HABBY – Technical documentation

Table des matières

[How to execute HABBY 1](#_Toc466383930)

[Source Code 1](#_Toc466383931)

[Main() 2](#_Toc466383932)

[Source for the graphical user interface 2](#_Toc466383933)

[Source for the calculation 15](#_Toc466383934)

[Translation of HABBY 22](#_Toc466383935)

[Log and HABBY on the command line 22](#_Toc466383936)

[HABBY as a .exe (single application) 22](#_Toc466383937)

[Git and version control 22](#_Toc466383938)

[Random notes 22](#_Toc466383939)

[Needed Python module and license 23](#_Toc466383940)

[Useful reference and books 23](#_Toc466383941)

# How to execute HABBY

To execute HABBY:

* Go to folder which contains habby.py (cd… on the command line)
* Open the command line and type python habby.py

The python version should be 3.4. HABBY should also function with most of the python 3 distributions.

If a module is missing, it is possible to install it using pip (pip install – m “module\_name”). Obviously, pip needs to installed, which should be done by default in python 3.4. If the installation of a module fails with pip or if you want to be sure to have the same version of the module than originally, go to zen\_file -> wheele with the command line and install the missing module from there (something like pip install –m “\*.whl”). Not all modules are there, only the one which were difficult to install.

# Source Code

The source code is separated in two folders one folder which contain the code source for the graphical user interface (GUI) and one folder for the rest of the code source.

The dependency between the different part of the source code can be visualized in the mindmap real\_GUI.xmind (necessitate to have xmind installed).

## Main()

The main of HABBY is habby.py. it has the usual form for an application using PyQt. The main creates an application of QWidget and call the Main\_Windows class, which we will discuss shortly. The last line closes the application.

## Source for the graphical user interface

#### Class MainWindows

In Main\_windows\_1.py

The class MainWindows contains the menu and the title of all the HABBY windows.  
It also creates all the widgets which can be called during execution.

**Walk-through:**

* Load the user setting using Qsettings: The settings by default of Qsettings are the name of the program (HABBY) and the name of the organization which develops the program (irstea). I have added three user settings (the name of the last project loaded into HABBY, the path to this project and the language used). The Qsetting are stored in the registry in Windows and also function with MAC and Linux even if the information is stored differently
* We set up the translation next. The translation of HABBY in different language is explained in more detail in “translation of HABBY”. We give here the path to the data related to the translation. More precisely, we indicate here the path to the translation data and the name of the qm file containing the data related to the translation in each language. If a new qm is added for a new language, it should be added here to the list.
* Now, two important attributes are defined: self.name\_prj and self.path\_prj. These attribute will be communicated to children classes. For each project, an xml file is created. This “project” file should be called name\_prj.xml and should be situated in the path indicated by self.path\_prj.
* We call the central\_widget which contains the different tabs.
* We create the menu of HABBY calling the function my menu\_bar().
* Two signal are connected, one to save the project (i.e to update the xml project file) and another to save an ESTIMHAB calculation.
* We show the created widget.

**Methods:**

* closeEvent: the exist() was modified here because it used to crash from time to time otherwise. Why exactly this is more stable is not totally clear to me.
* setlangue: see the chapter on the translation
* save\_project: This function saves or creates the xml file related to the projet. In this xml file there is the path and the name to all files related to the project, notably the hdf5 files containing the hydrological data. To find or create the xml file, we use the attribute self.path\_prj and self.name\_proj. If the path to the project directory is not found an error appears. The error is here sent though additional windows (to be sure that the user notice this problem), using the Qmesssage module. The user should give the general info about the project in the general tab of HABBY and they are collected here. User option (using Qsetting) is next updated so that the user will find his project open the next time it opens HABBY. Now there are two choice: a) This is a new project b) the project exists already. If the project is new, the xml file is created and general information is written in this file. In addition, the text file which are necessary to log the action of HABBY are created now. This part of the reason why it is not possible to run other part of HABBY (such as loading hydrological data) before a project is saved. In addition, it would create a lot of problems on where to store the data created. Hence, a project is needed before using HABBY. If the project exists already (i.e. the name and the path of the project have not been modified), the xml file is just updated to change its attributes as needed. Interesting point are the biologie path (by definition .\biologie) which contains the biological information such as the preference curve and the path\_im which is the path where all figures and most utpus of HABBY is saved. If path\_im is not given, HABBY automatically create a folder called figure\_habby when the user creates a new project. The user can however change this path if he wants. The next step is to communicate to all the children widget than the name and path of the project have changed. Finally the log is written (see “log and HABBY in the command line).
* Save\_project\_estimhab: This function save the data and result from the estimhab calculation. It would look more logic if it was in the esimhab.py script, but it was easier to call it from here instead of in the child class. This function get all estimhab input, create an hdf5 file using h5py and save the data in the hdf5. One specialty of hdf5 is that is cannot use Unicode. Hence all string have to be passed to ascii using the encode function. The size of each data should also be known. Finally, we save the name and path of the estimhab file in the xml project file.
* Open\_rech and close\_rech: It is possible to show extra tab in HABBY. These supplementary tab correspond to open for researcher. The plan is that these options are less tested than other mainstream options. It is not clear yet what will be added to these options, but the basic architecture is there when it will be needed.
* Clear log and do\_log: see the paragraph on the log.
* Erase\_pict: From the menu of HABBY, it is possible to ask to erase all files in the folder indicated by path\_im (usually figure\_HABBY). Of course, this is a bit dangerous. So the function asks the user for confirmation. However, it is practical because you do not have to go to the folder to erase all the images when there are too many of them.

#### Class CentralW

In Main\_Windows\_1.py

This class creates the different tabs of HABBY, which are then used as the central widget by MainWindows.

In the attribute list, there are a series of name which finish by “tab” such as stathab\_tab or output\_tab. Each of these names corresponds to one tab and a new name should be added to the attributed to add a new tab. The attribute self.child\_win calls the class ShowImageW. This is the extra windows which open in HABBY to show the different figures.

**Walk-through**

During the creation of the class, each tab is created. Then, the signals to show the figures are connected between this class and all the children classes which need it (often the classes used to load the hydrological data). When a class emits the signal “show\_fig”, CentralW collect this signal and show the figure, using the showfig function, which call the instance child\_win of the class ShowImageW.

Then we call a function which connects all the signals from each class which need to write into the log. It is a good policy to create a “send\_log” signal for each new important class. As there are a lot of signal to connect, there are written in the function “connect\_signal\_log”, where the signal for a new class can be added.

Afterwards, the info for the general tab is filled. This means that if the user has open a project in HABBY before, the name of the project and the other info related to it will be shown on the general tab. This part should be updated if the general tab is modified.

Finally, each tab is filled. The tabs have been created before, but it was empty. Now we fill each one with the adequate widget. This is the link with many of the other classes that we describe below. Indeed, many of the widget are based on complicated classes created for example in hydro\_GUI\_2.py.

Then, we create an area under it for the log. Here HABBY will write various infos for the user. Two things to note here: a) we should show the end of the scroll area. b) The size of the area should be controlled and not be changing even if a lot of text appears. Hence, the setSizePolicy should be fixed.

The write\_log() and write\_log\_file() method are explain in the section about the log.

#### Class WelcomeW()

In Main\_Windows\_1.py

The class WlecomeW() creates the first tab of HABBY (the tab called “General”). There is nothing really special about this tab. It is there to create a new project or to change the name, path,… of a projects.

The setfolder methodis used to select directory.

#### Class ShowImageW()

In Main\_Windows\_1.py

The ShowImageW() class is used to show the figures created by HABBY. To show an image, HABBY open a separate window and show the saved image in .png format. The user can just look at the image, not zoom or change it. This is a limited functionality compared to the image open in matplotlib. However, HABBY cannot open figures in matplotlib without serious bugs. Namely, it crashes if one figure is closed by the user while a second is let open. Changing the backend of matplotlib did not change this problem. So we decide to save the figures before and to show them in a second window. Currently, the figures shown are in .png, but other formats could be used. For this, one can change the variable self.imtype.

An important point for the ShowImageW class is where the images were saved by the functions which created them. In HABBY, all figures are saved in the same folder called “path\_im”. One “path\_im” is chosen at the start of each project. By default, it is the folder “Figure\_Habby”, but the user can modify this folder in the window created by ShowImageW(). The function for this is called “change\_folder”, also in ShowImageW(). The path\_im is written in the xml project file. The different functions which create image read this path and send the figure created to this folder.

ShowImageW() reads all figure of “.png” type in the” path\_im” folder and show the most recent figure. The user can use the drop-down menu to choose to see another figure. The names of the figure are added to the drop-down menu in the function update\_namefig. The function selectin change changes the figure shown based on the user action.

#### Class Hydro2W()

In hydro\_GUI\_2.py

The class Hydro2W is the second tab of HABBY. It is the class containing all the classes/Widgets which are used to load the hydrological data.

To call the different classes used to load the hydrological data, the user selects the name of the hydrological model from a QComboBox call self.mod. The method ‘selection\_change” call the class that the user chooses in self.mod. All the classes used to load the hydrological data are created when HABBY starts and are kept in a stack called self.stack. The function selectin\_change() just changes the selected item of the stack based on the user choice on self.mod. Any new hydrological model should also be added to the stack and to the list of models contained in self.mod (name of the list: self.name\_model).

In addition to the stack containing the hydrological information, hydro2W has two buttons. One button open a QMessageBox() which give information about the models, using the method “give\_info\_model”. It is useful if a special type of file is needed to load the data from a model or to give extra information about one hydrological model. The text which is shown on the QMessageBox is given in one text file for each model. These text file are contained in the folder ‘model\_hydro” which is in the HABBY folder. For the moment, there are models for which no text files have been prepared. The text file should have the following format:

* A short sentence with general info
* The keyword: MORE INFO
* all other info which are useful.

The second button allows the user to load an hdf5 file containing hydrological data from another project. As long as the hdf5 is in the right format, it does not matter from which hydrological model it was loaded from or even if this hydrological model is supported by HABBY.

**Here comes the decription of load\_hydro\_hdf5**

#### Class FreeSpace

In hydro\_GUI\_2.py

This classes is doing exactly what is name says. The idea is that the user see a free space when it opens the “Hydro” Tab instead of directly seeing one of the model. The idea is to avoid the case where a user tries to load data before selecting the real model. For example, if a user wants to load mascaret data and that an item is already slected in the stack of classes related to hydrology (such as HEC-RAS1D), it might be logical for the user to try to load masacret data using the HEC-RAS class. Because of the FreeSpace class, he actually has to select the model he wants to load.

#### Class SubHydroW

In hydro\_GUI\_2.py

This class is a bit special. It is not called directly by HABBY but the classes which load the hydrological data inherits from the SubHydroW class. The advantage of this architecture is that all the children classes can use the methods written in SubHydroW(). Indeed, all the children classes load hydrological data and therefore they are similar and can use similar functions. We will shortly list here these methods:

* was\_model\_loaded\_before: This method opens the xml project file and look in the attribute of the xml file to see if data from the hydrological model have been loaded before. If yes, the name of the data is written on the GUI of HABBY in the Widget related to the hydrological model. Now, there are often more than one data loaded. This method allows choosing what should be written. There are two different case to be separated: a) We have loaded two different models (like two rivers modeled by HEC-RAS) b) One model type needs two data file (like HEC-RAS would need a geometry and output data). For the case a), the default is to write only the first model loaded. If we wish to write all data, the switch “many\_file” should be True. This switch is also useful for the river2D model, because this model create one output file per time step. For the case b), the argument “i”(which is an int) allows us to choose which data type should be shown. “i” is in the order of the self.attributexml variable. The definition of this order is given in the definition of the class of each hydrological model.
* Show\_dialog: This is method to open a dialog so that the user select a file. This file is NOT loaded here. The name and path to this file is saved in an attribute. This attribute is then used to loaded the file in other function, which are different for each children class.
* Save\_xml: This function add the name and the path of the newly chosen hydrological data to the xml project file. First, it open the xml project file (and send an error if the project is not save, or if it cannot find the project file). Then, it open the xml file and add the path and name of the file to this xml file. If model data was already loaded, it adds the new name without erasing the old name IF the switch append\_name is True. Otherwise, it erase the old name and replace it by a new name. the variable “i” has the same role than in show\_dialog.
* Save\_hdf5: this function creates an hdf5 file which contains the hydrological data. First it creates an empty hdf5. Then it fill the hdf5 with data. For 1D model, it fill the data in 1D (the original data), then the 1.5D data created by dist\_vitess2.py and finally the 2D data. For model in 2D it only saved 2D data. Hence, 2D data is the data which is common to all model and which can always be loaded from a hydrological hdf5 created by HABBY. The 1D and 1.5D data is only present if the model is 1D or 1.5D. Here is some general info about the created hdf5:
  + *Name of the file*: name\_projet + ’\_’ + name model + date/time.h5. For example, test4\_HEC-RAS\_25\_10\_2016\_12\_23\_23.h5.
  + *Position of the file***:** in the folder figure\_habby currently (probably in a project folder in the final software)
  + *Format of the hdf5 file:* 
    - Dats\_gen: number of time step and number of reach
    - Data\_1D: xhzv\_data\_all (given profile by profile)
    - Data\_15D : vh\_pro, coord\_pro (given profile by profile in a dict) and nb\_pro\_reach.
    - Data\_2D : For each time step, for each reach: ikle, point, point\_c, inter\_h, inter\_vel

If a list has elements with a changing number of variables, it is necessary to create a dictionary to save this list in hdf5. For example, a dictionary will be needed to save the following list: [[1,2,3,4], [1,2,3]]. This is used for example, to save data by profile as we can have profile with more or less points. We also note in the hdf5 attribute some important info such as the project name, path to the project, hdf5 version. This can be useful if an hdf5 is lost and is not linked with any project. We also add the name of the created hdf5 to the xml project file. Now we can load the hydrological data using this hdf5 file and the xml project file.

* Find\_path\_im: This is a short function to find the path where the figures have to be saved. To avoid problem with the run time of matplotlib. All figure are created and saved but not shown outside of the class ShowImageW(). We lost our ability to zoom on an image, but we gain in stability. However, it is important to save the image somewhere where ShowImageW() can find them. This path is call path\_im and is written in the xml project file. By default, it is in a folder called “Figure\_Habby.
* Send-err\_log: see the sectin about log.
* **Grid\_and\_interpo: to be done**
* Distribute\_velocity: For all 1D models, this function calls the function to distribute the velocity and manage the log data.

#### Class Hec-Ras 1D()

In hydro\_GUI2.py

The class Hec\_ras 1D is there to manage the link between the graphical interface and the functions in src/hec\_ras06.py which loads the hec-ras data in 1D. It inherits from SubHydroW() so it have all the methods and the variables from the class SubHydroW(). For example, the send\_log signal is in the class HecRas1D because we define it in the class SubHydroW(). The class hec-ras 1D is added to the self.stack of Hydro2W(). So the class Hec-Ras 1D is called when the user in on the hydro tab and click on hec-ras1D as hydrological model.

**Walk-through:**

* The self.attributexml variable is the name of the attribute in the xml file. To load a hec-ras file, one needs to give to HABBY one file containing the geometry data and one file containing the simulation result. The name and path to these two file are saved in the xml project file under the attribute given in the self.attributexml variable.
* The variable self.extension is a list of list of the accepted file type. The first list is for the file with geometry data. The second list is the extension of the files containing the simulation results.
* Using the function self.was\_model\_loaded\_before, HABBY write the name of the hec-ras files which were loaded in HABBY in the same project before.
* Creation of the label and button related to wghich faile will be loaded
* Creation of label and button related to the creation of the grid. Indeed, Hec-Ras is a 1.5D model and so HABBY create a 2D grid based on the 1.5D input. One can choose the interpolation types and the number of extra profile. If the interpolation type is “interpolation by block”, the number of extra\_profile will always be one. See manage\_grid.py for more information on how to create a grid.

**Function load\_hec\_ras\_gui:**

This function is called when the user press on the button self.load\_b. It is the function which really calls the load function for hec\_ras. First, it updates the xml project file. It adds the name of the new file to xml project file under the attribute indicated by self.attributexml. It also gets the path\_im by reading the path\_im in the xml project file. Then it check if the user want to create the figure or not (if self.cb.isChecked(), figures should be created). Then, it manages the log as explained in the section about the log. Notably, it redirects the outstream to the mystdout stream. Hence, the “print” statement is now sent to the log windows at the bottom of HABBY window. Next, it loads the hec-ras data as explained in the section on hec\_ras06.py. It then creates the grid as explained in the manage\_grid.py based on the interpolation type wished by the user (linear, nearest neighbor or by block). Next, it creates the hdf5 with the loaded data. Finally, if necessary, it shows the figure by emitting the signal which call the instance of the class ShowImageW(). The signal is collected in the MainWindow() class.

#### Class Rubar2D

In hydro\_GUI2.py

The class Rubar2D is there to manage the link between the graphical interface and the functions in src/rubar.py which loads the RUBAR data in 2D. It inherits from SubHydroW() so it have all the methods and the variables from the class SubHydroW(). The form of the function is similar to hec-ras, but it does not have the part about the grid creation as we look here as the data created in 2D by RUBAR

It does a have a load\_rubar function, which is really similar to the load\_hec\_ras\_gui() function. Obviously, it calls rubar and hec\_ras this time. A small difference is that the rubar2D outputs are only given in one grid for all time steps and all reaches. It will be necessary to cut the grid for each time step as a function of the wetted area and maybe to separate the grid by reaches. This have not be done yet.

Another addition, it is that that we have a small function called propose\_next\_file which propose the other second RUBAR file. Indeed, to load rubar, we need one file with the geometry data and one file with the simulation results. If the user selects a file, this function looks if a file with the same name but with the extension of the other file type exists in the selected folder. This could be done for all hydrological models, but the function is harder to write is more than one extension is possible. This part is just not totally finished yet.

Another problem is that the data of Rubar2D is given on the cells of the grid and not the nodes. This will need to be corrected as data in HABBY is centered on the node (after correction).

#### Class Mascaret

In hydro\_GUI2.py

The class Mascaret is there to manage the link between the graphical interface and the functions in src/rubar.py which loads the Masacret data in 1D. It inherits from SubHydroW() so it have all the methods and the variables from the class SubHydroW(). It is similar to the HEC-Ras1D class. However, mascaret is 1D model, so the loading of mascaret has one additional step compared to the hec-ras load: The velocity must be distributed along the profile. For this, the load\_masacret\_gui call the self.distrbute \_velocity function. In addition, it prepares the manning value which is necessary to distribute the velocity. The user has two choices to input the manning value. The easiest one is just to give a value constant for the whole river. In the second choice, the user loads a text file with a serie of lines with the following info: p, dist, n where p is the profile number (starting at zero), dist is the distance in meter along the profile and n in the manning value.

#### Class River2D

In hydro\_GUI2.py

The class River2D t is there to manage the link between the graphical interface and the functions in src/rubar.py which loads the River2D data in 2D. It inherits from SubHydroW() so it have all the methods and the variables from the class SubHydroW(). It is similar generally to the hec-ras class. However, River2D create one file per time step. Hence, it is necessary to have a way to load all the files automatically. Indeed, loading one file after one file would be annoying. This is the function “add\_all\_file” which looks for all files in a folder selected by the user. Three additional functions exist to manage the large number of file:

* Remove\_file removes one file
* Remove\_all\_file removes all selected files
* Add\_file add just one selected file

None of this four function load the data, it just add the name and path of the files to be loaded to self.namefile and self.pathfile. Generally, in HABBY, we load hydrological data in two steps: a) select the files, b) load the data. For river2D, the step b) is done by the function load\_river2d\_gui(). This function is similar to the one used by Rubar2D. It has the same problem about the grid which is identical for all time steps and which contains all reaches together. So a temporary correction was applied. Data in River2D is given on the nodes as in HABBY.

#### Class Rubar1D

In hydro\_GUI2.py

The class Rubar1D is there to manage the link between the graphical interface and the functions in src/rubar.py which loads the Rubar1D data in 1D. It inherits from SubHydroW() so it have all the methods and the variables from the class SubHydroW(). It is very similar to mascaret class.

#### Class HEC\_RAS2D

In hydro\_GUI2.py

The class hec\_RAS2D is there to manage the link between the graphical interface and the functions in src/rubar.py which loads the hec\_ras2D data in 2D. It inherits from SubHydroW() so it have all the methods and the variables from the class SubHydroW(). It is very similar to RUBAR2D class and it has the same problem about node/cell which will need to be corrected.

#### Class Telemac

In hydro\_GUI2.py

The class Telemac is there to manage the link between the graphical interface and the functions in src/rubar.py which loads the Telemac data in 2D. It inherits from SubHydroW() so it have all the methods and the variables from the class SubHydroW(). It is very similar to RUBAR2D class, but data is on the node as in HABBY.

#### Class SubstrateW

**To be written**

#### Class EstimhabW

In estihmab\_GUI.py

The Estimhab Class provides the graphical interface for the version of the Estimhab model written in HABBY. The Estimhab model is described below (in the source for the calculation). This class just loads the data for Estimhab given by the user.

###### Walk-though for the init function

* First we looked if some data for Estimhab was saved before by an user. If yes, we will fill the GUI with the information saved before. Estimhab information is saved in hdf5 file format and the path/name of the hdf5 file is saved in the xml project file. So we open the xml project file and look if the name of an hdf5 file was saved. If yes, the hdf5 file is read. The format of hdf5 file is relatively simple. Each input data for Estimhab has its own dataset (qmes, hmes, wmes, q50, qrange, and substrate). Then, we a list of string which are a code for the fish species which were analyzed. All the data contained in hdf5 file is loaded into variable.
* The different label are written on the graphical interface
* Then, two QListWidget are modified. The first list contains all the fish species on which HABBY has info (see XML Estimhab format for more info). The second list is the fish selected by the user on which Estimhab will be run. Here, we link these lists with two functions so that the user can select/deselect fish using the mouse. The function name are add\_fish() and remove\_fish().
* Then, we fill the first list. HABBY look up all file of xml type in the “Path\_bio” folder (the one indicated in the xml project file under the attribute “Path\_bio”). The name are them modified so that the only the name of species appears (and not the full path).
* We set the layout with all the different QLineEdit where the user can write the needed data.
* Estimhab model is saved using a function situated in MainWindows\_1.py (frankly, I am not so sure why I did put the save function there, but anyway). So the save button just send a signal to MainWindows here, which save the data.

###### Walk-through for Run\_estimhab

This function loads in memory the data given in the graphical interface and call sthe Estimhab model. The data could be written by the user now or it could be data which was saved in the hdf5 file before and loaded when HABBY was open (and the init function called). We check that all necessary data is present and that the data given makes sense (e.g.,the minimum discharge should not be bigger than the maximal discharge, the data should be float). We then remove the duplicate fish species (in case the user select one species twice) and the Estimhab model is called. This is the function making the link between the GUI and the source code proper. The source code for Estimhab is in src/Estimhab.py. The log is then written (see the paragraph on the log for more information). Next, the figures created by Estimmhab are shown. As there is only a short number of outputs for Estimhab, we create a figure in all cases (it could be changed by adding a checkbox on the GUI like in the Telemac or other hydrological class).

###### Estimhab xml file format

The biological data, i.e., the “preference” curves of Estimhab are saved in xml files situated in the folder given by the path written in the xml project file under the attribute “Path\_bio”. By default, it is HABBY/biology. It is possible to change this folder using the GUI (there is a “change folder” button). To change the “Path\_bio” folder, the function “change\_folder” is used. It updates the “Path\_bio” attribute in the xml project file.

The form of the xml files containing the hydrological data is described here.

Estimhab is a statistical model, which functions using mathematical regressions. The different regressions of each fish are described in an xml file whose format is given here.

Conceptually, the regressions “R” are of two types:

* Type 0 R = C \* Q^m1 \* exp(m2\*Q)
* Type 1 R = C \* (1+m1\*exp(m2\*Q))

Where Q is the discharge, m1 and m2 are coefficients which depend on the fish type, and C is a constant which depends on the stream characteristic and the fish type.

The constant C is of the form C = a + ∑ ai \* ln(Si) where a and ai are coefficients which depend on the fish type. Si are particular stream characteristics. Which characteristics should be used is a function of the fish type and is so given in the xml file. The value of S i is a function of the stream and is calculated by the program.

In the xml file,

* Attribute **coeff\_q:** Give the main coefficient of the regression (m1 and m2)
* Attribute **func\_q** : Give the type of regression R used. Type 0 and type 1, as described above, are known by the program.
* Attribute **coeff\_const:** Give the coefficient used to construct the constant C (a, a1, a2, a3,…). The number of coefficient differs for each fish, but should be at least one.
* Attribute **var\_const:** Give which type of stream characteristics is used. This is not the value of the particular characteristic, but only which type is used. The following list of type is accepted:

|  |  |
| --- | --- |
| 0 | Q50, natural median discharge |
| 1 | H50, the height of the stream at q50 |
| 2 | L50, the width of the stream at q50 |
| 3 | V50 the velocity of the stream at q50 |
| 4 | Re50: discharge divided by 10 times the width at Q50 |
| 5 | Fr50: the Froude number at Q50 |
| 6 | Dh50: the mean substrate height divided by h50 |
| 7 | Exp(Dh50). Erase the log() of this particular term of the constant |

#### Class StathabW()

The class StathabW make the link between the data prepared by the user for Stathab and the stathab model which is in the src folder (stathab\_c.py) using the graphical interface. Most of the stathab input are given in form of text file. For more info on the preparation of text files for stathab, read the document call ‘stathabinfo.pdf”. To use Stathab in HABBY, all Stathab input should be in the same directory. The user select this directory (using the button “loadb”) and HABBY tries to find the file it needs. All found files are added to the list called “file found”. If file are missing, they are added to the “file still needed” list. The user can then select the fishes on which it wants to run stathab, then it run it by pressing on the “runb” button.

If file where loaded before by the user in the same project, StathabW looks for them and load them again. Here we can have two cases: a) the data was saved in hdf5 format (as it is done when a stathab run was done) and the path to this file noted in the xml project file. b) Only the name of the directory was written in the xml project file, indicated that data was loaded but not saved in hdf5 yet. HABBY manages both cases.

Next, we check in the xml project file where the folder to save the figure (path\_im) is. In case, there are no path\_im saved, Stathab create one folder to save the figure outputs. This should not be the usual case. Generally, path\_im is created with the xml project file, but you cannot be sure.

There is a list of error message which are there for the case where the data which was loaded before do not exist anymore. For example, somebody erased the directory with the Stathab data in the meantime. In this case, a pop-up message open and warn the user.

An important attribute of StathabW() is self.mystathab. This is an object fo the stahab class. The stathab model, which is in the form of a class and not a function, will be run on this object.

###### Select\_dir()

In StathabW(), the select\_dir function allows the user to select a directory. In addition to this, it empties all files which were selected before. Next, it saves the path to the new selected directory in the xml project file. Finally, it calls the function which loads the text data and which creates the hdf5 (see below load\_from\_txt\_gui).

Load\_from\_txt\_gui()

The main roles of load\_from\_text\_gui () are to call the load\_function of the stathab class (which is in stathab\_c.py in the folder src) and to call the function which create an hdf5 file. However, it does some modifications to the GUI before. Let’s look at them one by one.

First, it updates the label. Because a new directory was selected, we need to update the label containing the directory’s name. We only show the 30 last character of the directory name. In addition, we also need to update the other label. Indeed, it is possible that the data used by Stathab would be loaded from an hdf5 file (see below). In this case, the labels on the top of the list of file are slightly modified. Here, we insure that we are in the “text” version since we will load the data from text file.

Next, it gets the name of all the reach and adds them to the list of reach name. For this, it calls a function from the stathab class (in src). Then, it looks which files are present and add them to the list which contains the reach name called self.list\_re.

Afterwards, it checks if the files needed by Stathab are here. The list of file is given in the self.end\_file\_reach list. The form of the file is always the name of the reach + one item of self.end\_file\_reach. If it does not find all files, it add the name of the files not found to self.list\_needed, so that the user can be aware of which file he needs. The exception is Pref.txt. If HABBY do not find it in the directory, it uses the default “Pref.txt”. All files should be in the same directory.

Then, it calls a method of the Stathab class (in src) which reads the “pref.txt” file and adds the name of the fish to the GUI.

Next, if all files are present, it loads the data using the method written in Stathab (in src). When the data is loaded, it creates an hdf5 file from this data and save the name of this new hdf5 file in the xml project file (also using a method in the stathab class). Finally, it sends the log info as explained in the log section of the documentation.

###### Select\_hdf5

This function allows the user to choose an hsdf5 file as input from Stathab. For example, he would have created an hdf5 file for a Stathab model in another project and he would like to send the same model but on other fish.

For this, the user first selects an hdf5 file.

Then, HABBY writes the name of this hdf5 file in the xml project file. It also notes that the last data loaded was of hdf5 type. This is useful when HABBY is restarting because it is possible to have a directory name and the address of an hdf5 file in the part of the xml project file concerning Stathab. HABBY should know if the last file loaded was this hdf5 or the files in the directory.

Finally, it calls the function to load the hdf5 (see below).

###### Load\_hdf5\_from\_GUI

This is the function to call the load\_stathab\_from\_hdf5 function from the GUI. In addition to call the function to load the hdf5, it also updates the GUI according to the info contained in the hdf5.

First it updates the Qlabel similarly to “Loaf\_from\_txt\_gui()”.

Then, it loads the data calling the load\_stathab\_from\_hdf5 function from the Stathab class in src. It writes the log afterwards (see the section on the log).

The info contains in the hdf5 file are now in memory in various variables called self.mystathab.”something”. HABBY now used them to update the GUI. First, it updates the list which contains the name of the reaches (self.list\_re.). Next, it checks that each of the variable needed exists and that they contain some data.

Afterwards, HABBY looks which preference file to use. Either, it will use the default preference file (contains in HABBY/biologie) or a custom preference prepared by the user. This custom preference file should be in the same folder than the hdf5 file. When the preference file was found, HABBY reads all the fish type which are described and add their name to the self.list\_f list which show the available fish to the user in the GUI.

Finally we check if all the variables were found or if some were missing.

###### Reach\_selected

This is a small function which only impacts the GUI. When a Stathab model has more than one reach, the user can click on the name of the reach. When he does this, HABBY selects the first file linked with this reach and shown in self.list\_f. This first file is highlighted and the list is scrolled down so that the files linked with the selected reach are shown. This function manages this. It is connected with the list self.list\_re, which is the list with the name of the reaches.

###### Add fish

This function is there because we often have to load the name of the fish (for example when we load the preference file). So we wrote a function for this.

###### Remove fish or add\_all\_fish

Idem, but to remove a fish or to add all fish

###### Run\_stathab\_gui

This is the function which calls the function to run the Stathab model. First it read the list called self.list\_s . This is the list with the fish selected by the user. Then, it calls the function to run stathab and the one to create the figure if the figures were asked by the user. Fiannly it write the log.

#### Class OutputW

In output\_fig\_GUI.py

This part is not finished. The idea is to let the user select various options to create the figures, notably the colors or the size of the text. All these data would be saved in a dictionary which would be padded to all the function which creates the figures. It would be possible to read the figures options from the created dictionary. This implies to make small modifications to all function which creates the figures. It has not been done yet.

Currently, this class updates this dictionary and saved it into the xml project file where it could be loaded again, for example by the GUI function calling the function which create the figures. But it should still be added to the function which creates the figures.

## Source for the calculation

#### Dist\_vitesse2.py

The goal of this list of function is to distribute the velocity along the cross-section for 1D model such as mascaret or Rubar BE. Hec-Ras outputs do not need to uses this type of function as they are already distributed along the profiles.

The method of velocity distribution in HABBY is similar to the one used by Hec-Ras to distribute velocity.

###### Walk-through for Dist\_velocity\_hecras

As input data, we have:

* Coord\_pro: This is the form of the profile. For each profile, it gives the coordinates of each point of the profile, the elevation of the profile at this points and the distance along the profile (often noted “x” even if it is a bit confusing with the coordinates)
* Xhzv data: This is the 1D output as prepared by the loading of mascaret or Rubar BE. It contains water height and velocity.
* Manning\_pro is the manning input.
* Nb\_point : see below
* Eng: it should be let at 1.0, as long as we are in SI unit.
* On\_profile: is only used by mascaret at it could have output data between the profiles. It means that there would be place where the form of the profile is not known, but where the velocity was modelled.

First, we decide on which point along the profile we will calculate the velocity. This is controlled by the variable nb\_point. If nb\_point=-99, we will calculate the velocity at the same point than the profile (i.e., the velocity will be calculated at each point on which the elevation of the profile was measured). There are cases where this is not adequate. Let’s imagine for example a rectangular canal. The calculation would only give two velocity points, which is not enough. So, it is possible to give the number of velocity point on which the calculation must be made, using the variable nb\_point.

Currently, the velocity points are determined by dividing the whole profile in nb\_point segments. This means that some velocity point are not used afterwards because they are in the dry part of the profile and that it is not possible to select for a part of the profile where more velocity points would be calculated. This could be modified in the future if it is judged necessary.

To determine the point where velocity should be calculated we need to get two array: one “x” array, the distance along the profile and one “h” array, the elevation of the profile at this point. As we choose the position of the velocity point as regularly placed along the profile, the “x” array is easy to determine using linespace. For the “h” array, we use the hypothesis that the elevation of the profile changes linearly between the measured elevation points. We find between which elevation point are the new point and we use a linear interpolation to find the new”h”. To find between which points we are, we use the bisect.bisect function. It is a bit like the np.where function, but it is quicker when the array is ordered.

Then, we get the manning array as created by the get\_manning\_arr and the get\_manning function. It should be a float.

Next, we cut the profile to keep only the part under water. For this, we do two things: First we had a point on the profile where h==0. We should account for the fact that we might have “islands” (part of the profile which are dry, but surrounded by water on both side.). So we cannot only looked which part are dry, we need to look for each point where we pass from “wet to dry” or from “dry to wet”. At this place, we add one point where h= 0. For these new points water height is obviously known, but x 9distlance along profile) should be determined. It is determined assuming a linear change between the measured points of the profile.

If the profile is not entirely dry, we will now distribute the velocity along the profile. First, for each part of the profile where velocity will be calculated, it looks where is the higher height (like if this part of the profile is going up or down).

Next, we calculated the area, the wetted perimeter and the hydraulic radius of each part of the profile. By combining this geometrical information with the manning parameter, we can calculate the conveyance of each part of the profile.

We now calculate the conveyance of the whole profile.

Normally, the sum of the conveyance of the part is higher than the total conveyance. The next part of the script corrects for this, using the ratio of the total conveyance and the sum of the parts of the conveyance.

Next, we calculate the velocity using the modelled energy slope (Sf) and the manning equation. We then then add a velocity of zero where there are no water (velocity is not defined at his point).

###### Plot\_dist\_vit

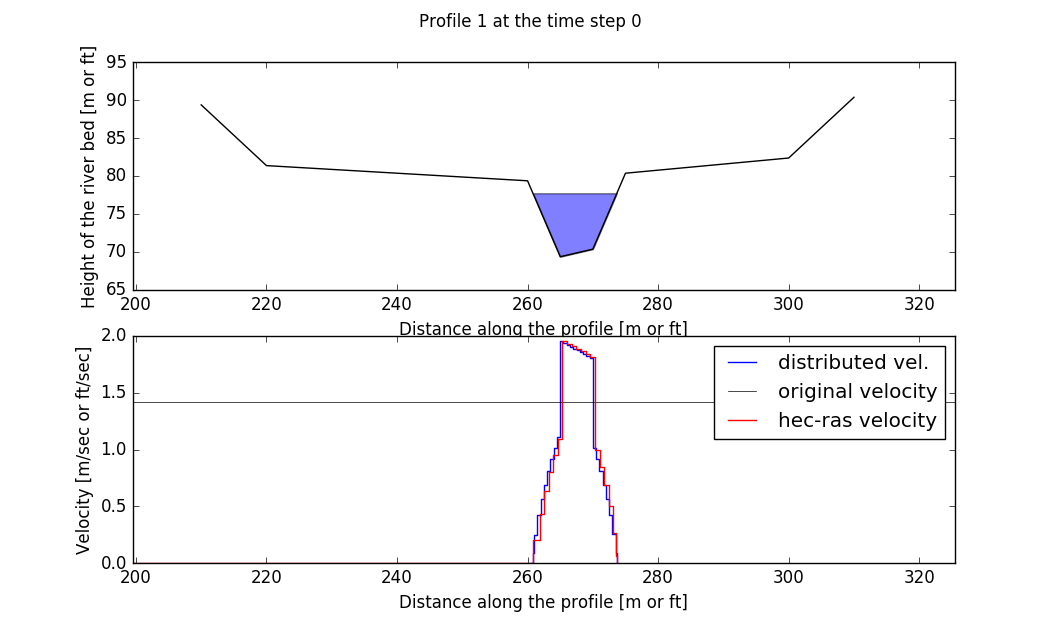
This is a function to plot the distribution of velocity and the elevation of the profile. It is quite close to the similar function which is in hec-ras (see this function for a more detailed explanation)

It can be used to test the program if we provide the variable zone\_v\_all which would be hec-ras output with a velocity distribution. In this case, it would plot the comparison between the output from this script and the output from hec-ras. Of course, it is necessary in this case to have prepared 1D output from hec-ras (using the function preparetest\_velocity) and have the same points on which to calculate the velocity.

###### Preparetest\_velocity()

This is a debugging function. It takes as input the output from the hec-ras model and gives a 1D velocity as output. This is only to test this program. It will not be used by HABBY directly. To use this function, it is necessary to use the function to load hec-ras data from HABBY, so that the hec-ras data is in the right form. The 1D-velocity is assumed to be the velocity as the lowest part of the profile. This is where a 1D-model would estimate the position of the river (the lowest part of the river bed).

A complicated point to test the program is to put the velocity point at the same point than hec-ras. As hec-ras calculate velocity between zones and not one point, this is more or less impossible to do with precision. However, one can count the number of velocity zone and give this as an input to dist\_velocity\_hecras() for the variable nb\_point. However, both line will not be exactly at the same place. A figure similar to one below is sufficient as a test.



Get\_manning()

The function dist\_velcoity\_hec\_ras needs a manning array with a length equal to the number of profile, where each row (representing a profile) have one value by velocity point which will be calculated. This function creates an array of this form based on a float. It creates a list of manning value which is identical for each point of the river.

It can be used for the cases where the same number of point is asked for each profile or for the case where the number of point is defined by the form of the profile (nb\_point = -99).

Get\_manning\_arr()

A function similar to get\_manning\_arr() but where the user can give various value of manning. For this, the user creates a txt file with a list of manning info. Each manning value is given the following way: the profile, the distance along the profile and the manning value. This function automatically fills the missing value, so that the user do not needs to give each manning value. He can describe one profile and this profile will be replicated until the next profile written in the text file.

#### Estimhab.py

This the function which forms the Estimhab model in HABBY. It is a reproduction in python of the excel file which forms the original Estimhab model.

First, we get all the discharges on which we want to calculate the SPU (surface ponderée utile), using the inputs from the user.

Next we use hydrological rating curves (info on google if needed) to get the height and the width of the river for all discharge. The calculation is based on the width and height of the river measured at two discharges (given by the user).

Next, we get other parameters which are used in the preference curves such as the Froude number of the mean discharge or the Reynolds number.

Next, we load the fish data contains in the xml files in the biology folder. Careful, this is not the xml project file. This are the xml files described above in the “Class EstimhabW” section. There are one xml file per fish and they described the preference curves. For the argumentation on the form of the relationship, report yourself to the documentation of Estimhab (one pdf file should in the folder “doc “ in HABBY).

Then, we calculate the habitat values (VH and SPU). Finally, we plot the results in a figure and we save it as a text file.

#### Hec\_ras2D.py

###### Load\_hec\_ras2d()

This is the function which takes hec-ras inputs *when the model is done in 2D*, a thing which is only possible from the fifth version of hec-ras.

#### Manage\_grid8.py

The goal of this list of function is to go from data in 1.5D in a profile form to a 2D grid. There are two main way of doing this: a) Through the usage of the triangle module b) through the definition of a middle profile used as a guide to create the grid . Method a) is implemented in create\_grid() and method b) is implemented in create\_grid\_only\_one\_profile().

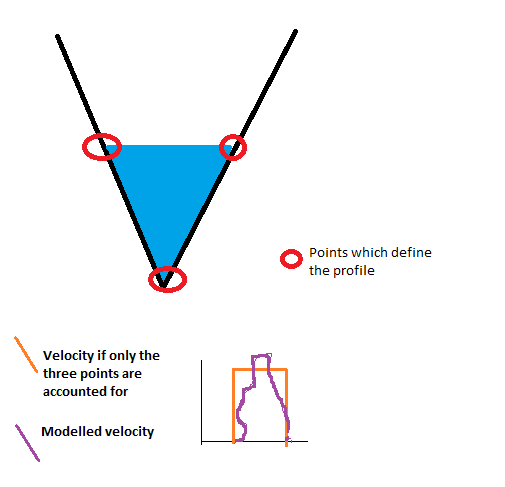
##### Walk-through for create\_grid()

The number of extra profile is the number of profile added by the function between the original profile of the hydrological model. We need at least one.

We can use this function in two modes. First, this function can create a grid for the whole profile, regardless on the wet and dry area of the profiles. It can also create a grid only for the sections of the profile which are wet, using the data from the vh\_pro variable. In this last case, we will need to cut the profile to extract only the wet area. In addition, we do not want to use only the points defining the form of the profile as guiding point for the grid. There are two reasons for this: a) HEC-RAS do not give the velocity only on the points defining the profile. b) In other 1D model, there are many cases where the velocity distribution should be composed of more points than only the ones defining the profile. See Figure 1.

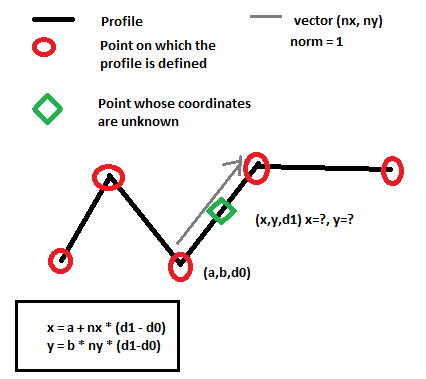
###### Update coord\_pro

This is done by the function update\_coord\_pro\_with\_vh\_pro(). We have the distance along the profile of all point of interest, but we only have the coordinate of the points defining the profile. This function gets the coordinates of all points the following way, profile by profile:

First, in case that the coordinate system is not meter, we calculate the variable coord\_change to pass from the distance in coordinate system to meter. We simply compare the distance between the first and last point of the profile with the total distance of the profile given in coord\_pro\_p[3][-1].

Figure

Then we take each point on which we know the distance. We find the point on which the profile is defined which is the closest before this this point. As the points on which is defined the profile are ordered, we can use bisect for this.

The coordinates of the under interest point is the vector sum of the point found with bisect and the difference of distance between the two point in the direction of the profile in the coordinate system of the (x,y) coordinate. The profiles are not always straight so the direction of the profile should be recalculated each time that the point found by bisect change. See Figure 2. It is not good for this function if you have two identical points in the profile so in this case it would one point of the point. However, the identical point should have been corrected before in the function related to the loading of the hydrological data.

Figure

###### Create extra profiles

Now that we have the coordinates of the points, we create the extra\_profile and all known point to vector (point\_all) which will be used by triangle. These “extra” profile are not added between the reaches. So if the river has more than one reach, we do not add profile at the junction between the reaches. In general, a separate grid is created for each reach.

For each profile in a reach, we first add the points which are already known. We do not forget to take not of the indices of the start and end of each of the “known” profile. We have three lists of indices:

* ind\_s: indices of the start of the known profile
* ind\_e: indices of the end of the known profile
* ind\_p the indices of the start of each profile (known and extra profiles)

To find the coordinates of the extra profile, we have two cases: a) we create straight extra profile b) we create extra profile in the similar form to the nearest profile. Both cases are implemented, but we use only the case b). To switch to case a), one should change the boolean all\_straight to True. We will now explain case b) as this is the one used here. Case b) is implanted in the function find\_profile\_between().

###### Find the profile between the main known profiles

To find the profile between the different profiles, we create first a line which is a straight line between the first and the last point of the profile situated before the extra-profile. This line is useful because the profiles are not always straight, so it gives a general direction.

We then project all points on this line. This is to be able to order the points when needed.

We then take each point and create a line passing by this point and perpendicular to this point. The length of this line is given by the ‘far” variable. It should be long enough but the variable “far” should be small enough to avoid problem with machine precision.

Now, we have a perpendicular line passing by one point. We find the intersection of this line with the other profileIf the point is on the profile before, the intersection will be on the profile after (in the direction of the river flow) and inversely. If there are more than one intersection possible, we take the first intersection. The method is similar is the two cases. We take each part of the profile on which we looks for an intersection (profile might not be straight). To see if there is an intersection, we use a classical algorithm based on the cross-product. It is explained for example in:

*http://stackoverflow.com/questions/563198/how-do-you-detect-where-two-line-segments-intersect*

Or any good book about computational geometry. An issue is about what to do if the segments[[1]](#footnote-1) are collinear or if the intersection is at the exact end of a segment. The function has a switch to change this behavior if needed. Currently, the function gives an intersection for these two special cases.

If not intersection is found, we create a line as a start and the end of the profile, using the direction of the last and first segment to see if we could find an intersection there. If we do not find an intersection, we note that we could not find an intersection for this point.

To create each profile, we now take each point and its intersection with the profile and we find the point between. If we only have one profile, this is the middle point. Otherwise the position of this point is given by the parameter div and div2.

We could stop here as this list of newly created point forms the added extra profile. However, nothing says that the limit of the profile is entirely coherent with the length of the basic profiles. If we want insure this (which is important for the triangle module) , we need to trim the extra-profiles using the trim switch. For this, we create a line between the first point of both “known”profile and a second line between the last points of both ‘known” profile. The direction of the profile does not matter as long as it does not change in one hydrological model. Then, we check if there is an intersection between these two lines and the extra profile. If yes, we take out the point situated before or after these lines.

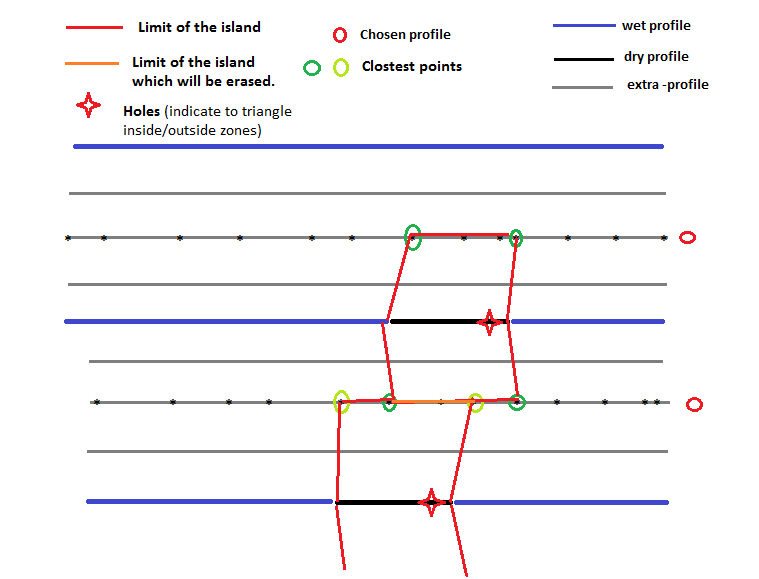
Manage islands

If we want to create a grid which reflects the wet/dry part of each profile, we need to account for the case where only part of the profile is dry. In other word, we need to account for the case where we have an island on the river. If we do a grid on the whole profile, this is not important.

As we only have info on the dry zone on the profile, we have to create a 2D zone representing the island based on the 1D information of the profile.

First we define on which extra-profile will end and start the island (indicated by the red circle in the next figure). As a summary, we take the extra-profile in the middle of both basic profiles, account for the fact that we can have an even or uneven number of profiles.

Then, for each profile, we check if there is an island, in other words, if there is a point where water height is zero, apart from the first and last water height data of the profile.

If yes, we check if we have consecutive points in the dry part of the profile. We are only interested by the first and the last point of each dry section of the profile. All the points in the middle of the dry section can be discarded. If there is a dry part in a profile, we now construct an island. For this we calculate the closest points on the selected extra\_profile of the first and last point of the island (the greens point on Figure 3). We usually have closest points, 2 on each profile. However, it is possible that the same point is the closest point of the first and the last points of the dry part of the profile. In this case, the island finishes on a peak, like a triangle.

Figure

To describe the island we use the list called seg\_island. Each segment is described by two items of the list, like the first segment starts on seg\_island[0] and ends on seg\_island[1]. So this list should be read two by two. They are four columns in this list: the first column is the index of the point. The coordinate of this point is in point\_all. So, to get the coordinates of the start point of the first segment, one should write: point\_all[seg\_island[0][0]]. The second column is on which reach is the island in case we have more than one reach. The third column is an indication if the segment is along the profile or perpendicular to the profile. If it perpendicular to the profile, it is indicated by -99, otherwise, we give the profile number. The last column is the island number, so that we can select all segments of each island. Careful with this column because we fuse together some of the islands afterwards, so this is the island number only when the ‘collinear” switch is False (see below).

The triangle module needs to know where is the inside and outside of the island. For this, it needs a point called hole indicating the island inside. The position of the hole is on the dry part of the file between the first and the last point of the profile. The exact position does not matter, but it is better to put close to one end in case the geometry of the profile is special and island have strange forms. The next step is to control that the hole is really in the island. For this, we use the inside\_polygon function. This function is an implementation of the “ray-casting” algorithm (which can be found on internet) which allows finding if a point is in a polygon. If yes, the island is kept. Otherwise, it is discarded. A warning is send as it result in a small error (the wet area is bigger than it should).

###### Various additional check

We could end here with the islands but we would have a lot of small islands which in reality would form one bigger island. To simplify the form of the grid we will now erase all ‘collinear” segment. This is not a very good term because these segments are only collinear if the profiles are straight (see the orange segment on Figure 3). Otherwise, they might cross each other. Crossing segments can be a problem for the triangle module. Triangle can handle segments crossing each other but only in simple, clear cases. It is hard to estimate in advance which kind of crossed segment will results in a failed triangle run. Hence, it is good to get of rid of as many as possible of crossed segments. This is what does this part of the code.

Practically, we select each segment “along” a particular profile. In other words, we select all segments which have the same index of the third column of the variable seg\_island. We then sort these segments so that they do not cross each other based on the fact that the points along a profile are ordered (so if you sort them, you get a list of segment on after the other).

We then have snippet of code to check if segment crosses exactly at the middle of another segments. It was useful before but is not use in the current version of the code.

We then have a script to check if segments cross as triangle can have problems with them in some cases. However, it tends to erase too different islands and to send warning which are not necessary. Moreover, it is very slow for complex cases. Hence, it is not used currently.

We then check if there is two identical points. This is somethings which will make triangle to fail in most cases so it should be avoided. If it finds identical points, it sends a warning. To keep this test quick, we use a special routing based on lexsort. Lexsort is a special way of sorting number where , if there are two time the same number, the array is sorted based on secondary info. For example, let’s say we want to sort a= [2,3,2,4] where b = [5,2,1,0] is the secondary info. The resulting sort is ind = [2,0,1,3].

###### Order the information as needed

The next part is to join all the segments (the one which gives the limits of the reaches and the ones giving the limit of the island). To give the segment which gives the limit of the reach, we use the indices collected during the creation of the profile (ind\_s and ind\_e). As a summary we define the limit of the river by taking the first and last point of each “known” profile. We also add to the list of segment representing the limits two segments along the first and the last profile to close the polygon which represent the reach. Then the segments representing the island of this reach are added. We do the same of each reach. So the limits for each reach is a polygon giving the limit of the reach and a list of polygon representing the islands

Then we have part of the code which is not used. It was written to be able to directly incorporate the limits of the substrate to the grid creation. It avoids to having to cut the substrate in a second step, which is easier and quicker. However, the resulting grid was not so smooth because of the number of constraint on it. So it is not used.

Then, we finally send the information to triangle to create the grid. We do this separately for each reach. There are three lists to be sent as information: a) the coordinate of all points. It can seem quicker to send only the points related to the analyzed reach. However, it would really slow down the search for the point which overlaps between the reach, so it is easier to send all point for all reach. Triangle will stop the grid at the limit regardless of the position of the points. We then send a list of segment representing the limits (reach limit and island). Finally, we send a list of point which define the inside and outside of the reach (the “holes”). There is a long list of option for triangle. In case, triangle fails, it can be useful to try to run it without the ‘p’ switch so without option. It will ignore the islands in this case and create a bigger grid than needed, but it can be useful to see where the problems are.

We then find the area which overlaps between the reaches. It might be important afterwards when we calculate the surface of the river.

Next, we have a part of the code to find the centroid of the grid elements, which can be useful.

The last part is a special form of “return” which is necessary when we use an external process to send create\_grid(). It is often done so because triangle can fail in difficult cases.

#### Create\_grid\_only\_1\_profile

This is a similar function than create\_grid(), but it uses only one extra profile and directly create the grid instead of calling triangle. Hence, it is more stable, but there is little control on the quality of the grid, For example, if two cells overlaps because of a bug or a complicated geometry, this function will not send a warning which this would not happen with triangle output.

The first part of the function is the same than create grid (see section above about “Update coord\_pro). The aim here is to reformat the profile to get the coordinate of all interesting points.

As in create\_grid, we create one grid by reach, so there is a “for” loop passing through each reach. Also, in a similar way, we can create a grid for the whole porfile or only for the wet area of the profile. If the wet area of the profile is analyzed, the variable vh\_pro is needed.

We then prepare the water height and velocity data. More precisely, we take out all the data where the profile is dry. This will be useful afterwards, when the island will be constructed. In create\_grid\_only\_1\_profile(), we do not interpolate velocity and height data, contrarily to create\_grid(). So in the former case, we need to update the data to account for this fact.

Afterwards, we add one extra profile in the middle of each couple of profile. We assume that the profile does not cross. We do that using the same function than the function which add the extra profile in create\_grid(). See the section about “Find the profile between the main known profiles“.

We then create the grid using the extra profile created before. The general idea is to create two rows of triangle, one before the added profile and one after. The position of the triangle is determined by the position of the velocity and height data. Each triangle has one velocity and height data linked with it. Because of that, there is no need for interpolation of velocity and height data as in create\_grid().

To create the grid, we take a vector perpendicular to the profile. If the profile is not straight, we take the vector perpendicular to the first segment of the profile. It could look more logical to take the vector perpendicular to each segment of the profile, but the grid cells might cross on each other in this case. Hence, it is better to use only one direction by profile.

Next, we find the intersections between the line parallel to this perpendicular vector and passing by each point of the profile. The method to find the intersection is similar to the one described in about “Find the profile between the main known profiles“. If do not find an intersection for a point, the intersection is found arbitrarily.

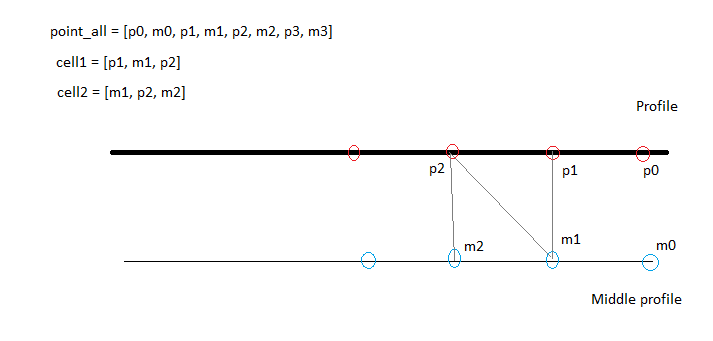
Now, we have the point forming the grid and they are in order. So we can create the connectivity table (called ikle here). For each point, we form two triangular cells (Figure 4). The creation on the connectivity table is based on the order of the variable point\_all which is: one point of the profile, the point which intersect in the middle profile, the next profile point, … So for the first cell we have the profile point (length of point\_all -1), the point of the profile before (length – 3) and the intersection point on the middle profile related to the point before (length -2). For the second cell, we have the intersection point related to this profile (length of point\_all), the intersection point related to the point before and the point of the profile. 

Figure 4

When the grid is created, we fill it with the data. As there is one data point per cell, this can be done directly as long as we are in a part of the profile with water.

The next part of the code is commented and not used, but it can be used to show the profile created (without the full grid) to check if this part was done successfully by HABBY.

# Translation of HABBY

# Log and HABBY on the command line

* Print statement

# HABBY as a .exe (single application)

# Git and version control

# Random notes

* For performance, avoid np.mean and np.where whenever possible.
* If use np.where, do not forgeot to add a zero: np.where(b==a)[0]
* Np.array are not list. In HABBY, I use both, so it is good to be careful to distinguish the two.

# Needed Python module and license

# Useful reference and books

1. Here I talk about line and segments without making a lot of difference. This is because “line” are not infinite in this function, their length is defined by the “far” variable. So, for the computer, they are all segments even if the length of some segments does not matter and could be conceptualized as line in our mind. [↑](#footnote-ref-1)