Geosci. Model Dev., 10, 3167–3187, 2017
https://doi.org/10.5194/gmd-10-3167-2017

Geoscientific Model Development
Open Access
EGU

The Oceanographic Multipurpose Software Environment (OMUSE v1.0)

Inti Pelupessy[1,2,3], Ben van Werkhoven[4], Arjen van Elteren[3], Jan Viebahn[1,2], Adam Candy[5], Simon Portegies Zwart[3], and Henk Dijkstra[1]

[1]Institute for Marine and Atmospheric Research, Utrecht University, Utrecht, the Netherlands
[2]Centrum Wiskunde & Informatica, Amsterdam, the Netherlands
[3]Leiden Observatory, Leiden University, Leiden, the Netherlands
[4]The Netherlands eScience Center, Amsterdam, the Netherlands
[5]Civil Engineering and Geosciences, Delft Technical University, Delft, the Netherlands

Correspondence to: Inti Pelupessy (f.i.pelupessy@cwi.nl)

Received: 8 July 2016 – Discussion started: 7 September 2016
Revised: 16 June 2017 – Accepted: 10 July 2017 – Published: 28 August 2017

Abstract. In this paper we present the Oceanographic Multipurpose Software Environment (OMUSE). OMUSE aims to provide a homogeneous environment for existing or newly developed numerical ocean simulation codes, simplifying their use and deployment. In this way, numerical experiments that combine ocean models representing different physics or spanning different ranges of physical scales can be easily designed. Rapid development of simulation models is made possible through the creation of simple high-level scripts. The low-level core of the abstraction in OMUSE is designed to deploy these simulations efficiently on heterogeneous high-performance computing resources. Cross-verification of simulation models with different codes and numerical methods is facilitated by the unified interface that OMUSE provides. Reproducibility in numerical experiments is fostered by allowing complex numerical experiments to be expressed in portable scripts that conform to a common OMUSE interface. Here, we present the design of OMUSE as well as the modules and model components currently included, which range from a simple conceptual quasi-geostrophic solver to the global circulation model POP (Parallel Ocean Program). The uniform access to the codes' simulation state and the extensive automation of data transfer and conversion operations aids the implementation of model couplings. We discuss the types of couplings that can be implemented using OMUSE. We also present example applications that demonstrate the straightforward model initialization and the concurrent use of data analysis tools on a running model. We give examples of multiscale and multiphysics simulations by embedding a regional ocean model into a global ocean model and by coupling a surface wave propagation model with a coastal circulation model.

1 Introduction

Numerical models of the global open ocean have now reached a mature state. Models such as the MIT Global Circulation Model (MITgcm), the Modular Ocean Model (MOM), the Max Planck Institute Ocean Model (MPIOM), the Parallel Ocean Program (POP) and Nucleus for European Modeling of the Ocean (NEMO) are widely used in the community. These models can be used as the ocean components in coupled global climate models such as those in the Coupled Model Intercomparison Project[1]. Such simulations, with horizontal resolutions as fine as 25 km, focus on projected forecasts of future climate change (IPCC, 2013). The models are also used in an ocean-only model configuration (Maltrud et al., 2010) at even higher resolutions (down to about 10 km) to adequately resolve western boundary currents, such as the Gulf Stream, the Agulhas Current and Kuroshio, and to explicitly represent mesoscale eddies.

At the coastal zone, very different models are required, incorporating, for example, tides, river run-off, sediment transport and wave dynamics (e.g., Zijlema, 2010). In many cases,

[1]http://pcmdi-cmip.llnl.gov

unstructured mesh models are used (Danilov, 2013; Leuttich and Westerink, 2004) in order to provide an accurate representation (Candy et al., 2014) of complex and irregular domain bounds that strongly influence local flows. An additional challenge in regional models, such as ADCIRC (Advanced 3-D Circulation) and SWAN (Simulating Waves Nearshore), is that they are not bounded entirely by a coastline and typically contain at least one boundary open to the global ocean. These open-ocean boundaries are usually handled by restoring them to observations (climatology or transient over a specific period in the past).

In order to evaluate the human-scale impacts of climate change, for example the effect of sea level rise on coastal erosion (Cazenave, 2004), both the open ocean and coastal zone need to be jointly considered. Increasing temperatures and the changes in wind field can give rise to changes in ocean currents, which in turn cause dynamical changes in sea level (Brunnabend et al., 2014). These conditions will affect the wave climate and may lead to changes in erosion at sandy coasts. To tackle such problems one could proceed by developing a single code to incorporate both regimes (a monolithic approach), or one can try to nest an existing regional model into a global ocean model (using a tool such as AGRIF[2] to take care of the transport of data between grids). The extension of this idea to different codes and to different components of Earth system and climate models suggests a modular approach (e.g., Valcke et al., 2012), where one develops an efficient way to couple different (e.g., open and coastal, but potentially more than two) models together.

In this paper, we follow the latter approach, borrowing from ideas in the astrophysical community. In simulations of the formation of stars and galaxies, a wide variety of codes need to be combined. For example, hydrodynamic codes (describing interstellar gas dynamics) are coupled with N-body codes (for the gravitational dynamics of stars), and processes on different scales, ranging from planetary to galactic, compete to determine the evolution of the coupled system. Given the need to correctly capture the interactions of the processes represented in the different codes, the community has come up with a Python framework[3] (AMUSE) allowing easy interaction of different codes (Portegies Zwart et al., 2013; Pelupessy et al., 2013).

In oceanography similar problems for multiscale and multiphysics are encountered, and a number of coupling frameworks exists in the Earth system modeling community (e.g., Hill et al., 2004; Buis et al., 2006; Gregersen et al., 2007; Jacob et al., 2005; Larson, 2005; Peckham et al., 2013; Valcke, 2013). These can be roughly divided into *integrated* and *coupling library* approaches (Valcke et al., 2012). In the integrated approach, the functionality provided by the component codes (e.g., by subroutines of the code) is separated out and joined into a new coupled model in a single executable. In the library approach, the original codes themselves are adapted to communicate with each other using an application programming interface (API) made available by the coupling library.

The AMUSE package provides a useful alternative since it takes the approach of integrating different codes in a high-level programming language (Python), using physically motivated programming interfaces to communicate with separately running instances of the simulation codes. This has the benefit of the parallelism and flexibility provided by a coupling library approach, and the benefit of abstracting much of the bookkeeping inherent to code couplings using modern high-level constructs. In this way, quite complex simulations can be described in compact scripts, that can be easily understood and easily distributed.

The aim of this paper is to present OMUSE, a framework which adapts the AMUSE approach for use in the Earth system modeling community, with an initial focus on oceanography. In Sect. 2, the design and architecture of OMUSE is presented, with a particular focus on data structures, unit conversion and grid remapping. The initial set of codes included is presented in Sect. 3. In Sect. 4 we discuss the code-coupling features of the OMUSE framework with particular emphasis on a quasi-geostrophic model as a conceptual test case. In Sect. 5, we present simple applications of OMUSE showing its capabilities. A summary and discussion of these results concludes the paper (Sect. 6).

2 Design and architecture

As inherited from AMUSE, the basic idea of OMUSE is the abstraction of the functionality of simulation codes (*the community code base*) into physically motivated interfaces that hide their complexity and numerical implementation. OMUSE provides the user optimized building blocks that can be combined to design numerical experiments. The requirement of the high-level glue language is not so much performance, but one of algorithmic flexibility and ease of programming. Hence, a modern interpreted scripting language with object-oriented features, in our case Python (van Rossum, 1995), is the natural choice. Furthermore, Python has a large user and developer base in scientific computing, and many libraries are available. Amongst these are libraries for numerical computations, data analysis and visualization, which can be used in an OMUSE scripts.

An OMUSE application consists, roughly speaking, of a *user script*, an *interface layer* and the *community code base* (Pelupessy et al., 2013), as illustrated in Fig. 1. The user script is constructed by the user and defines a numerical experiment by specifying the initial data, the simulation codes

[2] Adaptive Grid Refinement In Fortran, http://www-ljk.imag.fr/MOISE/AGRIF/

[3] *framework* as used here refers to an ensemble of tools, application programming interfaces and libraries which together can be used to construct new applications, in our case scientific simulations.

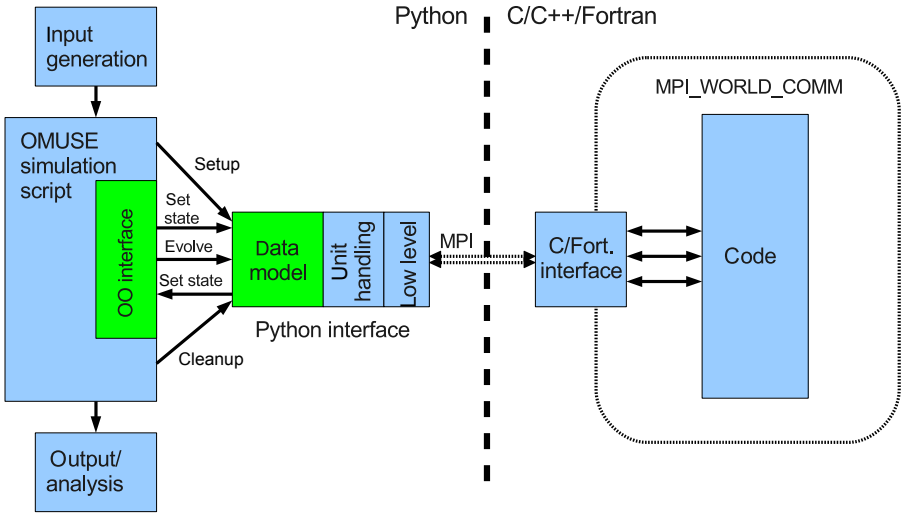


Figure 1. Design of the OMUSE framework. This schematic representation shows the design of the interface to a community code ("code") and the way it is accessed from the OMUSE framework. The code has a thin layer of interface functions in its native language (e.g., Fortran), which communicates through an MPI message channel with the Python host process. On the Python side, the user script ("OMUSE simulation script") makes only generic calls to a high-level interface. This high-level interface calls the low-level interface functions, hiding details about units and the code implementation (the communication through the MPI channel does not interfere with the code's own parallelization because the latter has its own MPI_WORLD_COMM context). Adapted from Pelupessy et al. (2013).

to be used and the interactions between the codes. It may include analysis or plotting functions, in addition to writing simulation data to file. The setup and communication with a community code is handled by the framework in the interface layer, which consists of a communication interface with the community code as well as unit handling facilities and an object-oriented interface. The interface layer also ensures the consistency of the interactions with the various simulation codes by maintaining a state model for each.

Below we give an overview of the design and architecture of OMUSE (as inherited from AMUSE; more details can be found in Pelupessy et al., 2013). The main developments compared with AMUSE, apart from the addition of oceanographic codes, are improvements in grid support, amongst these support for curvilinear grids and extensive framework support for grid remappings and grid generation routines. In addition, a number of domain-specific units and utility libraries and support for various file formats, such as NetCDF (Rew and Davis, 1990) output, have been added.

2.1 Remote function interface

The interface to a community code is provided by a set of functions, each communicating with the code through a remote function protocol. Currently the default implementation in OMUSE of this remote function protocol is based on the message passing interface (MPI). A community code is started by the instantiation of an interface object (Fig. 2), transparent to this. Python provides the possibility of linking Fortran or C/C++ codes directly; however, we found that

```
(1) qg=QG()
(2) qg=QG(debugger="gdb")
(3) pop=POP(number_of_workers=8)
(4) pop=POP(channel_type="distributed", hostname="Cartesius",
    number_of_workers=600)
```

Figure 2. Examples of the instantiation of simulation codes within OMUSE. (1) Simple instantiation on a local machine of the QG code, (2) instantiation of a code inside a debugger, (3) local instantiation of an MPI-parallel code (POP), (4) instantiation of POP on a remote machine for a massively parallel high-resolution run through the *distributed* channel (see Sect. 2.2).

a remote protocol provides two important benefits. First, it provides for build-in parallelism (this parallelism is exploited in the current setup for running the codes, although the data transport between codes is not yet fully parallel; see Sect. 6). The choice for a parallel interface means that it becomes possible to run and communicate with parallel running instances of different codes. In addition, a lot of existing simulation codes cannot handle multiple simultaneous instances of the same code. They may, for example, use global variables or assume a single global state. Using our approach, it is trivial to run separate instances of the same code, even if compiled with different options (in addition to this, the fact that codes are running as different processes prevents collisions between incompatible libraries when codes are built with different compilers).

Within the remote data communication channel, the MPI protocol can be replaced by a different method, two of which

are currently available: a channel based on sockets and one based on eStep[4] technology for distributed computing. At present, the sockets channel is mainly useful for cases in which a component process is started on the same machine as the user script. As its name implies, the sockets channel is based on standard TCP/IP (transmission control protocol/internet protocol) sockets. The distributed channel is described in Sect. 2.2 below. When using the MPI channel, different MPI implementations can be used (e.g., OpenMPI or MPICH), but not mixed (vendor implementations can also be used, although these sometimes do not fully conform to the standard).

The interface works as follows: when an instance of an imported simulation code is made, an MPI process is spawned as a separate process somewhere in the MPI cluster environment. This process consists of a simple event loop that waits for a message from the Python side. It will make the requested simulation code subroutine calls, on the basis of the incoming message ID as well as any additional data that may follow the initial MPI message, and subsequently send the results back (Portegies Zwart et al., 2013). Since there is no direct memory access, the interfaces themselves must be carefully designed to ensure all necessary information for a given physical domain can be retrieved.

Note that the interface design allows the parallelism of MPI parallel codes to be maintained even when the communication channel uses MPI (OMUSE can be used to run massively parallel codes with thousands of processes). This is guaranteed with the recursive parallelism mechanism in MPI-2. The spawned processes share a standard MPI_WORLD_COMM context, which ensures that an interface can be build around an existing MPI code with minimal adaptation (Fig. 1). Other parallelization paradigms, such as OpenMP[5], are also supported within OMUSE. In practice, for the implementation of the interface to an MPI code, one has to take into account similar issues as for the stand-alone MPI application. The socket and distributed channels also accommodate MPI parallel processes. The choice between the different available channels depends on the computing resources needed for a given run. For runs distributed over remote machines the distributed channel may be required, while locally on a cluster the MPI channel often provides the most optimized communication path.

2.2 Distributed computing

Current computing resources available to researchers are more diverse than simple workstations: clusters, clouds, grids, desktop grids, supercomputers and mobile devices complement stand-alone workstations, and in practice one may want to take advantage of this ecosystem.

[4]http://estep.esciencecenter.nl

[5]Open Multi-Processing, a shared memory multiprocessing API

```
(1) q = 1. | units.Sv
    dt= 1. | units.day
(2) (q*dt).as_quantity_in(units.m**3)
(3) (q*dt).value_in(units.km**3)
(4) def Reynolds_number(vel, length, visc):
        return vel*length / visc
(5) R = Reynolds_number( 0.1 | units.cm/units.s, 1000. | units.km,
        1.e-6 units.m**2/units.s)
```

Figure 3. An illustration of the use of the OMUSE unit algebra module, with (1) a definition of a scalar quantity using the | operator, (2) conversion of a quantity to different units, (3) conversion of quantity to float, and (4 and 5) definition of a function and its call using quantities.

To run in such a "jungle computing environment" (Seinstra et al., 2011), OMUSE also implements a communication channel based on eStep technology (Drost et al., 2012). This channel starts a daemon and connects with it, to communicate with remote workers. This daemon is aware of local and remote resources and the middleware (e.g., a Secure Shell connection) over which they communicate. The daemon uses the Xenon library to start the worker on a remote machine, executing the necessary authorization, queueing or scheduling automatically. Because OMUSE contains large portions of C, C++ and Fortran and requires a large number of libraries, it is not copied automatically, but it is assumed to be installed on the remote machine. A binary-only release can be generated for resources, such as clouds, that employ virtualization. With these modifications, OMUSE is capable of starting remote workers on any computer the user has access to, without significant effort required from the user. From the user point of view, to use the distributed resources, any OMUSE script can be distributed by simply adding properties to each worker instantiation in the script, specifying the channel used, as well as the name of the resource, and the number of nodes required for this worker (see Fig. 2).

2.3 Unit conversion

In order to simplify the handling of units, a unit algebra module is included in OMUSE (Fig. 3). This module wraps standard Python numeric types or Numpy arrays, such that the resulting quantities (i.e., a numeric value together with a unit) can transparently be used as numeric types (see the function definition example in Fig. 3). Even high-level algorithms, such as ODE solvers, typically do not need extensive modification to work with OMUSE quantities (and in many cases work without any changes, if they are formulated in a dimensionally consistent way).

OMUSE *enforces* the use of units in the interfaces of the community codes. The specification of the unit dimensions of the interface functions is part of the interface specification (much in the same way as the data types of the functions). Using the unit-aware interfaces, any data that are exchanged within modules will be automatically converted without ad-

```
(1) grid=new_cartesian_grid((100,100))
(2) grid.ssh=0. | units.m
(3) subgrid=grid[0:50,0:50]
(4) channel=QG.grid.new_channel_to( grid )
(5) channel.copy_attributes( ["psi"] )
(6) channel.transform( ["ssh"], lambda x:f0/g*x, ["psi"])
```

Figure 4. Example usage of the high-level grid data structure: (1) initialization of an empty Cartesian grid; (2) defining an attribute, here a scalar field of sea surface height; (3) subgrid generation by indexing; (4) definition of an explicit channel from in-code storage to a grid in memory; (5) update of grid attributes over the channel; (6) functional transform over a channel.

ditional user input, or – if the units are not commensurate – a code exception is generated. Keeping track of different systems of units and the various conversion factors when using different codes quickly becomes tedious. Enforcing the use of units therefore eliminates an important source of errors.

2.4 Data model

The interfaces to the code send low-level data types (e.g., an array of floats) over the remote function channel. While this is simple and closely matches the underlying C or Fortran interface, one needs to duplicate much of the "bookkeeping" (i.e., the organization of the arrays and their indexing) in the user script if the low-level data types are used directly. Therefore, in order to simplify working with the codes, a data model is added to the interfaces based on the construction of high-level objects that store the data (Fig. 4). Two base data stores are available: particle sets and grids. The main difference between these are that particle sets can be extended dynamically and are unordered, while grids are fixed when generated, ordered and can be multidimensional. Typically, grid data structures are used to store the state in oceanographic simulation codes. Particle sets can, for example, be used for the storage of properties for Lagrangian particle trackers. The data stores can either reference memory in the main Python memory space (for sets defined independent of any code) or reference the data in the (possibly distributed) memory space of the community code. Subsets can be defined on the sets without additional storage (see Fig. 4; these subsets are implemented as views on the underlying local or remote data), and new sets can be constructed using simple operations.

2.4.1 Grid support

Compared to AMUSE, OMUSE expands the support of grid data structures by introducing different grid data types. All types of grids share the same base functionality, including grid sampling and slicing, the creation of save points, and the creation of grid copies that include part or all of the grid attributes. The new grid types form a hierarchy (Fig. 5), where each grid type has its own set of (derived) grid attributes (such as cell sizes) and utility functions (for basic operations,

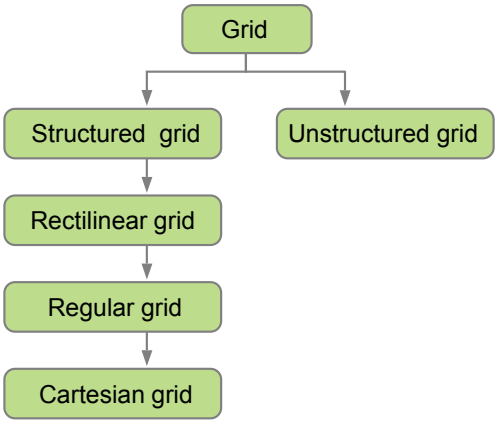


Figure 5. Hierarchy of grid data types in OMUSE. Arrows denote inheritance of the corresponding classes in OMUSE.

such as checking overlap or the extent of a grid). The grid types supported are Cartesian (single, constant cell size, same in each dimension), regular (constant cell size, different per dimension), rectilinear (cell boundaries specified per dimension), structured (cells specified by a grid of corner points) and unstructured (cell corners are specified for each cell individually).

2.4.2 Grid remappings

Grid remapping is a fundamental operation for coupled climate models, where heat and water fluxes are periodically transferred between different component models, each using different grids internally. In many cases, these remappings must be performed in an energy- or mass-conserving manner to maintain the global conservation conditions of the coupled climate system. As such, OMUSE interfaces with CDO for their implementation of a second-order conservative remapping scheme (see Sect. 3.2). However, different remapping backends can be used within OMUSE.

OMUSE extends AMUSE with support for remapping quantities between different grids (AMUSE included support only for copying data between two equivalent grids). OMUSE allows the user to instantiate grid remapping objects. The remapper is initialized by setting the source and destination grid and can be used to remap a list of grid attributes from one grid to the other.

The use of such a remapping object is illustrated in Fig. 6, where, as an example, the sea surface height values from one ocean model are remapped to the grid of another ocean model. Note that it does not matter (for the syntax) whether the grid values reside inside the community code or in Python memory. In this example both grids are stored in the memory of the community code, and, if needed, unit conversion of the values transferred between the models is automatically performed by the interface of the receiving code, as explained in Sect. 2.3.

```
(1) pop = POP(...)
(2) source = pop.elements
(3) adcirc = Adcirc(...)
(4) target = adcirc.elements
(5) remapper = conservative_spherical_remapper(source, target)
(6) remapper.forward_mapping(["ssh"])
```

Figure 6. Example usage of the high-level grid remapping functionality in OMUSE. In this example, the grid attribute `ssh` (for "sea surface height") is remapped from the source grid to the target grid, both stored inside the community codes, using a second-order conservative remapping scheme (the default). Unit conversions are performed automatically by the interface of the receiving community code.

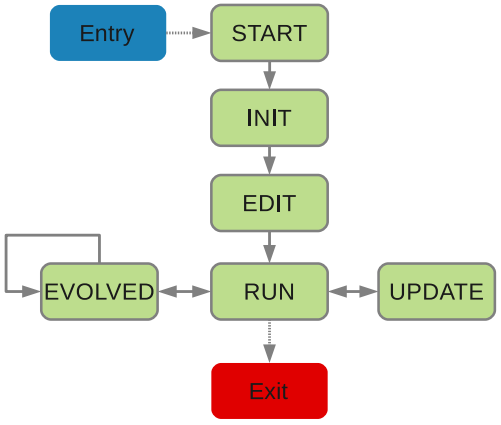


Figure 7. Example of a state model in OMUSE. The diagram gives the states that a simulation code can be in. Transitions between these can be triggered by explicit calls to the corresponding function (e.g., `initialize_code` from START to INIT) or implicitly (e.g., querying the grid state of a code may only be possible in the RUN state, and in this case the framework will call the necessary functions in order to get to the RUN state, guaranteeing a consistent state of the simulation code in the process). Adapted from Pelupessy et al. (2013).

Support for remapping between unstructured grids is limited in the CDO library. Conservative interpolation of fields represented on unstructured mesh discretizations (Farrell et al., 2009) is being generalized in the `libsupermesh` library (libSupermesh, 2016) and could be utilized in the future.

2.5 State model

The internal work flows of different codes are in general not the same, even if they represent similar physics. This can be due to the differences in the algorithms or simply because of design choices. For example, a change in one of the grid variables may necessitate a reinitialization of variables in one code, while in another code this may not be needed. It is easy to add the corresponding functions for such reinitialization to the interface. The problem with this is that it introduces differences between the interfaces, and is obviously error-prone if controlled by the user. In order to manage this, the interfaces in OMUSE can be supplied with a representation of the work flow of a code. This is done in the form of a graph consisting of model states as the vertices and the transitions between them as the edges. Model states each have a set of allowable interface function calls. Such an interface call can trigger a transition between states (and for each transition there is a respective interface function). With this *state model* OMUSE keeps track of the state of a code, changing the state when needed (and calling the corresponding interface methods). The state model will change state automatically if an operation is requested that is not allowed in the current state. If the request can not be fulfilled an error is returned. The state model is flexible: states can be added and removed as required. Most codes can be made to conform to a simple state model similar to the six-state model shown in Fig. 7.

2.6 Object-oriented interfaces

The object-oriented, or high-level, interfaces are the recommended way of interacting with the community codes. They consist of the low-level MPI interface to a code, with the unit handling, data model and state model on top of this. At this level the interactions with the code are uniform across different codes and the details of the code are hidden as much as possible. A lot of the bookkeeping (such as the explicit indexing of arrays and unit conversion) is absent in the high-level interface formulation. This makes the high-level interface much easier to work with and less prone to errors: the user does not need to know what internal units the code is using and does not need to remember the calling sequence or the specific order of calls.

2.7 I/O (input/output)

Community codes that are included into OMUSE will usually contain subroutines to read in and write simulation data. While calls to these can be added to the interface, this functionality is preferably not used within OMUSE. Instead, all simulation data are to be written and read from within the OMUSE script (although in practice there can be reasons to retain the original functionality as part of the interface, for example to use existing post-processing scripts). OMUSE includes a default output format based on HDF5[6] that writes out all data pertaining to a data set, effectively standardizing the I/O for all the codes included in the framework. In order to simplify import and export of data, OMUSE contains a framework for generic I/O to and from different file formats. A number of common file formats used in the oceanographic and climate modeling community are implemented (ADCIRC grid files, netCDF), as well as generic table-format file readers.

[6] http://www.hdfgroup.org

2.8 Data analysis

After a simulation, the generated data need to be analyzed. Python has good numerical and plotting libraries available, such as Numpy and Matplotlib (Dubois et al., 1996; Hunter, 2007), and thus data analysis can be easily incorporated into the OMUSE workflow. While the simulation codes are running, their internal state (as exposed through the interface) is accessible. This provides opportunities for efficient online data analysis and also monitoring (or visualizing) of the state of a running simulation. Based on the state of the model, the simulations can also be scripted beyond what is originally implemented in the simulation code (examples of the latter are event-driven data output, or repeat simulation and resampling according to predefined conditions).

3 Component modules

In the present version, OMUSE contains an initial set of ocean models, namely QG (a quasi-geostrophic solver), ADCIRC, POP and SWAN (ideally one would like to reach a "Noah's arc" milestone of having at least two independent application codes per domain; Portegies Zwart et al., 2009). The implementation in OMUSE of the code interfaces is described in this section. The models cover different physics and/or ranges of validity and allow for a number of different couplings between them. They also represent different levels of complexity in terms of code implementation, numerical schemes and a variety of discretizations (described below). In addition to the simulation codes, OMUSE also contains support codes, including for example the CDO package introduced above in Sect. 2.4.2 which is used to implement remapping schemes between different grids.

3.1 Simulation codes

3.1.1 QG

OMUSE includes QG, a code to calculate the dynamics of quasi-geostrophic ocean flow. The flow on a β-plane with Coriolis parameter $f = f_0 + \beta_0 y$ is described by the barotropic stream function ψ of the depth-integrated current velocity $\boldsymbol{u} = (u, v)$, with zonal velocity $u = -\partial\psi/\partial y$ and meridional velocity $v = \partial\psi/\partial x$. QG solves the governing barotropic vorticity equation (BVE) for ψ (Pedlosky, 1996),

$$\frac{\partial}{\partial t}\nabla^2\psi + J(\psi, \nabla^2\psi) + \beta_0\frac{\partial\psi}{\partial x} = \frac{1}{\rho_0 H}\left(\frac{\partial\tau^y}{\partial x} - \frac{\partial\tau^x}{\partial y}\right) - R_H\nabla^2\psi + A_H\nabla^4\psi, \quad (1)$$

where the Jacobian J, here representing the advection of relative vorticity, is defined by

$$J(F, G) = \frac{\partial F}{\partial x}\frac{\partial G}{\partial y} - \frac{\partial F}{\partial y}\frac{\partial G}{\partial x}, \quad (2)$$

and $\boldsymbol{\tau} = (\tau^x, \tau^y)$ represents the wind stress. QG can also solve for the first baroclinic mode of a mode expansion of the continuously stratified quasi-geostrophic vorticity equation (Flierl, 1978). The parameters ρ_0 and H are the reference ocean density and reference ocean depth, respectively. R_H and A_H are the bottom and lateral friction coefficients. QG solves Eq. (1) on a rectangular domain using a Cartesian grid. Boundary conditions consist of no-mass flux and/or no tangential stress (see for example Dijkstra and Katsman, 1997).

The QG code is written in Fortran 90 and uses the Poisson solver from the `fishpack`[7] or Intel `MKL`[8] libraries (depending on compiler). Although conceptually simple, QG provides an instructive case study for importing a code in OMUSE, with its simple internal state and without the complications of coordinate transformations, and serves as a template for other ocean models in OMUSE.

3.1.2 POP

The Parallel Ocean Program is a parallel global circulation model for ocean flows that solves the 3-D primitive equations for a stratified fluid using the hydrostatic and Boussinesq approximations (Smith et al., 2010). POP is often used to calculate strongly eddying ocean circulation models. However, resolving eddies on a scale that captures the instabilities that lead to ocean eddies requires the use of a high-resolution grid. Such high-resolution runs are computationally expensive, and POP is also frequently used for simulations at lower resolutions; in this case the effect of eddies is captured using subgrid parameterizations (Gent and McWilliams, 1990).

The POP grid is a structured 2-D grid in the horizontal dimensions, usually in a dipolar or tripolar configuration. POP requires that the grid dimensions are set at compile time. Therefore, we currently support two modes in which POP can be used through the OMUSE interface. The high-resolution mode assumes a grid size of 3600×2400, corresponding to a 0.1° resolution. The low-resolution mode assumes grid dimensions of 320×384 horizontal grid points, corresponding to a 1.0° resolution with tropical stretching. Vertically, the grid contains 40 or 42 nonequidistant layers, increasing in thickness from several meters near the surface to 250 m just above the lower boundary at 6000 m.

OMUSE interfaces with a version of POP (based on version 2.1) that contains several extensions (van Werkhoven et al., 2014) [9]. This implementation includes a flexible load-balancing scheme and optionally uses graphics processing units (GPUs) to accelerate computing-intensive parts of the code. Considering the fact that it takes at least 1000 simulated years to reach a near-statistical equilibrium state, it is common practice to restart POP from a spun-up solution. The

[7] www2.cisl.ucar.edu/

[8] software.intel.com/en-us/intel-mkl

[9] https://github.com/NLeSC/eSalsa-POP

so-called "restart file" and other settings can be set through the OMUSE Python interface after the code has been instantiated and reached the "START" state (see Fig. 7).

As with all codes in OMUSE, the POP interface employs a state machine that tracks the model state and ensures consistency by automatically calling the appropriate transition functions in the low-level interface. To be able to set many of the configuration options through the Python interface it was necessary to split several of the initialization routines in the POP source code. This was required because these routines used to read their configuration from a namelist file and immediately proceeded to initialize the model using that configuration. Within OMUSE, the model parameters are set through the interface as part of the Python script.

As such, the namelist file is only used to provide the code with default settings. After the settings have been read from the namelist, the model halts and waits for the settings that are specific to the experiment to be passed through the interface. When the user has completed configuring the experiment, the state machine will automatically call a state transition function to complete the model initialization and advance the model to a state from which the user can interact with the model data or begin evolving the model.

The POP interface provides two different ways to supply the model with forcings, such as wind stress, surface heat flux and surface freshwater flux. The first method involves setting the location of a file containing monthly averages of forcing data that will automatically be interpolated in time by the model. It is also possible to directly supply the model with forcing data through the interface, allowing POP to be coupled with, for example, an atmospheric model. When forcing data are supplied through the interface, POP will not use data from file for that type of forcing.

In the OMUSE examples repository[10], we have included an example Python script for setting up a POP run in high-resolution mode in a cluster environment. The user script has to specify the location of the cluster head node and provide the requested number of nodes and cores and time required for the simulation. After that the user can instantiate the interface to create a running simulation and interact with the model.

3.1.3 ADCIRC

The Advanced 3-D Circulation model solves the shallow-water primitive equations on a triangular unstructured mesh in either two or three dimensions. Water surface elevations ζ are obtained by solving the vertically integrated continuity equation in the generalized wave continuity equation (GWCE) formulation (Leuttich and Westerink, 2004). The momentum equations are either solved in vertically integrated form (2-D mode) or in 3-D (applying the Boussinesq and hydrostatic pressure approximations). In three dimensions, ADCIRC uses a generalized stretched vertical coordinate system (Leuttich and Westerink, 2004).

The ADCIRC mesh is represented in the OMUSE interface as an unstructured grid of nodes and elements (which can be accessed as the `nodes` and `elements` attributes of an ADCIRC instance), representing the nodes and triangular elements of the grid. In the case of ADCIRC all prognostic variables (with the exception of the wet–dry status of elements) are defined by a linear P_1 finite-element Galerkin representation over the entire domain, described by coefficients associated with mesh node positions. For example, in the simplest 2-D case these are the water level and its time derivative and the current velocities. The attributes of the elements are the nodes of each triangle and a status variable (indicating whether an element is dry or wet). In addition to this, the interface defines a `forcings` grid, which accepts the (possibly time-dependent) forcings. Depending on the parameters of the simulation these can be wind stresses, atmospheric pressure, tidal potential, wave stresses, etc. Boundaries are represented as sets of grids (one for each segment defined) with a reference to the nodes in the boundary segment, a type attribute (describing the type of boundary) and any extra attributes necessary to specify the boundary condition (e.g., the water level for a boundary with prescribed elevations).

3.1.4 SWAN

In addition to the above models of hydrodynamical ocean circulation, OMUSE includes an interface to the Simulating Waves Nearshore model, a code to calculate the propagation of wind-driven surface waves (Zijlema, 2010, and references therein). SWAN uses a statistical description of the space- and time-varying wave properties, solving for the evolution of the action density $N(\boldsymbol{x}, t; \sigma, \theta)$, defined in terms of the wave energy density spectrum E as $N = E/\sigma$, where N is a function of space $\boldsymbol{x}$, time t, relative radian frequency σ and direction θ. The evolution of the action density is governed by the action balance equation (e.g., Komen et al., 1994),

$$\frac{\partial N}{\partial t} + \nabla_{\boldsymbol{x}} \cdot \left[(\boldsymbol{c}_{\mathrm{g}} + \boldsymbol{U})N\right] + \frac{\partial (c_\sigma N)}{\partial \sigma} + \frac{(\partial c_\theta N)}{\partial \theta} = \frac{S_{\mathrm{tot}}}{\sigma}, \quad (3)$$

with $\boldsymbol{c}_{\mathrm{g}}$ the wave group velocity, $\boldsymbol{U}$ the (depth-averaged) current velocity, and c_σ and c_θ the propagation velocities in spectral and directional space, respectively. The source–sink term S_{tot} represents the physical processes which generate, dissipate or redistribute wave energy. Amongst them, SWAN includes generation of waves by wind, nonlinear transfer of wave energy (including three- and four-wave interactions) and wave decay due to whitecapping, bottom friction and wave breaking (see SWAN, 2015, for more information).

SWAN discretizes Eq. (3) on rectilinear, curvilinear (structured) or unstructured (triangular) grids in one or two dimensions. The OMUSE interface to SWAN supports rectilinear and unstructured grids (curvilinear grids can be added). The

[10] https://bitbucket.org/omuse/omuse-examples/

type of grid, as well as the type of grid for the forcings are determined when the code is instantiated. Depending on the selected grid the interface defines a regular grid `grid` or an unstructured grid with `nodes` and `elements` attributes. These have an attribute to access the action density N of the grid. In addition to this, the bathymetry can be specified and a number of potentially time-varying forcing inputs, such as water levels, water current velocities and wind velocities, can be used (again a separate grid is used for the forcings).

To simplify the interface a few restrictions are placed on the forcings. For example, all the forcings in the interface use the same grid (whereas SWAN supports different grids for different forcings). This is not a limitation: within OMUSE, any regridding (if necessary because the sources of the forcings use different grids) can be done on the framework level. If both calculation grid and input grid are unstructured, they are both assumed to use the same grid.

In the case of stationary calculations, the interface still defines an `evolve_model`, but it simply calculates the stationary action density (for all input times). It can still make sense to evaluate this in a time-dependent fashion, as the input forcings (and thus the equilibrium state) may change with time.

3.2 Support modules

In addition to the simulation codes, support modules written in different languages can be included in OMUSE. Such a support module may, for example, provide functionality for coupling models. A support module can be interfaced with the same remote function interface as that used for simulation codes. Currently, the only support module specific to OMUSE is CDO, which is used for computing grid remapping weights and performing the remapping of quantities between different grids.

3.2.1 CDO

Climate Data Operators (CDO, 2015) is a command-line tool, developed and maintained by the Max Planck Institute Hamburg, containing over 400 operators that can process and manipulate climate data stored in self-describing file formats, such as netCDF.

An OMUSE interface to CDO was created to be able to access the grid remapping functionality within CDO. This library contains a reimplementation of the SCRIP package (Jones, 1999), which is used in other climate model couplers, such as OASIS (Valcke, 2013) (while other couplers such as the Model Coupling Toolkit, Jacob et al., 2005, can use the remapping weights and addresses). In particular, the second-order conservative remapping scheme implemented in SCRIP is used to compute remapping weights for conservative exchanges of (e.g., heat and water) fluxes at the ocean–atmosphere interface.

A number of minor code modifications were necessary to be able to access the functionality in CDO as a library rather than as a command line tool. The low-level interface in OMUSE has to ensure that the internal state of CDO is consistent even though the code is not running as a command line tool. To do this, all grid information has to be propagated correctly to the different grid data storage structures used internally by CDO. In addition, the interface mimics some of the behavior of CDO to produce the exact same results as when invoked from the command line. These include ignoring any land masks in the source and target grids and increasing the number of search bins in the computation of remapping weights.

OMUSE implements a high-level object-oriented interface (called CDORemapper) on top of the low-level interface to CDO. This remapper can be initialized in three ways: (1) using a precomputed weights file as produced by CDO from the command line, containing all information about the source and destination grids, as well as the remapping weights; (2) using netCDF files for storing source and destination grid information (as used by CDO and SCRIP); and (3) setting OMUSE grid data types as a source or destination grid. Modes (2) and (3) can be combined (if desired), and for these modes the remapping weights are computed automatically as the remapper initializes.

When using the default second-order conservative remapping scheme, the implementation of CDO also computes the gradients of the source field each time a quantity is being remapped. Note that the second-order conservative remapping scheme comes with limitations: the source grid has to be a structured grid for the calculation of the gradients (needed for second-order accuracy; for more information see the CDO documentation).

In Fig. 8 we show the result of a remapping performed by the CDO remapper using the OMUSE interface. A sea surface temperature field is remapped from POP using a 0.1° tripole grid to an unstructured grid. The second-order conservative remapping scheme was used to compute the remapping weights based on the grid information presented by the OMUSE interfaces of both simulations.

3.3 Extending OMUSE

The effort required to import or interface an additional code with OMUSE varies with the code complexity and depending on whether a similar code already exists within the framework (in this respect the codes already included provide a good starting point). In order to be interfaced, a code needs to be written in a programming language for which MPI or socket bindings are available. The complete procedure (along with examples) is described in detail in the documentation section of the source distribution and the project website; here we only briefly outline the procedure.

To import a community code, one first creates a directory in the OMUSE community code base directory with

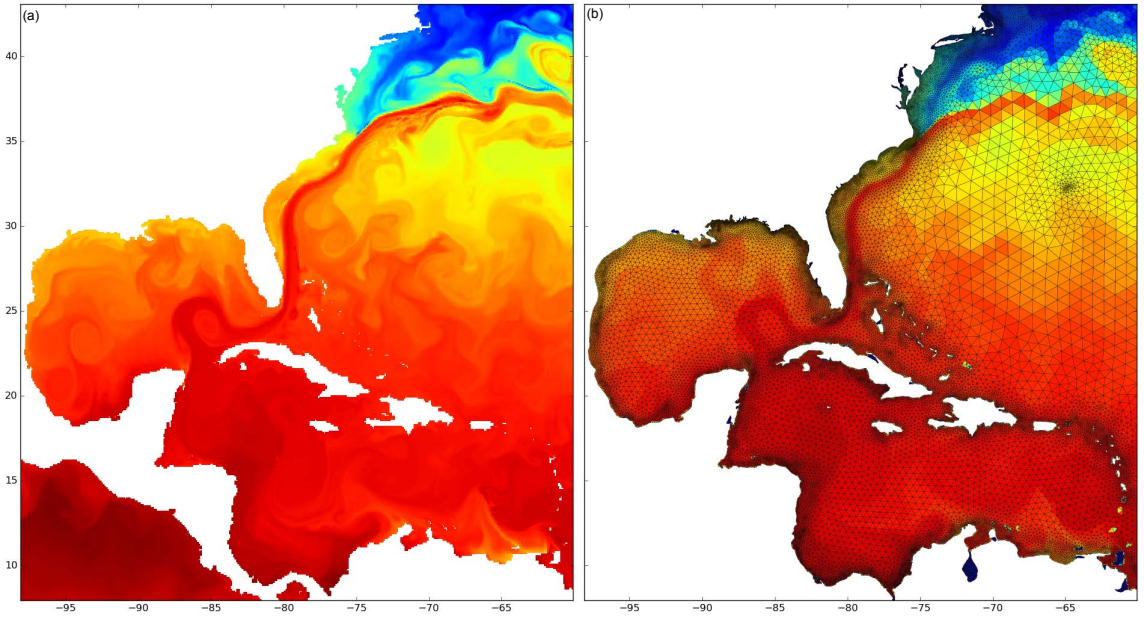


Figure 8. Result of a remapping performed by the CDO remapper using the OMUSE interface. A sea surface temperature field is remapped from POP using a 0.1° tripole grid **(a)** to the elements of an unstructured grid **(b)**.

the name of the module. The original source tree is imported in a subdirectory (by convention named "`src`"). The top-level directory contains the Python side of the interface ("`interface.py`"), the interface in the native language of the code (e.g., "`interface.c`") and a file for the build system ("`Makefile`").

The Python interface (described in the file `interface.py`) typically defines two classes: the low-level and the high-level interfaces. The former contains the function definitions of the calls which are redirected through the MPI communications channel to the corresponding call defined in the native interface file (`interface.c`). The high-level interface defines the units of the arguments of the function calls (see Sect. 2.3). In addition it specifies the parameters of the code, the state model (Sect. 2.5) and the mapping of the object-oriented data types to the corresponding low-level calls. By default, the data of the simulation is maintained in the community code's memory (and accessed transparently as described in Sect. 2.4).

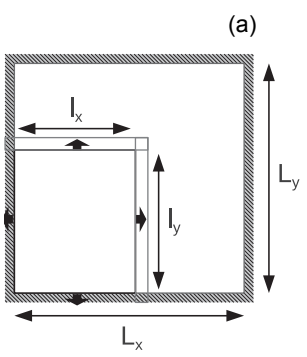
For modern and modular codes, often little to no changes in the original source code base (in "`src`") are needed. In other cases, a code may need significant source code changes (e.g., to separate the initialization stages and time stepping) or additions to implement functionality that is required for the OMUSE interface (e.g., externally imposed boundary conditions for grids). In these cases more effort is required to import the code and this will also make it more difficult to update the interface to a new version of the community code.

In our experience, writing an interface to a new code, which also involves writing tests, testing and debugging the interface, represents a modest amount of work. While every code is different and has its own peculiarities, it is typically something that can be completed (by someone with some familiarity with the source code) during a short working visit or small workshop. Defining an interface for a new physical domain can take longer, as these need refinement over time.

4 Code couplings

In addition to providing a unified interface to various types of codes, OMUSE has the objective of facilitating multiphysics simulations. For example, one would like to be able to couple a large-scale ocean circulation code with a regional ocean model (coupling across different scales) or couple a wave propagation model to an ocean flow model (coupling of different physics). Within OMUSE, community codes can be combined into coupled models which have wider applicability than the original codes. The setup of OMUSE allows for this in a transparent manner, such that the coupled models have a similar interface to the individual models.

The types of coupling that OMUSE can be applied to is large, and range from simple input–output coupling to dynamic one-way coupling and to the development of two-way coupled solvers (see Pelupessy et al., 2013 for more examples). OMUSE provides the following features to facilitate the building of coupled models: simplified, uniform access to the code simulation state; unified interfaces to the state of the simulation domain and its boundary conditions; and extensive automation of data transfer and conversion operations.




```
(1) class QGmodelWithRefinements(QG):                    (b)
      def interpolate_grid(self, grid):
        ...
      def add_refinement(self, offset, ..):
        ...
      def evolve_model(self):
        ...
      def update_refined_regions(self):
        ...

(2) qg=QGmodelWithRefinement()
    qg.add_refinement( offset=[0,0] | units.m, ...)
```

Figure 9. Schematic **(a)** and (abbreviated) definition of the refined QG model class **(b)** with an example (2) of its instantiation.

4.1 Example: QG model couplings

Some care is needed in the design of the code interfaces to ensure that couplings are as simple as possible. For example, the internal state of the QG simulation consists of the stream function ψ on two time levels; these are represented as a grid object with attributes `psi`, `dpsi_dt` and positions `x` and `y`. It is more convenient to represent the two time levels as the (backward) time derivative `dpsi_dt`, because this representation is independent of the time step (which can be different between codes). The stream function ψ (and its derivative) can also be queried at any position using an interface function `get_psi_state_at_point`. This function performs an (averaging) sampling and provides a grid-independent way to query and communicate the physical state. Another way to achieve this would be to perform a copy using a remapping channel, as described in Sect. 2.4.2.

In addition, QG has two mechanisms to receive input from other codes: it calculates the evolution of the stream function using an input wind stress field. This wind stress field can be set by changing the wind stress attributes `tau_x` and `tau_y` on the `forcings` grid. These can be copied or remapped from another grid (read in from disk or generated dynamically by another code) or by defining a (time- and/or position-dependent) functional form (from an analytic wind model, for example). Other possible inputs are the boundary conditions: ψ and $\partial\psi/\partial t$ on the domain boundary. These consist of four grid objects (one for each cardinal direction) of size $N_o \times 2$, where N_o is the number of grid points (in the corresponding dimension). Using these boundary grids, it is possible to implement two different strategies to vary the resolution over and/or the shape of the domain, namely grid nesting and domain decomposition.

4.1.1 Nested grid refinement

Depending on the parameters, Eq. (1) allows solutions with very narrow western boundary currents. Numerically this presents a challenge as the required resolution at this boundary may be much higher than for the rest of the basin. This is a typical situation where a nested solver (e.g., Debreu and Blayo, 2008) may efficiently be employed. We can implement such a multigrid coupled solver within OMUSE using the base QG as an underlying engine. The solution of Eq. (1) is obtained on a base grid with a refined region of higher resolution where the two grids are solved by separate instances of the QG.

Practically speaking, the following refinement strategy is followed (Fig. 9). Given a parent domain $L_x \times L_y$ a refined sub domain is defined by its offset, extension $l_x \times l_y$ and resolution dx. The low-resolution region consists of the whole domain $L_x \times L_y$ (including the refined region). The QG is used to solve for the flow on $L_x \times L_y$. A second instance of the QG is used to solve the flow Eq. (1) on the high-resolution subdomain $l_x \times l_y$ given appropriate boundary conditions. This high-resolution solution is then resampled and copied back (restriction operation) to correct the corresponding part of the domain on the low-resolution grid.

If the boundary of the high-resolution domain coincides with the boundaries of the parent domain (e.g., the east and south boundaries in Fig. 9), the boundary conditions are inherited from its parent. Otherwise, the boundary of the high-resolution region lies in the interior of $L_x \times L_y$, and in this case ψ and $\partial\psi/\partial t$ of the boundary can be obtained by interpolation of the low-resolution grid. In our template implementation of this multigrid solver, we implement it as a derived interface in OMUSE (Fig. 9). It implements the same high-level interface (i.e., it has the same methods) as the base QG, which allows these two to be used interchangeably. In particular, a refined region can itself have refinements.

4.1.2 Domain decomposition

Instead of overlapping domains, we can implement a similar coupling for (two or more) nonoverlapping (or partially overlapping) domains. A problem here is that the information used for the interpolated state on either side of a domain boundary does not carry information of the other domain. In the nested case the low-resolution solution is available over the whole domain, so it can provide this information.

This can be solved by iteration, but as the required step at each iteration (solving for $\partial\psi/\partial t$ using a Poisson solver) is quite expensive, this would be prohibitively inefficient. For this case, the problem can be accelerated by using accelerated vector extrapolation methods such as minimum polynomial extrapolation (MPE, Cabay and Jackson, 1976), i.e., we are solving for the fixed points of

$$\boldsymbol{x}^{k+1} = \boldsymbol{F}(\boldsymbol{x}^k), \tag{4}$$

where $\boldsymbol{x}^k$ is the vector consisting of the $\partial\psi_i/\partial t$ values on the boundaries (of all mutually neighboring domains). In Eq. (4), $\boldsymbol{F}$ is the operator determining the next vector in this sequence, with iteration index k. This operator is provided by the instances of the QG, which calculates a new set of $\partial\psi/\partial t$ values from previous set. The MPE method does not need explicit knowledge of the sequence generator, and as such is especially well-suited for the problem here (this information in our case is "hidden" in the QG code). In practice the solution converges within a handful of iterations to satisfactory precision.

The evolve loop of a compound QG consisting of N domains then proceeds as follows: (1) update the internal boundaries of each domain N. Values of ψ are interpolated from neighboring grids, a consistent set of $\partial\psi/\partial t$ values are calculated using the MPE method. (2) All the domains are stepped forward in time. An example of this will be shown in Sect. 5.2 below.

Note that both preceding examples (in Sect. 4.1.1 and 4.1.2) implement fairly close couplings. Nevertheless, the OMUSE framework can be used to implement these efficiently (both from the viewpoint of effort required to implement them and from a computational viewpoint). The most CPU-intensive parts of the computations (i.e., the solutions to the BVE, Eq. 1) are executed by the (optimized) QG solver, while on the framework level a limited amount of operations, such as data transfer and conversions, are handled.

5 Applications

To demonstrate the capabilities of OMUSE we present a number of example applications. These illustrate the application of the unified interfaces of OMUSE to calculate the same problem using different codes (Sect. 5.1), the use of OMUSE to implement intracode domain decomposition (Sect. 5.2), a two-way coupling between codes with different physics (Sect. 5.3), the embedding of a high-resolution region in a low-resolution domain using different codes (Sect. 5.4) and the addition of data analysis to a running computation (Sect. 5.5).

5.1 Critical transitions in a single-gyre ocean circulation model

The idealized classical model of a homogeneous midlatitude wind-driven ocean (Sverdrup, 1947; Stommel, 1948; Munk, 1950) has been extensively studied using dynamical systems theory (e.g., Ierley and Sheremet, 1995; Sheremet et al., 1997), where the successive bifurcations in single-layer (constant density) models are analyzed as the parameters of the model are varied. Here we will use two completely different simulation codes to obtain equilibrium solutions and study the bifurcation diagram in a single-gyre setup (Viebahn and Dijkstra, 2014).

The first code, QG, solves the BVE (Eq. 1), while ADCIRC solves the primitive equations and does not impose the quasi-geostrophic approximation. In this sense this simple numerical experiment will illustrate a posteriori the validity of the approximations made in deriving Eq. (1). We run the QG simulation for a 1000×1000 km basin with a resolution of $N_o = 200 \times 200$ with parameters $\beta_0 = 1.8616 \times 10^{-11}\,(\mathrm{m\,s})^{-1}$ $R_H = 0\,\mathrm{s}^{-1}$, $A_H = 1194\,\mathrm{m}^2\,\mathrm{s}^{-1}$ and a wind stress

$$\tau^x = -\frac{\tau_0}{\pi}\cos(\pi y/L)\ ;\ \tau^y = 0, \tag{5}$$

where τ_0 is determined by the adopted Reynolds number $Re = \tau_0/(\rho_0\beta_0 A_H H)$ ($\rho_0 = 1025\,\mathrm{kg\,m}^{-3}$ and $H = 4000$ m). For ADCIRC, a triangular grid matching this geometry is generated by subdividing the cells of a ($N_o = 50 \times 50$) Cartesian grid into four triangles by adding a vertex to the center of the cell. The parameters of ADCIRC are chosen to match the parameters in QG, and the same wind stress is applied.

In Fig. 10 we compare the stable stationary solutions of the two codes (these are obtained by running until the maximum fractional changes in either stream function ψ (for QG) or sea surface elevation η (for ADCIRC) between two successive diagnostic time intervals changes less than 10^{-4}). As can be seen, the two codes calculate solutions that agree well (although small differences can be seen). Figure 11 shows the corresponding bifurcation diagram when varying the Reynolds number. The correspondence between the two codes is good for low Reynolds number, showing the same qualitative behavior. At the bifurcation (above $Re \approx 25$) we found that the solutions obtained by ADCIRC become unstable to a basin-wide fast gravity wave mode, which is not represented in the QG model.

5.2 QG on a composite domain

As a first example of the use of OMUSE to construct new solvers by composing various subcodes, we show the results of an idealized calculation solving the BVE (Eq. 1) on composite domains. The coupled solver presented in Sect. 4.1.2 is employed for this. It uses separate instances of QG to calculate the ocean flow (i.e., solutions to Eq. 1) for a composite domain. In Fig. 12 the solution is calculated on a domain with a western boundary that is stepped. The domain (shown in Fig. 12) consists of a 4000×4000 km basin extended on the western side with a 1200×2000 km subdomain (the respective subdomains are indicated in the figure by the green

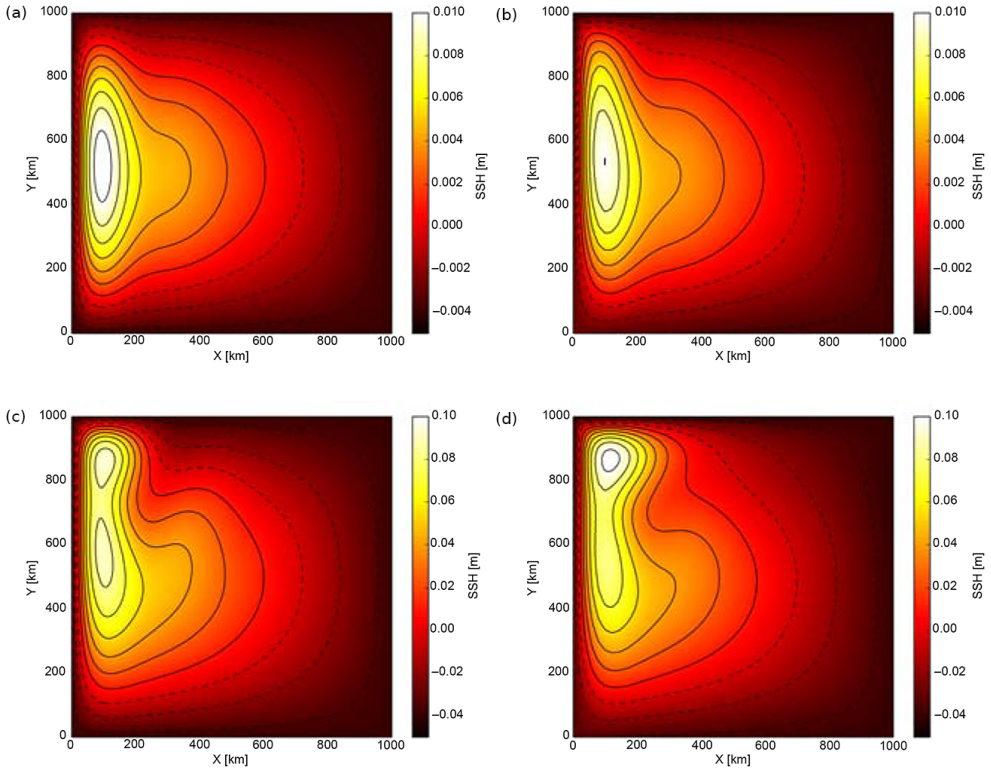


Figure 10. Comparison of QG and ADCIRC for a simplified midlatitude ocean configuration. Shown is the equilibrium SSH for a square domain basin of equal depth, driven by surface wind stress using the setup of Viebahn and Dijkstra (2014) (resulting in a single-gyre solution) at two different Reynolds numbers: $Re = 1$ **(a, b)** and $Re = 10$ **(c, d)**, where $Re = UL/A_H$ and $U = \tau_0/(\rho\beta_0 LH)$ is a characteristic horizontal velocity. In each case, the left panel shows the solution obtained using QG, and in the right panel the ADCIRC solution is shown.

and cyan rectangles). The single-gyre forcing of Eq. (5) is employed (with a Reynolds number $Re = 10$). The solution is shown after 15 days of evolution (at this early stage one can distinguish the Rossby waves moving east to west from the interior of the large basin, into the smaller domain).

Using such a composite domain, it is possible to calculate the effects of topographic features on the dynamics of boundary currents, or change the resolution across the domain. Such idealized modeling on a simplified domain is often useful to reduce the real-world topography to its essential features, e.g., Le Bars et al. (2012). The example above implements a tailored solver using the high-level OMUSE interface to QG. This demonstrates that the interfaces of OMUSE are capable of expressing fairly tight couplings. The alternative, and maybe more obvious, way to implement such a solver is to adapt the underlying Poisson solver to various domain shapes, which may involve changing the data representation. In contrast, the implementation here is done without reference to the underlying data structures and in principle does not depend on the grid type or shape used in the underlying solver.

5.3 Implementation of a coupled SWAN–ADCIRC model

The propagation of wind-driven surface waves is sensitive to water levels and current velocities. The properties of the underlying circulation will affect the evolution of the wind-driven wave field and the location of wave-breaking zones. However, wind-driven wave transport can generate radiation stress gradients that can in turn drive circulation set-up and currents. Currents can also be affected by changes in the vertical momentum mixing and bottom friction stresses generated by the wind-driven wave field. Thus, in many coastal applications, such as the calculation of storm surges, waves and circulation processes, should be mutually coupled.

Here we will demonstrate the implementation of such a coupling within the OMUSE framework, applying it to a coupling of the ADCIRC circulation model and the SWAN wave propagation model. A fully integrated coupled ADCIRC–SWAN model exists (Dietrich et al., 2011), and below we compare and contrast our method of coupling with this existing approach. The physical interactions between the different simulated components are schematically given in Fig. 13. Figure 14 shows the (somewhat simplified) OMUSE code corresponding to this model coupling. Note that in this cou-

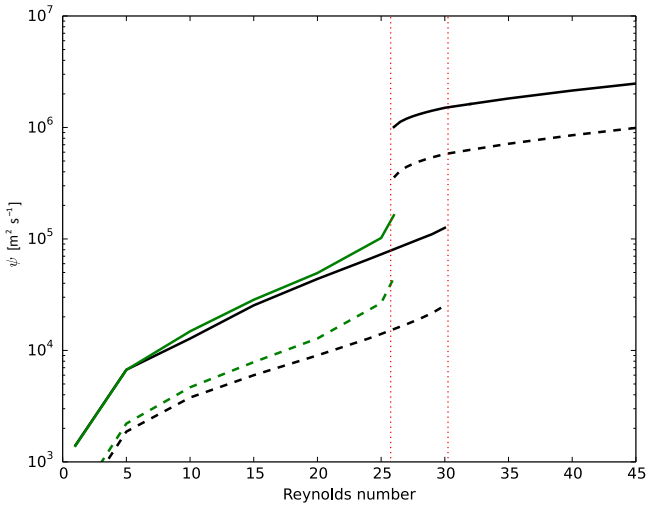


Figure 11. Part of the bifurcation diagram showing the upper and lower branches of steady and oscillatory solutions for a single-gyre ocean model. Shown are the mean (dashed) and maximum (solid) values of the stream function for QG (black) and ADCIRC (green) model runs, as a function of the Reynolds number Re. For ADCIRC the stream function is calculated as $\psi = g\zeta/f_0$, where ζ is the free-surface height. The values shown represent time-averaged values in case the system shows oscillatory behavior. The flow undergoes a cyclic fold bifurcation near $Re = 25$, as indicated by the vertical dashed lines (Viebahn and Dijkstra, 2014). The ADCIRC solution becomes (numerically) unstable at this bifurcation.

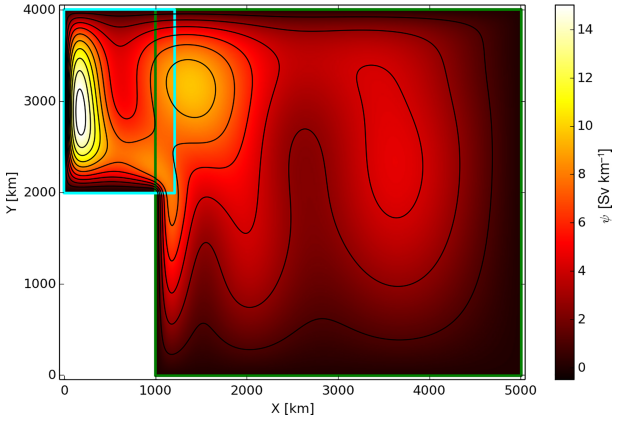


Figure 12. Stream function ψ for a nonrectangular domain run with QG on a composite domain. Plotted is ψ after 15 days of evolution with the composite QG code (Sect. 4.1.2) on a domain consisting of two coupled subdomains, indicated by the cyan and green rectangles.

pling both SWAN and ADCIRC use the same unstructured (triangular) grid. The communication between the codes (as shown in Fig. 14) is handled by `channels`, whereby the framework handles the copying (and unit conversion) of data.

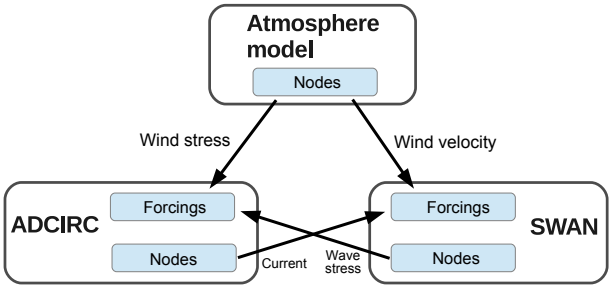


Figure 13. Schematic representation of the ADCIRC–SWAN coupling.

```
(1) channel1=hurricane.grid.new_channel_to( swan.forcings )
(2) channel2=hurricane.grid.new_channel_to( adcirc.forcings )
(3) channel3=adcirc.nodes.new_channel_to( swan.forcings )
(4) channel4=swan.nodes.new_channel_to( adcirc.forcings )
(5) while time<tend:
(3)     hurricane.evolve_model(time+dt/2)
(4)     channel1.copy_attributes(["tau_x","tau_y"])
(5)     channel2.copy_attributes(["vx","vy"])
(6)     adcirc.evolve_model(time+dt/2)
(7)     swan.evolve_model(time+dt/2)
(8)     channel3.copy_attributes(["current_vx","current_vy"])
(9)     channel4.copy_attributes(["wave_tau_x","wave_tau_y"])
```

Figure 14. Definition of communication channels and evolve step corresponding to Fig. 13.

As an example, we apply the coupled code to calculate the wave height and storm surge of hurricane Gustav (2008)[11] in the Gulf of Mexico. The hurricane is modeled using an analytic prescription (Holland, 1980) from data of a hurricane storm track (positions, central pressures, maximum wind speed, storm radius) read in from file. Implementation of this analytic model is in the form of a Python class mimicking a full simulation code. ADCIRC is run in 2-D barotropic mode with meteorological forcing from the hurricane model and wave stresses provided by SWAN. There is no forcing on the open-ocean boundaries. For the discretization of the action density, SWAN uses 36 bins in the directional space and 32 bins in frequency (from 0.05 to 1 Hz). The standard set of third-generation wave parameters, including the effects of wave breaking, bottom friction and 3-wave interaction is used. The time step (`dt`) between updates of the coupled quantities is 600 s.

In Fig. 15 we show the resulting wave heights calculated by the model during the development of hurricane Gustav at three different times. The results of the OMUSE coupling are similar to the results of the integrated coupling implementation (Dietrich et al., 2011, and above mentioned website). Technically the coupling in OMUSE differs from the implementation by Dietrich et al. (2011), as the latter directly copies data in the unified memory space of a single binary

[11] The data for this example come from http://www.caseydietrich.com/2012/06/27/example-input-files-for-swanadcirc/

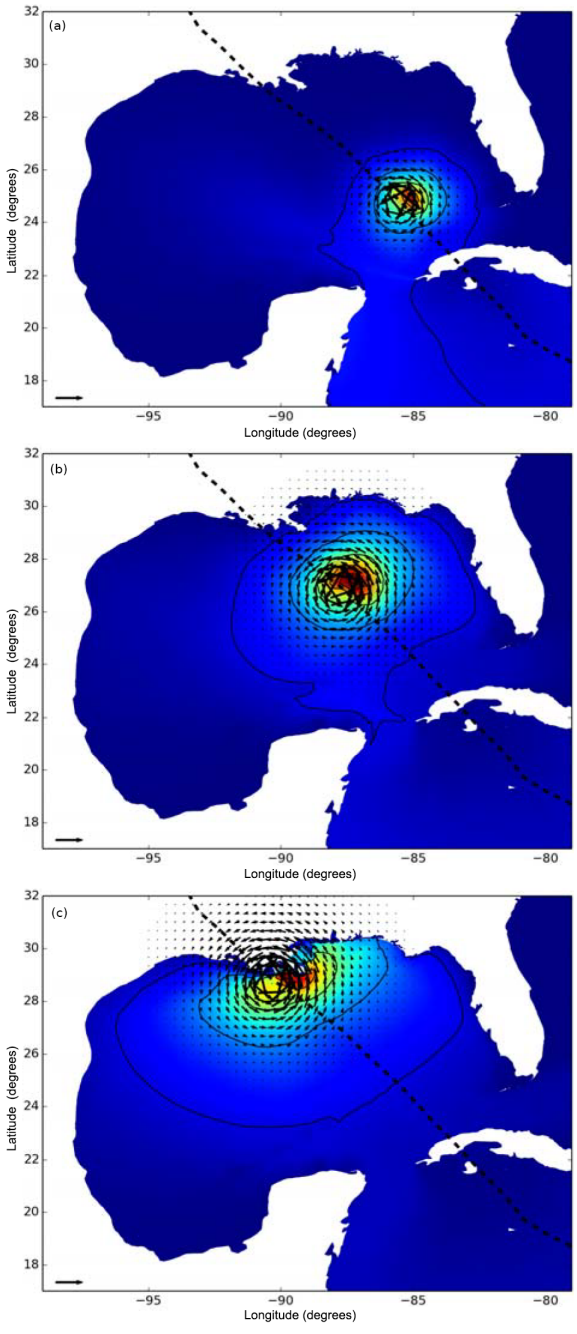


Figure 15. Significant wave heights for hurricane Gustav (2008), calculated using a coupled ADCIRC–SWAN simulation. The significant wave height field (shading, with contours at 1, 3, 6, 9 and 12 m) is shown with the (model) wind field superimposed (arrows, where the arrow on the lower left corresponds to 30 m s^{-1}), and the storm track (dashed line). Shown are frames **(a)** 156, **(b)** 168 and **(c)** 180 h after start of the simulation (25 August 2008, 00:00 UTC).

(and for that reason is more efficient). However, both implement the same coupled processes, and the approach taken by OMUSE does not depend on the particular aspects of the selected codes – exactly the same script could be used by other codes using the same interfaces.

5.4 Embedded regional model

A recurring problem for regional or coastal modeling is the application of realistic boundary conditions from the open ocean, even more so when one is interested in the effect of large-scale or global processes on the regional level. One approach to obtain realistic boundary conditions at the required scale is the nesting of a high-resolution and small-scale model in a lower resolution but larger scale model (e.g., Debreu et al., 2012; Djath et al., 2014).

Here we illustrate the implementation of (one-way) nesting in OMUSE by embedding a regional high-resolution barotropic ADCIRC model of the Caribbean and North American Atlantic coast into a POP global circulation model (see Fig. 16). In this case, since POP uses a curvilinear structured grid and ADCIRC an unstructured triangular mesh, it is necessary to perform a remapping when transporting variables from one code to the other (these functional *remapping* channels are indicated in Fig. 16 by the labeled arrows).

For the actual implementation of the coupling in OMUSE, the difference between using a remapping channel and a normal (data copying) channel (such as the ones used in Sect. 5.3) is small: the only difference with a normal channel is that upon initialization the actual remapping method to be used needs to be specified for a new remapping channel. The usage of the remapping channel to prescribe the data flow in the coupled model (Fig. 17) uses the same semantics.

In order to calculate the dynamics of the nested regional model, ADCIRC in 2-D barotropic mode needs an input wind stress field and the specification of either the sea surface level or normal fluxes on the boundary. In addition to this, the model can be initialized from remapped flow variables (barotropic velocities and sea surface heights). Note that a fully consistent coupling between the two codes is not possible since they solve for a different set of variables (2-D barotropic vs. 3-D baroclinic). For the (conceptual) example here, a coupling was made on the sea surface elevation, and the bathymetry of the ADCIRC grid was limited to 500 m depth (so the barotropic basin represented in ADCIRC can only be compared with the upper 500 m layer of POP). The time step for the coupling (updates of the boundary surface elevations) is taken to be equal to the POP internal time step of approximately 30 min. The remappings are performed at each time step for the wind stresses and for the sea surface heights.

Figure 18 shows the sea surface heights and velocities on the original low-resolution POP grid and the embedded higher resolution ADCIRC grid after 30 days of adjustment (after this the ADCIRC solution follows the (slow) variations

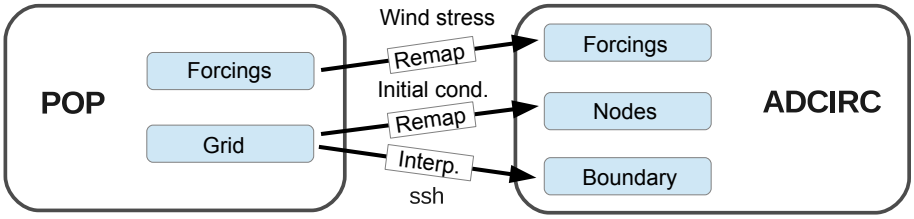


Figure 16. Schematic representation of the POP–ADCIRC one-way coupling for an embedded domain. The labeled arrows indicate the use of remapping channels. "Remap" stands for a conservative remapping between the structured POP grid and the unstructured ADCIRC grid, while "Interp." indicates that the variables are interpolated.

```
(1) forcings_channel=pop_forcings_grid.new_remapping_channel_to(
                        adcirc.forcings, conservative_spherical_remapper )
(2) boundary_channel=pop_grid.grid.new_remapping_channel_to(
                        adcirc.elevation_boundary, interpolating_remapper )
(3) while time<tend:
(4)     pop.evolve_model( time+dt/2 )
(5)     forcings_channel.copy_attributes( ["tau_x","tau_y"] )
(5)     boundary_channel.copy_attributes( ["ssh"] )
(6)     adcirc.evolve_model( time+dt )
(7)     pop.evolve_model( time+dt )
(8)     time+=dt
```

Figure 17. Definition and use of remapping channels for the POP-ADCIRC embedding of Fig. 16.

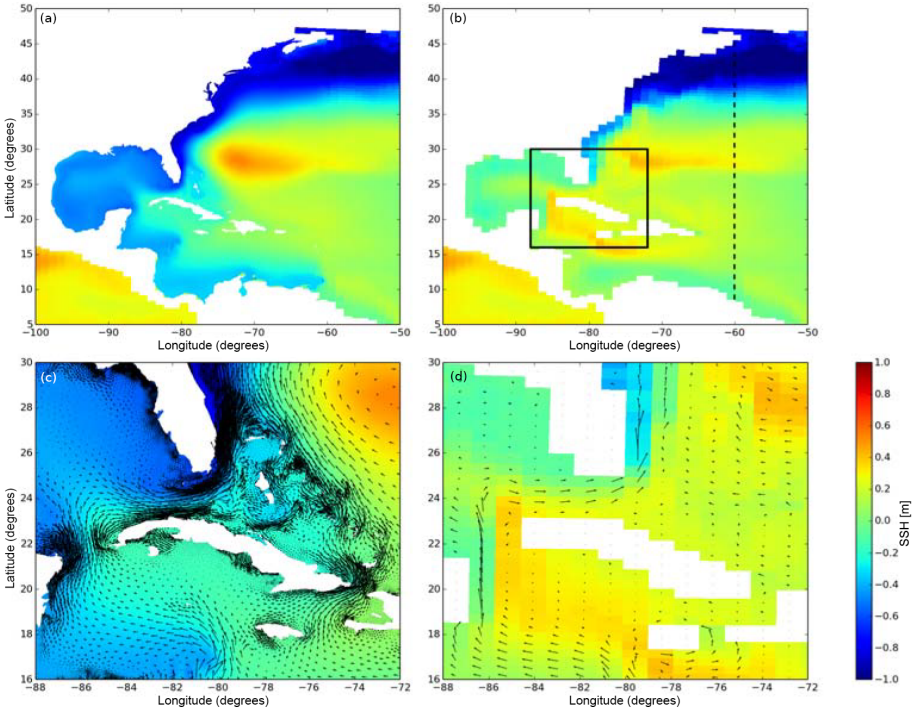


Figure 18. Sea surface heights and velocities of a ADCIRC run embedded in a global circulation POP model. Top panels show the sea surface height (SSH) of a region covering the western North Atlantic Ocean, Caribbean Sea and Gulf of Mexico. Panel **(a)** shows the high-resolution ADCIRC SSH field (superimposed on the POP field) and panel **(b)** the low-resolution POP field. The black square indicated in **(b)** is shown in more detail in **(c)** and **(d)**, where the SSH with velocities superimposed are shown (in the case of ADCIRC the barotropic velocities are shown, for POP the are the surface velocities). The dashed line in **(b)** is the open-ocean boundary of the regional ADCIRC model.

of POP). A fully consistent coupling is possible when using ADCIRC in baroclinic mode. In this case, the coupling proceeds (with a larger number of coupling variables involved) along similar lines.

5.5 On-the-fly data analysis

In addition to consuming massive amounts of CPU time, current large-scale simulations are capable of generating enormous amounts of data. Usually, it is possible to store only a very limited subset of this data; this limits the data analysis that can be performed. One solution to this has been to do all or part of the analysis on the fly. Online data analysis offers several opportunities, including the fact that special actions can be taken when interesting events occur. Such special actions may include inspecting the model internal data at resolutions, both spatial and temporal, that are not available or feasible with offline data analysis. While running simulations through OMUSE, the simulation state is accessible, and this allows for data analysis while a simulation is running.

As a proof-of-concept application we add an online ocean eddy tracker on top of the POP model. The interest in ocean eddies comes from the fact that eddies transport considerable energy and mass and as such influence the dynamics of large-scale ocean circulation and the climate (e.g., Viebahn and Eden, 2010; Griffies et al., 2015). To understand eddy properties and variability, several mesoscale eddy-tracking algorithms have been proposed in recent years. We have adapted a sea-surface-height-based eddy-tracking code that is implemented in Python, called `py-eddy-tracker` (Mason et al., 2014). The code uses high-pass-filtered sea level anomaly (SLA) fields. On the filtered fields, contours are computed at 1 cm intervals for levels between −100 and 100 cm. These contours are then searched to locate eddies based on their shape, area and amplitude. `py-eddy-tracker` tracks eddies across successive SLA fields using a search ellipse, bounded by the local (long baroclinic) Rossby wave speed.

We have generalized the code in order to use different data sources, including output that is obtained directly from numerical models. To this end, we have modified the `py-eddy-tracker` to be able to handle grids that contain gaps, as land-only blocks are not part of the simulation in POP. We use `Basemap`[12] to compute a land mask for the given grid and apply it to the SLA field. Finally, we have created a simple, but easy to use, interface to the `py-eddy-tracker` that understands the grid data structures and units used in OMUSE.

Figure 19 shows the code required to build an online eddy-tracking program with OMUSE. The interface `EddyTracker` is given the OMUSE grid data type used by POP and automatically performs unit conversions and extracts the information that it needs (i.e., the sea surface height and the coordinates of the grid points).

```
from omuse.ext.eddy_tracker.interface import EddyTracker
from omuse.community.pop.interface import POP
p=POP( ... ) # start POP as you would do normally

dt_analysis = 7 | units.day
tracker = EddyTracker( grid=p.nodes, domain='Regional',
  lonmin=0. | units.deg, lonmax=50. | units.deg,
  latmin=-45. | units.deg, latmax=-20. | units.deg, dt_analysis )

tnow = p.model_time
stop_time = p.model_time + (1 | units.yr)

while (tnow < stop_time):
    p.evolve_model( tnow + dt_analysis )
    tracker.find_eddies( ssh=p.nodes.ssh, rtime=p.model_time )
    tnow = p.model_time

tracker.stop()
p.stop()
```

Figure 19. This example demonstrates how to build an application that analyzes data from a running simulation using OMUSE. This code implements an online eddy-tracking program that tracks the eddies based on sea surface height every 7 days for 1 year of POP simulation.

Figure 20 shows the output of the online eddy-tracking program that uses sea surface height data directly from a running POP simulation. In this image, we can clearly see the large anticyclonic eddies that result from the retroflection of the Agulhas Current, as well as many smaller eddies being tracked over time by the online eddy-tracking algorithm. The data generated by the online eddy tracker can, for example, be used to compare the statistics of the simulated eddies to the analysis made using `py-eddy-tracker` (or other tools) of altimetry data.

6 Summary and discussion

We have presented OMUSE, an environment which provides a homogeneous interface to existing or newly developed ocean models. As illustrated by the results in the previous section, the usage opportunities for OMUSE range from running simple numerical experiments with single codes (e.g., Sect. 5.1), to combining simulation codes and data analysis tools (Sect. 5.5) and setting up fairly complicated and strongly coupled solvers (Sect. 5.2) to solve problems that are intrinsically multiscale (Sect. 5.4) and/or require different physics (Sect. 5.3). Using OMUSE, simulations can be easily scripted and on-the-fly data analysis can be added.

The implementation of the different use cases is facilitated by several aspects of the OMUSE design. OMUSE defines standardized interfaces and data structures for different codes. The data structures and the state model as well as the communication model used in OMUSE are flexible and allow a wide variety of codes, written in different languages, to

[12] http://matplotlib.org/basemap/

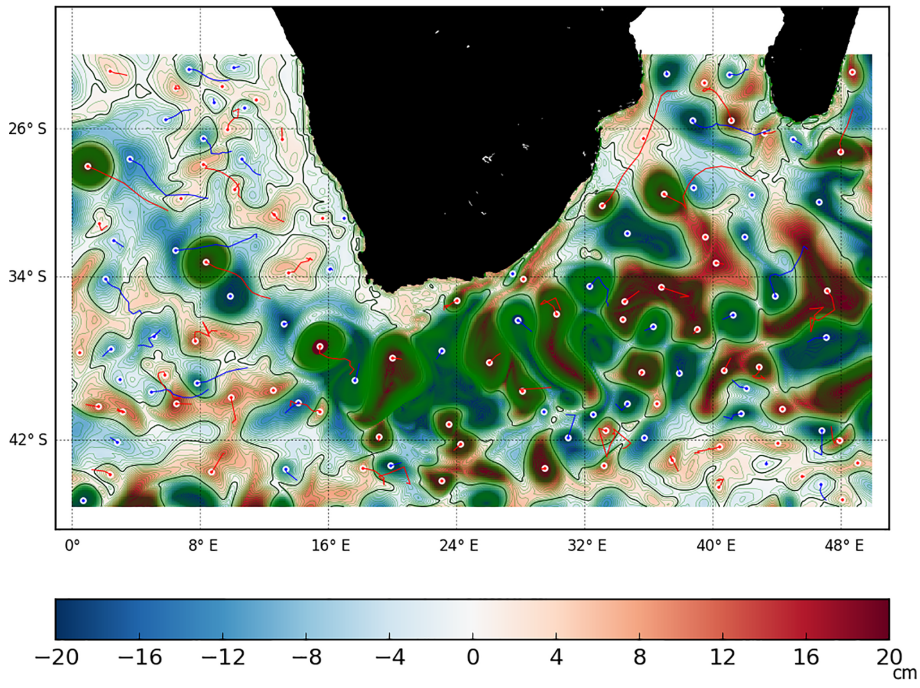


Figure 20. Output of the online eddy-tracking application using data from a running POP simulation, showing a region around the southern tip of Africa. The green lines show the contours between areas of different sea level anomaly values. Red indicates areas of elevated sea level, and is used to detect anticyclonic eddies. Similarly, blue indicates a lower sea level and is used to identify cyclonic eddies. The red or blue lines indicate the track that an eddy has traveled since it was first detected.

be integrated with OMUSE. OMUSE also works well with established methods to generate initial conditions and analyze the resulting data.

OMUSE shares some of the goals of a number of other coupling frameworks that have been developed in the Earth system modeling community (e.g., Hill et al., 2004; Buis et al., 2006; Gregersen et al., 2007; Jacob et al., 2005; Larson, 2005; Peckham et al., 2013; Valcke, 2013). While they follow quite different design strategies, these coupling frameworks share the ability to initialize different models, move (and if necessary regrid) data between them and manage the time evolution of the component models. As mentioned in the introduction, they can roughly be divided into integrated coupler systems and coupling library frameworks (Valcke et al., 2012). For example, the Earth system Modeling Framework (ESMF, Hill et al., 2004) presents an example of the integrated approach, where component modules are made to conform to a simple calling sequence and called from a single executable (which also implements the coupling algorithm). Cossarini et al. (2017) provide an example of the coupling library approach. In this case the coupler consists of a wrapper around a the Biogeochemical Flux Model (BFM) code that can be called from another code (MITgcm). OMUSE shares the characteristic of the integrated approaches that the component models are called from a single executable (in our case by writing a concise Python script expressing the physics of the model). However, the component models in OMUSE, while called from the framework, are running independently of one another and the user process acts like a coupler providing regridding and data conversion services, sharing this property with coupling library frameworks such as OASIS (Valcke, 2013) and PALM (Buis et al., 2006).

The closest equivalent to OMUSE may be the Community Surface Dynamics Modeling System (CSDMS; Peckham et al., 2013). CSDMS and OMUSE follow a similar design philosophy (as summarized in Peckham et al., 2013), by aiming for a modular-component-based modeling framework. The CSDMS BMI (basic model interface) and CMI (component model interface) are roughly equivalent to the OMUSE low-level interface. The main differences between OMUSE and CSDMS are that the former presents Python as the main user interface for programming an application, while for the CSDMS there are various choices, including a GUI front end. In addition, OMUSE simplifies the interaction with the community codes using high-level object-oriented data structures on top of the low-level interface and OMUSE has a more extensive and flexible state model, allowing for further automation. The similarity of the CSDMS and OMUSE interfaces translates, in principle, into interoperability between these frameworks, since the interface components of a code in the CSDMS can be converted to an OMUSE interface (possibly by a general converter of a BMI to an OMUSE low-level interface). The reverse (using an

OMUSE low-level interface as BMI) would also be possible, but not all functionality provided by an OMUSE interface necessarily maps to the BMI.

It is important to ensure the accuracy, reliability and reproducibility of a integrated framework such as OMUSE. We employ a number of strategies to ensure this is the case. The framework itself is tested daily and upon the commit of changes using more than 2000 component tests that cover approximately 80 % of the framework code and range from basic tests of the interfaces to the simulation codes as a whole. The simulation codes themselves are validated by comparing the results of test problems run using OMUSE with the results of the code running stand-alone (usually a number of test problems are developed for the simulation codes). In some cases (for example the ADCIRC–SWAN coupling) the results of a coupled solver implemented within OMUSE can be compared with a reference coupling implementation (e.g., Dietrich et al., 2011). In any case, to ensure the correctness of a new application in OMUSE, one should conduct the usual tests to ensure the validity and verify the results.

An important concern of a coupling framework such as OMUSE is performance. While the initial driver for the development of OMUSE is to simplify the setup and development of coupled simulations, the architecture of OMUSE is designed with a high degree of parallelism. The internal data structures are efficient. Also the individual simulation codes are often highly optimized. So the performance of an OMUSE application is rarely a concern, but this is strongly problem-dependent. In practice, the overhead imposed by the framework is often measured to be rather small (less than a few percent), but it is not difficult to formulate problems where the strength of the coupling is intrinsically so strong that very frequent communication between the component solvers is necessary.

In this respect a limitation of the current design of OMUSE is the fact that the communication between solvers is handled by a single-process user script. This imposes a bottleneck for the performance of the communication between, for example, two parallel codes. While in the current setup there are some mitigating techniques that can be applied (asynchronous communication or grouping and spawning the communication-intensive subprocesses), ultimately we would need to implement a *distributed* communication channel that would direct the data flow from the sending to the receiving process directly. Note that such distributed communication channels would not change the semantics of the use of a channel between data structures.

Code availability. OMUSE is available at the project website https://bitbucket.org/omuse (archived versions will be available at the Zenodo archive, doi:10.5281/zenodo.809336). It is foreseen to grow over time with new codes and capabilities and can easily be adapted for private use. OMUSE comes with basic tests, and a separate repository with examples is set up at the aforementioned website.

OMUSE is distributed under an Apache 2.0 license. This refers only to the framework and interface code, not to the simulation codes (including their native interface). Where these are distributed under an open-source license, community codes can be included in the framework source distribution; otherwise these codes must be downloaded separately. New codes or extensions, as well as bug fixes, may be submitted to the repository. OMUSE encourages the practice of distributing simulation codes by reporting automatically upon conclusion of an OMUSE script the community codes that were used and providing the relevant references for inclusion in publications resulting from their use.

Competing interests. The authors declare that they have no conflict of interest.

Acknowledgements. OMUSE was developed as part of the ABC-MUSE project, funded by Netherlands eScience Center (file number 027.013.701, 2013-2016). This research was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement no. 671564 (COMPAT project) and by the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (Netherlands Organisation for Scientific Research, NWO) under project no. 858.14.062. We want to thank Marcel Zijlema and Julie Pietrzak for discussing and commenting on the paper.

Edited by: Didier Roche
Reviewed by: Carsten Lemmen and one anonymous referee

References

Brunnabend, S.-E., Dijkstra, H. A., Kliphuis, M. A., van Werkhoven, B., Bal, H. E., Seinstra, F., Maassen, J., and van Meersbergen, M.: Changes in extreme regional sea surface height due to an abrupt weakening of the Atlantic meridional overturning circulation, Ocean Science, 10, 881–891, https://doi.org/10.5194/os-10-881-2014, 2014.

Buis, S., Piacentini, A., Déclat, D., and the PALM Group: PALM: a computational framework for assembling high-performance computing applications, Concurr. Comp.-Pract. E, 18, 231–245, 2006.

Cabay, S. and Jackson, L.: A Polynomial Extrapolation Method for Finding Limits and Antilimits of Vector Sequences, SIAM J. Numer. Anal., 13, 734–752, 1976.

Candy, A. S., Avdis, A., Hill, J., Gorman, G. J., and Piggott, M. D.: Integration of Geographic Information System frameworks into domain discretisation and meshing processes for geophysical models, Geosci. Model Dev. Discuss., 7, 5993–6060, https://doi.org/10.5194/gmdd-7-5993-2014, 2014.

Cazenave, A.: Present-day sea level change: Observations and causes, Rev. Geophys., 42, RG3001, https://doi.org/10.1029/2003RG000139, 2004.

CDO 2015: Climate Data Operators, available at: http://www.mpimet.mpg.de/cdo, 2015.

Cossarini, G., Querin, S., Solidoro, C., Sannino, G., Lazzari, P., Di Biagio, V., and Bolzon, G.: Development of BFMCOUPLER (v1.0), the coupling scheme that links the MITgcm and

BFM models for ocean biogeochemistry simulations, Geosci. Model Dev., 10, 1423–1445, https://doi.org/10.5194/gmd-10-1423-2017, 2017.

Danilov, S.: Ocean modeling on unstructured meshes, Ocean Model., 69, 195–210, https://doi.org/10.1016/j.ocemod.2013.05.005, 2013.

Debreu, L. and Blayo, E.: Two-way embedding algorithms: a review: Submitted to Ocean Dynamics: Special Issue on Multi-Scale Modelling: Nested Grid and Unstructured Mesh Approaches, Ocean Dynam., 58, 415–428, 2008.

Debreu, L., Marchesiello, P., Penven, P., and Cambon, G.: Two-way nesting in split-explicit ocean models: Algorithms, implementation and validation, Ocean Model., 49–50, 1–21, 2012.

Dietrich, J., Zijlema, M., Westerink, J., Holthuijsen, L., Dawson, C., Luettich, R., Jensen, R., Smith, J., Stelling, G., and Stone, G.: Modeling hurricane waves and storm surge using integrally-coupled, scalable computations, Coast. Eng., 58, 45–65, 2011.

Dijkstra, H. A. and Katsman, C. A.: Temporal variability of the Wind-Driven Quasi-geostrophic Double Gyre Ocean Circulation: Basic Bifurcation Diagrams, Geophys. Astrophys. Fluid Dyn., 85, 195–232, 1997.

Djath, B., Melet, A., Verron, J., Mollines, J.-M., Barnler, B., Gourdeau, L., and Debreu, L.: A 1/36° model of the Solomon Sea embedded into a global ocean model: On the setting up of an interactive open boundary nested model system, J. Oper. Oceanogr., 7, 34–46, 2014.

Drost, N., Maassen, J., Van Meersbergen, M. A., Bal, H. E., Pelupessy, F., Zwart, S. P., Kliphuis, M., Dijkstra, H. A., and Seinstra, F. J.: High-performance distributed multi-model/multi-kernel simulations: A case-study in jungle computing, 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 150–162, 2012.

Dubois, P. F., Hinsen, K., and Hugunin, J.: Numerical Python, Comput. Phys., 10, 262–267, 1996.

Farrell, P. E., Piggott, M. D., Pain, C. C., Gorman, G. J., and Wilson, C. R. G.: Conservative interpolation between unstructured meshes via supermesh construction, Comput. Method. Appl. M., 198, 2632–2642, https://doi.org/10.1016/j.cma.2009.03.004, 2009.

Flierl, G. R.: Models of vertical structure and the calibration of two-layer models, Dynam. Atmos. Oceans, 2, 341–381, 1978.

Gent, P. R. and McWilliams, J. C.: Isopycnal mixing in ocean circulation models, J. Phys. Oceanogr., 20, 150–155, 1990.

Gregersen, J. B., Gijsbers, P. J. A., and Westen, S. J. P.: OpenMI: Open Modelling Interface, J. Hydroinform., 9, 175–191, 2007.

Griffies, S. M., Winton, M., Anderson, W. G., Benson, R., Delworth, T. L., Dufour, C. O., Dunne, J. P., Goddard, P., Morrison, A. K., Rosati, A., Wittenberg, A. T., Yin, J., and Zhang, R.: Impacts on Ocean Heat from Transient Mesoscale Eddies in a Hierarchy of Climate Models, J. Climate, 28, 952–977, 2015.

Hill, C., DeLuca, C., Balaji, V., Suarez, M., and da Silva, A.: The architecture of the earth system modeling framework, Comput. Sci. Eng., 6, 18–28, 2004.

Holland, G. J.: An Analytic Model of the Wind and Pressure Profiles in Hurricanes, Mon. Weather Rev., 108, 1212–1218, 1980.

Hunter, J. D.: Matplotlib: A 2D graphics environment, Comput. Sci. Eng., 9, 90–95, 2007.

Ierley, G. R. and Sheremet, V. A.: Multiple solutions and advection-dominated flows in the wind-driven circulation – Part I: Slip, J. Mar. Res., 53, 703–737, 1995.

IPCC: Climate Change 2013: The Physical Science Basis, Contribution of Working Group I to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change (IPCC), edited by: Stocker, T. F., Qin, D., Plattner, G.-K., Tignor, M., Allen, S. K., Boschung, J., Nauels, A., Xia, Y., Bex, V., and Midgley, P. M., Cambridge University Press, Cambridge, UK, New York, NY, USA, 2013.

Jacob, R., Larson, J., and Ong, E.: $M \times N$ communication and parallel interpolation in Community Climate System Model version 3 using the Model Coupling Toolkit, Int. J. High Perform. C., 19, 293–307, 2005.

Jones, P. W.: First-and second-order conservative remapping schemes for grids in spherical coordinates, Mon. Weather Rev., 127, 2204–2210, 1999.

Komen, G., Cavaleri, L., Donelan, M., Hasselmann, K., Hasselmann, S., and Janssen, P.: Dynamics and Modelling of Ocean Waves, Cambridge University Press, 1994.

Larson, J.: The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models, Int. J. High Perform. C., 19, 277–292, 2005.

Le Bars, D., De Ruijter, W. P. M., and Dijkstra, H. A.: A New Regime of the Agulhas Current Retroflection: Turbulent Choking of Indian–Atlantic leakage, J. Phys. Oceanogr., 42, 1158–1172, 2012.

Leuttich, R. and Westerink, J.: Formulation and Numerical Implementation of the 2D/3D ADCIRC Finite Element Model Version 44.XX, Tech. rep., 2004.

libSupermesh: libsupermesh parallel supermeshing library, available at: https://bitbucket.org/libsupermesh/libsupermesh, 2016.

Maltrud, M., Bryan, F., and Peacock, S.: Boundary impulse response functions in a century-long eddying global ocean simulation, Environ. Fluid Mech., 10, 275–295, 2010.

Mason, E., Pascual, A., and McWilliams, J. C.: A new sea surface height–based code for oceanic mesoscale eddy tracking, J. Atmos. Ocean. Tech., 31, 1181–1188, 2014.

Munk, W.: On the wind driven ocean circulation, J. Meteorol., 7, 79–93, 1950.

Peckham, S. D., Hutton, E. W., and Norris, B.: A component-based approach to integrated modeling in the geosciences: The design of CSDMS, Comput. Geosci., 53, 3–12, 2013.

Pedlosky, J.: Ocean Circulation Theory, Springer, 1996.

Pelupessy, F. I., van Elteren, A., de Vries, N., McMillan, S. L. W., Drost, N., and Portegies Zwart, S. F.: The Astrophysical Multipurpose Software Environment, Astron. Astrophys., 557, 23 pp., 2013.

Portegies Zwart, S., McMillan, S., Harfst, S., Groen, D., Fujii, M., Nualláin, B. Ó., Glebbeek, E., Heggie, D., Lombardi, J., Hut, P., Angelou, V., Banerjee, S., Belkus, H., Fragos, T., Fregeau, J., Gaburov, E., Izzard, R., Jurić, M., Justham, S., Sottoriva, A., Teuben, P., van Bever, J., Yaron, O., and Zemp, M.: A multiphysics and multiscale software environment for modeling astrophysical systems, New Astron., 14, 369–378, https://doi.org/10.1016/j.newast.2008.10.006, 2009.

Portegies Zwart, S., McMillan, S. L. W., van Elteren, E., Pelupessy, I., and de Vries, N.: Multi-physics simulations using a hierarchi-

cal interchangeable software interface, Comput. Phys. Commun., 183, 456–468, 2013.

Rew, R. and Davis, G.: NetCDF: an interface for scientific data access, IEEE Comput. Graph., 10, 76–82, https://doi.org/10.1109/38.56302, 1990.

Seinstra, F. J., Maassen, J., Van Nieuwpoort, R. V., Drost, N., Van Kessel, T., Van Werkhoven, B., Urbani, J., Jacobs, C., Kielmann, T., and Bal, H. E.: Jungle computing: Distributed supercomputing beyond clusters, grids, and clouds, in: Grids, Clouds and Virtualization, Springer, 167–197, 2011.

Sheremet, V. A., Ierley, G. R., and Kamenkovich, V. M.: Eigenanalysis of the two-dimensional wind-driven ocean circulation problem, J. Mar. Res., 55, 57–92, 1997.

Smith, R. D., Jones, P. W., Briegleb, B., Bryan, F., Danabasoglu, G., Dennis, J., Dukowicz, J., Eden, C., Fox-Kemper, B., Gent, P., Hecht, M. , Jayne, S., Jochum, M. , Large, W., Lindsay, K., Maltrud, M., Norton, N., Peacock, S., Vertenstein, M., and Yeager, S.: The Parallel Ocean Program (POP) reference manual, Los Alamos National Laboratory, LAUR-10-01853, 2010.

Stommel, H.: The westward intensification of wind-driven ocean currents, EOS T. Am. Geophys. Un., 22, 202–206, 1948.

Sverdrup, H. U.: Wind-driven currents in a baroclinic ocean; with application to the equatorial currents of the eastern Pacific, P. Natl. Acad. Sci. USA, 33, 318–326, 1947.

SWAN: Scientific and Technical Documentation, Delft University, the Netherlands, 2015.

Valcke, S.: The OASIS3 coupler: a European climate modelling community software, Geosci. Model Dev., 6, 373–388, https://doi.org/10.5194/gmd-6-373-2013, 2013.

Valcke, S., Balaji, V., Craig, A., DeLuca, C., Dunlap, R., Ford, R. W., Jacob, R., Larson, J., O'Kuinghttons, R., Riley, G. D., and Vertenstein, M.: Coupling technologies for Earth System Modelling, Geosci. Model Dev., 5, 1589–1596, https://doi.org/10.5194/gmd-5-1589-2012, 2012.

van Rossum, G.: Python tutorial, Technical Report CS-R9526, Centrum voor Wiskunde en Informatica, Amsterdam, the Netherlands, 1995.

van Werkhoven, B., Maassen, J., Kliphuis, M., Dijkstra, H. A., Brunnabend, S. E., van Meersbergen, M., Seinstra, F. J., and Bal, H. E.: A distributed computing approach to improve the performance of the Parallel Ocean Program (v2.1), Geosci. Model Dev., 7, 267–281, https://doi.org/10.5194/gmd-7-267-2014, 2014.

Viebahn, J. and Dijkstra, H. A.: Critical Transition Analysis of the Deterministic Wind-Driven Ocean Circulation – A Flux-Based Network Approach, Int. J. Bifurcat. Chaos, 24, 1430007, https://doi.org/10.1142/S0218127414300079, 2014.

Viebahn, J. and Eden, C.: Towards the impact of eddies on the response of the Southern Ocean to climate change, Ocean Model., 34, 150–165, 2010.

Zijlema, M.: Computation of wind-wave spectra in coastal waters with SWAN on unstructured grids, Coast. Eng., 57, 267–277, 2010.