dnora

Release latest

KonstantinChri & bjorkqvi

Sep 12, 2023

CONTENTS:

dnora is a Python package for dynamical downscaling of regional and global wave hindcast/reanalysis using spectral and phase-resolving wave models.

The package contains functions that:

- create a high-resolution grid using open-access bathymetry/topography datasets,
- prepare the boundary conditions (NORA3 hindcast, WAM4km-, & WWIII_4km- operational wave models at MET Norway and ERA5 from ECMWF)
- prepare the wind (NORA3, MEPS, ERA5), ocean (Norkyst800) and water level (GTSM_ERA5) forcing
- create input parameter files for the spectral and phase-resolving models
- run the wave models i.e., SWAN, WWIII, SWASH, HOS-ocean, REEF3D::FNPF.

CHAPTER

ONE

INSTALLING DNORA

1. Install anaconda3 or miniconda3
2. Clone dnora:

```
$ git clone https://github.com/MET-OM/dnora.git
$ cd dnora/
```

3. Create environment with the required dependencies and install dnora

```
$ conda config --add channels conda-forge
$ conda env create -f environment.yml
$ conda activate dnora
$ pip install -e .
```

To update the enviroment using a new environment.yml, run:

```
$ conda env update --file environment.yml --prune
```

CHAPTER

TWO

DEPENDENCIES

1. Installation of **SWAN**. The latest SWAN version can be downloaded from https://sourceforge.net/projects/swanmodel/files/swan/ . The installation procedure can be found in: https://swanmodel.sourceforge.io/online_doc/swanimp/node4.html
2. Installation of **WAVEWATCH III**. The latest model version can be downloaded from https://github.com/NOAA-EMC/WW3 . The installation procedure can be found in: https://github.com/NOAA-EMC/WW3/wiki/Quick-Start
3. Installation of **SWASH**. The latest model version can be downloaded from https://sourceforge.net/projects/swash/ . The installation procedure can be found in: https://swash.sourceforge.io/online_doc/swashimp/node4.html
4. Installation of **HOS-ocean**. The latest HOS-ocean version can be downloaded from https://github.com/LHEEA/HOS-ocean . The installation procedure can be found in: https://github.com/LHEEA/HOS-ocean/wiki/Installation
5. Installation of **REEF3D**. The latest REEF3D and DIVEMesh versions can be downloaded from https://github.com/REEF3D . The installation procedure can be found in: https://reef3d.wordpress.com/installation/ . Note: Only REEF3D::FNPF has been tested in dnora.

To run the models within dnora, the paths, where the models are installed, need to be defined in .bashrc, e.g.,

```
export PATH=${PATH}:/home/user/Programs/swan

export PATH=${PATH}:/home/user/Programs/swash

export PATH=${PATH}:/home/user/Programs/HOS-ocean/bin

export PATH=${PATH}:/home/user/Programs/REEF3D_xx/DIVEMesh/bin
export PATH=${PATH}:/home/user/Programs/REEF3D_xx/REEF3D/bin

source ~/.bashrc
```

CHAPTER

THREE

CREATING A GRID-OBJECT

This section document the grd-module. The grid object is initialized with the following command:

```
grid = grd.Grid(lon=(lon_min, lon_max), lat=(lat_min, lat_max), name='GridName')
grid = grd.Grid(lon=(5.35, 5.6), lat=(59.00, 59.17), name='Boknafjorden') # example for␣
↪Boknafjorden
```

Use `print(grid)` to print out the status of the object.

A desired grid spacing can be set either by providing a desired grid spacing in metres (approximate) or defining the amounts of grid points (exact):

```
grid.set_spacing(dm=250) # Set spacing to around 250 metres
grid.set_spacing(nx=291, ny=249) # Create 291 (lon) x 249 (lat) grid points
```

Both of these options will convert the input to the native resolution in longitude and latitude. These can, of course, also be set directly by:

```
grid.set_spacing(dlon=0.0048780, dlat=0.0022573)
```

In this case `dlon` and `dlat` are not exact. If an exact resolution needs to be forced, the `floating_edge`-option can be used, e.g.,:

```
grid.set_spacing(dlon=1/205, dlat=1/443, floating_edge=True)
```

This will enforce the resolution and instead change the initially set `lon_max` and `lat_max` slightly (if needed). The main properties of the grid can be accessed by methods:

```
grid.lon() # Longitude vector
grid.lat() # Latitude vector
grid.name() # Name given at initialization
grid.nx() # Amount of point in longitude direction
grid.ny() # Amount of point in latitude direction
grid.size() # Tuple (nx, ny)
```

CHAPTER

FOUR

SETTING BOUNDARY POINTS

Setting boundary points is now only important for being able to write the grid-files, but are also of consequence when importing boundary spectra. The central method is to set the edged of the grid to boundary points. In addition, it is possible to search for available spectrum for every e.g. tenth point (step=10) in the boundary

```
bnd_set = grd.boundary.EdgesAsBoundary(edges=['N', 'W', 'S'], step = 10)
grid.set_boundary(boundary_setter=bnd_set)
```

Here the North, West, and South edges are set to boundary points, and this is suitable for e.g. WAVEWATCH III. In SWAN we might want to not set every edge point as a boundary point (and then let the wave model interpolate spectra), especially if the boundary spectra are only available at a coarse resolution. This can be done by initializing the boundary_setter as (every tenth point a boundary point):

```
bnd_set = grd.boundary.EdgesAsBoundary(edges=['N', 'W', 'S'], step=10)
```

Information about the boundary points that are set in the grid can be accessed using methods:

```
grid.boundary_mask() # Boolean array where True is a boundary point
grid.boundary_points() # Array containing a longitude, latitude list of the boundary␣
↪points
```

CHAPTER

FIVE

IMPORTING BATHYMETRICAL DATA

The main idea is that the Grid-object is created, and a fixed set of methods are used to import a topography, mesh it down to a grid, or filter the data. The functionality of these methods are controlled by passing them a callable object. Adding e.g. a topography source thus means adding a new `TopoReader`-class that can then me passed to the `Grid`-object's `.import_topo()`-method. Different bathymetry readers such as:

```
``EMODNET2020``: reads files with NetCDF format (version 2020, possible to read several␣
↪files using tile='*') from https://portal.emodnet-bathymetry.eu/,
``KartverketNo50m``: reads files with xyz format (possible to read several files using␣
↪tile='*') from https://kartkatalog.geonorge.no/metadata/dybdedata-terrengmodeller-50-
↪meters-grid-landsdekkende/bbd687d0-d34f-4d95-9e60-27e330e0f76e
``GEBCO2021``: reads files with NetCDF format from https://download.gebco.net/
```

Examples:

```
topo_reader=grd.read.EMODNET2020(tile='D5',folder='/home/user/bathy/')
topo_reader=grd.read.GEBCO2021(tile='n66.357421875_s57.041015625_w0.703125_e10.37109375',
↪folder='/home/user/bathy/')
topo_reader=grd.read.KartverketNo50m(tile='B1008',folder='/home/user/bathy/')

grid.import_topo(topo_reader=topo_reader)
```

```

```

where the tile indicates the geographical area. This "raw" data can be processed by the `.process_topo()` command, taking a `GridProcessor` object. The data can me meshed to the desired grid definition by:

```
grid.mesh_grid()
```

```

```

The default (and currently only available) `GridMesher` uses interpolation, and is set as default. After meshing the grid data can also be processed with a `GridProcessor`. For example, to set all depth below 1 metre to land and after that impose a minimum of 2 metre depth in wet points, use:

```
grid.process_grid(grd.process.SetMinDept(min_depth=1, to_land=True))
grid.process_grid(grd.process.SetMinDept(min_depth=2))
```

```

```

CHAPTER

SIX

CREATING A MODELRUN-OBJECT

The ModelRun-object is the second central object and contain all forcing and boundary data. This object is always defined for a certain grid and a certain time (if start_time/end_time is not given, it assumes default values:

```
model = mdl.ModelRun(grid, start_time='2018-08-25T00:00', end_time='2018-08-25T01:00')
```

The grid data can now be exported in a certain format using a GridWriter. To export in WAVEWATCH III format:

```
model.export_grid(grd.write.WW3())
```

Boundary and Forcing data can be read using BoundaryReaders and ForcingReaders. To read in boundary spectra and wind forcing from the MET Norway NORA3 hindcast:

```
model.import_boundary(bnd.read_metno.NORA3())
model.import_forcing(wnd.read_metno.NORA3())
```

For forecasting it is possible to use boundaries from WAM4km and wind forcing from MEPS providing the time of the available last_file at thredds.met.no (usually -6 or -12 hours from the current time), e.g.,:

```
model.import_boundary(bnd.read_metno.WAM4km(last_file='2022-11-17T00:00'))
model.import_forcing(wnd.read_metno.MEPS(last_file='2022-11-17T00:00'))
```

To write the boundary spectra in WAVEWATCH III format and wind forcing in SWAN format, use:

```
model.export_boundary(bnd.write.WW3())
model.export_forcing(wnd.write.SWAN())
```

The spectral convention is defined in the BoundaryReader and BoundaryWriter, and the ModelRun-object automatically takes care of convention changes (if needed).

NB! The WW3 convention here is thath of the WW3 *output* files, i.e. directional vector looks like a mathematical convention, but it is actually oceanic. This is in line with the bounc.ftn file used in the develop-branch of WAVEWATCH III.

CHAPTER

SEVEN

CREATING TEMPLATES

Several features that are typically used together can be packaged as a "template" by creating a subclass of the `ModelRun` object. These are defined in the `mdl/models.py`-file. For example, a `WW3`-template is defined as:

```
class WW3(ModelRun):
    def _get_boundary_writer(self):
        return bnd.write.WW3()

    def _get_forcing_writer(self):
        return wnd.write.WW3()

    def _get_point_picker(self):
        return bnd.pick.Area()

    def _get_grid_writer(self):
        return grd.write.WW3()
```

These defaults can be used by:

```
model = mdl.WW3(grid, start_time='2018-08-25T00:00', end_time='2018-08-25T01:00')

model.import_boundary(bnd.read_metno.NORA3()) # PointPicker defined in template
model.export_boundary() # BoundaryWriter defined in template
```

Further subclasses can also be defied. For example to have a `ModelRun`-object that uses WW3 conventions and gets the forcing data from the NORA3-hindcast, a `WW3_NORA3`-template is defined using the above `WW3`-template:

```
class WW3_NORA3(WW3):
    def _get_boundary_reader(self):
        return bnd.read_metno.NORA3()

    def _get_forcing_reader(self):
        return wnd.read_metno.NORA3()
```

The above importing and exporting of NORA3 boundary is now simplified to:

```
model = mdl.WW3_NORA3(grid, start_time='2018-08-25T00:00', end_time='2018-08-25T01:00')

model.import_boundary() # BoundaryReader and PointPicker defined in template
model.export_boundary() # BoundaryWriter defined in template
```

```

```

The defaults of the templates can always be overridden by explicitly providing an object to the method. For example, the following code import WAM 4km boundary spectra, not NORA3 spectra:

```
model = mdl.WW3_NORA3(grid, start_time='2018-08-25T00:00', end_time='2018-08-25T01:00')

model.import_boundary(bnd.read_metno.WAM4km()) # Override BoundaryReader but use␣
↪template PointPicker
model.export_boundary() # BoundaryWriter defined in template
```

```

```

CHAPTER

EIGHT

RUN THE SPECTRAL MODEL

This fucntionality is at the moment only available for SWAN and SWASH. To run the model automatically we first need to generate an input file:

```
model.write_input_file()
```

This generates an input file based on the grid, boundary and forcing data that is available in the object. After that, the model can be automatically ran by:

```
model.run_model()
```

CHAPTER

NINE

FILE NAMES AND DIRECTORIES

The default file names and directories used in dnora are defined in the `defaults.py`-file. Different default can be generated for different models, but the styles are not inherently linked to a certain models (see example below).

The default file names and folders used by the different writers are set within the writers, and they convey their preference to the `ModelRun`-object. These defaults are used if the user doesn't provide anything explicitly. For example, the default file name for writing wind forcing for the SWAN model is:

```
wind#Forcing#Grid#T0_#T1.asc
```

where `#Forcing` and `#Grid` will be replaced by the name of the forcing and grid of the `ModelRun`-object, and `#T0` and `#T1` will be replaced by the start and end times, formatted according to the default format: `%Y%m%d`.

Let's say we want to run an operation version of the SWAN model for the grid name "Sula", and want to have the forcing file name in the format: `WIND_Sula_2018010106Z.asc`, where the time is the start time. The first way to do this is to provide this information to the method doing the writing, e.g.:

```
model.export_forcing(wnd.write.SWAN(), filestring="WIND_#Grid_#T0Z.asc", datestring="%Y%m
↪%d%H")
```

If this is used often, then these values can be added to `defaults.py` under the name "SWAN_oper". Then we can simply set the preference format upon initialization of the `ForcingWriter`:

```
model.export_forcing(wnd.write.SWAN(out_format='SWAN_oper'))
```

The third level is to actualy create a new template for this type of model runs, which can be done (for example) as a subclass of the `SWAN`-template:

```
class SWAN_oper(SWAN):
    def _get_forcing_writer(self):
        return wnd.write.SWAN(out_format='SWAN_oper')
```