MASTER IN HIGH PERFORMANCE COMPUTING

A HPC approach to the Boundary Conditions for the Copernicus biogeochemical model of the Mediterranean Sea

Supervisor(s):
Gianpiero COSSARINI,
Alberto SARTORI

Candidate:
Marco BETTIOL

4th EDITION
2017–2018

Acknowledgements

The research reported in this work was supported by OGS and CINECA under HPC-TRES program award number 2017-19.

Contents

List of Abbreviations

OGSTM	**O**GS **T**ransport **M**odel
BFM	**B**iogeochemical **F**lux **M**odel
OGCM	**O**cean **G**lobal **C**irculation **M**odel
NEMO	**N**ucleus (for) **E**uropean **M**odeling (of) (the) **O**cean
CMEMS	**C**opernicus **M**arine **E**nvironment **M**onitoring **S**ervices
RAII	**R**esource **A**llocation **I**s **I**nitialization
PIMPL	**P**ointer (to) **IMPL**ementation
italic names	Proper nouns
`blue verbatim names`	Words used literally in the code

Chapter 1

Introduction

The code on which this work is focused is a coupled physical-biogeochemical model which solves both the transport equation and the biogeochemical reactions in the seawater. Up to the current version, the only domain in which the model is supposed to run is the Mediterranean Sea. Its boundary conditions are thought to be invariant, both in their geometry, in their numerical schemes and in their parameterization. In the current version, therefore, any change in the boundary conditions is possible only for specialized developers and is increasing a lot the time to solution anytime any new configuration is required. Referring to fig. 1.1, the most of the time in this case is spent in adding new subroutines on anyway modifying the existing ones, changing some hard-coded parameters, changing input files, scripts etc. and the run-time is only a small fraction of the overall time to solution.

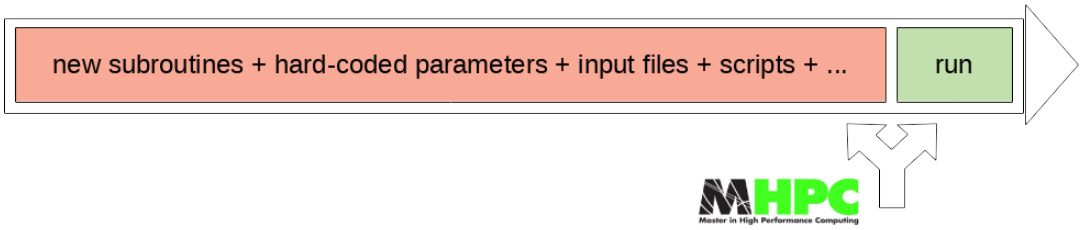

FIGURE 1.1: Current time to solution. The focus of the MHPC thesis is to reduce the red part first.

The final goal to which this project contributes to is to unlock the HPC potential of the model by making it adaptable to different domains and to ensemble simulations scenarios. The first step in this direction, and actually the real aim of this thesis, is to build a versatile interface and infrastructure to assign the boundary conditions. Configuring the existing boundaries, but also changing the boundaries themselves, have to be fully supported operations. In this way the model is able to be run easily multiple times, potentially in any domain and with different boundary conditions. Furthermore, this operations should be carried on by the user itself, without even touching the source code, in order to dramatically reduce the time to solution and to enable the model to be run by a wider community.

Chapter 2

Model overview

2.1 Model description

This chapter gives a brief overview of the biogeochemical model which is at the basis of the thesis work, together with a classification of its boundary conditions.

2.1.1 *MED-CURRENT* framework

The computational model which is at the basis of this work is the coupled physical-biogeochemical *OGSTM-BFM* model [1]. It is composed by the *OGSTM* transport model, which solves both the advection and the diffusion processes in seawater, and a biogeochemical reactor, namely Biogeochemical Flux Model (*BFM* [2]). The two models are online coupled and together solve the full transport/reaction equation [3]:

$$\frac{\partial \boldsymbol{C}}{\partial t} = -\boldsymbol{v} \cdot \nabla \boldsymbol{C} + \nabla(K \nabla \boldsymbol{C}) + R_{bio}(\boldsymbol{C}, ...)\,, \tag{2.1}$$

with the boundary conditions

$$\boldsymbol{C}(\boldsymbol{x}_0) = \boldsymbol{C}_0 \quad \text{or} \quad \frac{\partial \boldsymbol{C}}{\partial t}(\boldsymbol{x}_0) = \boldsymbol{\alpha}\,. \tag{2.2}$$

$\boldsymbol{C}$ is the array of the outputs of the model, i.e. a series of 51 biogeochemical tracers scalar fields the most important of which are:

- chlorophyll;
- phytoplankton carbon biomass;
- net primary production;
- phosphate;
- nitrate;
- oxygen;
- acidity (pH);
- partial pressure of carbon dioxide in seawater (pCO2).

The first term on RHS refers to the advection. v is the velocity, which is imposed as a forcing field, derived from an offline coupled Ocean Global Circulation Model (*OGCM*) and not computed by *OGSTM* itself. The second term represents the diffusion process, and K is the diffusivity. The biogeochemical reactions are finally grouped into the last term. Boundary conditions for tracers can be set either as a

tracer value (C_0) or a tracer flow (α). Both cases can be also time dependent, as for example when seasonal variability is considered.

The aforementioned components of the system are self-consistent. Nevertheless the model is run inside a wider framework including also a Data Assimilation scheme named *3DVAR-BIO*, used for the correction of phytoplankton functional type variables with surface chlorophyll data coming from satellite observations. The whole system is referred to as *MedBFM* model. The latest release of *MedBFM* is *MedBFM 2.1* which in turn includes *3DVAR-BIO 3.2*, *OGSTM 3.2.1* and *BFM 2.1*. A summary of the components is reported in fig. 2.1.

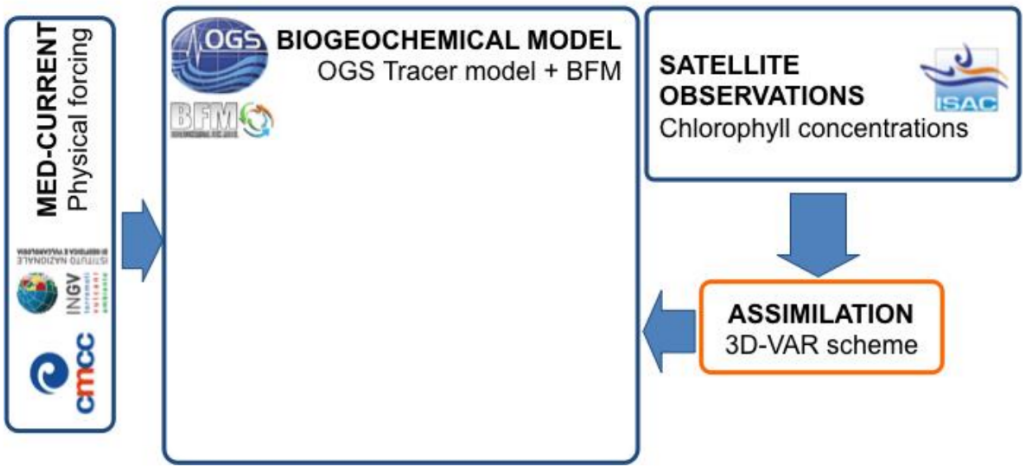

FIGURE 2.1: Model overview: the core component is the coupled physical-biogeochemical *OGSTM-BFM* model; velocity fields are fully consistent with the output of a *OGCM*, which in the picture is the *NEMO* Ocean model [4], on which *MED-CURRENT* system is based. *3DVARBIO* scheme is used for Data Assimilation of chlorophyll data.

The focus of the present work is on the modules of the system which set the boundary conditions, which are part only of *OGSTM*. Therefore, from now when referring to the *model*, *OGSTM* will be implied. The current version, *OGSTM 3.2.1*, has been modified in order to provide a brand new structure for the boundary conditions. Together with other major updates, the new boundary scheme will be part of the new release of the model, *OGSTM 4.0*. From now on, *OGSTM 3.2.1* and *OGSTM 4.0* will be referred to respectively as *current* version and *new* version.

2.1.2 CMEMS products

The *MedBFM* system is at the basis of two products developed for the European Commission under the CMEMS[1], respectively for the reanalysis of the biogeochemical state of the Mediterranean in the past 20 years and for its short-term forecasts. In this second product, namely *MEDSEA_ANALYSIS_FORECAST_BIO*, the workflow runs twice per week and produces a 10 days forecast. This configuration is used as a reference to introduce a standard computational setup for the model, as described below.

[1] Copernicus Marine Environment Monitoring Services

2.1.3 HPC configuration and setup

The model is written in Fortran (90 standard) and it is parallelized in distributed memory using *MPI* both when it is run on different cores inside a single compute node and when it requires multiple nodes. With reference to the CMEMS analysis and forecasting product, in its standard configuration the model runs on 20 computing nodes of Marconi (CINECA), using a total of 720 cores (36 cores/node). Domain decomposition minimizes the land/water ratio for the *MPI* processes domain cells, in order to optimize the load balancing. An example of domain decomposition, for a smaller number of processes, is provided in fig. 2.2.

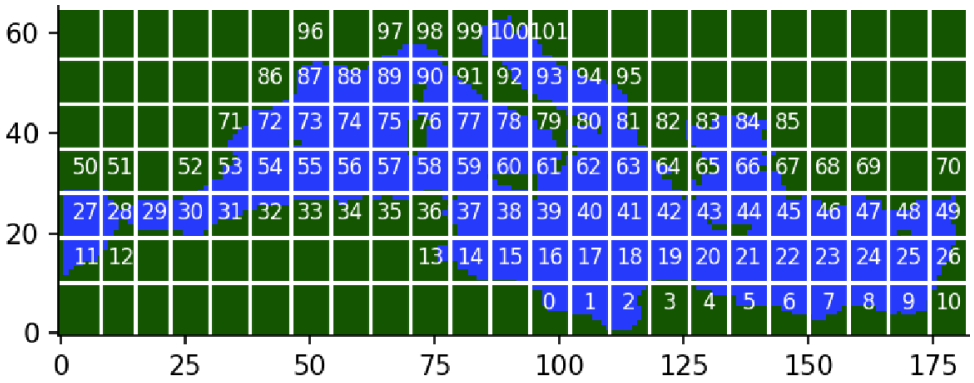

FIGURE 2.2: Domain decomposition in the case of 102 *MPI* processes. The total number of processes is less than the standard one just to provide a clearer picture.

2.2 Boundary conditions overview and classification

As seen before, boundary conditions for tracers are assigned either as a tracer value or a tracer flux, potentially time dependent (eq. 2.2). The second important distinction is whether a boundary is open or closed, i.e. whether a mass flow can occur or not at the boundaries. Since the velocity field is imposed by an offline coupled *OGCM*, a boundary of *OGSTM* can be open only if the corresponding boundary is open in the *OGCM*; otherwise it is forced to be closed.

With reference to fig. 2.3, if an *OGCM* boundary is closed there are two possible configurations. If a tracer flow is set (usually only in a few surface cells), the boundary is defined as a *river*, since this is the case in which an amount of tracers is discharged from the rivers or from a small strait into the ocean. Differently, if a tracer value is set, the boundary is defined just *closed*. Moving to the right side of the diagram in fig. 2.3, if the *OGCM* provides an open boundary (or also when *OGSTM* is run on a subset of the *OGCM* domain, and therefore a non null velocity field is available regardless of the *OGCM* boundary type) two more options are available. The velocity field can be artificially nullified in *OGSTM*, yielding a so called *sponge*[2] boundary, or it can be left unvaried, providing a fully *open* boundary also to *OGSTM*. In the first case usually a smoothing function is provided as well, in order to relax the velocity field to the *OGCM* values far from the boundary.

[2]The name is a common jargon in oceanographic modeling and derives from the fact that setting to zero the velocity field at the boundary is like "absorbing" the mass.

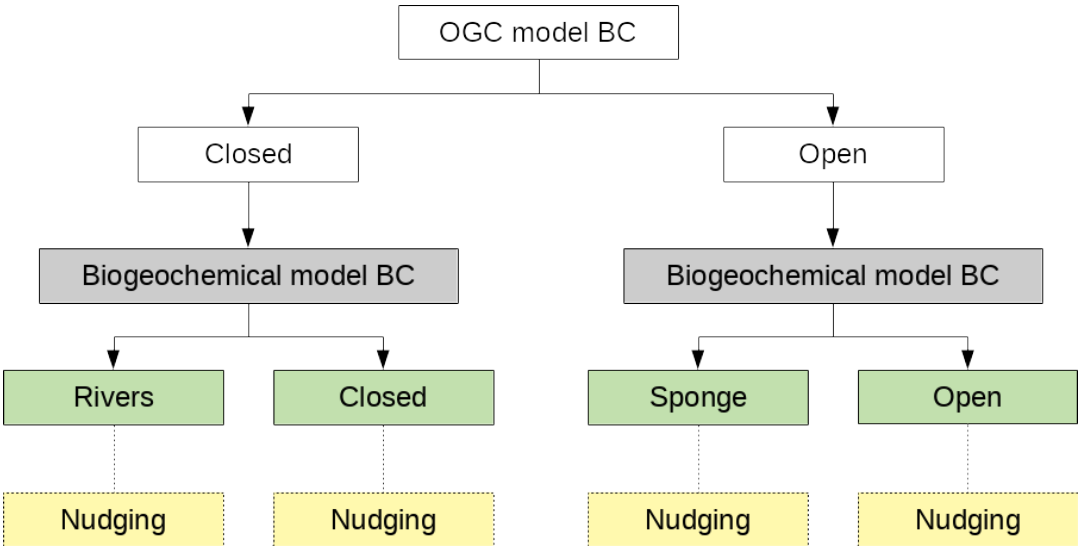

FIGURE 2.3: Boundary classification. Colors scheme is the same used later (see fig. 4.1) to map the boundary types into the actual classes. Generic boundary conditions for *OGSTM* are marked in grey, while biundaries of every specific type are marked in green. Nudging feature is marked in orange and it is meant to be an optional feature for any boundary type.

For each of the four boundary types, and additional specification is whether to gradually relax or not the tracer fields to the given values at the boundaries using a Newtonian dumping term, also known as *nudging*. The *nudging* term is thought to be optional and is useful to provide numerical stability to the model. In order to apply it, a 3D set of tracer values has to be assigned in proximity of the boundary and used to adjust the model outputs. This can be done through a weighted combination of imposed and predicted values, with the weight spanning from 1 at the boundary to 0 in the inner domain. The fact that this behavior can be applied or not to any of the boundary types is crucial for the choice of the design pattern for its implementation, as described in subsection 4.2.7.

Chapter 3

Current version: *OGSTM* 3.2.1

This chapter gives an overview of the current code structure, focusing mostly on the boundary conditions modules. A brief description of these modules is provided, together with some profiling data. The related issues and the reasons behind the choice of moving to a new implementation are discussed as well.

3.1 Overview of the involved procedures

A crucial point that is worth to introduce from the beginning is that, up to version *3.2.1*, *OGSTM* has been designed to run always in the same domain, i.e. the Mediterranean Sea. For this reason the current implementation does not allow to change the boundary conditions geometry and neither their type, unless with deep changes in the code. Among the boundaries needed by the model in this configuration, the ones of interest for this work are all the rivers that flows into the Mediterranean and the two straits of Dardanelles and Gibraltar. An overview is provided in fig. 3.1. Due to its dimensions, Dardanelles strait is actually treated as a river input and its boundary conditions fall inside the rivers boundary conditions subset. On the other hand, Gibraltar strait, much deeper and wider, is configured as a *sponge* boundary and a *nudging* term is applied on it. [5], [6], [7]

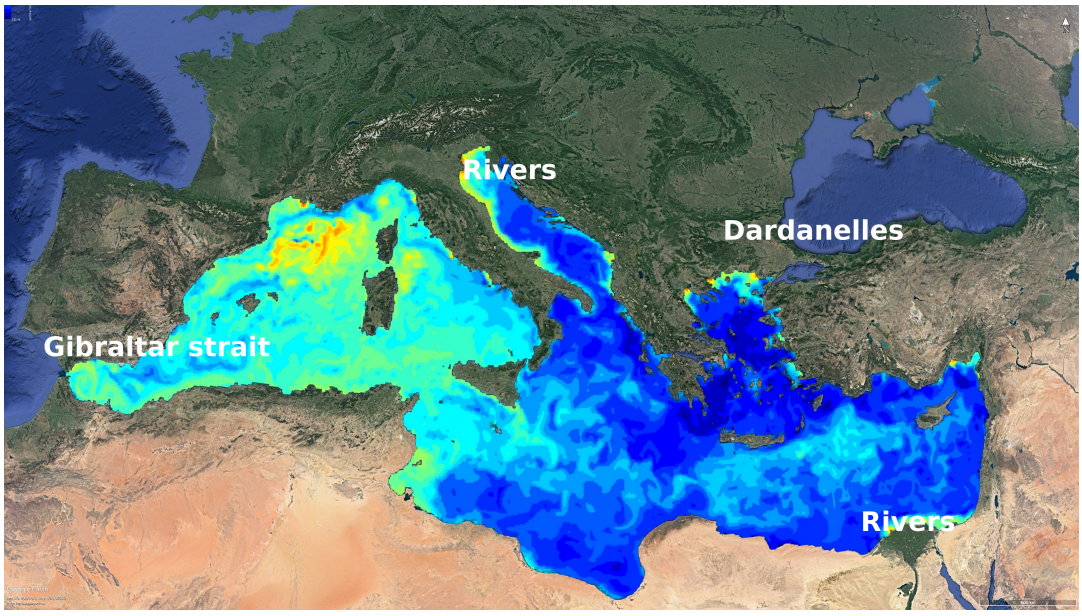

FIGURE 3.1: Boundary conditions overview.

Boundary conditions related subroutines are essentially divided in two groups. The first ones are called once in the initialization phase, while the others are invoked at every time step. All the variables and matrices needed for storing boundary data are declared as global variables in a dedicated module (BC_mem) and are allocated during the initialization phase. Allocation is performed inside the domrea subroutine, together with the operations for reading the boundary data from the data files, divide them among the MPI processes and re-index them accordingly. The sequence of operations is repeated for every boundary, i.e. for the Gibraltar strait and for the rivers and Dardanelles. An example is provided in code box 3.1, which shows the main workflow for the Gibraltar strait: read a *netcdf* file with the data (readnc_int_1d), check which boundary cells fall into the involved *MPI* process (COUNT_InSubDomain_GIB), allocate memory (alloc_DTATRC_local_gib) and re-index the cells (GIBRe_indexing).

LISTING 3.1: Workflow for the Gibraltar boundary condition initialization in the current code

```
filename = 'BC/GIB_'//TC_GIB%TimeStrings(1)//'.nc'
call readnc_int_1d(filename, 'gib_idxt_N1p', Gsizeglo, &
        gib_idxtglo)
Gsize = COUNT_InSubDomain_GIB(Gsizeglo, gib_idxtglo)
if (Gsize .ne. 0) then
    call alloc_DTATRC_local_gib
    B = GIBRe_indexing()
endif
```

Inside the step subroutine, which integrates the model at every time step, a separate subroutine is called for each boundary, so in this case both for the rivers (bc_tin) and for Gibraltar (bc_gib). Different sets of boundary conditions are provided for a series of times, and need to be interpolated at every new time step. The main procedures invoked for each boundary are three: load_<BC_NAME>, swap_<BC_NAME> and actualize_<BC_NAME>. They are used, respectively, to load in memory one single set of data, to swap in memory two sets of data referring to two different times and to interpolate the data. Further details on the interpolation will be given in section 4.2.1, together with the new implementation. Again, a few example lines are reported for the Gibraltar boundary in code box 3.2.

LISTING 3.2: Subroutines for the Gibraltar boundary condition interpolation in the current code

```
if (...) then
    ! [...]
    call swap_GIB
    ! [...]
    call load_GIB(...)
end if

select case (...)
    case (0)
        if (...) then
            call actualize_GIB(...)
        end if
    case (1)
        call actualize_GIB()
```

```
end select
```

Finally, the actual application of the boundary condition for the tracers is done inside the `trcdmp` subroutine (in which a different code block is contained for each boundary), whereas conditions on the outputs of the *OGCM*, as for example setting the velocity field to zero in a *sponge* boundary, are applied inside the `init_phys` subroutine.

3.2 Profiling

Before examining some possibles bottlenecks of this implementation, some profiling data are reported. Table 3.1 summarizes the run-times of a few subroutines of the model. Among them, the aforementioned `bc_tin` and `bc_gib` are included, together with two other boundary related subroutines (for the atmosphere and the CO2, which are out of the scope of this work but still useful for a time comparison) and with the most important subroutines which are launched at every time step, i.e `trcstp`, which solves both the advection and the diffusion equations, and `step` itself, which embeds all the previous ones. The model configuration is the standard one used in the CMEMS forecast.

TABLE 3.1: operative chain subroutines elapsed times for current version: boundaries and main integration step

Subroutine	Overall time (s)	Single call time (s)
bcTIN	1.97 e-01	1.97 e-05
bcGIB	1.13 e-01	1.13 e-05
bcATM	1.71 e-01	1.71 e-05
bcCO2	1.08 e-01	1.08 e-05
trcstp	6.29 e+03	6.29 e-01
stp	1.22 e+04	1.22 e+00

It can be easily seen how the run-times for the boundary conditions subroutines are negligible with respect to the time required by a full integration step. Nevertheless, the actual time to solution to be considered includes also, and above all, the time spent in adapting the code to different boundary conditions and to configure the model accordingly, which, as discussed later, is far from being negligible.

A further hint of the low impact of the boundary conditions operations in terms of computational effort is given by the following I/O profiling obtained with Darshan with the same configuration and summarized in fig. 3.2. Input files includes mesh files, restart files and boundary conditions files. Although dominant in terms of number of operations (right graph), input file readings together requires a very small portion of the actual run-time (left graph, red column), by far smaller also than the output files writing percentage (left graph, green column).

3.3 Issues

Despite its low impact in the actual run-time, such an implementation is fine only if every small detail of a model configuration at the boundaries is supposed to be left unchanged. As soon as some boundary conditions have to be added or removed,

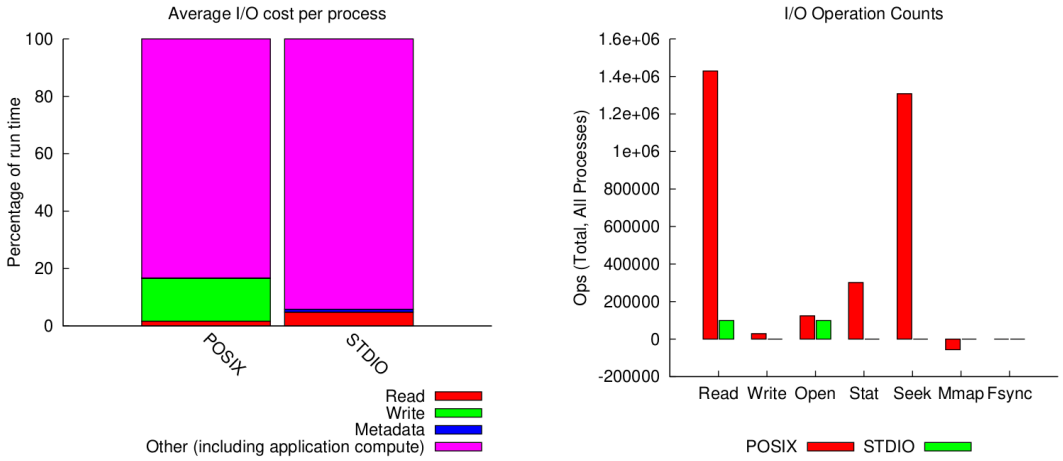

FIGURE 3.2: I/O profile of the current version.

or even modified only with respect to a few parameters, some important restrictions arise:

- many parameters are hard-coded, and replicated in many parts of the code. Even a minor change in the number or in the values of a boundary condition requires an analysis and a refactoring of the code, with the consequence of generating potential errors and typos and anyway increasing the time to solution;
- adding a new boundary means rewriting and adding calls to the full set of subroutines that are needed to upload and apply the boundary data. Code size would rapidly grow, together with the time spent by the developer and the chances to introduce bugs.

These are the main reasons behind the choice of writing from scratch a new implementation for the boundary conditions, which will be described in the following chapters. The aim is to minimize the time to solution focusing on the model initialization phase, of course without slowing down the computational phase.

Chapter 4

New Object-Oriented structure

The basic principle in Object-Oriented programming is to map a series of variables and a series procedures into a single programming abstraction named *class*; the two series are defined, respectively, *members* and *methods* of the class. Different classes can share common features by *inheriting* from a common base class. If a class inherits from a base class, it belongs to that class, so it shares the same *members* and *methods* of that class. Additional *members* and *methods* defined in the derived class contribute to the definition of the specific features of the new class. Objects are concrete representations of a class and are instantiated, i.e. allocated in memory, with a call to a dedicated method of the class named *constructor*.

In this chapter the new Object-Oriented implementation is introduced and discussed. A first section explains the reasons behind an Object-Oriented paradigm, whereas in the following part the single classes are introduced.

4.1 Why an Object-Oriented philosophy?

The main purpose of this work is to make the code flexible and easy to use under a change of:

- boundary parameters (boundary data, subset of tracers etc.);
- type of boundary;
- number of boundaries.

The final goal is to let the user free to assign an arbitrary number of boundaries of any kind, possibly doing this through a namelist[1] file, without even touching the source code. The whole implementation relies on an Object Oriented paradigm, in which the main idea is to associate each type of boundary to a different class. Some methods are the same for every boundary class. Furthermore, a series of methods needs to be overridden by every class, in order to come up with an unique interface and handle each object in the same way, regardless of its type. The reasons behind this approach are at least three:

- any number of boundaries, of any type, can be instantiated by simply calling the proper constructors;
- every customization for a specific boundary can be obtained by simply passing the desired parameters as argument to the constructor. In this way the boundary initialization, including memory allocation and data input, is carried out by a single call in the main code, without additional modules for the memory

[1] Usually this is the term used for files containing the list of parameters for the model initialization.

management and without declaring the needed arrays and matrices as global variables in different parts of the code. All the auxiliary variables and procedures declared and/or implemented in the current version are now hidden under the object instantiation, which requires all the arguments to be read from a namelist. In Object-Oriented programming, this concept is defined as *RAII* (*Resource Acquisition Is Initialization*) [8]. The idea behind it is to bind all the resources that are needed by the object to its lifetime. This means that resources such as the allocated heap memory, the execution thread etc. are allocated by the constructor and are preserved until they are deallocated by the destructor (see 4.2.8).

- every action that needs to be performed on the boundary can be included in a single loop over all the istantiated boundaries, calling the same methods on different objects, regardless of their types (object *polymorphism*).

The new version of the boundary modules has been developed using Modern Fortran (2003 standard), which, among the major updates, introduces also Object-Oriented syntax and features [9]. Compatibility of *OGSTM 3.2.1* under the new standard has been tested and guaranteed before moving to the new implementation.

4.2 Class structure

An overview of the final inheritance diagram is provided in fig. 4.1. The new scheme is based on a base[2] class, `bc`, from which every different boundary class inherits, implementing the same template. The other two members of the structure are a wrapper over the data files, `bc_data`, and a decorator (see subsection 4.2.7), `nudging`, which provides *nudging* features to any boundary type if necessary. The inheritance diagram follows the boundary classification provided in 2.2, and colors are set accordingly. A detailed description of the implemented objects follows.

4.2.1 bc_data

This is a wrapper over the data structures which contains the full dataset needed by the boundary. The dataset consists of a series of files referring to some specific times distributed along the whole simulation period. Boundary conditions can be yearly periodic, as for climatological data updated each month or each season, or they can just evolve without a given periodicity. At every time step a boundary condition object needs to know which are the two data files that refer to the current time interval. Therefore, `bc_data` class handles a list of *netcdf* data files and a corresponding list of times, with a method to get the corresponding data given the times list index (`get_file_by_index`). Lists are allocated and initialized inside the constructor, which is itself in charge of the memory management, according to *RAII*. It features two main constructors. The default one is inferring the time each file refers to from a time string contained in the file name itself. In this case the files and times lists will have the same length (fig. 4.2). The second, instead, is used to handle yearly periodic boundary conditions (e.g. climatological). Here only the constant part of the time string (i.e. month, day, hour etc.) is inferred from the files, and the list of times is computed and replicated for every simulation year, form start to end year

[2] Usually, in similar inheritance diagrams, the base class is defined *abstract*, whereas in this implementation it is not. The reasons behind this choice are explained in 4.2.2.

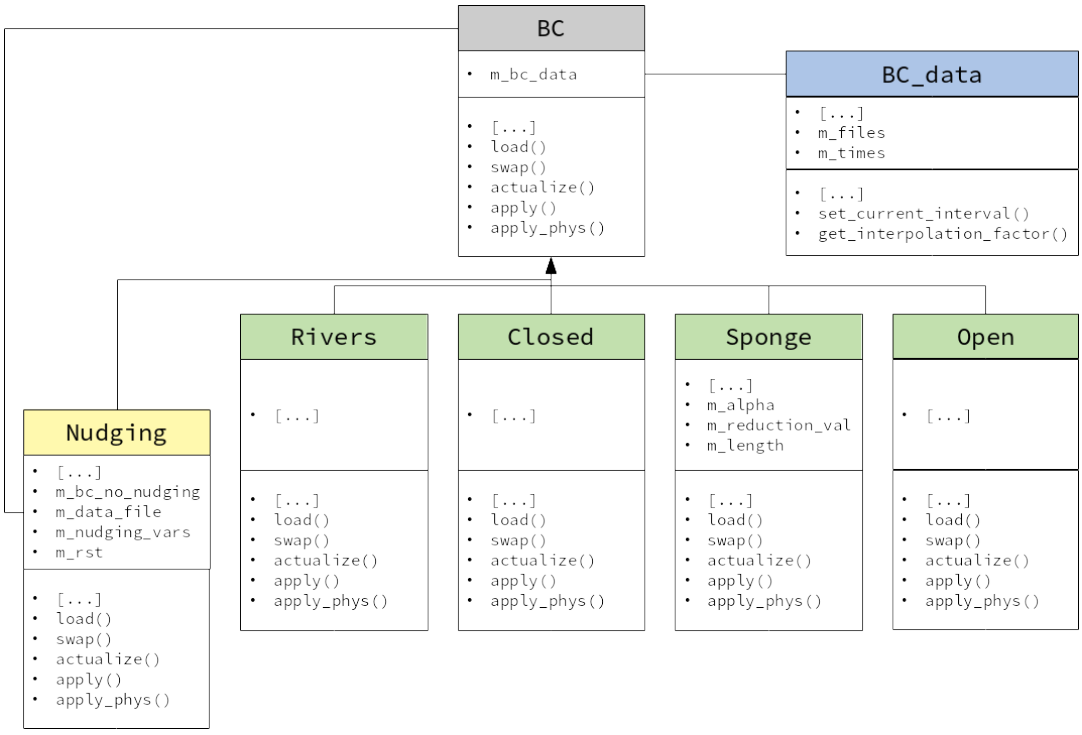

FIGURE 4.1: Class inheritance diagram based on the boundary classification provided in 2.2. Colors are mapping those of fig. 2.3. Class boxes are splitted in two parts, respectively for members and methods. Only the most significant members and methods are reported and omitted ones are replaced with the symbol ![...]. Arrows are used for inheritance, while plain lines are used when one of the class members points to an object of the linked class.

(fig. 4.3). Therefore, this constructor accepts two arguments more (simulation start and end times).

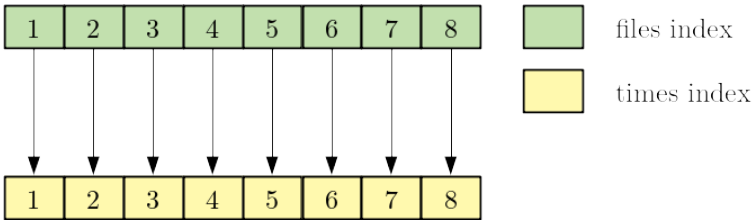

FIGURE 4.2: Non periodic files and times lists.

Constructor overloading (code box 4.1) is achieved through an interface. An empty constructor which does not allocate any list is also included. It is useful in the case a boundary with no data is needed (see for example the nudging class 4.2.7). The compiler will then select the right implementation depending on the number and the type of the arguments.

LISTING 4.1: constructor overloading for the bc_data class

```
interface bc_data
    module procedure bc_data_empty
    module procedure bc_data_default
```

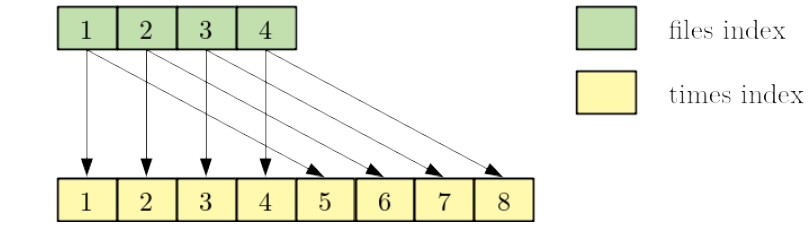

FIGURE 4.3: Periodic files and times lists.

```
    module procedure bc_data_year
end interface bc_data

! [...]

type(bc_data) function bc_data_empty()
    ! [...]
end function bc_data_empty

type(bc_data) function bc_data_default(files_namelist)
    character(len=22), intent(in) :: files_namelist
    ! [...]
end function bc_data_default

type(bc_data) function bc_data_year(files_namelist, &
        start_time_string, &
        end_time_string)
    character(len=27), intent(in) :: files_namelist
    character(len=17), intent(in) :: start_time_string
    character(len=17), intent(in) :: end_time_string
    ! [...]
end function bc_data_year
```

The current version linearly interpolates the data of the two extremes of the time interval in order to obtain the data at any given time inside the interval (see also fig. 4.4 in the following subsection). This part is now delegated to the `bc_data` class itself, through its `get_interpolation_factor` method, which indeed returns the weight for the linear interpolation. Besides doing this, it also updates, if necessary, two integer attributes of the class to keep track of the current interval's indexes. Sometimes it is just required to set the right interval without interpolating the data. This can be done through the `set_current_interval` method, which shares with the previous one only the updating part. Current interval indexes can be accessed via the `get_prev_idx` and `get_next_idx` getters, while the boolean `new_interval` method returns whether the interval has changed or not after the last call either to `set_current_interval` or `get_interpolation_factor`.

Data files are supposed to refer only to the tracers fields. They are *netcdf-4* files containing both an indexed set of values and the cells to which the values refer, i.e. the boundary geometry. Depending on the type of boundary, values can refer both to tracer values and to tracer flux values (each boundary class will then override its methods accordingly).

4.2.2 bc

This is the base class which defines the interface for every new boundary. Its members and methods will be common to every derived class. `bc` class role, indeed, is to own and/or implement common attributes and/or methods which provide common features to every type of boundary. Its only member, `m_bc_data`, follows a *Pointer To Implementation* (PIMPL) design pattern [10]. The purpose of a *PIMPL* is to move implementation details into a separate class, which is accessed through a pointer. `m_bc_data`, indeed, is a pointer to a `bc_data` object, which handles the operations on the list of files. The pointer allocation and the instantiation of the corresponding object are done inside the constructor. Overloaded `bc` constructors differs only in the call to the proper `bc_data` constructor (code box 4.2).

LISTING 4.2: constructor overloading for the bc class

```
type bc
    type(bc_data), pointer :: m_bc_data => null()
contains
    ! [...]
end type bc

! [...]

type(bc) function bc_empty()
    allocate(bc_empty%m_bc_data)
    bc_empty%m_bc_data = bc_data()
end function bc_empty

type(bc) function bc_default(files_namelist)
    ! [...]
    allocate(bc_default%m_bc_data)
    bc_default%m_bc_data = bc_data(files_namelist)
end function bc_default

type(bc) function bc_year(files_namelist, &
        start_time_string, &
        end_time_string)
    ! [...]
    allocate(bc_year%m_bc_data)
    bc_year%m_bc_data = bc_data(files_namelist, &
        start_time_string, &
        end_time_string)
end function bc_year
```

Even though the data files can be intrinsically different from each other (an *open* boundary for example relies on a 2D set of values, while a *nudging* is usually applied on a 3D geometry), every boundary will always refer to a list of files, and will always operate in the same way on them. For this reason, file operations are contained and implemented in the base class. Those methods are actually just a series of wrappers around the setters and the getters of the `bc_data` class, including `get_file_by_index`, `set_current_interval` and `get_interpolation_factor`. As an example, in code box 4.3 the implementation of the `get_interpolation_factor`

method is displayed. It calls the corresponding method in bc_data and uses a logical output variables to keep track of the possible jump on a new time interval.

LISTING 4.3: get_interpolation_factor method for the bc class

```
double precision function get_interpolation_factor(self, &
        current_time_string, &
        new_data)

    class(bc), intent(inout) :: self
    character(len=17), intent(in) :: current_time_string
    logical, optional, intent(out) :: new_data

    get_interpolation_factor = &
        self%m_bc_data%get_interpolation_factor( &
            current_time_string)
    new_data = self%m_bc_data%new_interval()

end function get_interpolation_factor
```

These are the only methods that are implemented in the bc base class. All the other methods are just declared in order to provide the common interface to the child classes, but not implemented. In a fully Object-Oriented implementation, these methods should have been declared *deferred*, similarly to what is done with *virtual* methods in other programming languages such as C++. bc class would have been thus a truly *abstract* class and the child classes would have been forced to implement the deferred methods, causing a compiler error otherwise. Complying with the Fortran 2003 standard, this would have led to a documented issue [11], i.e. to the fact that base class constructors would not have been callable inside a constructor of a derived class, for the same reason why an object of an abstract class just cannot be istantiated. Possible solutions are to move to Fortran 2008 standard (which overcomes this issue allowing the use of base constructors while inheriting from an abstract class), interposing an additional class between the abstract one and the child classes and avoid using the base class constructors, moving what is done here to other delegated subroutines. The first two solutions are definitely appropriate, but would add unnecessary complexity to the code, at least considering the relatively small number of classes involved. Opting for the third one would be actually as simple as renaming a few methods, but, still, it would drift the implementation apart from a pure Object-Oriented style, essentially moving members initialization outside the constructors. For this reason the final choice has been not to use *deferred* methods and not to define bc *abstract*, even though this is the intended behavior.

The methods that define the general interface are the followings:

- load, swap and actualize: they are all related to time interpolation and are needed, respectively, to load in memory the data of the extreme of a time interval, to swap in memory the data of two extremes and to set the right values according to the interpolation weight, as described in fig. 4.4.

  Their signatures are reported in code box 4.4.

LISTING 4.4: load swap and actualize methods for the bc class

```
subroutine load(self, idx)
    class(bc), intent(inout) :: self
```

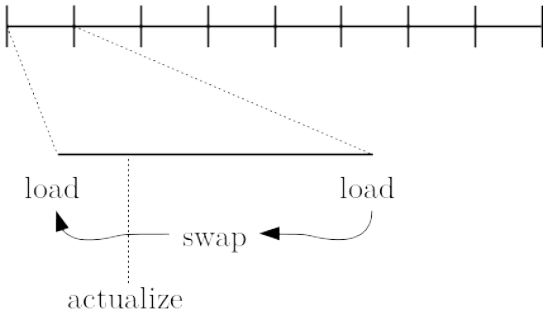

FIGURE 4.4: Role of the three methods load, swap and actualize in the time interpolation.

```
    integer, intent(in) :: idx
    ! [...]
end subroutine load

subroutine swap(self)
    class(bc), intent(inout) :: self
    ! [...]
end subroutine swap

subroutine actualize(self, weight)
    class(bc), intent(inout) :: self
    double precision, intent(in) :: weight
    ! [...]
end subroutine actualize
```

`load` needs to know which is the interval index to load the data from. `swap` just moves the loaded data to an auxiliary matrix, which is going to be a private member of the derived class. `actualize` requires a weight to correct the loaded data, even if the weight of the linear interpolation can be computed by the class itself through its `bc_data` object. In this way the method is more flexible when the linear interpolation option is not enabled in the namelist file, and the weight needs to be set always to 1.

- `apply`: this is called to set the final values of the tracer fields near the boundaries, according to the type of boundary. Model output fields are adjusted according to the boundary scheme. Its full signature is reported in code box 4.5.

LISTING 4.5: apply method for the bc class

```
subroutine apply(self, e3t, n_tracers, trb, tra)

    use modul_param, only: jpk, jpj, jpi

    implicit none

    class(bc), intent(inout) :: self
    double precision, &
        dimension(jpk, jpj, jpi), &
```

```
        intent(in) :: e3t
    integer, intent(in) :: n_tracers
    double precision, &
        dimension(jpk, jpj, jpi, n_tracers), &
        intent(in) :: trb
    double precision, &
        dimension(jpk, jpj, jpi, n_tracers), &
        intent(inout) :: tra

    ! [...]

end subroutine apply
```

`jpi`, `jpj` and `jpk` are the three local dimensions of the *MPI* process, set as global variables in the parameter module. Together with `n_tracers`, which is the total number of tracers, they are needed to specify the dimension of the tracers global matrices `tra` and `trb`. The method operates on the final output matrix (`tra`), modifying the model outputs according to the proper boundary scheme. `e3t` is just a vertical scale factor used to correct the (scalar) tracer fields according to the domain mesh.

- `apply_nudging`: a private method used to simplify the `apply` method if a nudging scheme has to be included in the fields correction. The signature is reported in code box 4.6.

LISTING 4.6: apply_nudging method for the bc class

```
subroutine apply_nudging(self, e3t, n_tracers, &
        rst_tracers, trb, tra)

    use modul_param, only: jpk, jpj, jpi

    implicit none

    class(bc), intent(inout) :: self
    double precision, &
        dimension(jpk, jpj, jpi), &
        intent(in) :: e3t
    integer, intent(in) :: n_tracers
    double precision, &
        dimension(jpk, jpj, jpi, n_tracers), &
        intent(in) :: rst_tracers
    double precision, &
        dimension(jpk, jpj, jpi, n_tracers), &
        intent(in) :: trb
    double precision, &
        dimension(jpk, jpj, jpi, n_tracers), &
        intent(inout) :: tra

    ! [...]

end subroutine apply_nudging
```

With respect to the previous one, it requires a restoration matrix (rst_tracers) to adjust the tracer values at the boundary. Such an information is part of the nudging scheme and therefore is owned by the nudging decorator.

- apply_phys: used to adjust the values of the velocity fields at the boundaries. Velocity fields are imposed to the model, usually according to the output of the *NEMO* Ocean Model or of an equivalent *OGCM*. Anyway they are not computed directly by *OGSTM*. Nevertheless with some specific boundary conditions schemes these outputs still need to be modified at the boundaries. This is the case for example of the *sponge* boundary, in which the velocity fields are forcefully set to zero at the boundary of *OGSTM*, in order for it to be closed. In such cases, this method is necessary both to nullify the velocity field components and to smooth the actual values of the *OGCM* forcing fields. The full signature is reported in code box 4.7.

LISTING 4.7: apply_phys method for the bc class

```
subroutine apply_phys(self, lat, sponge_t, sponge_vel)

    use modul_param, only: jpk, jpj, jpi

    implicit none

    class(bc), intent(inout) :: self
    double precision, &
        dimension(jpj, jpi), &
        intent(in) :: lat
    double precision, &
        dimension(jpj, jpi), &
        intent(out) :: sponge_t
    double precision, &
        dimension(jpk, jpj, jpi), &
        intent(out) :: sponge_vel

    ! [...]

end subroutine apply_phys
```

lat is a global variable containing the latitude values for the model domain. sponge_vel is the global variable which contains the attenuated values for the velocities and will then be used in the same way the current version does to adjust the external velocities at the boundaries. Sometimes it is necessary to adjust coherently also the scalar fields of the *OCG* model. This is why the sponge_t matrix is also provided as an argument, with the same role of sponge_vel but for scalar fields.

In the following sections all the derived classes are discussed in details. They are rivers, sponge, closed and open, plus a fifth one, nudging, which implements the *decorator* pattern. A final remark on *destructors* is also provided.

4.2.3 rivers

`rivers` is the first class that inherits from `bc` implementing all the base class methods, resulting in a fully self-consistent boundary object. As the name says, it maps boundaries that in the boundary classification are defined as *rivers*. This means that its data files are supposed to contain the values of a potentially time dependent tracer flow at the boundaries, i.e. discharges of an amount of tracers from the rivers to the ocean, with a given seasonal variability. In the standard configuration of the model, a single set of files includes all the rivers that flow into the Mediterranean Sea at once. The class structure is of course flexible enough to allow to instantiate different objects for different subsets of rivers.

In addition to the base class member (i.e. the data object), the most important members added in this class are the followings:[3]

- `name`: just a string with the name, chosen by the user and passed to the constructor.
- `global_size` and `size`: respectively, global and local (per single *MPI* process) size of the boundary, i.e. number of cells to which a boundary value is assigned. These values are inferred directly from the boundary data files.
- `n_vars` and `var_names`: respectively, number and names of the tracers to which a boundary condition is assigned. Usually only a subset of the tracers that constitute the model output are set at the boundaries. The subset is set by the user and passed to the constructor.
- `values`: matrix with pre-computed time interpolated values that represent the final set of values to be assigned at the boundaries. They are not the final values of the tracer fields at the boundaries, which indeed is set by the `apply` method and depends on the boundary scheme.

Like in `bc` class, the constructor is overloaded through an interface, in order to handle both periodic and non periodic data; number and type of the arguments will decide which is to be called (code box 4.8).

LISTING 4.8: constructor overloading for the rivers class

```
interface rivers
    module procedure rivers_default
    module procedure rivers_year
end interface rivers
```

The constructors implementation is in code box 4.9 (only the default one is reported):

LISTING 4.9: default constructor for the rivers class

```
type(rivers) function rivers_default(files_namelist, &
        bc_name, n_vars, vars, var_names_idx)

    character(len=22), intent(in) :: files_namelist
    character(len=3) :: bc_name
```

[3] In the actual implementation, all the class member names use an identifier to distinguish them, e.g., from the corresponding names that are passed to the constructors or to other methods; here, for clarity's sake, only the meaningful part of the name is reported.

```
    integer, intent(in) :: n_vars
    character(len=23), intent(in) :: vars ! var_names
    integer(4), dimension(n_vars), intent(in) :: &
        var_names_idx

    ! parent class constructor
    rivers_default%bc = bc(files_namelist)

    call rivers_default%init_members( &
        bc_name, n_vars, vars, var_names_idx)

end function rivers_default
```

The first argument (`files_namelist`) is passed to the base class constructor. This call is actually the only part that needs to be modified when overloading the constructor for periodic and non periodic data. `bc` constructor is already overloaded, so the only change that needs to be done is adding two more arguments to the call, namely the start and the end time strings of the simulation. Everything else is common to the two constructors, and this is why they are implemented as *delegating costructors* [12]. An auxiliary method, defined *target constructor*, (`init_members`) is implemented and it is called in turn right after the call to the constructors. The *target constructor* is in charge of allocating and initializing all the members that are added to the base class. To do this, *netcdf* data files are read through the standard *netcdf-fortran* subroutines. Furthermore, a specific subroutine is defined in order to assign the boundary cells to the right *MPI* process. The logic behind similar operations and their implementation is nearly the same of the current version, with the substantial difference that now everything is unified and executed right when the boundaries are instantiated, just with one line of code containing the call to the constructor. Among the constructor arguments, the only one that has not been introduced so far is `var_names_idx`, an array containing basically the same information as the variable names one, just indexed with respect to the complete list of tracers (the information is obviously redundant, and has been added just for convenience's sake).

`rivers` class then implements all the methods that are declared but not implemented in the base class interface. `load`, `swap`, `actualize` and `apply` implementations are similar to those of version *3.2.1*. In particular, `apply` is simply adding the boundary constant tracer flow to the final tracer fields. `apply_nudging` is left unimplemented, since so far the model does not need this feature for this type of boundary. Neither `apply_phys` is implemented, in this case because *rivers* boundaries, by definition, are closed also in the *OGCM*; velocities are already set to zero and there is no point in modifying them at the boundaries.

4.2.4 sponge

The `sponge` class maps a boundary which is forced to be closed even if in the *OGCM* is not. It needs to know how to modify the velocities at the boundary; in particular it should be able to set them to zero at the boundary and to adapt them to the *OGCM* values according to a given function. Furthermore, maybe less important from a numerical point of view but still not negligible (for example for the short wave radiation), a correction to the scalar fields of the *OGCM* is also to be provided. Therefore, besides the members that are added to the base class also by the `rivers` class, i.e. `name`, `global_size`, `size`, `n_vars`, `var_names` and `values` (just to cite the

most important ones), sponge class introduces also a few members that are needed to parameterize the smoothing function for the velocities and to adjust the selected scalar fields. For the moment it has been adopted the same scheme of the current version, i.e. a correction within a certain distance form the boundary (namely length), a constant attenuation value for the scalar fields (reduction_value_t) and a Gaussian smoothing function for the velocities with variance governed by the parameter alpha.

sponge class data are supposed to be a set of values for a subset of tracers, assigned to a 3D region around the boundary. The reason behind this data scheme is to provide numerical stability to the model solution. Forcing the boundary to be closed, when in the *OGCM* is not, may result in unreliable solutions for the tracer fields near the boundary. The availability of a large set of tracers data in the boundary area, usually coming from the literature, allows *OGSTM* to bind its output to more trustworthy values. In this case, this is obtained through a nudging term, and this is why sponge class does not even implement the apply method. Only apply_nudging is implemented, and the class is thought to be used along with a nudging decorator (see 4.2.7).

Constructors are overloaded as usual to handle both periodic and non periodic data (code box 4.10).

LISTING 4.10: constructor overloading for the sponge class

```
interface sponge
    module procedure sponge_default
    module procedure sponge_year
end interface sponge
```

Exactly like in the rivers class the most of the initialization work is carried out by a *target constructor*. Its implementation, as long as those of load, swap and actualize, are nearly identical to the previous ones, except for some technicalities due to the fact than now values are set on a 3D domain. Finally, the apply_phys method is now fully implemented and is the one in charge of providing the modified values to adjust both the velocities and the scalar fields of the *OGCM*.

4.2.5 closed

closed class maps a boundary which is *closed* also for the *OGCM* and in which, unlike in *rivers*, a set of tracer values is provided instead of a set of fluxes. Normally, also in this case a *nudging* term is applied in order to determine reasonable values for the tracers in the proximity of the boundary. With the current code structure, *nudging* features come for free due to the nudging decorator (see 4.2.7). The implementation of closed is exactly the same as sponge, with the only difference that apply_phys is not implemented here, since no changes have to be applied to the *OGCM* outputs.

4.2.6 open

open class, instead, will implement an *open* boundary for *OGSTM*, given an open boundary in the *OGCM* too. It is not fully developed yet. In fact its implementation will differ from the previous ones mostly for what concerns the apply method and a debate on its actual physical implementation is still underway at the time of writing. The main difference is that in this case the tracer fluxes at the boundary can be determined as long as a 2D surface of tracers values is provided there. This is due to the non-null velocity fields at the boundary, which allows a direct computation of

the tracer flows. Having `nudging` as an optional feature is particularly useful in this case, since it can help in assuring numerical stability while the boundary condition parameters are still being fine-tuned.

4.2.7 nudging

`nudging` class is both inheriting from `bc` class and owning a pointer to a `bc` object (fig. 4.5).

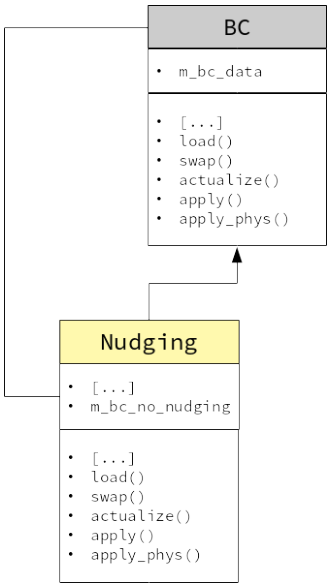

FIGURE 4.5: Decorator pattern for the nudging class.

This pattern is known as *decorator* and is a general pattern in Object-Oriented programming. Since it is inheriting from `bc`, it *is* a `bc`. Furthermore, associating its pointer to an already instantiated `bc` object of any kind (both of base class or any of the derived classes) it also *has* a `bc`, i.e. it can refer inside its methods directly to that object, *decorating* it with additional features. In this case this is particularly useful because, as stated in section 2.2, *nudging* features should be optional for every type of boundary. Using standard inheritance, this would have resulted in having twice the boundary classes seen so far. For example, it would have been necessary to implement both a `river` class and a `river_with_nudging` class, the same for the `sponge` class and so on. Using a decorator, instead, allows to instantiate an object of the desired type and, only if needed, to instantiate also a `nudging` object with its pointer associated to the first one. This is achieved by simply passing the first object to the `nudging` constructor and associating the member pointer accordingly, as can be seen from the implementation in code box 4.11.

LISTING 4.11: implementation of the decorator pattern for the nudging class

```
type, extends(bc) :: nudging

    class(bc), pointer :: m_bc_no_nudging => null()
    ! [...]

contains
```

```
    ! [...]

end type nudging

! [...]

type(nudging) function nudging_default(bc_no_nudging, ...)

    class(bc), target, intent(in) :: bc_no_nudging
    ! [...]

    ! parent class constructor
    nudging_default%bc = bc()

    call nudging_default%init_members(bc_no_nudging, ...)

end function nudging_default

! [...]

subroutine init_members(self, bc_no_nudging, ...)

    class(nudging), intent(inout) :: self
    class(bc), target, intent(in) :: bc_no_nudging
    ! [...]

    ! pointer to bc_no_nudging association
    self%m_bc_no_nudging => bc_no_nudging

end subroutine init_members
```

The `class` keyword, used for the member pointer, enables object *polymorphism*, allowing the pointer to a `bc` object to be associated to any object that inherits from the `bc` class. Note also that the base class constructor that is called in the `nudging` decorator is the empty one. In this way no dataset is associated to the nudging decorator, since a dataset is already associated to the decorated object through its own constructor.

The main members that are added to the base class, besides the pointer to the `bc` object, are the followings:

- `data_file`: a *netcdf4* file with restoration coefficients and other *nudging* related parameters, passed to the constructor.
- `n_nudging_vars` and `nudging_vars`: respectively, number and names of the tracers to which the *nudging* scheme is to be applied. Usually they coincide with the subset of tracers to which the boundary condition is applied, but this is not strictly necessary. The list of values is set by the user and passed to the constructor.
- `rst_tracers`: final matrix with the restoration values, which is used to apply the nudging to the tracer fields at the boundary.

All the methods are just wrappers of the decorated object methods, with the exception of the `apply` method (which calls the `apply_nudging` method of the decorated object, passing its `rst_tracers` matrix as an argument) and the `apply_nudging` one, which is just not implemented since there will be no need to apply an additional *nudging* to a `nudging` object.

4.2.8 Destructors

A destructor is needed to explicitly deallocate what has been allocated in the heap by the constructor, to deassociate pointers and in general to do what needs to be done when the object is destroyed in order to avoid memory leaks. Fortran provides a keyword, `final`, which should let the destructor be called automatically any time the object itself is deallocated. Unfortunately, Fortran 2003 does not provide full support for this feature [9] and in this version of the model destructors are just declared and implemented as ordinary methods. They need therefore to be called explicitly any time an object is going to be deallocated, so both when this is done explicitly for heap allocation and when the object is going out of scope.

Chapter 5

Improvements in the new version and benchmarks

In the previous chapter, the implementation of the new Object-Oriented structure has been introduced and discussed. Once the new classes have been implemented, the current version of the code has to be modified any time an action on the boundary condition is required. The idea indeed is to substitute the calls to the previous subroutines for the boundary conditions with calls the new methods. In the following section these changes will be introduced in detail. Furthermore, in section 5.2 some profiling data and benchmarks are discussed in order to give also a quantitative description of the benefits of the new implementation.

5.1 Changes in the main code

Once a complete Object-Oriented structure is provided, only a few lines have to be added to the main modules in order to perform all the boundary-related operations, from instantiating the objects to computing the boundary values and updating the output fields, up to deallocating the memory.

All the boundary-related code in the current version modules and subroutines has been removed in the new version, and replaced with calls to the new methods which can be essentially grouped into four parts:

- declaration and instantiation;
- boundary values update;
- boundary condition application;
- deallocation.

5.1.1 Declaration and instantiation

For every requested boundary condition, the corresponding object is first declared and then instantiated. Here is an example of the objects needed for the CMEMS configuration on the Mediterranean Sea. In this case three boundary objects are required: `all_rivers`, which takes care of all the rivers at once, `gibraltar_sponge`, which is supposed to be a sponge boundary at Gibraltar and `gibraltar`, which is the whole boundary at Gibraltar, decorated with a *nudging*. In the example in code box 5.1, for every boundary, the default constructor is called, i.e. the non periodic one.

LISTING 5.1: calls to the boundary condition constructors in the new version

```
type(rivers), pointer :: all_rivers => null()
type(sponge), pointer :: gibraltar_sponge => null()
type(nudging), pointer :: gibraltar => null()

allocate(all_rivers)
allocate(gibraltar_sponge)
allocate(gibraltar)

all_rivers = rivers( &
    "files_namelist_riv.dat", &
    "riv", &
    6, &
    "N1p␣N3n␣N5s␣O3c␣O3h␣O2o", &
    (/2, 3, 6, 49, 50, 1/) &
    )
gibraltar_sponge = sponge( &
    "files_namelist_gib.dat", &
    "gib", &
    7, &
    "O2o␣N1p␣N3n␣N5s␣O3c␣O3h␣N6r", &
    (/1, 2, 3, 6, 49, 50, 7/), &
    1.0d0, &
    1.0d-6, &
    -7.5d0 &
    )
gibraltar = nudging( &
    gibraltar_sponge, &
    "bounmask.nc", &
    7, &
    "O2o␣N1p␣N3n␣N5s␣O3c␣O3h␣N6r", &
    (/1, 2, 3, 6, 49, 50, 7/), &
    (/1.0d0, 1.0d0, 1.0d0, 1.0d0, 2.0d0, 2.0d0, 2.0d0/), &
    51&
    )
```

These lines of code replace actually the list of matrices and auxiliary variables that in the current version are declared and allocated for each boundary, along with the hard-coded lists of tracers, and all the auxiliary methods that in the current version are called explicitly and repeated for each boundary. Among them are for example the subroutines to handle *netcdf* files, those to compute the local size of the MPI processes etc.

5.1.2 Boundary values update

A combination of `load`, `swap` and `actualize` methods is required at every time step in order to interpolate the boundary data of the right time interval. In the current version a subroutine to update the boundary data at every time step is provided for each boundary. Its name changes depending on the boundary type, but inside it the three methods are called using the same logic. The new version is providing

a much shorter syntax, due both to the fact that `load`, `swap` and `actualize` are now declared in the parent class and overridden by every derived class, and to object *polymorphism*. A unique procedure, called `update_bc`, is provided to update the data of any boundary type. The procedure is contained inside a module, which defines also a pointer to a generic `bc` class object. The boundary which needs to be updated is supposed to be already instantiated. It is passed as an argument to `update_bc` and here declared as a target belonging itself to the `bc` class. After associating the module pointer to the desired object, any of the base class methods can be called on the pointer (and so on the object), regardless of the specific type. The same procedure can thus be used for every boundary object inheriting from the base class, improving a lot both the compactness and the maintainability of the code. The structure of the `update_bc` subroutine is reported in code box 5.2. Note also how, besides the three aforementioned methods, also methods referring to the data files (`get_prev_idx`, `get_next_idx`, `get_interpolation_factor`, ...) are overridden and used here. They are implemented once in the base class and are owned by definition by every inheriting class.

LISTING 5.2: implementation of the update subroutine in the new version

```
module bc_update_mod

    ! [...]

    class(bc), pointer :: m_bc => null()

contains

    subroutine update_bc(bc_iter, ...)

        class(bc), target, intent(inout) :: bc_iter
        ! [...]

        m_bc => bc_iter

        ! [...]

        weight = m_bc%get_interpolation_factor(...)

        ! [...]

        call m_bc%load(m_bc%get_prev_idx())
        call m_bc%swap()
        call m_bc%load(m_bc%get_next_idx())

        ! [...]

        call m_bc%actualize(weight)

    end subroutine update_bc

end module bc_update_mod
```

The logic used in the implementation of the `bc_update` module relies on the same concept of the *factory* design pattern [13]. The *factory* pattern is used to instantiate an object of any class that inherits from the same base class. A pointer to the base class is associated to a new object of the desired type, through a call to the specific constructor, and then returned. In a `factory` class this approach is used specifically to create new objects, but it can be of course generalized to other class methods, and this is exactly what has been done with the `update` method. In principle, any call to the boundary methods could be handled in this way, even though it may not always be the best choice. For example, this approach is useful when updating the boundary values, since using overridden methods simplifies a lot the implementation, as seen above. On the contrary, it can result more convoluted if used with constructors. In this case the factory method has to know how to construct each type of boundary, i.e. it has to call the right constructor. Now, different constructors require potentially a different list of arguments, both in number and in types. Deciding which arguments must be passed to the *factory* method may not be straightforward and anyway not easier than calling the constructors separately, which indeed is what has been established in the new version.

Once defined the new subroutine, the calls to the previous ones are replaced by the lines reported in code box 5.3.

LISTING 5.3: calls to the update subroutine in the new version

```
call update_bc(all_rivers, ...)
call update_bc(gibraltar, ...)
```

5.1.3 Boundary condition application

The section in which the boundary conditions for the tracers are actually applied on the final tracers matrix is straightforwardly modified. The involved subroutine is `trcdmp`. Here for each boundary two explicit nested loops are in charge of setting the final values in the tracers matrix. All that is needed is to replace each of them with a call to the `apply` method. In code box 5.4 there is an example for the configuration with two boundaries (rivers and Gibraltar).

LISTING 5.4: boundary conditions application in the new version

```
call all_rivers%apply(...)
call gibraltar%apply(...)
```

In case of a *sponge* boundary, also the fields of the *OGCM* need to be modified. Differently from the `apply` method, this operation has to be performed just once, during the initialization phase. After that a global matrix with the *sponge* correction terms is set and it is used at every time step to adjust the current data coming from the *OGCM*. Again, for each sponge boundary, the three nested loops which are setting the final matrix have to be replaced by just one single call to the `apply_phys` method (code box 5.5).

LISTING 5.5: call to the apply_phys method in the new version

```
call gibraltar%apply_phys(...)
```

5.1.4 Deallocation

Object destructors need to be called inside the `ogstm_finalize` subroutine in order to deallocate everything that has been allocated in the *heap*. A destructor is called for each instantiated object, as shown in code box 5.6.

LISTING 5.6: memory deallocation in the new version

```
call all_rivers%rivers_destructor()
call gibraltar_sponge%sponge_destructor()
call gibraltar%nudging_destructor()
```

Note that the order in which the destructors are called is the inverse of the instantiation order. This is to avoid bad pointers association and corrupted objects. For example, given a *sponge* decorated with a *nudging*, the *sponge* constructor is called before the nudging *constructor*. Attempting to destruct the *sponge* first would leave the *nudging* decorator in a corrupted state since its inner pointer would be pointing to a non-existing object and its methods, which are relying on it, would probably show undesired behaviors.

5.2 Profiling and benchmarks

The motivation behind this work is to build a flexible and versatile interface to assign and handle the boundary conditions. As a consequence, a quantitative analysis of the results shall not rely (at least not for the most part) on traditional metrics such as for example execution time or scalability. In an ensemble simulations scenario, in which boundary condition of the same type and with the same geometry are slightly varied, or new boundary conditions are added to the model, it is fundamental to minimize the impact of such perturbations on the model configuration. The current time to solution is dramatically increased due to the time spent in updating or adding new subroutines to the model, allocating new matrices etc. Such operations require both time resources and developers with a solid knowledge of the code. The final goal of the new version, instead, is to let users themselves change the boundary configuration, enabling the model to be run potentially in any domain and by a wider community.

5.2.1 Overall time to solution

Table 5.1 gives an as much quantitative as possible overview of the progress in this sense. It describes, for every possible change of the boundary configuration, the effort which is required to set up the model. Note that the focus of this analysis is the effort spent in refactoring the code, and not, for example, in preparing new input files for the boundaries or change the values of the data files. This activity is out of the scope of this work, but anyway it is worth to notice that new datasets are not necessarily written by the user, since for example they can be themselves the outputs of a coupled global model. The first two columns report the effort respectively for the current and the new version. A third column, namely *New ++* is also provided with the effort for a slightly different implementation of the new version, in which the constructors parameters are read from a namelist file and the methods are called inside a loop over all the boundaries. These additional features are not enabled yet, just because the current configurations of the model require only a few boundaries and the code is more readable without additional changes. Applying them is

straightforward and has the advantage of moving all the configurations to external namelist files, allowing to run different simulations without even recompiling the code.

TABLE 5.1: Comparison of the effort required to modify the boundary conditions for three versions of the code: current one, new one and new with additional namelist files and for loops over the boundaries. The generic term *lines* refers both to modified, added or removed lines. Note that the new version provides the option to apply or not the *nudging*.

	Current	**New**	**New ++**
Modify a BC	$\leq$ 30 lines	$\leq$ 2 lines	namelist
Add a new BC	$\simeq$ 300 to 400 lines	6 to 10 lines	namelist
Add / remove *nudging*	not allowed	$\leq$ 7 lines	namelist

5.2.2 Profiling

Tables 5.2 and 5.3 show the result of a simple profiling of the code with respect to the execution times of its different modules. As stated before, the aim of the work is not actually to improve the run-time of the model. Even in the current version, indeed, all the procedures related to the boundary conditions are taking a few orders of magnitude less time than the most computational intensive ones (such as the subroutines for the advection and the diffusion). Any possible improvement in their performances would therefore result imperceptible if compared with the overall run-time. Nevertheless, it is important to check whether with the new implementation the timings of the single procedures are still comparable or, for example, the new structure introduces new bottlenecks in the system.

A few tests have been run using a dataset with the resolution of 0.25° both in latitude and in longitude and one computational node of the KNL partition of CINECA, with 59 MPI processes. The number of processes is the result of an algorithm for the domain decomposition. Its aim is both to get as close as possible to the desired number of processes (which in this case has been set to 68, i.e. the available cores on a KNL partition) and to minimize the land / water ratio in the computational domains. The length of the simulation has been set to 2 days, i.e. 96 time steps. Two simulations have been run, first using the current version and then the new one. The methods and subroutines of interest are those which update the two boundary conditions at the rivers and at the Gibraltar strait (they are called in both cases with the original names `bcTIN` and `bcGIB`, but in the new code they refer to the newly implemented `update_bc`). For a time comparison also the `trcstp` and `stp` subroutines have been reported. The first one is in charge of solving both advection and diffusion, while the second one computes a full time step. All the aforementioned subroutines are called inside the `stp` one, and therefore its run-time will be greater or equal than the sum of the previous ones. The results are reported separately in table 5.2 and 5.3.

The resulting times are an average for the same subroutines over all the MPI processes. The possible choices here were at least three. One could have run a serial version of the code, but this would have lead to a very unrealistic domain, with only one rectangle covering the whole Mediterranean and a lot of land regions in it. The second choice would have been to consider the maximum time over all the MPI

process instead of the average. Opting for the average, however, has the advantage of focusing on how good a subroutine is performing overall, averaging the effects of a potential unbalance in the MPI processes, which is out of the scope of this work.

TABLE 5.2: subroutines elapsed times for current version: boundaries and main integration step

Subroutine	**Overall time (s)**	**Single call time (s)**
bcTIN	2.78 e-01	2.90 e-03
bcGIB	2.39 e+00	2.49 e-02
bcATM	4.12 e-02	4.30 e-04
bcCO2	1.29 e-02	1.34 e-04
trcstp	2.53 e+02	2.63 e+00
stp	3.14 e+02	3.27 e+00

TABLE 5.3: subroutines elapsed times for new version: boundaries and main integration step

Subroutine	**Overall time (s)**	**Single call time (s)**
bcTIN	2.36 e-01	2.45 e-03
bcGIB	3.61 e-01	3.76 e-03
bcATM	2.82 e-02	2.94 e-04
bcCO2	1.58 e-02	1.64 e-04
trcstp	2.54 e+02	2.64 e+00
stp	3.08 e+02	3.21 e+00

It can be seen from the profiling data how nothing changed in terms of ratio between run-times of the boundary conditions subroutines and the core ones. Actually, in the new version, the Gibraltar (i.e. a *sponge* decorated with a *nudging*) update subroutine execution time is even an order of magnitude less than in the current version. Most likely, this is the result of some optimization which has been applied when rewriting the methods. For example the time interpolation procedure has been considerably simplified due to the simpler structure for the data files; furthermore, also the workflow of the updating subroutine is more linear, with an optimized number of branches. This is an example of how a different design pattern, besides providing more flexibility, can also enhance the code performances.

Chapter 6

Conclusions and future work

The main results of this work can be summarized as follows:

- boundary conditions modules have been fully rewritten in an Object-Oriented style. In this way the structure is completely flexible with respect both to the configuration of any boundary and to the addition of new boundaries of any type that falls into the classification of chapter 3;
- the new design does not add any bottlenecks to the current version and, instead, the execution time for updating the boundaries at each time step is decreased by one order of magnitude;
- High Performance Computing is applied by dramatically reducing the time to solution. No more hard coding on the main code is required, and only very few lines of code (ore even none with a few further configurations) need to be modified for configuring the boundaries in the desired way, providing better maintenance and portability to the code;
- a more flexible management of data files has been achieved. Data periodicity is now fully enclosed in a dedicated class and any time distribution of the data files across the simulation time is supported.

This new structure opens the way to a lot of possible applications, any of which will probably require a comparable effort with respect to this thesis work. The key feature that is now enabled is the possibility to run ensemble simulations, and this is a base ground for many further studies, such as:

- implementing a flexible sensitivity/calibration framework to get more accurate model output;
- coupling the Mediterranean forecast system with global or adjacent CMEMS forecast systems;
- running an uncertainty quantification analysis;
- calibrating the model with a Reduced Basis approach.

Appendix A

Unit testing framework

The unit testing framework used in this project is *pFUnit 3.2.9* [14]. *pFUnit* enables *JUnit*-like testing of Fortran software and was originally created by developers from *NASA* and *NGC TASC*. It has been chosen as the default tool for the unit tests of the new version of the code essentially for two reasons:

- it is written in modern Fortran and not only it is compliant with modern Fortran programming techniques (including Object Oriented programming), but it makes also use of them itself;
- it supports both serial and *MPI* parallel unit testing.

The working principle is simple and it is better described through an example of a serial unit test taken from the project. Writing a new test requires to write a new class that extends one of the pre-built *pFUnit* classes. For example, with reference to code box A.1, the new test class here defined inherits from the `TestCase` class. `TestCase` is useful when multiple tests can share the same testing environment, like in this case, when, after instantiating an object of the `bc` class, multiple tests are performed on it.

LISTING A.1: test case for the bc class

```
module test_bc_default_mod

    use bc_mod
    use pfunit_mod

    implicit none

    public :: test_bc_default

@TestCase
    type, extends(TestCase) :: test_bc_default
        type(bc), pointer :: m_bc => null()
    contains
        procedure :: setUp ! overrrides generic
        procedure :: tearDown ! overrides generic
    end type test_bc_default

contains

    subroutine setUp(this)
        class(test_bc_default), intent(inout) :: this
```

```
        allocate(this%m_bc)
        this%m_bc = bc("files_namelist_gib.dat")
    end subroutine setUp

    subroutine tearDown(this)

        class(test_bc_default), intent(inout) :: this

        ! explicitly call destructor before deallocating
        call this%m_bc%bc_destructor()

        deallocate(this%m_bc)
        write(*, *) 'INFO: m_bc_deallocated'
        nullify(this%m_bc)
        write(*, *) 'INFO: m_bc_deassociated'

    end subroutine tearDown

@Test
    subroutine test_file_names(this)
        class(test_bc_default), intent(inout) :: this
        @assertEqual("GIB_20170215-12:00:00.nc", &
            this%m_bc%get_file_by_index(1))
        @assertEqual("GIB_20170515-12:00:00.nc", &
            this%m_bc%get_file_by_index(2))
        @assertEqual("GIB_20170815-12:00:00.nc", &
            this%m_bc%get_file_by_index(3))
        @assertEqual("GIB_20171115-12:00:00.nc", &
            this%m_bc%get_file_by_index(4))
    end subroutine test_file_names

@Test
    subroutine test_new_data(this)
        class(test_bc_default), intent(inout) :: this
        double precision :: interpolation_factor
        logical :: new_data
        interpolation_factor = &
            this%m_bc%get_interpolation_factor( &
                "20170814-00:00:00", new_data)
        @assertTrue(new_data, "should be new data")
        interpolation_factor = &
            this%m_bc%get_interpolation_factor( &
                "20170815-00:00:00", new_data)
        @assertFalse(new_data, "should be same data")

        ! [...]

    end subroutine test_new_data

end module test_bc_default_mod
```

With reference to the code, the new object defined here is `test_bc_default`, which extends `TestCase` and adds as a member a pointer to a `bc` class object. Two type-bound procedures, `setUp` and `tearDown`, which actually plays the role respectively of the constructor and the destructor, need to be overridden. In particular, here the role of the constructor is to instantiate the `bc` object. All the needed tests are then annotated with the `@Test` keyword and defined as type-bound procedures, which can in turn refer to the `bc` object and call any of its methods. Many assert directives are provided, as for example `assertEqual`, `assertTrue` and `assertFalse`.

Before being compiled, the code needs to be given as an input to a Python script, included in the distribution, which basically parses the annotations and provides the final source code. A configuration file, `testSuites.inc`, is used to tell the *pFUnit* makefile which test modules to include in the final executable. *pFUnit* output provides then a detailed stacktrace for every encountered failure.

Bibliography

[1] G. Cossarini, S. Salon, G. Bolzon, A. Teruzzi, P. Lazzari, E. Clementi, *Mediterranean Sea biogeochemistry reanalysis - Quality information document*, Copernicus Marine Environment Monitoring Service, 2017.

[2] *BFM* model website, *http://bfm-community.eu/*

[3] G. Cossarini, S. Querin, C. Solidoro, G. Sannino, P. Lazzari, V. Di Biagio, G. Bolzon, *Development of BFMCOUPLER (v1.0), the coupling scheme that links the MITgcm and BFM models for ocean biogeochemistry simulations*, Geosci. Model Dev., **10**, 2017, 1423–1445.

[4] *NEMO* model website, *https://www.nemo-ocean.eu/*

[5] I. E. Huertas, A. F. Ríos, J. García-Lafuente, A. Makaoui, S. Rodríguez-Gálvez, A. Sánchez-Román, A. Orbi, J. Ruíz, F. F. Pérez, *Anthropogenic and natural CO2 exchange through the Strait of Gibraltar*, Biogeosciences, **6**, 2009, 647-662.

[6] M. de la Paz, E. M. Huertas, X. A. Padín, M. Gónzalez-Dávila, M. Santana-Casiano, J. M. Forja, A. Orbi, F. F. Pérez, A. F. Ríos, *Reconstruction of the seasonal cycle of air-sea CO2 fluxes in the Strait of Gibraltar*, In Marine Chemistry, **126**, Issues 1-4, 2011, 155-162.

[7] M. Álvarez, H. Sanleón-Bartolomé, T. Tanhua, L. Mintrop, A. Luchetta, C. Cantoni, K. Schroeder, G. Civitarese, *The CO2 system in the Mediterranean Sea: a basin wide perspective*, Ocean Science, **10**(1), 2014, 69-92.

[8] C++ documentation website, *https://en.cppreference.com/w/cpp/language/raii*

[9] K. Holcomb, *Scientific programming in Fortran 2003 - A tutorial including Object-Oriented Programming*, University of Virginia, 2012.

[10] C++ documentation website, *https://en.cppreference.com/w/cpp/language/pimpl*

[11] Intel Developer Zone, forum topic, *https://software.intel.com/en-us/forums/intel-visual-fortran-compiler-for-windows/topic/559996*

[12] C++ documentation website, *https://en.cppreference.com/w/cpp/language/initializer_list*

[13] Fortran Wiki, *http://fortranwiki.org/fortran/show/Factory+Pattern*

[14] pFUnit project website, *http://pfunit.sourceforge.net/*