

---

# **Veros Documentation**

*Release 0.1.1+102.gec572ba.dirty*

**The Veros Team**

**Feb 14, 2019**



<b>1</b>	<b>A short introduction to Veros</b>	<b>3</b>
1.1	The vision . . . . .	3
1.2	Features . . . . .	3
<b>2</b>	<b>Getting started</b>	<b>7</b>
2.1	Installation . . . . .	7
2.2	Setting up a model . . . . .	8
2.3	Running Veros . . . . .	9
2.4	Enhancing Veros . . . . .	10
<b>3</b>	<b>Creating an advanced model setup</b>	<b>13</b>
3.1	The vision . . . . .	14
3.2	Model skeleton . . . . .	14
3.3	Step 1: Setup grid . . . . .	19
3.4	Step 2: Create idealized topography . . . . .	20
3.5	Step 3: Interpolate forcings & initial conditions . . . . .	21
3.6	Step 4: Set up diagnostics & final touches . . . . .	21
<b>4</b>	<b>Running Veros on a cluster</b>	<b>23</b>
4.1	Installation . . . . .	23
4.2	Usage . . . . .	23
<b>5</b>	<b>Setup gallery</b>	<b>25</b>
5.1	Idealized configurations . . . . .	27
5.2	Realistic configurations . . . . .	28
<b>6</b>	<b>Available settings</b>	<b>29</b>
<b>7</b>	<b>Model variables</b>	<b>35</b>
7.1	Variable class . . . . .	35
7.2	Available variables . . . . .	35
<b>8</b>	<b>Diagnostics</b>	<b>63</b>
8.1	Base class . . . . .	63
8.2	Available diagnostics . . . . .	64
<b>9</b>	<b>Command line tools</b>	<b>67</b>
9.1	veros . . . . .	67

9.2	veros-create-mask	67
9.3	veros-copy-setup	67
9.4	veros-resubmit	67
<b>10</b>	<b>The Veros interface</b>	<b>69</b>
10.1	Veros main class	69
10.2	veros_method decorator	72
10.3	Tools & utilities	72
<b>11</b>	<b>Frequently asked questions</b>	<b>73</b>
11.1	Which backend should I choose to run my model (NumPy / Bohrium)?	73
<b>12</b>	<b>Benchmarks</b>	<b>75</b>
12.1	Veros compared to PyOM2	75
12.2	Veros runtime estimates on the DC3 cluster	75
<b>13</b>	<b>Publications</b>	<b>77</b>
13.1	Papers	77
13.2	Talks	77
<b>14</b>	<b>Contact</b>	<b>79</b>
	<b>Bibliography</b>	<b>81</b>



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!



### 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 believe 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  $\zeta$  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

### Isonutral 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 responsible 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:**

- Python 3.x is not yet fully supported due to some issues with Bohrium.

### 1.2.6 References



## 2.1 Installation

### 2.1.1 Using pip (multi-platform)

---

**Note:** You should only install Veros via pip if you want to get going as quickly as possible, and do not plan to access or modify the Veros source code. The preferred way to install Veros is through Anaconda (see below).

---

If you already have Python installed, the quickest way to get a working Veros installation is to run:

```
$ pip install veros --user
```

and optionally:

```
$ pip install bohrium --user
```

to use Veros with Bohrium (Linux and OSX only).

### 2.1.2 Using Anaconda (multi-platform, recommended)

1. Download and install Anaconda. Make sure to grab the 64-bit version of the Python interpreter.
2. Clone the Veros repository:

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

2. Create a new conda environment for Veros, and install all relevant dependencies, by running

```
$ conda env create -f environment-unix.yml
```

on Linux and OSX, or

```
$ conda env create -f environment-windows.yml
```

on Windows.

3. To use Veros, just activate your new conda environment! This can be done through either **conda activate veros**, **source activate veros**, or **activate veros**, depending on your platform and Anaconda installation.

### 2.1.3 On bare metal (Ubuntu / Debian)

1. Install some dependencies:

```
$ sudo apt-get install git python3-dev python3-pip libhdf5-dev libnetcdf-dev
```

2. Clone our repository:

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

3. Install Veros (preferably in a virtual environment) via:

```
$ pip3 install -e ./veros
```

4. Optionally, install Bohrium via:

```
$ pip3 install bohrium
```

## 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 general 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 implemented 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 achieve 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:

```
@veros.tools.cli
def run(*args, **kwargs):
    simulation = MyVerosSetup()
    simulation.setup()
    simulation.run()

if __name__ == "__main__":
    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 decorated with `veros.tools.cli()` 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:

```
$ pytest -v . --pyom2-lib /path/to/pyOM2/py_src/pyOM_code.so
```

from the main folder of the Veros repository.

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 YAML 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 explicit 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](#)).



---

## 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.

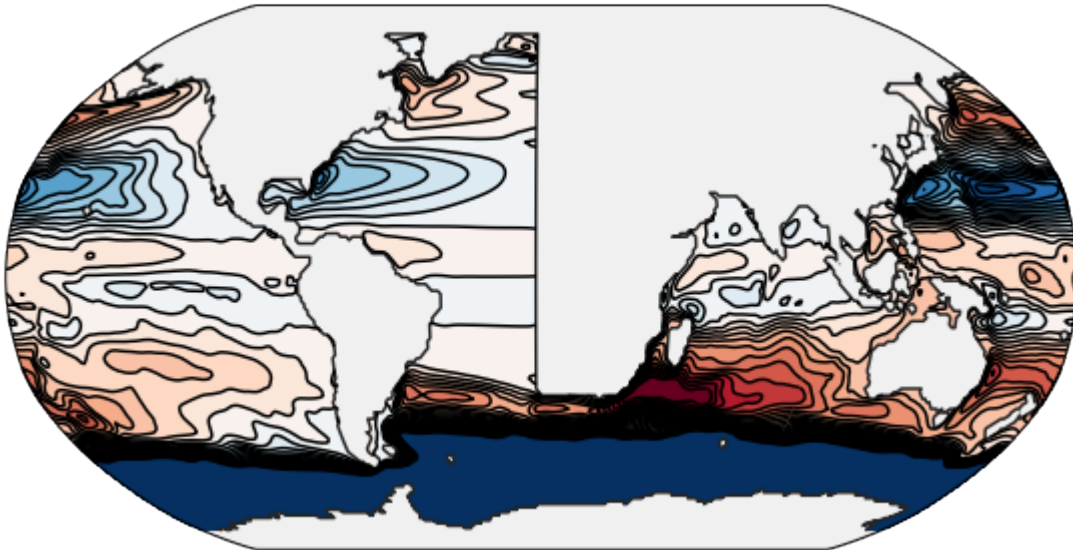


Fig. 3.1: The resulting stream function after about 1 year of integration.

## 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

import veros
import veros.tools

BASE_PATH = os.path.dirname(os.path.realpath(__file__))
DATA_FILES = veros.tools.get_assets("global_1deg", os.path.join(BASE_PATH, "assets.yml"
↪))

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

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

    @veros.veros_method
    def set_parameter(self):
        """
        set main parameters
        """
        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.veros_method
def _read_forcing(self, var):
    with Dataset(DATA_FILES["forcing"], "r") as infile:
        return np.array(infile.variables[var][...]).T

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

    @veros.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.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
        efoldl_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]

qnetc_data = self._read_forcing("dqdt")
self.qnetc[2:-2, 2:-2, :] = qnetc_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.veros_method
def set_forcing(self):
    t_rest = 30. * 86400.
    cp_0 = 3991.86795711963 # J/kg /K

    year_in_seconds = veros.time.convert_time(self, 1., "years", "seconds")
    (n1, f1), (n2, f2) = veros.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
↪n2] 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.qq nec = f1 * self.q nec [..., n1] + f2 * self.q nec [..., n2]
        self.qq net = f1 * self.q net [..., n1] + f2 * self.q net [..., n2]
        self.forc_temp_surface[...] = (self.qq net + self.qq nec * (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.veros_method

```

```

def set_diagnostics(self):
    average_vars = ["surface_tau", "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", "mxi", "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.

def after_timestep(self):
    pass

@veros.tools.cli
def run(*args, **kwargs):
    simulation = GlobalOneDegree(*args, **kwargs)
    simulation.setup()
    simulation.run()

if __name__ == "__main__":
    run()

```

The biggest changes in the new wave propagation setup will be located in the `set_grid()` `set_topography()` and `set_initial_conditions()` methods to accommodate 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:

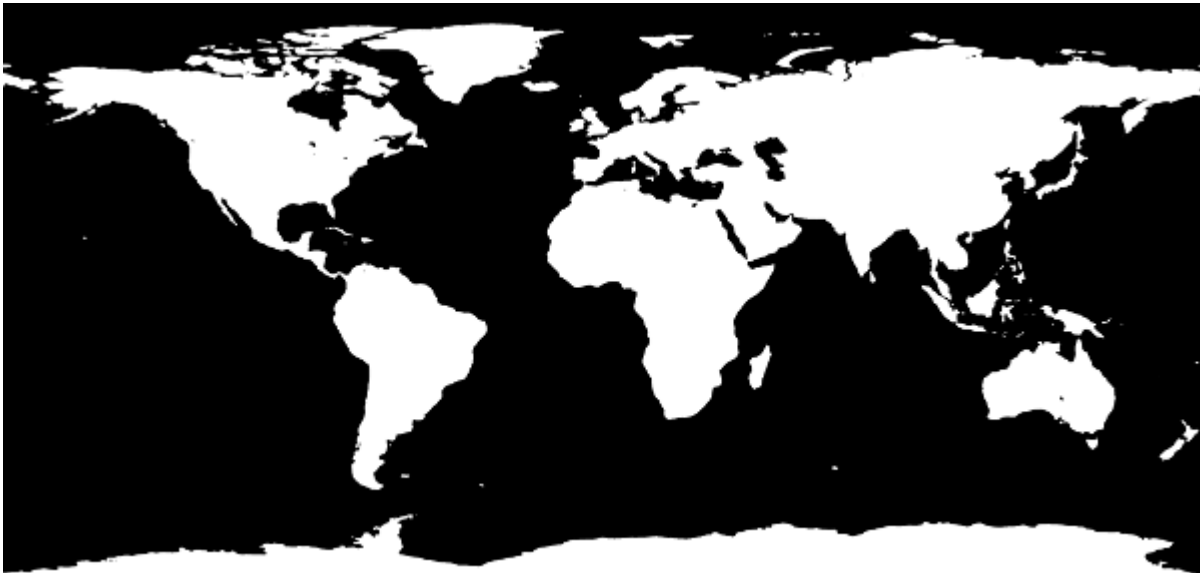


Fig. 3.2: Smoothed topography mask

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 available; 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:



Fig. 3.3: Idealized 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;
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

---

## 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

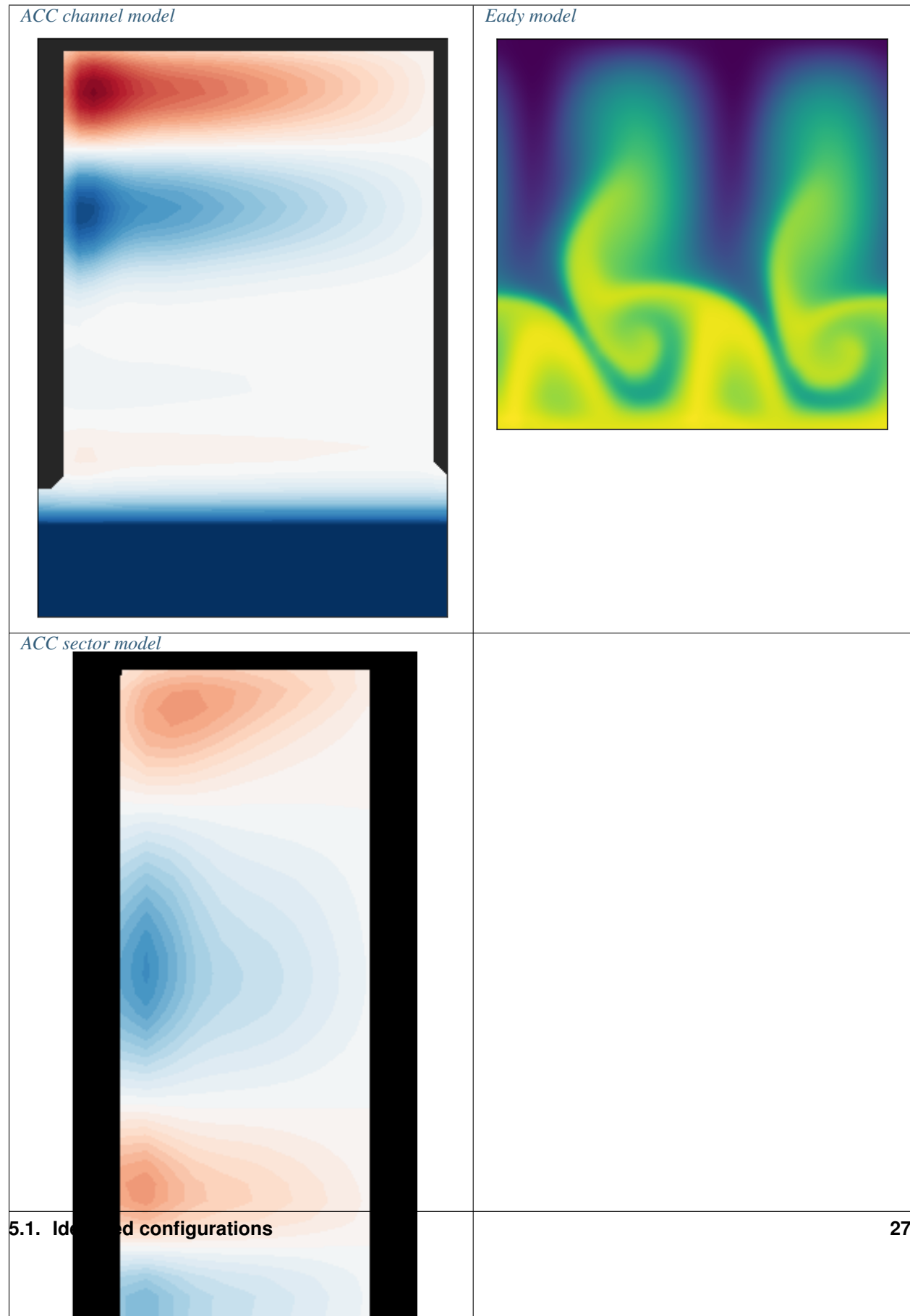
---

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

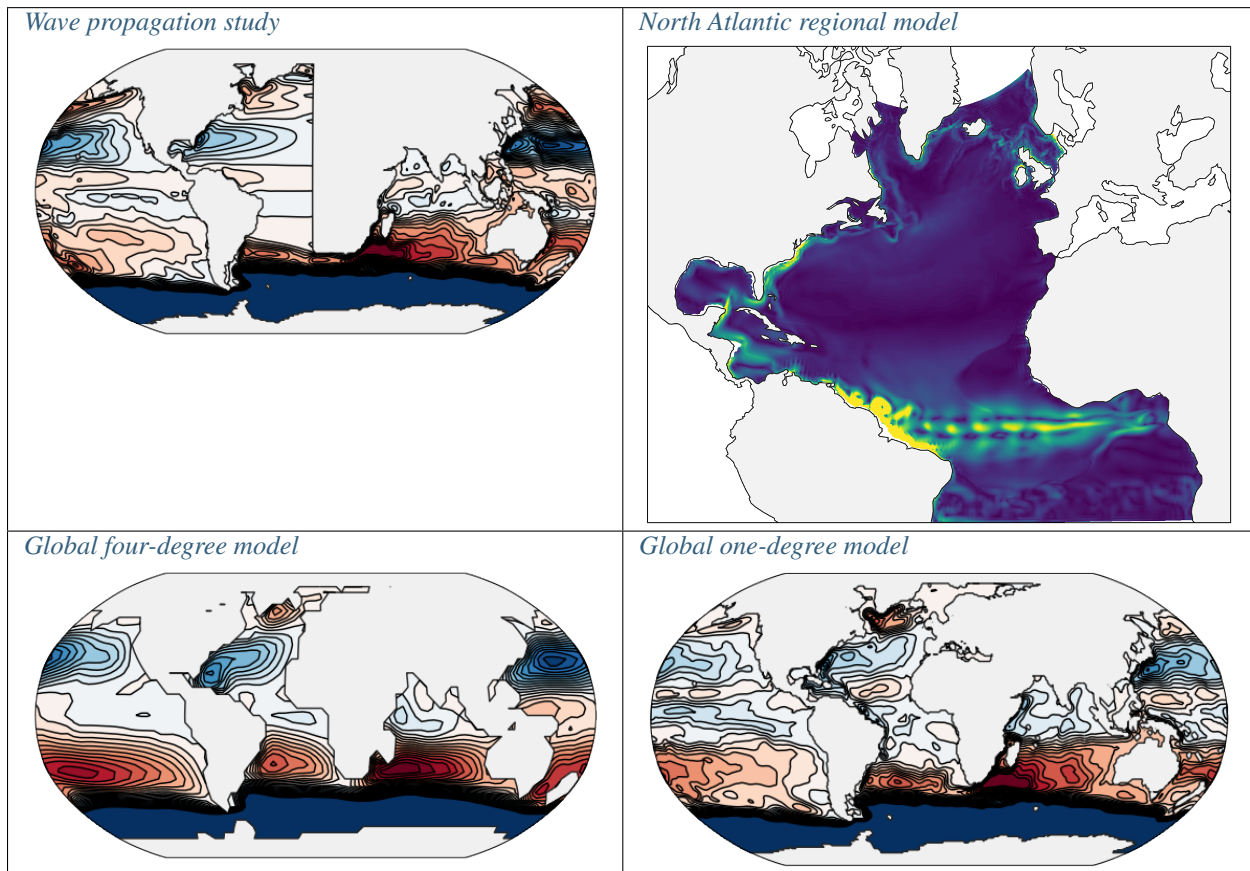


### 5.1.1 ACC channel model

### 5.1.2 Eady model

### 5.1.3 ACC sector model

## 5.2 Realistic configurations



### 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



---

## 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
```

**identifier = UNNAMED**

Identifier of the current simulation

**nx = 0**

Grid points in zonal (x) direction

**ny = 0**

Grid points in meridional (y,j) direction

**nz = 0**

Grid points in vertical (z,k) direction

**dt\_mom = 0.0**

Time step in seconds for momentum

**dt\_tracer = 0.0**

Time step for tracers, can be larger than dt\_mom

**dt\_tke = 0.0**

Time step for TKE module, currently set to dt\_mom (unused)

**runlen = 0.0**

Length of simulation in seconds

**AB\_eps = 0.1**

Deviation from Adam-Bashforth weighting

**coord\_degree = False**

either spherical (True) or cartesian (False) coordinates

**enable\_cyclic\_x = False**  
enable cyclic boundary conditions

**eq\_of\_state\_type = 1**  
equation of state: 1: linear, 3: nonlinear with comp., 5: TEOS

**enable\_implicit\_vert\_friction = False**  
enable implicit vertical friction

**enable\_explicit\_vert\_friction = False**  
enable explicit vertical friction

**enable\_hor\_friction = False**  
enable horizontal friction

**enable\_hor\_diffusion = False**  
enable horizontal diffusion

**enable\_biharmonic\_friction = False**  
enable biharmonic horizontal friction

**enable\_biharmonic\_mixing = False**  
enable biharmonic horizontal mixing

**enable\_hor\_friction\_cos\_scaling = False**  
scaling of hor. viscosity with  $\cos(\text{latitude})^{**}\text{cosPower}$

**enable\_ray\_friction = False**  
enable Rayleigh damping

**enable\_bottom\_friction = False**  
enable bottom friction

**enable\_bottom\_friction\_var = False**  
enable bottom friction with lateral variations

**enable\_quadratic\_bottom\_friction = False**  
enable quadratic bottom friction

**enable\_tempsalt\_sources = False**  
enable restoring zones, etc

**enable\_momentum\_sources = False**  
enable restoring zones, etc

**enable\_superbee\_advection = False**  
enable advection scheme with implicit mixing

**enable\_conserve\_energy = True**  
exchange energy consistently

**enable\_store\_bottom\_friction\_tke = False**  
transfer dissipated energy by bottom/rayleig fric. to TKE, else transfer to internal waves

**enable\_store\_cabbeling\_heat = False**  
transfer non-linear mixing terms to potential enthalpy, else transfer to TKE and EKE

**enable\_noslip\_lateral = False**  
enable lateral no-slip boundary conditions in harmonic- and biharmonic friction.

**congr\_epsilon = 1e-12**  
convergence criteria for Poisson solver

**congr\_max\_iterations = 1000**  
maximum number of Poisson solver iterations

**A\_h = 0.0**  
lateral viscosity in  $m^2/s$

**K\_h = 0.0**  
lateral diffusivity in  $m^2/s$

**r\_ray = 0.0**  
Rayleigh damping coefficient in  $1/s$

**r\_bot = 0.0**  
bottom friction coefficient in  $1/s$

**r\_quad\_bot = 0.0**  
quadratic bottom friction coefficient

**hor\_friction\_cosPower = 3**

**A\_hbi = 0.0**  
lateral biharmonic viscosity in  $m^4/s$

**K\_hbi = 0.0**  
lateral biharmonic diffusivity in  $m^4/s$

**kappaH\_0 = 0.0**

**kappaM\_0 = 0.0**  
fixed values for vertical viscosity/diffusivity which are set for no TKE model

**enable\_neutral\_diffusion = False**  
enable isopycnal mixing

**enable\_skew\_diffusion = False**  
enable skew diffusion approach for eddy-driven velocities

**enable\_TEM\_friction = False**  
TEM approach for eddy-driven velocities

**K\_iso\_0 = 0.0**  
constant for isopycnal diffusivity in  $m^2/s$

**K\_iso\_steep = 0.0**  
lateral diffusivity for steep slopes in  $m^2/s$

**K\_gm\_0 = 0.0**  
fixed value for  $K_{gm}$  which is set for no EKE model

**iso\_dslope = 0.0008**  
parameters controlling max allowed isopycnal slopes

**iso\_slopec = 0.001**  
parameters controlling max allowed isopycnal slopes

**enable\_idemix = False**

**tau\_v = 86400.0**  
time scale for vertical symmetrisation

**tau\_h = 1296000.0**  
time scale for horizontal symmetrisation

**gamma = 1.57**

```
jstar = 10.0
    spectral bandwidth in modes
mu0 = 1.333333333333
    dissipation parameter
enable_idemix_hor_diffusion = False
enable_eke_diss_bottom = False
enable_eke_diss_surfbot = False
eke_diss_surfbot_frac = 1.0
    fraction which goes into bottom
enable_idemix_superbee_advection = False
enable_idemix_upwind_advection = False
enable_tke = False
c_k = 0.1
c_eps = 0.7
alpha_tke = 1.0
mxl_min = 1e-12
kappaM_min = 0.0
kappaM_max = 100.0
tke_mxl_choice = 1
enable_tke_superbee_advection = False
enable_tke_upwind_advection = False
enable_tke_hor_diffusion = False
K_h_tke = 2000.0
    lateral diffusivity for tke
enable_eke = False
eke_lmin = 100.0
    minimal length scale in m
eke_c_k = 1.0
eke_cross = 1.0
    Parameter for EKE model
eke_crhin = 1.0
    Parameter for EKE model
eke_c_eps = 1.0
    Parameter for EKE model
eke_k_max = 10000.0
    maximum of K_gm
alpha_eke = 1.0
    factor vertical friction
enable_eke_superbee_advection = False
```

**enable\_eke\_upwind\_advection = False**

**enable\_eke\_isopycnal\_diffusion = False**  
 use `K_gm` also for isopycnal diffusivity

**enable\_eke\_leewave\_dissipation = False**

**c\_lee0 = 1.0**

**eke\_Ri0 = 200.0**

**eke\_Ri1 = 50.0**

**eke\_int\_diss0 = 5.78703703704e-07**

**kappa\_EKE0 = 0.1**

**eke\_r\_bot = 0.0**  
 bottom friction coefficient

**eke\_hrms\_k0\_min = 0.0**  
 min value for bottom roughness parameter

**kappaH\_min = 0.0**  
 minimum value for vertical diffusivity

**enable\_Prandtl\_tke = True**  
 Compute Prandtl number from stratification levels in TKE routine

**Prandtl\_tke0 = 10.0**  
 Constant Prandtl number when stratification is neglected for `kappaH` computation in TKE routine

**verbose\_island\_routines = False**  
 Print extra debugging output in island / boundary integral routines

**use\_io\_threads = True**  
 Start extra threads for disk writes

**io\_timeout = 20**  
 Timeout in seconds while waiting for IO locks to be released

**enable\_netcdf\_zlib\_compression = True**  
 Use netCDF4's native zlib interface, which leads to smaller output files (but carries some computational overhead).

**enable\_hdf5\_gzip\_compression = True**  
 Use h5py's native gzip interface, which leads to smaller restart files (but carries some computational overhead).

**restart\_input\_filename =**  
 File name of restart input. If not given, no restart data will be read.

**restart\_output\_filename = {identifier}\_{itt:0>4d}.restart.h5**  
 File name of restart output. May contain Python format syntax that is substituted with Veros attributes.

**restart\_frequency = 0**  
 Frequency (in seconds) to write restart data

**force\_overwrite = False**  
 Overwrite existing output files

**pyom\_compatibility\_mode = False**  
 Force compatibility to pyOM2 (even reproducing bugs and other quirks). For testing purposes only.

**diskless\_mode = False**  
 Suppress all output to disk. Mainly used for testing purposes.

**default\_float\_type = float64**

Default type to use for floating point arrays (e.g. float32 or float64).

**use\_amg\_preconditioner = True**

Use AMG preconditioner in Poisson solver if pyamg is installed.

---

## 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 attributes 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

```
class veros.variables.Variable (name, dims, units, long_description, dtype=None, output=False,
time_dependent=True, scale=1.0, write_to_restart=False, extra_attributes=None)
```

### 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

## 7.2.1 Main variables

Veros.**dxt**

**Units** m

**Dimensions** xt

**Type** float

**Attributes**

Zonal (x) spacing of T-grid point

Veros.**dxu**

**Units** m

**Dimensions** xu

**Type** float

**Attributes**

Zonal (x) spacing of U-grid point

Veros.**dyt**

**Units** m

**Dimensions** yt

**Type** float

**Attributes**

Meridional (y) spacing of T-grid point

Veros.**dyu**

**Units** m

**Dimensions** yu

**Type** float

**Attributes**

Meridional (y) spacing of U-grid point

Veros.**zt**

**Units** m

**Dimensions** zt

**Type** float

**Attributes**

Vertical coordinate

Veros.**zw**

**Units** m

**Dimensions** zw

**Type** float

**Attributes**



Vertical coordinate

Veros.**dzt**

**Units** m

**Dimensions** zt

**Type** float

**Attributes**

Vertical spacing

Veros.**dzw**

**Units** m

**Dimensions** zw

**Type** float

**Attributes**

Vertical spacing

Veros.**cost**

**Units** 1

**Dimensions** yt

**Type** float

**Attributes**

Metric factor for spherical coordinates

Veros.**cosu**

**Units** 1

**Dimensions** yu

**Type** float

**Attributes**

Metric factor for spherical coordinates

Veros.**tantr**

**Units** 1

**Dimensions** yt

**Type** float

**Attributes**

Metric factor for spherical coordinates

Veros.**coriolis\_t**

**Units** 1/s

**Dimensions** xt, yt

**Type** float

**Attributes**

Coriolis frequency at T grid point

Veros.**coriolis\_h**

**Units** 1/s

**Dimensions** xt, yt

**Type** float

**Attributes**

Horizontal Coriolis frequency at T grid point

Veros.**kbot**

**Units**

**Dimensions** xt, yt

**Type** int

**Attributes**

Index of the deepest grid cell (counting from 1, 0 means all land)

Veros.**ht**

**Units** m

**Dimensions** xt, yt

**Type** float

**Attributes**

Total depth of the water column

Veros.**hu**

**Units** m

**Dimensions** xu, yt

**Type** float

**Attributes**

Total depth of the water column

Veros.**hv**

**Units** m

**Dimensions** xt, yu

**Type** float

**Attributes**

Total depth of the water column

Veros.**hur**

**Units** m

**Dimensions** xu, yt

**Type** float

**Attributes**

Total depth of the water column (masked)

Veros.**hvr**

**Units** m

**Dimensions** xt, yu

**Type** float

**Attributes**

Total depth of the water column (masked)

Veros.**beta**

**Units** 1/(ms)

**Dimensions** xt, yt

**Type** float

**Attributes**

Change of Coriolis frequency with latitude

Veros.**area\_t**

**Units** m<sup>2</sup>

**Dimensions** xt, yt

**Type** float

**Attributes**

Area of T-box

Veros.**area\_u**

**Units** m<sup>2</sup>

**Dimensions** xu, yt

**Type** float

**Attributes**

Area of U-box

Veros.**area\_v**

**Units** m<sup>2</sup>

**Dimensions** xt, yu

**Type** float

**Attributes**

Area of V-box

Veros.**maskT**

**Units**

**Dimensions** xt, yt, zt

**Type** int

**Attributes**

Mask in physical space for tracer points

Veros.**maskU**

**Units**

**Dimensions** xu, yt, zt

**Type** int

**Attributes**

Mask in physical space for U points

Veros.**maskV**

**Units**

**Dimensions** xt, yu, zt

**Type** int

**Attributes**

Mask in physical space for V points

Veros.**maskW**

**Units**

**Dimensions** xt, yt, zw

**Type** int

**Attributes**

Mask in physical space for W points

Veros.**maskZ**

**Units**

**Dimensions** xu, yu, zt

**Type** int

**Attributes**

Mask in physical space for Zeta points

Veros.**rho**

**Units** kg/m<sup>3</sup>

**Dimensions** xt, yt, zt, timesteps

**Type** float

**Attributes**

In-situ density anomaly, relative to the surface mean value of 1024 kg/m<sup>3</sup>

Veros.**prho**

**Units** kg/m<sup>3</sup>

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Potential density anomaly, relative to the surface mean value of 1024 kg/m<sup>3</sup> (equal to in-situ density anomaly for equation of state 1 to 4)

Veros.**int\_drhodT**

**Units** ?

**Dimensions** xt, yt, zt, timesteps

**Type** float

**Attributes**

Partial derivative of dynamic enthalpy by temperature

Veros.**int\_drhodS**

**Units** ?

**Dimensions** xt, yt, zt, timesteps

**Type** float

**Attributes**

Partial derivative of dynamic enthalpy by salinity

Veros.**Nsqf**

**Units** 1/s<sup>2</sup>

**Dimensions** xt, yt, zw, timesteps

**Type** float

**Attributes**

Square of stability frequency

Veros.**Hd**

**Units** m<sup>2</sup>/s<sup>2</sup>

**Dimensions** xt, yt, zt, timesteps

**Type** float

**Attributes**

Dynamic enthalpy

Veros.**dHd**

**Units** m<sup>2</sup>/s<sup>3</sup>

**Dimensions** xt, yt, zt, timesteps

**Type** float

**Attributes**

Change of dynamic enthalpy due to advection

Veros.**temp**

**Units** deg C

**Dimensions** xt, yt, zt, timesteps

**Type** float

**Attributes**

Conservative temperature

Veros.**dttemp**

**Units** deg C/s

**Dimensions** xt, yt, zt, timesteps

**Type** float

**Attributes**

Conservative temperature tendency

Veros.**salt**

**Units** g/kg

**Dimensions** xt, yt, zt, timesteps

**Type** float

**Attributes**

Salinity

Veros.**dsalt**

**Units** g/(kg s)

**Dimensions** xt, yt, zt, timesteps

**Type** float

**Attributes**

Salinity tendency

Veros.**dttemp\_vmix**

**Units** deg C/s

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Change of temperature due to vertical mixing

Veros.**dttemp\_hmix**

**Units** deg C/s

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Change of temperature due to horizontal mixing

Veros.**dsalt\_vmix**

**Units** deg C/s

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Change of salinity due to vertical mixing

Veros.**dsalt\_hmix**

**Units** deg C/s

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Change of salinity due to horizontal mixing

Veros.**dtemp\_iso**

**Units** deg C/s

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Change of temperature due to isopycnal mixing plus skew mixing

Veros.**dsalt\_iso**

**Units** deg C/s

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Change of salinity due to isopycnal mixing plus skew mixing

Veros.**forc\_temp\_surface**

**Units** m K/s

**Dimensions** xt, yt

**Type** float

**Attributes**

Surface temperature flux

Veros.**forc\_salt\_surface**

**Units** m g/s kg

**Dimensions** xt, yt

**Type** float

**Attributes**

Surface salinity flux

Veros.**flux\_east**

**Units** ?

**Dimensions** xu, yt, zt

**Type** float

**Attributes**

Multi-purpose flux

Veros.**flux\_north**

**Units** ?

**Dimensions** xt, yu, zt

**Type** float

**Attributes**

Multi-purpose flux

Veros.**flux\_top**

**Units** ?

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Multi-purpose flux

Veros.**u**

**Units** m/s

**Dimensions** xu, yt, zt, timesteps

**Type** float

**Attributes**

Zonal velocity

Veros.**v**

**Units** m/s

**Dimensions** xt, yu, zt, timesteps

**Type** float

**Attributes**

Meridional velocity

Veros.**w**

**Units** m/s

**Dimensions** xt, yt, zw, timesteps

**Type** float

**Attributes**

Vertical velocity

Veros.**du**

**Units** m/s

**Dimensions** xu, yt, zt, timesteps

**Type** float



**Attributes**

Zonal velocity tendency

Veros.**dv**

**Units** m/s

**Dimensions** xt, yu, zt, timesteps

**Type** float

**Attributes**

Meridional velocity tendency

Veros.**du\_cor**

**Units** m/s<sup>2</sup>

**Dimensions** xu, yt, zt

**Type** float

**Attributes**

Change of u due to Coriolis force

Veros.**dv\_cor**

**Units** m/s<sup>2</sup>

**Dimensions** xt, yu, zt

**Type** float

**Attributes**

Change of v due to Coriolis force

Veros.**du\_mix**

**Units** m/s<sup>2</sup>

**Dimensions** xu, yt, zt

**Type** float

**Attributes**

Change of u due to implicit vertical mixing

Veros.**dv\_mix**

**Units** m/s<sup>2</sup>

**Dimensions** xt, yu, zt

**Type** float

**Attributes**

Change of v due to implicit vertical mixing

Veros.**du\_adv**

**Units** m/s<sup>2</sup>

**Dimensions** xu, yt, zt

**Type** float

**Attributes**

Change of u due to advection

Veros.**dv\_adv**

**Units** m/s<sup>2</sup>

**Dimensions** xt, yu, zt

**Type** float

**Attributes**

Change of v due to advection

Veros.**p\_hydro**

**Units** m<sup>2</sup>/s<sup>2</sup>

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Hydrostatic pressure

Veros.**kappaM**

**Units** m<sup>2</sup>/s

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Vertical viscosity

Veros.**kappaH**

**Units** m<sup>2</sup>/s

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Vertical diffusivity

Veros.**surface\_taux**

**Units** m<sup>2</sup>/s<sup>2</sup>

**Dimensions** xu, yt

**Type** float

**Attributes**

Zonal surface wind stress

Veros.**surface\_tauy**

**Units** m<sup>2</sup>/s<sup>2</sup>

**Dimensions** xt, yu

**Type** float

**Attributes**

Meridional surface wind stress

Veros.**forc\_rho\_surface**

**Units** ?

**Dimensions** xt, yt

**Type** float

**Attributes**

Surface potential density flux

Veros.**psi**

**Units** m<sup>3</sup>/s

**Dimensions** xu, yu, timesteps

**Type** float

**Attributes**

Streamfunction

Veros.**dpsi**

**Units** m<sup>3</sup>/s<sup>2</sup>

**Dimensions** xu, yu, timesteps

**Type** float

**Attributes**

Streamfunction tendency

Veros.**isle**

**Units**

**Dimensions** isle

**Type** float

**Attributes**

Island number

Veros.**psin**

**Units** m<sup>3</sup>/s

**Dimensions** xu, yu, isle

**Type** float

**Attributes**

Boundary streamfunction

Veros.**dpsin**

**Units** ?

**Dimensions** isle, timesteps

**Type** float

**Attributes**

Boundary streamfunction factor

Veros.**line\_psin**

**Units** ?

**Dimensions** isle, isle

**Type** float

**Attributes**

Boundary line integrals

Veros.**boundary\_mask**

**Units**

**Dimensions** xt, yt, isle

**Type** float

**Attributes**

Boundary mask

Veros.**line\_dir\_south\_mask**

**Units**

**Dimensions** xt, yt, isle

**Type** float

**Attributes**

Line integral mask

Veros.**line\_dir\_north\_mask**

**Units**

**Dimensions** xt, yt, isle

**Type** float

**Attributes**

Line integral mask

Veros.**line\_dir\_east\_mask**

**Units**

**Dimensions** xt, yt, isle

**Type** float

**Attributes**

Line integral mask

Veros.**line\_dir\_west\_mask**

**Units**

**Dimensions** xt, yt, isle

**Type** float

**Attributes**

Line integral mask

Veros.**K\_gm**

**Units** m<sup>2</sup>/s

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

GM diffusivity, either constant or from EKE model

Veros.**K\_iso**

**Units** m<sup>2</sup>/s

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Along-isopycnal diffusivity

Veros.**K\_diss\_v**

**Units** m<sup>2</sup>/s<sup>3</sup>

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Kinetic energy dissipation by vertical, rayleigh and bottom friction

Veros.**K\_diss\_bot**

**Units** m<sup>2</sup>/s<sup>3</sup>

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Mean energy dissipation by bottom and rayleigh friction

Veros.**K\_diss\_h**

**Units** m<sup>2</sup>/s<sup>3</sup>

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Kinetic energy dissipation by horizontal friction

Veros.**K\_diss\_gm**

**Units** m<sup>2</sup>/s<sup>3</sup>

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Mean energy dissipation by GM (TRM formalism only)

Veros.**P\_diss\_v**

**Units**  $m^2/s^3$

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Potential energy dissipation by vertical diffusion

Veros.**P\_diss\_nonlin**

**Units**  $m^2/s^3$

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Potential energy dissipation by nonlinear equation of state

Veros.**P\_diss\_iso**

**Units**  $m^2/s^3$

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Potential energy dissipation by isopycnal mixing

Veros.**P\_diss\_skew**

**Units**  $m^2/s^3$

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Potential energy dissipation by GM (w/o TRM)

Veros.**P\_diss\_hmix**

**Units**  $m^2/s^3$

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Potential energy dissipation by horizontal mixing

Veros.**P\_diss\_adv**

**Units**  $m^2/s^3$

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Potential energy dissipation by advection

Veros.**P\_diss\_comp**

**Units** m<sup>2</sup>/s<sup>3</sup>

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Potential energy dissipation by compression

Veros.**P\_diss\_sources**

**Units** m<sup>2</sup>/s<sup>3</sup>

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Potential energy dissipation by external sources (e.g. restoring zones)

Veros.**u\_wgrid**

**Units** m/s

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Zonal velocity interpolated to W grid points

Veros.**v\_wgrid**

**Units** m/s

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Meridional velocity interpolated to W grid points

Veros.**w\_wgrid**

**Units** m/s

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Vertical velocity interpolated to W grid points

## 7.2.2 Conditional variables

### coord\_degree

Veros.**.xt**

**Units** degrees\_east

**Dimensions** xt

**Type** float

**Attributes**

Zonal (x) coordinate of T-grid point

Veros.**.xu**

**Units** degrees\_east

**Dimensions** xu

**Type** float

**Attributes**

Zonal (x) coordinate of U-grid point

Veros.**.yt**

**Units** degrees\_north

**Dimensions** yt

**Type** float

**Attributes**

Meridional (y) coordinate of T-grid point

Veros.**.yu**

**Units** degrees\_north

**Dimensions** yu

**Type** float

**Attributes**

Meridional (y) coordinate of U-grid point

### not coord\_degree

Veros.**.xt**

**Units** km

**Dimensions** xt

**Type** float

**Attributes**

Zonal (x) coordinate of T-grid point

Veros.**.xu**



**Units** km

**Dimensions** xu

**Type** float

**Attributes**

Zonal (x) coordinate of U-grid point

Veros.**yt**

**Units** km

**Dimensions** yt

**Type** float

**Attributes**

Meridional (y) coordinate of T-grid point

Veros.**yu**

**Units** km

**Dimensions** yu

**Type** float

**Attributes**

Meridional (y) coordinate of U-grid point

### **enable\_tempsalt\_sources**

Veros.**temp\_source**

**Units** K/s

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Non-conservative source of temperature

Veros.**salt\_source**

**Units** g/(kg s)

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Non-conservative source of salt

### **enable\_momentum\_sources**

Veros.**u\_source**

**Units** m/s<sup>2</sup> (?)

**Dimensions** xu, yt, zt

**Type** float

**Attributes**

Non-conservative source of zonal velocity

Veros.**v\_source**

**Units** m/s<sup>2</sup> (?)

**Dimensions** xt, yu, zt

**Type** float

**Attributes**

Non-conservative source of meridional velocity

### **enable\_neutral\_diffusion**

Veros.**K\_11**

**Units** ?

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Isopycnal mixing tensor component

Veros.**K\_13**

**Units** ?

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Isopycnal mixing tensor component

Veros.**K\_22**

**Units** ?

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Isopycnal mixing tensor component

Veros.**K\_23**

**Units** ?

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Isopycnal mixing tensor component

Veros.**K\_31**

**Units** ?

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Isopycnal mixing tensor component

Veros.**K\_32**

**Units** ?

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Isopycnal mixing tensor component

Veros.**K\_33**

**Units** ?

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Isopycnal mixing tensor component

Veros.**Ai\_ez**

**Units** ?

**Dimensions** xt, yt, zt, tensor1, tensor2

**Type** float

**Attributes**

?

Veros.**Ai\_nz**

**Units** ?

**Dimensions** xt, yt, zt, tensor1, tensor2

**Type** float

**Attributes**

?

Veros.**Ai\_bx**

**Units** ?

**Dimensions** xt, yt, zt, tensor1, tensor2

**Type** float

**Attributes**

?

Veros.**Ai\_by**

**Units** ?

**Dimensions** xt, yt, zt, tensor1, tensor2

**Type** float

**Attributes**

?

### **enable\_skew\_diffusion**

Veros.**B1\_gm**

**Units** m<sup>2</sup>/s

**Dimensions** xt, yu, zt

**Type** float

**Attributes**

Zonal component of GM streamfunction

Veros.**B2\_gm**

**Units** m<sup>2</sup>/s

**Dimensions** xu, yt, zt

**Type** float

**Attributes**

Meridional component of GM streamfunction

### **enable\_bottom\_friction\_var**

Veros.**r\_bot\_var\_u**

**Units** ?

**Dimensions** xu, yt

**Type** float

**Attributes**

Zonal bottom friction coefficient

Veros.**r\_bot\_var\_v**

**Units** ?

**Dimensions** xt, yu

**Type** float

**Attributes**

Meridional bottom friction coefficient

**enable\_TEM\_friction**Veros.**kappa\_gm****Units** m<sup>2</sup>/s**Dimensions** xt, yt, zw**Type** float**Attributes**

Vertical diffusivity

**enable\_tke**Veros.**tke****Units** m<sup>2</sup>/s<sup>2</sup>**Dimensions** xt, yt, zw, timesteps**Type** float**Attributes**

Turbulent kinetic energy

Veros.**sqrttke****Units** m/s**Dimensions** xt, yt, zw**Type** float**Attributes**

Square-root of TKE

Veros.**dtke****Units** m<sup>2</sup>/s<sup>3</sup>**Dimensions** xt, yt, zw, timesteps**Type** float**Attributes**

Turbulent kinetic energy tendency

Veros.**Prandtlnumber****Units****Dimensions** xt, yt, zw**Type** float**Attributes**

Prandtl number

Veros.**mx1****Units** m**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Mixing length

Veros.**forc\_tke\_surface**

**Units**  $m^3/s^3$

**Dimensions** xt, yt

**Type** float

**Attributes**

TKE surface flux

Veros.**tke\_diss**

**Units**  $m^2/s^3$

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

TKE dissipation

Veros.**tke\_surf\_corr**

**Units**  $m^3/s^3$

**Dimensions** xt, yt

**Type** float

**Attributes**

Correction of TKE surface flux

## **enable\_eke**

Veros.**eke**

**Units**  $m^2/s^2$

**Dimensions** xt, yt, zw, timesteps

**Type** float

**Attributes**

meso-scale energy

Veros.**deke**

**Units**  $m^2/s^3$

**Dimensions** xt, yt, zw, timesteps

**Type** float

**Attributes**

meso-scale energy tendency

Veros.**qrteke**

**Units** m/s

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

square-root of eke

Veros.**L\_rossby**

**Units** m

**Dimensions** xt, yt

**Type** float

**Attributes**

Rossby radius

Veros.**L\_rhines**

**Units** m

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Rhines scale

Veros.**eke\_len**

**Units** m

**Dimensions** xt, yt, zt

**Type** float

**Attributes**

Eddy length scale

Veros.**eke\_diss\_iw**

**Units** m<sup>2</sup>/s<sup>3</sup>

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Dissipation of EKE to internal waves

Veros.**eke\_diss\_tke**

**Units** m<sup>2</sup>/s<sup>3</sup>

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Dissipation of EKE to TKE

Veros.**eke\_bot\_flux**

**Units**  $m^3/s^3$

**Dimensions** xt, yt

**Type** float

**Attributes**

Flux by bottom friction

### **enable\_eke\_leewave\_dissipation**

Veros.**eke\_topo\_hrms**

**Units** ?

**Dimensions** xt, yt

**Type** float

**Attributes**

?

Veros.**eke\_topo\_lam**

**Units** ?

**Dimensions** xt, yt

**Type** float

**Attributes**

?

Veros.**hrms\_k0**

**Units** ?

**Dimensions** xt, yt

**Type** float

**Attributes**

?

Veros.**c\_lee**

**Units** 1/s

**Dimensions** xt, yt

**Type** float

**Attributes**

Lee wave dissipation coefficient

Veros.**eke\_lee\_flux**

**Units**  $m^3/s^3$

**Dimensions** xt, yt

**Type** float

**Attributes**



Lee wave flux

Veros.**c\_Ri\_diss**

**Units** 1/s

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Interior dissipation coefficient

## **enable\_idemix**

Veros.**E\_iw**

**Units** m<sup>2</sup>/s<sup>2</sup>

**Dimensions** xt, yt, zw, timesteps

**Type** float

**Attributes**

Internal wave energy

Veros.**dE\_iw**

**Units** m<sup>2</sup>/s<sup>2</sup>

**Dimensions** xt, yt, zw, timesteps

**Type** float

**Attributes**

Internal wave energy tendency

Veros.**c0**

**Units** m/s

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Vertical internal wave group velocity

Veros.**v0**

**Units** m/s

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Horizontal internal wave group velocity

Veros.**alpha\_c**

**Units** ?

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

?

Veros.**iw\_diss**

**Units**  $m^2/s^3$

**Dimensions** xt, yt, zw

**Type** float

**Attributes**

Internal wave dissipation

Veros.**forc\_iw\_surface**

**Units**  $m^3/s^3$

**Dimensions** xt, yt

**Type** float

**Attributes**

Internal wave surface forcing

Veros.**forc\_iw\_bottom**

**Units**  $m^3/s^3$

**Dimensions** xt, yt

**Type** float

**Attributes**

Internal wave bottom forcing

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.

**class** `veros.diagnostics.diagnostic.VerosDiagnostic` (*vs*)

Bases: `object`

Base class for diagnostics. Provides an interface and wrappers for common I/O.

Any diagnostic needs to implement the five interface methods and set some attributes.

**name** = `None`

Name that identifies the current diagnostic

**initialize** (*vs*)

Called at the end of setup. Use this to process user settings and handle setup.

**diagnose** (*vs*)

Called with frequency `sampling_frequency`.

**output** (*vs*)  
 Called with frequency `output_frequency`.

**write\_restart** (*vs*)  
 Responsible for writing restart files.

**read\_restart** (*vs*)  
 Responsible for reading restart files.

## 8.2 Available diagnostics

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

### 8.2.1 Snapshot

**class** `veros.diagnostics.snapshot.Snapshot` (*vs*)  
 Bases: `veros.diagnostics.diagnostic.VerosDiagnostic`

Writes snapshots of the current solution. Also reads and writes the main restart data required for restarting a Veros simulation.

**output\_path** = `'{identifier}.snapshot.nc'`  
 File to write to. May contain format strings that are replaced with Veros attributes.

**name** = `'snapshot'`

**output\_frequency** = `None`  
 Frequency (in seconds) in which output is written.

**output\_variables** = `None`  
 Variables to be written to output. Defaults to all Veros variables that have the attribute `output`.

**restart\_variables** = `None`  
 Variables to be written to restart. Defaults to all Veros variables that have the attribute `write_to_restart`.

### 8.2.2 Averages

**class** `veros.diagnostics.averages.Averages` (*vs*)  
 Bases: `veros.diagnostics.diagnostic.VerosDiagnostic`

Time average output diagnostic.

All registered variables are summed up when `diagnose()` is called, and averaged and output upon calling `output()`.

**name** = `'averages'`

**output\_path** = `'{identifier}.averages.nc'`  
 File to write to. May contain format strings that are replaced with Veros attributes.

**output\_variables** = `None`

**output\_frequency** = `None`  
 Frequency (in seconds) in which output is written.

**sampling\_frequency** = `None`  
 Frequency (in seconds) in which variables are accumulated.

### 8.2.3 CFL monitor

```
class veros.diagnostics.cfl_monitor.CFLMonitor(vs)
    Bases: veros.diagnostics.diagnostic.VerosDiagnostic

    Diagnostic monitoring the maximum CFL number of the solution to detect instabilities.

    Writes output to stdout (no binary output).

    name = 'cfl_monitor'
```

### 8.2.4 Tracer monitor

```
class veros.diagnostics.tracer_monitor.TracerMonitor(vs)
    Bases: veros.diagnostics.diagnostic.VerosDiagnostic

    Diagnostic monitoring global tracer contents / fluxes.

    Writes output to stdout (no binary output).

    name = 'tracer_monitor'

    output_frequency = None
        Frequency (in seconds) in which output is written.
```

### 8.2.5 Energy

```
class veros.diagnostics.energy.Energy(vs)
    Bases: veros.diagnostics.diagnostic.VerosDiagnostic

    Diagnose globally averaged energy cycle. Also averages energy in time.

    name = 'energy'

    output_path = '{identifier}.energy.nc'
        File to write to. May contain format strings that are replaced with Veros attributes.

    output_frequency = None
        Frequency (in seconds) in which output is written.

    sampling_frequency = None
        Frequency (in seconds) in which variables are accumulated.
```

### 8.2.6 Overturning

```
class veros.diagnostics.overturning.Overturning(vs)
    Bases: veros.diagnostics.diagnostic.VerosDiagnostic

    Isopycnal overturning diagnostic. Computes and writes vertical streamfunctions (zonally averaged).

    name = 'overturning'

    output_path = '{identifier}.overturning.nc'
        File to write to. May contain format strings that are replaced with Veros attributes.

    output_frequency = None
        Frequency (in seconds) in which output is written.

    sampling_frequency = None
        Frequency (in seconds) in which variables are accumulated.
```

`p_ref = 2000.0`  
Reference pressure for isopycnals

---

## 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.

```
python: can't open file '../bin/veros': [Errno 2] No such file or directory
```

### 9.2 veros-create-mask

```
python: can't open file '../bin/veros': [Errno 2] No such file or directory
```

### 9.3 veros-copy-setup

```
python: can't open file '../bin/veros': [Errno 2] No such file or directory
```

### 9.4 veros-resubmit

```
python: can't open file '../bin/veros': [Errno 2] No such file or directory
```





## 10.1 Veros main class

**class** `veros.Veros` (*backend='numpy', loglevel='info', logfile=None, profile=False, override=None*)  
Bases: `object`

Main class for Veros, used for building a model and running it.

---

**Note:** This class is meant to be subclassed. Subclasses need to implement the methods `set_parameter()`, `set_topography()`, `set_grid()`, `set_coriolis()`, `set_initial_conditions()`, `set_forcing()`, and `set_diagnostics()`.

---

### Parameters

- **backend** (`bool`, optional) – Backend to use for array operations. Possible values are `numpy` and `bohrium`. Defaults to `None`, which tries to read the backend from the command line (set via a flag `-b/--backend`), and uses `numpy` if no command line argument is given.
- **loglevel** (*one of {debug, info, warning, error, critical}, optional*) – Verbosity of the model. Tries to read value from command line if not given (`-v/--loglevel`). Defaults to `info`.
- **logfile** (*path, optional*) – Path to a log file to write output to. Tries to read value from command line if not given (`-l/--logfile`). Defaults to `stdout`.

### Example

```
>>> import matplotlib.pyplot as plt
>>> from climate.veros import Veros
>>>
>>> class MyModel(Veros):
```

```
>>>     ...
>>>
>>> simulation = MyModel(backend="bohrium")
>>> simulation.run()
>>> plt.imshow(simulation.psi[... , 0])
>>> plt.show()
```

#### **set\_parameter()**

To be implemented by subclass.

First function to be called during setup. Use this to modify the model settings.

#### **Example**

```
>>> def set_parameter(self):
>>>     self.nx, self.ny, self.nz = (360, 120, 50)
>>>     self.coord_degree = True
>>>     self.enable_cyclic = True
```

#### **set\_initial\_conditions()**

To be implemented by subclass.

May be used to set initial conditions.

#### **Example**

```
>>> @veros_method
>>> def set_initial_conditions(self):
>>>     self.u[:, :, :, self.tau] = np.random.rand(self.u.shape[:-1])
```

#### **set\_grid()**

To be implemented by subclass.

Has to set the grid spacings *dxt*, *dxt*, and *dzt*, along with the coordinates of the grid origin, *x\_origin* and *y\_origin*.

#### **Example**

```
>>> @veros_method
>>> def set_grid(self):
>>>     self.x_origin, self.y_origin = 0, 0
>>>     self.dxt[...] = [0.1, 0.05, 0.025, 0.025, 0.05, 0.1]
>>>     self.dyt[...] = 1.
>>>     self.dzt[...] = [10, 10, 20, 50, 100, 200]
```

#### **set\_coriolis()**

To be implemented by subclass.

Has to set the Coriolis parameter *coriolis\_t* at T grid cells.

### Example

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

#### `set_topography()`

To be implemented by subclass.

Must specify the model topography by setting *kbot*.

### Example

```
>>> @veros_method
>>> def set_topography(self):
>>>     self.kbot[:, :] = 10
>>>     # add a rectangular island somewhere inside the domain
>>>     self.kbot[10:20, 10:20] = 0
```

#### `set_forcing()`

To be implemented by subclass.

Called before every time step to update the external forcing, e.g. through *forc\_temp\_surface*, *forc\_salt\_surface*, *surface\_taux*, *surface\_tauy*, *forc\_tke\_surface*, *temp\_source*, or *salt\_source*. Use this method to implement time-dependent forcing.

### Example

```
>>> @veros_method
>>> def set_forcing(self):
>>>     current_month = (self.time / (31 * 24 * 60 * 60)) % 12
>>>     self.surface_taux[:, :] = self._windstress_data[:, :, current_month]
```

#### `set_diagnostics()`

To be implemented by subclass.

Called before setting up the *diagnostics*. Use this method e.g. to mark additional *variables* for output.

### Example

```
>>> @veros_method
>>> def set_diagnostics(self):
>>>     self.diagnostics["snapshot"].output_vars += ["drho", "dsalt", "dtemp"]
```

#### `after_timestep()`

Called at the end of each time step. Can be used to define custom, setup-specific events.

#### `flush()`

Flush computations if supported by the current backend.

#### `run()`

Main routine of the simulation.

## 10.2 veros\_method decorator

`veros.veros_method` (*function*)

Decorator that injects the current backend as variable `np` into the wrapped function.

---

**Note:** This decorator should be applied to all functions that make use of the computational backend (even when subclassing `climate.veros.Veros`). The first argument to the decorated function must be a `Veros` instance.

---

### Example

```
>>> from climate.veros import Veros, veros_method
>>>
>>> class MyModel(Veros):
>>>     @veros_method
>>>     def set_topography(self):
>>>         self.kbot[...] = np.random.randint(0, self.nz, size=self.kbot.shape)
```

`veros.veros_inline_method` (*function*)

## 10.3 Tools & utilities

### 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.



---

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

---

### 12.1 Veros compared to PyOM2

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.

### 12.2 Veros runtime estimates on the DC3 cluster

The following figure presents estimates of Veros runtime required for 1 year prediction on CPU and GPU nodes of the DC3 cluster. The estimates were done for ACC, Wave propagation and Global cases with different spatial and temporal resolution.

For more details on the cases configuration see *the setup gallery*.

Fig. 12.2: Runtime estimates on a cluster node with 32 CPU cores (Bh CPU) and on a cluster node with 24 CPU cores and an NVidia Tesla P100 GPU (Bh GPU).





### 13.1 Papers

- Häfner, D., Jacobsen, R. L., Eden, C., Kristensen, M. R. B., Jochum, M., Nuterman, R., and Vinter, B.: Veros v0.1 – a fast and versatile ocean simulator in pure Python, *Geosci. Model Dev.*, 11, 3299-3312, <https://doi.org/10.5194/gmd-11-3299-2018>, 2018.

### 13.2 Talks

- Veros — A High-Performance Ocean Simulator Written in Pure Python. A talk held at the 98th Annual Meeting of the American Meteorological Society (AMS). [Abstract](#) and [slides](#) available.
- A Fast Versatile Ocean Simulator (Veros) in Pure Python. A talk held at the European Geosciences Union General Assembly 2018, Vienna, Austria. [Abstract](#)



## CHAPTER 14

---

### 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 Jean-Michel Lefevre. "A simple eddy kinetic energy model for simulations of the oceanic vertical mixing: Tests at station Papa and Long-Term Upper Ocean Study site." *Journal of Geophysical Research: Oceans* 95.C9 (1990): 16179-16193.



**A**

after\_timestep() (veros.Veros method), 71  
Ai\_bx (Veros attribute), 55  
Ai\_by (Veros attribute), 55  
Ai\_ez (Veros attribute), 55  
Ai\_nz (Veros attribute), 55  
alpha\_c (Veros attribute), 61  
area\_t (Veros attribute), 39  
area\_u (Veros attribute), 39  
area\_v (Veros attribute), 39  
Averages (class in veros.diagnostics.averages), 64

**B**

B1\_gm (Veros attribute), 56  
B2\_gm (Veros attribute), 56  
beta (Veros attribute), 39  
BH\_STACK=cuda, 10  
BH\_STACK=opencl, 10  
boundary\_mask (Veros attribute), 48

**C**

c0 (Veros attribute), 61  
c\_lee (Veros attribute), 60  
c\_Ri\_diss (Veros attribute), 61  
CFLMonitor (class in veros.diagnostics.cfl\_monitor), 65  
coriolis\_h (Veros attribute), 38  
coriolis\_t (Veros attribute), 37  
cost (Veros attribute), 37  
cosu (Veros attribute), 37

**D**

dE\_iw (Veros attribute), 61  
deke (Veros attribute), 58  
dHd (Veros attribute), 41  
diagnose() (veros.diagnostics.diagnostic.VerosDiagnostic method), 63  
dpsi (Veros attribute), 47  
dpsin (Veros attribute), 47  
dsalt (Veros attribute), 42

dsalt\_hmix (Veros attribute), 43  
dsalt\_iso (Veros attribute), 43  
dsalt\_vmix (Veros attribute), 42  
dtemp (Veros attribute), 42  
dtemp\_hmix (Veros attribute), 42  
dtemp\_iso (Veros attribute), 43  
dtemp\_vmix (Veros attribute), 42  
dtke (Veros attribute), 57  
du (Veros attribute), 44  
du\_adv (Veros attribute), 45  
du\_cor (Veros attribute), 45  
du\_mix (Veros attribute), 45  
dv (Veros attribute), 45  
dv\_adv (Veros attribute), 46  
dv\_cor (Veros attribute), 45  
dv\_mix (Veros attribute), 45  
dxt (Veros attribute), 36  
dxu (Veros attribute), 36  
dyt (Veros attribute), 36  
dyu (Veros attribute), 36  
dzt (Veros attribute), 37  
dzw (Veros attribute), 37

**E**

E\_iw (Veros attribute), 61  
eke (Veros attribute), 58  
eke\_bot\_flux (Veros attribute), 59  
eke\_diss\_iw (Veros attribute), 59  
eke\_diss\_tke (Veros attribute), 59  
eke\_lee\_flux (Veros attribute), 60  
eke\_len (Veros attribute), 59  
eke\_topo\_hrms (Veros attribute), 60  
eke\_topo\_lam (Veros attribute), 60  
Energy (class in veros.diagnostics.energy), 65  
environment variable  
    BH\_STACK=cuda, 10  
    BH\_STACK=opencl, 10

**F**

flush() (veros.Veros method), 71

flux\_east (Veros attribute), 43  
 flux\_north (Veros attribute), 44  
 flux\_top (Veros attribute), 44  
 forc\_iw\_bottom (Veros attribute), 62  
 forc\_iw\_surface (Veros attribute), 62  
 forc\_rho\_surface (Veros attribute), 47  
 forc\_salt\_surface (Veros attribute), 43  
 forc\_temp\_surface (Veros attribute), 43  
 forc\_tke\_surface (Veros attribute), 58

## H

Hd (Veros attribute), 41  
 hrms\_k0 (Veros attribute), 60  
 ht (Veros attribute), 38  
 hu (Veros attribute), 38  
 hur (Veros attribute), 38  
 hv (Veros attribute), 38  
 hvr (Veros attribute), 39

## I

initialize() (veros.diagnostics.diagnostic.VerosDiagnostic method), 63  
 int\_drhodS (Veros attribute), 41  
 int\_drhodT (Veros attribute), 41  
 isle (Veros attribute), 47  
 iw\_diss (Veros attribute), 62

## K

K\_11 (Veros attribute), 54  
 K\_13 (Veros attribute), 54  
 K\_22 (Veros attribute), 54  
 K\_23 (Veros attribute), 54  
 K\_31 (Veros attribute), 54  
 K\_32 (Veros attribute), 55  
 K\_33 (Veros attribute), 55  
 K\_diss\_bot (Veros attribute), 49  
 K\_diss\_gm (Veros attribute), 49  
 K\_diss\_h (Veros attribute), 49  
 K\_diss\_v (Veros attribute), 49  
 K\_gm (Veros attribute), 49  
 K\_iso (Veros attribute), 49  
 kappa\_gm (Veros attribute), 57  
 kappaH (Veros attribute), 46  
 kappaM (Veros attribute), 46  
 kbot (Veros attribute), 38

## L

L\_rhines (Veros attribute), 59  
 L\_rossby (Veros attribute), 59  
 line\_dir\_east\_mask (Veros attribute), 48  
 line\_dir\_north\_mask (Veros attribute), 48  
 line\_dir\_south\_mask (Veros attribute), 48  
 line\_dir\_west\_mask (Veros attribute), 48  
 line\_psin (Veros attribute), 48

## M

maskT (Veros attribute), 39  
 maskU (Veros attribute), 40  
 maskV (Veros attribute), 40  
 maskW (Veros attribute), 40  
 maskZ (Veros attribute), 40  
 mxl (Veros attribute), 57

## N

name (veros.diagnostics.averages.Averages attribute), 64  
 name (veros.diagnostics.cfl\_monitor.CFLMonitor attribute), 65  
 name (veros.diagnostics.diagnostic.VerosDiagnostic attribute), 63  
 name (veros.diagnostics.energy.Energy attribute), 65  
 name (veros.diagnostics.overturning.Overturning attribute), 65  
 name (veros.diagnostics.snapshot.Snapshot attribute), 64  
 name (veros.diagnostics.tracer\_monitor.TracerMonitor attribute), 65  
 Nsq (Veros attribute), 41

## O

output() (veros.diagnostics.diagnostic.VerosDiagnostic method), 63  
 output\_frequency (veros.diagnostics.averages.Averages attribute), 64  
 output\_frequency (veros.diagnostics.energy.Energy attribute), 65  
 output\_frequency (veros.diagnostics.overturning.Overturning attribute), 65  
 output\_frequency (veros.diagnostics.snapshot.Snapshot attribute), 64  
 output\_frequency (veros.diagnostics.tracer\_monitor.TracerMonitor attribute), 65  
 output\_path (veros.diagnostics.averages.Averages attribute), 64  
 output\_path (veros.diagnostics.energy.Energy attribute), 65  
 output\_path (veros.diagnostics.overturning.Overturning attribute), 65  
 output\_path (veros.diagnostics.snapshot.Snapshot attribute), 64  
 output\_variables (veros.diagnostics.averages.Averages attribute), 64  
 output\_variables (veros.diagnostics.snapshot.Snapshot attribute), 64  
 Overturning (class in veros.diagnostics.overturning), 65

## P

P\_diss\_adv (Veros attribute), 50  
 P\_diss\_comp (Veros attribute), 51  
 P\_diss\_hmix (Veros attribute), 50



P\_diss\_iso (Veros attribute), 50  
 P\_diss\_nonlin (Veros attribute), 50  
 P\_diss\_skew (Veros attribute), 50  
 P\_diss\_sources (Veros attribute), 51  
 P\_diss\_v (Veros attribute), 50  
 p\_hydro (Veros attribute), 46  
 p\_ref (veros.diagnostics.overturning.Overturning attribute), 65  
 Prandtlnumber (Veros attribute), 57  
 prho (Veros attribute), 40  
 psi (Veros attribute), 47  
 psin (Veros attribute), 47

## R

r\_bot\_var\_u (Veros attribute), 56  
 r\_bot\_var\_v (Veros attribute), 56  
 read\_restart() (veros.diagnostics.diagnostic.VerosDiagnostic method), 64  
 restart\_variables (veros.diagnostics.snapshot.Snapshot attribute), 64  
 rho (Veros attribute), 40  
 run() (veros.Veros method), 71

## S

salt (Veros attribute), 42  
 salt\_source (Veros attribute), 53  
 sampling\_frequency (veros.diagnostics.averages.Averages attribute), 64  
 sampling\_frequency (veros.diagnostics.energy.Energy attribute), 65  
 sampling\_frequency (veros.diagnostics.overturning.Overturning attribute), 65  
 set\_coriolis() (veros.Veros method), 70  
 set\_diagnostics() (veros.Veros method), 71  
 set\_forcing() (veros.Veros method), 71  
 set\_grid() (veros.Veros method), 70  
 set\_initial\_conditions() (veros.Veros method), 70  
 set\_parameter() (veros.Veros method), 70  
 set\_topography() (veros.Veros method), 71  
 Snapshot (class in veros.diagnostics.snapshot), 64  
 sqrteke (Veros attribute), 58  
 sqrttke (Veros attribute), 57  
 surface\_taux (Veros attribute), 46  
 surface\_tauy (Veros attribute), 46

## T

tantr (Veros attribute), 37  
 temp (Veros attribute), 41  
 temp\_source (Veros attribute), 53  
 tke (Veros attribute), 57  
 tke\_diss (Veros attribute), 58  
 tke\_surf\_corr (Veros attribute), 58  
 TracerMonitor (class in veros.diagnostics.tracer\_monitor), 65

## U

u (Veros attribute), 44  
 u\_source (Veros attribute), 53  
 u\_wgrid (Veros attribute), 51

## V

v (Veros attribute), 44  
 v0 (Veros attribute), 61  
 v\_source (Veros attribute), 54  
 v\_wgrid (Veros attribute), 51  
 Variable (class in veros.variables), 35  
 Veros (class in veros), 69  
 veros\_inline\_method() (in module veros), 72  
 veros\_method() (in module veros), 72  
 VerosDiagnostic (class in veros.diagnostics.diagnostic), 63

## W

w (Veros attribute), 44  
 w\_wgrid (Veros attribute), 51  
 write\_restart() (veros.diagnostics.diagnostic.VerosDiagnostic method), 64

## X

xt (Veros attribute), 52  
 xu (Veros attribute), 52

## Y

yt (Veros attribute), 52, 53  
 yu (Veros attribute), 52, 53

## Z

zt (Veros attribute), 36  
 zw (Veros attribute), 36