# Mascaret-Telemac2D Longitudinal coupling - User Guide

Andrea Piacentini, Marie-Pierre Daou, Sophie Ricci, Nicole Goutal

# Mascaret-Telemac2D longitudinal coupling

## User Manual

**Version trunk**
**November 25, 2020**

# Contents

# 1. Introduction

Several research works have recently been published on the longitudinal coupling of hydraulic solvers from the Open Telemac-Mascaret suite. [7], [8], and [3] can be cited for example. The integrated Mascaret - Telemac2D longitudinal coupling described in this document is based on Sébastien Barthélémy's post-doc at CERFACS, under the direction of S. Ricci, N. Goutal and É. Le Pape, who implemented a fully working prototype for coupling the 1D and 2D free surface hydraulics models MASCARET and TELEMAC interfaced at their boundary conditions, denoted in the following as longitudinal coupling [1, 2, 6]. Fig. 1.1 illustrates the 1D sub-domains (in black) coupled to the 2D domains (in blue) at the interfaces (in red).

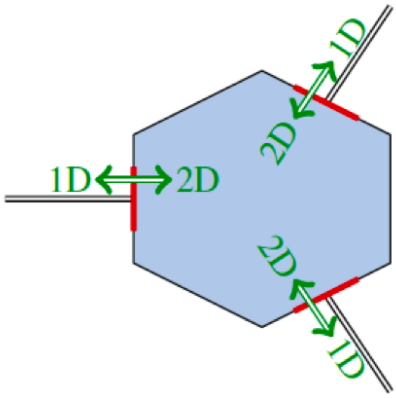



Figure 1.1: Schematic view of longitudinal coupling

The original prototype, based on the article by [4], has been implemented by the explicit coupling of two separate F90 codes via the OpenPALM coupler developed at CERFACS [5]. The exchanged data was accessed and modified via the F90 generic API's of the version 7 of the TELEMAC-MASCARET code suite.
The prototype has been tuned and tested on a single real applicative case where the confluence of the Nive and the Adour rivers in Bayonne is modeled with TELEMAC-2D and the upstream Nive and Adour and the downstream Adour stretches are modeled with MASCARET. For this case study, three 1D sub-domains are considered: two 1D sud-domains represent the Nive and Adour rivers (respectively Adour upstream and Nive upstream) that gather in the 2D domain, then the Adour river is represented by a downstream 1D model (Adour downstream) as illustrated in Fig. 1.2.

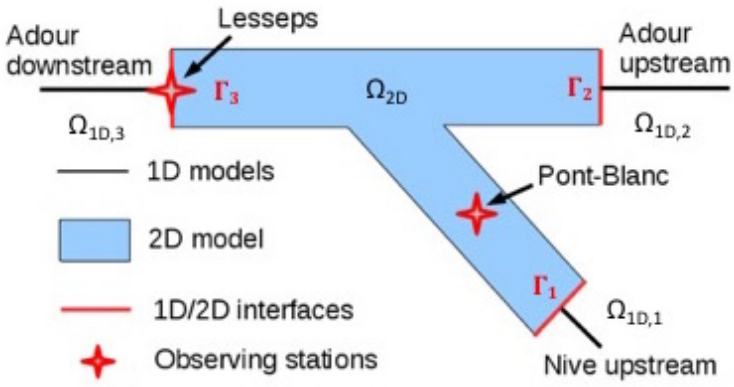



Figure 1.2: Schematic representation of the relative placement of the 1D models compared to the 2D model and the coupling interfaces for the Adour catchment.

Lately, the code has been rewritten, keeping the same algorithmic approach, but relying entirely on the python API's and directly coupling the 1D and 2D components via MPI without any third parties coupler. An academic test case of hydrodynamics in a 1D rectangular channel was also added for validation with respect to the analytical solution.
At the current state of development, the coupling tools described in this document allow for the implementation of longitudinal couplings for configurations similar to the one tested in the prototype: a single 2D basin with upstream and downstream 1D coupled stretches. This means that the coupling can handle several 1D sub-domains and a single 2D sub-domain.
Note that in such configuration a coupled 1D stretch can only have a single interface with the 2D region. This limitation could be overcome in order to allow for 2D-1D-2D configurations: the formalism for the description of the coupling geometry is already well adapted to complex configurations.

In the following, we'll quickly recall the algorithmic principles, we'll provide a quick start installation guide (with details in appendix) and user guide and we'll explain thoroughly the syntax of the python dictionaries (optionaly stored in json files) used to describe the coupling configuration and the current run definition.

# 2. The algorithm

The longitudinal coupling is dealt with via the Schwarz algorithm in its multiplicative or additive variants. It will be illustrated here on the case of a 1D stretch upstream of a 2D region as shown in Fig. 2.1. The flow in the 1D region is assumed to be sub-critical (for further considerations on the most adapted interface conditions accordingly to the flow regime, the reader can refer to [Goutal. HDR]).

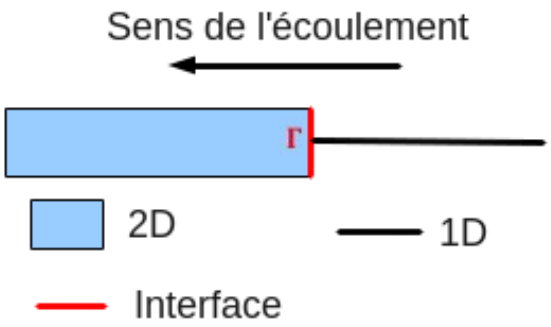



Figure 2.1: Schematic representation of the relative placement of the 1D models compared to the 2D model on a simple 1D-2D configuration.

The Schwarz method is an iterative algorithm illustrated in Fig. 2.2: the time range of the coupling event is split into a sequence of time intervals, as in $[t_1, t_2]$. The iterations in a given interval are represented by the red loop. Over each interval, models (code A and code B) are integrated in time, with their respective time steps. It should be noted that one code could be faster then the other, and thus would have to wait before exchanging data. Once both models have computed their hydraulic states, they exchange (via communications denoted by com) some hydraulic state variables – here the free surface height and the discharge values – at the boundary of the 1D and 2D sub-domains, until a convergence criteria is reached. In case of a sub-critical flow, the upstream 1D model provides the discharge boundary condition value to the 2D model which, in turns, provides the 1D with the free surface height condition. On the same time interval, the models are therefore restarted and integrated with the new boundary conditions. This is referred to as a *coupling step*. Once the convergence (or the maximum number of allowed iterates) is reached, the models are integrated on the following time interval, for the next coupling step. The coupling steps are chained to cover the entire coupling event.

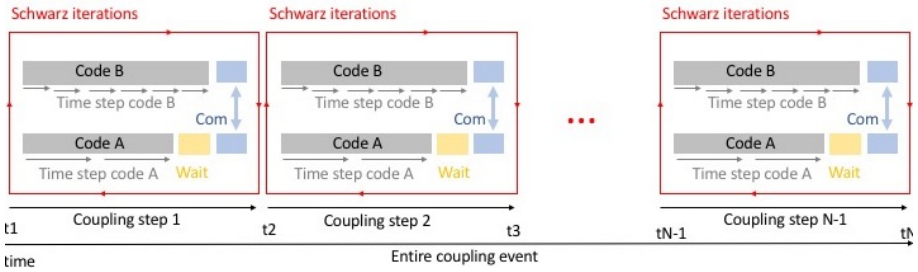



Figure 2.2: Schema

The Schwarz method is derived in a multiplicative or additive variant depending on how time is dealt with during the iterative process at the coupling interfaces $\Gamma$.
In the *multiplicative* variant, the models are integrated one at a time in a sequence, starting in the present study with the 1D model. As a consequence, only the 1D model restarts from initially prescribed boundary conditions at the coupling interfaces or from conditions inherited from the previous time interval as in Eq. 2.1. The 2D model always runs after at least a 1D integration, therefore the boundary conditions at the coupling interfaces are always provided by the 1D model as in Eq. 2.2.

$$\begin{cases} \mathscr{L}_1(S,Q)^k = 0 \text{ on } \Omega_{1D} \times [t_1;t_2] \\ S_{1D}^k = A_{21}(\bar{h}_{2D}^{k-1}) \text{ on } \Gamma \times [t_1;t_2] \end{cases} \tag{2.1}$$

$$\begin{cases} \mathscr{L}_2(h,u,v)^k = 0 \text{ on } \Omega_{2D} \times [t_1;t_2] \\ (u,v)_{2D}^k = B_{12}(Q_{1D}^k) \text{ on } \Gamma \times [t_1;t_2] \end{cases} \tag{2.2}$$

$$\begin{cases} \mathscr{L}_2(h,u,v)^k = 0 \text{ on } \Omega_{2D} \times [t_1;t_2] \\ (u,v)_{2D}^k = B_{12}(Q_{1D}^{k-1}) \text{ on } \Gamma \times [t_1;t_2] \end{cases} \tag{2.3}$$

In the *additive* variant, the models are integrated simultaneously: both restart from initially prescribed boundary conditions at the coupling interfaces or from conditions inherited from the previous time interval as in Eq. 2.1 and Eq. 2.3. The additive variant is available as an option because it has the advantage that the models can run simultaneously in parallel, but it has to be noted that in some situations the additive formulation has a slower convergence than the multiplicative formulation, and, even if the models run in parallel, the overall coupling time could be higher than with the multiplicative variant. The default choice is therefore the multiplicative Schwarz method.
The algorithm is also described by the way boundary conditions at the interface are inherited from the previous to the current coupling step as illustrated in Fig. 2.3. A possible choice is to reuse the same conditions as in the *previous* time interval. If the time interval spans more than one model time-step, the whole time series is translated to the next interval. An alternative is to make the last value of the time series *persist* as a constant on the following interval. A further distinction, especially if the iterates are stopped before satisfactory convergence, is to use the last value at the boundary as it has been provided by the other model before the last integration or as it has been updated by the model itself after the last integration. The default choice is the persistence of the last value provided by the other model.
In order to cope with the oscillation in values during iterates before convergence, the average of the values (or time series) from the last two iterates is used instead of the last value itself. This approximation could be avoided if the algorithm is iterated till full convergence.

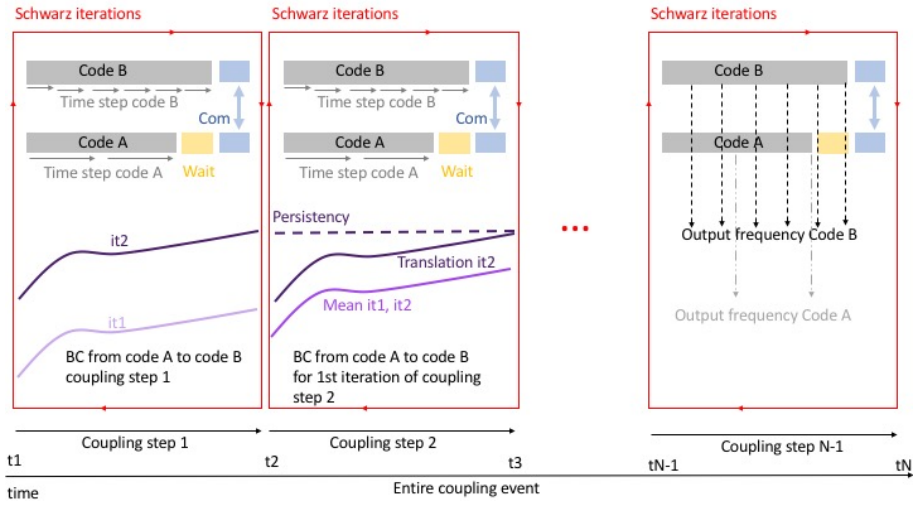



Figure 2.3: Schematic representation of the coupling with coupling step sequneces.

# 3. Software requirements and installation

The longitudinal coupling has been implemented on top of the version 8.1 of Open TELEMAC-MASCARET, provided with the recently updated python3 API's. In addition, some specific python3 procedures have been added for the coupling.

In order to have a coherent complete python3 environment with all the needed packages, we suggest creating a dedicated environment either via a python3 venv – as illustrated in appendix A – or miniconda – as illustrated in appendix B.

The Open TELEMAC-MASCARET code has to be compiled with the "api" target activated. Since the coupling needs the message passing MPI library, choose a compilation environment with its own compatible MPI library, even if TELEMAC2D will be used in its sequential declination. Notice and remember that the python environment will include `mpi4py` and some caution has to be paid to be sure that the underlying MPI library is the same as that for the model. More details are provided in sections A.1 and B.1 of the appendix on the venv and conda installations.

Once the environment has been created and loaded, you can proceed to the compilation of the models with the python3 utility `scripts/python3/compile_telemac.py`. Please refer to the Open TELEMAC-MASCARET documentation for details.
Note that the configuration for the longitudinal coupling in `configs` must contain the following informations (examples here refer to gfortran and OpenMPI4, please modify the syntax accordingly to your compiler)

```
options:     mpi api
f2py_name: f2py
pyd_fcompiler: gfortran
sfx_lib:    .so
cmd_obj:    mpif90 -c -cpp -fPIC [...]
cmd_lib:    mpif90 -fPIC -shared [...]
cmd_exe:    mpif90 -fPIC [...]
cmd_obj_c: mpicc -c -fPIC <srcName> -o <objName>
cmd_obj_partel: mpif90 -c -cpp -fPIC [...]
mpi_cmdexec: mpirun [...] -np <ncsize> <exename>
```

Finally, make sure that the python modules are made accessible for import, by adding the

`$HOMETEL/scripts/python3` directory to the `PYTHONPATH` environment variable.

# 4. Naming conventions

In order to be consistent throughout the document, we point out in this chapter some naming conventions and keywords we'll use in the following.

## 4.1 Time reference and cycling

A coupled simulation is a time evolving prognostic model used to simulate a multi-physics phenomenon. The simulation implies different specialized models that cooperate to represent a complex system. All models must reference the same physical simulated situation, therefore it is mandatory to define a common time reference.
Furthermore a complex system on a long event could not be simulated in a single integration, neither it is safe to perform such integration without restarting capabilities. For this reason the coupled simulation is implemented by cycling runs as illustrated in Fig. 4.1 and detailed in the following section.

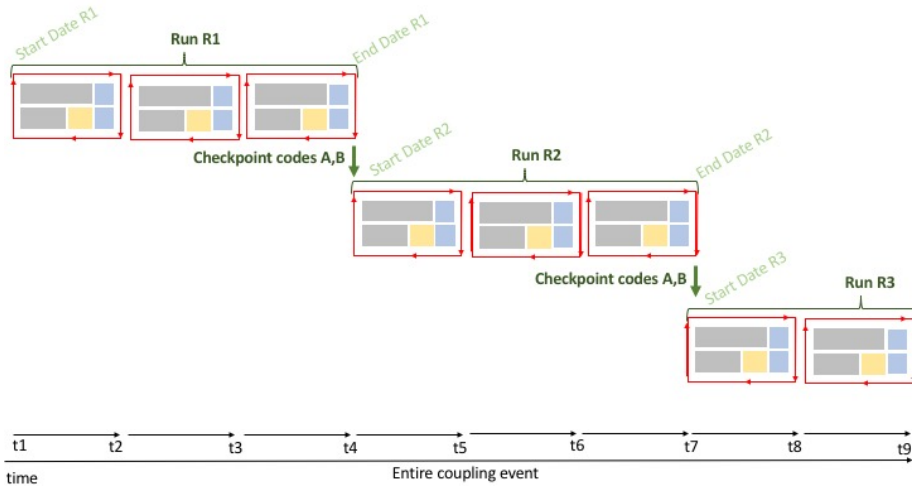



Figure 4.1: Schematic representation of coupling with sequential runs

All time instances are referenced as **elapsed seconds** from a common **reference date**. The elapsed seconds are the common unit, and as such, are used in files and directories names. Yet, in the configuration files, human readable dates are used and automatically converted into

elapsed seconds by the coupling scripts. The reference date is the initial instant of an **event** and the simulation cannot start before that date.
A **run** denotes a coupled simulation over a sub period of the event. It is defined by a **start date** (which should be greater or equal than the reference date) and an **end date** (which should be strictly greater than the start date). Restarting capabilities allow for chaining a subsequent run starting from the previous end date. Yet a cold start is possible, provided that initial conditions for each model are available at the start date. A *run* is either **restarted from a file containing a previously saved information**, or if *initialized from provided conditions at the beginning of the event*. Each run can be subdivided into a sequence of chained shorter **executions** of a prescribed **duration**, for the purpose of reducing the resources needed for every single parallel application and of generating frequent **checkpoints** to increase fault resiliency.

For illustration purposes, the time line in Fig. 4.1 is detailed here:
A 1-month *event* is considered, starting at midnight of July the 1st 2019. The *reference date* is therefore `1/7/2019 00:00:00`.
In order simulate the 6-hour interval from noon to 7 pm of the same day, the *run* is defined with *start date* `1/7/2019 12:00:00` and *end date* `1/7/2019 18:00:00`, provided that at least the initial conditions for all the models are available at the start date, for instance $t_1$ to $t_7$ in Fig. 4.1.
The run could also be slit the run into 2 checkpointed executions of 3 hours each with *execution duration* of `03:00:00` between $t_1$ and $t_4$ and then between $t_4$ and $t_7$.
In order to pursue the simulation for 3 more hours, a *new run* is defined with *start date* `1/7/2019 18:00:00` and *end date* `1/7/2019 21:00:00` strating the restart produced by the previous run at $t_7$.

## 4.2 Coupling definition

The main algorithmic concepts are described in chapter 2. We draw here some correspondences between the algorithm and naming used in the configuration dictionaries or files, relying on Fig. 2.2, Fig. 2.3 and Fig. 4.1.
The toplevel choice, referencing to the multiplicative and additive variants of the Schwarz formulation, is the **coupling method**.

Successively, a coupled simulation is realised as a time sequence of **coupling steps**. The step is the *coupling time interval* mentioned in chapter 2, i.e. the frequency at which the boundary conditions are recomputed and exchanged till convergence. A short coupling step makes the convergence easier, a longer time step reduces the overall number of repetitions but could require more iterates per repetition. In order to control the execution time and prevent hanging execution in case of divergence, the iterative process is bounded to a prescribed **maximum number of iterates**.

## 4.3 1D models description

The coupling framework relies on a hydraulic network that features at least one 1D stretch modeled with Mascaret. Each 1D domain is identified with a unique **identifier** string.

The **time step** is indicated for each model; it must divide exactly the *coupling step*. Note that, in the current version of the coupling scripts, all 1D models share the same time step while the 2D model has its own time step. The need for distinct 1D model time steops will be further

evaluated.

The paths for Mascaret input and output files should be given, as detailled in section 5.1. The information is stored in a disctionary (optionally in a json file) for each 1D model.

## 4.4 2D model description

Whether the representation of non contiguous 2D domains should be treated with separate instances of Telemac or with a single instance with a non contiguous computational domain is still open for discussion. Most likely, a single instance would be used, nevertheless, in order to maintain the two options, the 2D model is also identified with a unique **identifier** string. Similarly to the 1D model, the 2D model **time step** must be prescribed and divide exactly the *coupling step*.
The **output frequency**, at which the model writes its results in the output files, and the **output location**, for nodes of interest should be specified.
Path for Telemac input and output files should also be given, as detailled in section 5.2, in the form of a dictionnary (optionnaly in a json file).

## 4.5 Interfaces description

Interfaces description explains how the models are connected. For the longitudinal coupling, there is no overlapping between 1D and 2D domains. The coupling interfaces are located at the extremities of 1D stretches and free boundaries of the 2D domain as shown in Fig. 1.2.
An **interface** $\Gamma$ is fully described by the **identifier of the 1D model**, the **identifier of the 1D stretch free extremity** coupled to the 2D model, the **identifier of the 2D model**, the **identifier of the 2D liquid boundary** coupled to the 1D model, and by the **relative position of the 1D model** w.r.t. the 2D model (upstream or downstream).

The iterative coupling process described in Sect. 2 is stopped when specific **convergence criteria** at the interfaces are met. For each interface, the coupled variables and the convergence threshold should be specified.

# 5. Inputs for a coupled application

A coupled application is based on a set of MASCARET and TELEMAC2D instances along with the description of the simulation (Section 4.1) and the coupling configuration.

The `LongCplDriver` class from `telapy.coupling.long_cpl_driver` drives a coupled application. It receives its inputs via dictionaries. Arguments for initialisation of the `LongCplDriver` class are:

- The directory path `caseo_path` for toplevel of the coupling filetree, naled `CPLROOT in the following`,
- A dictionnary `coupling_def` for coupling information (Sect. 5.3),
- A dictionnary `model_configs` for 1D and 2D models information, espacially path for input files for the 1D and 2D submodels (Sect. 5.1 and Sect. 5.2),
- A bolean `keep_exec` for saving of execution directory `CPLROOT/EXEC`. Note that this path is the reference for input and output relative paths.

Arguments for call of the `LongCplDriver` class stand in a dictionnary that gathers information on the simulation (Sect. 5.4)

There are two ways to run a coupling. The first way consists in invoking the driver from a user python procedure that provides the previously mentionned dictionaries, initialize and call the `LongCplDriver` class. The second way relies on a provided launcher `run_cpl.py long` as explained in Chapt. 6. In the latter case, the input information is read from configuration files gathered as a dictionnary in a single python file `coupling_description.py` or in a set of json files with similar dictionnary structure. The syntax for the arguments of initialisation and call methods of `LongCplDriver`, as well as how this information is stored in in python or json as described in the following subsections.

The actual input files for 1D and 2D submodels should be available in a remote directory, on a file system accessible for linking from inside `CPLROOT/EXEC`. The output file location `Results` should be specified (Chap. 6), possibly not under the temporary `CPLROOT/EXEC` directory.

## 5.1 MASCARET

### 5.1.1 Description of MASCARET inputs

A Mascaret 1D instance is described by an xml parameter file with extension `.xcas`. The geometry is described in a geometry file with extension `.geo`. The initial condition for restart is read from an ascii file with extension `.lig`. The time varying boundary conditions are described in several files (one per condition) with extension `.loi` or `.law`. The data describing MASCARET input files come as the third argument of `LongCplDriver` class. Three posibilities are handled to provide this information to `LongCplDriver` class: as a dictionnary, in a python file or in a json file. The checkpointed boundary conditions at the coupling interfaces are automatically stored in a json file named `bc1D_restart_`; indexed by the model identifier and seconds since the *reference date* (e.g. `bc1D_restart_AAM_57600.json`). These files are not provided by the user and are created during the coupling cycles. The data describing MASCARET input files come as an element of the third argument of `LongCplDriver` class given in the `model_configs` dictionnary. Three posibilities are handled to provide this information to `LongCplDriver` class: as a dictionnary, in a python file or in a json file.

### 5.1.2 Description of MASCARET input data as a dictionary for `LongCplDriver`

The third positional argument of the initialization of `LongCplDriver` is a dictionnary named `models_configs` that describes the configurations for the 1D and 2D submodel distinguished by their identier as "`config_`*identifier*". For instance, if the identifier for the "Channel 1D" model is `ch1d`, its corresponding key in the dictionnary is `config_ch1d`

Under each of the 1D submodel key, the paths for MASCARET input files are listed in a dictionnary under the single key **`files`**:

**`xcas`** string with the path of the xml parameter file

**`geo`** string with the path of the geometry file

**`res`** string with the name (and only the name as the path is set at run time) of the results file

**`listing`** string with the name (and only the name, because the path is determined at run time) of the listing file

**`damocle`** only kept for compatibility. Its value is a string but the content is neglected

**`lig`** string with the path of the initial condition file. Note that during the coupling process this can be overidden in order to reuse restart.

**`loi`** list of strings with the path of the time dependent boundary condition files.

As an example, if the dictionary to be passed as third argument to `LongCplDriver` is stored in `model_configs`, the section for the "Channel 1D" model will be as follows:

```
models_configs = {
  "config_ch1d" : {
     "files": {

        "xcas": "../Input_Data/1d_stretch/ParametresMascaret.xcas",
        "geo": "../Input_Data/1d_stretch/Channel1d.geo",
        "res": "ResultatsOpthyca_ch1d.opt",
        "listing": "ResultatsListing_ch1d.lis",
```

```
        "damocle": "listing.damoc",
        "lig": "../Input_Data/1d_stretch/WaterLine_ch1d.lig",
        "loi": [
                "../Input_Data/1d_stretch/1d_up.loi",
                "../Input_Data/1d_stretch/1d_down.loi"
               ]
    }
  },
  [...]
```

### 5.1.3 Description of MASCARET input data as an entry in `coupling_description.py`

If the coupling is started by the standalone launcher `run_cpl.py long`, the coupling description can be stored in a python file with compulsory name `coupling_description.py`. In this file, for each 1D model instance, a dictionary must be defined and given the mandatory name
`config_`*identifier*
The dictionaries content is the same as in the dictionary argument described in the previous section. For instance, the dictionary for the "Channel 1D" model is:

```
config_ch1d = {
  "files": {

     "xcas": "../Input_Data/1d_stretch/ParametresMascaret.xcas",
     "geo": "../Input_Data/1d_stretch/Channel1d.geo",
     "res": "ResultatsOpthyca_ch1d.opt",
     "listing": "ResultatsListing_ch1d.lis",
     "damocle": "listing.damoc",
     "lig": "../Input_Data/1d_stretch/WaterLine_ch1d.lig",
     "loi": [
             "../Input_Data/1d_stretch/1d_up.loi",
             "../Input_Data/1d_stretch/1d_down.loi"
            ]
          }
}
```

### 5.1.4 Description of MASCARET input data as a json configuration file

Another option, if the coupling is started by the standalone launcher `run_cpl.py long`, is to provide a json configuration file for each model instance, named
`config_`*identifier*`.json`.
This json file contains a dictionary with a single key `files`. For instance, the json configuration file named after the identifier of the "Channel 1D" submodel as `config_ch1d.json`, contains:

```
{
 "files": {
    "xcas": "../Input_Data/1d_stretch/ParametresMascaret.xcas",
    "geo": "../Input_Data/1d_stretch/Channel1d.geo",
    "res": "ResultatsOpthyca_ch1d.opt",
    "listing": "ResultatsListing_ch1d.lis",
    "damocle": "listing.damoc",
```

```
    "lig": "../Input_Data/1d_stretch/WaterLine_ch1d.lig",
    "loi": [
            "../Input_Data/1d_stretch/1d_up.loi",
            "../Input_Data/1d_stretch/1d_down.loi"
           ]
    }
}
```

## 5.2 TELEMAC2D

### 5.2.1 Description of TELEMAC2D inputs

A Telemac 2D instance is described by an ascii parameter file in the *damocles* format, usually called the *cas* (french for case) file. Its extension is usually (but not necessarily) `.cas`. The geometry and the physical characteristics of the domain are described in a binary file in the *selafin* format, with extension `.slf`. If the model is restarted, the initial condition has to be stored in another selafin file with extension `.slf`. The boudary conditions settings are given for the peripheral grid cells in an ascii file with extension `.cli`. When needed, the time varying boundary conditions are stored in a separate ascii file with extension `.liq`

For restarted coupling, the checkpointed boundary conditions at the coupling interfaces are automatically stored in a json file named adfter the model identifier and time in seconds since the *reference date* as `bc2D_restart_` (for instance `bc2D_restart_Bayonne_57600.json`). These files are not provided by the user and are created during the executions cycling.

#### An important remark about restart file names

In order to allow for the time cycling described in section 4.1, the names of the initial condition files for the 2D model must respect the mandatory naming convention
`WaterLineInit_`*seconds*`.slf`
where the seconds are computed w.r.t. the common reference date. The coupling driver will link the currently selected initial condition file to the constant name
`WaterLineInit_in.slf`
before transferring it to the execution directory as `T2DPRE`. It is therefore recommended to describe the restart file in the `.cas` description by
`FICHIER DU CALCUL PRECEDENT = WaterLineInit_in.slf`
or
`PREVIOUS COMPUTATION FILE = WaterLineInit_in.slf`

### 5.2.2 Description of TELEMAC2D inputs as a dictionary for `LongCplDriver`

The third positional argument of the initialization of `LongCplDriver` is a dictionnary named `models_configs` that describes the configurations for the 1D and 2D submodel distinguished by their identier as `"config_`*identifier*`"`. For instance, if the identifier for the "Channel 2D" model is `ch2d`, its corresponding key in the dictionnary is `config_ch2d`.

Under this key is a dictionnary that gathers the following keys and information:

**`cas`** string with the path of the case parameter file,

**`config_file`** Optional: string with the path of the systel compilation configuration file. Its default is the content of the `SYSTELCFG` environment variable. The same environment variable can be referenced in the dictionary (especially in the json files) by the conventional string `"${SYSTELCFG}"`,

**config_option** Optional: string with the identifier of the compilation configuration of the Telemac system. Its default is the content of the USETELCFG environment variable. The same environment variable can be referenced in the dictionary (especially in the json files) by the conventional string "${USETELCFG}"

As an example, if the dictionary to be passed as third argument to LongCplDriver is stored in model_configs, the section for the "Channel 2D" model is:

```
models_configs = {
  [...]
  "config_ch2d" : {
     "files": {
        "cas": "../Input_Data/2d_rect/T2DCAS",
        "config_file": "${SYSTELCFG}",
        "config_option": "${USETELCFG}"
     }
  }
}
```

### 5.2.3 Description of TELEMAC2D inputs as an entry in coupling_description.py

If the coupling is started by the standalone launcher run_cpl.py long, the coupling description can be stored in a python file with compulsory name coupling_description.py. In this file, for the 2D model, a dictionary must be defined and given the mandatory name config_*identifier*.
The dictionaries content is the same as in the dictionary argument described in the previous section. For instance, the dictionary for the "Channel 2D" model is:

```
config_ch2d = {
  "files": {
     "cas": "../Input_Data/2d_rect/T2DCAS",
      "config_file": "${SYSTELCFG}",
      "config_option": "${USETELCFG}"
  }
}
```

### 5.2.4 Description of TELEMAC2D inputs as a json configuration file

Another option, if the coupling is started by the standalone launcher run_cpl.py long, is to provide a json configuration file for the 2D model, named
config_*identifier*.json.
This json file contains a dictionary with a single key files. For instance, the json configuration file named after the identifier of the "Channel 2D" submodel as config_ch2d.json, contains:

```
{
 "files": {
    "cas" : "../Input_Data/2d_rect/T2DCAS",
    "config_file": "${SYSTELCFG}",
    "config_option": "${USETELCFG}"
    }
}
```

## 5.3 The Coupling Definition

The second argument for initialization of the coupling driver `LongCplDriver` class is the `coupling_def` dictionary that describes the coupling information.
If the coupling is started by the standalone launcher `run_cpl.py long`, the coupling description is stored in a python file with the compulsory name of `coupling_description.py` stored in `CPLROOT` and the coupling definition is stored in a dictionary with the mandatory name `coupling_def`.
As an alternative it can be stored in a json in `CPLROOT` with the mandatory name `CouplingDef.json`

In detail, the definition is organised in 4 sections, described in the following, and corresponding to 4 toplelvel keys in the the dictionary or separate json blocks in the json file:

- `"Coupling"`
- `"1D"`
- `"2D"`
- `"Interfaces"`

Each block contains in turn a dictionary. A full example is provided in Appendix C or D or E.1

### 5.3.1 The `Coupling` entry

The `Coupling` entry contains the following keys (*cf.* 4.2):

**`TimeStep`** the value is a real number with the coupling step in seconds

**`Method`** the value is a string. The choice is limited to `MultiplicativeSchwarz` or `AdditiveSchwarz`

**`MaxIter`** the value is an integer with the maximum number of iterates allowed per coupling step

**`CplStepRestart1D`** (optional, defaulting to `Persist2D`) the value is a string indicating how to reinitialize the 1D boundary conditions at coupling interfaces at each new coupling step

**`CplStepRestart2D`** (optional, defaulting to `Persist1D` and relevant only for the additive variant of the Schwarz method) the value is a string indicating how to reinitialize the 2D boundary conditions at coupling interfaces at each new coupling step

### 5.3.2 The `1D` entry

The `1D` entry contains as many dictionaries as instances of the 1D model. Each dictionary uses as key the identifier of the 1D model (*cf.* 4.3) and contains two keys:

**`TimeStep`** the value is a real number with the timestep for the model integration in seconds. Remember that, in the current version, the time step has to be the same for all the 1D models

**`OutputFreq`** the value is a real number with the results output frequency in seconds

### 5.3.3 The `2D` entry

The `2D` entry contains as many dictionaries as instances of the 2D model (only one possible as of today) and contains three keys:

**`TimeStep`** the value is a real number with the timestep for the model integration in seconds.

**`OutputFreq`** the value is a real number with the results output frequency in seconds

**`OutputSites`** the value is a list of integers with the node number at which the resulsts are output. Note that also for a single point the list syntax `[32939]` has to be respected

### 5.3.4 The `Interfaces` entry

The `Interfaces` entry is a list of dictionaries. Each dictionary describes a single coupling interface (*cf.* 4.5) and contains the following keys:

**`Id1D`** the value is a string with the identifier of the 1D model

**`IdExtr1D`** the value is a string with the identifier of the 1D model coupling free extremity as it is listed in the xml `.xcas` file

**`Condition1D`** the value is a string with the kind of boundary condition at the coupling interface for the 1D model. The choice is limited to `Discharge` or `WaterLevel`

**`Id2D`** the value is a string with the identifier of the 2D model

**`LiqBdry2D`** the value is an integer with the number of the 2D model coupling liquid boundary as it is listed in the damocles *cas* file

**`1DPosition`** the value is a string with the relative position of the 1D model w.r.t the 2D model. The choice is limited to `UpStream` or `DownStream`

**`ConvCriteria`** the value is a dictionary with as many keys as variables to be checked. For each key the value is a real number indicating the convergence threshold for that variable. Currently possible keys are `Height` or `Velocity`

## 5.4 The Current Run Configuration

In order to describe the simulation for the current run, the user must provide every run call of an instance of the coupling driver `LongCplDriver` with the information listed in Sect. 4.1. The first and only for a run call to an instance of `LongCplDriver`, the information has to be organised in a python dictionary, named `config_run`.

If the coupling is started by the standalone launcher `run_cpl.py long`, the coupling description is stored in a python file with the compulsory name of `coupling_description.py` stored in `CPLROOT` and the coupling definition is stored in a dictionary with the mandatory name `config_run`.
As an alternative it can be stored in a json in `CPLROOT` with the mandatory name `ConfigRun.json`. The configuration is organised in 2 sections, corresponding to two toplelvel keys in the dictionary or two separate json blocks in the json file:

- `"Run"`
- `"2D"`

Each of them contains in turn a dictionary. A full example is provided in Appendix C or D or E.2

### 5.4.1 The `Run` entry

The `Run` entry contains the following keys (*cf.* 4.1):

**`RefDate`** the value is a string in the format `[d]d/[m]m/yyyy hh:mm:ss` with the *event* reference date used as origin for the time counters in seconds

**`StartDate`** the value is a string in the format `[d]d/[m]m/yyyy hh:mm:ss` with the *start date* for the current *run*

**`EndDate`** the value is a string in the format `[d]d/[m]m/yyyy hh:mm:ss` with the *end date* for the current *run*

**`SingleExecDuration`** the value is a string in the format `[dd day[s]] hh:mm:ss` or `[ddd][hhh][mmm][sss]` with the duration of every *single execution* splitting the current *run*. Valid equivalent examples are `"36:00:00"` or `"1 day 12:00:00"` or `"36h"` or `"1d12h"`. Notice that this field is optional: if missing, the whole run is performed with just one execution.

**`RestartFromFile`** the value is the string `yes` or `no` to indicate if the run restarts from previously stored conditions (if available) or if it is a cold start

**`SaveCheckPoints`** the value is the `yes` or `no` string indicating if the restart information have to be stored between every single execution (yes) or only at the end of the run (no)

### 5.4.2 The `2D` entry

The `2D` entry contains as many dictionaries as instances of the 2D model (only one possible as of today) and contains two keys:

**`Parallel`** the value is the `yes` or `no` string indicating if we run the parallel (yes) or the sequential (no) telemac2d executable,

**`NbProc`** the value is an integer, relevant only if the run is parallel, with the number of processors for the 2D model.

# 6. Running a coupling

Once the input filetree has been prepared and the configuration dictionaries fully filled, running a coupled application is quite straightforward.

The coupling can be driven from a user application if the `LongCplDriver` class is imported from `telapy.coupling.long_cpl_driver`
In such case the coupling definition and model description information described in the previous sections is passed by arguments to the initialization of an instance of the `LongCplDriver` class:

```
[...]
from telapy.coupling.long_cpl_driver import LongCplDriver
[...]
the_coupling = LongCplDriver(case_path = ".",
                             coupling_def = user_coupling,
                             models_configs = models_configs)
```

A fourth optional boolean argument to the initialization, named `keep_exec` allows to conserve the `EXEC` directory after deletion of the driver instance or at the end of the procedure, for debugging purposes. The default is `keep_exec=False` and the `EXEC` directory is removed.

The coupling is actually launched by a direct call of the `LongCplDriver` instance which receives the current run configuration by argument, as in the following example:

```
[...]
the_coupling(config_run = driven_run)
```

More than one run can be chained in the same application, provided that the `config_run` dates are correctly updated as illustrated in the example in Sect. 8.1.3 and appendix D.

The coupling can also be started as a standalone application with the python procedure
`run_cpl.py long`
provided in the `scripts/python3/coupling` directory of the distributed sources.

If the procedure is started from the `CPLROOT` directory, where the `coupling_description.py` – or the `CouplingDef.json` and the `CurrentRun.json` – files are, it can be started without any argument.

If it is started from a generic directory, the `CPLROOT` has to be passed as an argument of the `--src` option.

As an example, for the `bayonne` test case from the distribution, let's say that `CPLROOT` is `$HOMETEL/examples/python3/telapy_coupling/bayonne` while `run_cpl.py long` is in `$HOMETEL/scripts/python3/coupling`
The run can be started from within `CPLROOT` by

```
python $HOMETEL/scripts/python3/run_cpl.py long
```

or from whatever directory by

```
python $HOMETEL/scripts/python3/coupling/run_cpl.py long \
       --src $HOMETEL/examples/python3/telapy_coupling/bayonne
```

A summary of the loaded configuration is output

```
+-----------------------------------------------------+
|------- LONGITUDINAL MASCARET TELEMAC2D COUPLING -------|
+-----------------------------------------------------+
|
|  Coupling Method: MultiplicativeSchwarz
|  Max It. nr.:     5
|  Coupling TStep:  8.0
|
|  Initial time:    1/7/2019 12:00:00
|  End time:        1/7/2019 16:00:00
|  Split Run every: 02:00:00 (2 exec[s])
|  Restarted run:   yes
|
|  1D models:       ['AAM', 'AAV', 'NA']
|  2D models:       ['Bayonne']
|
|  Interfaces:
|    AAM:limite2  (Discharge ) UpStream   of Bayonne:2
|    AAV:limite1  (WaterLevel) DownStream of Bayonne:3
|    NA:limite2   (Discharge ) UpStream   of Bayonne:1
+-----------------------------------------------------+
```

The executions start and a progress counter is displayed

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~   RUN THE COUPLED MODEL    ~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Running the coupled model between init time 43200 and end time 50400
[====================] 100%

Running the coupled model between init time 50400 and end time 57600
[===============     ]  75%
```

When the run ends the overall elapsed time is shown

```
My work is done. Coupled job lasted : 81.89077615737915 s
```

# 7. The Results directory

After every single execution completes, a subdirectory is created in `$CPLROOT/Results` (Note that the `Results` directory is created if not already present). The subdirectory is indexed by the initial time of the single execution: `COUPLING_FROM_XXXX`, where `XXXX` stands for the number of seconds from the event reference date to the execution initial time.

Each subdirectory contains:

- the standard output of the simulation redirected into a file named after the *cas* configuration file for Telemac2D with an additional suffix `.stdout`
- the single models results files in the format of Opthyca files for MASCARET and Selafin files for TELEMAC2D at the output frequencies entered in the respective sections of the `coupling_def` dictionary or `CouplingDef.json` file. (Note that only the results after the iterative process has converged or reached the maximum number of iterates are output in these files)
- the convergence criteria files (*cf.* sect. 7.1) in human readable ascii or in comma separated value (`.csv`) format to be imported in a spreadsheet

Some other output files are stored directly in the input directories because they are used for restarting the computation. The 1D waterlines, the 2D state, the boundary conditions for the 1D and the 2D models are stored at the end of every execution if the checkpointing is activated and, in any case, at the end of the run.

## 7.1 The `Convergence_criteria` files

At every coupling step, when the iterative process converges or reaches the maximum number of iterations, six convergence criteria are computed.
In the `Convergence_criteria.out` file (and in its comma separated value counterpart) at the end of every coupling step, the effective number of iterates is printed together with the six quantities evaluated on the 1D and on the 2D (spatial average) side of each interface.
The quantities are:

**Velocity** symbolized by `V` computed as $Q/(S1+S2)$

**Wetted Area** symbolized by `SM` computed as $S1+S2$ where $S1$ is the main channel wetted area and $S2$ is the floodplain wetted area

**Discharge** symbolized by Q

**Height** symbolized by H computed as $Z - ZREF$ i.e. the water level minus the reference level

**First invariant** symbolized by I computed as $Q/(S1+S2)+2\sqrt{g(Z-ZREF)}$

**Second invariant** symbolized by J computed as $Q/(S1+S2)-2\sqrt{g(Z-ZREF)}$

# 8. Examples and validation test cases

Four test cases are provided in order to illustrate the differents ways of describing and running a coupling. Moreover these test cases are run as part of the telemac validation procedure under the tag `coupling`:

```
python validate_telemac.py --tags coupling
```

Three tests are implemented a rectangular channel, partly modelled in 1D, partly in 2D as shown in Fig. 2.1. The last case represents the Adour catchment displayed in Fig. 1.2.

## 8.1 Rectangular channel test case

### 8.1.1 Description of the test case and experiment settings

The rectangular channel test case describes a 100 m wide channel modelled over 10 km with a bathymetry from 5 m upstream to 0 downstream and thus a uniform slope $I = 5.10^{-4}$.
From the Manning equation, in a permanent regime and with a homogeneous friction coefficient, the free surface at the equilibrium is parallel to the bathymetry. Imposing a friction coefficient of 30.6, an upstream inflow $Q = 1000m^3/s$, and a downstream boundary condition (BC) $Z = 5m$, the analytical solution is $h = 5m$ everywhere, with a resulting free surface $Z$ ranging from 10 m upstream to 5 m downstream.
The hydrodynamics of the channel is modelled with a 5km long 1D model upstream of a 2D model for the 5 downstream kilometers. There is thus a unique coupling 1D-2D interface between the 1D downstream BC and the 2D upstream liquid boundary.
In permanent regime, the 1D-2D coupled solution can be validated against the analytical solution.
We design three different experiment settings modifying the initial conditions (IC) or the upstream inflow boundary condition:

- R1: Constant Upstream and downstream BC, 1D and 2D IC already set to Manning equilibrium. The solution should remain identical to the IC, proving that the coupling does not degrade the permanent solution.
- R2: Constant Upstream and downstream BC, 1D IC to Manning equilibrium, 2D IC set to a horizontal free surface $Z = 8.5m$ at rest. The solution should converge towards the Manning equilibrium after a sufficiently long integration, proving that the coupling is able to reconstruct the global solution even with non coherent initial conditions and a misfit of 1 m on $Z$ and 2 m/s on $V$ at the coupling interface. Note that, since the downstream BC

condition limits $Z$ at the exit, this situation takes longer to reach the equilibrium than if the 2D initial condition were lower than the analytical value of 7.5 m.

- R3: Varying Upstream BC, 1D and 2D IC to Manning equilibrium. In this last case, the model starts from the Manning equilibrium, then the prescribed upstream inflow is perturbed for 15 minutes, doubling the equilibrium value, then dicreases to the original value. The Manning equilibrium is temporary lost, but eventually recovered, proving that perturbations are correctly propagated throughout the coupled models without spurious effects.

### 8.1.2 Test case directory

The test case files are in

- `examples/python3/telapy_coupling/channel_manning` for the equilibrium experimental setting (R1),
- `examples/python3/telapy_coupling/channel_ic` for the equilibrium experimental setting with perturbed initial condition (R2),
- `examples/python3/telapy_coupling/channel_bc` for non permanent experimental setting with perturbed upstream boundary condition (R3),

The 1D and 2D model input files are organized as follows :

- `/Input_Data/1d_stretch` for the 1D model MASCARET, containing the `.geo`, `.xcas`, `.loi` and `.lig` files for geometry, parameters, BC and IC respectively.
- `/Input_Data/2d_rect` for the 2D model TELEMAC, containing the `T2DCAS`, `T2DCLI`, `T2DDICO`, `GEO` and `WaterLineInit_0.slf` files for parameters, BC, parameters options, geometry and IC respectively. Note that for R1 and R3, the IC is prescribed as a field, read from a pre-calculated restart file that is provided here where the Manning equilibrium is installed in the 2D model. For R2, the IC is prescribed in `T2DCAS` as a uniform value.

### 8.1.3 Running the examples

**`channel_manning`**

The test case corresponding to configuration R1 can be run as a user driven application or as a standalone run with input provided in a single pyton file.

The user application driving the coupling is in the `driven_coupling.py` file. The launching procedure is:

```
cd $HOMETEL/examples/python3/telapy_coupling/channel_manning
python driven_coupling.py
```

Note that two consecutive runs are executed with two successive calls to the instance of `LongCplDriver` after updating the current run configuration dictionary.

As an alternative, the standalone run can be invoked from wathever directory as:

```
python $HOMETEL/scripts/python3/telapy/coupling/run_cpl.py long \
       --src $HOMETEL/examples/python3/telapy_coupling/channel_manning
```

or from inside the $HOMETEL/examples/python3/telapy_coupling/channel_manning simply as

```
python $HOMETEL/scripts/python3/telapy/coupling/run_cpl.py long
```

As such it is also part of the validation suite.

**channel_ic**
The test case corresponding to configuration R2 is run as a standalone run with input provided in a single pyton file.

The standalone run can be invoked from wathever directory as:

```
python $HOMETEL/scripts/python3/telapy/coupling/run_cpl.py long \
       --src $HOMETEL/examples/python3/telapy_coupling/channel_ic
```

or from inside the $HOMETEL/examples/python3/telapy_coupling/channel_ic simply as

```
python $HOMETEL/scripts/python3/telapy/coupling/run_cpl.py long \
```

As such it is also part of the validation suite.

**channel_bc**
The test case corresponding to configuration R3 is run as a standalone run with input provided in a single pyton file.

The standalone run can be invoked from wathever directory as:

```
python $HOMETEL/scripts/python3/telapy/coupling/run_cpl.py long \
       --src $HOMETEL/examples/python3/telapy_coupling/channel_bc
```

or from inside the $HOMETEL/examples/python3/telapy_coupling/channel_bc simply as

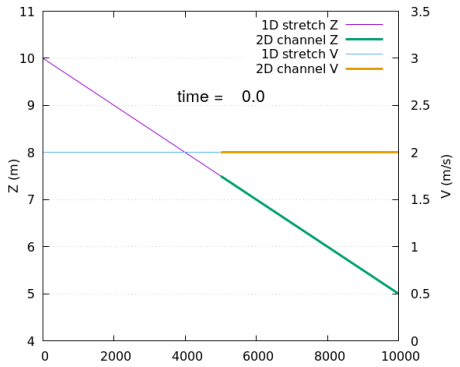
```
python $HOMETEL/scripts/python3/telapy/coupling/run_cpl.py long
```

As such it is also part of the validation suite.

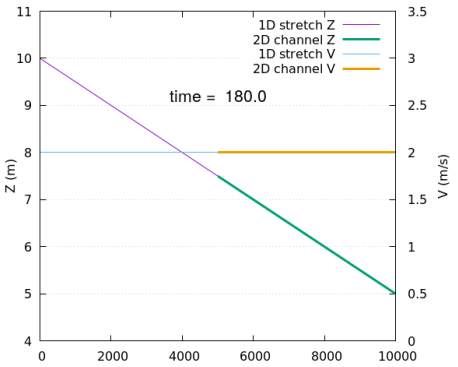
### 8.1.4 Checking the results

Note that in the coupling_def dictionary or in the CouplingDef.json file, you can specify a list of 2D element numbers under the "OutputSites" key where the hydraulic variable at the center of the channel are output into a text file (time, curvilinear abscissa, water level, velocity for 2D outputs) at the temporal frequency specified as well in the coupling_def dictionary or in the CouplingDef.json file. Same outputs are generated for each 1D node (in .xcas file, key stockage, option 1, otherwise specify sites) for time, curvilinear abscissa, water level, discharge and velocity at spatial frequency specified in .xcas file and temporal frequency specified in the coupling_def dictionary or in the CouplingDef.json file.

Use the animate.gp, debanimate.gp and velanimate.gp scripts provided in $HOMETEL/examples/python3/telapy_coupling/plotting for visualisation and animation. For that purpose, copy the .gp scripts in the results directory containing the .plt files and run the command gnuplot animate.gp or similar for the other scripts.
Some output examples are in Figs. 8.1, 8.2, 8.3.

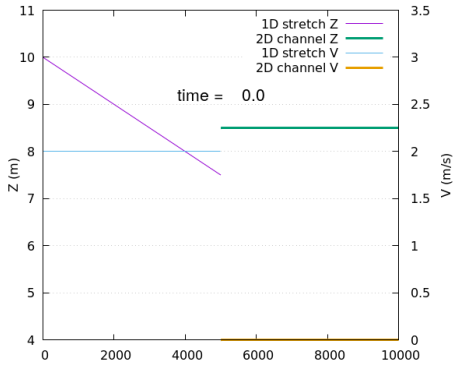



(a) coupled solution for R1 at t=0s

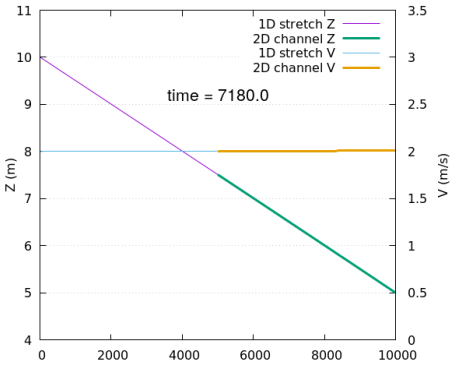



(b) coupled solution for R1 at t=180s

Figure 8.1: 1D-2D coupled solution for R1

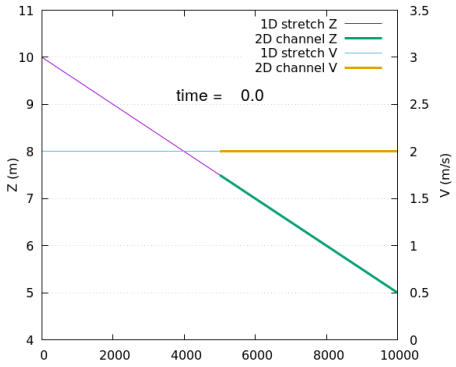



(a) coupled solution for R2 at t=0s

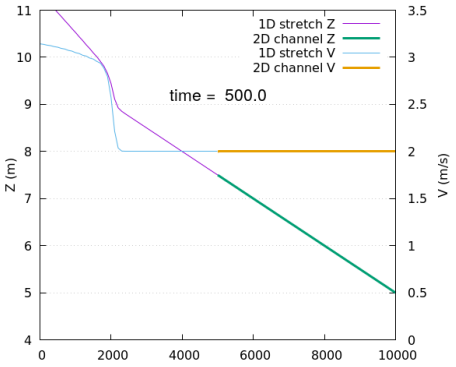



(b) coupled solution for R2 at t=7180s

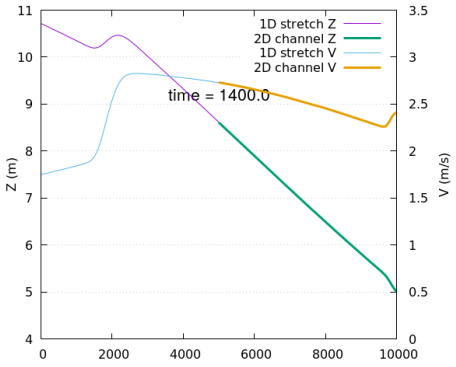
Figure 8.2: 1D-2D coupled solution for R2



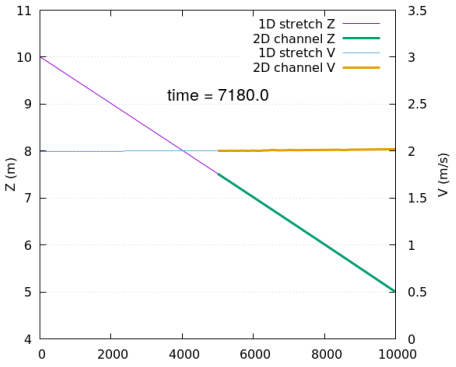

(a) coupled solution for R3 at t=0s




(b) coupled solution for R3 at t=500s




(c) coupled solution for R3 at t=1400s




(d) coupled solution for R3 at t=7180s

Figure 8.3: 1D-2D coupled solution for R3

## 8.2 Adour catchment test case

### 8.2.1 Description of the test case and experiment settings

The modelled domain is the confluence of the Nive and the Adour rivers in Bayonne: the confluence is modeled with TELEMAC-2D and the upstream Nive and Adour and the downstream Adour stretches are modeled with MASCARET. For this case study, three 1D sub-domains are considered: two 1D sud-domains represent the Nive and Adour rivers (respectively Adour upstream and Nive upstream) that gather in the 2D domain, then the Adour river is represented by a downstream 1D model (Adour downstream) as illustrated in Fig. 1.2.

### 8.2.2 Test case directory

The test case files are in `examples/python3/telapy_coupling/channel_bayonne`.

The coupling is described as a user driven application and the `driven_coupling.py` procedure is provided in the test case directory.
Notice that the procedure checks in the telemac configuration file if the 2D model has been compiled with the `mpi` option and trigger the `"Parallel"` option for `"Bayonne"` in the `"2D"` section `config_run` dictionary accordingly.

The 1D and 2D model input files are organized as follows :

- `/Input_Data/RUN_Adour_amont` for the 1D model MASCARET on the upstream Adour strecth, containing, in turn, the `DonneesStat` directory containing the `.geo`, `.xcas`, and `.lig` files for geometry, parameters and initial condition respectively and the `DonneesDyn` directory containing the `.loi` files for the boundary conditions.
- `/Input_Data/RUN_Adour_aval` for the 1D model MASCARET on the downstream Adour strecth, containing, in turn, the `DonneesStat` directory containing the `.geo`, `.xcas`, and `.lig` files for geometry, parameters and initial condition respectively and the `DonneesDyn` directory containing the `.loi` files for the boundary conditions.
- `/Input_Data/RUN_Nive_amont` for the 1D model MASCARET on the upstream Nive strecth, containing, in turn, the `DonneesStat` directory containing the `.geo`, `.xcas`, and `.lig` files for geometry, parameters and initial condition respectively and the `DonneesDyn` directory containing the `.loi` files for the boundary conditions.
- `/Input_Data/Bayonne` for the 2D model TELEMAC, containing the `TelemacBayonne.cas`, `TelemacBayonne_TypeDeConditionsLimites.cli`, `TelemacBayonne_ConditionsLimites.liq`, `TelemacBayonne_Geometrie_Ks.slf` and `WaterLineInit_43200.slf` files for parameters, BC, parameters options, geometry and IC respectively.

### 8.2.3 Running the example

The test case can be run as a user driven application.

The user application driving the coupling is in the `driven_coupling.py` file. The launching procedure is:

```
cd $HOMETEL/examples/python3/telapy_coupling/channel_bayonne
python driven_coupling.py
```

As such it is also part of the validation suite.

# Appendices

# A. Installation of the python venv

You need to create a fresh virtual environment based on `python3` and to activate it. You have to provide a name for the dedicated environment: in the following instructions we arbitrarily chose `cpl_py3`. The `VENVROOT` environment variable has to be defined beforehands and point to the location on your machine where the site packages will be installed. Make sure that there is enough room avalaible and, in case of clusters with different computational and front-end nodes, that the location can be accessed from both kinds of nodes.

```
xxx>python3 -m venv $VENVROOT/cpl_py3

xxx>source $VENVROOT/cpl_py3/bin/activate
```

If the virtual evironment has been correctly charged, the prompt will be modified and display the active venv name. You can check that the default `python` command now points to version 3.

```
(cpl_py3) xxx>python --version
Python 3.6.5
(cpl_py3) xxx>
```

In order to access the most up to date view of the python packages we suggest to upgrade the `pip` package manager itself to its latest version

```
(cpl_py3) xxx>pip install --upgrade pip
```

Since `pip` takes care of the dependencies, by installing a few packages we have a complete environment that can be used for the coupling, the data assimilation via Smurf and the Uncertainty Quantification studies.

```
(cpl_py3) xxx>pip install ot-batman
(cpl_py3) xxx>pip install shapely
(cpl_py3) xxx>pip install pyyaml
```

An optional package which can be useful if the user needs to determine the number of physical cores (and not hyperthreaded processing units) on his machine is `cpu_cores`

```
(cpl_py3) xxx>pip install cpu_cores
```

## A.1 Installation of mpi4py

The coupling communications being performed by MPI we need the `mpi4py` package, but the installation has to be coherent with the MPI flavour used for the compilation of telemac, if its parallel version has been compiled.
Please, check that the commands `mpicc` and `mpirun` found by default in your environment (as determined by the `PATH` variable) are the same that you used to compile Open TELEMAC-MASCARET.
In our case we check that

```
(cpl_py3) xxx>which mpicc
/softs/local/openmpi400_gcc731/bin/mpicc

(cpl_py3) xxx>mpicc --version
gcc (GCC) 7.3.1 20180130 (Red Hat 7.3.1-2)
```

and

```
(cpl_py3) xxx>which mpirun
/softs/local/openmpi400_gcc731/bin/mpirun

(cpl_py3) xxx>mpirun --version
mpirun (Open MPI) 4.0.0
```

Once the paths are correctly set, you can proceed to the installation

```
(cpl_py3) xxx>pip install mpi4py
```

# B. Installation of the conda environment

If conda is not yet installed on your machine, please refer to `https://www.anaconda.com/`
For a lighter installation, we suggest the use of miniconda3 [`https://docs.conda.io/en/latest/miniconda.html`].

You need to create a complete environment under python3 (the default python version for miniconda3), but you should avoid to install `mpi4py` from conda itself (*cf.* section B.1). You have to provide a name for the dedicated environment: in the following instructions we arbitrarily chose `cpl_py3`.

```
xxx>source deactivate

xxx>conda update -n base conda

xxx>conda create -n cpl_py3 -c conda-forge \
>python openturns=1.10 matplotlib numpy pandas scipy scikit-learn \
>pathos jsonschema paramiko sphinx sphinx_rtd_theme pytest \
>pytest-runner mock ffmpeg shapely cython

xxx>source activate cpl_py3
```

## B.1 Installation of mpi4py

It is mandatory that the python MPI interface `mpi4py` is compiled against the same version of MPI used for the coupled codes. For this reason we should not install the precompiled version provided by conda, but we have to compile and install it via `pip`.

Please, check that the commands `mpicc` and `mpirun` found by default in your environment (as determined by the `PATH` variable) are the same that you used to compile Open TELEMAC-MASCARET.
In our case we check that

```
xxx>which mpicc
/softs/local/openmpi400_gcc731/bin/mpicc
```

```
xxx>mpicc --version
gcc (GCC) 7.3.1 20180130 (Red Hat 7.3.1-2)
```

and

```
xxx>which mpirun
/softs/local/openmpi400_gcc731/bin/mpirun

xxx>mpirun --version
mpirun (Open MPI) 4.0.0
```

Once the paths are correctly set, you can proceed to the installation of `mpi4py`. Notice that the installation is normally driven by

```
xxx>pip install mpi4py
```

but that a specific feature of OpenMPI 4 requires a maintenance version, until the official mpi4py distro isn't updated

```
xxx>pip install --user https://bitbucket.org/mpi4py/mpi4py/get/maint.zip
```

# C. A complete example of `coupling_description.py`

Taken from

```
$HOMETEL/examples/python3/telapy_coupling/channel_manning/\
coupling_description.py
```

```
coupling_def = {
    "Coupling": {
        "TimeStep": 10.0,
        "Method": "MultiplicativeSchwarz",
        "MaxIter": 5,
        "CplStepRestart1D": "Persist2D",
        "CplStepRestart2D": "Persist1D"
    },
    "1D": {
        "ch1d": {
            "TimeStep": 10.0,
            "OutputFreq": 10.0
        }
    },
    "2D": {
        "ch2d": {
            "TimeStep": 1.0,
            "OutputFreq": 10.0
        }
    },
    "Interfaces": [
        {
            "Id1D": "ch1d",
            "IdExtr1D": "downstream",
            "Condition1D": "Discharge",
            "Id2D": "ch2d",
            "LiqBdry2D": 2,
            "1DPosition": "UpStream",
            "ConvCriteria": {
                "Height": 0.01,
```

```
                "Velocity": 0.015
            }
        }
    ]
}

config_run = {
    "Run": {
        "RefDate": "1/1/2019 00:00:00",
        "StartDate": "1/1/2019 00:00:00",
        "EndDate": "1/1/2019 00:03:20",
        "SingleExecDuration": "00:01:40",
        "RestartFromFile": "yes"
    },
    "2D": {
        "ch2d": {
            "Parallel": "no",
            "NbProc": 4
        }
    }
}

config_ch1d = {
    "files": {

        "xcas": "../Input_Data/1d_stretch/ParametresMascaret.xcas",
        "geo": "../Input_Data/1d_stretch/Channel1d.geo",
        "res": "ResultatsOpthyca_ch1d.opt",
        "listing": "ResultatsListing_ch1d.lis",
        "damocle": "listing.damoc",
        "lig": "../Input_Data/1d_stretch/WaterLine_ch1d.lig",
        "loi": [
            "../Input_Data/1d_stretch/1d_up.loi",
            "../Input_Data/1d_stretch/1d_down.loi"
        ]
    }
}
config_ch2d = {
    "files": {
        "cas": "../Input_Data/2d_rect/T2DCAS"
    }
}
```

# D. A complete example of user defined procedure

Taken from

`$HOMETEL/examples/python3/telapy_coupling/channel_manning/\`
`driven_coupling.py`

Notice that, w.r.t. to the run configuration for the same case described in App. C, in this example the two single executions are run via two calls to the driver with two different `config_run` dictionaries.

```
import os
from telapy.coupling.long_cpl_driver import LongCplDriver

user_coupling = {
    "Coupling": {
        "TimeStep": 10.0,
        "Method": "MultiplicativeSchwarz",
        "MaxIter": 5,
        "CplStepRestart1D": "Persist2D",
        "CplStepRestart2D": "Persist1D"
    },
    "1D": {
        "ch1d": {
            "TimeStep": 10.0,
            "OutputFreq": 10.0
        }
    },
    "2D": {
        "ch2d": {
            "TimeStep": 1.0,
            "OutputFreq": 10.0
        }
    },
    "Interfaces": [
        {
            "Id1D": "ch1d",
```

```
            "IdExtr1D": "downstream",
            "Condition1D": "Discharge",
            "Id2D": "ch2d",
            "LiqBdry2D": 2,
            "1DPosition": "UpStream",
            "ConvCriteria": {
                "Height": 0.01,
                "Velocity": 0.015
            }
        }
    ]
}

models_configs = {
    "config_ch1d" : {
        "files": {

            "xcas": "../Input_Data/1d_stretch/ParametresMascaret.xcas",
            "geo": "../Input_Data/1d_stretch/Channel1d.geo",
            "res": "ResultatsOpthyca_ch1d.opt",
            "listing": "ResultatsListing_ch1d.lis",
            "damocle": "listing.damoc",
            "lig": "../Input_Data/1d_stretch/WaterLine_ch1d.lig",
            "loi": [
                "../Input_Data/1d_stretch/1d_up.loi",
                "../Input_Data/1d_stretch/1d_down.loi"
            ]
        }
    },
    "config_ch2d" : {
        "files": {
            "cas": "../Input_Data/2d_rect/T2DCAS"
        }
    }
}


driven_run = {
    "Run": {
        "RefDate": "1/1/2019 00:00:00",
        "StartDate": "1/1/2019 00:00:00",
        "EndDate": "1/1/2019 00:01:40",
        "SingleExecDuration": "00:01:40",
        "RestartFromFile": "yes"
    },
    "2D": {
        "ch2d": {
            "Parallel": "no",
            "NbProc": 4
```

```
        }
    }
}

the_coupling = LongCplDriver(".", user_coupling, models_configs)
the_coupling(driven_run)

driven_run["Run"]["StartDate"] = driven_run["Run"]["EndDate"]
driven_run["Run"]["EndDate"] = "1/1/2019 00:03:20"
the_coupling(driven_run)
```

# E. A complete example of json input files

This example translates into json files the configuration of
`$HOMETEL/examples/python3/telapy_coupling/channel_ic`

## E.1 A complete example of `CouplingDef.json`

```
{
    "Coupling" : {
        "TimeStep" : 10.0,
        "Method" : "MultiplicativeSchwarz",
        "MaxIter"  : 5,
        "CplStepRestart1D" : "Persist2D",
        "CplStepRestart2D" : "Persist1D"
    },
    "1D" : {
        "ch1d" : {
            "TimeStep" : 10.0,
            "OutputFreq" : 20.0
        }
    },
    "2D" : {
        "ch2d" : {
            "TimeStep" : 1.0,
            "OutputFreq" : 20.0,
            "OutputSites" : [18, 33, 1018, 1024, 1034, 1044, 1054,
                             1064, 1074, 1084, 1094, 1104,
                             1114, 1124, 1134, 1144, 1154,
                             1164, 1174, 1184, 1194, 1204,
                             1214, 1224, 1234, 1244, 1254,
                             1264, 1274, 1284, 1294, 1304,
                             1314, 1324, 1334, 1344, 1354,
                             1364, 1374, 1384, 1394, 1404,
                             1414, 1424, 1434, 1444, 1454,
                             1464, 1474, 1484, 1494, 1504,
                             30]
```

```
        }
    },
    "Interfaces" : [
        {
            "Id1D" : "ch1d",
            "IdExtr1D" : "downstream",
            "Condition1D" : "Discharge",
            "Id2D" : "ch2d",
            "LiqBdry2D" : 2,
            "1DPosition" : "UpStream",
            "ConvCriteria" : {
                "Height" : 0.01,
                "Velocity" : 0.015
            }
        }
    ]
}
```

## E.2 A complete example of `ConfigRun.json`

```
{
    "Run" : {
        "RefDate" : "1/1/2019 00:00:00",
        "StartDate" : "1/1/2019 00:00:00",
        "EndDate"   : "1/1/2019 02:00:00",
        "SingleExecDuration" : "02:00:00",
        "RestartFromFile" : "no"
    },
    "2D" : {
        "ch2d" : {
            "Parallel" : "no",
            "NbProc" : 4
        }
    }
}
```

## E.3 A complete example of `config_ch1d.json`

```
{
 "files": {
        "xcas": "../Input_Data/1d_stretch/ParametresMascaret.xcas",
        "geo": "../Input_Data/1d_stretch/Channel1d.geo",
        "res": "ResultatsOpthyca_ch1d.opt",
        "listing": "ResultatsListing_ch1d.lis",
        "damocle": "listing.damoc",
        "lig": "../Input_Data/1d_stretch/WaterLine_ch1d.lig",
        "loi": [
                "../Input_Data/1d_stretch/1d_up.loi",
                "../Input_Data/1d_stretch/1d_down.loi"
               ]
```

```
        }
}
```

## E.4 A complete example of `config_ch2d.json`

```
{
 "files": {
        "config_file" : "${SYSTELCFG}",
        "config_option" : "${USETELCFG}",
        "cas" : "../Input_Data/2d_rect/T2DCAS"
        }
}
```

[1] S. Barthélémy. *Assimilation de données ensembliste et couplage de modèles hydrauliques 1D-2D pour la prévisiond es crues en temps réel. Application au réseau hydraulique Adour Maritime.* PhD thesis, Institut National Polytechnique de Toulouse, 2015.

[2] S. Barthélémy, S. Ricci, T. Morel, N. Goutal, E. Le Pape, and F. Zaoui. On operational flood forecasting system involving 1d/2d coupled hydraulic model and data assimilation. *Journal of Hydrology*, 562:623–634, 2018. doi: 10.1016/j.jhydrol.2018.05.007.

[3] M-P. Daou. *Développement d'une méthodologie de couplage multimodèle avec changements de dimension. Validation sur un cas-test réaliste.* PhD thesis, Université Grenoble Alpes, 2016.

[4] N. Malleron, F. Zaoui, N. Goutal, and T. Morel. On the use of a high-performance framework for efficient model coupling in hydroinformatics. *Environmental Modelling and Software*, 26(12):1747–1758, 2011. doi: 10.1016/j.envsoft.2011.05.017.

[5] A. Piacentini, T. Morel, F. Duchaine, and A. Thevenin. O-palm: An open source dynamic parallel coupler. *Proceedings of the 4th International Conference on Computational Methods for Coupled Problems in Science and Engineering, COUPLED PROBLEMS 2011*, 2011.

[6] S. Ricci, N. Goutal, M. Parisot, S. Barthélémy, P. Roy, and M. De Lozzo. Estimates of hydraulic variables and uncertainty estimates for major rivers in france used for hydropower production. In *TR-CMGC-18-204, CERFACS*, 2018.

[7] M. Tayachi. *Couplage de modèles de dimensions hétérogènes et application en hydrodynamique*. PhD thesis, Univ. Grenoble Alpes, 2013.

[8] M. Tayachi, A. Rousseau, E. Blayo, N. Goutal, and V. Martin. Design and analysis of a Schwarz coupling method for a dimensionally heterogeneous problem. *International Journal for Numerical Methods in Fluids*, 75(6):446–465, 2014.