TelApy

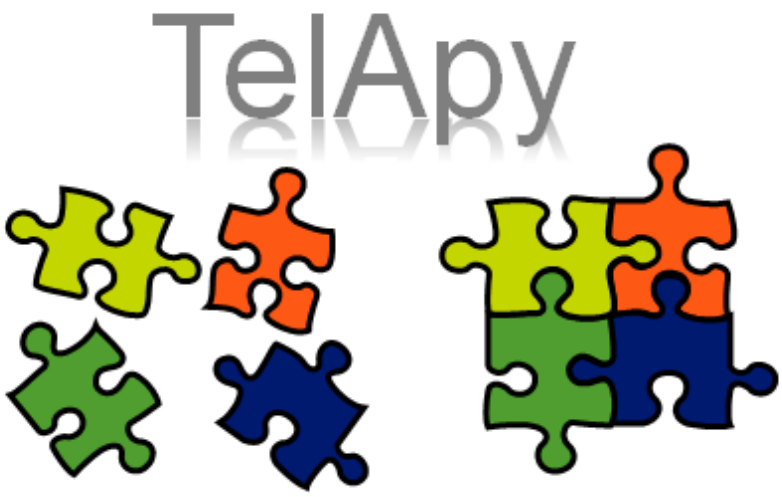
User Manual

Cedric Goeury, Yoann Audouin and Fabrice Zaoui

Version v7p3
March 16, 2018

Contents

1. INTRODUCTION



This guide aims to explain the use of the TELAPY module of the TELEMAC-MASCARET SYSTEM. This module aims to provide python control of TELEMAC API (Application Program Interface).
The API's main goal is to have control on a simulation while running a case. For example, it must allow the user to stop the simulation at any time step, retrieve some variable values and change them. In order to make this possible, a Fortran structure called instance is used in the API. It contains a list of variables declared as pointers that are pointing to variables. This gives direct access to the physical memory of variables, and allows therefore to retrieve their values, and modify them. Furthermore, based on modifications in TELEMAC main subroutines, it is possible to run hydraulic case time step by time step.
Thus, the APIs development allows the interoperability of the TELEMAC-MASCARET SYSTEM modules. Interoperability is the ability of a computer system to operate with other existing or future informatic products without restricting access or implementation. It, then, becomes natural to drive these APIs using Python programming language. In fact, Python is a portable, dynamic, extensible, free language, which allows (without imposing) a modular approach and

object oriented programming. Python has been developed since 1989 by Guido van Rossum and many volunteer contributors. In addition of the benefits of this programming language, Python offers a large amounts of interoperable libraries. The link between various interoperable libraries with TELEMAC-MASCARET SYSTEM APIs allows the creation of an ever more efficient computing chain able to more finely respond to various complex problems.
Consequently, the TelApy module has the ambition to facilitate the usage of TELEMAC-MASCARET SYSTEM for optimization, coupling, uncertainty quantification,... applications.

2. TelApy module description

As mentioned in the introduction part (section 1), the TELAPY module is used to control the APIs of TELEMAC-MASCARET SYSTEM in the Python programming language. The TELEMAC-MASCARET SYSTEM APIs are developed in Fortran. However, it is relatively easy to use these Fortran routines directly in Python using the "f2py" tool of the Python Scipy library [1]. This tool will make it possible to compile Fortran code to make it accessible and usable in Python. This compilation step is directly integrated into the compilation of the TELEMAC-MASCARET SYSTEM and is thus transparent to the user. Moreover, in order to make the TELAPY module more user friendly, a Python wrapper has been developed in order to encapsulate and simplify the different API Python calls. This set of transformation constitutes the TELAPY module.
The first section of this chapter is dedicated to the Fortran API of TELEMAC-MASCARET SYSTEM. Then, the PythonTELAPY module is presented.

2.1 TELEMAC-MASCARET SYSTEM Fortran API description

An API (Application Programming Interface) is a library allowing to control the execution of a program. Here is part of the definition from Wikipedia:

"In computer programming, an application programming interface (API) specifies a software component in terms of its operations, their inputs and outputs and underlying types. Its main purpose is to define a set of functionalities that are independent of their respective implementation, allowing both definition and implementation to vary without compromising each other.
In addition to accessing databases or computer hardware, such as hard disk drives or video cards, an API can be used to ease the work of programming graphical user interface components, to allow integration of new features into existing applications (a so-called "plug-in API"), or to share data between otherwise distinct applications. In practice, many times an API comes in the form of a library that includes specifications for routines, data structures, object classes, and variables."

The API's main goal is to have control on a simulation while running a case. For example, it must allow the user to stop the simulation at any time step, retrieve some variables values and change them. In order to make this possible, a Fortran structure called instance is used in the API. This informatic struture is described in the paragraph 2.1.1. The instance structure gives direct access to the physical memory of variables, and allows therefore the variable control (see paragraph 2.1.2). Furthermore, based on modifications in TELEMAC-MASCARET SYSTEM

main subroutines, it is possible to run hydraulic case time step by time step. This will be presented in the paragraph 2.1.3. And finaly, the parallelism is evoked (paragraph 2.1.4). All Fortran routines are available in the directory "api" of TELEMAC-MASCARET SYSTEM sources.

Warning:
In the following sections, all presented API routines are related to TELEMAC-2D. However, the API implementation of TELEMAC-MASCARET SYSTEM modules is generic that is to say based on the same structure (in the following routines the sequence "t2d" related to TELEMAC-2D module can be replace by "sis" related to SISYPHE module for instance).

2.1.1 Instanciation

An instance is an informatic structure that gathers all the variables alterable by the API. The definition of the "instance" structure is made in a Fortran module dedicated to this purpose and is composed of:

- An index defining the instance I.D.
- A string which can contain error messages.
- Some pointers to the concerned module variables. This is what makes it possible to manipulate the variables of the module by having a direct access to their memory location. The list of variables that can be accessed is given in the Table 2.1. Not all variables within TELEMAC-2D are there. However adding a new variable is pretty easy, the 5 steps procedure is described in section 4.1.2.

In addition to the instance definition, the module includes all routines needed to manipulate it (creation, deletion, and so on).

2.1.2 Variable control

The way in which the instance is defined (pointers) allows manipulation of variables during the simulation. So, to get information on the variables the following set of functions has been implemented:

- get the list of variables reachable with the API:

```
subroutine get_var_list_t2d(varname, varinfo, ierr)
!
character(len=*), intent(out) :: varname(nb_var_t2d)
character(len=*), intent(out) :: varinfo(nb_var_t2d)
integer, intent(out) :: ierr
!
end subroutine
```

  - **varname**: An array of dimension `nb_var_t2d` containing strings of size `t2d_var_len`, which gives the list of the variable names.
  - **varinfo**: An array of dimension `nb_var_t2d` containing strings of size `t2d_var_len`, which gives a short description for each variable.
  - **ierr**: 0 if everything went smoothly an error index otherwise.

- get the type of a variable.

```
subroutine get_var_type_t2d(varname, vartype,
                            readonly, ndim, ierr)
!
character(len=t2d_var_len),  intent(in)  :: varname
character(len=t2d_type_len), intent(out) :: vartype
integer,                     intent(out) :: readonly
integer,                     intent(out) :: ndim
integer,                     intent(out) :: ient
integer,                     intent(out) :: jent
integer,                     intent(out) :: kent
integer,                     intent(out) :: getpos
integer,                     intent(out) :: setpos
integer,                     intent(out) :: ierr
!
end subroutine get_var_type_t2d
```

  - **varname**: The name of the variable.
  - **vartype**: The type of the variable (DOUBLE, INTEGER, STRING, BOOLEAN).
  - **readonly**: 0 if the variable can be only read, and 1 if it can be write as well.
  - **ndim**: Number of dimensions of the variable (maximum is 3 and for a scalar the number of dimension is 0)
  - **ient**: 1 if the index corresponds to a mesh point
  - **jent**: 1 if the index corresponds to a mesh point
  - **kent**: 1 if the index corresponds to a mesh point
  - **getpos**: give the function position after which one the variable information can be "get"
  - **setpos**: give the function position after which one the variable information can be "set"
  - **ierr**: 0 if everything went smoothly, an error index otherwise.

- get the size of a variable.

```
subroutine get_var_size_t2d(id, varname,
                            dim1, dim2, dim3, ierr)
!
  integer,                  intent(in) :: id
  character(len=t2d_var_len), intent(in)  :: varname
  integer,                  intent(out) :: dim1
  integer,                  intent(out) :: dim2
  integer,                  intent(out) :: dim3
  integer,                  intent(out) :: ierr
!
end subroutine get_var_size_t2d
```

  - **id**: The id of the instance.
  - **varname**: The name of the variable.

- **dim1**: Size of the first dimension.
- **dim2**: Size of the second dimension.
- **dim3**: Size of the third dimension.
- **ierr**: 0 if everything went smoothly, an error index otherwise.

Comment:
If the desired variable is a scalar then, it has no first, second or third dimension and `dim1, dim2, dim3` are equal to zero.

- get/set the value of a variable for a given index.

  These functions depend on the variable type. However, the concept is the same for all kind of types. Moreover, the type distinction is removed in TelApy based on Python benefits. Therefore, for these reasons, the next routine header is focused only on the double type variable. The information for the other routine are available in the Fortran file section of API in the TELEMAC-MASCARET SYSTEM.

```
subroutine get_double_t2d
  (id, varname, value, global_num,
   index1, index2, index3, ierr)
!
  integer,              intent(in)  :: id
  character(len=t2d_var_len), intent(in)  :: varname
  double precision,     intent(out) :: value
  integer,              intent(in)  :: index1
  integer,              intent(in)  :: index2
  integer,              intent(in)  :: index3
  logical,              intent(in)  :: global_num
  integer,              intent(out) :: ierr
!
end subroutine get_double_t2d
```

```
subroutine set_double_t2d
  (id, varname, value, global_num,
   index1, index2, index3, ierr)
!
  integer,              intent(in)  :: id
  character(len=t2d_var_len), intent(in)  :: varname
  double precision,     intent(in)  :: value
  integer,              intent(in)  :: index1
  integer,              intent(in)  :: index2
  integer,              intent(in)  :: index3
  logical,              intent(in)  :: global_num
  integer,              intent(out) :: ierr
!
end subroutine set_double_t2d
```

  - **id**: the id of the instance.

- **varname**: The name of the variable.
- **value**: Contains the value to be read/written.
- **index1**: Index of the first dimension (For array of at least one dimension, not used otherwise).
- **index2**: Index of the second dimension (For array of at least two dimension, not used otherwise).
- **index3**: Index of the third dimension (For array of at least three dimension, not used otherwise).
- **global_num**: Logical to know if the given indexes are given in mesh global numbering
- **ierr**: 0 if everything went smoothly, an error index otherwise.

2.1.3 Computation control

The computation control is carried out using some specific routines to launch the simulation. These routines constitute a decomposition of the main program of each TELEMAC-MASCARET SYSTEM modules corresponding to the following different computation steps:

- Configuration setup. This function initialises the instance and the listing output. The instance, characterised by the **id** integer parameter, represents a run of TELEMAC-2D.

```
subroutine run_set_config_t2d(id,lu,lng,ierr)
!
  integer, intent(out) :: id
  integer, intent(in)  :: lu, lng
  integer, intent(out) :: ierr
!
end subroutine run_set_config_t2d
```

  - **id**: Contains the id of the instance.
  - **lu**: Defines the output canal for TELEMAC-2D (6 will be the standard output).
  - **lng**: Defines the output language of TELEMAC-2D (1 For French, 2 for English).
  - **ierr**: 0 if everything went smoothly an error index otherwise.

> **Comment:**
> In further version, the API will be able to have multiple instances running at the same time. In the current version you can only have one instance at a time.

- Reading the TELEMAC-2D steering file. This function reads the case file and set the variable of the TELEMAC-2D steering file accordingly.

```
subroutine run_read_case_t2d(id,cas_file, dico_file, ierr)
!
  integer, intent(out) :: id
  character(len=144), intent(in)  :: cas_file
  character(len=144), intent(in)  :: dico_file
  integer, intent(out) :: ierr
!
end subroutine run_read_case_t2d
```

  - **id**: The id of the instance.
  - **cas_file**: Path to the steering file.
  - **dico_file**: Path to the TELEMAC-2D dictionary.
  - **ierr**: 0 if everything went smoothly, an error index otherwise.

Warning:
With the API we are not using the temporary folder (this folder was created by the Python environment and all the file declared in the steering file where copied and renamed inside that folder) which means that the name and path given in the steering file will be used.

- Memory allocation. This function run the allocation of all the data needed in TELEMAC-2D. Any modifications to quantities of TELEMAC-2D should be done before the call to that function.

```
subroutine run_allocation_t2d(id,ierr)
!
  integer, intent(in)  :: id
  integer, intent(out) :: ierr
!
end subroutine run_allocation_t2d
```

  - **id**: The id of the instance.
  - **ierr**: 0 if everything went smoothly, an error index otherwise.

- Initialization. This function will do the setting of the initial conditions of TELEMAC-2D It corresponds to the time-step 0 of a TELEMAC-2D run.

```
subroutine run_init_t2d(id,ierr)(id,ierr)
!
  integer, intent(in)  :: id
  integer, intent(out) :: ierr
!
end subroutine run_run_init_t2d
```

  - **id**: The id of the instance.
  - **ierr**: 0 if everything went smoothly an error index otherwise.

- Computation function that runs one time-step of TELEMAC-2D. To compute all time steps, a loop on this function must be done.

```
subroutine run_timestep_t2d(id,ierr)
!
  integer, intent(in)  :: id
  integer, intent(out) :: ierr
!
end subroutine run_timestep_t2d
```

  - **id**: The id of the instance.
  - **ierr**: 0 if everything went smoothly, an error index otherwise.

- Finalization. This function concludes the run of TELEMAC-2D and will deallocate all the arrays and delete the instance. To start a new execution of TELEMAC-2D the function RUN_SET_CONFIG must be run again.

```
subroutine run_finalize_t2d(id,ierr)
!
  integer, intent(in)  :: id
  integer, intent(out) :: ierr
!
end subroutine run_finalize_t2d
```

  - **id**: The id of the instance.
  - **ierr**: 0 if everything went smoothly an error index otherwise.

For each routine defined above, the identity number of the instance is used as an input argument allowing all computation variables to be linked with the corresponding instance pointers. These routines are then called in the same order to insure a correct execution of the computation in the API main program.

2.1.4 Parallelisation

All steps associated with parallel computation must be performed by the user when he chooses to launch his calculation on several processors. In this case, after initializing the MPI environment, the user must partition the input files (geometry file, boundary conditions files, and so on) using the Fortran function "partel". Then, when the calculation is complete, it is necessary to merge each subdomains result files using the "gretel" routine of the TELEMAC-MASCARET SYSTEM. The MPI environment can then be closed.

2.2 TelApy module

It is relatively easy to use the Fortran API routines directly in Python using the "f2py" tool of the Python Scipy library. This tool will make it possible to compile Fortran code such as it is accessible and usable in Python. For more details on this tool, the interested reader can refer directly to [1]. However, using the advantage of the Python language, it is possible to implement a wrapper in order to provide user friendly function of the Fortran API. Thus, a Python overlay was developed in order to encapsulate and simplify the different API Python calls. The different Python functions written to simplify the use of API are available in the directory "*HOMETEL/scripts/python27/TelAPy/*". A Doxygen documentation is available and allows the user to visualize Python classes, functions that can be used as well as its input and output variables and so on.
In order to launch the Doxygen documentation, the user needs to copy and paste the link "*HOMETEL/documentation/TelAPy/doxygen/html/index.html*" into his favorite internet browser. Then, the user can navigate in the Doxygen environment in order to find information. For example, if the package list widget is selected, then, all Python tools are visible such as:

- TelApy
  - api

        * api_module: The generic Python class for TELEMAC-MASCARET SYSTEM APIs
        * generate_study: Automatic generator of TELEMAC-MASCARET SYSTEM API template Python
        * hermes: Input and Output library allowing to write and read different TELEMAC-MASCARET SYSTEM formats
        * sis: The SISYPHE Python class for APIs
        * t2d: The TELEMAC-2D Python class for APIs

    - tools
        * genop: Optimization tool based on genetic algorithms
        * newop: Optimization tool based onthe SciPy minimizer function
        * polygon: Function allowing to give point indices which are in a polygon defined by the user

Then, if some information are desired on a specific class as for example "api_module", by clicking on this class, all function contained in it are listed as follow:

- `def _init_`: Constructor for api module
- `def set_case`: Read the steering file and run allocation
- `def init_state_default`: Initialize the state of the model Telemac 2D with the values of discharges and water levels as indicated by the steering file
- `def run_one_time_step`: Run one the time step
- `def run_all_time_steps`: Run all the time steps
- `def get_mesh`: Get the 2D mesh of triangular cells
- `def get_node`: Get the nearest node number for the coordinates (xval, yval)
- `def get_elem`: Get the triangle where the point (xval, yval) is
- `def show_mesh`: Show the 2D mesh with topography
- `def list_variables`: List the names and the meaning of available variables and parameters
- `def get`: Get the value of a variable of TELEMAC-MASCARET SYSTEM modules
- `def set`: Set the value of a variable of TELEMAC-MASCARET SYSTEM modules
- `def get_array`: Retrieves all the values from a variable into a numpy array
- `def set_array`: Changes all the values from a variable into a numpy array
- `def get_on_polygon`: Retrieves values for point within the polygon poly. Warning: this works only on array that are of size NPOIN
- `def set_on_polygon`: Set a varname value on all points that are within the polygon poly Warning this works only on array that are of size NPOIN

- `def get_on_range`: Retrieves the values of the variable on the range given as argument
- `def set_on_range`: set the values of the variable on the range given as argument
- `def get_error_message`: Get the error message from the Fortran sources
- `def finalize`: Delete the TELEMAC-MASCARET SYSTEM instance
- `def generate_var_info`: Returns a dictionary containing specific information for each variable

For each of the previous functions all information concerning the inputs and outputs are available such as for example for the "set" fonction which allows to get the value of the variable of TELEMAC-MASCARET SYSTEM module:

```
def set(self,varname,value,i=0,j=0,k=0,global_num=True)
```

- **varname**: Name of the variable
- **i**: index otherwise.index of the first dimension
- **j**: index otherwise.index of the second dimension
- **k**: index otherwise.index of the third dimension
- **global_num**: are the index on local/global numbering variable value

Variable name	Definition
MODEL.TIME	Current time
MODEL.BCFILE	Boundary condition file
MODEL.AT	Current time
MODEL.BCFILE	Boundary condition file name
MODEL.BND_TIDE	Option for tidal boundary conditions
MODEL.BOTTOMELEVATION	Level of the bottom
MODEL.CHESTR	Strikler on point
MODEL.FAIR	Fair on point
MODEL.COTE	Prescribed elevation value
MODEL.CPL_PERIOD	Coupling period with sisyphe
MODEL.DEBIT	Discharge on frontier
MODEL.DEBUG	Activating debug mode
MODEL.FLUX_BOUNDARIES	Flux at boundaries
MODEL.GEOMETRYFILE	Name of the geomery file
MODEL.METEOFILE	Name of the binary atmospheric file
MODEL.FO2FILE	Name of the formatted data file 2
MODEL.LIQBCFILE	Name of the liquid boundaries file
MODEL.GRAPH_PERIOD	Graphical output period
MODEL.HBOR	Boundary value on h for each boundary point
MODEL.IKLE	Connectivity table between element and nodes
MODEL.INCWATERDEPTH	Increase in the the depth of the water
MODEL.KP1BOR	Points following and preceding a boundary point
MODEL.LIHBOR	Boundary type on h for each boundary point
MODEL.LISTIN_PERIOD	Listing output period
MODEL.LIUBOR	Boundary type on u for each boundary point
MODEL.LIVBOR	Boundary type on v for each boundary point
MODEL.LT	Current time step
MODEL.COMPLEO	Graphic output counter
MODEL.NBOR	Global number of boundary points
MODEL.NELEM	Number of element in the mesh
MODEL.NELMAX	Maximum number of elements envisaged
MODEL.NPOIN	Number of point in the mesh
MODEL.NPTFR	Number of boundary points
MODEL.NTIMESTEPS	Number of time steps
MODEL.NUMLIQ	Liquid boundary numbers
MODEL.POROSITY	Porosity
MODEL.RESULTFILE	Name of the result file
MODEL.SEALEVEL	Coefficient to calibrate sea level
MODEL.TIDALRANGE	Coefficient to calibrate tidal range
MODEL.UBOR	Boundary value on u for each boundary point
MODEL.VBOR	Boundary value on v for each boundary point
MODEL.VELOCITYU	Velocity on u
MODEL.VELOCITYV	Velocity on v
MODEL.WATERDEPTH	Depth of the water
MODEL.X	X coordinates for each point of the mesh
MODEL.XNEBOR	Normal X to 1d boundary points
MODEL.Y	Y coordinates for each point of the mesh
MODEL.YNEBOR	Normal Y to 1d boundary points
MODEL.EQUATION	Name of the equation used

Table 2.1: Accessible variables through the API

3. Getting Started with TelApy module

3.1 TelApy module installation

In order to be able to use TelApy module, the TELEMAC-MASCARET SYSTEM and all its external libraries must be compiled in dynamic form. The explanation of dynamic compilation is available on the TELEMAC-MASCARET SYSTEM website in the wiki category "installation notes 2" (`http://wiki.opentelemac.org/doku.php?id=installation_notes_2_beta`).
Then after compiling the module, the use of TelApy is presented and explained in some notebooks documentation. In fact, the TelApy module is provided with some tutorial intended for people who want to run TELEMAC-2D in an interactive mode with the help of the Python programming language.

3.2 How to run notebook documentation

In order to read notebooks, the user needs to install a notebook viewer such as jupyter notebook. Notebook documents (or "notebooks", all lower case) are documents which contain both computer code (e.g. Python) and rich text elements (paragraph, equations, figures, links, etc...). Notebook documents are both human-readable documents containing the analysis description and the results (figures, tables, etc..) as well as executable documents which can be run to perform data analysis.
First and foremost, the Jupyter Notebook is an interactive environment for writing and running code. The notebook is capable of running code in a wide range of languages. However, each notebook is associated with a single kernel. This notebook is associated with the IPython kernel, therefore runs Python code. More details on the installation and use can be found in the Jupyter website `http://jupyter.org`.

3.3 Notebook examples in TelApy

As already mentioned, the TelApy module is provided with some tutorial intended for people who want to run TELEMAC-2D in an interactive mode with the help of the Python programming language. Currently, these tutorials based on notebook examples are located in the folder $\$HOMETEL/scripts/python27/TelApy/notebooks$. In this directory, two notebooks categories are available:

- the folder *api* contains tutorials to run TELEMAC-2D in an interactive mode with the help

of the Python programming language. The interactive mode means that communication with TELEMAC-2D becomes possible throughout the simulation without having to stop it. One can easily set or get the value of any variables at each time step with the help of special communication functions: the Application Programming Interfaces (API).

  - "telemac2d.ipynb" is a tutorial dedicated to the API description and the presentation of TELEMAC-2D Python functionalities.
  - "telemac2d_example.ipynb" is a tutorial showing a computation run of TELEMAC-2D in an interactive mode with the help of the Python programming language.

- The folder *optim* is dedicated, as indicated by his name, to optimization problem.

  - "genop.ipynb" is a tutorial intended for people who want to use the Genop optimizer. Genop (Genetic optimizer) is a Python package implementing the Genetic Algorithm (GA) for a mono-objective minimization. GA is a derivative-free optimizer. This metaheuristic mimics the natural evolution with the repeated application of operators (selection, mutation, crossover, etc.) in order to evolve a set of solutions towards the optimality. People interested in this class of algorithms may refer to Genetic Algorithms or Genetic Programming for more information.
  - "telemac2d_optim_genop.ipynb" is a tutorial showing how to optimize a TELEMAC-2D case with a genetic algorithm based on Genop Python module.
  - "telemac2d_optim_newop.ipynb" is a tutorial showing how to optimize a TELEMAC-2D case with the deterministic algorithm based on the SciPy package.

In order to see and run the notebook examples, the user need to launch the command "jupyter notebook example_name.ipynb". This will launch a Jupyter server and a page should appear in your default internet browser.

4. Developer manual

This section will described firstly how the API is implemented in Fortran. The first section will describe the instance type used in the API. This structure shows all variables accessible by the Fortran API. Then, a section is dedicated to guidelines and advices to allow a user to add access to a new variable from TELEMAC-2D to the API. Then the second part is focused on the Python guidelines and convention of the TelApy module.

4.1 Instance

As already explained in section 2, in order to control the data used by TELEMAC-2D a Fortran Structure called "instance" containing all global variables of TELEMAC-2D is used. A part of this strucrure is presented below. In addition, some functions, described below, are available in order to handle that structure. All the instance functions can be found in the module `m_instance_t2d`.

```
type instance_t2d
   ! run position
   integer myposition
   ! list of all the variable for model
   type(bief_obj), pointer :: hbor
   type(bief_obj), pointer :: ubor
   type(bief_obj), pointer :: vbor
   type(bief_obj), pointer :: u
   type(bief_obj), pointer :: v
   type(bief_obj), pointer :: chestr
   double precision, pointer :: flux_boundaries(:)
   double precision, pointer :: cote(:)
   double precision, pointer :: debit(:)
!
   type(bief_mesh), pointer :: mesh
!
   type(bief_obj), pointer :: lihbor
   type(bief_obj), pointer :: liubor
   type(bief_obj), pointer :: livbor
   type(bief_obj), pointer :: numliq
!
```

```
    integer,          pointer :: nit
    integer,          pointer :: lt
!
    type(bief_file), pointer :: t2d_files(:)
    integer :: maxlu_t2d
    integer :: maxkey
    integer, pointer :: t2dres
    integer, pointer :: t2dgeo
    integer, pointer :: t2dcli
!
    character(len=144), pointer :: coupling
!
    type(bief_obj), pointer :: te5
    type(bief_obj), pointer :: zf
    type(bief_obj), pointer :: h
    type(bief_obj), pointer :: dh
!
    integer, pointer :: debug
!
    end type ! model_t2d
```

During the use of the API, two arrays are available in order to keep track the used instances.

```
INTEGER, PARAMETER :: MAX_INSTANCES=10
TYPE(INSTANCE_T2D), POINTER :: INSTANCE_LIST(:)
LOGICAL, ALLOCATABLE :: USED_INSTANCE(:)
```

Where `INSTANCE_LIST` will contain all the instances used and `USED_INSTANCE` tells you if the id is used.

4.1.1 Instance functions

In addition to the instance definition, the API includes all routines needed to manipulate it:

- `CHECK_INSTANCE_T2D`. This function just checks that the id number is valid (between 1 and max_instances).
- `CREATE_INSTANCE_T2D`. This function creates new instance and returns the id of that instance.
- `DELETE_INSTANCE_T2D`. This function deletes the instance and make the id available.

4.1.2 How to add access to a new variable

In order to add access to a new variable via the API, the following steps must be done:

1. Get the name of the variable in `declarations_telemac2d` and add ", target" in its declaration.
2. Add that variable in the instance structure (file `m_instance_t2d.f`).
3. Add the initialisation in `create_instance_t2d`.
4. Add the variable in the get/set function that befits it.

5. Add the size of the variable in `get_var_size_t2d_d`.
6. Add the type of the variable in `get_var_type_t2d_d`.
7. Add the variable in `GET_VAR_LIST_T2D_D`.
8. Increase the value of `t2d_nb_var` in `m_handle_var.f`.

4.2 Coding Convention

Warning:
Be careful, the node numbering is dependent of the convention code used. When Fortran API are used the node numbering is considered from 1 to *npoin* and in python is considered from 0 to $(npoin-1)$

4.2.1 Fortran part of API

The Fortran part of TELEMAC-MASCARET SYSTEM API is submitted to the main rules presented in the developer guide.

4.2.2 TelApy module

The Python part is developed with the python convention "PEP 8". The aim of PEP 8 guidelines used in the TelApy module is to improve the readability of code and make it consistent across the wide spectrum of Python code. A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.
The PEP 8 convention coding can be easily checked using the Pylint code analyser `https://www.pylint.org`. Pylint is a tool that checks for errors in Python code, tries to enforce a coding standard and looks for code smells. It can also look for certain type errors, it can recommend suggestions about how particular blocks can be refactored and can offer you details about the code's complexity. Pylint will display a number of messages as it analyzes the code and it can also be used for displaying some statistics about the number of warnings and errors found in different files. The messages are classified under various categories such as errors and warnings.

5. Outlooks

The work on TELEMAC-MASCARET SYSTEM interoperability is always in progress. In fact, all modules have not yet API structure. This will be completed in future with at least one new module available per new TELEMAC-MASCARET SYSTEM version. Moreover, for the existing API, some works must be done in order to obtain a full API:

- Switch the instance use, normally API modules should be pointing on the instance and not the other way around. This will allow multiple instances of considered module to run at the same time.
- Add more variables (the one from the steering file for example)
- Remove all uses of save and common in the code they can induce memory leaks.

The outlooks in longer term vision is to remove "user fortran" because all modifications can be done directly via the APIs. Also, the coupling between modules using the APIs will be rewritted. And finally, more tutorials will be added in notebook format in order to facilitate the user life with APIs.

[1] PETERSON P. F2py: a tool for connecting fortran and python programs. *International Journal of Computational Science and Engineering*, 4(4):296–305, january 2009.