PAMHYR2

A GRAPHICAL USER INTERFACE FOR 1D HYDRO-SEDIMENTARY MODELLING OF RIVERS

DEVELOPERS DOCUMENTATION

VERSION: V0.0.7
DOCUMENT LICENCE: GPLV3

April 26, 2024

Contents

Abstract

This document is for the use of developers. It describes the project architecture, the tools available to assist development and debugging. It also describes the procedures for creating packages and the configurations required to set up the gitlab runners. Finally, this document explains how documentation is written and modified, and how to contribute to the project by modifying, improving or adding documentation, translations or code.

1 Introduction

Pamhyr2 is free and open source software (FOSS) graphical user interface (GUI) for 1D hydro-sedimentary modelling of rivers developed in Python (with version 3.8). It use PyQt at version 5 and matplotlib in version 3.4.1 or later for the user insterface (see `/requirements.txt` for details). The architecture of project code follow the Qt Model/View architecture [1] (see details in section 2). Pamhyr2 packages can be build manually (see section 3.1), but there are automatically build with the gitlab-ci (see the section 3.2). Documentation files are written with org-mode[2], let see section 4. Finally, to see the contribution rules, see the section 5.

2 Architecture

Pamhyr2's architecture is based on Qt Model/View, see Figure 1. It is made up of several different components: the model (in blue), the graphical components (in red), the actions/delegates (in green), the commands (in purple), the solvers (in yellow) and the save file (in grey).

The model is a set of python classes and can be exported to a single SQLite3 format backup file. The view can be made up of various components, generally a Qt window with other view components, such as: a table, a text box, a button, a plot, and so on. The user can view the data using the view and interact with certain components. These components are linked to an action (such as a Python function) or to a delegate class. These

[1]Qt Model/View documentation: `https://doc.qt.io/qt-5/model-view-programming.html` (last access 2023-09-15)

[2]The org-mode website: `https://orgmode.org/` (last access 2023-09-15)

actions or delegate can create a command (based on Qt UndoCommand class), this command must implement two functions: One to modify the model, one to reverte this modification and reset the model to previous state. All model modification must be perform by a command to be cancelled. The user can also run a solver and add some simulation results to model data.

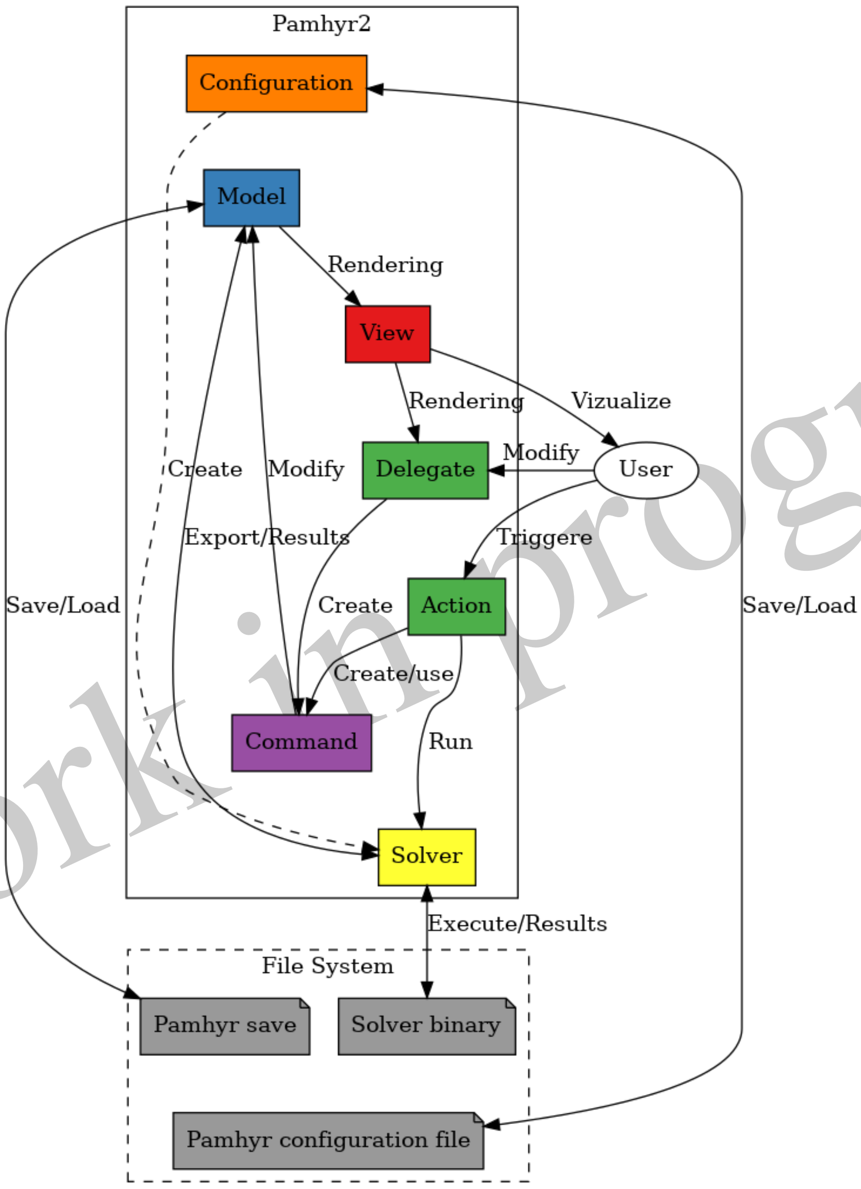


Figure 1: Pamhyr2 Model/View architecture scheme (inspired by Qt Model/View architecture `https://doc.qt.io/qt-5/model-view-programming.html`)

All the model source code are in the directory `src/Model` (let see section 2.1 for more details), the View components, delegate and command are in `src/View` (see section 2.2). Solvers classes are in `src/Solver` (see section 2.3).

The following sub section show examples of main Pamhyr2 internal class for view componants, but this documentation is not exhaustive, be free to watch existing code for more details and examples. In, addition some features are not factorise and must be implemented from scratch (directly with Qt for example).

2.1 Model

The model is a set of Python classes. In Pamhyr2, this classes must respect some constraint. Each model class must inherits `Model.Tools.SQLSubModel` abstract class, except the `Model.Study` class who inherits `Model.Tools.SQLModel` (see 2.1.1).

The model entry point is the Study class. It contains infomation about the study: their name, description, time system, and so on. Their contains a River object too. This river object inherits the network graph and contains a list of `RiverNode` and a list of `RiverReach` (an edge who contains a source node, and destination node). `RiverReach` contrains geometry, so, the river network (node and edge) associated with the geometry forms the basis of the model, and the other components are linked to one of these basic components.

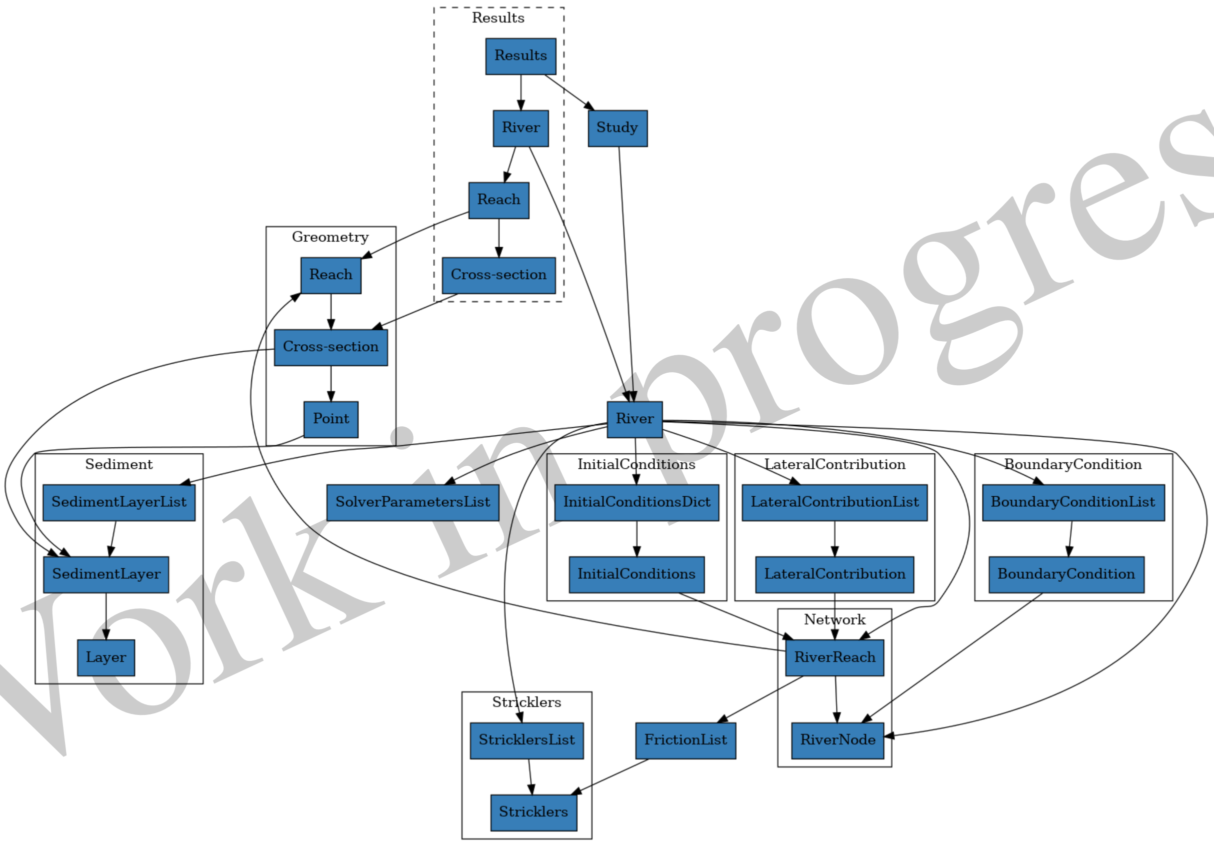


Figure 2: Pamhyr2 model class dependencies (A -> B means A can contain references to B)

2.1.1 SQL

The model must be export to a database file to create a study save file. This file use SQLite3[3] format and the extention `.pamhyr`. So, each model componante must be register into this study file. To create, update, set and get information into SQLite database we use SQL command. The database use version number and some modification could be perform to update database. For each model componante, correspond one or more SQL table to store information. To normalize the interaction with database we made two classes, SQLModel and SQLSubModel. The Study class use SQLModel because is the top of the model hierachy. The rest of model class inherits to SQLSubModel.

[3]The SQLite web site: `https://www.sqlite.org/index.html` (last access 2023-09-20)

A class who inherits SQLSubModel, must implement some methods:

- `_sql_create`: Class method to create the database scheme
- `_sql_update`: Class method to update the database scheme if necessary
- `_sql_load`: Class method to load data from DB
- `_sql_save`: Method to save current object into DB

Class method take in arguments: The class (`cls`), a function to execute SQL command into the database (`execute`). In addition, the update method take the previous version of database, load method take an optional arguments `data` if additional infomation ar needed, and who can contains whatever you want. The method save take in arguments the current object (`self`), a function to execute SQL command into the database (`execute`), and optional data (`data`).

The class who inherits SQLSubModel can also define an class attribute `_sub_classes` to set a formal class dependencies into database. This attribute is use at database creation to create all table, and at update to update all the database table. Let see examples of SQLSubModel usage for two classes Foo and Bar with Foo contains list of Bar (Listing 1 and 2).

Let see the results database scheme for Pamhyr2 at version v0.0.7 in Figure 3.

2.1.2 List class

A abstract class PamhyrModelList is available and provide some of basic methods for object list in Model. This abstract class implement some classic action in View like: insert new object, delete object, sort, move object up, move object down, and so on. An variant exists for multiple list with same type of object, each sublist is called tab, because in View, this kind of list si prensented in different table PamhyrModelListWithTab.

2.1.3 Dict class

A abstract class PamhyrModelDict is available and provide some of basic methods for object dictionary in Model. This class is like PamhyrModelList but use a dictionary instead of list.

2.2 View

Pamhyr2 use Qt as graphical user interface library with the application "Qt designer" for windows or widget creation (see 2.2.1) and "Qt linguist" for interface translate (see 2.2.2). In addition, we use matplotlib as ploting library (see 2.2.6).

Typically, each model componant have an associated window in application to add, delete or edit this componant. At top level of View directory we found the `MainWindow.py` file and some sub-directories. A view sub-directory contains: A `Window.py` file, a `Table.py` file with table model definition if nessessary, one or more `Plot*.py` file with plot class definition, a `translate.py` file with componant translate, and possible other files or sub-directories.

2.2.1 UI file

We define as possible all Pamhyr2 windows and custom widgets with "Qt designer". This application generate UI file who describes interface organisation with table, layout, button, etc. This method is faster than hand made windows and widget creation, and saves us some purely descriptive code. The UI files are saved into `src/View/ui` for window, and `/src/View/ui/Widgets` for custom widget.

2.2.2 Translate

```python
from Model.Tools.PamhyrDB import SQLSubModel

class Bar(SQLSubModel):
    _id_cnt = 0
    def __init__(self, id = -1, x = 0, y = 0):
        self._x = x
        self._y = y
        if id == -1:
            self.id = Bar._id_cnt + 1
        else:
            self.id = id
        Bar._id_cnt = max(id, Bar._id_cnt+1)

    @classmethod
    def _sql_create(cls, execute):
        execute("""
          CREATE TABLE bar (
            id INTEGER NOT NULL PRIMARY KEY,
            x INTEGER NOT NULL,
            y INTEGER NOT NULL,
            foo_id INTEGER NOT NULL,
            FOREIGN KEY(foo_id) REFERENCES foo(id),
          )""")
        return True

    @classmethod
    def _sql_update(cls, execute, version):
        # If version is lesser than 0.0.2, add column to bar table
        major, minor, release = version.strip().split(".")
        if major == minor == "0":
            if int(release) < 2:
                execute("ALTER TABLE bar ADD COLUMN y INTEGER")
        return True

    @classmethod
    def _sql_load(cls, execute, data = None):
        new = []
        table = execute(
            f"SELECT id, x, y FROM bar WHERE foo_id = {data['id']}"
        )
        for row in table:
            bar = cls(
                id = row[0], x = row[1], y = row[2],
            )
            new.append(bar)
        return new

    def _sql_save(self, execute, data = None):
        execute("INSERT INTO bar (id,x,y,foo_id) VALUES " +
                f"({self.id}, {self._x}, {self._y}, {data['id']})")
```

Listing 1: Exemple of class Bar inherits SQLSubModel.

```python
class Foo(SQLSubModel):
    _id_cnt = 0
    _sub_classes = [Bar]

    def __init__(self, id = -1, name = ""):
        self._name = name
        self._bar = []
        # ...

    @classmethod
    def _sql_create(cls, execute):
        execute("""
          CREATE TABLE foo (
            id INTEGER NOT NULL PRIMARY KEY,
            name TEXT NOT NULL,
          )
        """)
        return cls._create_submodel(execute)

    @classmethod
    def _sql_update(cls, excute, version):
        return cls._update_submodel(execute, version)

    @classmethod
    def _sql_load(cls, execute, data = None):
        new = []
        table = execute(
            "SELECT id, name FROM foo"
        )
        for row in table:
            foo = cls(
                id = row[0],
                name = row[1],
            )
            data = {
                "id": row[0],      # Current Foo ID
            }
            foo._bar = Bar._sql_load(execute, data=data)
            new.append(foo)
        return new

    def _sql_save(self, execute, data = None):
        execute(f"DELETE FROM foo WHERE id = {self.id}")
        execute(f"DELETE FROM bar WHERE foo_id = {self.id}")
        # Save new data
        execute(f"INSERT INTO bar (id,name) VALUES ({self.id}, {self._name})")
        data = {"id": self.id}
        for bar in self._bar:
            bar._sql_save(execute, data=data)
```

Listing 2: Exemple of class Foo inherits SQLSubModel and contains a list of Bar object (Listing 1).

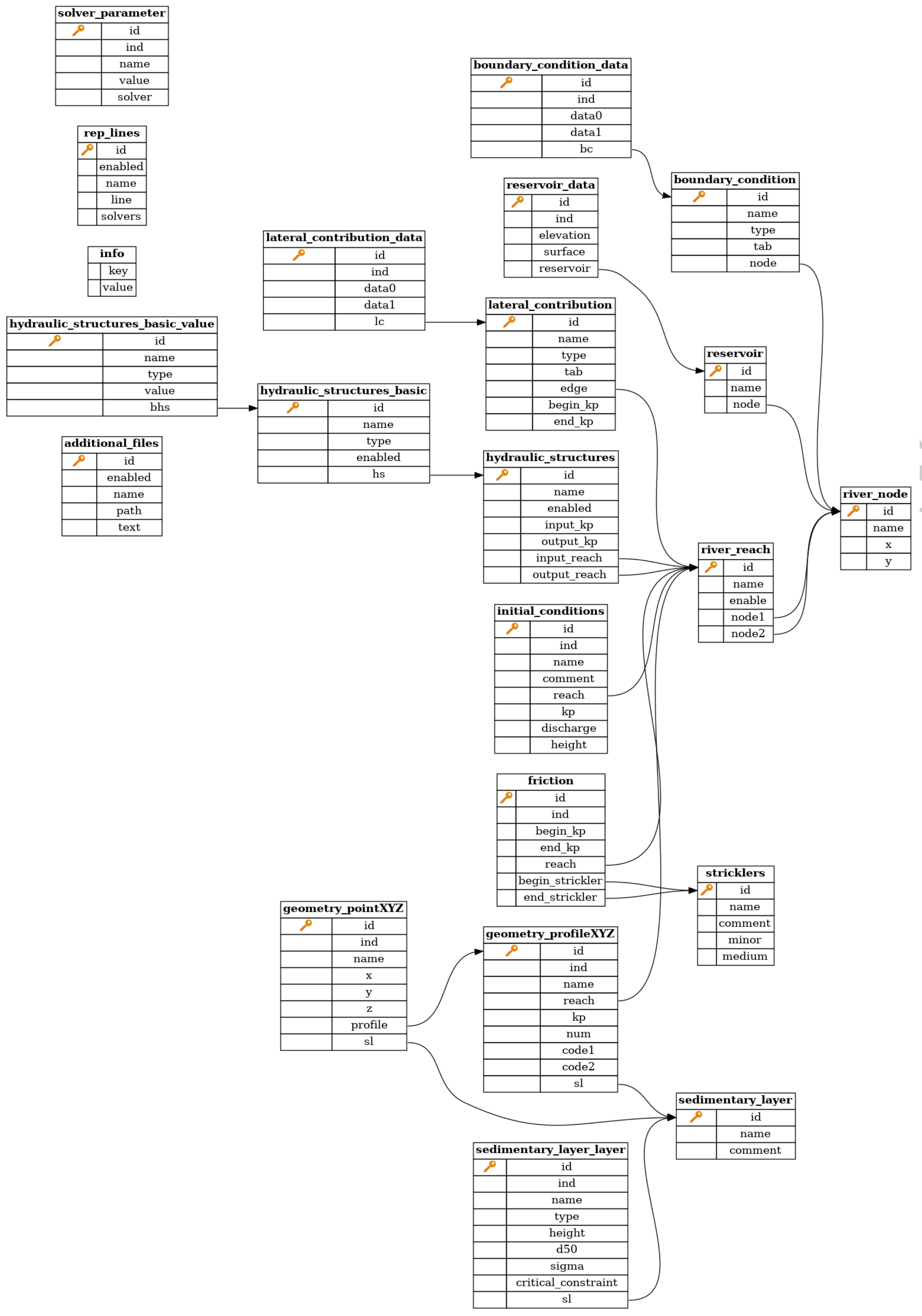

solver_parameter: id, ind, name, value, solver
rep_lines: id, enabled, name, line, solvers
info: key, value
hydraulic_structures_basic_value: id, name, type, value, bhs
additional_files: id, enabled, name, path, text
lateral_contribution_data: id, ind, data0, data1, lc
hydraulic_structures_basic: id, name, type, enabled, hs
boundary_condition_data: id, ind, data0, data1, bc
reservoir_data: id, ind, elevation, surface, reservoir
lateral_contribution: id, name, type, tab, edge, begin_kp, end_kp
hydraulic_structures: id, name, enabled, input_kp, output_kp, input_reach, output_reach
initial_conditions: id, ind, name, comment, reach, kp, discharge, height
friction: id, ind, begin_kp, end_kp, reach, begin_strickler, end_strickler
geometry_pointXYZ: id, ind, name, x, y, z, profile, sl
geometry_profileXYZ: id, ind, name, reach, kp, num, code1, code2, sl
sedimentary_layer_layer: id, ind, name, type, height, d50, sigma, critical_constraint, sl
boundary_condition: id, name, type, tab, node
reservoir: id, name, node
river_reach: id, name, enable, node1, node2
river_node: id, name, x, y
stricklers: id, name, comment, minor, medium
sedimentary_layer: id, name, comment

Figure 3: SQLite database scheme at Pamhyr2 version v0.0.7 (generate with `https://gitlab.com/Screwtapello/sqlite-schema-diagram`)

```python
from PyQt5.QtCore import QCoreApplication
from View.Tools.PamhyrTranslate import PamhyrTranslate
_translate = QCoreApplication.translate

class MyTranslate(PamhyrTranslate):
    def __init__(self):
        super(MyTranslate, self).__init__()

        # Add traduction to global dictionary
        self._dict["My"] = _translate("My", "FooBar")
        # Add an additional translate dictionary
        self._sub_dict["table_headers"] = {
            "foo": _translate("My", "Foo"),
            "bar": _translate("My", "Bar"),
            "baz": _translate("My", "Baz"),
        }
```

Listing 3: Example of `PamhyrTranslate` class implementation with a global traduction for *FooBar* and a additional dictionary `table_headers`

2.2.3 Window

The abstract class PamhyrWindow and PamhyrDialog are used for most of Pamhyr2 window. These class allow to create an window for Pamhyr2 GUI and implemente some useful methods. The super class method difine some generic value from optional parameters, for examples:

- `self._study`: The study giving in constructor parameters `study` (typically a `Model.Study` class object)
- `self._config`: The configuration giving in constructor parameters `config` (typically a `Config` class object)
- `self._trad`: The traductor dictionary giving in constructor parameters `trad` (typically a `Model.Tools.Pamhy` class object)

Typically we called method `setup_*`, the method to initialize some window componants or connections.

2.2.4 Table

An abstract class PamhyrTableModel is available to define a simple QAbstractTableModel shortly. In simple cases, there are only `data` and `setData` methode to implement, but the constructor needs more information than a classic QAbstractTableModel class.

2.2.5 UndoCommand

All model modification must be done by an QUndoCommand, this command allow to undo and redo an action. This a Qt class wi can inherit to define custom undo command (see example Listing 7)

All undo command must be push into a `QUndoStack` (see Listing 8) to perform the action and allow user undo and redo this action. In PamhyrWindow (and PamhyrDialog) the undo stack is automatically create if the option `"undo"` is activate at window creation, this stack is accessible at `self._undo_stack`.

2.2.6 Plot

To define a new plot you can create a class who inherit to PamhyrPlot. The creator need at leaste five argument:

- A `canvas` of type `MplCanvas`

```python
from View.Tools.PamhyrWindow import PamhyrWindow
from View.My.Translate import MyTranslate
from View.My.Table import MyTableModel

class MyWindow(PamhyrWindow):
    _pamhyr_ui = "MyUI"
    _pamhyr_name = "My window"

    def __init__(self, study=None, config=None,
                 my_data=None,
                 parent=None):
        self._my_data = my_data

        super(MyWindow, self).__init__(
            # Window title
            title = self._pamhyr_name + " - " + study.name,
            # Window standard data
            study = study, config = config,
            trad = MyTranslate(),
            parent = parent,
            # Activate undo/redo and copy/paste shortcut
            options = ["undo", "copy"]
        )

        # Add custom data to hash window computation
        self._hash_data.append(self._my_data)

        # Setup custom window components
        self.setup_table()
        self.setup_connections()

    def setup_table(self):
        # Init table(s)...

    def setup_connections(self):
        # Init action connection(s)...

    # ...
```

Listing 4: Example of Pamhyr2 window

```python
from View.Tools.PamhyrTable import PamhyrTableModel

class MyTableModel(PamhyrTableModel):
    def data(self, index, role):
        # Retrun data at INDEX...

    @pyqtSlot()
    def setData(self, index, value, role=Qt.EditRole):
        # Set VALUE at INDEX...
```

Listing 5: Definition of a table model from `PamhyrTableModel` in a file `View/My/Table.py`.

```python
# Table model creation (Window.py: setup_table)
table_headers = self._trad.get_dict("table_headers")
self._model = MyTableModel(
    table_view = table,          # The table view object
    table_headers = table_headers, # The table column headers dict
                                   # (with traduction)
    editable_headers = ["foo", "bar"], # List of editable column name
    delegates = {
        "bar": self.my_delegate, # Custom delegate for column 'bar'
    },
    data = self._my_lst,          # The data
    undo = self._undo_stack,      # The window undo command stack
)
```

Listing 6: Using the table model defined in Listing 5 in window funtion `setup_table` defined Listing 4.

```python
class AddNodeCommand(QUndoCommand):
    def __init__(self, graph, node):
        QUndoCommand.__init__(self)

        self._graph = graph
        self._node = node

    def undo(self):
        self._graph.remove_node(self._node.name)

    def redo(self):
        self._graph.insert_node(self._node)
```

Listing 7: Example of custom UndoCommand, this command allow to add a node to graph in river network window (method redo), and delete it to graph with undo method

```python
self._undo_stack.push(
    AddNodeCommand(
        self._graph,
        node
    )
)
```

Listing 8: Example of UndoCommand push into an undo stack.

- A (optional) `trad` of type `PamhyrTranslate`
- A `data` used in `draw` and `update` to create and update the plot
- A optional `toolbar` of type `PamhyrToolbar`
- A `parent` window

This class must implement two method `draw` and `update`, the first method to draw the plot from scratch, the second to update the plot if data has changed.

```python
from View.Tools.PamhyrPlot import PamhyrPlot

class MyPlot(PamhyrPlot):
    def __init__(self, canvas=None, trad=None, toolbar=None
                 data=None, parent=None):
        super(MyPlot, self).__init__(
            canvas=canvas,
            trad=trad,
            data=data,
            toolbar=toolbar,
            parent=parent
        )

      self.label_x = self._trad["x"]
      self.label_y = self._trad["y"]

      # Optional configuration
      self._isometric_axis = False

      self._auto_relim_update = True
      self._autoscale_update = True

    def draw(self):
        # Draw function code...

    def update(self):
        # Update function code...

    def clear(self):
        # Clear plot values...

    # ...
```

2.3 Solver

The Pamhyr2 architecture allow to define multiple solver. A solver is define by a:

- type
- name
- description,
- path
- command line pattern
- (optional) input formater path
- (optional) input formater command line
- (optional) output formater path
- (optional) output formater command line

Let see Figure 4, the application can implement different solver type, this solver type implement the code for export study to solver input format, and read the solver output to study results. There exists a generic solver with a generic input and output format, the type could be use to use a solver not implemented in Pamhyr2, but this solver must can read/write input and output generic format or use external script. There is possible to define different solver with the same type, for example two differents version of the same solver. Finaly, with input and output formater is possible to execute a code on distant computer, for example, over ssh.

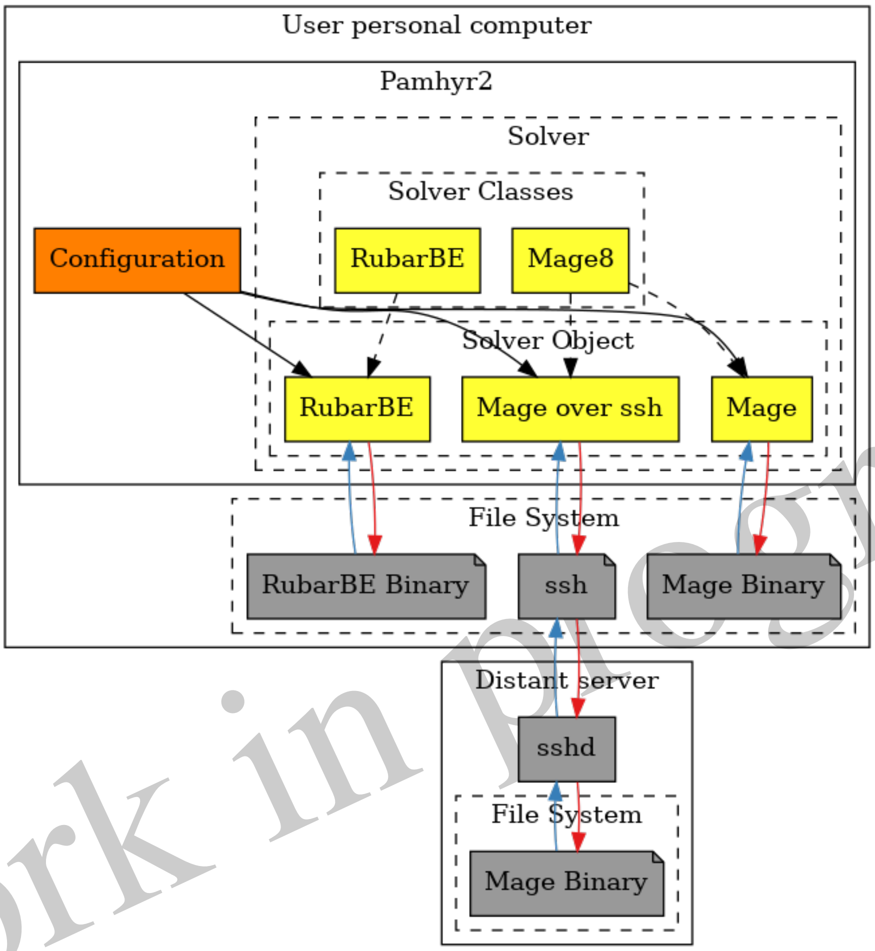


Figure 4: Scheme of multiple solver configured, one Rubarbe solver and two Mage solver with one on local machine and one on a distant machine accessed over ssh

Let see Figure 5 the temporal order of action to run a solver and get results:

- (1) Write solver input file(s) using the study data
- (2) Run the solver
- (2.1) The solver read the input file(s)
- (2.2) The solver compute results and write it to solver output file(s)
- (3) Pamhyr2 create a `Results` object
- (3.1) The Pamhyr2 solver class read solver output file(s) and complete Results with readed data

In case of generic solver (or a solver with input and output formater) the temporal order of action is prensented in Figure 6.

To implement a Solver in Pamhyr2, there exists a abstract class `Solver.AbstractSolver`. A class who herits this class, must implement different methods:

- `export`: Export the study to solver input file(s)
- `input_param`: Return the solver input parameter(s) as string

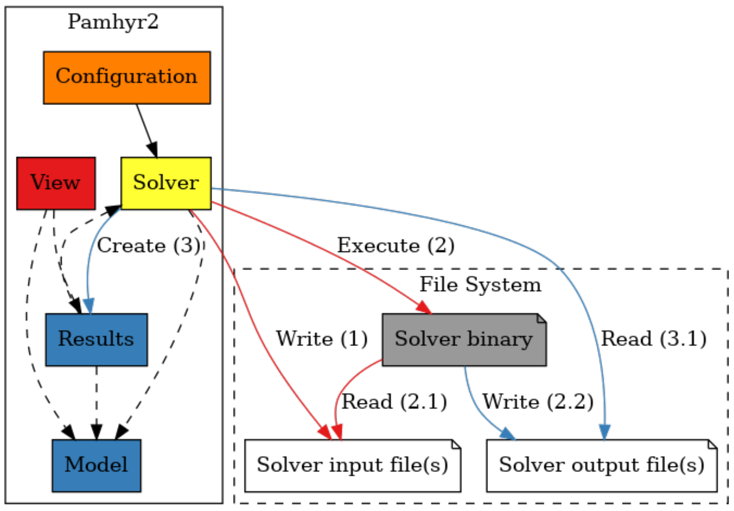


Figure 5: Pamhyr2 solver execution pipeline architecture scheme

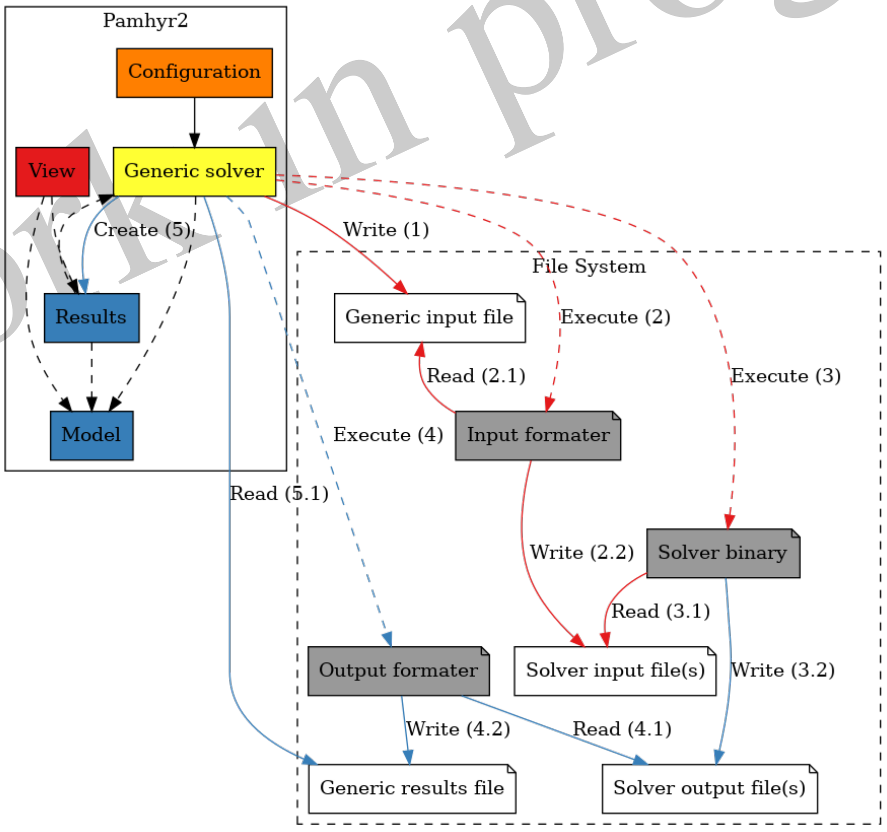


Figure 6: Pamhyr2 generic solver execution pipeline architecture scheme

- `log_file`: Return the solver log file name as string
- `results`: Read the solver output file(s) and return a `Model.Results` object.

2.4 Unit tests

A very small part of Pamhyr2 has unit test. This part is limited to the Model.

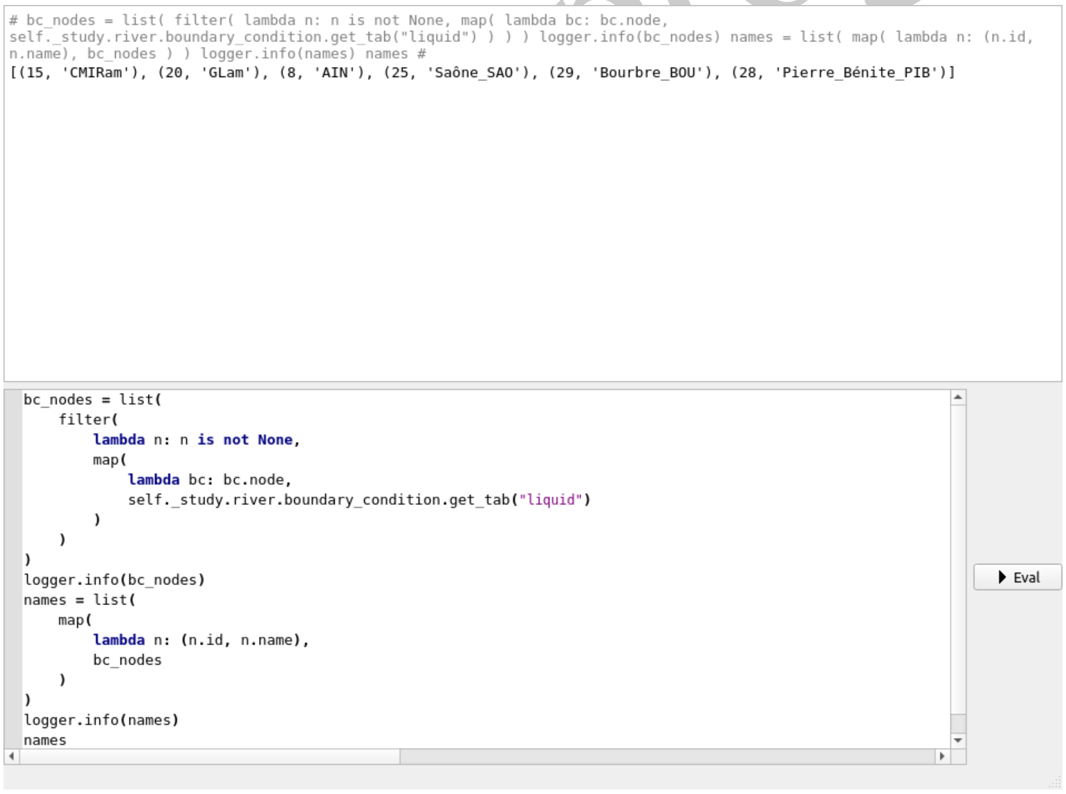
```
python3 -m venv test
. test test/bin/activate
pip3 install -U -r ./full-requirements.txt

cd src/
python3 -Walways -m unittest discovert -v -t .
```

2.5 The debug mode

To activate an deactivate the Pamhyr2 debug mode you can open the configuration window and type "Ctrl+G" or run Pamhyr2 with command line:

```
./Pamhyr2 debug
```

This mode add some log and add two action in main window menu: "About > Debug" open a window with Python Repl in current Python environement, and "About > Debug SQLite" who open the application SQLiteBrowser (if installed) on current Study to explore the study data base file.



```
# bc_nodes = list( filter( lambda n: n is not None, map( lambda bc: bc.node,
self._study.river.boundary_condition.get_tab("liquid") ) ) ) logger.info(bc_nodes) names = list( map( lambda n: (n.id,
n.name), bc_nodes ) ) logger.info(names) names #
[(15, 'CMIRam'), (20, 'GLam'), (8, 'AIN'), (25, 'Saône_SAO'), (29, 'Bourbre_BOU'), (28, 'Pierre_Bénite_PIB')]
```

```
bc_nodes = list(
    filter(
        lambda n: n is not None,
        map(
            lambda bc: bc.node,
            self._study.river.boundary_condition.get_tab("liquid")
        )
    )
)
logger.info(bc_nodes)
names = list(
    map(
        lambda n: (n.id, n.name),
        bc_nodes
    )
)
logger.info(names)
names
```

Figure 7: Pamhyr2 debug Python REPL

3 Build the project

The project uses gitlab-ci runners to build packages, but it is possible to build packages manually.

3.1 Building packages

If you need an hand made package, you can script available in `packages` directory.

3.1.1 GNU/Linux

On GNU/Linux building GNU/Linux packages is easy, you just need python in version 3.8 must be installed with venv and pyinstaller packages (see Listing 9 for Debian and derived system). Finally, run the `linux.sh` script (see Listing 10).

```
sudo apt install python3.8
python3 -m pip install venv
python3 -m pip install pyinstaller
```

Listing 9: Install environment on GNU/Linux

```
cd packages
./linux.sh
```

Listing 10: Build GNU/Linux package

3.1.2 Windows

To make the Windows packages you have two choice: If you use Windows you can use the script `packages/windows.bat`, other else you can use the script `packages/wine.sh`. Each script need a specific software environment.

On windows, you needs python on version 3.8, pyinstaller and NSIS[4] installed. On GNU/Linux you need wget, wine and winetricks installed.

3.2 Setup the CI environment

Pamhyr2 need a Linux ci-runner and a Windows ci-runner for building package. The windows ci-runner could run on a Wine environement.

3.2.1 Linux

The Linux ci-runner need some software and dependencies in addtion of gitlab-ci.

```
sudo apt install                                   \
     emacs emacs-goodies-el                        \
     texlive-full                                  \
     python3.8 python3.8-venv
sudo python3 -m pip install pyinstaller
```

[4]The NSIS web site: `https://sourceforge.net/projects/nsis/`

3.2.2 Windows (Wine)

The ci-runner environment for Wine need at least wine version 8, let see who to add wine official depot to your linux distribution.

```
sudo apt install wine-stable winetricks
```

In addition, the environment need windows version of:

- Python 3.8.10
- Git
- PowerShell
- Gitlab-ci
- Nsis

Now, we can install `pyinstaller` on this windows environment:

```
wine python -m pip install pyinstaller
```

4 Documentation files

This document and the user documentation are org files. This text file format is formatted so that it can be exported in different formats: PDF (with latex), ODT, HTML, etc. It was originally designed for the GNUEmacs[5] text editor, but can be edited with any text editor. Here we take a look at the different features used in these documents.

4.1 Org-mode

4.1.1 Document structure

Org uses the `*` character to define a new document section. To add a sub-section, you can add an additional `*` to the current section[6].

```
* Top level headline
** Second level
*** Third level
    some text
*** Third level
    more text
* Another top level headline
```

4.1.2 Format

Org-mode is a markup file, using markup in the text to modify the appearance of a portion of text[7].

Markup	Results
`*Bolt*`	**Bolt**
`/Italic/`	*Italic*
`_underline_`	<u>underline</u>
`=verbatim=`	`verbatim`
`~code~`	`code`
`+strike-through+`	~~strike-through~~

[5]The GNUEmacs project website: `https://gnu.org/s/emacs/` (last access 2023-09-15)
[6]See document structure documentation: `https://orgmode.org/org.html#Headlines` (last access 2023-09-15)
[7]See markup documentation: `https://orgmode.org/org.html#Emphasis-and-Monospace` (last access 2023-09-15)

4.1.3 Source code blocks

You can add some code blocks[8] in the document.

Here is an example for python source code:

```
#+CAPTION: Get os type name in Python code
#+begin_src python
import os

print(f"Document build on system: {os.name}")
#+end_src
```

If you use GNUEmacs, it is also possible to run the code inside a block and export (or not) the reuslts in the document.

```python
import os

print(f"Document build on system: {os.name}")
```

Listing 11: Get os type name in Python code

```
Document build on system: posix
```

4.1.4 LaTeX

If we export the file to PDF, org-mode use LaTeX. So we can add some piece of LaTeXinto the document[9]. For exemple, we can add math formula like `$E=mc^2$` ($E = mc^2$) or `\[E=mc^2\]`:

$$E = mc^2$$

But we can also add every type of LaTeX:

```
# Add latex in line
#+LATEX: <my line of latex>

# Add multiple line of LaTeX
#+BEGIN_EXPORT latex
<my latex here>
#+END_EXPORT
```

It is also possible to add specific LaTeXfile header with `#+LATEX_HEADER`. In this document we use the file `doc/tools/latex.org` for all LaTeXheaders.

4.1.5 Macro

In this document, we use a few macros[10] to simplify writing. They allow you to define sequences of text to be replaced, so that the macro name is replaced by its value. They are defined in the `doc/tools/macro.org` file. Once defined, they can be used in the document as follows: `{{{<macro-name>}}}`. You can also have macros with arguments, in this case: `{{{<macro-name>(arg1,...)}}}`. Les macros peuvent aussi utiliser du code emacs-lisp.

[8] See org-mode documentation for source code: `https://orgmode.org/org.html#Working-with-Source-Code` (last access 2023-09-15)

[9] See LaTeXpart in documentation: `https://orgmode.org/org.html#Embedded-LaTeX` (last access 2023-09-15)

[10] See marcos documentation `https://orgmode.org/org.html#Macro-Replacement` (last access 2023-09-15)

```
# Exemple of macro définition

#+MACRO: toto              tata
#+MACRO: add               \(($1 + $2)\)
#+MACRO: emacs-version      (eval (nth 2 (split-string (emacs-version))))
```

Macro apply:

- Marco `{{{toto}}}`: tata
- Marco `{{{add(x,y)}}}`: $(x + y)$
- Marco `{{{emacs-version}}}`: 26.3

4.1.6 References

The references use the LaTeXbibtex tools. The bib file is in `/doc/tools/ref.bib` and use for developers and user documentation. In document, use `{{{cite(<name>)}}}` to cite a paper.

4.2 Export

To export the files, a `build.sh` script is available in the org files directories. On GNU/Linux system you can build the documentation PDF file with the command `./build.sh`. Texlive package must be installed, you can install only needed packages or all texlive packages, for example on Debian (and some derived system) use command Listing 12.

```
sudo apt install texlive-full
```

Listing 12: Installation command for texlive full on Debian system

Some org-mode configuration used in documentations files are define in `/doc/tools/`:

- `PamhyrDoc.cls`: The LaTeXdocument class
- `macro.org`: Available macro
- `latex.org`: LaTeXconfigutation for documentations files
- `setup.el`: GNUEmacs configuration to build documentations
- `ref.bib`: Bibtex files for documentations files

5 How to contribute?

Pamhyr2 is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License[11], either version 3 of the License, or any later version.

5.1 Guidelines

To contribute to Pamhyr2, we expect a minimum of respect between contributors. We therefore ask you to respect the following rules regarding communication and contribution content:

- No gender, racial, religious or social discrimination
- No insults, personal attacks or potentially offensive remarks
- Pamhyr2 is free software, and intended to remain so, so take care with the licensing of libraries and external content you want to add to the project
- Humour or hidden easter eggs are welcome if they respect the previous rules

[11]The GPLv3 web page: `https://www.gnu.org/licenses/gpl-3.0.en.html`

5.2 Make a contribution

There are several ways to contribute: you can report a bug by creating an issue on the project's gitlab page[12], or you can create a merge request on the same page with the changes you have made to the code, translation or documentation.

The Pamhyr2 copyright is owned by INRAE[13], but we keep a record of each contributors. If you made a modification to pamhyr2 software, please add your name at the end of `AUTHORS` file and respect the Listing 13 format. You can update this file information for following contribution.

```
<first name> <last name> [(optional) email], <organisation>, <years>
```

Listing 13: `AUTHORS` file format

```
Sylvain COULIBALY, INRAE, 2020 - 2022
Théophile TERRAZ, INRAE, 2022 - 2024
Pierre-Antoine ROUBY, INRAE, 2023 - 2024
```

Listing 14: Current `AUTHORS` file

5.3 Translate

You can improve or add translation for the project. To contribute to Pamhyr2 translate, you need to use Qt Linguist[14]. Open Qt-linguist and edite the translation (`.ts`) file, finally, commit the new version of file and make a merge request.

If you want add a new language, edit the script `src/lang/create_ts.sh` like Listing 15. Run the script and open the new file with Qt-linguist, setup target language (Figure 8) and complete translation. Finally, commit the new file and make a merge request.

```
...
LANG="fr it"
...
```

Listing 15: Example of modified `src/lang/create_ts.sh` to add italian (it) translate for Pamhyr2

5.4 Code contribution

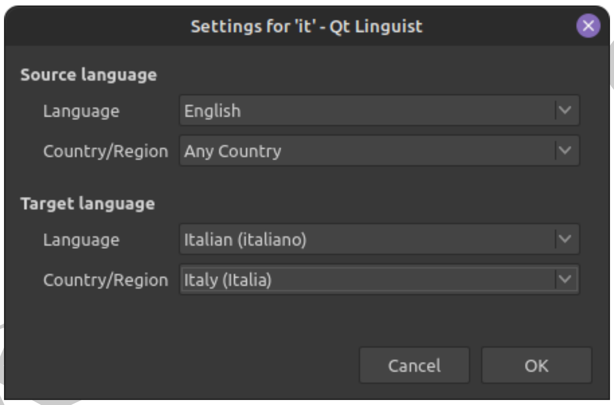
If you are developper you can improve and/or add features to Pamhyr2. Please, follow the architecture described in section 2 as closely as possible. Keep the code simple, clear and efficient as possible. The master branch is reserved for the project maintainer; you can create a new branch or fork the project before the request.

References

[12]The Pamhyr2 Gitlab project page: `https://gitlab.irstea.fr/theophile.terraz/pamhyr`

[13]The INRAE web site: `https://www.inrae.fr/`

[14]The Qt linguist documentation web page: `https://doc.qt.io/qt-5/qtlinguist-index.html` (last access 2023-09-18)


Settings for 'it' - Qt Linguist

Source language

Language: English

Country/Region: Any Country

Target language

Language: Italian (italiano)

Country/Region: Italy (Italia)

Figure 8: Qt linguist lang setup example with italian.