Xcroco documentation

Xcroco is a library written in python to post-process history files genarated by the Croco model.
The `Xcroco` directory is provided by the CROCO_PYTOOLS.
For the installation, see Xcroco main page

Content of the library

The Xcroco library is composed of 5 modules:

- model.py : to create a class which defines the concordances between the variables of your files and those used in the Xcroco library
- gridop.py : for opening the history files and for all the operations relative to the grid
- diags.py : brings together the diagnostics available in the Xcroco library
- plot.py : for plotting images or making movies
- tools.py : methods to manage dask clusters and to store a dataset to the disk.

and 2 tutorials:

- tuto_xcroco.ipynb : a notebook with several examples of diagnostics
- tuto_movie.ipynb : a notebook how to make a movie

First, customize your own class for your history files

Currently, your have two models availables in the "*model.py* module:

- croco_xios: for history files created through the XIOS library
- croco_native: for history files directly created by the CROCO model

Either you modify one of the available models to make it match your files or you create a new one.

If you modify an existing template, you must not delete any rows from the dictionary and only modify the dictionary keys (the left part before the "😊.
If you choose to create a new template, start from an axisting one, keep all the rows and as above only change the keys of the directory.

```
elif name == "mytemplate":
    self.rename_vars = {
        # surface wind stress
        "mysustr"        : "xtau_sfc_u",        # x-wind stress component
        ...
    }
```

Once your model is defined, you can use it in your notebook in the following way

```
from model import Model

mymodel = Model("mytemplate")
```

Open your history files

Once your model is defined to describe the variables of your history files, you can open them:

```
import gridop as gop
from model import Model

# Initialisation locale
path = 'your_path'
filenames = [path+'history.nc']
gridname  = path+'grid.nc'

mymodel = Model("mytemplate")
ds, xgrid = gop.open_files(mymodel, gridname, filenames)
```

- ds is a xarray dataset that reflects everything in your grid and history files
- xgrid is a XGCM grid which will be useful for carrying out operations on the spatial grid

Details of the modules

To find out all the arguments available for each method, consult the method directly in the module.

1. model.py :
    - "*_init_*" Retrieves the dictionary to match file variables with those used in the library

      ```
      from model import Model
      model = Model("croco_xios")
      ```

2. gridop.py
    - **open_files** : open Netcdf files or a zarr archive
      `ds,xgrid = gop.open_files(model, gridname, filenames)`
        - Args:
            - model : instance of the Model class defined in the model.py module
            - gridname : path to the grid file
            - filenames : path to the Netcdf files or to the zarr archive
        - Returns:
            - ds: an xarray dataset
            - xgrid: the associated xgcm grid

- **open_catalog**: open files through an intake catalog
  `ds,xgrid = gop.open_files(model, gridname, filenames)`
  - Args:
    - model : instance of the Model class defined in the model.py module
    - gridname : path to the grid file
    - catalog : path to the intake yaml catalog
  - Returns:
    - ds: an xarray dataset
    - xgrid: the associated xgcm grid

- **force_cf_convention** : Force CF convention attributes of dimensions and coordinates for using cf_xarray
  `ds = force_cf_convention(ds)`
  - Args:
    - ds (dataset): input xarray dataset
  - Returns:
    - ds (dataset): xarray dataset with CF convention

- **find_var** : Find a variable, in the gridname or history files variables or attributes
  `find_var(model, varname, ds, gd)`
  - Args:
    - model (string): model class
    - varname (string): variable name to find
    - ds (dataset): dataset of history file
    - gd (dataset): dataset of the grid
  - Returns:
    - (DataArray): the DataArray corresponding to varname

- **get_cs** : get croco vertical grid stretching https://www.myroms.org/wiki/Vertical_S-coordinate
  `cs = get_cs(model, ds, gd, vgrid)`
  - Args:
    - model (class): classe of the model
    - ds (DataSet): input dataset from the history files
    - gd (DataSet): DataSet of the grid file
    - vgrid (character): type of metrics ('r'': rho level, 'w': w level)
  - Returns:
    - DataArray: vertical grid stretching
    -

- **add_grid** : from the gridname file, add the grid to the dataset and compute the XGCM grid
  `ds, xgrid = add_grid(model, ds, gridname)`
  - Args:
    - model (class): classe of the model

        - ds (DataSet): input dataset from the history files
        - gridname (string): name of the grid file
    - Returns:
        - DataSet: the input dataset with the grid inside
        - XGCM grid: the XGCM grid associated to the dataset
        -

- **remove_ghost_points** : Remove ghost points from the DataSet
  `ds = remove_ghost_points(ds)`

    - Args:
        - ds (DataSet): input dataset from the history files
    - Returns:
        - DataSet: the input dataset without any ghost points

- **xgcm_grid** : Create the xgcm grid of the dataset
  `ds, xgrid = xgcm_grid(model)`

    - Args:
        - model: (Model class) the model class
    - Returns:
        - DataSet : the dataset with the news metrics
        - XGCM grid: the xgcm grid of the dataset

- **fast_xgcm_grid** : Create the xgcm grid without computing any metrics. Just use those which are already in the dataset
  `xgrid = fast_xgcm_grid(ds)`

    - Args:
        - ds: (Xarray Dataset) the dataset to create the xgcm grid
    - Returns:
        - XGCM grid: the xgcm grid of the DataSet

- **dll_dist** : Converts lat/lon differentials into distances in meters
  `dx, dy = dll_dist(dlon, dlat, lon, lat)`

    - Args:
        - dlon : xarray.DataArray longitude differentials
        - dlat : xarray.DataArray latitude differentials
        - lon : xarray.DataArray longitude values
        - lat : xarray.DataArray latitude values
    - Returns:
        - dx : xarray.DataArray distance inferred from dlon
        - dy : xarray.DataArray distance inferred from dlat

- **adjust_grid** : Change the names in the dataset according to the model class
  `ds = adjust_grid(model, ds)`

    - Args:
        - model (Model class): Instance of the model class

        - ds (Dataset): dataset to change
    - Returns:
        - DataSet : changed dataset

- **get_spatial_dims** : Return an ordered dict of spatial dimensions in the s, y, x order
  `dims = get_spatial_dims(v)`

    - Args:
        - v (DataArray) : variable for which you have to guess the dimensions
    - Returns:
        - Dictionary : ordered dimensions

- **get_spatial_coords** : Return an ordered dict of spatial coordinates in the z, lat, lon order
  `coords = get_spatial_coords(v)`

    - Args:
        - v (DataArray) : variable for which you have to guess the coordinates
    - Returns:
        - Dictionary: ordered coordinates

- **order_dims** : Reorder the input variable to typical dimensional ordering `var = order_dims(var)`

    - Args:
        - var (DataArray) : Variable to operate on.
    - Returns:
        - DataArray : with dimensional order ['T', 'Z', 'Y', 'X'], or whatever subset of dimensions are present in var.

- **to_rho** : Interpolate to rho horizontal grid
  `var = to_rho(v, grid)`

    - Args:
        - v (DataArray): variable to interpolate
        - grid (xgcm.grid): grid object associated with v
    - Returns:
        - DataArray: input variable interpolated on a rho horizontal point

- **to_u** Interpolate to u horizontal grid
  `var = to_u(v, grid)`

    - Args:
        - v (DataArray): variable to interpolate
        - grid (xgcm.grid): grid object associated with v
    - Returns:
        - DataArray: input variable interpolated on a u horizontal point

- **to_v** Interpolate to v horizontal grid
  `var = to_v(v, grid)`

    - Args:

        - v (DataArray): variable to interpolate
        - grid (xgcm.grid): grid object associated with v
    - Returns:
        - DataArray: input variable interpolated on a v horizontal point

- **to_psi** Interpolate to psi horizontal grid
  `var = to_psi(v, grid)`
    - Args:
        - v (DataArray): variable to interpolate
        - grid (xgcm.grid): grid object associated with v
    - Returns:
        - DataArray: input variable interpolated on a psi horizontal point

- **to_s_rho** : Interpolate to rho vertical grid
  `var = to_s_rho(v, grid)`
    - Args:
        - v (DataArray): variable to interpolate
        - grid (xgcm.grid): grid object associated with v
    - Returns:
        - DataArray: input variable interpolated on a rho vertical level

- **to_s_w** : Interpolate to w vertical grid
  `var = to_s_w(v, grid)`
    - Args:
        - v (DataArray): variable to interpolate
        - grid (xgcm.grid): grid object associated with v
    - Returns:
        - DataArray: input variable interpolated on a w vertical level

- **to_grid_point** : Interpolate to a new grid point
  `var = to_grid_point(var, grid, hcoord=None, vcoord=None)`
    - Args:
        - var: DataArray or ndarray Variable to operate on.
        - xgrid: xgcm.grid Grid object associated with var
        - hcoord: string, optional. Name of horizontal grid to interpolate output to. Options are 'r', 'rho','p', 'psi', 'u', 'v'.
        - vcoord: string, optional. Name of vertical grid to interpolate output to. Options are 's_rho', 's_w', 'rho', 'r', 'w'.
    - Returns:
        - DataArray or ndarray interpolated onto hcoord horizontal and vcoord vertical point.

- **get_z** : Compute the vertical coordinates
  `z = get_z(model)`
    - Args:

        - model (Model class) : the class of the model (containing ds as default)
    - Returns:
        - DataArray : the z coordinate

- **rot_uv** : Rotate u,v to lat,lon coordinates `[urot, vrot] = rot_uv(u, v, angle, xgrid)`
    - Args:
        - u: (DataArray) 3D velocity components in XI direction
        - v: (DataArray) 3D velocity components in ETA direction
        - angle: (DataArray) Angle [radians] between XI-axis and the direction to the EAST at RHO-points
        - xgrid: (xgcm.grid) grid object associated with u and v
    - Returns:
        - DatArray: rotated velocities, urot/vrot at the horizontal u/v grid point

- **get_grid_point** : Get the horizontal and vertical grid point of a variable `hpoint, vpoint = get_grid_point(var)`
    - Args:
        - var (DataArray): variable to operate on
    - Returns:
        - character, character: horizontal, vertical grid point

- **slices** : interpolate a 3D variable on slices at constant depths/longitude/latitude `slice = slices(model, var, z, ds=None, xgrid=None, longitude=None, latitude=None, depth=None)`
    - Args:
        - model (Model class) instance of the Model class
        - var (dataArray) Variable to process (3D matrix).
        - z (dataArray) Depths at the same point than var (3D matrix).
        - ds dataset to find the grid
        - xgrid (XGCM grid) XGCM grid of the dataset
        - longitude (scalar,list or ndarray) longitude of the slice
        - latitude (scalar,list or ndarray) latitude of the slice
        - depth (scalar,list or ndarray) depth of the slice (meters, negative)
    - Returns:
        - (dataArray) slice

- **isoslice** : Interpolate var to target `isovar = isoslice(var, target, xgrid)`
    - Args:
        - var: DataArray Variable to operate on.
        - target: ndarray Values to interpolate to. If calculating var at fixed depths, target are the fixed depths, which should be negative if below mean sea level. If input as array, should be 1D.
        - xgrid: xgcm.grid, optional Grid object associated with var.

    - Returns:
        - DataArray of var interpolated to target

- **cross_section** : Extract a section between 2 geographic points
  `cross = cross_section(grid, da, lon1, lat1, lon2, lat2)`

    - Args:
        - grid (XGCM grid): the XGCM grid associated
        - da (DataArray): variable to operate on
        - lon1 (float): minimum longitude
        - lat1 (float): minimum latitude
        - lon2 (float): maximum longitude
        - lat2 (float): maximum latitude
    - Returns:
        - DataArray: new section

- **interp_regular** : Interpolate on a regular grid `var = interp_regular(da, grid, axis, tgrid)`

    - Args:
        - da (DataArray) : variable to interpolate
        - grid (xgcm grid): xgcm grid
        - axis (str): axis of the xgcm grid for the interpolation ('x', 'y' or 'z')
        - tgrid (numpy vector): target relular grid space
    - Returns:
        - (DataArray): regurlarly interpolated variable

- **haversine** : Calculate the great circle distance between two points on the earth (specified in decimal degrees)
  `distance = haversine(lon1, lat1, lon2, lat2)`

    - Args:
        - lon1 (float): minimum longitude
        - lat1 (float): minimum latitude
        - lon2 (float): maximum longitude
        - lat2 (float): maximum latitude
    - Returns:
        - float: distance in km

- **auto_chunk** : Rechunk a Dataset or DataArray such as each partition size is about a specified chunk `ds = auto_chunk(ds)`

    - Args:
        - ds : (Dataset or DataArray) object to rechunk
    - Returns:
        - (same as input) object rechunked

---

3. diags.py

- **relative_vorticity_z** : Compute the relative vorticity at a constant z depth `vort = relative_vorticity_z(u, v, xgrid)`
  - Args:
    - u : xarray DataArray: velocity component in the x direction
    - v : xarray DataArray: velocity component in the y direction
    - xgrid : xgcm.grid: Grid object associated with u, v
  - Returns:
    - DataArray : the relative vorticity
- **relative_vorticity_sigma** : Compute the vertical component of the relative vorticity [1/s] `vort = relative_vorticity_sigma(u, v, xgrid)`
  - Args:
    - u : xarray DataArray: velocity component in the x direction
    - v : xarray DataArray: velocity component in the y direction
    - xgrid : xgcm.grid: Grid object associated with u, v
  - Returns:
    - DataArray : the relative vorticity at the psi/w grid point
- **dtempdz** : Compute dT/dz
  `dtdz = dtempdz(model, ds=None, xgrid=None, temp=None, z=None)`
  - Args:
    - model (Model class) : Model class instance
    - ds (DataSet, optional) : the dataset containing the fields (T, z)
    - xgrid (XGCM grid): the XGCM grid associated to the dataset
    - temp (DataArray) : temperature
    - z (DataArray): z coordinate
  - Returns:
    - (DataArray) : dTdz at the horizontal rho/vertical w grid point
- **richardson** : Compute the Richardson number
  `Ri = richardson(model, ds=None, u=None, v=None, rho=None, z=None, xgrid=None)`
  - Args:
    - model (Model class) : Model class instance
    - ds (DataSet, optional) : the dataset containing the fields u,v,rho,z
    - u (DataArray) : xi component of the velocity
    - v (DataArray) : eta component of the velocity
    - rho (DataArray) : density
    - z (DataArray): z coordinate
    - xgrid (XGCM grid): the XGCM grid associated to the dataset
  - Returns:
    - (DataArray) : the Richardson number at the horizontal rho/vertical w grid point
- **get_N2** : Compute square buoyancy frequency N2
  `N2 = get_N2(model, ds=None, rho=None, z=None, rho0=None, g=None,`

`xgrid=None)`

- Args:
  - model (Model class) : Model class instance
  - ds (DataSet, optional) : the dataset containing the fields rho,z
  - rho (DataArray) : density
  - z (DataArray): z coordinate
  - rho0 (float) : reference density
  - g (float) : acceleration of the gravity
  - xgrid (XGCM grid): the XGCM grid associated to the dataset
- Returns:
  - (DataArray) : computed square buoyancy frequency at (rho horizontal, w vertical) grid point

- **get_p** : Compute the pressure by integration from the surface
  `p = get_p(model, rho, z_w, z_r, ds=None, g=None, rho0=None, xgrid=None)`
  - Args:
    - model (Model class) : Model class instance
    - rho (DataArray) : density
    - z_w (DataArray): z coordinate on w levels
    - z_r (DataArray): z coordinate on rho levels
    - ds (DataSet, optional) : the dataset containing the fields rho
    - rho0 (float) : reference density
    - g (float) : acceleration of the gravity
    - xgrid (XGCM grid): the XGCM grid associated to the dataset
  - Returns:
    - (DataArray) : Pressure at (rho horizontal, rho vertical) grid point

- **power_spectrum** : Compute the spectrum of the dataarray over the dimensions dims
  `spectrum = power_spectrum(da, dims)`
  - Args:
    - da : (DataArray) input data
    - dims : (str or list of str) dimensions of da on which to take the FFT
  - Returns:
    - DataArray : the power spectrum of the input DataArray

---

4. plot.py

- **plotfig** : Plot an 2d xarray DataArray
  `plotfig(da)`
  - Args:
    - da (DataArray) : 2D variable to plot

- **movie_wrapper** : Make a movie in time
  `movie_wrapper(da, client)`

- Args:
  - da (DataArray) : 3D variable to operate on (time, 2D spatial)

---

5. tools.py

- **wait_cluster_ready** : Wait for the client to be ready (all workers started)
  `wait_cluster_ready(cluster, nworkers)`
  - Args:
    - cluster (dask cluster)
    - nworkers (float) : number of workers in the cluster
- **dask_compute_batch** : breaks down a list of computations into batches
  `outputs = dask_compute_batch(computations, client)`
  - Args:
    - computations (dask delayed computation)
    - client (dask cluster client)
  - Returns:
    - (tuple of tuples) : outputs of the delayed computations
- **store_zarr** : writes a DataSet to a zarr archive
  `store_zarr(ds, zarr_archive)`
  - Args:
    - ds (DataSet) : dataset to store
    - zarr_archive (string) : path to the zarr archive
- **store_netcdf** writes a DataSet to a Netcdf file
  `store_netcdf(ds, filename)`
  - Args:
    - ds (DataSet) : dataset to store
    - filename (string) : path to the Netcdf file