,\frac{i}{N}$$
$$\beta\,[i] = \left(0.3 + (1.6 - 0.3)\,\frac{i}{N}\right) \times 10^{-7}$$

Now we need to create a Python script to calculate arrays with the α and β parameters. Take a look at listing 3.

Listing 3: Calculating Variation Parameters

```
1  # define variable that contains the number of variation steps
2  N = 10
3
4  # create a loop where we calculate the properties
5  for i in range(N+1):
6      alpha = 8 + (20-8)*float(i)/N
7      beta = (0.3 + (1.6-0.3)*float(i)/N)*1e-7
8      # print the properties (for testing only)
9      print alpha
10     print beta
```

Most of the code is easy to understand. Pitfalls are the colon (':') at the end of the `for` statement and the indentation that defines the loop. In Python, you can use spaces or tabs, but they shouldn't be mixed. In any case, using spaces is recommended and the number of spaces used to indent the statements in a `for` loop must be exactly the same. Also, mind that the range needs to include the right limit as well, which is why we pass *N+1* to the function `range()` (which, btw, returns a list with the indices from 0 to the argument with step size 1).

A small comment may be needed regarding the `float()` function, which takes a number as argument and returns a floating point number. Just like other programming languages, Python distinguishes between whole number operations and floating point operations. By explicitly converting the index i into a floating point number, the whole calculation is done in floating point. Otherwise, using only integer arithmetic, truncation errors could produce wrong results.

2.3 Reading and Writing Project Files

Now that we can create arrays with modified parameters, we need to generate a new project file for each variant we want to simulate. For that we need to read in the base variant of the project (see listing 4).

Listing 4: Reading the Base Variant of the Project

```
1  # open a text file
2  basefile_name = 'hamstad_benchmark_4'
```

```
3  basefile_obj = open(basefile_name + '.dpj', 'r')
4  # read all lines
5  basefile = basefile_obj.readlines()
6  # close the file and remove the file object
7  del basefile_obj
8  # print the content of the file (again, only for debugging)
9  print basefile
```

In this script we open a file read-only, that's why we pass the 'r' flag. If the file does not exist or if a wrong path is given, the script will abort with an error message. In line 5 we read all lines of the file into an array. Then, in line 7 we close the file and free the allocated memory (this is not really necessary in Python, but it is good practice to get rid of file objects once they are no longer needed). The content of the file remains available in the list of strings named *basefile*.

Now we are free to create new project files. We recycle the loop that was introduced in the section above and simply compose file names based on the variation number (listing 5).

Listing 5: Composing Filenames

```
1  # we have N and basefile_name from above
2  for i in range(N+1):
3      filename = basefile_name + '_%02d' % i
4      print filename
```

This piece of script gives the following output.

```
hamstad_benchmark_4_00
hamstad_benchmark_4_01
hamstad_benchmark_4_02
hamstad_benchmark_4_03
hamstad_benchmark_4_04
hamstad_benchmark_4_05
hamstad_benchmark_4_06
hamstad_benchmark_4_07
hamstad_benchmark_4_08
hamstad_benchmark_4_09
hamstad_benchmark_4_10
```

The only 'magic' in this piece of the script is the composition of the new string *filename*. '%02d' can be read backwards: 'd' for decimal number, with 2 digits, and 0 as padding character. The '%' after the string means that i is inserted into the number placeholder.

Now we are already able to write our modified projects. We simply replace the lines that we want to modify with new lines and write the results to the new project file (Listing 6).

Listing 6: Complete Script to Create Project File Variations

```
1  basefile_name = 'hamstad_benchmark_4'
2  basefile_obj = open(basefile_name + '.dpj', 'r')
3  basefile = basefile_obj.readlines()
4  del basefile_obj
5
6  N = 10 # define the number of variation steps
7
8  for i in range(N+1):
9      # calculate modified parameters
10     alpha = 8 + (20-8)*float(i)/N
11     beta = (0.3 + (1.6-0.3)*float(i)/N)*1e-7
12
13     # exchange line with alpha parameter
14     basefile[173] = '      EXCOEFF                  = %g W/m2K\n' % alpha
15     # exchange line with beta parameter
16     basefile[180] = '      EXCOEFF                  = %g s/m\n' % beta
17     # don't forget to compose a new output folder
18     basefile[225] = '    OUTPUT_FOLDER            = ' + filename + '.results\n'
```

```
19
20      # compose new filename
21      filename = basefile_name + '_%02d' % i
22      print 'Creating: ' + filename
23
24      # open and write file
25      fileobj = open(filename + '.dpj', 'w')
26      fileobj.writelines(basefile)
27      del fileobj
```

In this version of the script we have to discuss a few somewhat tricky things. At first, we use again the '%' method to insert a number argument into a string. Here, 'g' specifies a general number type (decimal and floating point). Then, we use the index operator to assign new strings to items of the string list that represents the project file. Since indices are zero based, so line 175 is accessed via 174. Also, when assigning new strings each string needs to end with a '\n' (line feed character).

The output folder needs to be changed as well (in line 18), otherwise all variations would write their output files into the output directory of the base variant and that wouldn't be of much use. However, in some cases, particularly when many simulations with lots of output data are run but only a part of the data needs to be extracted, it may be useful to use the same output directory for all cases.

Of course, the actual line numbers may be different depending on your actual project. For more advanced scripts, you may want to search for a specific line or modify the project based on the context. This is, however, beyond the scope of this tutorial.

Lastly, the file is written using the `writelines()` member function of the file object. The flag 'w' is used as second argument to `open()` because the file is opened for writing.

3 Calling the Solver

Once we have created our project variations we can start the solver for each of the variations (or several in parallel). The simplest possibility is to run one solver at a time and wait until it is finished. For multi-processor machines it may be useful to run several tasks in parallel.

3.1 Running one Solver at a Time

Running an external program is easy. All that is really necessary is to add the line below to the script in listing 6 (mind the indentation).

```
retcode = subprocess.call([delphin_executable, "-x", "-v0", filename + '.dpj'])
```

The function `call()` requires a list as argument. The first item in the list is always the path to the executable whereas the following items are treated as command line arguments.

The arguments -x and -v0 are regular command line arguments of the DELPHIN solver. The first ensures that the solver returns the control to the calling process, after the simulation is done (and does not wait for a key press by the user), and the -v0 argument sets the verbosity level to 0 (very little output). The last argument in the path to the project file.

Listing 7: Parallel Solver Run Script

```
1  # import the module for calling external programs (creating subprocesses)
2  import subprocess
3  # import the module for multi-threading
4  import threading,Queue
5
6  # specify the path to the DELPHIN solver executable
7  delphin_executable = r'C:\Programme\IBK\Delphin 5.6\delphin_solver.exe'
8
9  # here we define our solver thread function
10 def solver_runner():
11     """This function runs a DELPHIN solver."""
12
13     # run as long as we have filenames in the Queue
14     while 1:
15         # get a new task from the Queue
16         filename = q.get()
17         print 'Running: ' + filename
18         retcode = subprocess.call([delphin_executable, "-x", "-v0", filename])
19         # tell the Queue that the task was successfully done
20         q.task_done()
21
22 # define number of threads to use
23 N_THREADS = 4
24 # create a queue for task scheduling
25 q = Queue.Queue()
26
27 # now create as many threads as we need
28 for i in range(N_THREADS):
29     t = threading.Thread(target=solver_runner)
30     t.daemon = True
31     t.start()
32
33 # now add project files to the queue, one per processor
34 q.put('project_1.dpj')
35 q.put('project_2.dpj')
36 q.put('project_3.dpj')
37 q.put('project_4.dpj')
38
39 # wait until all tasks are done
40 q.join()
```

3.2 Running Several Solver Tasks in Parallel

Running several tasks at the same time requires multi-threaded programming. Listing 7 shows a complete example for such a script.

In line 4 we import two more modules. In lines 10..20 we have the definition of the function `solver_runner()`. The function doesn't take any arguments but relies on two global variables: *q* and *delphin_solver*. These global variables must be defined when the function is first used (in line 29). The function is fairly simple. It stays in an endless loop that is only terminated by the main thread. In line 16 it takes a new filename from the queue, prints a message and then spawns a child process that runs the solver. Once the solver is done it tells the queue (in line 20) that the item was successfully processed and then goes on to fetch the next filename.

Line 23 is a simple definition of a variable and in line 25 the variable *q* is defined as Queue. In lines 28 to 31 all thread objects are created. Each thread object is told to use the function `solver_runner()` (line 29). In line 30 the thread daemon is turned on, just before the thread itself is started (line 31).

The rest of the script populates the queue with project file names (lines 34 to 37) and the final call to the `join()` member function of the queue tells the main thread to wait until all child threads

have finished and the queue is empty.

Basically, by using this script and simply modifying lines 34 to 37, or adding more jobs to the queue, you have a very effective replacement for the old-fashioned batch files.

Note: When running several threads in a single console window the outputs of the different threads are mangled together and the output can become confusing. It is best to look at the respective solver log files to check the progress of individual simulations.

3.3 Monitoring the Outcome of a Simulation

For automatic processing of the simulation outputs it is useful to know if a simulation completed successfully or if it failed. The return code of the DELPHIN solver can be used to find out whether the simulation has finished successfully or crashed. For this purpose we can use if-statements like in listing 8.

Listing 8: Using If–Statements to Check for Successful Calculation

```
1 retcode = subprocess.call([delphin_executable, "-x", "-v0", filename + '.dpj'])
2 # check for return code of 1 or 0
3 if retcode != 0:
4     print 'Calculation error'
5 else:
6     print 'Calculation completed successfully'
```

Again, the only tricky thing with this statement is the colon at the end of lines 2 and 4, which tells Python to expect an indented block of lines.

4 Extracting Sensitivity Parameters

Now that we can create project variations, and start the solver and check that the solver run was successfully completed, we need to extract all relevant information.

Normally what we want to get is some sensitivity quantity. Using the output formats available in DELPHIN most of the sensitivity parameters can be directly obtained. However, sometimes we want to get something more specific, like the minimum or maximum value or a certain quantity that was calculated in one year. For instance, we want to know the minimal temperatures at the inside surface of the wall and the maximum relative humidity.

First we need to read in the output files. Then we perform the analysis. And finally we compose an output file that contains the variable of interest and the modified parameters. Listing 9 shows the complete single-process version of the script so far.

Compared to the previous code snippets we find a few interesting new Python features in the code. For instance, in lines 18 and 19 we create the file with the variation study results and write the header. The '`\t`' and '`\n`' represent tab and linefeed characters. In line 44 we use the `continue` statement to skip the output file processing in case that a simulation failed. In lines 49..52 we open and read the output file just like we did in section 2.3 for the base variant of the project file.

Line 55 shows a special range operation that can be done with Python lists. Basically, the operator '[i:j]' tells Python to extract the range of items from i to j and return a new list object with that data. Omitting the second parameter j means that all lines beginning with line i should be

Listing 9: Complete Single-Process Variation Study Script

```
# import the module for calling external programs (creating subprocesses)
import subprocess

# specify the path to the DELPHIN solver executable
delphin_executable = r'C:\Programme\IBK\Delphin 5.6\delphin_solver.exe'

# print out the path to the executable
print 'Using: ' + delphin_executable

basefile_name = 'hamstad_benchmark_4'
basefile_obj = open(basefile_name + '.dpj', 'r')
basefile = basefile_obj.readlines()
del basefile_obj

N = 10 # define the number of variation steps

# open file for result
resultfile_obj = open(basefile_name + '_variation_results.txt', 'w')
resultfile_obj.write('Alpha\t Beta\t Min. Temperature\n')

for i in range(N+1):
    alpha = 8 + (20-8)*float(i)/N
    beta = (0.3 + (1.6-0.3)*float(i)/N)*1e-7
    filename = basefile_name + '_%02d' % i
    print 'Creating: ' + filename

    # exchange line with alpha parameter
    basefile[173] = '      EXCOEFF                    = %g W/m2K\n' % alpha
    # exchange line with beta parameter
    basefile[180] = '      EXCOEFF                    = %g s/m\n' % beta

    # don't forget to compose a new output folder
    folder = filename + '.results'
    basefile[225] = '    OUTPUT_FOLDER            = ' + folder + '\n'

    fname_dpj = filename + '.dpj'
    fileobj = open(fname_dpj, 'w')
    fileobj.writelines(basefile)
    del fileobj

    retcode = subprocess.call([delphin_executable, "-x", "-v0", fname_dpj])
    if retcode != 0:
        print '\nCalculation error'
        continue
    else:
        print '\nCalculation completed successfully'

    # open and read output file
    output_filename = folder + '\\temp_inside.out'
    outfile_obj = open(output_filename, 'r')
    outdata_T = outfile_obj.readlines()
    del outfile_obj

    # strip header information (we know how it looks like)
    outdata_T = outdata_T[13:]

    # extract all values
    values = []
    for line in outdata_T:
        nums = line.split()
        values.append( float(nums[1]) )

    # perform analysis and write results
    minval = min(values)
    resultfile_obj.write( '%g\t %g\t %g\n' % (alpha, beta, minval) )
    # flush buffered output (so that we can monitor the progress from outside)
    resultfile_obj.flush()

# close result file
del resultfile_obj
```

returned. DELPHIN output file headers of 2D data files have always 13 lines before the actual data starts.

Line 60 and 61 show one possible way of extracting data from each line in the output file. In this case, the data to parse looks like:

```
0               20
1               20.0313189
2               19.6297104
3               19.1840687
4               18.858877
5               18.6324045
...
```

The `split()` member function of string objects creates a list of strings and uses whitespaces (space, tabs, etc.) as separation characters. The `float()` function used in line 61 converts a string to a floating point number.

In line 64 one of the algorithms provided by Python is used. The `max()` or `min()` algorithms return the maximum or minimum value in a range.

Finally, in line 65 the modified parameters and the computed minimum temperature are written to the results file. Here, again the string format operation '%' is used. However, since we insert several numbers into the string, we need to use a Python tuple, a list of arguments enclosed in parenthesis. The member function `flush()` of the file object is used to write the new line directly to file. Normally, file output is written to a memory buffer and then written in larger chunks to file. However, it might be more useful to see the progress of the computation right away.

Of course, when used for a large number of variations, the multi-processor version is to be preferred. Listing 10 shows the corresponding version of the script for multi-processor usage.

Listing 10: Complete Multi-Process Variation Study Script

```
1  # import the module for calling external programs (creating subprocesses)
2  import subprocess
3  # import the module for multi-threading
4  import threading,Queue
5
6  # specify the path to the DELPHIN solver executable
7  delphin_executable = r'C:\Programme\IBK\Delphin 5.6\delphin_solver.exe'
8
9  # print out the path
10 print 'Using: ' + delphin_executable
11
12 basefile_name = 'hamstad_benchmark_4'
13 basefile_obj = open(basefile_name + '.dpj', 'r')
14 basefile = basefile_obj.readlines()
15 del basefile_obj
16
17 N = 40 # define the number of variation steps
18
19 # create array to contain the filenames of the created projects
20 filenames = []
21 # create arrays for modified parameters (only remembered for output later)
22 alphas = []
23 betas = []
24
25 # create project variations
26 for i in range(N+1):
27     alpha = 8 + (20-8)*float(i)/N
28     beta = (0.3 + (1.6-0.3)*float(i)/N)*1e-7
```

```
29      filename = basefile_name + '_%02d' % i
30      print 'Creating: ' + filename
31
32      # exchange line with alpha parameter
33      basefile[173] = '      EXCOEFF                    = %g W/m2K\n' % alpha
34      # exchange line with beta parameter
35      basefile[180] = '      EXCOEFF                    = %g s/m\n' % beta
36
37      # don't forget to compose a new output folder
38      folder = filename + '.results'
39      basefile[225] = '    OUTPUT_FOLDER             = ' + folder + '\n'
40
41      fileobj = open(filename + '.dpj', 'w')
42      fileobj.writelines(basefile)
43      del fileobj
44
45      # store filename and parameters
46      filenames.append(filename)
47      alphas.append(alpha)
48      betas.append(beta)
49
50
51  # create array with run results
52  run_results = []
53
54  # here we define our solver runner function
55  def solver_runner():
56      """This function runs a DELPHIN solver."""
57
58      # run as long as we have filenames in the Queue
59      while 1:
60          # get a new task from the Queue
61          job = q.get()
62          index = job[0]
63          filename = job[1]
64          print 'Running: ' + filename + '.dpj'
65          retcode = subprocess.call( \
66              [delphin_executable, "-x", "-v0", filename + '.dpj'])
67          # if run was successful, remember project file for analysis
68          if retcode == 0:
69                  run_results.append( job )
70          # tell the Queue that the task was successfully done
71          q.task_done()
72
73  q = Queue.Queue()
74
75  # add threads
76  N_threads = 4
77  for i in range(N_threads):
78      t = threading.Thread(target=solver_runner)
79      t.setDaemon(True)
80      t.start()
81
82  # add the filenames,alphas and betas to the queue
83  for i in range(len(filenames)):
84      q.put( [i, filenames[i]] )
85
86  # wait until all tasks/filenames in the queue have been processed
87  q.join()
88
89  # now start the analysis of the
90  # first sort the values
91  run_results.sort()
92
93  resultfile_obj = open(basefile_name + '_variation_results.txt', 'w')
94  resultfile_obj.write('Alpha\t Beta\t Min. Temperature\n')
95  for job in run_results:
96      index = job[0]
97      filename = job[1]
98      print 'Processing: ' + filename
99      folder = filename + '.results'
100     # open and read output file
```

```
101     output_filename = folder + '\\temp_inside.out'
102     outfile_obj = open(output_filename, 'r')
103     outdata_T = outfile_obj.readlines()
104     del outfile_obj
105
106     # strip header information (we know how it looks like)
107     outdata_T = outdata_T[13:]
108
109     # extract all values
110     values = []
111     for line in outdata_T:
112         nums = line.split()
113         values.append( float(nums[1]) )
114
115     # perform analysis and write results
116     minval = min(values)
117     resultfile_obj.write( '%g\t %g\t %g\n' % (alphas[index], betas[index], minval) )
118     # flush buffered output (so that we can monitor the progress from outside)
119     resultfile_obj.flush()
120
121 # close result file
122 del resultfile_obj
```

5 Summary

The Python scripts developed in this tutorial are fairly simple, yet already quite powerful. Within few days of learning Python it is possible to create already advanced scripts. Also, a huge library of functions and classes is at your disposal, among those a number of scientific and math libraries (e.g. for random number distribution and statistical analysis).

But is Python really a good choice for this task? Of course, you could always use a 'real' compiler language like C++ to get the job done. But already when it comes to multi-threaded programming, the C++ standard library isn't enough (you need at least the Boost extension). Other alternatives would be Ruby and of course Bash and Perl. But the biggest plus for using Python for scripting and automation purposes is its simplicity, clear language layout and quick learning curve.

But judge for yourself and take a look at listing 11 which is the C++ version of listing 9. It is a bit longer, and you need to learn already a lot of C++ library code and constructs to understand and reproduce this code from scratch.

Listing 11: Complete Single-Process Variation Study Program in C++

```
1  #include <iostream>
2  #include <iomanip>
3  #include <vector>
4  #include <sstream>
5  #include <fstream>
6  #include <algorithm>
7  #include <string>
8  #include <iterator>
9  using namespace std;
10
11 int main() {
12         // specify the path to the DELPHIN solver executable
13         string delphin_executable =
14                 "C:\\Programme\\IBK\\Delphin 5.6\\delphin_solver.exe";
15
16         // print out the path to the executable
17         cout << "Using: " << delphin_executable << endl;
18
19         string basefile_name = "hamstad_benchmark_4";
20         ifstream basefile_obj((basefile_name + ".dpj").c_str());
21         vector<string> basefile;
```

```
22          string line;
23          while (getline(basefile_obj,line))
24                  basefile.push_back(line);
25          basefile_obj.close(); // cleanup is done later
26
27          const int N = 10; // define the number of variation steps
28
29          // open file for result
30          ofstream resultfile_obj((basefile_name + "_variation_results.txt").c_str());
31          resultfile_obj << "Alpha\t Beta\t Min. Temperature\n";
32
33          for (int i=0; i< N+1; ++i) {
34                  double alpha = 8 + (20-8)*double(i)/N;
35                  double beta = (0.3 + (1.6-0.3)*double(i)/N)*1e-7;
36                  stringstream filename_strm;
37                  filename_strm << basefile_name << "_" << setw(2) << setfill('0') << i;
38                  string filename = filename_strm.str();
39                  cout << "Creating: " << filename << endl;
40
41                  //  exchange line with alpha parameter
42                  stringstream lstrm1;
43                  lstrm1 << "      EXCOEFF                  = " << alpha << " W/m2K";
44                  basefile[173] = lstrm1.str();
45                  // exchange line with beta parameter
46                  stringstream lstrm2;
47                  lstrm2 << "      EXCOEFF                  = " << beta << " s/m";
48                  basefile[180] = lstrm2.str();
49
50                  // don't forget to compose a new output folder
51                  string folder = filename + ".results";
52                  basefile[225] = "    OUTPUT_FOLDER            = " + folder;
53
54                  string fname_dpj = filename + ".dpj";
55                  ofstream fileobj(fname_dpj.c_str());
56                  copy(basefile.begin(), basefile.end(),
57                          ostream_iterator<string>(fileobj, "\n"));
58                  fileobj.close();
59
60                  stringstream cmdline;
61                  cmdline << "\"\"" << delphin_executable << "\" -x -v0 \""
62                                  << fname_dpj << "\"\"";
63                  int retcode = system(cmdline.str().c_str());
64                  if (retcode != 0) {
65                          cerr << "\nCalculation error" << endl;
66                          continue;
67                  }
68                  else {
69                          cout << "\nCalculation completed successfully" << endl;
70                  }
71
72                  // open and read output file
73                  string output_filename = folder + "\\temp_inside.out";
74                  ifstream outfile_obj(output_filename.c_str());
75                  vector<string> outdata_T;
76                  while (getline(outfile_obj,line))
77                          outdata_T.push_back(line);
78                  outfile_obj.close();
79
80                  // strip header information (we know how it looks like)
81                  if (outdata_T.size() < 13)
82                          continue; // safety check to avoid access violation
83                  outdata_T.erase(outdata_T.begin(), outdata_T.begin() + 13);
84
85                  // extract all values
86                  vector<double> values;
87                  for (vector<string>::const_iterator line_it = outdata_T.begin();
88                          line_it != outdata_T.end(); ++line_it)
89                  {
90                          stringstream lstrm(*line_it);
91                          double tp, val;
92                          lstrm >> tp >> val;
93                          values.push_back(val);
```

```
94                  }
95
96                  // perform analysis and write results
97                  double minval = *min_element(values.begin(), values.end());
98                  stringstream outputstrm;
99                  outputstrm << alpha << "\t " << beta << "\t " << minval << endl;
100                 resultfile_obj << outputstrm.str() << endl;
101                 // flush buffered output
102                 resultfile_obj.flush();
103         }
104
105         // close result file
106         resultfile_obj.close();
107
108         return EXIT_SUCCESS;
109 }
```

Listings

1 Python test script 2

2 DELPHIN Project File 3

3 Calculating Variation Parameters 4

4 Reading the Base Variant of the Project 4

5 Composing Filenames 5

6 Complete Script to Create Project File Variations 5

7 Parallel Solver Run Script 7

8 Using If–Statements to Check for Successful Calculation 8

9 Complete Single-Process Variation Study Script 9

10 Complete Multi-Process Variation Study Script 10

11 Complete Single-Process Variation Study Program in C++ 12