HABBY – Technical documentation

Table des matières

[How to execute HABBY 1](#_Toc464723843)

[Source Code 1](#_Toc464723844)

[Source for the graphical user interface 2](#_Toc464723845)

[Source for the calculation 4](#_Toc464723846)

[Translation of HABBY 8](#_Toc464723847)

[Log and HABBY on the command line 8](#_Toc464723848)

[HABBY as a .exe (single application) 8](#_Toc464723849)

[Git and version control 8](#_Toc464723850)

[Random notes 8](#_Toc464723851)

[Needed Python module and license 9](#_Toc464723852)

[Useful reference and books 9](#_Toc464723853)

# 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

# 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).

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 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 call the function to distribute the velocity and manage the log data.

.

## Source for the calculation

#### 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().

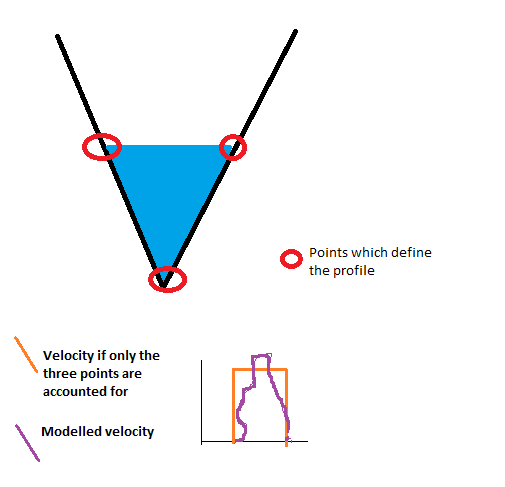
##### Walk-through for create\_grid()

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 th evh\_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.

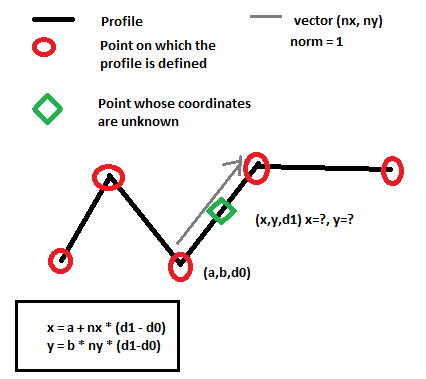
###### Update coord\_pro

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.

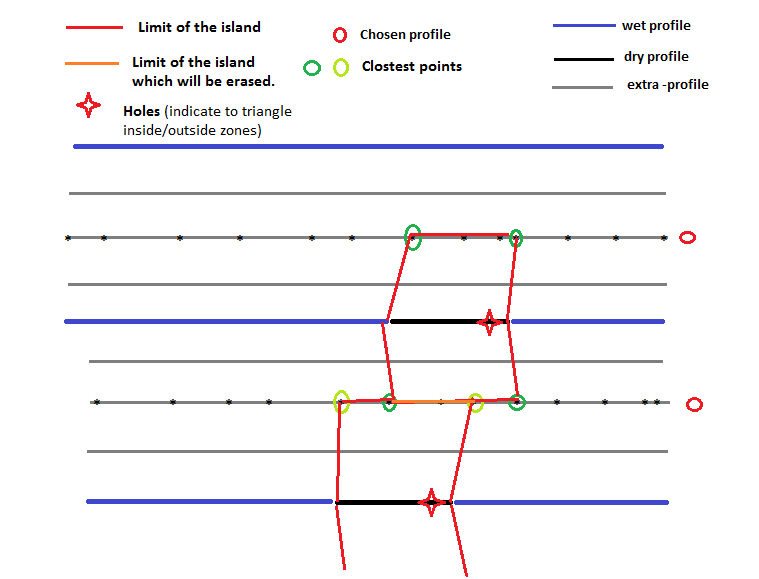
Manage islands

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.

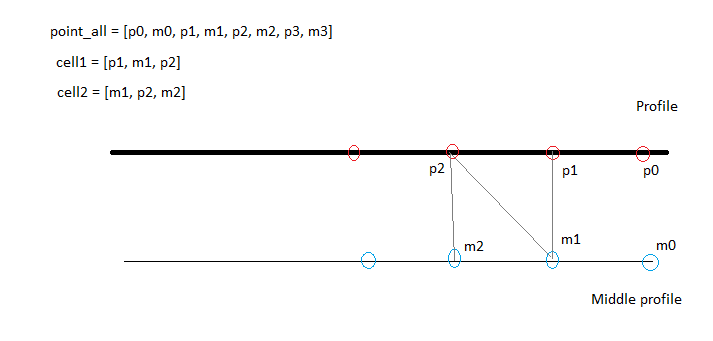
Now, we have the point forming the grid and they are in order. So we can create the connectivity table (called ikle here). For each point, we form two triangular cells (Figure 4). The creation on the connectivity table is based on the order of the variable point\_all which is: one point of the profile, 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.

# 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)