HABBY Documentation

Release 1

Diane von Gunten, Yann Le Coarer and Fabrice Zaoui

Jan 09, 2017

CONTENTS

HABBY is a program to estimate the habitat of fish using various hydrological models and preference curve as input.

CHAPTER

ONE

HOW TO EXECUTE HABBY

To execute HABBY:

- Go to folder which contains habby.py using 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 you want to be sure to have the same version of the module than originally, go to the folder zen_file/wheele with the command line and install the missing module from there (something similar to "pip install -m *.whl*"). Not all modules are in this folder, only the ones which were difficult to install.

CHAPTER

TWO

MAIN( ) AND 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 (xmind should be installed).

The main of HABBY is habby.py. It has the usual form for an application using PyQt5. The main() creates an application of QWidget and call the Main_Windows class, which we will discuss shortly. The last line closes the application.

CHAPTER

THREE

GRAPHICAL INTERFACE

Here is the list of all modules contains in the src_GUI folder.

3.1 Main_windows of HABBY

in src_GUI/Main_Windows_1.py

class `src_GUI.Main_windows_1.`**`CentralW`**(*rech*, *path_prj*, *name_prj*)

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

Parameters

- **`rech`** – A bollean which is True if the tabs for the "research option" are shown. False otherwise.
- **`path_prj`** – A string with the path to the project xml file
- **`name_prj`** – A string with the name of the project

Technical comments

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 attributes to add a new tab.

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 this are 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.

Show_fig is mostly a "plt.show()". To avoid problem between matplotlib and PyQt, it is however important that matplotlib use the backend "Qt5Agg" in the .py where the "plt.plot" is called. Practically, this means modifying the matplotlib import.

Showfig shows only one figure. To show all existing figures, one can call the function show_fig2 from the menu. Show_fig2 call the instance child_win of the class ShowImageW to open a new Windows with all figure. However, this would only show the figure without any option for the zoom.

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, these connections are written in the function "connect_signal_log", where the signal for a new class can be added.

When this is done, the info for the general tab (created before) is filled. If the user has opened a project in HABBY before, the name of the project and the other info related to it will be shown on the general tab. If the general tab is modified in the class WelcomeW(), this part of the code which fill the general tab will probably needs to be modified.

Finally, each tab is filled. The tabs have been created before, but there were 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 more 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 explained in the section about the log.

`connect_signal_log``()`
connect all the signal linked to the log. This is in a function only to improve lisibility.

`init_iu``()`
A function to initilize an instance of CentralW. Called by __init___().

`optfig``()`
A small function which open the output tab. It contains the different options for the figures. Output should be the 6th tab, otherwise it will not work.

`scrolldown``()`
Move the scroll bar to the bottow if the ScollArea is getting bigger

`showfig``()`
A small function to show the last figure

`showfig2``()`
A function to see all saved figures without possibility to zoom

`write_log``(`*text_log*`)`
A function to write the different log. Please read the section of the doc on the log.

Parameters **`text_log`** – the text which should be added to the log (a string)

- if text_log start with # -> added it to self.l2 (QLabel) and the .log file (comments)
- if text_log start with restart -> added it restart_nameproject.txt
- if text_log start with WARNING -> added it to self.l2 (QLabel) and the .log file
- if text_log start with ERROR -> added it to self.l2 (QLabel) and the .log file
- if text_log start with py -> added to the .log file (python command)
- if text_log start with nothing -> just print to the Qlabel
- if text_log out from stdout -> added it to self.l2 (QLabel) and the .log file (comments)

if logon = false, do not write in log.txt

`write_log_file``(`*text_log*, *pathname_logfile*`)`
A function to write to the .log text. Called by write_log.

Parameters

- **`text_log`** – the text to be written (string)
- **`pathname_logfile`** – the path+name where the log is

class `src_GUI.Main_windows_1.`**`EmptyTab`**
This class is used to fill empty tabs with something during the developement. It will not be use in the final version.

addtext()
This function print a string on the command line. This is useful if you need to check if a button (or similar). is connected.

init_iu()
Used in the initialization.

class src_GUI.Main_windows_1.**MainWindows**
The class MainWindows contains the menu and the title of all the HABBY windows. It also create all the widgets which can be called during execution

Technical comments and walk-through

First, we 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. Qsettings also function with Apple 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 the section "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.

clear_log()
Clear the log in the GUI.

closeEvent(*event*)
Close the program better than before (where it used to crash about 1 times in ten). It is not really clear why.

Parameters **event** – managed by the operating system.

close_rech()
Close the additional research menu (see open_rech for more information)

do_log(*save_log*)
Save or not save the log

Parameters **save_log** – an int which indicates if the log should be saved or not

- 0: do not save log
- 1: save the log in the .log file and restart file

erase_pict()
All files contained in the folder indicated by path_im will be deleted.

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.

`init_ui``()`

Used by __init__() to create an instance of the class MainWindows

`my_menu_bar``()`

This function creates the menu bar of HABBY.

`open_rech``()`

Open the additional research tab, which can be used to create Tab with more experimental contents.

Indeed, 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.

`save_project``()`

A function to save the xml file with the information on the project

Technical comments

This function saves or creates the xml file related to the projet. In this xml file, there are 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.

When HABBY open, there are therefore 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 path are a) the biologie path (named “biologie” by default) which contains the biological information such as the preference curve and b) the path_im which is the path where all figures and most outputs 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``()`

A function to save the information linked with Estimhab in an hdf5 file.

Technical comments

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.

setlangue(*nb_lang*)

A function which change the language of the programme. It change the menu and the central widget. It uses the self.lang attribute which should be set to the new language before calling this function.

Parameters **nb_lang** – the number representing the language (int)

- 0 is for English
- 1 for French
- n for any additionnal language

test_entry_float(*var_in*)

An utility function to test if var_in are float or not the boolean self.does_it_work is used to know if the functions run until the end.

Parameters **var_in** – the QlineEdit which contains the data (so var_in.text is a string)

Returns the tested variable var_in

class src_GUI.Main_windows_1.**ShowImageW**(*path_prj*, *name_prj*)

The widget which shows the saved images. Used only to show all the saved figure together iwhtout zoom or other options.

Technical comments

The ShowImageW() class is used to show all the figures created by HABBY. It is a class which can only be called from the menu (In Option/Option Image). This is not the usual way of opening a figure which is usually done by plt.show from matplotlib. This is the way to look at all figures together, which can be useful, even if zooming is not possible anymore.

To show all image, HABBY open a separate window and show the saved image in .png format. 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 "selectionchange" changes the figure shown based on the user action.

change_folder()

A function to change the folder where are stored the image (i.e., the path_im)

init_iu()

Used in the initialization.

selectionchange(*i*)

A function to change the figure shown by ShowImageW() :return:

send_log

A PyQt signal used to write the log

update_namefig()

This function add the different figure name to the drop-down list.

class src_GUI.Main_windows_1.**WelcomeW**

The class WeLcomeW() creates the first tab of HABBY (the tab called "General"). This tab is there to create a new project or to change the name, path, etc. of a project.

init_iu()
Used in the initialization of a new instance of the class WelcomeW()

save_signal
A PyQt signal used to save the figure

send_log
A PyQt signal used to write the log

setfolder()
This function is used by the user to select the folder where the xml project file will be located.

`src_GUI.Main_windows_1.`**new_project**()

`src_GUI.Main_windows_1.`**open_project**()

3.2 Hydrological information - GUI

in src_GUI/hydro_GUI_2.py

This python module contains the class which forms the hydrological tab in HABBY. It contains the information for the graphical interface and make the link with the scripts used for the hydrological calculations.

class `src_GUI.hydro_GUI_2.`**FreeSpace**
Bases: `PyQt5.QtWidgets.QWidget`

Simple class with empty space, just to have only Qwidget in the stack.

Technical comment

The idea of this class is that the user see a free space when it opens the "Hydro" Tab instead of directly seeing one of the hydraulic model. The goal 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 selected by default 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 `src_GUI.hydro_GUI_2.`**HEC_RAS1D**(*path_prj*, *name_prj*)
Bases: *src_GUI.hydro_GUI_2.SubHydroW*

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. The class HEC_RAS1D inherits from SubHydroW() so it have all the methods and the variables from the class ubHydroW(). The class hec-ras 1D is added to the self.stack of Hydro2W(). So the class Hec-Ras 1D is called when the user is on the hydrological tab and click on hec-ras1D as hydrological model.

init_iu()
This function is called by __init__() durring the initialization.

Technical comment

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.

Hec-Ras is a 1.5D model and so HABBY create a 2D grid based on the 1.5D input. The user can choose the interpolation type 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.

`load_hec_ras_gui``()`

A function to execute the loading and saving of the HEC-ras file using Hec_ras.py

Technical comments

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). It also 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). It creates the hdf5 with the loaded data. Finally, if necessary, it shows the figure by emitting a signal. This signal is collected in the MainWindow() class.

`show_fig`

PyQtsignal to show the figure.

class `src_GUI.hydro_GUI_2.`**`HEC_RAS2D`**`(`*path_prj*, *name_prj*`)`

Bases: *`src_GUI.hydro_GUI_2.SubHydroW`*

The class hec_RAS2D is there to manage the link between the graphical interface and the functions in src/hec_ras2D.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.

`init_iu``()`

This method is used to by __init__() during the initialization.

`load_hec_2d_gui``()`

This function calls the function which load hecras 2d and save the names of file in the project file. It is similar to the function to load_rubar2D.

`show_fig`

PyQtsignal to show the figures.

class `src_GUI.hydro_GUI_2.`**`Hydro2W`**`(`*path_prj*, *name_prj*`)`

Bases: `PyQt5.QtWidgets.QWidget`

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.

List of model supported by Hydro2W: files separetly. However, sometime the file was not found * Telemac (2D) * Hec-Ras (1.5D et 2D) * Rubar BE et 2(1D et 2D) * Mascaret (1D) * River2D (2D)

Technical comments

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" calls 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 selection_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 infomation which are needed.

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.

`get_new_hydro_hdf5``()`
This is a function which allows the user to select an hdf5 file containing the hydrological data from a previous project and add it to the current project. It modifies the xml project file and test that the data is in correct form by loading it. The hdf5 should have the same form than the hydrological data created by HABBY in the method save_hdf5 of the class SubHydroW.

`give_info_model``()`
A function to show extra information about each hydrological model. The information should be in a text file with the same name as the model in the model_hydo folder. General info goes as the start of the text file. If the text is too long, add the keyword "MORE INFO" and add the longer text afterwards. The message box will show the supplementary information only if the user asks for detailed information.

`init_iu``()`
Used in the initialization by __init__()

`selectionchange``(`*i*`)`
Change the shown widget which represents each hydrological model (all widget are in a stack)

Parameters **`i`** – the number of the model (0=no model, 1=hecras1d, 2= hecras2D,...)

`send_log`
A PyQt signal to send the log.

class `src_GUI.hydro_GUI_2.`**`Mascaret`**(*path_prj*, *name_prj*)
Bases: *`src_GUI.hydro_GUI_2.SubHydroW`*

The class Mascaret is there to manage the link between the graphical interface and the functions in src/mascaret.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 (see this class for more information). 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 (see the method load_manning_text of the class SubHydroW for more information)

`init_iu``()`
Used in the initialization by __init__()

load_mascaret_gui()
The function is used to load the mascaret data, calling the function contained in the script mascaret.py

show_fig
A PyQtsignal to show the figure.

class src_GUI.hydro_GUI_2.**River2D**(*path_prj*, *name_prj*)
Bases: *src_GUI.hydro_GUI_2.SubHydroW*

The class River2D t is there to manage the link between the graphical interface and the functions in src/river2D.py which loads the River2D data in 2D.

Technical comments

The class River2D inherits from SubHydroW() so it have all the methods and the variables from the class SubHydroW(). It is similar generally to the hec-ras2D class. However, the hydrological model River2D create one file per time step. Hence, it is necessary to have a way to load all the files automatically. Loading one file after one file would be annoying. There are four functions to manage the large number of file:

- add_all_file: find all files in a folder selected by the user.
- add_file_river2D: add just one selected file
- Remove_all_file: remove all selected files
- Remove_file: remove one selected file

None of this four functions 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.

add_all_file()
The function finds all .cdg file in one directory to add there names to the list of files to be loaded

add_file_river2d()
This function is used to add one file to the list of file to be loaded. It calls show_dialog, prepare some data for it and update the QWidgetList with the name of the file containted in the variable self.namefile.

add_file_to_list()
This function to add all file contained in self.namefile to the QWidgetlist. Called by add_file_river2D and add_all_file.

init_iu()
used by __init__ in the initialization

load_river2d_gui()
This function is used to load the river 2d data.

remove_all_file()
This function removes all files from the list of files to be loaded and from the QlistWidget.

remove_file()
This is small function to remove a .cdg file from the list of files to be loaded and from the QlistWidget.

show_fig
A PyQtsignal to show the figure.

class src_GUI.hydro_GUI_2.**Rubar1D**(*path_prj*, *name_prj*)
Bases: *src_GUI.hydro_GUI_2.SubHydroW*

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.

`init_iu``()`
Used in the initalizatin by __init__()

`load_rubar1d``()`
A function to execute the loading and saving the the rubar file using rubar.py. After loading the data, it distribute the velocity along the profiles by calling self.distribute_velocity() and it created the 2D grid by calling the method self.grid_and_interpo.

`show_fig`
A PyQtsignal to show the figures.

class `src_GUI.hydro_GUI_2.`**`Rubar2D`**`(`*path_prj*, *name_prj*`)`
Bases: *`src_GUI.hydro_GUI_2.SubHydroW`*

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.

`init_iu``()`
used by ___init__() in the initialization.

`load_rubar``()`
A function to execture the loading and saving the the rubar file using rubar.py. It is similar to the load_hec_ras_gui() function. Obviously, it calls rubar and not 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. Moreover, 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 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.

`propose_next_file``()`
This function proposes the second RUBAR file when the first is selected. 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 when more than one extension is possible, so it has not been done yet.

`show_fig`
A PyQtsignal to show the figure.

class `src_GUI.hydro_GUI_2.`**`SubHydroW`**`(`*path_prj*, *name_prj*`)`
Bases: `PyQt5.QtWidgets.QWidget`

SubHydroW is class which is the parent of the classes which can be used to open the hydrological models. This class is a bit special. It is not called directly by HABBY but by the classes which load the hydrological data and which inherits from this 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.

In other word, there are MainWindows() which provides the windows around the widget and Hydro2W which provide the widget for the hydrological Tab and one class by hydrological model to really load the model. The latter classes have various methods in common, so they inherit from SubHydroW, this class.

`distribute_velocity``()`
This function make the link between the GUI and the functions of dist_vitesse2. It is used by 1D model,

notably rubar and masacret.

Dist vitess needs a manning parameters. It can be given by the user in two forms: a constant (float) or an array created by the function load_manning_text.

`drop_hydro`
A PyQtsignal signal for the substrate tab so it can account for the new hydrological info.

`find_path_im``()`
A function to find the path where to save the figues, careful a simialr one is in estimhab_GUI.py. By default, path_im is in a folder calls "Figure_Habby".

`grid_and_interpo``(`*cb_im*`)`
This function forms the link between GUI and the various grid and interpolation functions. Is called by the "loading' function of hec-ras 1D, Mascaret and Rubar BE. :param cb_im: A boolean if true, the figures are created and shown.

Technical comment to be added

`load_manning_text``()`
This function loads the manning data in case where manning number is not simply a constant. In this case, the manning parameter is given in a .txt file. The manning parameter used by 1D model such as mascaret or Rubar BE to distribute velocity along the profiles. The format of the txt file is "p, dist, n" where p is the profile number (start at zero), dist is the distance along the profile in meter and n is the manning value (in SI unit). One point per line so something like:

0, 150, 0.035

0, 200, 0.025

1, 120, 0.035, etc.

White space is neglected and a line starting with the character # is also neglected.

`read_attribute_xml``(`*att_here*`)`
A function to read the text of an attribute in the xml project file.

Parameters **`att_here`** – the attribute name (string).

`save_hdf5``()`
This function save the hydrological data in the hdf5 format.

Techincal comments

This function cannot be used outside of the class, so it needs to be re-written if used from the command line.

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, the 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.

Hdf5 file do not support unicode. It is necessary to encode string to write them in ascii.

`save_xml`(*i=0*, *append_name=False*)
A function to save the loaded data in the xml file.

This function adds 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 saved, or if it cannot find the project file). Then, it opens the xml file and add the path and name of the file to this xml file. If the 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.

Parameters

- **`i`** – a int for the case where there is more than one file to load
- **`append_name`** – A boolean. If True, the name found will be append to the existing name in the xml file, instead of remplacing the old name by the new name.

`send_err_log`()
This function sends the errors and the warnings to the logs. The stdout was redirected to self.mystdout.

`send_log`
A Pyqtsignal to write the log.

`show_dialog`(*i=0*)
A function to obtain the name of the file chosen by the user. This method 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.

Parameters **`i`** – a int for the case where there is more than one file to load

`was_model_loaded_before`(*i=0*, *many_file=False*)
A function to test if the model loaded before. If yes, it updates the attibutes anf the widgets of the hydrological model on consideration.

Parameters

- **`i`** – an int used in cases where there is more than one file to load (geometry and output for example)
- **`many_file`** – A bollean. If true this function will load more than one file, separated by ','. If False, it will only loads the file of one model (see the comment below).

Technical comment

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.

class `src_GUI.hydro_GUI_2.`**`SubstrateW`**(*path_prj*, *name_prj*)

Bases: *`src_GUI.hydro_GUI_2.SubHydroW`*

This is the widget used to load the substrate. It is practical to re-use some of the method from SubHydroW. So this class inherit from SubHydroW.

`get_att_name`()

A function to get the attribute name of the shapefile which contains the substrate data. it is given by the user in the GUI.

`init_iu`()

Used in the initialization by __init__().

`load_sub_gui`()

This function is used to load the substrate data. The substrate data can be in two forms: a) in the form of a shp file form ArGIS (or another GIS-program). b) in the form of a text file (x,y, substrate data line by line). Generally this function has some similarities to the functions used to load the hydrological data and it re-uses some of the methods developed for them.

`log_txt`()

This function gives the log for the substrate in text form. this is in a function because it is used twice in the function load_sub_gui()

`save_hdf5_sub`()

This function save the substrate data in its own hdf5 file and write the name of this hdf5 file in the xml project file. The format of the hdf5 file is not finalzed yet so it is not documented.

`send_merge_grid`()

This function calls the function merge grid in substrate.py. The goal is to have the substrate and hydrological data on the same grid. Hence, the hydrological grid will need to be cut to the form of the substrate grid.

`show_fig`

A PyQtsignal to show the figures.

`update_hydro_hdf5_name`()

This is a short function used to read all the hydrological data contained in an hdf5 files and available in one project. When these files are read, they are added to the drop-down menu; This should be a function because an update to this list can be triggered by the loading of a new hydrological data. The class SubstrateW() noticed this through the signal drop_hydro send by the hydrological class. The signal drop_hydro is connected to this function in the class CentralW in MainWindows.py. Indeed, it is not possible to do it in SubstrateW().

class `src_GUI.hydro_GUI_2.`**`TELEMAC`**(*path_prj*, *name_prj*)

Bases: *`src_GUI.hydro_GUI_2.SubHydroW`*

The class Telemac is there to manage the link between the graphical interface and the functions in src/selafin_habby1.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 from Telemac is on the node as in HABBY.

init_iu()
Used by __init__() during the initialization.

load_telemac_gui()
The function which call the function which load telemac and save the name of files in the project file

show_fig
A PyQtsignal to show the figure.

3.3 Figure Option - GUI

in src_GUI/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 colour or the size of the text.

src_GUI.output_fig_GUI.**create_default_figoption**()
This function creates the default dictionnary of option for the figure.

src_GUI.output_fig_GUI.**load_fig_option**(*path_prj*, *name_prj*)
This function loads the figure option saved in the xml file and create a dictionnary will be given to the functions which create the figures to know the different options chosen by the user. If the options are not written, this function uses data by default which are in the fonction create_default_fig_options().

Parameters

- **path_prj** – the path to the xml project file
- **name_prj** – the name to this file

Returns the dictionary containing the figure options

class src_GUI.output_fig_GUI.**outputW**(*path_prj*, *name_prj*)
The class which support the creation and management of the output. It is notably used to select the otions to create the figures.

init_iu()

save_option_fig()
A function which save the options for the figures in the xlm project file. The options for the figures are contained in a dictionnary. The idea is to give this dictinnory in argument to all the fonction which create figures. In the xml project file, the options for the figures are saved under the attribute "Figure_Option".

send_log
A PyQtsignal used to write the log.

3.4 The Stathab model - GUI

class src_GUI.stathab_GUI.**StathabW**(*path_prj*, *name_prj*)
The class to load and manage the widget controlling the Stathab model.

Technical comments

The class StathabW makes 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 called '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.

`add_all_fish``()`

This function add the name of all known fish (the ones in Pref.txt) to the QListWidget.

`add_fish``()`

This function add the name of one fish species to the selected list of fish species.

`init_iu``()`

`load_from_hdf5_gui``()`

This function calls from the GUI the load_stathab_from_hdf5 function. In addition to call the function to load the hdf5, it also updates the GUI according to the info contained in the hdf5.

Technical comments

This functino updates the Qlabel similarly to the function "load_from_txt_gui()". It also loads the data calling the load_stathab_from_hdf5 function from the Stathab class in src. The info contains in the hdf5 file are now in the memory in various variables called self.mystathab."something". HABBY 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 (contained 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 it checks if all the variables were found or if some were missing

`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.

Technical comments

Here is the list of the modifications done to the graphical user interface before calling the load_function of Stathab.

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. 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 (apart from Pref.txt) 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 the src folder). 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

`reach_selected()`
A function which indcates which files are linked with which reach.

Technical comment

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 shows it 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.

`remove_fish()`
This function remove the name of one fish species to the selected list of fish species.

`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 fishes 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. Finally, it writes the log.

`select_dir()`
This function is used to select the directory and find the files to laod stathab from txt files. It calls load_from_txt_gui() when done.

`select_hdf5()`
This function allows the user to choose an hsdf5 file as input from Stathab.

Technical comment

This function is for example useful if the user would have created an hdf5 file for a Stathab model in another project and he would like to send the same model on other fish species.

This function writes the name of the new 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 called load_from_hdf5_gui.

`send_err_log()`
Send the errors and warnings to the logs. It is useful to note that the stdout was redirected to self.mystdout.

`send_log`
A PyQtsignal used to write the log.

`show_fig`
A PyQtsignal used to show the figures.

3.5 Estimhab - GUI

in src_GUI/estimhab_GUI.py

class `src_GUI.estimhab_GUI.`**`EstimhabW`**(*path_prj*, *name_prj*)

The Estimhab class provides the graphical interface for the version of the Estimhab model written in HABBY. The Estimhab model is described elsewhere. EstimhabW() just loads the data for Estimhab given by the user.

`add_fish`()

The function is used to select a new fish species

`change_folder`()

A small method to change the folder which indicates where is the biological data

`find_path_im_est`()

A function to find the path where to save the figues. Careful there is similar function in hydro_GUI_2.py. Do not mix it up

Returns path_im a string which indicates the path to the folder where are save the images.

`init_iu`()

This function is used to initialized an instance of the EstimhabW() class. It is called be __init__().

Technical comments and walk-through

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 for Estimhab. 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.

`remove_fish`()

The function is used to remove fish species

`run_estmihab`()

A function to execute Estimhab by calling the estimhab function.

Technical comment

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.

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 a float, etc.). We then remove the duplicate fish species (in case the user select one specie twice) and the Estimhab model is called. 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).

`save_signal_estimhab`
PyQtsignal to save the Estimhab data.

`send_log`
PyQtsignal to write the log.

`show_fig`
PyQtsignal to show the figures.

3.5.1 Biological data - Estimhab

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.

Estimhab is a statistical model, which functions using mathematical regressions. The different regressions (or preference curve) 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 + sum 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 coefficients 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 HABBY.
- 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 for Q50, natural median discharge
  - 1 for H50, the height of the stream at q50
  - 2 for L50, the width of the stream at q50
  - 3 for V50, the velocity of the stream at q50
  - 4 for Re50, the discharge divided by 10 times the width at Q50
  - 5 for Fr50, the Froude number at Q50

- 6 for Dh50, the mean substrate height divided by h50
- 7 for Exp(Dh50). Erase the log() of this particular term of the constant

CHAPTER

FOUR

CALCULATION OF FISH'S HABITAT

The src folder contains the python module which are not linked with the graphical user interface.

4.1 Hec-ras model 1D

in src/Hec_ras06.py

This module contains the functions used to load the outputs from the hec-ras model in 1.5D.

`src.Hec_ras06.`**`coord_profile_non_georeferenced`**(*data_bank_all*, *data_dist_all*, *data_river_all*, *data_profile_all*, *nb_pro_reach*)

This is a function to create the coordinates of the profile in the non-georeferenced case. This function is called by open geo_file(). Hypothesis: The profile are straight and perpendicular to the river. The last profile is at the end of the river.

Parameters

- **`data_bank_all`** – distance along the profile of bank station
- **`data_dist_all`** – the distance between the profile (left, center channel, right)
- **`data_river_all`** – the coordinate of the river
- **`data_profile_all`** – the (d,z) data of the profile
- **`nb_pro_reach`** – the number of profile by reach

Returns the coordinates of the profile

Technical comments

For each profile, we create an array composed of five points: Start of profile, left bank, intersection between river and profile, right bank and end of profile. The intersection with the river is directly given as input to the function. Then we find the vector perpendicular to this river and we get the four other points on the same line.

To get the distance for these four other point, we must be careful to pass from the distance given in meter and the distance in the model coordinates (scaled between [0, 1] usually). The way to go from one coordinate system to another is to use the "alpha" variable. We only need to correct distance, no problem with a system of coordinate which would not be in the same direction (as the data is given along a profile). The river passes in the middle of the right and left bank, so we can find where is left and right bank is. Because we know the total length of the profile, we can also find the beginning and end of the profile.

`src.Hec_ras06.`**`figure_xml`**(*data_profile*, *coord_pro_old*, *coord_r*, *xy_h_all*, *zone_v_all*, *pro*, *path_im*, *nb_sim=0*, *name_profile='no_name'*, *coord_p2=-99*)

A function to plot the results of the loading of hec-ras data.

Parameters

- **`data_profile`** – (list with np.array)
- **`coord_pro_old`** – (x,y) data of the profile (list with np.array)
- **`coord_r`** – (x,y) data of the river (list with np.array)
- **`xy_h_all`** – (x,y, h) for the height data for each simulation (list with np.array)
- **`zone_v_all`** – (x,y, v) for the velocity data. velocity is by zone of profile. for each simulation. the (x,y) indicates the start of the zone which end with the next velocity
- **`pro`** – a list of int with the profile whcih should be ploted [2,3,4]
- **`nb_sim`** – which simulation should be plotted. In fact, it often relates to the time step.
- **`name_profile`** – a list of string with the name of the profiles
- **`coord_p2`** – the data of the profile when non geo-referenced, optional
- **`path_im`** – the path where the figure should be saved (string)

Technical comments

We first choose the size of the font to be written. At term, it should be given by the options.

Two main groups of figure will be done: One list of figure with the form of the profil, the water height, and the velocity for the chosen profiles and one (x,y) view of the position of each profile.

We chose the time step to be written (the variable nb_sim here). The variable pro is a list which says which profiles are to be plotted. Hence, we get the velocity and water height for the time step and profile of interest.

To plot the velocity, we first get the distance along the profile where the water level cut the profile elevation. This is the variable xint1 and xint2. We then get the velocity data for the region under the water. We add three points for velocity at 0, xint1 and xint2. We then used the step function to plot the vecloity. Because of the added point, we will have a zero velocity from 0 to xint1, then the velocity data, then again zeros from xint2 to the end.

To plot the elevation of the profile, we plot the variable xz and we use the function fill_between to fill in blue the region under water. This function creates a line at the water elevation and fills in blue between this line and the profile elevation. We add some titles and save the figures.

For the second type of figure (view in x,y coordinates), We first plot the river position which is saved in the coord_r variable. Then we plot the coordinate of each profile and their names. If the name of the profile is not known, we plot the profile number. We also plot the position of each velocity data and height data (as it could be useful). If the figure gets too complicated, this can be taken away by changing the two lines which finish with height or velocity as comment. We add some titles and save the figures.

`src.Hec_ras06.`**`find_coord_height_velocity`**(*coord_pro*, *data_profile*, *vel*, *wse*, *nb_sim*, *max_vel_dist=0*)

This function finds the coordinates of the height/velocity. In hec-ras outputs the data are often written in the form (profile, distance along the profile, data). This function passes this type of information in the usual coordinate form.

Parameters

- **`coord_pro`** – the coordinate (x,y) of the profile. List of np.array.
- **`data_profile`** – data concening the geometry of the profile, notably its elevation (x,z). List of np.array.
- **`vel`** – the velocity data. List of np.array.
- **`wse`** – the water surface elevation. List of np.array.
- **`nb_sim`** – the number of simulation in case there is more than one

- **max_vel_dist** – the minimum number of velocity point by ten meter before a warnings appears

Returns for each simulation, a list of np.array representing (x,y,v) and (x,y,h,)

Technical comments

This is a function called after having loaded the data. Hec-Ras present the data in (profil, distance along profile, data) form for the height. For the velocity, it is similar but the distance is given by a number between 0 and 1 (0 is the start of the profile, 1 is the end of the profile). This function transforms this data in the form (x,y, dist, data) using the (x,y) coordinates given in the coord_pro variable. In other word, we have the coordinate of the profile, not of the coordinates of the height and velocity data.

First, we get the distance between all points in (x,y) system. Then, we get the length of the profile in meter or feet. It is possible to have a (x,y) coordinate system in a different unit. Hence, the length of the profil is valid for the (profile, distance along profile, data) view. We multiply the velocity distance data by this length. Hence, the distance information is now in meter or feet along the profile for water height and velocity.

There are some lines added to account for the last and first points of the profile (annoying in hec-ras). We then calculate the new coordinates. For each velocity and water height point, we find the last known point in the (x,y) coordinates. We do a vectorial addition from this point plus the vector between this point and the next multiplied by the distance from this point to the point that we tried to calculate. The variable alpha is used to pass from one coordinate system to the next one.

Careful the height is on the node and the velocity is by zone.

`src.Hec_ras06.`**`get_rid_of_white_space`**(*stream_str*)

This is a small fonction to get rid of white space at the end of name which could contain white space. Not used anymore as str.strip() functions well. But, as it was done already, we let it here.

Parameters **stream_str** – the name of the string

Returns the same name without white space.

`src.Hec_ras06.`**`load_xml`**(*xml_file*, *path*)

This is a function used by openxml_file and opengml_file to load an xml file.

Parameters

- **xml_file** – the name of an xml file (string)
- **path** – the path where the xml file is (string)

Returns the loaded data from the XML file in the form of the root of the xml file.

`src.Hec_ras06.`**`main`**()

This is not the main() of HABBY. This function is used to test this module independently of the rest of HABBY.

`src.Hec_ras06.`**`open_geofile`**(*geo_file*, *path*)

This function opens the geometry file (.g0X) from Hecr-rad. It extracts the (x,z) from each profile and the (x,y) if georeferenced,

Parameters

- **geo_file** – the name of the Hec-Ras geometry file (string)
- **path** – the path to the geo file (string)

Returns A list with each river profile (each profile is represented by a numpy array with the x and the altitude of each point in the profile), the coordinate of the profile (list of np.array), the coordinate of the river and the name of the reaches/ river in the file order (list of string)

Technical comments

The geofile is a text file with contains the geographical information. Because it is written to be read by human, it is complicated to load and regular expression are needed. It is written profile by profile.

Generally, to give a position, hec-ras indicates the profile number and the distance along this profile. In addition, data can be georeferenced or not. If it is geo-referenced we have some data in an (x,y) form. Otherwise, we only have geometrical data in the form (profile, dist).

First, for each profile, we get the elevation of the points forming each profile in the form (dist, elevation). The list of elevation for each profile starts with the keyword "Sta/Elev". The data found in the text file is in a string format. We use the function pass_in_float_from_geo to pass it in float. It is usually done using the function float. However, there are cases where there are no space between two number. However, in this case, the number of character per number is constant. In this case, we separate the number first.

Then, we get the coordinate of the river. If no coordinate are available the river is assumed to be straight. Next, we get the bank limit (even if we do not really used afterwards), and the name of the reach. It is also important to save the order in which the names of the reach are given. Indeed, we want this order to be the same in all functions, but they can be different between the geo file and the data output.

Next, we want to get the position (x,y) of each profile. If it is georeferenced, we will be able to get this position directly from the file and put it in the data_dist_str variable. We will then pass it to float. If not, we will use the function coord_profil_non_georeferenced to estimate the position of the profile (see below).

If the profile is not georeferenced, it is important to have the distance between two profile, so we extract the information from the geo file in all cases (georeferenced or not). The last profile of a reach does not have a distance to the next (not existing) profile. However, if a profile does not have a distance to the next profile and is not the last profile, we ignore this profile. It is usually not a problem because this profile is usually not a "real" profile, but the representation of a bridge or a culvert.

`src.Hec_ras06.`**`open_hecras`**(*geo_file*, *res_file*, *path_geo*, *path_res*, *path_im*, *save_fig=False*)

This function will open HEC-RAS outputs, i.e. the .geo file and the outputs (either .XML, .sdf or .rep) from HEC-RAS. All arguments from this function are string.

Parameters

- **`geo_file`** – the name of .goX (example .go3) file which is an output from hec-ras containg the profile data
- **`res_file`** – the name of O0X.xml file for the name of the .sdf file or the name of the .rep file (output data)
- **`path_res`** – path to the result file
- **`path_geo`** – path to the geo file
- **`path_im`** – the path to the folder where the images should be saved
- **`save_fig`** – if True image is saved

Returns coord_pro (for each profile, x,y,elev, dist along the profile), vh_pro (for each profile, dist along the profile, water height, velocity). Both variable are a list of numpy array.

How to obtain the input files

To obtain the xml file in HEC-RAS version 4:

•open the project in HEC-RAS.

•click on File , then export geometry and result (RAS Mapper), then OK

To obtain the sdf file in HEC-RAS version 5 which should be used if the model is georeferenced:

•click on File, then Export GIS data

•Export all reaches (select Reaches to export -. Full List -> Ok)

- Export all needed profile (select Profile to export -> Select all -> ok)

To obtain the report file .rep in HEC-RAS version which should be used if the model is NOT geo-referenced

- click on File, generate report
- Select Flow data and Geometry data in input data and, in Specific Table, select Flow distribution and Cross section Table

Technical comments

This is function which loads the hec_ras inputs in 1D for the version 4 and 5 of HEC-RAS. It accepts different type of hec-ras output as input and calls the appropriate sub-function for each input file. The geometrical data is always given in the geo file (with the extension g01, G01, g02, G02, g03, etc.). The output data can be in an xml file for the hec-ras in the version 4, an sdf file for hec-ras in version 5 or a .rep file in the version 5 if the model is not georeferenced. The xml file is the format which has been tested the most.

First, it loads the geometrical data. Then it select the function to load the output data and loads it. Then, it transforms the loaded data in a (x,y) coordinates system. Indeed, most of the data in hec-ras is given by indicating a profile (which crossed the modelled river) and the distance along this profile. For HABBY, it is better to get (x,y) coordinates. Then it create figure if asked by the switch "save_fig". Finally, it updates the forms of the output to be coherent with the dist_velocity_hecras function. This way, in HABBY, the output from mascaret and rubar after the velocity distribution have the same form than the output from hec-ras, which is useful afterwards to save all these data in the hdf5 file.

`src.Hec_ras06.`**`open_repfile`**`(`*report_file*, *reach_name*, *path*, *data_profile*, *data_bank*`)`

A function to open the report file (.rep) from HEC-RAS. To obtain the report file, see the doc of the function open_hecras.

Parameters

- **`report_file`** – a string with the name of the report file (.rep)
- **`reach_name`** – a list of string containing the name of the reaches/rivers in the order of the geo file, which might not be the order of the sdf file.
- **`path`** – the path where the report file is stored (string)
- **`data_profile`** – the data from each profile from the geofile (output from the open_geofile function)
- **`data_bank`** – the position of the bank limit (output from the open_geofile function)

Returns velocity and the water surface elevation for each river profiles in a list of np.array, the number of simulation (int) and the name of the river profile (list of string)

Technical comments and walk-through

This function is used to open output from models which were not geo-referenced in hec-ras v5. It cannot be used if the model was georeferenced (or at least one should make some tests before).

First, we obtain the water height. Then, we obtain the number of time step (which is called the number of simulation by hec-ras). To get the number of time step, we count each outputs given (one by profiles) and we divided it by the number of profile in the river. It is a bit indirect, but I did not find a simpler solution.

We get the name of each profile and reach. Then, we get the velocity data. We have in a case which is not geo-referenced. By consequence, there are only three velocities: one the left bank, one in the main river channel and one the right bank. Next we get the distance along the profile for these three velocities. Finally, we use the function reoder_reach for the same reason than in open_sdffile and open_xml.

`src.Hec_ras06.`**`open_sdffile`**`(`*sdf_file*, *reach_name*, *path*`)`

This is a function to load .sdf file from HEC-RAS v5 used if the model is georeferenced. To find how to obtain the sdf file, read the doc of open_hecras.

Parameters

- **sdf_file** – the name of the sdf file (string)
- **reach_name** – a list of string containing the name of the reaches/rivers in the order of the geo file which might not be the one of the sdf file. Output from open_geofile.
- **path** – the path where the file is stored (string)

Returns velocity, water height, river_name, number of time step (nb_sim)

Technical comments

To strat loading the sdf file, we open the sdf file. It is mostly a text file. Then we find velocity data and we pass this velocity data from string to float. The process is a bit similar to the one used in the function open_geofile with a healthy dose of regular expressions. We do this again for height data.

We also extract the name of the river, reaches and profile. The number of simulation (nb_sim) is a bit confusing for a variable name. In fact, it is the number of time step. Hec-Ras considers that one simulation is the simulation for one time step. Hence, nb_sim is more or less nb_timestep.

As in the xml file, we finally re-order the data as in the geo file. Indeed, it is possible to have different order between the reaches in the geo file and in the sdf file. Here, we use the function reorder_reach for this.

`src.Hec_ras06.`**`open_xmlfile`**(*xml_file*, *reach_name*, *path*)

This function open the xml file from HEC-RAS v4 to get the velocity and water surface elevation. To know how to obtain this xml file, read the doc of open_hecras.

Parameters

- **xml_file** – the name of O0X.xml file from HEC-RAS. (string)
- **reach_name** – a list of string containing the name of the reaches/rivers in the order of the geo file which might not be the one of the xml file.
- **path** – path to the xml file (string)

Returns velocity and the water surface elevation for each river profiles (list of np.array), the number of simulation(int) and the name of the river profile (list of string)

Technical comments

To load the xml file, we first call the load_xml function. It is a function which check that the xml file is well formed and which return the "root" part fo the xml. With this "root", it is possible to load other part of the xml file using the Etree module.

Then, we load the velocity and water height data from the xml file. We also load the name of the profiles and of the reach names. Next, we pass the data into float. For each velocity of height point, we get its position along the profile (see below for format) and the value at this point.

Finally, we re-order the data as in the geo file. Indeed, it is possible to have different order between the reaches in the geo file and in the xml file. The last part of this function is there to order all the data as in the geo file. There is a function reorder_reach which does something similar, but could not be used by the output from the xml file (it is slighty different). However the reorder_reach function and this part of the open_xml function is very similar.

`src.Hec_ras06.`**`pass_in_float_from_geo`**(*data_str*, *len_number*)

This is a function to pass the string data into float for open_geofile() and open_sdffile(). It is in a function because it is possible that two number are not separated by a space in the input data.

Parameters

- **data_str** – the data in a string form
- **len_number** – the number of digit for one number (int)

Returns a np.array of float with 2 columns (x,y) or (x,z)

`src.Hec_ras06.`**`reorder_reach`**(*wse*, *vel*, *riv_name*, *reach_name*, *reach_str*, *stream_str*, *nb_sim*)

The order of the reach in HABBY is in the order given in the geo file. However, it can be given in any order in the other file. (xml, sdf, rep,...). This function re-order the reaches based on their name.

Parameters

- **`wse`** – water height data (list of np.array for each profile)
- **`vel`** – velocity data (list of np.array for each profile)
- **`riv_name`** – the name of the profile (yeah I know it is not really logical as a name)
- **`reach_name`** – the name of the reach and stream (stream,reach) in the geo file order
- **`reach_str`** – the name of the reach in the anaylsed file order
- **`stream_str`** – the name of the stream in the anaylsed file order
- **`nb_sim`** – the number of simulation

Returns wse, vel, riv_name all re-ordered

Technical comments

The reach name should not have white space at the end/start but can have white space into them.

`src.Hec_ras06.`**`update_output`**(*zone_v*, *coord_pro_old*, *data_profile*, *xy_h*, *nb_pro_reach_old*)

This function updates the form of the output so it is coherent with mascaret and rubar after the lateral distribution of velocity for these two models. There are three important changes. First, coord_pro contains dist along the profile (x) and height in addition to the coordinates. Secondly, vh_pro contains only height if height is above or equal to zero. Thirdly, a point is created at the water limits and v and height are given at the same points. nb_pro_reach is also modified as in mascaret. We want to modify it so it start by zero and is additive, i.e., that it gives total number of profile before, not the number of profile by reach.

Parameters

- **`zone_v`** – (x,y, dist along profile, v) for each time step. However, the zone are the one from the models. They are different than the one from xy_h, which is unpractical for the rest of HABBY.
- **`coord_pro_old`** – the (x,y) coordinate for the profile
- **`data_profile`** – the distance along the porfile and height of each profile
- **`xy_h`** – the water height
- **`nb_pro_reach_old`** – the number of the profile by reach in the old form.

Returns coord_pro, vh_pro, nb_pro_reach

[doc to be finished]

4.1.1 Notes on hec-ras outputs

- Data in HEC-RAS can be geo-referenced or not georeferenced. It is advised to geo-reference all model in HEC-RAS. If the model is not geo-referenced, the function makes some assumptions to load the data: 1) the river profile are straight and perpendicular to the river. 2) the last profile is at the end of the river.
- To geo-reference a model in hec_ras: In the “geometric data” window, GIS tool, GIS Cut Line, Accept Display location, choose all profile
- Numerical data are sometime not separated (0.4556 0.3453233.454 05.343). In this case, the number of digit is assumed to be 8 for the profile and 16 for the river coordinates.

- Part of the profile can be vertical: The function also functions in this case.
- There is sometimes more than one reach in the modelled river and these reaches sometimes form loops: The function load each reach one after the other.
- The river reaches are sometimes not in the same order in the xml file and in the .goX file. The order of the .goX is used by the function. Reach are automatically re-ordered.
- If the river is straight, the coordinates of the river are given differently. The function try to load the river in the "straight" style if the usual style fail.
- The .goX file includes data on bridges and culvert. Currently, the function neglects this information.
- Sometimes distances between profiles are not given in the .goX file. The function neglects the distance data of this profile as long as it is not the last profile.
- The velocity data for the end and the beginning of the river profile is indicated by a large number (example 1.23e35 or -1.234e36). The function considers that velocity info is situated at the start of the profile if x>-1e30 and at the end of the profile if x> 1e30.
- There are two concepts called "profile" in HEC-RAS: The river profiles and the simulation profiles. The river profiles are the geometry perpendicular to the river and the simulation profile are the different simulations.
- Data in many of the example cases of HEC-RAS are in foot and miles. 1 miles = 5280 foot, and not 1000 foot.

4.2 Hec-ras model 2D

in src/Hec_ras2D.py

This module contains the functions used to load the outputs from the hec-ras model in 2D.

`src.hec_ras2D.`**`figure_hec_ras2d`**(*v_all, h_all, elev_all, coord_p_all, coord_c_all, ikle_all, path_im, time_step=[0], flow_area=[0], max_point=-99*)

This is a function to plot figure of the output from hec-ras 2D.

Parameters

- **v_all** – a list of np array representing the velocity at the center of the cells
- **h_all** – a list of np array representing the water depth at the center of the cells
- **elev_all** – a list of np array representing the mimium elevation of each cells
- **coord_p_all** – a list of np array representing the coordinates of the points of the grid
- **coord_c_all** – a list of np array representing the coordinates of the centers of the grid
- **ikle_all** – a list of np array representing the connectivity table one array by flow area
- **time_step** – which time_step should be plotted (default, the first one)
- **flow_area** – which flow_area should be plotted (default, the first one)
- **max_point** – the number of cell to be drawn when reconstructing the grid (it might long)
- **path_im** – the path where the figure should be saved

Technical comment

This function creates three figures which represent: a) the grid of the loaded models b) the water height and c) the velocity.

The two last figures will be modified when the data will be loaded by node and not by cells. So we will not explai n them here as they should be re-written.

The first figure is used to plot the gird. If we would plot the grid by drawing one side of each triangle separately, it would be very long to draw. To optimize the process, we use the prepare_grid function.

`src.hec_ras2D.`**`load_hec_ras2d`**(*filename*, *path*)

The goal of this function is to load 2D data from Hec-RAS in the version 5.

Parameters

- **`filename`** – the name of the file containg the results of HEC-RAS in 2D. (string)
- **`path`** – the path where the file is (string)

Returns velocity and height at the center of the cells, the coordinate of the point of the cells, the coordinates of the center of the cells and the connectivity table. Each output is a list of numpy array (one array by 2D flow area)

How to obtain the input file

The file neede as input is an hdf5 file (.hdf) created automatically by Hec-Ras. There are many .hdf created by Hec-Ras. The one to choose is the one with the extension p0X.hdf (not g0x.hdf). It is usually the largest file in the results folder.

Technical comments

Outputs from HEC-RAS in 2D are in the hdf5 format. However, it is not possible to directly use the output of HEC-RAS as an hdf5 input for HABBY. Indeed, even if they are both in hdf5, the formats of the hdf5 files are different (and would miss some important info for HABBY). So we still need to load the HEC-RAS data in HABBY even if in 2D.

This function should be modified because currently it gets the data by cells. However, we should get the data by node. So this function should be changed.

Walk-through

The name and path of the file is given as input to the load_hec_ras_2D function. Usually this is done by the class HEC_RAS() in the GUI. We load the file using the h5py module. This module opens and reads hdf5 file.

Then we can read different part of the hdf5 file when we know the address of it (this is a bit like a file system). In hdf5 file of Hec-RAS, this first thing is to get the names of the flow area in "Geometry/2D Flow Area". In general, this is the name of each reach, but it could be lake or pond also.

Then, we go to "Geometry/2D Flow Area/<name>/FacePoint Coordinates" to get the points forming the grid. We can also get the connectivity table (or ikle) to the path "Geometry/2D Flow Area/<name>/Cells Face Point Indexes" We also get the elevations of the cells. Currently, this is just the minimum elevation of the cells, but it should be modified to get the elevation by node (in the vocabulary of HEC-RAS by "FacePoints"). We then get the water depth by cell. Somethings should be done to get it by node. I think that we did have the elevation by node somewhere in the hdf5 file. For there, water height can be found. The velocity is given by face of the cells. It should be averaged differently to get it on the point and not on the side.

`src.hec_ras2D.`**`main`**()

Used to test this module independantly of HABBY.

`src.hec_ras2D.`**`prepare_grid`**(*ikle*, *coord_p*, *max_point=-99*)

This is a function to put in the new form the data forming the grid to accelerate the plotting of the grid. This function creates a list of points of the grid which are re-ordered compared to the usual list of grid point (the variable coord_p here). These points are reordered so that it is possible to draw only one line to form the grid (one point can appears more than once). The grid is drawn as one long line and not as a succession of small lines, which is quicker. When this new list is created by prepare_function(), it is send back to figure-hec_ras_2D and plotted.

Parameters

- **`ikle`** – the connectivity table

- **coord_p** – the coordinates of the point
- **max_point** – if the grid is very big, it is possible to only plot the first points, up to max_points (int)

Returns a list of x and y coordinates ordered.

`src.hec_ras2D.`**`scatter_plot`**(*coord*, *data*, *data_name*, *my_cmap*, *s1*, *t*)

The function to plot the scatter of the data. Will not be used in the final version, but can be useful to plot data by cells.

Parameters

- **coord** – the coordinates of the point
- **data** – the data to be plotted (np.array)
- **data_name** – the name of the data (string)
- **my_cmap** – the color map (string with matplotlib colormap name)
- **s1** – the size of the dot for the scatter
- **t** – the time step being plotted

4.3 Mascaret

in src/mascaret.py

This module contains the functions used to load the outputs from the mascaret model.

`src.mascaret.`**`correct_duplicate`**(*seq*, *send_warn*, *idfun=None*)

It is possible to have a vertical line on a profile (different h, identical x). This is not possible for HABBY and the 2D grid. So this function correct duplicates along the profile.

A similiar function exists in rubar, for the case where input is (x,y) coordinates and not distance along the profile. This function is inspired by https://www.peterbe.com/plog/uniqifiers-benchmark

It should be tested more as manage_grid sometime still send warning about duplicate data in profile.

Parameters

- **seq** – the list to be corrected (list)
- **send_warn** – a bool to avoid printing certains warning too many time
- **idfun** – support an optional transform function (not used)

Returns the profile data without duplicate and the bollean which manages the warning.

`src.mascaret.`**`define_stream_network`**(*node_number*, *start_node*, *end_node*, *angles*, *nb_pro_reach*, *nb_reach*, *abcisse*)

This function extracts the stream network from the node and angle data. This is used if we have more than one reach to define the geometry of the junction.

Parameters

- **node_number** – the start/end number of the reaches for each nodes (list of list)
- **start_node** – the numbers indicating the start of each reach (list)
- **end_node** – the numbers indicating the end of each reach
- **angles** – for each node the angle between the reach

- **nb_pro_reach** – the number of profile by reach
- **nb_reach** – the number of reach
- **abcisse** – the distance along the river of each reach

Returns the river coordinates and the unit vector indicating the river direction

src.mascaret.**figure_mascaret**(*coord_pro, coord_r, xhzv_data, on_profile, nb_pro_reach, name_pro, name_reach, path_im, pro, plot_timestep=[-1], reach_plot=[0]*)

The function to plot the figures related to mascaret.

Parameters

- **coord_pro** – the cordinates (x,y,h, dist along the river) of the profiles
- **coord_r** – the coordinate (x,y) of the river
- **name_pro** – the name of the profile
- **name_reach** – the name of the reach
- **on_profile** – which result are on the profile. Some output are not the profiles.
- **nb_pro_reach** – the number of profile by reach (careful this is the number of profile, not the number of output)
- **xhzv_data** – the height and velcoity (x,h,v) list by time step
- **profile** (*pro*) – which profile to be plotted (list of int)
- **plot_timestep** – which timestep to be plotted
- **reach_plot** – the reach to be plotted for the river view
- **path_im** – the path where to save the figure

src.mascaret.**find_node**(*node_number*, *reach_to_find*)

This function finds which node is a stream end or a stream start. It is associated by the function define_stream_network()

Parameters

- **node_number** – the list of list of the reaches linked with one node
- **reach_to_find** – the number indicating the start or end of the reach

Returns the node number, ordered as in the xcas file

src.mascaret.**flat_coord_pro**(*coord_pro*)

This function is not used anymroe.

The variable coord_pro was a list of profile by reach. Finally, it was useful to have each profile one after the other with accounting for the reach. So we stop to use this function whose goal was to pass from one form of coord_pro to the other form 9with or wihtout reach information).

Parameters **coord_pro** – the list of profile (x,y,h, dist along the river) by reach

Returns coord_pro_f: a list of profile without the reach information. The list is flatten

src.mascaret.**get_geo_name_from_xcas**(*file_gen*, *path_gen*)

This function gets the name of the .geo file from the .xcas xml file. It is not used yet, but it could be useful in the GUI to simplify the loading of mascaret. The user would not need to give the name of the geo and the xcas files separetly. However, it is not written in yet.

Parameters

- **file_gen** – the xcas file
- **path_gen** – the path to the xcas file

Returns the name of the .geo file (no path indicated)

src.mascaret.**get_name_from_cas**(*file_gen*, *path_gen*)
This function gets the name of the .geo file from the .cas text file. It is not used yet, but it could be useful in the GUI to simplify the loading of mascaret. The user would not need to give the name of the geo and the cas files separetly. However, it is not written in yet.

Parameters

- **file_gen** – the name of .cas file (string)
- **path_gen** – the path to the cas file (string

Returns the name of the .geo file (no path indicated)

src.mascaret.**is_this_res_on_the_profile**(*abscisse*, *xhzv_data_all*)
The output of mascaret can be given at points of the river where there is no profile. The function here says which results are on the profiles. All profiles are linked with an output, but some output are not linked with a profile.

Parameters

- **abscisse** – the distance between each profile (list of float)
- **xhzv_data_all** – the outputs from mascaret by time step

Returns a list of bool of the length of xhzv_data, True on profile, False not on profile

Technical comment

In the mascaret outputs, some rounding are suprising. For example, 0.49 can be transformed to 0.50 in an otehr file (not 0.5). To avoid this type of problem, we says that outputs with a distance smaller than 3cm of the profile are on the profile. If there are more than one output by profile, we takes the output which is the closest to the profile.

src.mascaret.**load_mascaret**(*file_gen*, *file_geo*, *file_res*, *path_gen*, *path_geo*, *path_res*)
The function is used to load the mascaret data. It load the geofile and the general file. Then, it re-forms the geometrical data. Next, it loads the output data from mascaret. Fianally, it looks which outputs is close to a profile and which outputs is not linked with a profile as there are some outputs given between profiles.

Parameters

- **file_gen** – the xcas .xml file giving general info about the model (string)
- **file_geo** – the file containting the profile data (.geo) (string)
- **file_res** – the files containting the mascaret output in the Optyca format (.opt) (string)
- **path_gen** – the path to the xcas file or .cas file (string). By default, choose the xcas file.
- **path_geo** – the path to the geo file (string)
- **path_res** – the path to the res file (string)

Returns the coordinates of the profile (x,y,z, dist along the profile), the coordinate of the river (x,y), name of reach and profile, data height and velocity (list by time step), list of bollean indicating which data is on the profile and the number of profile by reach.

src.mascaret.**main**()
Used to test this module separately.

src.mascaret.**open_geo_mascaret**(*file_geo*, *path_geo*)
This function load the mascaret geo file. Generally, the profile are not geo-referenced when using this function.

Parameters

- **`file_geo`** – the name of the geo file (string)
- **`path_geo`** – the path to the geo file (string)

Returns the profile data (x,y), profile name (list of string), brief name (list of string), the number of profile in each reach and distance along the river/abcisse (list)

`src.mascaret.`**`open_res_file`**(*file_res*, *path_res*)
The function to load the output from mascaret (.opt file). The format is Optyca.

Parameters

- **`file_res`** – the name of the .opt file (string)
- **`path_res`** – the path to this file (string)

Returns

`src.mascaret.`**`open_rub_file`**(*file_res*, *path_res*)
The function to open the binary output file from mascaret (.rub format).

Parameters

- **`file_res`** – the name of the rub binary file (string)
- **`path_res`** – the path to this file (string)

Returns xhzv_data, timestep

Technical comment

The binary output file was done using a program written in FORTRAN. So there are often suprising octet which are added to the binary file. Be careful before changing anything.

`src.mascaret.`**`profil_coord_non_georef`**(*coord_pro*, *coord_r*, *nr*, *nb_pro_reach*, *bt=None*)
This function gets the coordinates (x,y) of the profile as masacret outputs are not georeferenced.

Hypothesis: The river and the profile are straight. The profile is perpendicular to the river. The river pass at the minimum elevation of the river bed. If there is a distinction between the main bed the secondary bed is given, we take the minimum elevation of the main bed

The origin of the coordinate system is the start of the river.

Parameters

- **`coord_pro`** – the coordinate of the profile. This variable is not in the general coordinate system, just distance along the profile and bed elevation (p, dist, h)
- **`coord_r`** – the river coordinates
- **`n`** – the vector indicating the river direction
- **`nb_pro_reach`** – the number of profile by reach (additive)
- **`bt`** – optional, it indicates which points in the profiles are in the minor/major bed

Returns the velocity and height data, the timestep

`src.mascaret.`**`river_coord_non_georef_from_cas`**(*file_gen*, *path_gen*, *abcisse*, *nb_pro_reach*)
Get the coordinates of the river based on the cas text file. If there are only one river, this is an easy task as the river is straight. If there are more than one reach, the junctions and the angles between the reach sould be managed using the define_stream_network function and the information in the .cas file.

Parameters

- **`file_gen`** – the .cas file whcih contains general info (string)

- **path_gen** – the path to this file (string)
- **abcisse** – ditance along the profiles
- **nb_pro_reach** – the number of reach by profile

Returns the river coordinate and the unit vector indicating the river direction

src.mascaret.**river_coord_non_georef_from_xcas**(*file_gen*, *path_gen*, *abcisse*, *nb_pro_reach*)

Get the coordinates of the river based on the xcas xml file. If there are only one river, this is an easy task as the river is straight. If there are more than one reach, the junctions and the angles between the reach sould be managed using the define_stream_network function and the information in the .xcas file.

Parameters

- **file_gen** – the .xcas file with the information concerning the reach (string)
- **path_gen** – the path to the xcas file (string)
- **abcisse** – the distance along the river
- **nb_pro_reach** – the number of profile by reach

Returns coord_r the coordinate of the river

4.4 River 2D

in src/river2D.py

This module contains the functions used to load the outputs from the River2D model.

src.river2d.**figure_river2d**(*xyzhv*, *ikle*, *path_im*, *t=0*)

A function to plot the output from river 2d. Need hec-ras2d as import because it re-used most of the plot from this script.

Plot only one time step because river 2d output have one file by time step.

Parameters

- **xyzhv** – the x,y, coordinates of the node (h,v are nodal output in river 2d), the river bed, the water height and the velocity (one data by column, row are node)
- **ikle** – connectivity table
- **path_im** – the path where to save the figure
- **t** – the time step which is being plotted

Returns

src.river2d.**get_rid_of_lines**(*datahere*, *nb_data*)

There are lines which are useless in the cdg file. This function is used to correct ikle and data_node

Parameters

- **datahere** – the data with the empty lines
- **nb_data** – nb_node or nb_el

Returns datahere wihtout the useless lines

src.river2d.**load_river2d_cdg**(*file_cdg*, *path*)

The file to load the output data from River2D. Careful the input data of River2D has the same ending and nearly the same format as the output. However, it is nessary to have the output here. River2D gives one cdg. file by timestep. Hence, this function read only one timeste. HABBY read all time step by calling this function once for each time step.

Parameters

- **`file_cdg`** – the name of the cdg file (string)
- **`path`** – the path to this file (string).

Returns the velcoity and height data, the coordinate and the connectivity table.

`src.river2d.`**`main`**`()`

Used to test this module.

4.5 Rubar

in src/rubar.py

This module contains the functions used to load the Rubar data in 2D and 1D.

`src.rubar.`**`correct_duplicate_xy`**`(`*seq3D*, *send_warn*, *idfun=None*`)`

It is possible to have a vertical line on a profile (different h, identical x). This is not possible for HABBY and the 2D grid. So this function correct duplicates along the profile.

A similiar function exists in mascaret, for the case where the input is the distance along the profile and not (x,y) coordinates. This function is inspired by https://www.peterbe.com/plog/uniqifiers-benchmark.

It should be tested more as manage_grid sometime still send warning about duplicate data in profile.

Parameters

- **`seq3D`** – the list to be corrected in this case (x,y,z,dist along the profile)
- **`send_warn`** – a bool to avoid printing the warning too many time
- **`idfun`** – support an optional transform function (not tested)

Returns the list wihtout duplicate and the boolean which helps manage the warnings

`src.rubar.`**`figure_rubar1d`**`(`*coord_pro, lim_riv, data_xhzv, name_profile, path_im, pro, plot_timestep, nb_pro_reach=[0, 10000000000]*`)`

The function to plot the loaded RUBAR 1D data (Rubar BE).

Parameters

- **`coord_pro`** – the coordinate of the profile (x, y, z, dist along the river)
- **`lim_riv`** – the right bank, river center, left bank
- **`data_xhzv`** – the data by time step with x the distance along the river, h the water height and v the vlocity
- **`cote`** – the altitude of the river center
- **`name_profile`** – the name of the profile
- **`path_im`** – the path where to save the image
- **`pro`** – the profile number which should be plotted
- **`plot_timestep`** – which timestep should be plotted
- **`nb_pro_reach`** – the number of profile by reach

Returns none

src.rubar.**figure_rubar2d**(*xy, coord_c, ikle, v, h, path_im, time_step=[-1]*)
This functions plots the rubar 2d data

Parameters

- **xy** – coordinates of the points
- **coord_c** – the center of the point
- **ikle** – connectivity table
- **v** – speed
- **h** – height
- **path_im** – the path where to save the figure
- **time_step** – The time step which will be plotted

src.rubar.**get_triangular_grid**(*ikle*, *coord_c*, *xy*, *h*, *v*)

In Rubar it is possible to have non-triangular cells. It is possible to have a grid composed of a mix of pentagonal, 4-sided and triangualr cells. This function transform the "mixed" grid to a triangular grid. For this, it uses the centroid of each cell with more than three side and it create a triangle by side (linked with the center of the cell)

Parameters

- **ikle** – the connectivity table
- **coord_c** – the coordinate of the centroid of the cell
- **xy** – the points of the grid
- **h** – data on water height
- **v** – data on velocity

Returns the updated ikle, coord_c (the center of the cell , must be updated) and xy (the grid coordinate)

src.rubar.**load_coord_1d**(*name_rbe*, *path*)
the function to load the rbe file, which is an xml file. The gives the geometry of the river system.

Parameters

- **name_rbe** – The name fo the rbe file (string)
- **path** – the path to this file (string)

Returns the coordinates of the profiles and the coordinates of the right bank, center of the river, left bank (list of np.array with x,y,z coordinate), name of the profile (list of string), dist along the river (list of float) number of cells (int)

src.rubar.**load_dat_2d**(*geofile*, *path*)
This function is used to load the geomtery info for the 2D case, using the .dat file The .dat file has the same role than the .mai file but with more information (number of side and more complicated connectivity table).

Parameters

- **geofile** – the .dat file which contain the connectivity table and the (x,y)
- **path** – the path to this file

Returns connectivity table, point coordinates, coordinates of the cell centers

`src.rubar.`**`load_data_1d`**`(`*name_data_vh*, *path*, *x*`)`
This function loads the output data for Rubar BE (in 1D). The geometry data should be loaded before using this function.

Parameters

- **`name_data_vh`** – the name of the profile.ETUDE file (string)
- **`path`** – the path to this file
- **`x`** – the distance along the river (from the .geo file)

Returns data x, velocity height, cote for each time step (list of np.array), time step

`src.rubar.`**`load_mai_1d`**`(`*mailfile*, *path*`)`
This function is not used anymore. It was used to load the coordinate of the 1D data. It might become useful again in the case where we found a Rubar model with more than one reach (which we do not have yet).

Parameters

- **`mailfile`** – the name of the file which contain the (x,z) data
- **`path`** – the path to this file

Returns x of the river, np.array and the number of mail

`src.rubar.`**`load_mai_2d`**`(`*geofile*, *path*`)`
The function to load the geomtery info for the 2D case when we use the .mai file. It would also be possible to use the .dat file. In fact, it is advised to use the dat file when possible as there are more info in the .dat file.

Parameters

- **`geofile`** – the .mai file which contain the connectivity table and the (x,y)
- **`path`** – the path to this file

Returns connectivity table, point coordinates, coordinates of the cell centers

`src.rubar.`**`load_rubar1d`**`(`*geofile*, *data_vh*, *pathgeo*, *pathdata*, *path_im*, *savefig*`)`
the function to load the RUBAR BE data (in 1D).

Parameters

- **`geofile`** – the name of .rbe file which gives the coordinates of each profile (string)
- **`data_vh`** – the name of the profile.ETUDE file which contains the height and velocity data (string)
- **`pathgeo`** – the path to the geofile - string
- **`pathdata`** – the path to the data_vh file
- **`path_im`** – the file where to save the image
- **`savefig`** – a boolean. If True create and save the figure.

Returns coordinates of the profile (x,y,z dist along the profile) coordinates (x,y) of the river and the bed, data xhzv by time step where x is the distance along the river, h the water height, z the elevation of the bed and v the velocity

`src.rubar.`**`load_rubar2d`**`(`*geofile*, *tpsfile*, *pathgeo*, *pathtps*, *path_im*, *save_fig*`)`
This is the function used to load the RUBAR data in 2D

Parameters

- **`geofile`** – the name of the .mai or .dat file which contains the connectivity table and the coordinates (string)

- **tpsfile** – the name of the .tps file (string)
- **pathgeo** – path to the geo file (string)
- **pathtps** – path to the tps file which contains the outputs (string)
- **path_im** – the path where to save the figure (string)
- **save_fig** – a boolean indicating if the figures should be created or not

Returns velocity and height at the center of the cells, the coordinate of the point of the cells, the coordinates of the center of the cells and the connectivity table.

`src.rubar.`**`load_tps_2d`**(*tpsfile*, *path*, *nb_cell*)

The function to load the output data in the 2D rubar case. The geometry file (.mai or .dat) should be loaded before.

Parameters

- **tpsfile** – the name of the file with the data for the 2d case
- **path** – the path to the tps file.
- **nb_cell** – the number of cell extracted from the .mai file

Returns v, h, timestep (all in list of np.array)

`src.rubar.`**`m_file_load_coord_1d`**(*geofile_name*, *pathgeo*)

This function loads the m.ETUDE file which is based on .st format from cemagref. When we use the M.ETUDE file instead of the rbe file, more than one reach can be studied but the center and side of the river is not indicated anymore.

Parameters

- **geofile_name** – The name to the m.ETUDE file (string)
- **pathgeo** – the path to this file (string)

Returns the coordinates of the profiles (list of np.array with x,y,z coordinate), name of the profile (list of string), dist along the river (list of float), number of profile by reach

`src.rubar.`**`main`**()

Used to test this module

4.6 Telemac

in src/selafin_habby1.py

This module contains the functions used to load the Telemac data.

class `src.selafin_habby1.`**`Selafin`**(*filename*)

Selafin file format reader for Telemac 2D. Create an object for reading data from a slf file. Adapted from the original script 'parserSELAFIN.py' from the open Telemac distribution.

Parameters **filename** – the name of the binary Selafin file

`addcontent`(*fileName*, *times*, *values*)

`appendcoretimeslf`(*t*)

`appendcorevarsslf`(*varsor*)

`appendheaderslf`()

Write the header file

getheaderfloatsslf()
Get the mesh coordinates

getheaderintegersslf()
Get dimensions and descritions (mesh)

getheadermetadataslf()
Get header information

gettimehistoryslf()
Get the timesteps

getvalues(*t*)
Get the values for the variables at time t

getvariablesat(*frame*, *varindexes*)
Get the values for the variables at a particular time step

putcontent(*fileName*, *times*, *values*)

src.selafin_habby1.**getendianfromchar**(*fileslf*, *nchar*)

Get the endian encoding "<" means little-endian ">" means big-endian

src.selafin_habby1.**getfloattypefromfloat**(*fileslf*, *endian*, *nfloat*)
Get float precision

src.selafin_habby1.**load_telemac**(*namefilet*, *pathfilet*)
A function which load the telemac data using the Selafin class

Parameters

- **namefilet** – the name of the selafin file (string)
- **pathfilet** – the path to this file (string)

Returns the velocity, the height, the coordinate of the points of the grid, the connectivity table.

src.selafin_habby1.**plot_vel_h**(*coord_p2, h, v, path_im, timestep=[-1]*)
a function to plot the velocity and height which are the output from TELEMAC

Parameters

- **coord_p2** – the coordinates of the point forming the grid
- **h** – the water height
- **v** – the velocity
- **path_im** – the path where the image should be saved (string)
- **timestep** – which time step should be plotted

4.7 Load HABBY hdf5 file

in src/load_hdf5.py

This module contains some functions to load and manage hdf5 input/outputs. This is still in progress.

src.load_hdf5.**get_all_filename**(*dirname*, *ext*)
This function gets the name of all file with a particular extension in a folder. Useful to get all the output from one hydraulic model.

Parameters

- **dirname** – the path to the directory (string)
- **ext** – the extension (.txt for example). It is a string, the point needs to be the first character.

Returns a list with the filename (filename+dir) for each extension

`src.load_hdf5.`**`load_hdf5_hyd`**`(`*hdf5_name_hyd*`)`
A function to load the 2D hydrological data contains in the hdf5 file in the form required by HABBY.

Parameters **hdf5_name_hyd** – path and filename of the hdf5 file (string)

Returns the connectivity table, the coordinates of the point, the height data, the velocity data on the coordinates.

`src.load_hdf5.`**`load_hdf5_sub`**`(`*hdf5_name_sub*`)`
A function to load the substrate data contained in the hdf5 file.

Parameters **hdf5_name_sub** – path and file name to the hdf5 file (string)

`src.load_hdf5.`**`open_hdf5`**`(`*hdf5_name*`)`
This is a function which open an hdf5 file and check that it exists. it does not load the data. It only opens the files. :param hdf5_name: the path and name of the hdf5 file (string)

4.7.1 Form of the hdf5 files

Here is the actual form of the hdf5 containing the 2D hydrological data.

- Number of timestep: Data_gen/Nb_timestep
- Number of reach: Data_gen/Nb_reach
- Connectivity table for the whole profile: Data_2D/Whole_Profile/Reach_<r>/ikle
- Connectivity table for the wetted area (by time step): Data_2D/Timestep<t>/Reach_<r>/ikle
- Coordinates for the whole profile: Data_2D/Whole_Profile/Reach_<r>/point_all
- Coordinates for the wetted area (by time steps): Data_2D/Timestep<t>/Reach_<r>/point_all
- Data for the velocity: Data_2D/Timestep<t>/Reach_<r>/inter_vel_all
- Data for the height: Data_2D/Timestep<t>/Reach_<r>/inter_h_all

Here is the actual form of the hdf5 containing the substrate data.

- the coordinate of the point forming the substrate "grid": coord_p_sub/
- the connectivity table of the substrate "grid": ikle_sub/
- Substrate data; not done yet

4.8 Velocity distribution

in src/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.

`src.dist_vistess2.`**`dist_velocity_hecras`**(*coord_pro*, *xhzv_data_all*, *manning_pro*, *nb_point=-99*, *eng=1.0*, *on_profile=[]*)

This function distribute the velocity along the profile using the method from hec-ras which is described in the hydraulic reference manual p 4-20 (Flow distribtion calculation)

Parameters

- **`coord_pro`** – the coordinates and elevation of the river bed for each profile (x,y, h, dist along the profile) this list is flatten No reach info.
- **`xhzv_data_all`** – water height and velocity at each profile, 1D
- **`manning_pro`** – the manning coefficient for zone between point of each profile. For a particular profile, the length of manning_pro is the length of coord_pro[0]
- **`nb_point`** – number of velocity points (-99 takes the number of measured elevation as the number of velocity points).
- **`eng`** – in case the output from hec-ras are in US unit (eng=1 for SI unit and 1.486 for US unit)
- **`on_profile`** – Mascaret also gives outputs in poitns between profile. on_profile is true if the results are close or on the profile (les than 3cm of difference). This is not important for rubar or other models

Returns the velocity for each profile by time step (x,v)

Technical comment and walk-through

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 (as it is the case here).

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 (the distlance 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 calculate 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. See manual of hec-ras p.4-20 for the conveyance definition.

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).

`src.dist_vistess2.`**`get_manning`**(*manning1*, *nb_point*, *nb_profil*, *coord_pro*)

This function creates an array with the manning value when a single float is given, so when the manning value is a constant for the whole river.

Parameters

- **`manning1`** – the manning value (can be a value or an array)
- **`nb_point`** – the number of velocity point by profile
- **`nb_profil`** – the number of profile
- **`coord_pro`** – necessary if the number is -99 as we need to know the length of each profile

Technical comments

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).

`src.dist_vistess2.`**`get_manning_arr`**(*manning_arr*, *nb_point*, *coord_pro*)

This function create the manning array when manning data is loaded using a text file. In this case, the manning value do not needs to be a constant.

Parameters

- **`manning_arr`** – the data for manning
- **`nb_point`** – the number of velocity point by profile
- **`coord_pro`** – x,y,dist

Technical comment

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. One value by line in SI unit.

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.

`src.dist_vistess2.`**`main`**()

Used to test this module.

`src.dist_vistess2.`**`plot_dist_vit`**(*v_pro*, *coord_pro*, *xhzv_data*, *plot_timestep*, *pro*, *name_pro=[]*, *on_profile=[]*, *zone_v_all=[]*, *data_profile=[]*, *xy_h_all=[]*)

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 where zone_v_all is an 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, for this, it is necessary to have prepared the 1D output from hec-ras (using the function preparetest_velocity) and to have the same points on which to calculate the velocity.

Parameters

- **v_pro** – the calculated velcocity distribution by time step
- **coord_pro** – the coordinate of the profiles
- **xhzv_data** – the output data from the model, before the velocity distrbution
- **plot_timestep** – which time step to be plottied
- **name_pro** – the name of the profile (optionnal just for the title)
- **pro** – which porfile to be plotted
- **on_profile** – select the data which is on the profile
- **zone_v** – output from hec-ras used to test dist_vitesse
- **data_profile** – output from hec-ras used to test dist_vitesse
- **xy_h** – output from hec-ras used to test dist_vitesse

`src.dist_vistess2.`**`preparetest_velocity`**(*coord_pro*, *vh_pro_orr*, *v_in*)

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 on 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. The results should however be close enough.

Parameters

- **coord_pro** – the coordinate of the profile (x,y,h,dist along profile)
- **vh_pro_orr** – the velocity distribution which is the output from hec ras (produced by hec-ras06.py)
- **v_in** – the uni-dimensional velocity

4.9 Create a grid

in src/manage_grid_8

This module is composed of the functions used to manage the grid, notably to create 2D grid from the output from 1D model.

There are two main way to go from data in 1.5D in a profile form to a 2D grid:

- through the usage of the triangle module in create_grid().
- through the definition of a middle profile used as a guide to create the grid in create_grid_only_one_profile().

For an in-depth explanation on how to create the grids, please see the pdf document `More info on the grid`

`src.manage_grid_8.`**`add_point`**(*point_all*, *point*)

To manage the substrate data, we modify the hydrological grid to avoid to have cells with two substrate type. This function add one coordinate point to the list of coordinates which compose the hydrological grid. This point is the intersection between one side of one triangluar cell of the hydrological grid and one side of the sibstrate layer (which is a shp). It only adds this intersection point if it is not already in point_all.

Parameters

- **point_all** – the coordinates of the hydrological grid
- **point** – one intersection point between substrat and hydrological grids

Returns the updated point_all (the coordinates of the hydrological grid)

`src.manage_grid_8.`**`create_dummy_substrate`**(*coord_pro*, *sqrtnp*)

For testing purposes, it can be useful to create a substrate input even if one does not exist. This substrate is compose of n triangle situated on the rivers in the same coodinates system.

Parameters

- **coord_pro** – the coordinate of each profile
- **sqrtnp** – the number of point which will compose one side of the new substrate grid (so the total number of point is sqrtnb squared).

Returns dummy coord_sub, ikle_sub

`src.manage_grid_8.`**`create_grid`**(*coord_pro, extra_pro, coord_sub, ikle_sub, nb_pro_reach=[0, 10000000000.0], vh_pro_t=[], q=[], pnew_add=1*)

It creates a grid from the coord_pro data using the triangle module. It creates the grid up to the end of the profile if vh_pro_t is not present or up to the water limit if vh_pro_t is present

Parameters

- **q** – used in the secondary process (like in hydro_gui2) when we do not call this function direclty, but we call it in a second process so that the GUI do not crash if something go wrong
- **coord_pro** – the profile coordinates (x,y, h, dist along) the profile
- **extra_pro** – the number of "extra" profiles to be added between profile to simplify the grid
- **coord_sub** – (not used anymore) the coordinate of the point forming the substrate layer (often created with substrate.load_sub)
- **ikle_sub** – (not used anymore) the connectivity table of the substrate grid (often created with substrate.load_sub)
- **nb_pro_reach** – the number of reach by profile starting with 0
- **vh_pro_t** – the velocity and height of the water (used to cut the limit of the river).
- **pnew_add** – (not used anymore) a parameter to cut the substrate side in smaller part (improve grid quality) in the form dist along profile, h , v for the analyzed time step. f not given, gird is contructed on the whole profile.

Returns connectivity table and grid point

Form of the function in summary

- if vh_pro_t:
  - find cordinate under water and used this to update coord_pro
  - see if there is islands, find the island limits and the holes indicating the inside/outside of the islands
- find the point which give the end/start of the segment defining the grid limit
- find all point which need to be added to the grid and add extra profile if needed
- based on the start/end points and the island limits, create the segments which gives the grid limit
- triangulate and so create the grid

•flag point which are overlapping in two grids

For more info, see the document "More info on the grid".

`src.manage_grid_8.`**`create_grid_only_1_profile`**(*coord_pro, nb_pro_reach=[0, 10000000000.0], vh_pro_t=[]*)

This function creates the grid from the coord_pro data using one additional profil in the middle. No triangulation. The interpolation of the data is done in this function also, contrarily to create_grid().

Parameters

- **`coord_pro`** – the profile coordinates (x,y, h, dist along) the profile
- **`nb_pro_reach`** – the number of profile by reach
- **`vh_pro_t`** – the data with heigh and velocity, giving the river limits

Returns the connevtivity table, the coordinate of the grid, the centroid of the grid, the velocity data on this grid, the height data on this grid.

For more info on this function, see the document "More info on the grid".

`src.manage_grid_8.`**`cut_2d_grid`**(*ikle, point_all, water_height, velocity*)

This function cut the grid of the 2D model to have correct wet surface. If we have a node with h<0 and other node(s) with h>0, this function cut the cells to find the wetted perimeter, assuminga linear decrease in the water elevation. This function works for one time steps and for one reach

Parameters

- **`ikle`** – the connectivity table of the 2D grid
- **`point_all`** – the coordinate of the point
- **`water_height`** – the water height data given on the nodes
- **`velocity`** – the velcoity given on the nodes

Returns the update connectivity table, the coodinate of the point, the height of the water and the velocity on the updated grid

`src.manage_grid_8.`**`find_profile_between`**(*coord_pro_p0, coord_pro_p1, nb_pro, trim=True*)

Find n profile between two profiles which are not straight. This functions is useful to create the grid from 1D model as profile in 1D model are often far away from another.

Parameters

- **`coord_pro_p0`** – the coord_pro (x,y,h, z) of the first profile
- **`coord_pro_p1`** – the coord_pro (x,y,h, z) of the second profile
- **`nb_pro`** – the number of profile to add
- **`Trim`** – If True cut the end and start of profile to avoid to have part of the grid outside of the water limit

Returns a list with the updated profiles

`src.manage_grid_8.`**`get_crossing_segment_sub`**(*p1sub, p2sub, lim_here, lim_by_reachr, point_all, island, ind_seg_sub_ini=[0]*)

This function looks at one substrate segment and find the crossing points of this semgent with the different segment which composed the hydrological grid. This function is useful to cut the grid as a function of the form of the substrate layer (to avoid having cells in the hydrological grid which have two substrate value).

If island switch is True, lim_here is the limit of the island, so inside the polygon is outside the river. If island is false, lim_here is the limit of the reach under investigation

Parameters

- **p1sub** – the start point of the substrate semgent
- **p2sub** – the end point of the substrate segment
- **lim_here** – the reach?island limit given in the coordinate system
- **lim_by_reachr** – the limits for reach r which will be given to triangle given by point_all indices.
- **point_all** – all the point (ccordinates) which will be given to triangle
- **island** – a boolean indicating if we are on an island or not
- **ind_seg_sub_ini** – the indices of the first segment add by p1sub et p2sub by the reach. Only used island = true

Returns the updated point_all and lim_by_reach

src.manage_grid_8.**get_new_point_and_cell_1_profil**(*coord_pro_p*, *vh_pro_t_p*, *point_mid_x*, *point_mid_y*, *point_all*, *ikle*, *point_c*, *dir*)

This function is use by create_grid_one_profile. It creates the grid for one profile (one "line" of triangle). To create the whole grod this function is called for each profile.

Parameters

- **coord_pro_p** – the coordinates of the profile
- **vh_pro_t_p** – the height and velocity data of the profile analysed
- **point_mid_x** – the x coodinate of the points forming the middle profile
- **point_mid_y** – the y coordinate of the points forming the middle profile
- **point_all** – the point of the grid
- **ikle** – the connectivity table of the grid
- **point_c** – the central point of each cell
- **dir** – in which direction are we going around the profile (upstream/downstram)

Returns point_all, ikle, point_c (the centroid of the cell)

For more info, see the document "More info on the grid".

src.manage_grid_8.**inside_polygon**(*seg_poly*, *point*)

This function find if a point is inside a polygon, using a ray casting algorythm. It is called by various functions.

Parameters

- **seg_poly** – the segment forming the polygon
- **point** – the point which is indide or outside the polygon

Returns True is the point is inside the polygon, false otherwise

src.manage_grid_8.**interpo_linear**(*point_all*, *coord_pro*, *vh_pro_t*)

Using scipy.gridata, this function interpolates the 1.5 D velocity and height to the new grid It can be used for only one time step. The interpolation is linear. It is usually called after create_grid have been called.

Parameters

- **point_all** – the coordinate of the grid point
- **coord_pro** – the coordinate of the profile. It should be coherent with the coordinate from vh_pro. To insure this, pass coord_pro through the function "create_grid" with the same vh_pro as input

- **vh_pro_t** – for each profile, dist along the profile, water height and velocity at a particular time step

Returns the new interpolated data for velocity and water height

`src.manage_grid_8.`**`interpo_nearest`**(*point_all*, *coord_pro*, *vh_pro_t*)

Using scipy.gridata, this function interpolates the 1.5 D velocity and height to the new grid It can be used for only one time step. The interpolation is nearest neighbours. It is usually called after create_grid have been called.

Parameters

- **point_all** – the coordinate of the grid point
- **coord_pro** – the coordinate of the profile. It should be coherent with the coordinate from vh_pro. To insure this, pass coord_pro through the function "create_grid" with the same vh_pro as input
- **vh_pro_t** – for each profile, dist along the profile, water height and velocity at a particular time step

Returns the new interpolated data for velocity and water height

`src.manage_grid_8.`**`intersection_seg`**(*p1hyd*, *p2hyd*, *p1sub*, *p2sub*, *col=True*)

This function finds if there is an intersection between two segment (AB and CD). Idea from : http://stackoverflow.com/questions/563198/how-do-you-detect-where-two-line-segments-intersect It is based on the caluclaion of the cross-product z= 0 for 2D

Careful there is many function using this function, so change here should be thought about.

Parameters

- **p1hyd** – point A
- **p2hyd** – point B
- **p1sub** – point C
- **p2sub** – point D
- **col** – if True, colinear segment crossed. If false, they do not cross

Returns intersect (True or False) and the crossing point (if True, empty is False)

`src.manage_grid_8.`**`linear_h_cross`**(*p1*, *p2*, *h1*, *h2*)

This function is called by cut_2D_grid. It find the intersection point along a side of the triangle if part of a cells is dry.

Parameters

- **p1** – the coordinate (x,y) of the first point
- **p2** – the coordinate (x,y) of the first point
- **h1** – the water height at p1 (might be negative or positive)
- **h2** – the water height at p2 (might be negative or positive)

Returns the intersection point

`src.manage_grid_8.`**`main`**()

Used to test this module

`src.manage_grid_8.`**`newp`**(*p0*, *p1*, *extra_pro*)

This function find the start/end of the added profile. If only one profile is needed, it is just the point in the middle of the start/end of the profile. If mroe than one profile is needed, there are linearly distributed. This function only give the start and the end of the profile, the profile in full are constructed using find_profile_between()

Parameters

- **`p0`** – the point at the profile p
- **`p1`** – the point at the profile p-1
- **`extra_pro`** – the number of extra profile needed

Returns the start/end of the new profile

`src.manage_grid_8.`**`plot_grid`**(*point_all_reach*, *ikle_all*, *lim_by_reach*, *hole_all*, *overlap*, *point_c_all=[]*, *inter_vel_all=[]*, *inter_h_all=[]*, *path_im=[]*, *coord_pro2=[]*)

This is a function to plot a grid, it is very similar to the one from hec-ras2D.

Parameters

- **`point_all_reach`** – the grid point by reach
- **`ikle_all`** – the connectivity table by reach
- **`lim_by_reach`** – the segment giving the limits of the grid
- **`hole_all`** – the coordinates of the holes
- **`overlap`** – the point of each reach which are also on an other reach
- **`point_c_all`** – the centroid of each element
- **`inter_vel_all`** – the interpolated velocity for each reach
- **`inter_h_all`** – the interpolated height
- **`path_im`** – the path where to save the image

`src.manage_grid_8.`**`update_coord_pro_with_vh_pro`**(*coord_pro*, *vh_pro_t*)

The points describing the profile elevation and the points where velocity is measured might not be the same. Additionally,part of the profile might be dry and we have added points giving the wetted limit in vh_pro_t. They were are not in the original profil (coord_pro). In this function, coord_pro is recalculated to account for these modicfications. It is used by create_grid() and create_grid_one_profile, but only if vh_pro_t exists.

Parameters

- **`coord_pro`** – the original coord_pro
- **`vh_pro_t`** – the value and position of h and velcoity measurement with the river limits

Returns updated coord_pro

More information in the document "More info on the grid" (linked above)

4.10 Estimhab -source

in src/estimhab.py

The module contains the Estimhab model. For an explanation on the estimhab model, please see the pdf document `estimhab2008`

`src.estimhab.`**`estimhab`**(*qmes*, *width*, *height*, *q50*, *qrange*, *substrat*, *path_bio*, *fish_name*, *path_im*, *pict=False*)

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.. Unit in meter amd m^3/sec

Parameters

- **`qmes`** – the two measured discharge
- **`width`** – the two measured width
- **`height`** – the two measured height
- **`q50`** – the natural median discharge
- **`qrange`** – the range of discharge
- **`substrat`** – mean height of substrat
- **`pict`** – if true the figure is shown. If false, the figure is not shown
- **`path_im`** – the path where the image should be saved
- **`path_bio`** – the path to the xml file with the information on the fishes
- **`fish_name`** – the name of the fish which have to be analyzed

Returns habitat value and useful surface (VH and SPU) as a function of discharge

Technical comments and walk-through

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.

`src.estimhab.`**`main`**`()`

Used to test this module.

`src.estimhab.`**`pass_to_float_estimhab`**`(`*var_name*, *root*`)`

This is a function to pass from an xml element to a float

Parameters

- **`root`** – the root of the open xml file
- **`var_name`** – the name of the attribute in the xml file

Returns the float data

4.11 Stathab - source

in src/stathab_c

This module contains the function used to run the model stathab.For an explanation on the form of the stathab input, please see the pdf document `stathabinfo`

class `src.stathab_c.`**`Stathab`**(*name_prj*, *path_prj*)

The class for the Stathab model

`create_hdf5`()

A function to create an hdf5 file from the loaded txt. It creates "name_prj"_STATHAB.h5, an hdf5 file with the info from stathab

`dengauss`(*x*)

gaussian density, used only for debugging purposes. This is not used in Habby, but can be useful if scipy is not available (remplace all stat.norm.cdf with dengauss)

Parameters **`x`** – the parameter of the gaussian

Returns the gaussian density

`dist_h`(*sh0*, *h0*, *bornh*, *h*)

The calculation of height distribution acrros the river The distribution is a mix of an exponential and guassian.

Parameters

- **`sh0`** – the sh of the original data sh is the parameter of the distribution, gives the relative importance of ganussian and exp distrbution
- **`h`** – the mean height data
- **`h0`** – the mean height
- **`bornh`** – the limits of each class of height

Returns disth the distribution of heights across the river for the mean height h.

`dist_v`(*h*, *d*, *bornv*, *v*)

The calculation of velocity distribution across the river The distribution is a mix of an exponential and guassian.

Parameters

- **`h`** – the height which is related to the mean velocity v
- **`d`** – granulo moyenne
- **`bornv`** – the born of the velocity
- **`v`** – the mean velocity

Returns the distribution of velocity across the river

`find_sh0`(*disthmesr*, *h0*)

the function to find sh0, using a minimzation technique. Not used because the output was string. Possibly an error on the bornes? We remplaced this function by the function find_sh0_maxvrais().

Parameters

- **`disthmesr`** – the measured distribution of height
- **`h0`** – the measured mean height

Returns the optimized sh0

`find_sh0_maxvrais`(*disthmesr*, *h0*)

the function to find sh0, using the maximum of vraisemblance. This function aims at reproducing the results from the c++ code. Hence, no use of scipy

Parameters

- **`disthmesr`** – the measured distribution of height

- **h0** – the measured mean height

Returns the optimized sh0

load_stathab_from_hdf5()
A function to load the file from an hdf5 whose name is given in the xml project file

load_stathab_from_txt(*reachname_file*, *end_file_reach*, *name_file_allreach*, *path*)
A function to read and check the input from stathab based on the text files. All files should be in the same folder. The file Pref.txt is read in run_stathab. If self.fish_chosen is not present, all fish in the preference file are read.

Parameters

- **reachname_file** – the file with the name of the reaches to study (usually listirv.txt)
- **end_file_reach** – the ending of the files whose names depends on the reach
- **name_file_allreach** – the name of the file common to all reaches
- **path** – the path to the file

Returns the inputs needed for run_stathab

power_law(*qwh_r*)
The function to calculate power law for discharge and width ln(h0 = a1 + a2 ln(Q)

Parameters **qwh_r** – an array where each line in one observatino of Q, width and height

Returns the coeff of the regression

savefig_stahab()
A function to save the results in text and the figure

savetxt_stathab()
A function to save the stathab result in .txt form

stathab_calc(*path_pref='.'*, *name_pref='Pref.txt'*)
The function to calculate stathab output.

Parameters

- **path_pref** – the path to the preference file
- **name_pref** – the name of the preference file

Returns the biological preferrence index (np.array of [reach, specices, nbclaq] size), surface or volume by class, etc.

test_stathab(*path_ori*)
A short function to test part of the outputs against the C++ code, It is not used in Habby but it is practical to debug.

Parameters **path_ori** – the path to the files from stathab based on the c++ code

src.stathab_c.**load_float_stathab**(*filename*, *check_neg*)
A function to load float with extra checks

Parameters

- **filename** – the file to load with the path
- **check_neg** – if true negative value are not allowed in the data

Returns data if ok, -99 if failed

`src.stathab_c.`**`load_namereach`**(*path*, *name_file_reach='listriv.txt'*)

A function to only load the reach names (useful for the GUI)

:param path : the path to the file listriv.txt :param name_file_reach: In case the file name is not listriv.txt :return: the list of reach name

`src.stathab_c.`**`load_pref`**(*filepref*, *path*)

The function loads the different pref coeffficient contained in filepref

Parameters

- **`filepref`** – the name of the file (usually Pref.txt)
- **`path`** – the path to this file

Returns the name of the fish, a np.array with the differen coeff

`src.stathab_c.`**`main`**()

used to test this module.

4.12 Substrate

in src/substrate.py

This module contains the function to manage the substrate data. This is still a work in progress.

`src.substrate.`**`fig_merge_grid`**(*point_all_both_t*, *ikle_both_t*, *path_im*, *ikle_orr=[]*, *point_all_orr=[]*)

A function to plot the grid after it was merged with the substrate data. It plots one time step at the time.

Parameters

- **`point_all_both`** – the coordinate of the points of the updated grid
- **`ikle_both`** – the connectivity table
- **`path_im`** – the path where the image should be saved
- **`ikle_orr`** – the orginial ikle
- **`point_all_orr`** – the orginal point_all

`src.substrate.`**`fig_substrate`**(*coord_p*, *ikle*, *sub_info*, *path_im*, *xtxt=[-99]*, *ytxt=[-99]*, *subtxt=[-99]*)

The function to plot the raw substrate data, which was loaded before

Parameters

- **`coord_p`** – the coordinate of the point
- **`ikle`** – the connectivity table
- **`sub_info`** – the information on subtrate by element
- **`xtxt`** – if the data was given in txt form, the orignal x data
- **`ytxt`** – if the data was given in txt form, the orignal y data
- **`subtxt`** – if the data was given in txt form, the orignal sub data
- **`path_im`** – the path where to save the figure

`src.substrate.`**`grid_update_sub2`**(*ikle*, *coord_p*, *point_crossing*, *coord_sub*)

A function to find the updated grid with the substrate. More complicated than grid_update3 because it tries to makes new cell based on the lines linkes the centroid and the side of the trianlge. Looks more elegant at first but quite complicated and do not work for all cases. So it is not used.

Parameters

- **`ikle`** – the hydrological grid to be merge with the substrate grid
- **`coord_p`** – the coordinate of the point of the hydrological grid
- **`point_crossing`** – the crossing point, with the elemtn of the hydrological grid linked with it and the direction (nx,ny) of the substrate line at this point
- **`coord_sub`** – the coordinate of the substrate, only useful to if the the substrate cut two time the samie of a cell of the hydrological grid

Returns the new grid

`src.substrate.`**`grid_update_sub3`**(*ikle*, *coord_p*, *point_crossing*, *coord_sub*)

A function to update the grid after finding the crossing points

Parameters

- **`ikle`** – the hydrological grid to be merge with the substrate grid
- **`coord_p`** – the coordinate of the point of the hydrological grid
- **`point_crossing`** – the crossing point, with the elemtn of the hydrological grid linked with it and the direction (nx,ny) of the substrate line at this point
- **`coord_sub`** – the coordinate of the substrate, only useful to if the the substrate cut two time the samie of a cell of the hydrological grid

Returns the new grid

`src.substrate.`**`intersec_cross`**(*hyd1*, *hyd2*, *sub1*, *sub2*, *e=-99*, *nx=[]*, *ny=[]*)

A function function to calculate the intersection, segment are not parralel, used in case where we know that the intersection exists Also save various info with the intersection (element, direction, etc.)

Parameters

- **`hyd1`** – the first hydrological point
- **`hyd2`** – the second
- **`sub1`** – the first substrate point
- **`sub2`** – the second
- **`e`** – the element of the hydrological grid (optional)
- **`nx`** – the direction of the cutting part of the substrate grid (x dir)
- **`ny`** – the direction of the cutting part of the substrate grid (y dir)

Returns intersection and the direction of cutting part.

`src.substrate.`**`load_sub_shp`**(*filename*, *path*, *name_att='SUBSTRATE'*)

A function to load the substrate in form of shapefile.

Parameters

- **`filename`** – the name of the shapefile
- **`path`** – the path where the shapefile is
- **`name_att`** – the name of the substrate column in the attribute table

Returns grid in form of list of coordinate and connectivity table (two list) and an array with substrate type

`src.substrate.`**`load_sub_txt`**(*filename*, *path*)
A function to load the substrate in form of a text file. The text file must have 3 column x,y coordinate and substrate info, no header or title.

Parameters

- **`filename`** – the name of the shapefile
- **`path`** – the path where the shapefile is

Returns grid in form of list of coordinate and connectivity table (two list) and an array with substrate type and (x,y,sub) of the orginal data

`src.substrate.`**`main`**()
Used to test this module.

`src.substrate.`**`merge_grid_hydro_sub`**(*hdf5_name_hyd*, *hdf5_name_sub*, *default_data*)
After the data for the substrate and the hydrological data are loaded, they are still in different grids. This functions will merge both grid together. This is done for all time step and all reaches

Parameters

- **`hdf5_name_hyd`** – the path and name of the hdf5 file with the hydrological data
- **`hdf5_name_sub`** – the path and the name of the hdf5 with the substrate data
- **`default_data`** – The substrate data given in the region of the hydrological grid where no substrate is given

Returns the connectivity table, the coordinates, the substrated data, the velocity and height data all in a merge form.

`src.substrate.`**`point_cross2`**(*ikle*, *coord_p*, *ikle_sub*, *coord_p_sub*)
A function which find where the crossing points are. Crossing pitn are the points on the triangular side of the hydrological grid which cross with a side of the substrate grid. The algo based on finding if points of one elements are in the same polygon using a ray casting method

Parameters

- **`ikle`** – the connectivity table for the hydrological data
- **`coord_p`** – the coordinates of the points of the hydrological grid
- **`ikle_sub`** – the connecity vity table of the substrate
- **`coord_p_sub`** – the coordinates of the points of the substrate grid

Returns intersection

`src.substrate.`**`point_cross_bis`**(*ikle*, *coord_p*, *ikle_sub*, *coord_p_sub*)
A function which find where the crossing points are. Crossing pitn are the points on the triangular side of the hydrological grid which cross with a side of the substrate grid. Easier than point_cross 2 but slow, so it is not used.

Parameters

- **`ikle`** – the connectivity table for the hydrological data
- **`coord_p`** – the coordinates of the points of the hydrological grid
- **`ikle_sub`** – the connecity vity table of the substrate
- **`coord_p_sub`** – the coordinates of the points of the substrate grid

Returns intersection

CHAPTER

FIVE

VARIOUS NOTES

5.1 Translation of HABBY

In HABBY, it is possible to translate all strings which are in a python file (.py) which is in the src_GUI folder. It should be possible to translate also strings which are in a .py file which is in the .src folder if one modifies the .pro file, but this is not done yet. Also, it might not be necessary because ./src contains code which is not linked with the graphical interface. Hence, English might be sufficient here.

To add a new string to translate:

- Code as usual and write the string in English.
- Add self.tr() around the string a = Qlabel(self.tr("My message"))
- If the code is in a new python file (like the .py was just created), open the habby_trans.pro file which is src_GUI. Then add the line SOURCES+= new_file.py where new_file.py is the new python file.
- If you want to add a new language, add the line TRANSLATIONS += Zen_ES.ts in the case you want to add Spanish or any other language.
- Copy the files ZEN_EN.ts and ZEN_FR.ts from HABBY/translation to /src_GUI
- In the src_GUI folder, run the following command on the cmd: pylupdate5 habby_trans.pro. it will work if pylupdate is installed.
- It should update the .ts file (which is an xml file)
- Copy both .ts file back to HABBY/translation
- Open Qt linguist. This is a program that you need to install before. Open the French .ts file. The English should not need translation.
- Translate as needed and save in Qt Linguist.
- A .qm file is the binary representing the .ts file with all the translation. To create .qm file, type (in the cmd) lrelease file.ts. It will create a file.qm file
- Run HABBY. The string should be updated.

In the code

If the user asked for a new language, we need to reload the translator with the following lines:

app = QApplication.instance()

self.languageTranslator.load(file.qm, self.path_trans)

app.installTranslator(self.languageTranslator) with the appropriate name for "file.qm".

In HABBY, the list of the name of all qm file are in the variable self.file_langue in class MainWindows. Hence, we can follow the selected language using an integer self.lang (0 for English and 1 for French). We can now call self.file_langue[self.lang] to get the qm file in the right language. If a new language is added, it is necessary to add one string to this list and to modify the menu.

When the translator has been created, it is necessary to re-do all Widgets and Windows. This is not a problem when we open HABBY, but it can be a bit of work if the user asks for a change in language when HABBY is running. This is the function of the setlangue function. This function would work for all language (it takes an integer as input to know which language to use), but it needs to be modified if one modifies the Main_Windows Class strongly (notably if one add signals). The language should be saved in the user setting using Qsettings as it is done at the end of the setlangue function.

5.2 Create a .exe

Here are step to create a .exe using PyInstaller

- install Pyinstaller (pip install pyinstaller)
- cd "folder with source code"
- pyinstaller.exe [option] habby.py, with the option –onefile to get only one .exe and –windowed to not have the cmd which opens with the application.

Here are some common problems:

- ImportError: (No module named 'PyQt5.QtGui'): Copy the folder platform with qwindows.dll and add to the set_up.py "includes": ["PyQt5.QtCore", "PyQt5.QtGui"]
- This application fails to start because ... the Qt platform pugin windows: Copy the folder platform with qwindows.dll in it
- ImportError: h5Py "includes": ["h5py","h5py.defs", "h5py.utils", "h5py.h5ac", 'h5py._proxy'] etc if necessary
- Intel MKL fatal error copy the .dll missing (or just find an old dist and copy all mkl stuff) AND the libiomp5md.dll
- The translation does not work: Add the translation folder into the dist folder
- Do not find log0.txt (or crash when saving project): create a folder called src_GUI, copy the files log0.txt and restart_log0.txt from the src_GUI folder in the python module

5.3 Logging

General information

There are two different logs for HABBY. By default, the first one is called "name_projet".log and the second is called restart/_'name_project'.log. Their name and path can be changed in the xml project file. Both file are text file.

The first log is in the form of a python file with comments. If python and the necessary modules are installed on the machine, this log can be renamed "name.py" and started as a python file. In the command line, the following command should be used: python name.py. This file can be modified to create a new script to use HABBY in a different ways. For this, python syntax should be used.

The second log, called restart/_'name_project'.log, has limited functionalities but allows to re-start the HABBY simulation from the command line, without the need for python. Format of this file is described below. It is aimed to be readable and easily modifiable. To use the restart file, type in the command line: habby restart/_'name_project'.log. (not done yet)

Type of log and format

Currently, there are five types of outputs, which can be sent to the log:

- Comment, which should start with #. They will be sent to the python-type log file and to the GUI of Habby.
- Errors, which should start with word "Error". They will be sent to the python-type log file and to the GUI of Habby. In the GUI, they will appear in red.
- Warnings, which should start with the word "Warning". They will be sent to the python-type log file and to the GUI of Habby. In the GUI, they will appear in orange.
- Restart info, which should start with the word "restart'. They will be sent to the restart_'name_project'.log. The format will be developed afterwards.
- All types of text which do not start with these code words are only shown to the GUI of Habby.
- Python code, which should start with the line py followed by four spaces. It will be sent to the python-type log file. It is usually a function which is part of Habby code. The different arguments of the function should be given in the preceding lines.

Example

Let's write to the log a function which takes an integer and a string as input. The function is in the module called habby1, which is imported by default in the .log file. The strings to send as log would be:

- "# this my fancy function"
- "py my_int = " + str(my_int_in_code)
- "py my_string = '" + my_string_in_the _code+ "'"
- "py habby1.myfunc(my_int, my_string)"

A comment should be added before each chunk of python code to improve the readability.

Update the log

Let's consider a scenario where a new function has been written in a non-GUI module (class or function) and has to be called in the GUI in a method of a class. Let's call the new function new_func and the class in the GUI my_class.

To create a new line of log for new_func, one should follow these steps:

- A PyQtsignal with a string as argument should be added to my_class: send_log = pyqtSignal(str, name='send_log')
- If a log should be sent directly from my_class (for example, to say that new_func has been called), the signal should be emitted: self.send_log.emit('# new_func has been called'))
- In the new function, error and warning are written as follows: print("Error: here is an error.n") or print("Warning: This is just a warning.n")
- In my_class, error and warning are collected by redirecting stdout to a string. The following lines of code should be added around the calling of my_func():
  - sys.stdout = mystdout = StringIO() # redirect stdout
  - my_func(my_int,my_string)
  - sys.stdout = sys.__stdout__ # re-sent stdout to the cmd
  - str_found = mystdout.getvalue() # get all warning, error, text,...
  - str_found = str_found.split('n') # separate each message

    - for i in range(0, len(str_found)):
        - if len(str_found[i]) > 1:
            - self.send_log.emit(str_found[i]) #send the text

- To import StringIO, the following statement is needed at the start of the code: from io import StringIO
- If new_func is called from the command line, stdout will not be redirected and the errors or warnings will be printed on the cmd as usual. Stderr should be re-directed in a similar manner if needed.
- The signal should be collected in the function connect_signal_log in the Main_Windows_1.py. For this, a line should be added in the function: * self.my_class.send_log.connect(self.write_log)

Format of the restart file

To be determined.

5.4 Git - code management

Pour commencer:

- Choisir un dossier sur l'ordinateur local ou va se trouver les fichiers sources.
- cd « dossier avec les codes source»
- git config - - global user.name « username »
- git config - - global user.email « mail »
- git init
- lier le repertoire local avec le repertoire distant sur forge.irstea.fr

Pour mettre une nouvelle version sur le site web

- cd « dossier avec les codes source»
- git pull (prend la dernière version à jour sur le site et mets tous les fichiers ensemble) ou git fetch (prend juste les derniers fichiers sans mettre tous les fichiers ensemble).
- git add 'my_file.py ou .pyc' (choisit les fichiers qui doivent être envoyé), le signe * fonctionne.
- git log (donne l'historique)
- git status (donne les nouveaux fichiers locals)
- git commit –m « description » (commit localement)
- git push

Pour ajouter une nouvelle branche

- Donc pour avoir une partie du travail sépare du reste
- git checkout –b [branchname] pour créer la branche et y travailler
- git checkout [branchname] pour y travailler

5.5 Write the documentation

Habby uses Sphinx to document the code. Sphinx uses the docstring given in each function. Hence, it is necessary to write a docstring for each function which has to be documented.

To update the html documentation, go to the doc folder and execute the command: “make html”.

To update the Latex documentation, use the commande “make latex” and use Miketex to create the pdf. rts2pdf does not work with Python 3.

To add text in the documentation, modify the index.rst file in the doc folder. To add a new module to the documentation, add the module as written in the index.rts file in the doc folder. To add text comment, the index.rts file can also be direclty modified.

It is important to keep the formatting and the alignment.

If the module is in a new folder, the address of the folder must be added to the config.py file. It is better to not use absolute path for this, so it is possible to move the documentation on another computer. If the documentation does not run on a new computer, check the path given in the config.py file.

In the docstring, add as many blank lines as possible (in reasonable limit). This is easier for the formatting. To make a bullet list, one should use a tab and the symbol “*”. Using only the symbol * will fail.

To add a new title, do not start the title or the line of symbol under the title with a blank space.

5.6 License of used python modules

- h5py: BSD License
- Element tree (XML): MIT License
- numpy: BSD License
- matplotlib: BSD License
- PyQt5: GNU License

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX