

Hapi Documentation

Mostafa Farrag

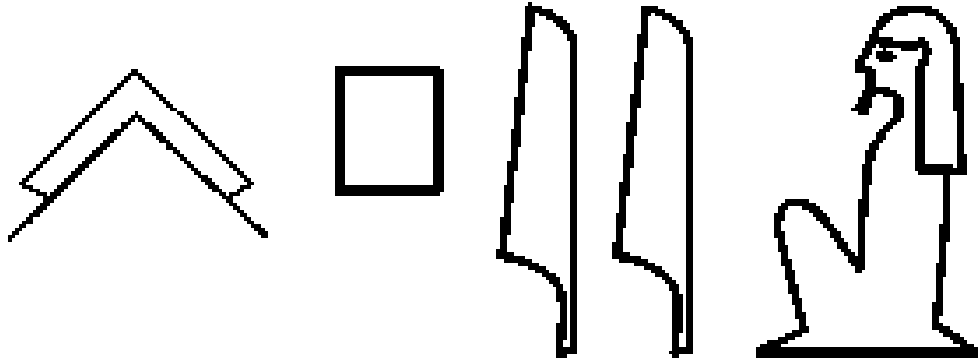
Feb 02, 2023

CONTENTS

1	Hapi - Hydrological library for Python	2
2	References	5

HAPI - HYDROLOGICAL LIBRARY FOR PYTHON





Hapi is a Python package providing a fast and flexible way to build Hydrological models with different spatial representations (lumped, semidistributed and conceptual distributed) using HBV96. The package is very flexible to an extent that it allows developers to change the structure of the defined conceptual model or to enter their own model, it contains two routing functions muskingum cunge, and MAXBAS triangular function.

1.1 Main Features

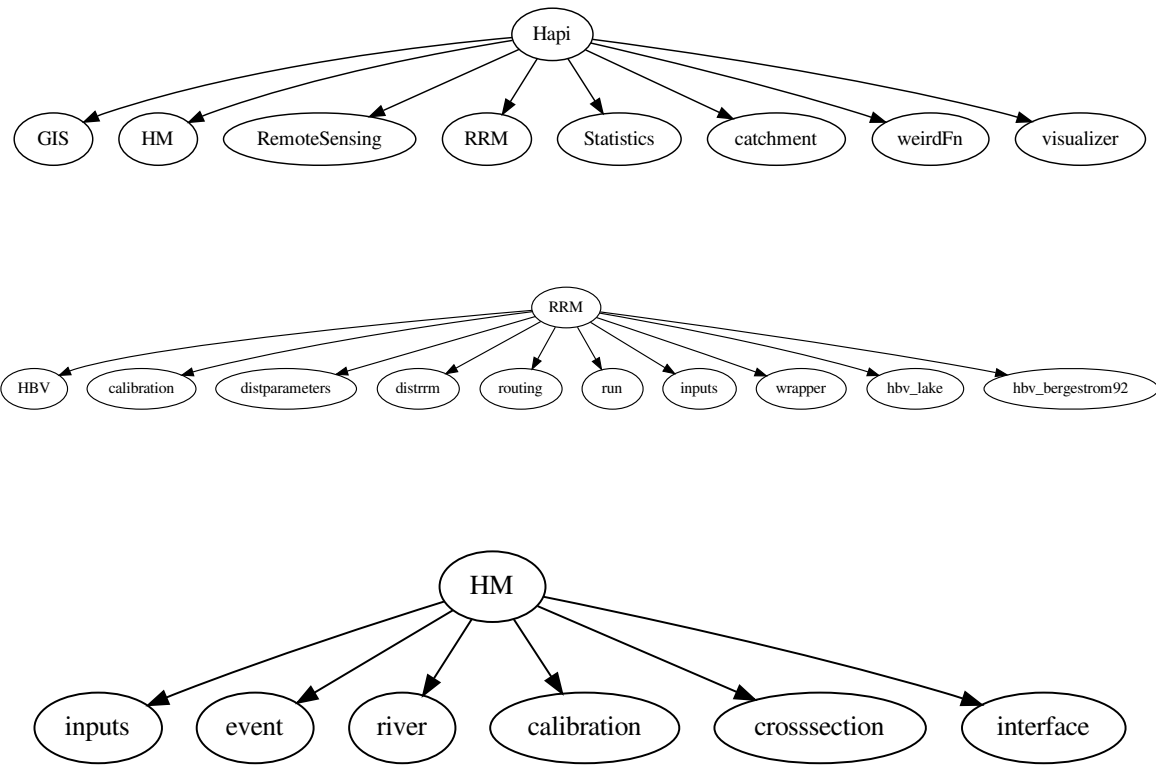
- Modified version of HBV96 hydrological model (Bergström, 1992) with 15 parameters in case of considering snow processes, and 10 parameters without snow, in addition to 2 parameters of Muskingum routing method
- Remote sensing module to download the meteorological inputs required for the hydrologic model simulation (ECMWF)
- GIS modules to enable the modeler to fully prepare the meteorological inputs and do all the preprocessing needed to build the model (align rasters with the DEM), in addition to various methods to manipulate and convert different forms of distributed data (rasters, NetCDF, shapefiles)
- Sensitivity analysis module based on the concept of one-at-a-time OAT and analysis of the interaction among model parameters using the Sobol concept ((Rusli et al., 2015)) and a visualization
- Statistical module containing interpolation methods for generating distributed data from gauge data, some distribution for frequency analysis and Maximum likelihood method for distribution parameter estimation.
- Visualization module for animating the results of the distributed model, and the meteorological inputs
- Optimization module, for calibrating the model based on the Harmony search method

The recent version of Hapi (Hapi 1.0.1 Farrag et al. (2021)) integrates the global hydrological parameters obtained by Beck et al., (2016), to reduce model complexity and uncertainty of parameters.

For using Hapi please cite Farrag et al. (2021) and Farrag & Corzo (2021) References

1.2 IHE-Delft sessions

- In April 14-15 we had a two days session for Masters and PhD student in IHE-Delft to explain the different modules and the distributed hydrological model in Hapi [Day 1](<https://youtu.be/HbmUdN9ehSo>) , [Day 2](<https://youtu.be/m7kHdOFQFIY>)



1.3 Future work

- Developing a regionalization method for connection model parameters with some catchment characteristics for better model calibration.
- Developing and integrate river routing method (kinematic and diffusive wave approximation)
- Apply the model for large scale (regional/continental) cases
- Developing a DEM processing module for generating the river network at different DEM spatial resolutions.

REFERENCES

Farrag, M. & Corzo, G. (2021) MAfarrag/Hapi: Hapi. doi:10.5281/ZENODO.4662170

Farrag, M., Perez, G. C. & Solomatine, D. (2021) Spatio-Temporal Hydrological Model Structure and Parametrization Analysis. *J. Mar. Sci. Eng.* 9(5), 467. doi:10.3390/jmse9050467

Beck, H. E., Dijk, A. I. J. M. van, Ad de Roo, Diego G. Miralles, T. R. M. & Jaap Schellekens, and L. A. B. (2016) Global-scale regionalization of hydrologic model parameters-Supporting materials 3599–3622. doi:10.1002/2015WR018247.Received

Bergström, S. (1992) The HBV model - its structure and applications. *Smhi Rh* 4(4), 35.

Rusli, S. R., Yudianto, D. & Liu, J. tao. (2015) Effects of temporal variability on HBV model calibration. *Water Sci. Eng.* 8(4), 291–300. Elsevier Ltd. doi:10.1016/j.wse.2015.12.002

2.1 Installation

2.1.1 Stable release

Please install Hapi in a Virtual environment so that its requirements don't tamper with your system's python Hapi works with all Python versions

2.1.2 conda

the easiest way to install Hapi is using conda package manager. Hapi is available in the [conda-forge](#) channel. To install you can use the following command:

- `conda install -c conda-forge hapi`

If this works it will install Hapi with all dependencies including Python and gdal, and you skip the rest of the installation instructions.

2.1.3 Installing Python and gdal dependencies

The main dependencies for Hapi are an installation of Python 2.7+, and gdal

2.1.4 Installing Python

For Python we recommend using the Anaconda Distribution for Python 3, which is available for download from <https://www.anaconda.com/download/>. The installer gives the option to add `python` to your `PATH` environment variable. We will assume in the instructions below that it is available in the path, such that `python`, `pip`, and `conda` are all available from the command line.

Note that there is no hard requirement specifically for Anaconda's Python, but often it makes installation of required dependencies easier using the `conda` package manager.

2.1.5 Install as a conda environment

The easiest and most robust way to install Hapi is by installing it in a separate conda environment. In the root repository directory there is an `environment.yml` file. This file lists all dependencies. Either use the `environment.yml` file from the master branch (please note that the master branch can change rapidly and break functionality without warning), or from one of the releases `{release}`.

Run this command to start installing all Hapi dependencies:

- `conda env create -f environment.yml`

This creates a new environment with the name `hapi`. To activate this environment in a session, run:

- `activate hapi`

For the installation of Hapi there are two options (from the Python Package Index (PyPI) or from Github). To install a release of Hapi from the PyPI (available from release 2018.1):

- `pip install HAPI-Nile=={release}`

2.1.6 From sources

The sources for HapiSM can be downloaded from the [Github repo](#).

You can either clone the public repository:

```
$ git clone git://github.com/MAfarrag/HapiSM
```

Or download the [tarball](#):

```
$ curl -OJL https://github.com/MAfarrag/HapiSM/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

To install directly from GitHub (from the HEAD of the master branch):

- `pip install git+https://github.com/MAfarrag/HAPI.git`

or from Github from a specific release:

- `pip install git+https://github.com/MAfarrag/HAPI.git@{release}`

Now you should be able to start this environment's Python with `python`, try `import Hapi` to see if the package is installed.

More details on how to work with conda environments can be found here: <https://conda.io/docs/user-guide/tasks/manage-environments.html>

If you are planning to make changes and contribute to the development of Hapi, it is best to make a git clone of the repository, and do a editable install in the location of you clone. This will not move a copy to your Python installation directory, but instead create a link in your Python installation pointing to the folder you installed it from, such that any changes you make there are directly reflected in your install.

- `git clone https://github.com/MAfarrag/HAPI.git`
- `cd Hapi`
- `activate Hapi`
- `pip install -e .`

Alternatively, if you want to avoid using `git` and simply want to test the latest version from the `master` branch, you can replace the first line with downloading a zip archive from GitHub: <https://github.com/MAfarrag/HAPI/archive/master.zip> libraries.io.

2.1.7 Install using pip

Besides the recommended conda environment setup described above, you can also install Hapi with `pip`. For the more difficult to install Python dependencies, it is best to use the conda package manager:

- `conda install numpy scipy gdal netcdf4 pyproj`

you can check libraries.io. to check versions of the libraries

Then install a release {release} of Hapi (available from release 2018.1) with pip:

- `pip install HAPI-Nile=={release}`

2.1.8 Check if the installation is successful

To check if the install is successful, go to the examples directory and run the following command:

- `python -m Hapi.*****`

This should run without errors.

Note: This documentation was generated on Feb 02, 2023

Documentation for the development version: <https://Hapi.readthedocs.org/en/latest/>

Documentation for the stable version: <https://Hapi.readthedocs.org/en/stable/>

2.2 Tutorials

2.2.1 Inputs

2.2.2 Lumped Model Run

To run the HBV lumped model inside Hapi you need to prepare the meteorological inputs (rainfall, temperature and potential evapotranspiration), HBV parameters, and the HBV model (you can load Bergström, 1992 version of HBV from Hapi)

- First load the prepared lumped version of the HBV module inside Hapi, the triangular routing function and the wrapper function that runs the lumped model *RUN*.

```
1 import Hapi.rrm.hbv_bergstrom92 as HBVLumped
2 from Hapi.run import Run
3 from Hapi.catchment import Catchment
4 from Hapi.rrm.routing import Routing
```

- read the meteorological data, data has be in the form of numpy array with the following order [rainfall, ET, Temp, Tm], ET is the potential evapotranspiration, Temp is the temperature (C), and Tm is the long term monthly average temperature.

```
1 Parameterpath = Comp + "/data/lumped/Coello_Lumped2021-03-08_muskingum.txt"
2 MeteodataPath = Comp + "/data/lumped/meteo_data-MSWEP.csv"
3
4 ### meteorological data
5 start = "2009-01-01"
6 end = "2011-12-31"
7 name = "Coello"
8 Coello = Catchment(name, start, end)
9 Coello.readLumpedInputs(MeteodataPath)
```

- Meteorological data

```
1 start = "2009-01-01"
2 end = "2011-12-31"
3 name = "Coello"
4 Coello = Catchment(name, start, end)
5 Coello.readLumpedInputs(MeteodataPath)
```

- Lumped model

prepare the initial conditions, cathcment area and the lumped model.

```
1 # catchment area
2 AreaCoeff = 1530
3 # [Snow pack, Soil moisture, Upper zone, Lower Zone, Water content]
4 InitialCond = [0,10,10,10,0]
5
6 Coello.readLumpedModel(HBVLumped, AreaCoeff, InitialCond)
```

- **Load the pre-estimated parameters**
snow option (if you want to simulate snow accumulation and snow melt or not)

```

1 Snow = 0 # no snow subroutine
2 # if routing using Maxbas True, if Muskingum False
3 Coello.readParameters(Parameterpath, Snow)

```

- Prepare the routing options.

```

1 # RoutingFn = Routing.TriangularRouting2
2 RoutingFn = Routing.Muskingum_V
3 Route = 1

```

- now all the data required for the model are prepared in the right form, now you can call the *runLumped* wrapper to initiate the calculation

```

1 Run.runLumped(Coello, Route, RoutingFn)

```

to calculate some metrics for the quality assessment of the calculate discharge the *performancecriteria* contains some metrics like *RMSE*, *NSE*, *KGE* and *WB* , you need to load it, a measured time series of discharge for the same period of the simulation is also needed for the comparison.

all methods in *performancecriteria* takes two numpy arrays of the same length and return real number.

To plot the calculated and measured discharge import matplotlib

```

1 gaugei = 0
2 plotstart = "2009-01-01"
3 plotend = "2011-12-31"
4 Coello.plotHydrograph(plotstart, plotend, gaugei, Title= "Lumped Model")
5
6
7 .. image:: /img/lumpedmodel.png
8 :width: 400pt

```

- To save the results

```

1 StartDate = "2009-01-01"
2 EndDate = "2010-04-20"
3
4 Path = SaveTo + "Results-Lumped-Model" + str(dt.datetime.now())[0:10] + ".txt"
5 Coello.saveResults(Result=5, StartDate=StartDate, EndDate=EndDate, Path=Path)

```

2.2.3 Lumped Model Calibration

To calibrate the HBV lumped model inside Hapi you need to follow the same steps of running the lumped model with few extra steps to define the requirement of the calibration algorithm.

```

1 import pandas as pd
2 import datetime as dt
3 import Hapi.rrm.hbv_bergstrom92 as HBVLumped
4 from Hapi.rrm.calibration import Calibration
5 from Hapi.rrm.routing import Routing
6 from Hapi.run import Run
7 import Hapi.statistics.performancecriteria as PC
8

```

(continues on next page)

(continued from previous page)

```
9 Parameterpath = Comp + "/data/lumped/Coello_Lumped2021-03-08_muskingum.txt"
10 MeteoDataPath = Comp + "/data/lumped/meteo_data-MSWEP.csv"
11 Path = Comp + "/data/lumped/"
12
13 start = "2009-01-01"
14 end = "2011-12-31"
15 name = "Coello"
16
17 Coello = Calibration(name, start, end)
18 Coello.readLumpedInputs(MeteoDataPath)
19
20
21 # catchment area
22 AreaCoeff = 1530
23 # temporal resolution
24 # [Snow pack, Soil moisture, Upper zone, Lower Zone, Water content]
25 InitialCond = [0,10,10,10,0]
26 # no snow subroutine
27 Snow = 0
28 Coello.readLumpedModel(HBVLumped, AreaCoeff, InitialCond)
29
30 # Calibration boundaries
31 UB = pd.read_csv(Path + "/lumped/UB-3.txt", index_col = 0, header = None)
32 parnames = UB.index
33 UB = UB[1].tolist()
34 LB = pd.read_csv(Path + "/lumped/LB-3.txt", index_col = 0, header = None)
35 LB = LB[1].tolist()
36
37 Maxbas = True
38 Coello.readParametersBounds(UB, LB, Snow, Maxbas=Maxbas)
39
40 parameters = []
41 # Routing
42 Route = 1
43 RoutingFn = Routing.TriangularRouting1
44
45 Basic_inputs = dict(Route=Route, RoutingFn=RoutingFn, InitialValues = parameters)
46
47 ### Objective function
48 # outlet discharge
49 Coello.readDischargeGauges(Path+"Qout_c.csv", fmt="%Y-%m-%d")
50
51 OF_args=[]
52 OF=PC.RMSE
53
54 Coello.readObjectiveFn(PC.RMSE, OF_args)
```

- after defining all the components of the lumped model, we have to define the calibration arguments

```
1 ApiObjArgs = dict(hms=100, hmcr=0.95, par=0.65, dbw=2000, fileout=1, xinit =0,
2                 filename=Path + "/Lumped_History"+str(dt.datetime.now())[0:10]+".
↪txt")
```

(continues on next page)

(continued from previous page)

```
3
4 for i in range(len(ApiObjArgs)):
5     print(list(ApiObjArgs.keys())[i], str(ApiObjArgs[list(ApiObjArgs.keys())[i]]))
6
7 # pll_type = 'POA'
8 pll_type = None
9
10 ApiSolveArgs = dict(store_sol=True, display_opts=True, store_hst=True, hot_start=False)
11
12 OptimizationArgs=[ApiObjArgs, pll_type, ApiSolveArgs]
```

- Run Calibration

```
1 cal_parameters = Coello.lumpedCalibration(Basic_inputs, OptimizationArgs,
2     ↪ printError=None)
3
4 print("Objective Function = " + str(round(cal_parameters[0],2)))
5 print("Parameters are " + str(cal_parameters[1]))
6 print("Time = " + str(round(cal_parameters[2]['time']/60,2)) + " min")
```

2.2.4 Distributed Hydrological Model Calibration

The calibration of the Distributed rainfall runoff model follows the same steps of running the model with extra steps to define the calibration algorithm arguments

1- Catchment Object

- Import the Catchment object which is the main object in the distributed model, to read and check the input data, and when the model finish the simulation it stores the results and do the visualization

```
1 class Catchment():
2
3     =====
4     Catchment
5     =====
6     Catchment class include methods to read the meteorological and Spatial inputs
7     of the distributed hydrological model. Catchment class also reads the data
8     of the gauges, it is a super class that has the run subclass, so you
9     need to build the catchment object and hand it as an input to the Run class
10    to run the model
11
12    methods:
13        1-readRainfall
14        2-readTemperature
15        3-readET
16        4-readFlowAcc
17        5-readFlowDir
18        6-ReadFlowPathLength
19        7-readParameters
20        8-readLumpedModel
```

(continues on next page)

```

21     9-readLumpedInputs
22     10-readGaugeTable
23     11-readDischargeGauges
24     12-readParametersBounds
25     13-extractDischarge
26     14-plotHydrograph
27     15-PlotDistributedQ
28     16-saveResults
29
30     def __init__(self, name, StartDate, EndDate, fmt="%Y-%m-%d", SpatialResolution =
↪ 'Lumped',
31                 TemporalResolution = "Daily"):
32         =====
33         Catchment(name, StartDate, EndDate, fmt="%Y-%m-%d", SpatialResolution =
↪ 'Lumped',
34                 TemporalResolution = "Daily")
35         =====
36         Parameters
37         -----
38         name : [str]
39             Name of the Catchment.
40         StartDate : [str]
41             starting date.
42         EndDate : [str]
43             end date.
44         fmt : [str], optional
45             format of the given date. The default is "%Y-%m-%d".
46         SpatialResolution : TYPE, optional
47             Lumped or 'Distributed' . The default is 'Lumped'.
48         TemporalResolution : TYPE, optional
49             "Hourly" or "Daily". The default is "Daily".

```

- To instantiate the object you need to provide the *name*, *startdate*, *enddate*, and the *SpatialResolution*

```

1     from Hapi.catchment import Catchment
2
3     start = "2009-01-01"
4     end = "2011-12-31"
5     name = "Coello"
6
7     Coello = Catchment(name, start, end, SpatialResolution = "Distributed")

```

Read Meteorological Inputs

- First define the directory where the data exist

```
1 Path = Comp + "/data/distributed/coello"  
2 PrecPath = Path + "/prec"  
3 Evap_Path = Path + "/evap"  
4 TempPath = Path + "/temp"  
5 FlowAccPath = Path + "/GIS/acc4000.tif"  
6 FlowDPath = Path + "/GIS/fd4000.tif"  
7 ParPathRun = Path + "/Parameter set-Avg/"
```

- Then use the each method in the object to read the corresponding data

```
1 Coello.readRainfall(PrecPath)  
2 Coello.readTemperature(TempPath)  
3 Coello.readET(Evap_Path)  
4 Coello.readFlowAcc(FlowAccPath)  
5 Coello.readFlowDir(FlowDPath)
```

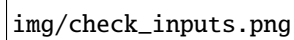
- To read the parameters you need to provide whether you need to consider the snow subroutine or not

2- Lumped Model

- **Get the Lumped conceptual model you want to couple it with the distributed routing module which in our case HBV**
and define the initial condition, and catchment area.

```
1 import Hapi.hbv_bergstrom92 as HBV  
2  
3 CatchmentArea = 1530  
4 InitialCond = [0,5,5,5,0]  
5 Coello.readLumpedModel(HBV, CatchmentArea, InitialCond)
```

- If the Inpus are consistent in dimensions you will get a the following message



- to check the performance of the model we need to read the gauge hydrographs

```
1 Coello.readGaugeTable("Hapi/Data/00inputs/Discharge/stations/gauges.csv", FlowAccPath)
2 GaugesPath = "Hapi/Data/00inputs/Discharge/stations/"
3 Coello.readDischargeGauges(GaugesPath, column='id', fmt="%Y-%m-%d")
```

3-Run Object

- The *Run* object connects all the components of the simulation together, the *Catchment* object, the *Lake* object and the *distributedrouting* object
- import the *Run* object and use the *Catchment* object as a parameter to the *Run* object, then call the *RunHapi* method to start the simulation

```
from Hapi.run import Run
Run.RunHapi(Coello)
```

- the result of the simulation will be stored as attributes in the *Catchment* object as follow

Outputs:

```
1-statevariables: [numpy attribute]
    4D array (rows,cols,time,states) states are [sp,wc,sm,uz,lv]
2-qlz: [numpy attribute]
    3D array of the lower zone discharge
3-quz: [numpy attribute]
    3D array of the upper zone discharge
4-qout: [numpy attribute]
    1D timeseries of discharge at the outlet of the catchment
    of unit m3/sec
5-quz_routed: [numpy attribute]
    3D array of the upper zone discharge accumulated and
    routed at each time step
6-qlz_translated: [numpy attribute]
    3D array of the lower zone discharge translated at each time step
```

4-Extract Hydrographs

- The final step is to extract the simulated Hydrograph from the cells at the location of the gauges to compare
- The *extractDischarge* method extracts the hydrographs, however you have to provide in the gauge file the coordinates of the gauges with the same coordinate system of the *FlowAcc* raster
- The *extractDischarge* will print the performance metrics

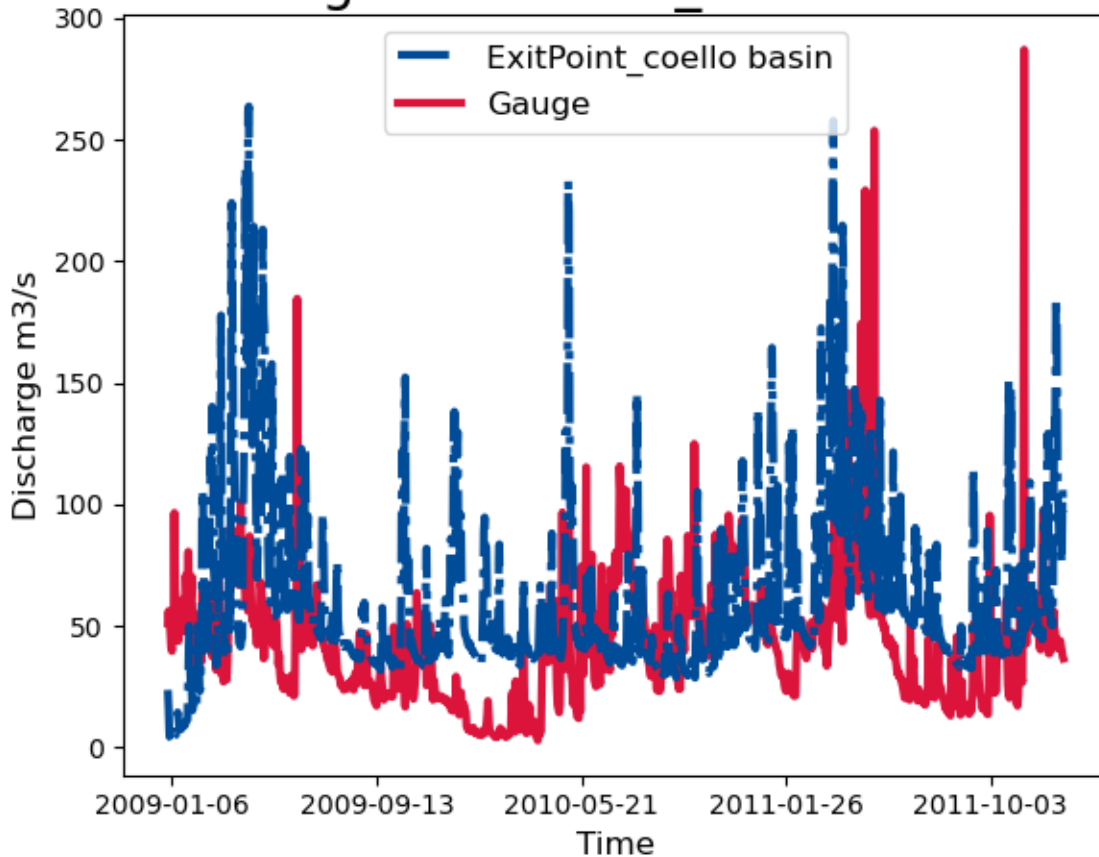
5-Visualization

- First type of visualization we can do with the results is to compare the gauge hydrograph with the simulated hydrographs
- Call the *plotHydrograph* method and provide the period you want to visualize with the order of the gauge

```
gaugei = 5
plotstart = "2009-01-01"
plotend = "2011-12-31"

Coello.plotHydrograph(plotstart, plotend, gaugei)
```


Gauge - ExitPoint_coello basin



width
400pt

6-Animation

- the best way to visualize time series of distributed data is through visualization, for this reason, The *Catchment* object has *plotDistributedResults* method which can animate all the results of the model

```
=====
AnimateArray(Arr, Time, NoElem, TicksSpacing = 2, Figsiz=(8,8), PlotNumbers=True,
             NumSize= 8, Title = 'Total Discharge',titlesize = 15,
             Backgroundcolorthreshold=None,
             cbarlabel = 'Discharge m3/s', cbarlabelsize = 12, textcolors=("white","black"),
             Cbarlength = 0.75, Interval = 200,cmap='coolwarm_r', Textloc=[0.1,0.2],
             Gaugecolor='red',Gaugesize=100, ColorScale = 1,gamma=1./2.,linthresh=0.0001,
             linscale=0.001, midpoint=0, orientation='vertical', rotation=-90,IDcolor = "blue",
             IDsize =10, **kwargs)
=====
```

Parameters

```
-----
Arr : [array]
      the array you want to animate.
```

(continues on next page)

```

Time : [dataframe]
    dataframe contains the date of values.
NoElem : [integer]
    Number of the cells that has values.
TicksSpacing : [integer], optional
    Spacing in the colorbar ticks. The default is 2.
Figsize : [tuple], optional
    figure size. The default is (8,8).
PlotNumbers : [bool], optional
    True to plot the values intop of each cell. The default is True.
NumSize : integer, optional
    size of the numbers plotted intop of each cells. The default is 8.
Title : [str], optional
    title of the plot. The default is 'Total Discharge'.
titlesize : [integer], optional
    title size. The default is 15.
Backgroundcolorthreshold : [float/integer], optional
    threshold value if the value of the cell is greater, the plotted
    numbers will be black and if smaller the plotted number will be white
    if None given the maxvalue/2 will be considered. The default is None.
textcolors : TYPE, optional
    Two colors to be used to plot the values i top of each cell. The default is ("white",
    ↪"black").
cbarlabel : str, optional
    label of the color bar. The default is 'Discharge m3/s'.
cbarlabelsize : integer, optional
    size of the color bar label. The default is 12.
Cbarlength : [float], optional
    ratio to control the height of the colorbar. The default is 0.75.
Interval : [integer], optional
    number to controlthe speed of the animation. The default is 200.
cmap : [str], optional
    color style. The default is 'coolwarm_r'.
Textloc : [list], optional
    location of the date text. The default is [0.1,0.2].
Gaugecolor : [str], optional
    color of the points. The default is 'red'.
Gaugesize : [integer], optional
    size of the points. The default is 100.
IDcolor : [str]
    the ID of the Point.The default is "blue".
IDsize : [integer]
    size of the ID text. The default is 10.
ColorScale : integer, optional
    there are 5 options to change the scale of the colors. The default is 1.
    1- ColorScale 1 is the normal scale
    2- ColorScale 2 is the power scale
    3- ColorScale 3 is the SymLogNorm scale
    4- ColorScale 4 is the PowerNorm scale
    5- ColorScale 5 is the BoundaryNorm scale
    -----
gamma : [float], optional

```

(continues on next page)

```

    value needed for option 2 . The default is 1./2..
    linthresh : [float], optional
        value needed for option 3. The default is 0.0001.
    linscale : [float], optional
        value needed for option 3. The default is 0.001.
    midpoint : [float], optional
        value needed for option 5. The default is 0.
    -----
orientation : [string], optional
    orintation of the colorbar horizontal/vertical. The default is 'vertical'.
rotation : [number], optional
    rotation of the colorbar label. The default is -90.
**kwargs : [dict]
    keys:
        Points : [dataframe].
            dataframe contains two columns 'cell_row', and cell_col to
            plot the point at this location

Returns
-----
animation.FuncAnimation.

```

- choose the period of time you want to animate and the result (total discharge, upper zone discharge, soil moisture,...)
 - to save the animation
 - Please visit <https://ffmpeg.org/> and download a version of ffmpeg compitable with your operating system
 - Copy the content of the folder and paste it in the “c:/user/.matplotlib/ffmpeg-static/”
- or
- define the path where the downloaded folder “ffmpeg-static” exist to matplotlib using the following lines

7-Save the result into rasters

- To save the results as rasters provide the period and the path

```

StartDate = "2009-01-01"
EndDate = "2010-04-20"
Prefix = 'Qtot_'

Coello.saveResults(FlowAccPath, Result=1, StartDate=StartDate, EndDate=EndDate, Path="F:/
↳02Case studies/Coello/Hapi/Model/results/", Prefix=Prefix)

```

2.2.5 Distributed Hydrological Model

After preparing all the meteorological, GIS inputs required for the model, and Extracting the parameters for the catchment

```
import numpy as np
import datetime as dt
import gdal
from Hapi.rrm.calibration import Calibration
import Hapi.rrm.hbv_bergstrom92 as HBV

import Hapi.statistics.performancecriteria as PC

Path = Comp + "/data/distributed/coello"
PrecPath = Path + "/prec"
Evap_Path = Path + "/evap"
TempPath = Path + "/temp"
FlowAccPath = Path + "/GIS/acc4000.tif"
FlowDPath = Path + "/GIS/fd4000.tif"
CalibPath = Path + "/calibration"
SaveTo = Path + "/results"

AreaCoeff = 1530
#[sp, sm, uz, lz, wc]
InitialCond = [0,5,5,5,0]
Snow = 0

# Create the model object and read the input data

Sdate = '2009-01-01'
Edate = '2011-12-31'
name = "Coello"
Coello = Calibration(name, Sdate, Edate, SpatialResolution = "Distributed")

# Meteorological & GIS Data
Coello.readRainfall(PrecPath)
Coello.readTemperature(TempPath)
Coello.readET(Evap_Path)

Coello.readFlowAcc(FlowAccPath)
Coello.readFlowDir(FlowDPath)

# Lumped Model
Coello.readLumpedModel(HBV, AreaCoeff, InitialCond)

# Gauges Data
Coello.readGaugeTable(Path+"/stations/gauges.csv", FlowAccPath)
GaugesPath = Path+"/stations/"
Coello.readDischargeGauges(GaugesPath, column='id', fmt="%Y-%m-%d")
```

-Spatial Variability Object

from Hapi.rrm.distparameters import DistParameters as DP

- The *DistParameters* distribute the parameter vector on the cells following some spatial logic (same set of parameters for all cells, different parameters for each cell, HRU, different parameters for each class in additional map)

```
raster = gdal.Open(FlowAccPath)
#-----
# for lumped catchment parameters
no_parameters = 12
klb = 0.5
kub = 1
#-----
no_lumped_par = 1
lumped_par_pos = [7]

SpatialVarFun = DP(raster, no_parameters, no_lumped_par=no_lumped_par,
                   lumped_par_pos=lumped_par_pos,Function=2, Klb=klb, Kub=kub)
# calculate no of parameters that optimization algorithm is going to generate
SpatialVarFun.ParametersNO
```

- Define the objective function

```
1 coordinates = Coello.GaugesTable[['id','x','y','weight']][:]
2
3 # define the objective function and its arguments
4 OF_args = [coordinates]
5
6 def OF(Qobs, Qout, q_uz_routed, q_lz_trans, coordinates):
7     Coello.extractDischarge()
8     all_errors=[]
9     # error for all internal stations
10    for i in range(len(coordinates)):
11        all_errors.append((PC.RMSE(Qobs.loc[:,Qobs.columns[0]],Coello.Qsim[:,i])))
12    ↪#*coordinates.loc[coordinates.index[i],'weight']
13    print(all_errors)
14    error = sum(all_errors)
15    return error
16 Coello.readObjectiveFn(OF, OF_args)
```

-Calibration algorithm Arguments

- Create the options dictionary all the optimization parameters should be passed to the optimization object inside the option dictionary:

to see all options import Optimizer class and check the documentation of the method setOption

```
1 ApiObjArgs = dict(hms=50, hmcr=0.95, par=0.65, dbw=2000, fileout=1,
2                 filename=SaveTo + "/Coello_"+str(dt.datetime.now())[0:10]+".txt")
3
4 for i in range(len(ApiObjArgs)):
5     print(list(ApiObjArgs.keys())[i], str(ApiObjArgs[list(ApiObjArgs.keys())[i]]))
6
7 pll_type = 'POA'
8 pll_type = None
9
10 ApiSolveArgs = dict(store_sol=True, display_opts=True, store_hst=True, hot_start=False)
11
12 OptimizationArgs=[ApiObjArgs, pll_type, ApiSolveArgs]
```

- Run Calibration algorithm

```
cal_parameters = Coello.runCalibration(SpatialVarFun, OptimizationArgs, printError=0)
```

- Save results

```
SpatialVarFun.Function(Coello.Parameters, kub=SpatialVarFun.Kub, klb=SpatialVarFun.Klb)
SpatialVarFun.saveParameters(SaveTo)
```

2.3 Hydrodynamic model