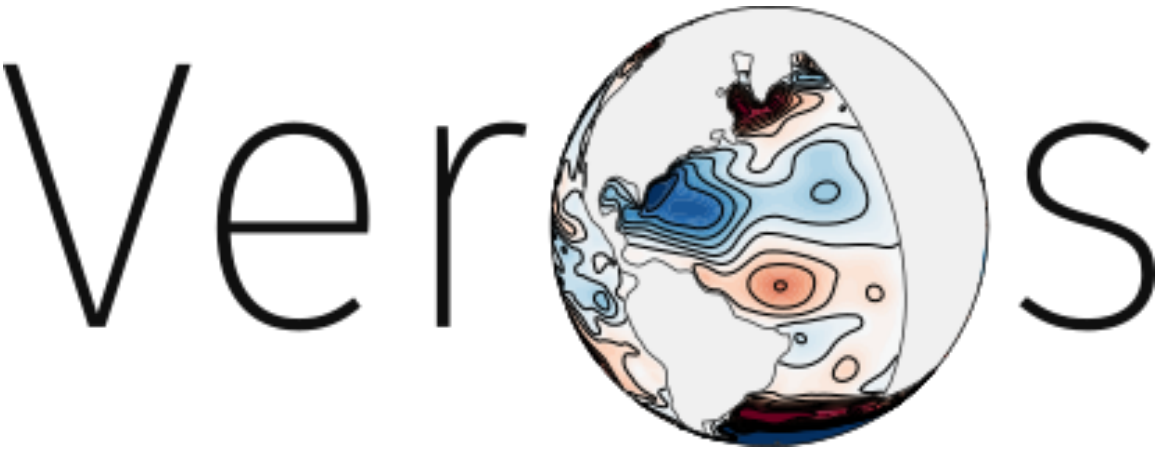
Veros Documentation

Release 0.0.1b

The Veros Team

Feb 16, 2018

Quick Start



Veros, *the versatile ocean simulator*, aims to be the swiss army knife of ocean modeling. It is a full-fledged GCM (general circulation model) that supports anything between highly idealized configurations and realistic set-ups, targeting students and seasoned researchers alike. Thanks to its seamless interplay with Bohrium, Veros runs efficiently on your laptop, gaming PC (with experimental GPU support through OpenCL & CUDA), and small cluster. Veros has a clear focus on simplicity, usability, and adaptability - *because the Baroque is over*.

If you want to learn more about the background and capabilities of Veros, you should check out *A short introduction to Veros*. If you are already convinced, you can jump right into action, and *learn how to get started* instead!

CHAPTER 1

A short introduction to Veros

1.1 The vision

Veros is an adaptation of pyOM2 (v2.1.0), developed by Carsten Eden (Institut für Meereskunde, Hamburg University). In contrast to pyOM2, however, this implementation does not rely on a Fortran backend for computations - everything runs in pure Python, down to the last parameterization. We belive that using this approach it is possible to create an open source ocean model that is:

1. **Easy to access**: Python modules are simple to install, and projects like Anaconda are doing a great job in creating platform-independent environments.
2. **Easy to use**: Anyone with some experience can use their favorite Python tools to set up, control, and post-process Veros.
3. **Easy to verify**: Python code tends to be concise and easy to read, even for people with little practical programming experience. This enables a wide range of people to spot errors in our code, solidifying it in the process.
4. **Easy to modify**: Due to the popularity of Python, its dynamic code structure, and OOP (object-oriented programming)-capabilities, Veros can be extended and modified with minimal effort.

However, using Python over a compiled language like Fortran usually comes at a high computational cost. We try to overcome this gap for large models by providing an interface to Bohrium, a framework that acts as a high-performance replacement for NumPy. Bohrium takes care of all parallelism in the background for us, so we can concentrate on writing a nice, readable ocean model.

In case you are curious about how Veros is currently stacking up against pyOM2 in terms of performance, you should check out *our benchmarks*.

1.2 Features

Note: This section provides a quick overview of the capabilities and limitations of Veros. For a comprehensive description of the physics and numerics behind Veros, please refer to the documentation of pyOM2. You can also

obtain a copy of the PDF documentation `here`.

1.2.1 The model domain

The numerical solution is calculated using finite differences on an *Arakawa C-grid*, which is staggered in every dimension. *Tracers* (like temperature and salinity) are calculated at different positions than zonal, meridional, and vertical *fluxes* (like the velocities u, v, and w). The following figure shows the relative positions of the so-called T, U, V, and ζ grid points (W not shown):

Fig. 1.1: The structure of the Arakawa C-grid.

Veros supports both Cartesian and pseudo-spherical (i.e., including additional metric terms) coordinate systems. Islands or holes in the domain are fully supported by the streamfunction solver. Zonal boundaries can either be cyclic or regraded as walls (with free-slip boundary conditions).

1.2.2 Available parameterizations

At its core, Veros currently offers the following solvers, numerical schemes, parameterizations, and closures:

Surface pressure:

- a streamfunction solver with MOM's island algorithm and an iterative Poisson solver

Equation of state:

- the full 48-term TEOS equation of state
- various linear and nonlinear model equations from *[Vallis2006]*

Friction:

- harmonic or biharmonic lateral friction
- linear or quadratic bottom friction
- interior Rayleigh friction
- explicit or fully implicit harmonic vertical friction

Advection:

- a classical second-order central difference scheme
- a second-order scheme with a superbee flux-limiter

Diffusion:

- harmonic or biharmonic lateral diffusion
- explicit or implicit harmonic vertical diffusion

Isoneutral mixing:

- lateral mixing of tracers along neutral surfaces following *[Griffies1998]* (optional)

Internal wave breaking:

- IDEMIX as in *[OlbersEden2013]* (optional)

EKE model (eddy kinetic energy):

- meso-scale eddy mixing closure after *[Gent1995]*, either with constant coefficients or calculated using the prognostic EKE closure by *[EdenGreatbatch2008]* (optional)

TKE model (turbulent kinetic energy):

- prognostic TKE model for vertical mixing as introduced in *[Gaspar1990]* (optional)

1.2.3 Diagnostics

Diagnostics are reponsible for handling all model output, runtime checks of the solution, and restart file handling. They are implemented in a modular fashion, so additional diagnostics can be implemented easily. Already implemented diagnostics handle snapshot output, time-averaging of variables, monitoring of energy fluxes, and calculation of the overturning streamfunction.

For more information, see *Diagnostics*.

1.2.4 Pre-configured model setups

Veros supports a wide range of model configurations. Several setups are already implemented that highlight some of the capabilities of Veros, and that serve as a basis for users to set up their own configuration: *Setup gallery*.

1.2.5 Current limitations

Veros is still in early development. There are several open issues that we would like to fix later on:

Physics:

- Veros does not yet implement any of the more recent pyOM2.2 features such as the ROSSMIX parameterization, IDEMIX v3.0, open boundary conditions, or cyclic meridional boundaries. It neither implements all of pyOM2.1's features - missing are e.g. the non-hydrostatic solver, IDEMIX v2.0, and the surface pressure solver.
- Since the grid is required to be rectilinear, there is currently no natural way to handle the singularity at the North Pole. The northern and southern boundaries of the domain are thus always "walls".
- There is currently no ice sheet model in Veros. Some realistic setups employ a simple ice mask that cut off atmospheric forcing for water that gets too cold instead.

Technical issues:

- The GPU backend is experimental, and there is still some work required on both Veros and Bohrium to make it run efficiently.
- Python 3.x is not yet fully supported due to some issues with Bohrium.

1.2.6 References

CHAPTER 2

Getting started

2.1 Installation

2.1.1 Using Anaconda (multi-platform)

1. Download and install Anaconda. Make sure to grab the 64-bit version of the Python 2.7 interpreter.
2. Install some dependencies:

```
$ conda install hdf5 libnetcdf
$ conda install -c conda-forge git-lfs
```

   and optionally:

```
$ conda install -c bohrium bohrium
```

3. Clone our repository:

```
$ git clone https://github.com/dionhaefner/veros.git
```

4. Install Veros via:

```
$ pip install -e ./veros
```

2.1.2 Using apt-get (Ubuntu / Debian)

1. Install some dependencies:

```
$ sudo apt-get install git python-dev python-pip libhdf5-dev libnetcdf-dev
```

   and optionally:

```
$ sudo add-apt-repository ppa:bohrium/nightly
$ sudo apt-get update
$ sudo apt-get install bohrium
```

If you want to clone the input files needed for running the larger setups, you will also need to install git lfs.

2. Clone our repository:

```
$ git clone https://github.com/dionhaefner/veros.git
```

3. Install Veros via:

```
$ pip install -e ./veros
```

2.2 Setting up a model

To run Veros, you need to set up a model - i.e., specify which settings and model domain you want to use. This is done by subclassing the `Veros base class` in a *setup script* that is written in Python. You should have a look at the pre-implemented model setups in the repository's `setup` folder, or use the **`veros copy-setup`** command to copy one into your current folder. A good place to start is the `ACC model`:

```
$ veros copy-setup acc
```

By working through the existing models, you should quickly be able to figure out how to write your own simulation. Just keep in mind this genral advice:

- You can (and should) use any (external) Python tools you want in your model setup. Before implementing a certain functionality, you should check whether it is already provided by a common library. Especially the SciPy module family provides countless implementations of common scientific functions (and SciPy is installed along with Veros).
- If you decorate your methods with `@veros_method`, the variable `np` inside that function will point to the currently used backend (i.e., NumPy or Bohrium). Thus, if you want your setup to be able to dynamically switch between backends, you should write your methods like this:

```
from veros import Veros, veros_method

class MyVerosSetup(Veros):
    ...
    @veros_method
    def my_function(self):
        arr = np.array([1,2,3,4]) # "np" uses either NumPy or Bohrium
```

- If you are curious about the general procedure in which a model is set up and ran, you should read the source code of `veros.Veros` (especially the `setup()` and `run()` methods). This is also the best way to find out about the order in which methods and routines are called.
- Out of all functions that need to be imlpemented by your subclass of `veros.Veros`, the only one that is called in every time step is `set_forcing()` (at the beginning of each iteration). This implies that, to achive optimal performance, you should consider moving calculations that are constant in time to other functions.

If you want to learn more about setting up advanced configurations, you should *check out our tutorial* that walks you through the creation of a realistic configuration with an idealized Atlantic.

2.3 Running Veros

After adapting your setup script, you are ready to run your first simulation. It is advisable to include something like:

```
if __name__ == "__main__":
    simulation = MyVerosSetup()
    simulation.setup()
    simulation.run()
```

in your setup file, so you can run it as a script:

```
$ python my_setup.py
```

However, you are not required to do so, and you are welcome to write include your simulation class into other Python files and call it dynamically or interactively (e.g. in an IPython session).

All Veros setups accept additional options via the command line when called as a script or as arguments to their `__init__()` function when called from another Python module. You can check the available commands through

```
$ python my_setup.py --help
```

2.3.1 Reading Veros output

All output is handled by *the available diagnostics*. The most basic diagnostic, snapshot, writes *some model variables* to netCDF files in regular intervals (and puts them into your current working directory).

NetCDF is a binary format that is widely adopted in the geophysical modeling community. There are various packages for reading, visualizing and processing netCDF files (such as ncview and ferret), and bindings for many programming languages (such as C, Fortran, MATLAB, and Python).

In fact, after installing Veros, you will already have installed the netCDF bindings for Python, so reading data from an output file and plotting it is as easy as:

```
import matplotlib.pyplot as plt
from netCDF4 import Dataset

with Dataset("veros.snapshot.nc", "r") as datafile:
    # read variable "u" and save it to a NumPy array
    u = datafile.variables["u"][...]

# plot surface velocity at the last time step included in the file
plt.imshow(u[-1, -1, ...])
plt.show()
```

For further reference refer to the netcdf4-python documentation.

2.3.2 Using Bohrium

Warning: While Bohrium yields significant speed-ups for large to very large setups, the overhead introduced by Bohrium often leads to (sometimes considerably) slower execution for problems below a certain threshold size (see also *Which backend should I choose to run my model (NumPy / Bohrium)?*). You are thus advised to test carefully whether Bohrium is beneficial in your particular use case.

For large simulations, it is often beneficial to use the Bohrium backend for computations. When using Bohrium, all number crunching will make full use of your available architecture, i.e., computations are executed in parallel on all of your CPU cores, or even GPU when using `BH_STACK=opencl` or `BH_STACK=cuda` (experimental). You may switch between NumPy and Bohrium with a simple command line switch:

```
$ python my_setup.py -b bohrium
```

or, when running inside another Python module:

```
simulation = MyVerosSetup(backend="bohrium")
```

2.3.3 Re-starting from a previous run

Restart data (in HDF5 format) is written at the end of each simulation or after a regular time interval if the setting restart_frequency is set to a finite value. To use this restart file as initial conditions for another simulation, you will have to point restart_input_filename of the new simulation to the corresponding restart file. This can (as all settings) also be given via command line:

```
$ python my_setup.py -s restart_input_filename /path/to/restart_file.h5
```

2.4 Enhancing Veros

Veros was written with extensibility in mind. If you already know some Python and have worked with NumPy, you are pretty much ready to write your own extension. The model code is located in the `veros` subfolder, while all of the numerical routines are located in `veros/core`.

We believe that the best way to learn how Veros works is to read its source code. Starting from the `Veros base class`, you should be able to work your way through the flow of the program, and figure out where to add your modifications. If you installed Veros through **`pip -e`** or **`setup.py develop`**, all changes you make will immediately be reflected when running the code.

In case you want to add additional output capabilities or compute additional quantities without changing the main solution of the simulation, you should consider *adding a custom diagnostic*.

A convenient way to implement your modifications is to create your own fork of Veros on GitHub, and submit a pull request if you think your modifications could be useful for the Veros community.

2.4.1 Code conventions

When contributing to Veros, please adhere to the following general guidelines:

- Your first guide should be the surrounding Veros code. Look around, and be consistent with your modifications.
- Unless you have a very good reason not to do so, please stick to the PEP8 style guide throughout your code. One exception we make in Veros is in regard to the maximum line length - since numerical operations can take up quite a lot of horizontal space, you may use longer lines if it increases readability.
- Please follow the PEP8 naming conventions, and use meaningful, telling names for your variables, functions, and classes. The variable name `stretching_factor` is infinitely more meaningful than `k`. This is especially important for settings and generic helper functions.
- "Private" helper functions that are not meant to be called from outside the current source file should be prefixed with an underscore (_).

- Use double quotes (") for all strings longer than a single character.
- Document your functions using Google-style docstrings. This is especially important if you are implementing a user-facing API (such as a diagnostic, a setup, or tools that are meant to be called from setups).

2.4.2 Running tests and benchmarks

If you want to make sure that your changes did not break anything, you can run our test suite that compares the results of each subroutine to pyOM2. To do that, you will need to compile the Python interface of pyOM2 on your machine, and then point the testing suite to the library location, e.g. through:

```
$ python run_tests.py /path/to/pyOM2/py_src/pyOM_code.so
```

from Veros's `test` folder.

If you deliberately introduced breaking changes, you can disable them during testing by prefixing them with:

```
if not vs.pyom_compatibility_mode:
    # your changes
```

Automated benchmarks are provided in a similar fashion. The benchmarks run some dummy problems with varying problem sizes and all available computational backends: `numpy`, `bohrium-openmp`, `bohrium-opencl`, `bohrium-cuda`, `fortran` (pyOM2), and `fortran-mpi` (parallel pyOM2). For options and further information run:

```
$ python run_benchmarks.py --help
```

from the `test` folder. Timings are written in JSON format.

2.4.3 Performance tweaks

If your changes to Veros turn out to have a negative effect on the runtime of the model, there several ways to investigate and solve performance problems:

- Run your model with the `-v debug` option to get additional debugging output (such as timings for each time step, and a timing summary after the run has finished).
- Run your model with the `-p` option to profile Veros with pyinstrument. You may have to run **`pip install pyinstrument`** before being able to do so. After completion of the run, a file `profile.html` will be written that can be opened with a web browser and contains timings for the entire call stack.
- You should try and avoid explict loops over arrays at all cost (even more so when using Bohrium). You should always try to work on the whole array at once.
- When using Bohrium, it is sometimes beneficial to copy an array to NumPy before passing it to an external module or performing an operation that cannot be vectorized efficiently. Just don't forget to copy it back to Bohrium after you are finished, e.g. like so:

  ```
  if vs.backend_name == "bohrium":
      u_np = vs.u.copy2numpy()
  else:
      u_np = vs.u
  vs.u[...] = np.asarray(external_function(u_np))
  ```

- If you are still having trouble, don't hesitate to ask for help (e.g. on GitHub).

CHAPTER 3

Creating an advanced model setup

Note: This guide is still work in progress.

This is a step-by-step guide that illustrates how even complicated setups can be created with relative ease (thanks to the tools provided by the scientific Python community). As an example, we will re-create the *wave propagation setup*, which is a global ocean model with an idealized Atlantic.

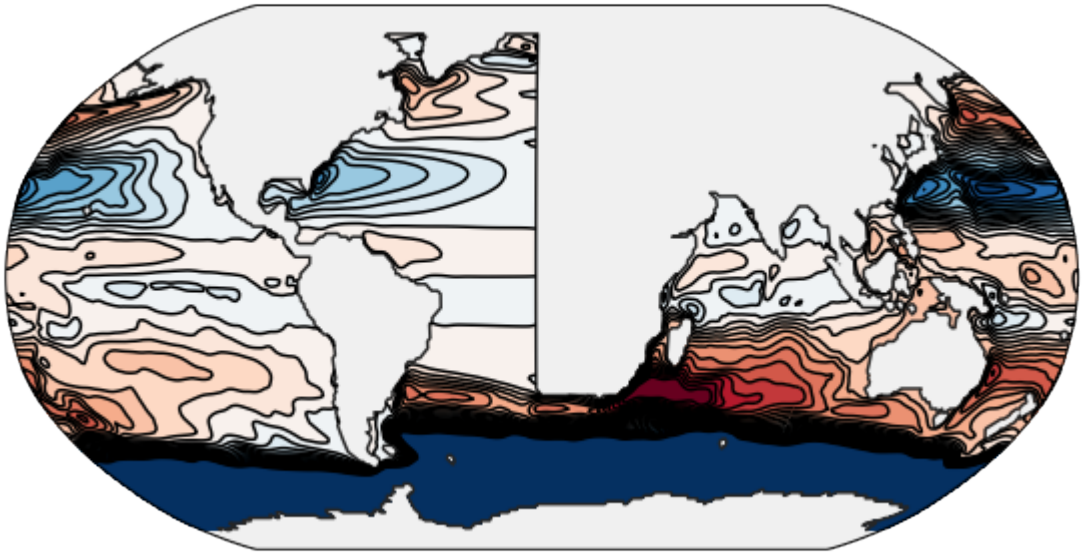


Fig. 3.1: The resulting stream function after about 1 year of integration.

3.1 The vision

The purpose of this model is to examine wave propagation along the eastern boundary of the North Atlantic. Since it is difficult to track propagating waves along ragged geometry or through uneven forcing fields, we will idealize the representation of the North Atlantic; and as the presence of the Pacific in the model is crucial to achieve a realistic ocean circulation, we want to use a global model.

This leaves us with the following requirements for the final wave propagation model:

1. A global model with a resolution of around 2 degrees and meridional stretching.
2. Convert the eastern boundary of the Atlantic to a straight line, so analytically derived wave properties hold.
3. A refined grid resolution at the eastern boundary of the Atlantic.
4. Zonally averaged forcings in the Atlantic.
5. A somehow interpolated initial state for cells that have been converted from land to ocean in the North Atlantic.
6. Options for shelf and continental slope.
7. A multiplier setting for the Southern Ocean wind stress.

3.2 Model skeleton

Instead of starting from scratch, we can use the *global one degree model* as a template, which looks like this:

```
#!/usr/bin/env python

import os
from netCDF4 import Dataset

from veros import Veros, veros_method, time, tools

BASE_PATH = os.path.dirname(os.path.realpath(__file__))
FORCING_FILE = os.path.join(BASE_PATH, "forcing_1deg_global.nc")


class GlobalOneDegree(Veros):
    """Global 1 degree model with 115 vertical levels.

    `Adapted from pyOM2 <https://wiki.zmaw.de/ifm/TO/pyOM2/1x1%20global%20model>`_.
    """

    @veros_method
    def set_parameter(self):
        """
        set main parameters
        """
        if not os.path.isfile(FORCING_FILE):
            raise RuntimeError("{} data file {} not found".format(name, filepath))

        self.nx = 360
        self.ny = 160
        self.nz = 115
        self.dt_mom = 1800.0
        self.dt_tracer = 1800.0
        self.runlen = 0.
```

```
        self.coord_degree = True
        self.enable_cyclic_x = True

        self.congr_epsilon = 1e-10
        self.congr_max_iterations = 10000

        self.enable_hor_friction = True
        self.A_h = 5e4
        self.enable_hor_friction_cos_scaling = True
        self.hor_friction_cosPower = 1
        self.enable_tempsalt_sources = True
        self.enable_implicit_vert_friction = True

        self.eq_of_state_type = 5

        # isoneutral
        self.enable_neutral_diffusion = True
        self.K_iso_0 = 1000.0
        self.K_iso_steep = 50.0
        self.iso_dslope = 0.005
        self.iso_slopec = 0.005
        self.enable_skew_diffusion = True

        # tke
        self.enable_tke = True
        self.c_k = 0.1
        self.c_eps = 0.7
        self.alpha_tke = 30.0
        self.mxl_min = 1e-8
        self.tke_mxl_choice = 2
        self.enable_tke_superbee_advection = True

        # eke
        self.enable_eke = True
        self.eke_k_max = 1e4
        self.eke_c_k = 0.4
        self.eke_c_eps = 0.5
        self.eke_cross = 2.
        self.eke_crhin = 1.0
        self.eke_lmin = 100.0
        self.enable_eke_superbee_advection = True
        self.enable_eke_isopycnal_diffusion = True

        # idemix
        self.enable_idemix = True
        self.enable_eke_diss_surfbot = True
        self.eke_diss_surfbot_frac = 0.2
        self.enable_idemix_superbee_advection = True
        self.enable_idemix_hor_diffusion = True

    @veros_method
    def _read_forcing(self, var):
        with Dataset(FORCING_FILE, "r") as infile:
            return np.array(infile.variables[var][...]).T

    @veros_method
    def set_grid(self):
```

```
        dz_data = self._read_forcing("dz")
        self.dzt[...] = dz_data[::-1]
        self.dxt[...] = 1.0
        self.dyt[...] = 1.0
        self.y_origin = -79.
        self.x_origin = 91.

    @veros_method
    def set_coriolis(self):
        self.coriolis_t[...] = 2 * self.omega * np.sin(self.yt[np.newaxis, :] / 180.␣
↪* self.pi)

    @veros_method
    def set_topography(self):
        bathymetry_data = self._read_forcing("bathymetry")
        salt_data = self._read_forcing("salinity")[:, :, ::-1]

        mask_salt = salt_data == 0.
        self.kbot[2:-2, 2:-2] = 1 + np.sum(mask_salt.astype(np.int), axis=2)

        mask_bathy = bathymetry_data == 0
        self.kbot[2:-2, 2:-2][mask_bathy] = 0

        self.kbot[self.kbot >= self.nz] = 0

        # close some channels
        i, j = np.indices((self.nx, self.ny))

        mask_channel = (i >= 207) & (i < 214) & (j < 5)  # i = 208,214; j = 1,5
        self.kbot[2:-2, 2:-2][mask_channel] = 0

        # Aleutian islands
        mask_channel = (i == 104) & (j == 134)  # i = 105; j = 135
        self.kbot[2:-2, 2:-2][mask_channel] = 0

        # Engl channel
        mask_channel = (i >= 269) & (i < 271) & (j == 130)  # i = 270,271; j = 131
        self.kbot[2:-2, 2:-2][mask_channel] = 0

    @veros_method
    def set_initial_conditions(self):
        self.t_star = np.zeros((self.nx + 4, self.ny + 4, 12), dtype=self.default_
↪float_type)
        self.s_star = np.zeros((self.nx + 4, self.ny + 4, 12), dtype=self.default_
↪float_type)
        self.qnec = np.zeros((self.nx + 4, self.ny + 4, 12), dtype=self.default_float_
↪type)
        self.qnet = np.zeros((self.nx + 4, self.ny + 4, 12), dtype=self.default_float_
↪type)
        self.qsol = np.zeros((self.nx + 4, self.ny + 4, 12), dtype=self.default_float_
↪type)
        self.divpen_shortwave = np.zeros(self.nz, dtype=self.default_float_type)
        self.taux = np.zeros((self.nx + 4, self.ny + 4, 12), dtype=self.default_float_
↪type)
        self.tauy = np.zeros((self.nx + 4, self.ny + 4, 12), dtype=self.default_float_
↪type)

        rpart_shortwave = 0.58
```

```
        efold1_shortwave = 0.35
        efold2_shortwave = 23.0

        # initial conditions
        temp_data = self._read_forcing("temperature")
        self.temp[2:-2, 2:-2, :, 0] = temp_data[..., ::-1] * self.maskT[2:-2, 2:-2, :]
        self.temp[2:-2, 2:-2, :, 1] = temp_data[..., ::-1] * self.maskT[2:-2, 2:-2, :]

        salt_data = self._read_forcing("salinity")
        self.salt[2:-2, 2:-2, :, 0] = salt_data[..., ::-1] * self.maskT[2:-2, 2:-2, :]
        self.salt[2:-2, 2:-2, :, 1] = salt_data[..., ::-1] * self.maskT[2:-2, 2:-2, :]

        # wind stress on MIT grid
        taux_data = self._read_forcing("tau_x")
        self.taux[2:-2, 2:-2, :] = taux_data / self.rho_0

        tauy_data = self._read_forcing("tau_y")
        self.tauy[2:-2, 2:-2, :] = tauy_data / self.rho_0

        # Qnet and dQ/dT and Qsol
        qnet_data = self._read_forcing("q_net")
        self.qnet[2:-2, 2:-2, :] = -qnet_data * self.maskT[2:-2, 2:-2, -1, np.newaxis]

        qnec_data = self._read_forcing("dqdt")
        self.qnec[2:-2, 2:-2, :] = qnec_data * self.maskT[2:-2, 2:-2, -1, np.newaxis]

        qsol_data = self._read_forcing("swf")
        self.qsol[2:-2, 2:-2, :] = -qsol_data * self.maskT[2:-2, 2:-2, -1, np.newaxis]

        # SST and SSS
        sst_data = self._read_forcing("sst")
        self.t_star[2:-2, 2:-2, :] = sst_data * self.maskT[2:-2, 2:-2, -1, np.newaxis]

        sss_data = self._read_forcing("sss")
        self.s_star[2:-2, 2:-2, :] = sss_data * self.maskT[2:-2, 2:-2, -1, np.newaxis]

        if self.enable_idemix:
            tidal_energy_data = self._read_forcing("tidal_energy")
            mask = np.maximum(0, self.kbot[2:-2, 2:-2] - 1)[:, :, np.newaxis] == np.
↪arange(self.nz)[np.newaxis, np.newaxis, :]
            tidal_energy_data[:, :] *= self.maskW[2:-2, 2:-2, :][mask].reshape(self.
↪nx, self.ny) / self.rho_0
            self.forc_iw_bottom[2:-2, 2:-2] = tidal_energy_data

            wind_energy_data = self._read_forcing("wind_energy")
            wind_energy_data[:, :] *= self.maskW[2:-2, 2:-2, -1] / self.rho_0 * 0.2
            self.forc_iw_surface[2:-2, 2:-2] = wind_energy_data

        """
        Initialize penetration profile for solar radiation and store divergence in␣
↪divpen
        note that pen is set to 0.0 at the surface instead of 1.0 to compensate for␣
↪the
        shortwave part of the total surface flux
        """
        swarg1 = self.zw / efold1_shortwave
        swarg2 = self.zw / efold2_shortwave
        pen = rpart_shortwave * np.exp(swarg1) + (1.0 - rpart_shortwave) * np.
↪exp(swarg2)
```

```
        pen[-1] = 0.
        self.divpen_shortwave = np.zeros(self.nz, dtype=self.default_float_type)
        self.divpen_shortwave[1:] = (pen[1:] - pen[:-1]) / self.dzt[1:]
        self.divpen_shortwave[0] = pen[0] / self.dzt[0]


    @veros_method
    def set_forcing(self):
        t_rest = 30. * 86400.
        cp_0 = 3991.86795711963  # J/kg /K

        year_in_seconds = time.convert_time(self, 1., "years", "seconds")
        (n1, f1), (n2, f2) = tools.get_periodic_interval(self.time, year_in_seconds,
                                                         year_in_seconds / 12., 12)

        # linearly interpolate wind stress and shift from MITgcm U/V grid to this grid
        self.surface_taux[:-1, :] = f1 * self.taux[1:, :, n1] + f2 * self.taux[1:, :,␣
↪n2]
        self.surface_tauy[:, :-1] = f1 * self.tauy[:, 1:, n1] + f2 * self.tauy[:, 1:,␣
↪n2]

        if self.enable_tke:
            self.forc_tke_surface[1:-1, 1:-1] = np.sqrt((0.5 * (self.surface_taux[1:-
↪1, 1:-1] \
                                                                + self.surface_taux[:-2,
↪ 1:-1])) ** 2 \
                                                        + (0.5 * (self.surface_tauy[1:-
↪1, 1:-1] \
                                                                + self.surface_tauy[1:-
↪1, :-2])) ** 2) ** (3. / 2.)

        # W/m^2 K kg/J m^3/kg = K m/s
        t_star_cur = f1 * self.t_star[..., n1] + f2 * self.t_star[..., n2]
        self.qqnec = f1 * self.qnec[..., n1] + f2 * self.qnec[..., n2]
        self.qqnet = f1 * self.qnet[..., n1] + f2 * self.qnet[..., n2]
        self.forc_temp_surface[...] = (self.qqnet + self.qqnec * (t_star_cur - self.
↪temp[..., -1, self.tau])) \
            * self.maskT[..., -1] / cp_0 / self.rho_0
        s_star_cur = f1 * self.s_star[..., n1] + f2 * self.s_star[..., n2]
        self.forc_salt_surface[...] = 1. / t_rest * \
            (s_star_cur - self.salt[..., -1, self.tau]) * self.maskT[..., -1] * self.
↪dzt[-1]

        # apply simple ice mask
        mask1 = self.temp[:, :, -1, self.tau] * self.maskT[:, :, -1] <= -1.8
        mask2 = self.forc_temp_surface <= 0
        ice = ~(mask1 & mask2)
        self.forc_temp_surface *= ice
        self.forc_salt_surface *= ice

        # solar radiation
        if self.enable_tempsalt_sources:
            self.temp_source[..., :] = (f1 * self.qsol[..., n1, None] + f2 * self.
↪qsol[..., n2, None]) \
                * self.divpen_shortwave[None, None, :] * ice[..., None] \
                * self.maskT[..., :] / cp_0 / self.rho_0

    @veros_method
```

```
    def set_diagnostics(self):
        average_vars = ["surface_taux", "surface_tauy", "forc_temp_surface", "forc_
↪salt_surface",
                        "psi", "temp", "salt", "u", "v", "w", "Nsqr", "Hd", "rho",
                        "K_diss_v", "P_diss_v", "P_diss_nonlin", "P_diss_iso", "kappaH
↪"]
        if self.enable_skew_diffusion:
            average_vars += ["B1_gm", "B2_gm"]
        if self.enable_TEM_friction:
            average_vars += ["kappa_gm", "K_diss_gm"]
        if self.enable_tke:
            average_vars += ["tke", "Prandtlnumber", "mxl", "tke_diss",
                             "forc_tke_surface", "tke_surf_corr"]
        if self.enable_idemix:
            average_vars += ["E_iw", "forc_iw_surface", "forc_iw_bottom", "iw_diss",
                             "c0", "v0"]
        if self.enable_eke:
            average_vars += ["eke", "K_gm", "L_rossby", "L_rhines"]

        self.diagnostics["averages"].output_variables = average_vars
        self.diagnostics["cfl_monitor"].output_frequency = 86400.0
        self.diagnostics["snapshot"].output_frequency = 365 * 86400 / 24.
        self.diagnostics["overturning"].output_frequency = 365 * 86400
        self.diagnostics["overturning"].sampling_frequency = 365 * 86400 / 24.
        self.diagnostics["energy"].output_frequency = 365 * 86400
        self.diagnostics["energy"].sampling_frequency = 365 * 86400 / 24.
        self.diagnostics["averages"].output_frequency = 365 * 86400
        self.diagnostics["averages"].sampling_frequency = 365 * 86400 / 24.


if __name__ == "__main__":
    simulation = GlobalOneDegree()
    simulation.setup()
    simulation.run()
```

The biggest changes in the new wave propgation setup will be located in the `set_grid()` `set_topography()` and `set_initial_conditions()` methods to accomodate for the new geometry and the interpolation of initial conditions to the modified grid, so we can concentrate on implementing those first.

3.3 Step 1: Setup grid

Warning: When using a non-uniform grid,

3.4 Step 2: Create idealized topography

Usually, to create an idealized topography, one would simply hand-craft some input and forcing files that reflect the desired changes. However, since we want our setup to have flexible resolution, we will have to write an algorithm that creates these input files for any given number of grid cells. One convenient way to achieve this is by creating some high-resolution *masks* representing the target topography by hand, and then interpolate these masks to the desired resolution.

3.4.1 Create a mask image

Before we can start, we need to download a high-resolution topography dataset. There are many freely available topographical data sets on the internet; one of them is ETOPO5 (with a resolution of 5 arc-minutes), which we will be using throughout this tutorial. To create a mask image from the topography file, you can use the *command line tool veros create-mask*, e.g. like

```
$ veros create-mask ETOPO5_Ice_g_gmt4.nc
```

This creates a one-to-one representation of the topography file as a PNG image. However, in the case of the 5 arc-minute topography, the resulting image includes a lot of small islands and complicated coastlines that might cause problems when being interpolated to a numerical grid with a much lower resolution. To address this, the *create-mask* script accepts a *scale* argument. When given, a Gaussian filter with standard deviation *scale* (in grid cells) is applied to the resulting image, smoothing out small features. The command

```
$ veros create-mask ETOPO5_Ice_g_gmt4 --scale 3 3
```

results in the following mask:

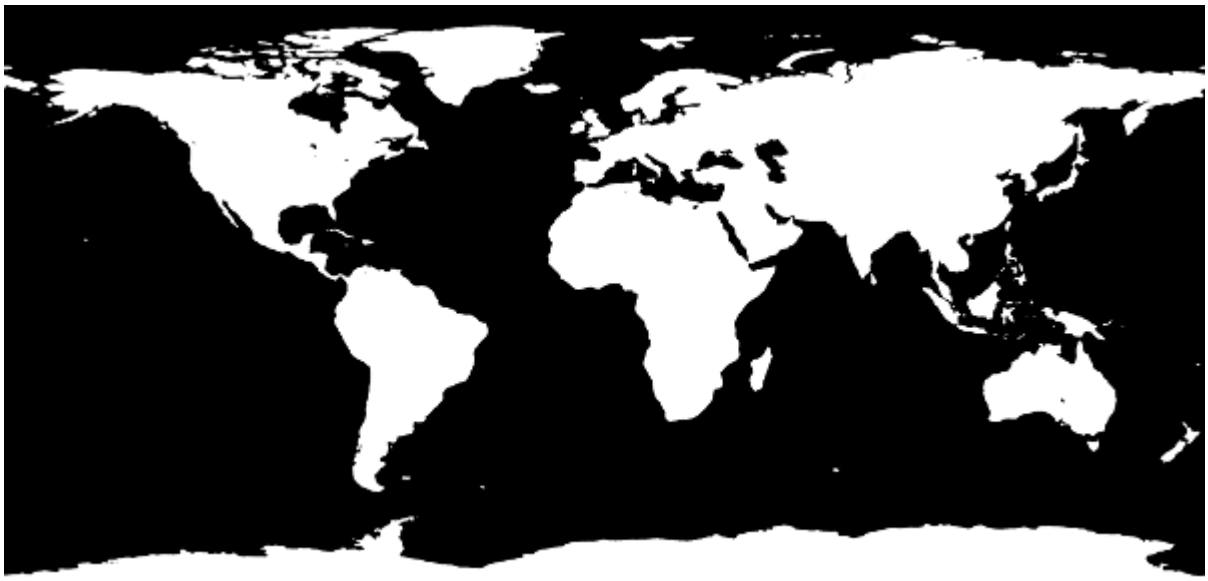


Fig. 3.2: Smoothed topography mask

which looks good enough to serve as a basis for horizontal resolutions of around one degree.

3.4.2 Modify the mask

We can now proceed to mold this realistic version of the global topography into the desired idealized shape. You can use any image editor you have availble; one possibility is the free software GIMP. Inside the editor, we can use the pencil tools to create a modified version of the topography mask:

In this modified version, I have

1. replaced the eastern boundary of the North Atlantic by a meridional line;
2. removed all lakes and inland seas;

Fig. 3.3: Idealized topography mask

3. thickened Central America (to prevent North and South America to become disconnected due to interpolation artifacts); and
4. removed the Arctic Ocean and Hudson Bay.

Now that our topography mask is finished, we can go ahead and implement it in the Veros setup!

3.4.3 Import to Veros

To read the mask in PNG format, we are going to use the Python Imaging Library (PIL).

3.5 Step 3: Interpolate forcings & initial conditions

3.6 Step 4: Set up diagnostics & final touches

Fig. 3.4: Mask to identify grid cells in the North Atlantic

CHAPTER 4

Running Veros on a cluster

Note: Since Bohrium does not (yet) support distributed memory architectures, Veros is currently limited to running on a single computational node.

This tutorial walks you through some of the most common challenges that are specific to large, shared architectures like clusters and supercomputers. In case you are still having trouble setting up or running Veros on a large architecture after reading it, you should first contact the administrator of your cluster. Otherwise, you should of course feel free to open an issue.

4.1 Installation

Probably the easiest way to install Veros on a cluster is to, once again, *use Anaconda*. Since it is mostly platform independent and does not require elevated permissions, Anaconda is the perfect way to try out Veros without too much hassle.

If you are an administrator and want to make Veros accessible to multiple users on your cluster, we recommend that you do *not* install Veros system-wide, since it severely limits the possibilities of the users: First of all, they won't be able to install additional Python modules they might want to use for post-processing or development. And second of all, the source code (and playing with it) is supposed to be a critical part of the Veros experience. Instead, you could e.g. use virtualenv to create a lightweight Python environment for every user that they can freely manage.

4.2 Usage

If you want to run Veros on a shared computing architecture, there are several issues that require special handling:

1. **Preventing timeouts**: In cloud computing, it is common that scheduling constraints limit the maximum execution time of a given process. Processes that exceed this time are killed. To prevent that long-running processes

have to be restarted manually after each timeout, one usually makes use of a *resubmit* mechanism: The long-running process is split into chunks that each finish before a timeout is triggered, with subsequent runs starting from the restart files that the previous process has written.

2. **Allocation of resources**: Most applications use MPI to distribute work across processors; however, this is not supported by Bohrium. We therefore need to make sure that just one single process on a single node is started for our simulation (Bohrium will then divide the workload among different threads using OpenMP).

To solve these issues, the scheduling manager needs to be told exactly how it should run our model, which is usually being done by writing a batch script that prepares the environment and states which resources to request. The exact set-up of such a script will vary depending on the scheduling manager running on your cluster, and how exactly you chose to install Veros and Bohrium. One possible way to write such a batch script for the scheduling manager SLURM is presented here:

```
#!/bin/bash -l
#
#SBATCH -p mycluster
#SBATCH -A myaccount
#SBATCH --job-name=veros_mysetup
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=16
#SBATCH --exclusive
#SBATCH --mail-type=ALL
#SBATCH --mail-user=your@email.xyz

# load module dependencies
module load bohrium

# only needed if not found automatically
export BH_CONFIG=/path/to/bohrium/config.ini

# if needed, you can modify the internal Bohrium compiler flags
export BH_OPENMP_COMPILER_FLG="-x c -fPIC -shared -std=gnu99 -O3 -Werror -fopenmp"

veros resubmit my_run 8 7776000 "python my_setup.py -b bohrium -v debug" --callback
↪"sbatch veros_batch.sh"
```

which is `saved as veros_batch.sh` in the model setup folder and called using `sbatch`.

This script makes use of the **`veros resubmit`** command and its `--callback` option to create a script that automatically re-runs itself in a new process after each successful run (see also *Command line tools*). Upon execution, a job is created on one node, using 16 processors in one process, that runs the Veros setup located in `my_setup.py` a total of eight times for 90 days (7776000 seconds) each, with identifier `my_run`. Note that the `--callback "sbatch veros_batch.sh"` part of the command is needed to actually create a new job after every run, to prevent the script from being killed after a timeout.

CHAPTER 5

Setup gallery

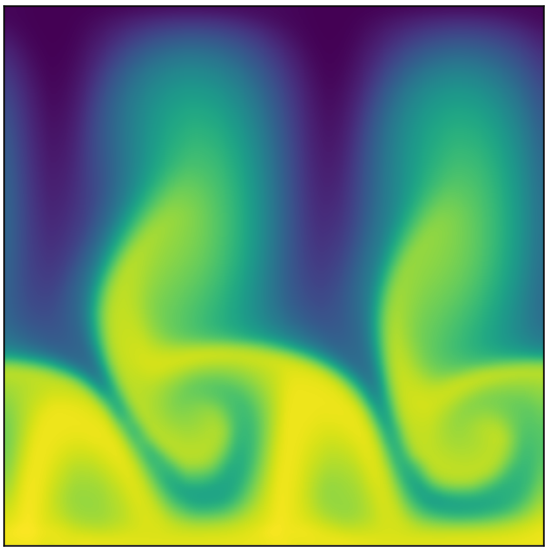
This page gives an overview of the available model setups. To copy the setup file and additional input files (if applicable) to the current working directory, you can make use of the **`veros copy-setup`** command, e.g.:

```
veros copy-setup acc
```

5.1 Idealized configurations

ACC channel model

Eady model

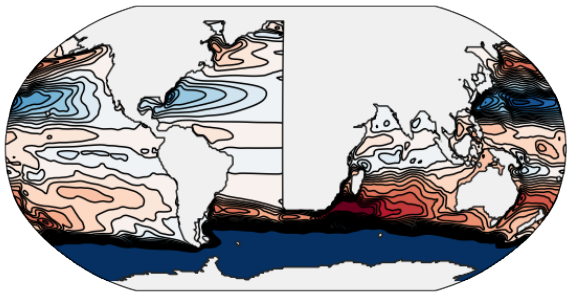


5.1.1 ACC channel model

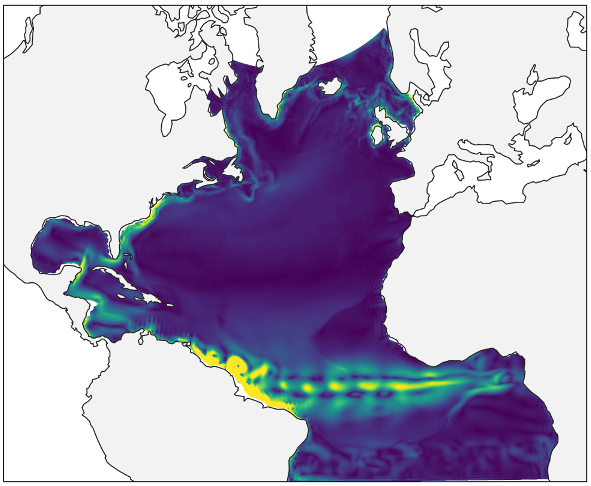
5.1.2 Eady model

5.2 Realistic configurations

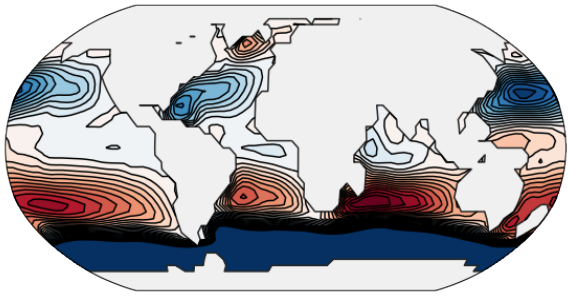
Wave propagation study



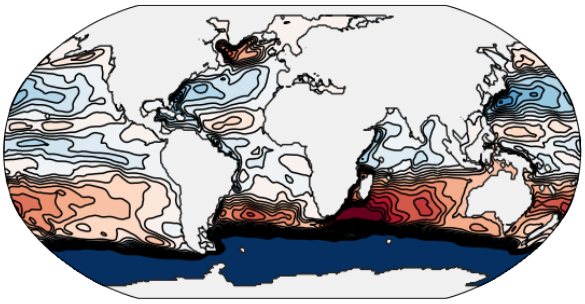
North Atlantic regional model



Global four-degree model



Global one-degree model



5.2.1 Wave propagation study

5.2.2 North Atlantic regional model

5.2.3 Global four-degree model

5.2.4 Global one-degree model

CHAPTER 6

Available settings

The following list of available settings is automatically created from the file `settings.py` in the Veros main folder. They are available as attributes of all instances of the `Veros main class`, e.g.:

```
>>> simulation = MyVerosClass()
>>> print(simulation.eq_of_state_type)
1
```

Error: Unable to execute python code at settings.rst:689:

cannot import name variables

CHAPTER 7

Model variables

The variable meta-data (i.e., all instances of `veros.variables.Variable`) are available in a dictionary as the attribute `Veros.variables`. The actual data arrays are added directly as atrributes to `Veros`. The following code snippet (as commonly used in the *Diagnostics*) illustrates this behavior:

```
var_meta = {key: val for key, val in vs.variables.items() if val.time_dependent and␣
↪val.output}
var_data = {key: getattr(veros, key) for key in var_meta.keys()}
```

In this case, `var_meta` is a dictionary containing all metadata for variables that are time dependent and should be added to the output, while `var_data` is a dictionary with the same keys containing the corresponding data arrays.

7.1 Variable class

7.2 Available variables

There are two kinds of variables in Veros. Main variables are always present in a simulation, while conditional variables are only available if their respective condition is `True` at the time of variable allocation.

Attributes:

: Time-dependent
: Included in snapshot output by default
: Written to restart files by default

Error: Unable to execute python code at variables.rst:718:

cannot import name variables

CHAPTER 8

Diagnostics

Diagnostics are separate objects (instances of subclasses of `VerosDiagnostic`) responsible for handling I/O, restart mechanics, and monitoring of the numerical solution. All available diagnostics are instantiated and added to a dictionary attribute `Veros.diagnostics` (with a key determined by their *name* attribute). Options for diagnostics may be set during the `Veros.set_diagnostics()` method:

```
class MyModelSetup(Veros):
    ...
    def set_diagnostics(self):
        self.diagnostics["averages"].output_variables = ["psi","u","v"]
        self.diagnostics["averages"].sampling_frequency = 3600.
        self.diagnostics["snapshot"].output_variables += ["du"]
```

8.1 Base class

This class implements some common logic for all diagnostics. This makes it easy to write your own diagnostics: Just derive from this class, and implement the virtual functions.

8.2 Available diagnostics

Currently, the following diagnostics are implemented and added to `Veros.diagnostics`:

8.2.1 Snapshot

8.2.2 Averages

8.2.3 CFL monitor

8.2.4 Tracer monitor

8.2.5 Energy

8.2.6 Overturning

CHAPTER 9

Command line tools

After installing Veros, you can call these scripts from the command line from any location on your system.

9.1 veros

This is a wrapper script that provides easy access to all Veros command line tools.

```
usage: veros COMMAND

available commands:
  copy-setup   Copies a Veros setup to another directory
  create-mask  Creates a mask image from a given netCDF file
  resubmit     Performs several runs of Veros back to back, using the
               previous run as restart input.  Intended to be used
               with scheduling systems (e.g. SLURM or PBS).
```

9.2 veros-create-mask

```
usage: veros-create-mask.py [-h] [-v V] [-o OUT] [-s SCALE SCALE] file

Creates a mask image from a given netCDF file

positional arguments:
  file                  Input file holding topography information

optional arguments:
  -h, --help            show this help message and exit
  -v V                  Variable holding topography data (defaults to 'z')
  -o OUT, --out OUT     Output filename (default 'topography')
  -s SCALE SCALE, --scale SCALE SCALE
                        Standard deviation in grid cells for Gaussian smoother
```

9.3 veros-copy-setup

```
usage: veros-copy-setup.py [-h]
                           {global_4deg,wave_propagation,acc,global_1deg,eady,north_
↪atlantic}
                           [target_dir]

Copies a Veros setup to another directory

positional arguments:
  {global_4deg,wave_propagation,acc,global_1deg,eady,north_atlantic}
                        Set-up to copy
  target_dir            Target directory (defaults to current directory)

optional arguments:
  -h, --help            show this help message and exit
```

9.4 veros-resubmit

```
usage: veros-resubmit.py [-h] [--callback CMD]
                         IDENTIFIER N_RUNS LENGTH_PER_RUN VEROS_CMD

Performs several runs of Veros back to back, using the previous run as restart
input. Intended to be used with scheduling systems (e.g. SLURM or PBS).

positional arguments:
  IDENTIFIER      base identifier of the simulation
  N_RUNS          total number of runs
  LENGTH_PER_RUN  length (in seconds) of each run
  VEROS_CMD       the command that is used to call veros (in quotes)

optional arguments:
  -h, --help      show this help message and exit
  --callback CMD  command to call after each run has finished (defaults to
                  calling itself)
```

CHAPTER 10

The Veros interface

10.1 Veros main class

10.2 veros_method decorator

10.3 Tools & utilities

CHAPTER 11

Frequently asked questions

11.1 Which backend should I choose to run my model (NumPy / Bohrium)?

Because in its current state Bohrium carries some computational overhead, this mostly depends on your problem size and the architecture you want to use. As a rule of thumb, switching from NumPy to Bohrium is beneficial if your set-up contains at least 1,000,000 elements (total number of elements in a 3-dimensional array, i.e., $n_x n_y n_z$). You can also use *our benchmarks* for general orientation.

CHAPTER 12

Benchmarks

Note: The following benchmarks are for general orientation only. Benchmark results are highly platform dependent; your mileage may vary.

The following figure presents some benchmarks that compare the performance of Veros and pyOM 2.1 depending on the problem size:

Fig. 12.1: Benchmarks on a Desktop PC with 4 CPU cores (I) and a cluster node with 24 CPU cores and an NVidia Tesla P100 GPU (II). Line fits suggest a linear scaling with constant overhead for all components.

CHAPTER 13

Contact

If you want to report a bug in Veros, have a technical inquiry, or want to ask for a missing feature, please use our issue tracker on GitHub.

In case you have general questions about Veros, please contact the maintainer of the Veros repository.

Bibliography

[EdenGreatbatch2008] Eden, Carsten, and Richard J. Greatbatch. "Towards a mesoscale eddy closure." Ocean Modelling 20.3 (2008): 223-239.

[OlbersEden2013] Olbers, Dirk, and Carsten Eden. "A global model for the diapycnal diffusivity induced by internal gravity waves." Journal of Physical Oceanography 43.8 (2013): 1759-1779.

[Gent1995] Gent, Peter R., et al. "Parameterizing eddy-induced tracer transports in ocean circulation models." Journal of Physical Oceanography 25.4 (1995): 463-474.

[Griffies1998] Griffies, Stephen M. "The Gent–McWilliams skew flux." Journal of Physical Oceanography 28.5 (1998): 831-841.

[Vallis2006] Vallis, Geoffrey K. Atmospheric and oceanic fluid dynamics: fundamentals and large-scale circulation. Cambridge University Press, 2006.

[Gaspar1990] Gaspar, Philippe, Yves Grégoris, and JeanMichel Lefevre. "A simple eddy kinetic energy model for simulations of the oceanic vertical mixing: Tests at station Papa and LongTerm Upper Ocean Study site." Journal of Geophysical Research: Oceans 95.C9 (1990): 16179-16193.

Index