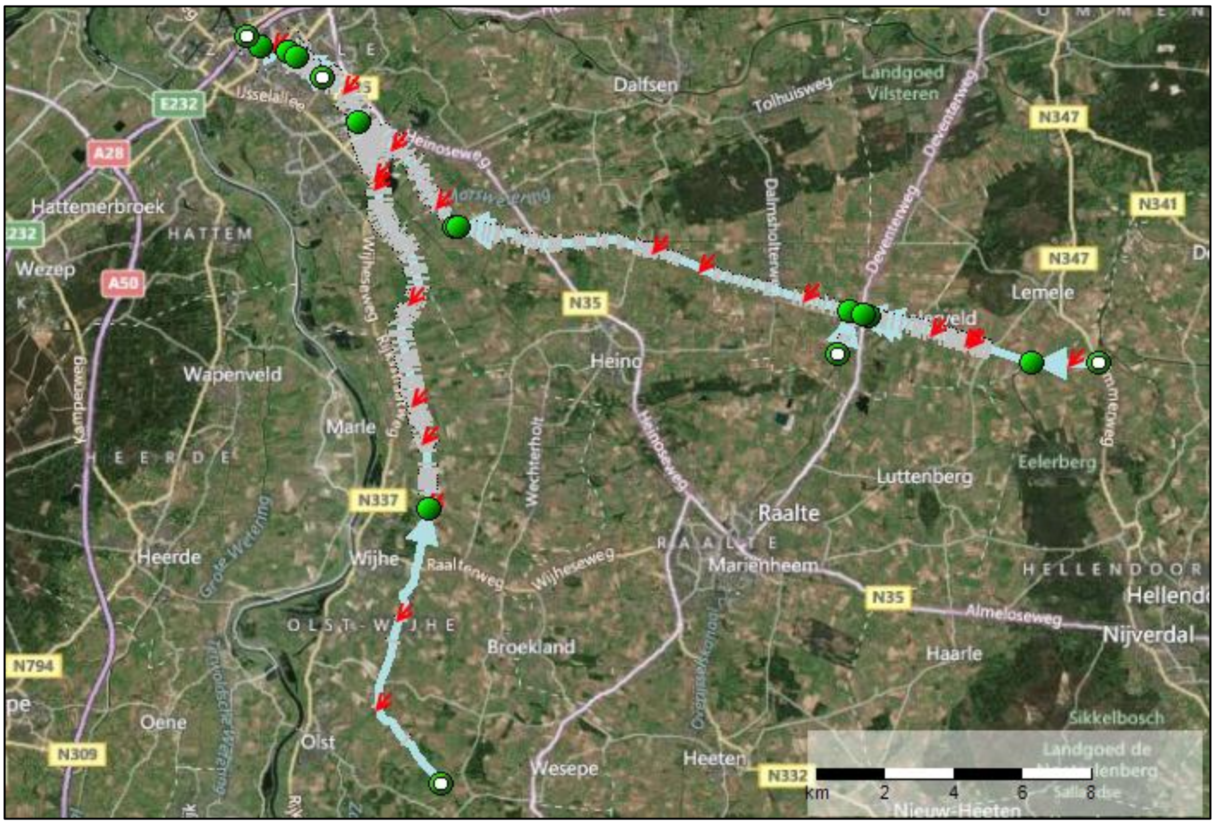
Calibration of a flow (SOBEK) model using scripting

We have a hydrodynamic flow model, which has been built with Sobek 2.12, of the fluvial system around Zwolle. The model has been completely set up, except for the Boundary condition at one of the tributaries, at Boxbergen, and the roughness of both main tributaries. We also have a time series of water level measurements at the entrance of the channels which flow across the center of Zwolle. Using this information, build a script to automatically calibrate the model.



The workflow of this tutorial has been divided into the following steps:

1) Import required libraries.
2) Import current model and measured data.
3) Compare current results with measurements.
4) Take objects that will be used for model calibration.
5) Loop running model for different parameters combinations.
6) Add to the previous plot the results of these runs.
7) Calculate goodness of different cases and select the best fit.
8) Export results.

1 Import required libraries

Start Delta Shell (DS) by double clicking on the corresponding icon on the desktop. First, the current scripting folder needs to be changed to a separate one from the default value. Right click on the toolbox icon of the corresponding panel, and using the context menu change the scripting directory to D:\scripting. The panel will be refreshed showing the folders and scripts contained in the new location.

Scripts inside DS use IronPython programming language. This is a variety of the Pyhton programming language strongly integrated with the .NET Framework. All the scripts can have access to all classes and their respective members inside the framework of the .NET environment, as well as those of Delta Shell. Initially, however, they are not loaded into the interpreter. They need to be explicitly imported.

Open the script named *SobekScriptStart* (the extension, *.py, is hidden by the interface of DS). Some imports have already been included in this file. Others will be added during the present tutorial as they are required. In some cases, all members (attributes) are imported from a model (by using an asterisk), in some other cases, only some of them.

```
# python libs

# .Net libs
from System import DateTime, Double
from System.Collections.Generic import List

# DeltaShell libs
from ??? import Map
from ??? import ActivityRunner
from ??? import CsvFunctionExporter
```

The members ***Map*** (to assign a coordinate system to a model, and load a Bing map as background), ***ActivityRunner*** (to run a model) and ***CsvFunctionExporter*** (which imports and exports, among others, time series from a csv file) need to be imported for later use. Find in which namespaces those members can be found by looking for those terms in the scripting help (F1). Then replace the triple questions marks with those namespaces.

For the time being, nothing else will be modified at this part of the code. If you want, you can collapse the region to help you continue the scripting procedure more easily.

2 Import current model and measured data

Both the non-calibrated model and the measurement data are located in one same folder. We will assign that folder name to a variable:

```
rootPath = "D:\\scripting\\"
```

The water level has been measured at the entrance of the channels in the center of Zwolle (node name is **3**). The data can be found in the file *D:\scripting\waterLevel_at_3_Measured.csv*

In the first place, load this data. A very complete importer for comma separated value (csv) files containing time series has been implemented inside DS. However, not all this functionality is required now. The script *SobekWorkshopHelperFunctions* contains an adaptation of that generic importer to the specific situation dealt with. Open the above mentioned script, and take a look at the code. We will have to use this, and all the other helper functions contained in this file, in the script we are developing. Therefore, you need to add a line to import all attributes from this script file. Set the focus again in the editor view for the script being developed, *SobekScript*. Then, right click on the icon for the script file *SobekWorkshopHelperFunctions*, and select the option *Add as import*. The following line will be added at the top of the script.

```
from SobekWorkshopHelperFunctions import *
```

Place this line of code inside the region import libraries, under the comment line for python libraries. If desired, you can collapse again the region to keep the code tidy.

Add the following lines to import the csv time series for the file:

```
# Import of csv (station data)
csvPath = rootPath + "waterLevel_at_3_Measured.csv"
measuredTimeSeries = ImportCsvTimeSeries(csvPath,"Time","Value","dd-MMM-yy HH:mm:ss")
```

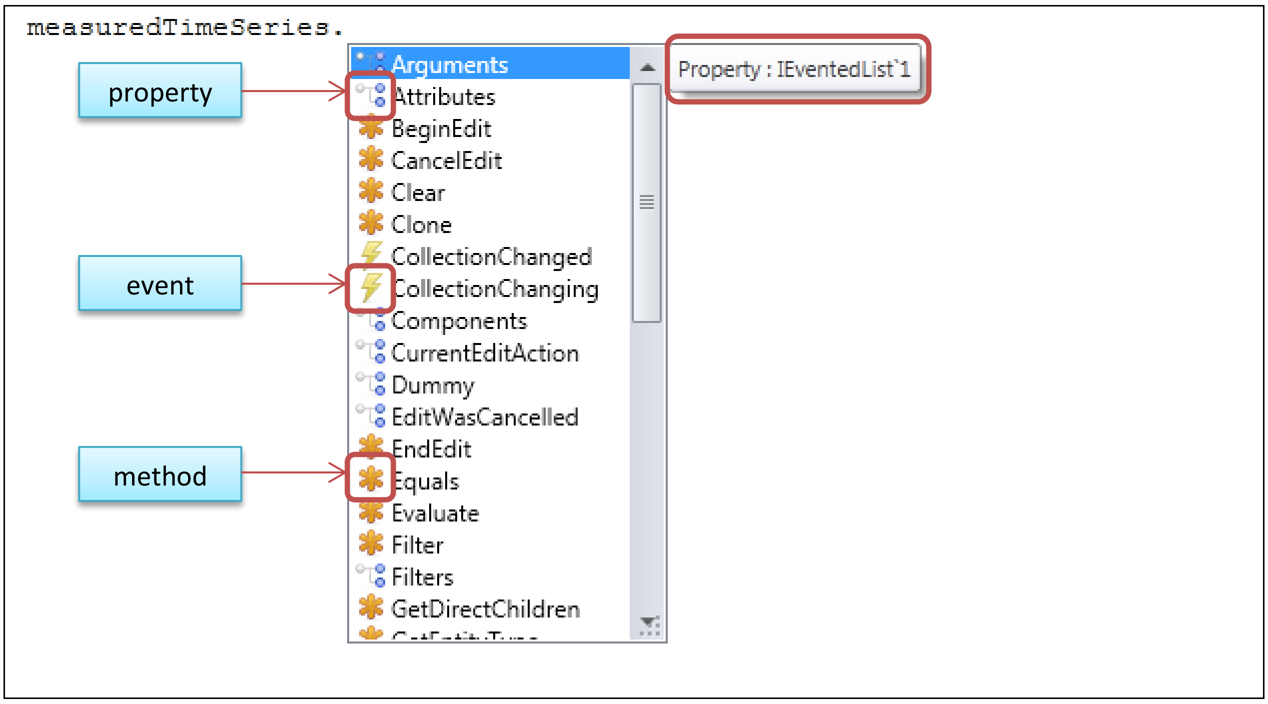
Run all the code written in the script so far, starting with the imports region. This will enable new capabilities of the scripting editor view, like the autocomplete functionality.

Let's give a new name to the previously imported time series. Start typing the name of the variable:

```
meas
```

Then, press CTRL + Space bar. A context menu will be shown containing all the possibilities which contain the same letter combination. In this case, there is only one. So simple press ENTER and the variable name will be automatically completed.

If you now type a dot, a context menu will be automatically presented, showing all the members of that object. All these members have been defined in the source code of Delta Shell. Each member is presented after an icon which indicates its kind (property, event or method). Additionally, the type of the currently selected one is also indicated in a tooltip. In the case of methods, also the arguments and their respective types are indicated.


```
measuredTimeSeries.
```

In order to identify the different time series when they are plotted, a different informative legend will be assigned to each of them. This will easily help identify every line in the final chart. Select the property *Components*. Its first element (number zero) contains the property *Name*. Assign a new string to the name, for example “Measured time series”.

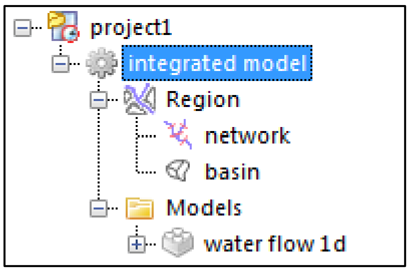
```
measuredTimeSeries.Components[0].Name = "Measured time series"
```

The available SOBEK 2.1x model can be found in the path D:\scripting\SW_max.lit\2\. Assign this value to a new variable *modelPath*. Use for this the previously created variable *rootPath*, and add the missing folders. Remember to escape the backslash character, as shown above in the assignment of *rootPath*.

The SOBEK 2.1x model will be imported using the *ImportSobek2Model* function defined in the *SobekWorkshopHelperFunctions* library. From that entire model, only the flow sub model will be imported. The other parts (water quality, rainfall-runoff, and real time control) will be omitted. Use the information contained in that python script to complete the import command:

```
model = ImportSobek2Model(modelPath + "NETWORK.TP", , , ,)
```

The model has now been loaded into memory, but it is not shown yet in the DS GUI. This is, strictly speaking, not necessary to continue working with it. However, it can be handy to have it accessible not only from the scripting environment but also directly in the GUI. Use the *Add* method of the *RootFolder* object to attach the object *model* to the project object in the project explorer panel. Once you have run it, the model will be added to the project and shown in the GUI.



Assign the flow model to a new variable *waterFlowModel*, by using the function *GetItemByName* contained in the *SobekWorkshopHelperFunctions* library. The list where that object can be found contains all the sub-models of the model variable (*model.Models*). The name of the flow model is, as shown in the previous figure, “water flow 1d”.

Next, add a coordinate system to the water flow model previously assigned:

```
cs = Map.CoordinateSystemFactory.CreateFromEPSG(28992) #RD new
waterFlowModel.Network.CoordinateSystem = cs
```

Actually, the previous code can be combined into one single line. Finally, open the view with a Bing map as background, by using the function defined in the help library:

```
OpenModelMapViewWithBackground(model)
```

3 Compare current results with measurements

Run the flow model to obtain the results with the current configuration:

```
ActivityRunner.RunActivity(waterFlowModel)
```

The previously loaded time-series was measured at point where the Almelose Kanaal forks into three different branches in the center of Zwolle. Zoom in the map to that point and select the node. In the properties panel, you will see its name, “3”, as indicated previously. Assign that calculation point object named “3” to a new variable *calculationPoint*. You can do this with the *GetItemByName* function previously used. Now, the list inside which to search is the set containing the values at all locations:

```
waterFlowModel.NetworkDiscretization.Locations.Values
```

Next, obtain the time series at that location, and set it to the variable *timeSeriesBase*:

```
timeSeriesBase = waterFlowModel.OutputWaterLevel.GetTimeSeries(calculationPoint)
```

Change the name of this time series to “Base time series” as done before with the imported one.

Now, create an empty list that will contain all the time series result of this exercise, so that they can be easily compared the imported time series.

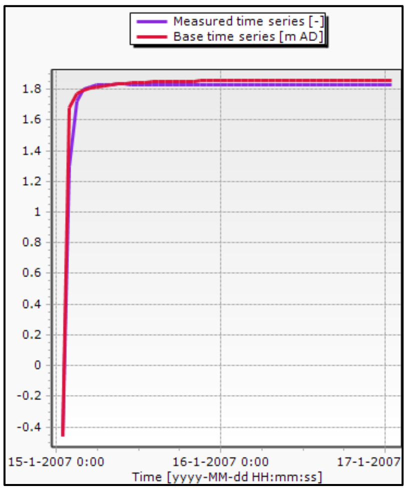
```
# Create time series list containing all timeseries to compare
timeSeriesToCompare = List[TimeSeries]()
```

The first two time series that will be added are the imported one, and the one obtained from the model with the initial configuration as just imported.

Use the Add method of the list of time series just created with each of the above mentioned series to include them.

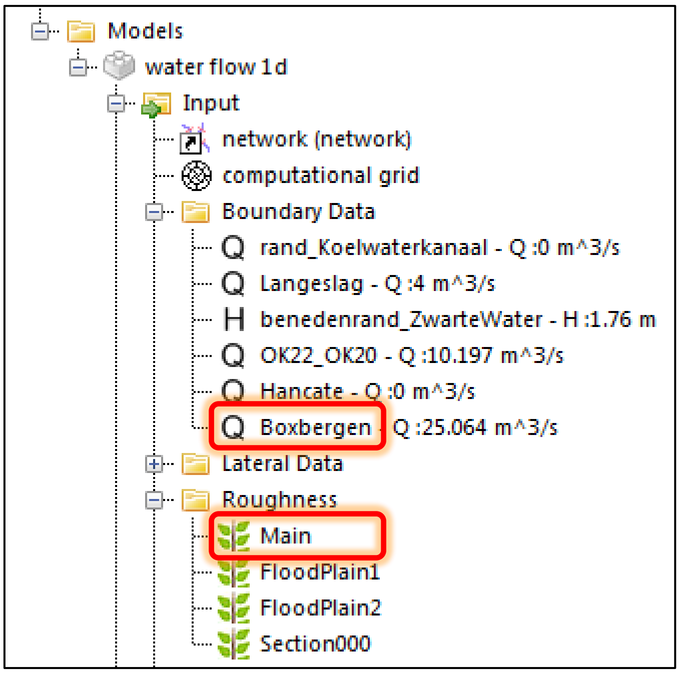
At this time, you can plot both time series to see how the results do not agree with the measurements obtained at the calculation point used for model calibration.

```
Gui.DocumentViewsResolver.OpenViewForData(timeSeriesToCompare)
```



4 Take objects that will be used for model calibration

The model calibration will be done by changing the less reliable parameters from the loaded model. Let's assume that these factors are the flow boundary condition at the node Boxbergen, and the roughness of the main tributaries leading to Zwolle:



- Models
  - water flow 1d
    - Input
      - network (network)
      - computational grid
      - Boundary Data
        - Q rand_Koelwaterkanaal - Q :0 m^3/s
        - Q Langeslag - Q :4 m^3/s
        - H benedenrand_ZwarteWater - H :1.76 m
        - Q OK22_OK20 - Q :10.197 m^3/s
        - Q Hancate - Q :0 m^3/s
        - Q Boxbergen - Q :25.064 m^3/s
      - Lateral Data
      - Roughness
        - Main
        - FloodPlain1
        - FloodPlain2
        - Section000

The definitions of boundary conditions object are not known yet in the scripting environment. The object *WaterFlowModel1DBoundaryNodeDataType* needs to be imported from the corresponding DS namespace. Use the online help to find which namespace it is, and add the corresponding line in the import libraries region.

The complete name of the flow boundary condition at Boxbergen is, as shown in the previous figure, composed by the name of the location and the current value. We will use the first part, the location name, to grab that object. Assign the beginning of the name of the BC to a new variable:

```
bc1NameBegin ="Boxbergen"
```

A *for* loop that goes along all the BC's in the flow model (object *waterFlowModel.BoundaryConditions*) can be used to pick up the one whose name starts with the previous string, and assign it to the variable bc1:

```
for bc in waterFlowModel.BoundaryConditions:
    if bc.Name.startswith(bc1NameBegin):
        bc1 = bc
```

You can save the initial boundary condition for later reference. Assign it to the variable *bc1Ini*. If you next type in:

```
bc1Ini = bc1.
```

a context menu will be automatically displayed next to the dot, showing all the members available for this boundary condition object. If you can't see the context menu, you can obtain it by pressing CTRL + Space bar. Then, select the Flow property, which corresponds to the current value of that boundary.

You can open the corresponding editor view by double clicking on the corresponding item of the project explorer panel. This will show the BC automatically changing while the script modifies its value in the next step.

The other object that will be used for tuning the model up is the roughness at the main tributaries. First of all, this object should be available in the scripting environment. Look for the object *RoughnessType* in the online help, and add the corresponding import to the first region of the script.

The roughness at the main tributaries is the first object (so the index is [0]) in the list of roughness items, as shown in the picture above. The object to grab is the corresponding network coverage.

```
rghnsMain = waterFlowModel.RoughnessSections[0].RoughnessNetworkCoverage
```

You can save the current value for future reference, as done before with the BC.

```
rghnsIni = (rghnsMain.GetValues())[0]
```

Similarly, if you double click on the item in the project explorer panel, you will open the corresponding editor view and will be able to see how it automatically changes while the calibration loop is being run.

5 Loop running model for different parameters combinations

The possible values that the calibration parameters may have, can be indicated in different ways, for example, through their respective statistical distributions. However, for the sake of simplicity, in this tutorial, let's assume that only a limited amount of values is possible for each of them, and that the possible combinations of those cases are equally likely.

For the water flow boundary condition, we will assume that the possible values are 8, 10 and 12 m^3/s. For the roughness along the main tributaries, we will accept as possible values, using the roughness type Strickler coefficient k_s, 24, 26, 28, 30, 32 and 34 $m^{1/3}s^{-1}$, so a relatively rough river bed surface. Provide these values in both respective vectors:

rangeBC1 = [, ,]
rangeRoughness = [, , , , ,]

Now build the loop that will iterate along all possible combinations of the paramaters:

```
for bc1Value in rangeBC1:
    # change under this comment the property Flow of bc1 to the BC iterator bc1Value
    for rgnhVal in rangeRoughness:
        for k in rghnsMain.Locations.Values:
            rghnsMain[k] = [rgnhVal, RoughnessType.StricklerKs]
```

The second line needs to be modified to assign the BC a different value in the range each time that the loop executes that code. The loop of the last two lines above changes the roughness for each location inside the Main roughness item to the roughness iterator value *rghnVal*, while keeping the type as *StricklerKs*.

The code above, once the second line has been modified, only changes the parameter values but the model is not run. Add a line inside the second level of the loop, once the roughness has been changed, to run the flow model.

At this point, the model would be run for each case combination of both parameter ranges. However, the results after each run need to be saved, and added to the list *timeSeriesToCompare* created before. In order to do this, first assign the time series at the calculation point "3" to a new variable *timeSeries*. Then modify the name of this time series to a string indicating the exact case that series represents:

```
timeSeries.Components[0].Name = "bc1= " + str(bc1Value) + " rghness=" + str(rgnhVal)
```

Finally, add that time series to the collection *timeSeriesToCompare*, as done earlier in this tutorial. A list with all results at the observation point will then be contained in this collection. Just to check that everything went fine, you can print the amount of time series present in it:

```
print timeSeriesToCompare.Count
```

There should be 1 (measurements) + 1 (initial time series) + 3 x 6 (calibration cases) = 20 elements in total contained in the list.

6 Plot the results of the calibration runs together with the previous ones

The results obtained for the different model cases at the observation point can now be plotted together with the previously obtained for base model, and the imported data. To begin with, close the view plotting the former content of the object *timeSeriesToCompare*:

```
Gui.DocumentViewsResolver.CloseAllViewsFor(timeSeriesToCompare)
```

Next, open the view again. The graph will be generated with the current content of *timeSeriesToCompare*. In Delta Shell, there is a default maximum number of time series (10) which can be plotted simultaneously in a chart. You can easily override this limitation by first assigning the view object to a variable, then modifying the *MaxSeries* property to the number of time series to plot. Finally, refresh the view to take into account his modification:

```
Gui.DocumentViewsResolver.OpenViewForData(timeSeriesToCompare)
view = Gui.DocumentViews[Gui.DocumentViews.Count -1]
view.MaxSeries = timeSeriesToCompare.Count
view.RefreshChartView()
```

At the original scale, it may look like all lines fit very well, but this is due to the initial condition being very different from the later evolution of the water level. If you zoom in around the end of the simulation, the vertical scale will be modified and you will see the differences more clearly. This will be taken into account subsequently, when calculating the goodness of each simulation.

7 Calculate the goodness of the different cases and select the best fit

As just shown, a visual inspection can be misleading, and, anyway, is not an accurate method of comparison. A numerical algorithm is required. A function to subtract two lists has been implemented in the helper library. Take a look at it to learn how to use it. The parameter *startIndex*, which, if not explicitly indicated, is, by default, assumed to be zero.

Additionally, a function is required to estimate the magnitude of the total deviation. For example, you can add up all the differences in the vector, regardless their sign. Even in this case, there are several possibilities, for example to sum the squared elements of the difference lists, or to sum their absolute values, among others. Choose the option you prefer, and implement it as a new function which takes one list (of deviations) and returns one number indicating the magnitude of the vector. You can implement this auxiliary function in the helper library file or directly in the script you are developing. The first option is preferred.

One possibility would be:

```
def sumAbsValues(list):
    sum = 0
    for val in list:
        sum += abs(val)
    return sum
```

The goodness of a simulation will be based on the deviation from the measured time series. Get those values, and assign them to a variable, since they will be used repeatedly:

```
tsRef = measuredTimeSeries.GetValues()
```

Now, create a list in which the deviations of the results for each simulation with respect to the measurements will be saved. When a model is run, the initial conditions are predominant at the beginning of the simulation. Also, very frequently, numerical oscillations occur due to the inconsistency of the proposed initial conditions. To get rid of all this spurious data, select a number of time steps to be discarded at the beginning of each time series during the goodness calculation (warm-up period), and write a loop which iterates along all the results:

```
deviation =[]
startIndex = 20 #remove influence of initial conditions
for i in range(1,timeSeriesToCompare.Count):
    devItem =
sumAbsValues(list(substractLists(timeSeriesToCompare[i].GetValues(),tsRef,startIndex)
))
    deviation.append(devItem)
```

Notice that the first time series of *timeSeriesToCompare*, with index 0, has been omitted. Why is that?

The best fit will be given by the smallest deviation from the measurements.

```
bestFitDeviation = min(deviation)
bestFitIndex = deviation.index(bestFitDeviation)+1
```

Once the best fit has been found, the values for the calibration parameters corresponding to that case can be extracted:

```
if bestFitIndex == 1: #basis model
    bc1BestFit = bc1Ini
    roughnessBestFit = rghnsIni
else:
    idxBestBC = (bestFitIndex-1)/len(rangeRoughness)
    idxBestRoughness = (bestFitIndex-1)%len(rangeRoughness)
    bc1BestFit = rangeBC1[idxBestBC]
    roughnessBestFit = rangeRoughness[idxBestRoughness]
```

You can optionally print the results of the calibration:

```
print "selected bc: " + str(bc1BestFit) + "m3/s"
print "selected roughness " + str(roughnessBestFit) + "m^(1/3) s^-1"
print timeSeriesToCompare[bestFitIndex].Components[0].Name
```

8 Export results

Finally, all the time series obtained during the model calibration can be exported to external csv files for further analysis, if desired:

```
exporter = CsvFunctionExporter()
index = 0
for ts in timeSeriesToCompare:
    exporter.Export(ts, rootPath + "waterflow_at_3_" + str(index) + ".csv")
    index += 1
```

It is also possible to add a flag to the name of the files to indicate which one corresponds to the best fit:

```
exporter = CsvFunctionExporter()
index = 0
for ts in timeSeriesToCompare:
    if index == bestFitIndex:
        flagBF = " best fit "
    else:
        flagBF = " "
    fileName = rootPath + "waterflow_at_3_" + str(index) + flagBF + ".csv"
    print fileName
    exporter.Export(ts, fileName)
    index += 1
```