extract_model

Release 1.4.0

Kristen Thyng

Feb 12, 2024

EXAMPLES AND DEMOS

Use *extract_model* to read select output from model output files by time and/or space. Output will be selected using *xarray* by a combination of interpolation and index selection. Horizontal interpolation is accomplished using *xESMF* and time interpolation is done with *xarray*'s native 1D interpolation. Currently vertical interpolation is only possible using *xarray*'s 1D interpolation too and is not set up to interpolate in 4D as would be required for ROMS output if not simply selecting the surface layer.

CHAPTER

ONE

INSTALLATION

To install from conda-forge:

```
>>> conda install -c conda-forge extract_model
```

To install from PyPI:

```
>>> pip install extract_model
```

1.1 Feature types

Feature types are defined by NCEI and provide structure types of data to expect. More information is available in general and for the current NCEI NetCDF Templates 2.0. The following information may be useful for thinking about this. In particular, you select `locstream`, `locstreamT`, and `locstreamZ` as a user for `em.select()` and this table can guide how to select.

	timeSeries	profile	timeSeriesProfile	trajectory (TODO)	trajectoryProfile	grid (TODO)
Definition	only t changes	only z changes	t and z change	t, y, and x change	t, z, y, and x change	t changes, y/x grid
Data types	mooring, buoy	CTD profile	moored ADCP	flow through, 2D drifter	glider, transect of CTD profiles, towed ADCP, 3D drifter	satellite, HF Radar
Model extraction	time series at surface or depth		vertical cross section	2D drifters	3D drifters	regridding, x/y slice in depth
X/Y are pairs ("locstream") or grid	either locstream or grid	either locstream or grid	either locstream or grid	locstream	locstream	grid
Which dimensions are independent from X/Y choice?						
T	Independent	Independent	Independent	locstreamT	locstreamT	Independent
Z	Independent	Independent	Independent	Independent	locstreamZ	Independent

1.2 Generically access model output

```
import cf_xarray
import numpy as np
import xarray as xr
import matplotlib.pyplot as plt
import xcmocean
import cmocean.cm as cmo
import extract_model as em
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning: scipy._lib.messagestream.
↪MessageStream size changed, may indicate binary incompatibility. Expected 56 from C␣
↪header, got 64 from PyObject
```

1.2.1 ROMS

```
# open an example dataset from xarray's tutorials
ds = xr.tutorial.open_dataset('ROMS_example.nc', chunks={'ocean_time': 1})
# normally could run the `preprocess` code as part of reading in the dataset
# but with the tutorial model output, run it separately:
ds = em.preprocess(ds)
ds
```

```
<xarray.Dataset>
Dimensions:     (ocean_time: 2, s_rho: 30, eta_rho: 191, xi_rho: 371)
Coordinates:
    Cs_r        (s_rho) float64 dask.array<chunksize=(30,), meta=np.ndarray>
    lon_rho     (eta_rho, xi_rho) float64 dask.array<chunksize=(191, 371), meta=np.
↪ndarray>
    hc          float64 20.0
    h           (eta_rho, xi_rho) float64 dask.array<chunksize=(191, 371), meta=np.
↪ndarray>
    lat_rho     (eta_rho, xi_rho) float64 dask.array<chunksize=(191, 371), meta=np.
↪ndarray>
    Vtransform  int32 2
  * ocean_time  (ocean_time) datetime64[ns] 2001-08-01 2001-08-08
  * s_rho       (s_rho) float64 -0.9833 -0.95 -0.9167 ... -0.05 -0.01667
  * xi_rho      (xi_rho) int64 0 1 2 3 4 5 6 7 ... 364 365 366 367 368 369 370
  * eta_rho     (eta_rho) int64 0 1 2 3 4 5 6 7 ... 184 185 186 187 188 189 190
    z_rho       (ocean_time, s_rho, eta_rho, xi_rho) float64 dask.array<chunksize=(1, 30,
↪ 191, 371), meta=np.ndarray>
Data variables:
    salt        (ocean_time, s_rho, eta_rho, xi_rho) float32 dask.array<chunksize=(1, 30,
↪ 191, 371), meta=np.ndarray>
    zeta        (ocean_time, eta_rho, xi_rho) float32 dask.array<chunksize=(1, 191, 371),
↪ meta=np.ndarray>
Attributes: (12/34)
    file:               ../output_20yr_obc/2001/ocean_his_0015.nc
    format:             netCDF-4/HDF5 file
    Conventions:        CF-1.4
```

(continues on next page)

(continued from previous page)

```
    type:               ROMS/TOMS history file
    title:              TXLA ROMS hindcast run with dyes and oxygen
    rst_file:           ../output_20yr_obc/2001/ocean_rst.nc
    ...                 ...
    compiler_flags:     -heap-arrays -fp-model fast -mt_mpi -ip -O3 -msse2 -free
    tiling:             010x012
    history:            Tue Jul 24 11:04:43 2018: /opt/nco/ncks -D 4 -t 8 /cop...
    ana_file:           /home/d.kobashi/TXLA_ROMS_reana/Functionals/ana_btflux...
    CPP_options:        TXLA2, ANA_BPFLUX, ANA_BSFLUX, ANA_BTFLUX, ANA_NUDGCOE...
    NCO:                netCDF Operators version 4.7.6-alpha04 (Homepage = htt...
```

Note that the preprocessing code sets up a ROMS dataset so that it can be used with `cf-xarray`. For example, axis and coordinate variables have been identified:

```
ds.cf
```

```
Coordinates:
- CF Axes: * X: ['xi_rho']
           * Y: ['eta_rho']
           * Z: ['s_rho']
           * T: ['ocean_time']

- CF Coordinates:   longitude: ['lon_rho']
                    latitude: ['lat_rho']
                    vertical: ['z_rho']
                  * time: ['ocean_time']

- Cell Measures:   area, volume: n/a

- Standard Names:   latitude: ['lat_rho']
                    longitude: ['lon_rho']
                  * ocean_s_coordinate_g2: ['s_rho']
                  * time: ['ocean_time']

- Bounds:   n/a

Data Variables:
- Cell Measures:   area, volume: n/a

- Standard Names:   sea_surface_elevation: ['zeta']
                    sea_water_practical_salinity: ['salt']

- Bounds:   n/a
```

Variable to use, by standard_name:

```
zeta = 'sea_surface_elevation'
salt = 'sea_water_practical_salinity'
```

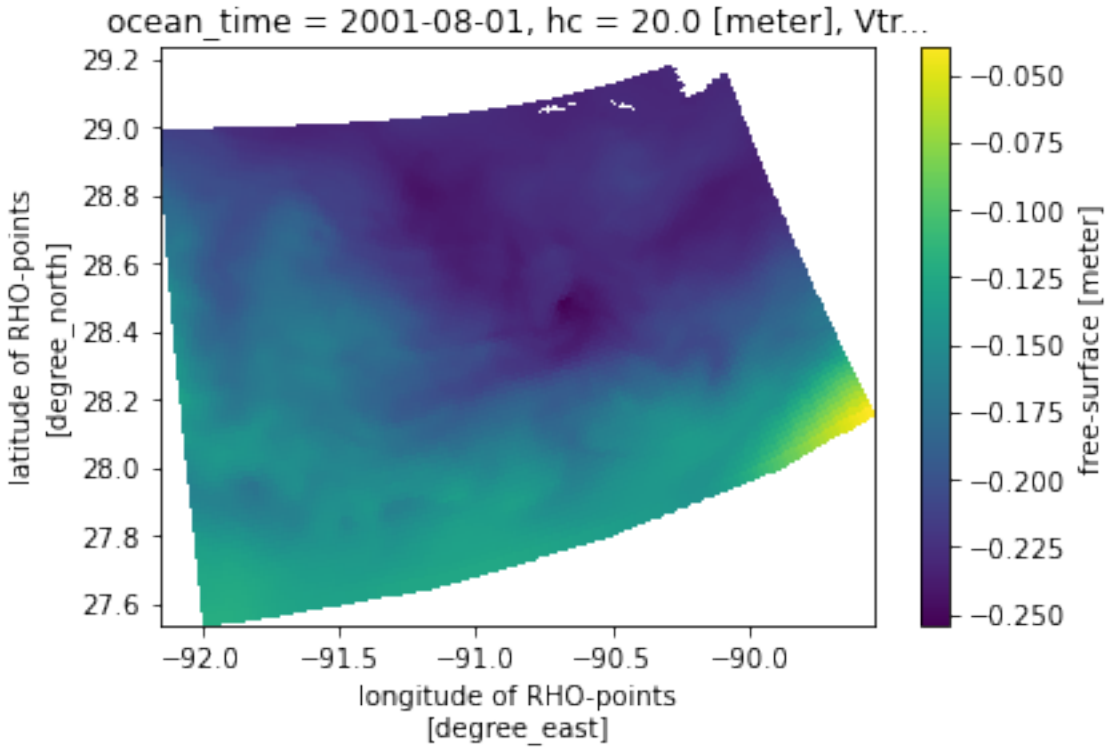
Subset numerical domain

Use `.em.sub_grid()` to narrow the model area down using a bounding box on a Dataset which respects the horizontal structure of multiple grids. Currently only is relevant for ROMS models but will run on any ROMS model or models with a single longitude/latitude set of coordinates.

Resulting area of model will not be exactly the bounding box if the domain is curvilinear.

```
ds_sub = ds.em.sub_grid([-92, 27, -90, 29])
ds_sub.cf[zeta].cf.isel(T=0).cf.plot(x='longitude', y='latitude')
```

```
<matplotlib.collections.QuadMesh at 0x7fbfa182a7c0>
```



Note that this is an unusual ROMS Dataset because it has only one horizontal grid.

```
ds_sub
```

```
<xarray.Dataset>
Dimensions:     (ocean_time: 2, s_rho: 30, eta_rho: 100, xi_rho: 144)
Coordinates:
    Cs_r        (s_rho) float64 dask.array<chunksize=(30,), meta=np.ndarray>
    lon_rho     (eta_rho, xi_rho) float64 dask.array<chunksize=(100, 144), meta=np.
↪ndarray>
    hc          float64 20.0
    h           (eta_rho, xi_rho) float64 dask.array<chunksize=(100, 144), meta=np.
↪ndarray>
    lat_rho     (eta_rho, xi_rho) float64 dask.array<chunksize=(100, 144), meta=np.
↪ndarray>
    Vtransform  int32 2
```

(continues on next page)

(continued from previous page)

```
  * ocean_time  (ocean_time) datetime64[ns] 2001-08-01 2001-08-08
  * s_rho       (s_rho) float64 -0.9833 -0.95 -0.9167 ... -0.05 -0.01667
  * xi_rho      (xi_rho) int64 98 99 100 101 102 103 ... 236 237 238 239 240 241
  * eta_rho     (eta_rho) int64 0 1 2 3 4 5 6 7 8 ... 91 92 93 94 95 96 97 98 99
    z_rho       (ocean_time, s_rho, eta_rho, xi_rho) float64 dask.array<chunksize=(1, 30,
↪ 100, 144), meta=np.ndarray>
Data variables:
    salt        (ocean_time, s_rho, eta_rho, xi_rho) float32 dask.array<chunksize=(1, 30,
↪ 100, 144), meta=np.ndarray>
    zeta        (ocean_time, eta_rho, xi_rho) float32 dask.array<chunksize=(1, 100, 144),
↪ meta=np.ndarray>
Attributes: (12/34)
    file:               ../output_20yr_obc/2001/ocean_his_0015.nc
    format:             netCDF-4/HDF5 file
    Conventions:        CF-1.4
    type:               ROMS/TOMS history file
    title:              TXLA ROMS hindcast run with dyes and oxygen
    rst_file:           ../output_20yr_obc/2001/ocean_rst.nc
    ...                 ...
    compiler_flags:     -heap-arrays -fp-model fast -mt_mpi -ip -O3 -msse2 -free
    tiling:             010x012
    history:            Tue Jul 24 11:04:43 2018: /opt/nco/ncks -D 4 -t 8 /cop...
    ana_file:           /home/d.kobashi/TXLA_ROMS_reana/Functionals/ana_btflux...
    CPP_options:        TXLA2, ANA_BPFLUX, ANA_BSFLUX, ANA_BTFLUX, ANA_NUDGCOE...
    NCO:                netCDF Operators version 4.7.6-alpha04 (Homepage = htt...
```

Subset to a horizontal box

Use `.em.sub_bbox()` to narrow the model area down using a bounding box on either a Dataset or DataArray. There is no expectation of multiple horizontal grids having the "correct" relationship to each other.

Dataset

In the case of a Dataset, all map-based variables are filtered using the same bounding box.

```
ds.em.sub_bbox([-92, 27, -90, 29], drop=True).cf[salt].cf.isel(T=0).cf.sel(Z=0, method=
↪'nearest')
```

```
<xarray.DataArray 'salt' (eta_rho: 100, xi_rho: 144)>
dask.array<getitem, shape=(100, 144), dtype=float32, chunksize=(100, 144),␣
↪chunktype=numpy.ndarray>
Coordinates:
    ocean_time  datetime64[ns] 2001-08-01
    s_rho       float64 -0.01667
  * xi_rho      (xi_rho) int64 98 99 100 101 102 103 ... 236 237 238 239 240 241
  * eta_rho     (eta_rho) int64 0 1 2 3 4 5 6 7 8 ... 91 92 93 94 95 96 97 98 99
Attributes:
    long_name:      salinity
    time:           ocean_time
```

(continues on next page)

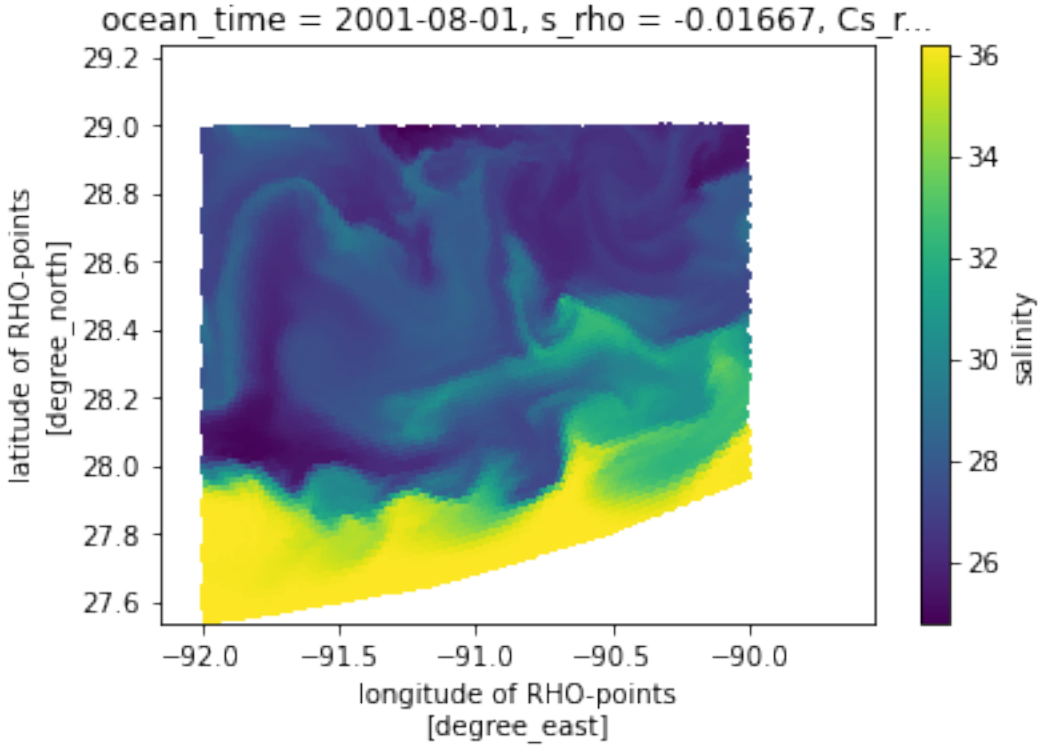
(continued from previous page)

```
    field:          salinity, scalar, series
    standard_name:  sea_water_practical_salinity
```

DataArray

```
ds.cf[salt].em.sub_bbox([-92, 27, -90, 29], drop=True).cf.isel(T=0, Z=-1).cf.plot(x=
↪'longitude', y='latitude')
```

```
<matplotlib.collections.QuadMesh at 0x7fbf81ce3fd0>
```



grid point (interpolation and selecting nearest)

Interpolate to a single existing horizontal grid point (and any additional depth and time values for that location) and compare it with method selecting the nearest point to demonstrate we get the same value.

```
%%time

varname = salt

# Set up a single lon/lat location
j, i = 50, 10
longitude = float(ds.cf[varname].cf['longitude'][j,i])
latitude = float(ds.cf[varname].cf['latitude'][j,i])

```

(continues on next page)

(continued from previous page)

```
# Interpolation
da_out = ds.cf[varname].em.interp2d(longitude, latitude)

# Selection of nearest location in 2D
da_check = ds.cf[varname].em.sel2dcf(longitude=longitude, latitude=latitude).squeeze()

assert np.allclose(da_out, da_check)
```

```
CPU times: user 4.18 s, sys: 164 ms, total: 4.34 s
Wall time: 4.36 s
```

You could also select a time and/or depth index or interpolate in time and/or depth at the same time:

```
# Select time index and depth index
ds.cf[varname].em.interp2d(longitude, latitude, iT=0, iZ=0)
```

```
<xarray.DataArray 'salt' ()>
dask.array<getitem, shape=(), dtype=float32, chunksize=(), chunktype=numpy.ndarray>
Coordinates:
    ocean_time  datetime64[ns] 2001-08-01
    s_rho       float64 -0.9833
    Cs_r        float64 dask.array<chunksize=(), meta=np.ndarray>
    hc          float64 20.0
    Vtransform  int32 2
    lat         float64 28.23
    lon         float64 -93.59
    z_rho       float64 dask.array<chunksize=(), meta=np.ndarray>
Attributes:
    long_name:      salinity
    time:           ocean_time
    field:          salinity, scalar, series
    standard_name:  sea_water_practical_salinity
```

```
ds.cf[varname].cf
```

```
Coordinates:
- CF Axes: * X: ['xi_rho']
           * Y: ['eta_rho']
           * Z: ['s_rho']
           * T: ['ocean_time']

- CF Coordinates:   longitude: ['lon_rho']
                    latitude: ['lat_rho']
                    vertical: ['z_rho']
                  * time: ['ocean_time']

- Cell Measures:   area, volume: n/a

- Standard Names:   latitude: ['lat_rho']
                    longitude: ['lon_rho']
                  * ocean_s_coordinate_g2: ['s_rho']
```

(continues on next page)

(continued from previous page)

```
                  * time: ['ocean_time']

- Bounds:   n/a
```

```
# Interpolate to time value and depth value
ds.cf[varname].em.interp2d(longitude, latitude, T=ds.cf['T'][0], Z=-10)
```

```
<xarray.DataArray 'salt' ()>
dask.array<dask_aware_interpnd, shape=(), dtype=float32, chunksize=(), chunktype=numpy.
↪ndarray>
Coordinates:
    s_rho       float64 -0.4475
    Cs_r        float64 dask.array<chunksize=(), meta=np.ndarray>
    hc          float64 20.0
    Vtransform  int32 2
    lat         float64 28.23
    lon         float64 -93.59
    ocean_time  datetime64[ns] 2001-08-01
    z_rho       int64 -10
Attributes:
    long_name:      salinity
    time:           ocean_time
    field:          salinity, scalar, series
    standard_name:  sea_water_practical_salinity
```

The interpolation is faster the second time the regridder is used — it is saved by the `extract_model` accessor and reused if the lon/lat locations to be interpolated to are the same. Here we interpolate to salinity and it is faster than it was the first time it was used for interpolation the sea surface elevation.

```
%%time

varname = zeta

# Set up a single lon/lat location
j, i = 50, 10
longitude = float(ds.cf[varname].cf['longitude'][j,i])
latitude = float(ds.cf[varname].cf['latitude'][j,i])

# Interpolation
da_out = ds.cf[varname].em.interp2d(longitude, latitude)

# Selection of nearest location in 2D
da_check = ds.cf[varname].em.sel2dcf(longitude=longitude, latitude=latitude).squeeze()

assert np.allclose(da_out, da_check)
```

```
CPU times: user 956 ms, sys: 34.3 ms, total: 990 ms
Wall time: 1 s
```

not grid point

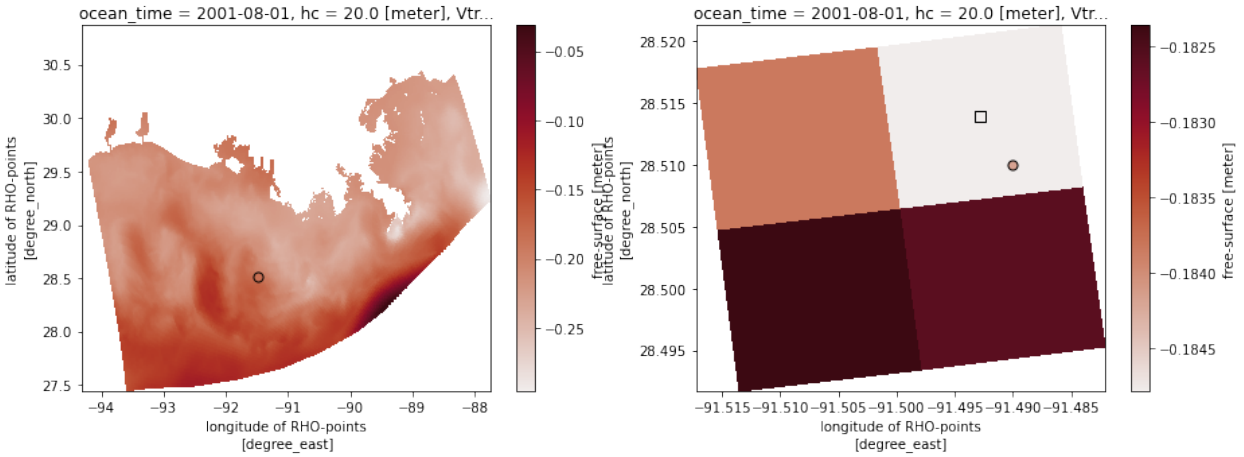
inside domain (interpolation and selecting nearest)

For a selected location that is not a grid point (so we can't check it exactly), we show here both interpolating to that location horizontally and selecting the nearest point to that location.

The square in the right hand side plot shows the nearest point selected using `.em.sel2d()` and the circle shows the interpolated value at the exact selected location using `.em.interp2d()`.

```
varname = zeta

# sel
longitude = -91.49
latitude = 28.510

# isel
iZ = None
iT = 0
isel = dict(T=iT)

# Interpolation
da_out = ds.cf[varname].em.interp2d(longitude, latitude, iT=iT, iZ=iZ)

# Selection of nearest location in 2D
da_sel = ds.cf[varname].em.sel2dcf(longitude=longitude, latitude=latitude, distances_
↪name="distance").cf.isel(T=iT).squeeze()

# Plot
cmap = ds.cf[varname].cmo.seq
dacheck = ds.cf[varname].cf.isel(isel)
fig, axes = plt.subplots(1, 2, figsize=(15,5))

dacheck.cmo.cfplot(ax=axes[0], x='longitude', y='latitude')
axes[0].scatter(da_out.cf['longitude'], da_out.cf['latitude'], s=50, c=da_out,
           vmin=dacheck.min().values, vmax=dacheck.max().values, cmap=cmap, edgecolors='k
↪')

# make smaller area of model to show
# want model output only within the box defined by these lat/lon values
dacheck_min = dacheck.em.sub_bbox([-91.52, 28.49, -91.49, 28.525], drop=True)
dacheck_min.cmo.cfplot(ax=axes[1], x='longitude', y='latitude')
# interpolation
axes[1].scatter(da_out.cf['longitude'], da_out.cf['latitude'], s=50, c=da_out,
           vmin=dacheck_min.min().values, vmax=dacheck_min.max().values,
                cmap=cmap, edgecolors='k')
# selection
axes[1].scatter(da_sel.cf['longitude'], da_sel.cf['latitude'], s=50, c=da_sel.
↪cf[varname],
           vmin=dacheck_min.min().values, vmax=dacheck_min.max().values,
                cmap=cmap, edgecolors='k', marker='s')
```

```
<matplotlib.collections.PathCollection at 0x7fbfc254cb50>
```



We input the extra keyword argument `distances_name` into the call `ds.cf[varname].em.sel2dcf` in order to also return the distance between the requested location and the returned model location. This value is shown here in km:

```
da_sel["distance"]
```

```
<xarray.DataArray 'distance' ()>
array(0.51134)
Coordinates:
    ocean_time  datetime64[ns] 2001-08-01
    xi_rho      int64 136
    eta_rho     int64 54
    lon_rho     float64 dask.array<chunksize=(), meta=np.ndarray>
    hc          float64 20.0
    h           float64 dask.array<chunksize=(), meta=np.ndarray>
    Vtransform  int32 2
    lat_rho     float64 dask.array<chunksize=(), meta=np.ndarray>
Attributes:
    units:    km
```

outside domain

Don't extrapolate

This is commented out since it purposefully raises an error:

> ValueError: Longitude outside of available domain. Use extrap=True to extrapolate.

```
# varname = zeta

# # sel
# longitude = -166
# latitude = 48
# sel = dict(longitude=longitude, latitude=latitude)

# # isel
# iZ = 0
# iT = 0
# isel = dict(Z=iZ, T=iT)
```

(continues on next page)

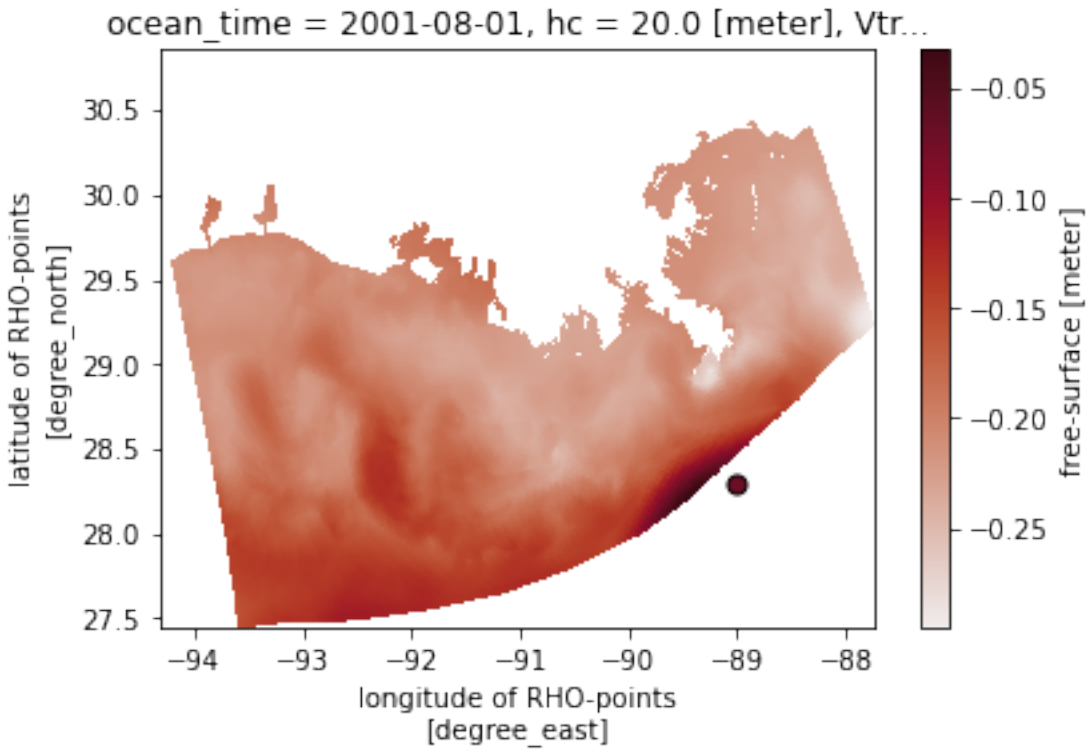
(continued from previous page)

```
# da_out = ds.cf[varname].em.interp2d(longitude, latitude, iT=iT, iZ=iZ, extrap=False)

# da_out
```

Extrapolate

```
varname = zeta

# sel
longitude = -89
latitude = 28.3
sel = dict(longitude=longitude, latitude=latitude)

# isel
iZ = None
iT = 0
isel = dict(T=iT)

da_out = ds.cf[varname].em.interp2d(longitude, latitude, iT=iT, iZ=iZ, extrap=True)

# plot
cmap = ds.cf[varname].cmo.seq
dacheck = ds.cf[varname].cf.isel(isel)
fig, ax = plt.subplots(1,1)
dacheck.cmo.cfplot(ax=ax, x='longitude', y='latitude')
ax.scatter(da_out.cf['longitude'], da_out.cf['latitude'], s=50, c=da_out,
           vmin=dacheck.min().values, vmax=dacheck.max().values, cmap=cmap, edgecolors='k
↪')
```

```
<matplotlib.collections.PathCollection at 0x7fbfa1e57700>
```



points (locstream, interpolation)

Interpolate to unstructured pairs of lon/lat locations instead of grids of lon/lat locations, using `locstream`. Choose grid points so that we can check the accuracy of the results.

```
varname = zeta

# sel
# this creates 12 pairs of lon/lat points that
# align with grid points so we can check the
# interpolation
longitude = ds.cf[varname].cf['longitude'].isel(eta_rho=60, xi_rho=slice(None,None,10))
latitude = ds.cf[varname].cf['latitude'].isel(eta_rho=60, xi_rho=slice(None,None,10))
sel = dict(X=longitude.xi_rho, Y=longitude.eta_rho)

# isel
iZ = None
iT = 0
isel = dict(T=iT)

da_out = ds.cf[varname].em.interp2d(longitude, latitude, iT=iT, iZ=iZ, locstream=True)

# check
da_check = ds.cf[varname].cf.isel(isel).cf.sel(sel)

assert np.allclose(da_out, da_check, equal_nan=True)
```

It is not currently possible to interpolate in depth with both more than one time and location.

This cell is commented out because it purposefully returns an error:

> NotImplementedError: Currently it is not possible to interpolate in depth with more than 1 other (time) dimension.

```
# ds.cf[salt].em.interp2d(longitude, latitude, Z=-10, locstream=True)
```

grid of known locations (interpolation)

```
varname = zeta

# sel
longitude = ds.cf[varname].cf['longitude'][:-50:20,:-200:100]
latitude = ds.cf[varname].cf['latitude'][:-50:20,:-200:100]
sel = dict(X=longitude.xi_rho, Y=longitude.eta_rho)

# isel
iZ = None
iT = 0
isel = dict(T=iT)

da_out = ds.cf[varname].em.interp2d(longitude, latitude, iT=iT, iZ=iZ, locstream=False)

# check
da_check = ds.cf[varname].cf.sel(sel).cf.isel(isel)

assert np.allclose(da_out, da_check)
```

grid of new locations (interpolation, regridding)

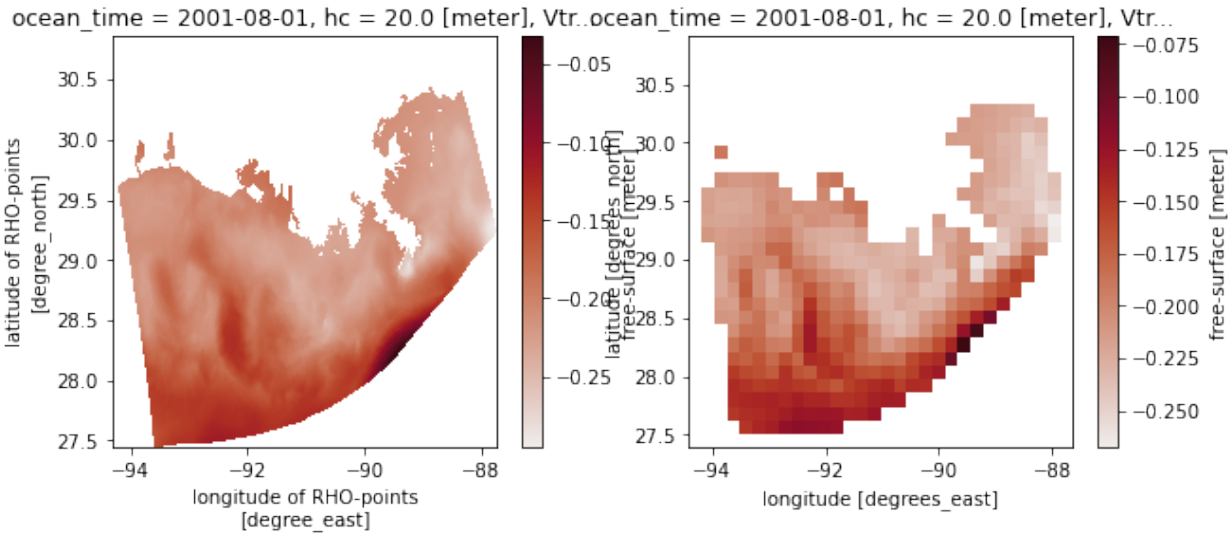
```
varname = zeta

# sel
longitude = np.linspace(ds.cf[varname].cf['longitude'].min(), ds.cf[varname].cf[
↪'longitude'].max(), 30)
latitude = np.linspace(ds.cf[varname].cf['latitude'].min(), ds.cf[varname].cf['latitude
↪'].max(), 30)

# isel
iZ = None
iT = 0
isel = dict(T=iT)

da_out = ds.cf[varname].em.interp2d(longitude, latitude, iT=iT, iZ=iZ, locstream=False,␣
↪extrap=False, extrap_val=np.nan)

# plot
cmap = cmo.delta
dacheck = ds.cf[varname].cf.isel(isel)

fig, axes = plt.subplots(1,2, figsize=(10,4))
```

(continues on next page)

(continued from previous page)

```
dacheck.cmo.cfplot(ax=axes[0], x='longitude', y='latitude')
da_out.cmo.cfplot(ax=axes[1], x='longitude', y='latitude')
```

```
<matplotlib.collections.QuadMesh at 0x7fbf8254f970>
```



1.2.2 HYCOM

```
# url = ['http://tds.hycom.org/thredds/dodsC/GLBy0.08/latest']
# ds = xr.open_mfdataset(url, preprocess=em.preprocess, drop_variables='tau')
# ds.isel(time=slice(0,2)).sel(lat=slice(-20, 30), lon=slice(140,190)).to_netcdf('hycom.nc
↪')
# ds = xr.open_mfdataset('hycom.nc', preprocess=em.preprocess)

url = 'http://tds.hycom.org/thredds/dodsC/GLBy0.08/latest'
ds = xr.open_dataset(url, drop_variables='tau')["water_u"].isel(time=slice(0,2),␣
↪depth=0).sel(lat=slice(-20, 30), lon=slice(140,190))
ds = em.preprocess(ds)
ds = ds.load()
ds
```

```
<xarray.DataArray 'water_u' (time: 2, lat: 1251, lon: 626)>
array([[[        nan,         nan,         nan, ...,  0.002     ,
          0.008     ,  0.011     ],
        [        nan,         nan,         nan, ..., -0.022     ,
         -0.013     , -0.008     ],
        [        nan,         nan,         nan, ..., -0.056     ,
         -0.044     , -0.028     ],
        ...,
        [-0.24200001, -0.23400001, -0.231     , ...,  0.19800001,
          0.163     ,  0.142     ],
        [-0.2       , -0.194     , -0.18800001, ...,  0.142     ,
          0.11800001,  0.109     ],
```

(continues on next page)

(continued from previous page)

```
        [-0.163     , -0.155     , -0.148     , ...,  0.097     ,
          0.08800001,  0.09200001]],

       [[        nan,         nan,         nan, ...,  0.024     ,
          0.028     ,  0.028     ],
        [        nan,         nan,         nan, ..., -0.006     ,
          0.001     ,  0.006     ],
        [        nan,         nan,         nan, ..., -0.042     ,
         -0.031     , -0.015     ],
        ...,
        [-0.22900002, -0.21900001, -0.20500001, ...,  0.132     ,
          0.094     ,  0.07600001],
        [-0.177     , -0.17300001, -0.16600001, ...,  0.081     ,
          0.057     ,  0.053     ],
        [-0.127     , -0.128     , -0.126     , ...,  0.043     ,
          0.035     ,  0.043     ]]], dtype=float32)
Coordinates:
  * lat       (lat) float64 -20.0 -19.96 -19.92 -19.88 ... 29.92 29.96 30.0
  * lon       (lon) float64 140.0 140.1 140.2 140.2 ... 189.8 189.8 189.9 190.0
    depth     float64 0.0
  * time      (time) datetime64[ns] 2023-01-21T12:00:00 2023-01-21T15:00:00
    time_run  (time) datetime64[ns] 2023-01-21T12:00:00 2023-01-21T12:00:00
Attributes:
    _CoordinateAxes:  time_run time depth lat lon
    units:            m/s
    long_name:        Eastward Water Velocity
    standard_name:    eastward_sea_water_velocity
    NAVO_code:        17
```

```
ds.cf
```

```
Coordinates:
- CF Axes: * X: ['lon']
           * Y: ['lat']
             Z: ['depth']
           * T: ['time']

- CF Coordinates: * longitude: ['lon']
                  * latitude: ['lat']
                    vertical: ['depth']
                  * time: ['time']

- Cell Measures:   area, volume: n/a

- Standard Names:   depth: ['depth']
                    forecast_reference_time: ['time_run']
                  * latitude: ['lat']
                  * longitude: ['lon']
                  * time: ['time']

- Bounds:   n/a
```

grid point

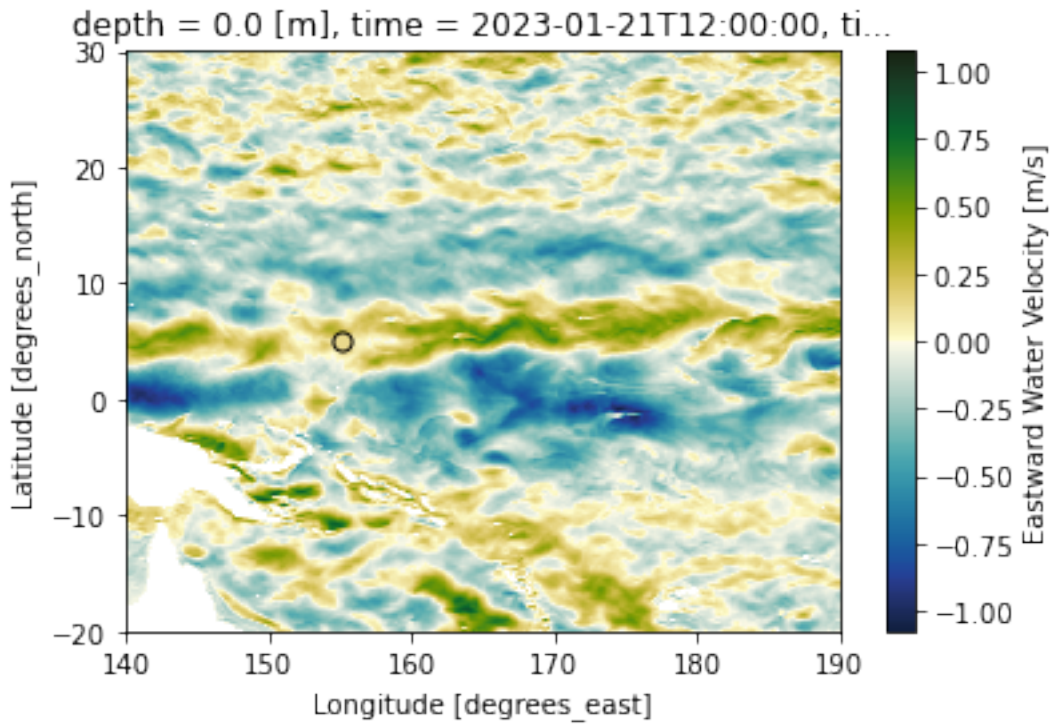
```
# sel
longitude = float(ds.cf['X'][100])
latitude = float(ds.cf['Y'][150])
sel = dict(longitude=longitude, latitude=latitude)

# isel
iZ = None
iT = None
# isel = dict(Z=iZ)

da_out = ds.em.interp2d(longitude, latitude, iT=iT, iZ=iZ)

# check
da_check = ds.cf.sel(sel)#.cf.isel(isel)

assert np.allclose(da_out, da_check)
```

not grid point

inside domain

```
# sel
longitude = 155
latitude = 5
sel = dict(longitude=longitude, latitude=latitude)

# isel
iZ = None
iT = 0
isel = dict(T=iT)

da_out = ds.em.interp2d(longitude, latitude, iT=iT, iZ=iZ)

# plot
cmap = cmo.delta
dacheck = ds.cf.isel(isel)
fig, ax = plt.subplots(1,1)
dacheck.cmo.plot(ax=ax)
ax.scatter(da_out.cf['longitude'], da_out.cf['latitude'], s=50, c=da_out,
           vmin=dacheck.min().values, vmax=dacheck.max().values, cmap=cmap, edgecolors='k
↪')
```

```
<matplotlib.collections.PathCollection at 0x7fbf824e8550>
```



outside domain

Don't extrapolate

This purposefully raises an error so is commented out:

> ValueError: Longitude outside of available domain. Use extrap=True to extrapolate.

```
# # sel
# longitude = -166
# latitude = 48
# sel = dict(longitude=longitude, latitude=latitude)

# # isel
# iZ = None
# iT = 0
# isel = dict(T=iT)

# da_out = ds.em.interp2d(longitude, latitude, iT=iT, iZ=iZ, extrap=False)

# da_out = em.select(**kwargs)
# da_out
```

Extrapolate

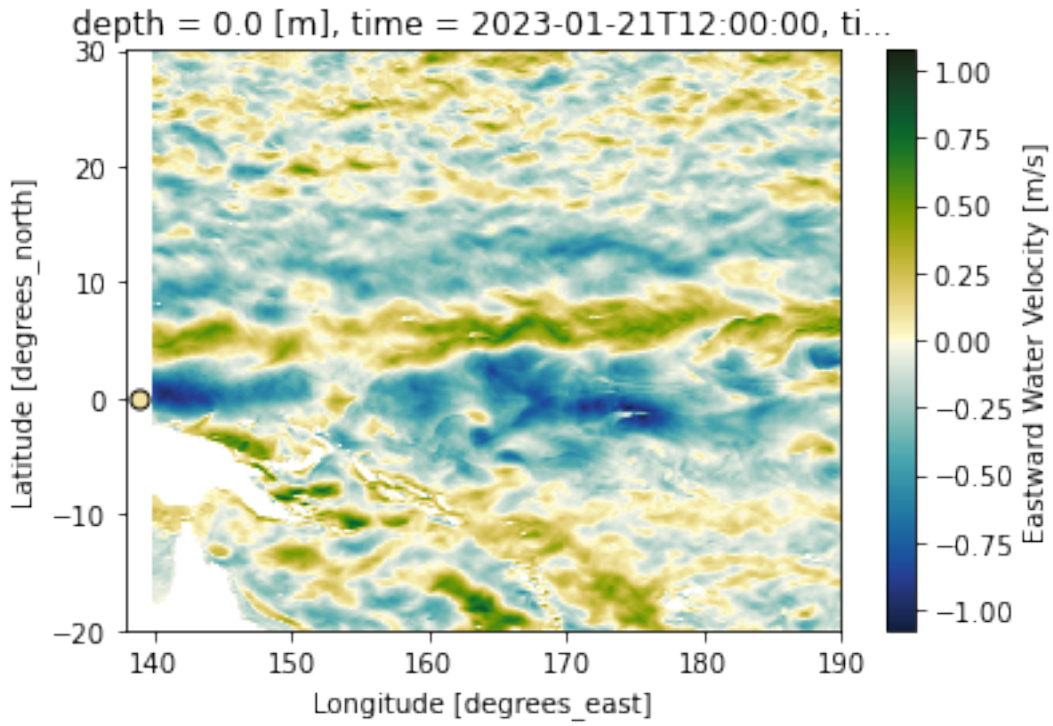
```
# sel
longitude = 139
latitude = 0
sel = dict(longitude=longitude, latitude=latitude)
```

(continues on next page)

(continued from previous page)

```
# isel
iZ = None
iT = 0
isel = dict(T=iT)

da_out = ds.em.interp2d(longitude, latitude, iT=iT, iZ=iZ, extrap=True)

# plot
cmap = cmo.delta
dacheck = ds.cf.isel(isel)
fig, ax = plt.subplots(1,1)
dacheck.cmo.plot(ax=ax)
ax.scatter(da_out.cf['longitude'], da_out.cf['latitude'], s=50, c=da_out,
           vmin=dacheck.min().values, vmax=dacheck.max().values, cmap=cmap, edgecolors='k
↪')

ax.set_xlim(138,190)
```

```
(138.0, 190.0)
```



points (locstream)

Unstructured pairs of lon/lat locations instead of grids of lon/lat locations, using `locstream`.

```
# sel
# this creates 12 pairs of lon/lat points that
# align with grid points so we can check the
# interpolation
longitude = ds.cf['X'][::40].values
latitude = ds.cf['Y'][::80].values
# selecting individual lon/lat locations with advanced xarray indexing
sel = dict(longitude=xr.DataArray(longitude, dims="pts"), latitude=xr.DataArray(latitude,
↪ dims="pts"))

# isel
iZ = None
iT = 0
isel = dict(T=iT)

da_out = ds.em.interp2d(longitude, latitude, iT=iT, iZ=iZ, locstream=True)

# check
da_check = ds.cf.isel(isel).cf.sel(sel)

assert np.allclose(da_out, da_check, equal_nan=True)
```

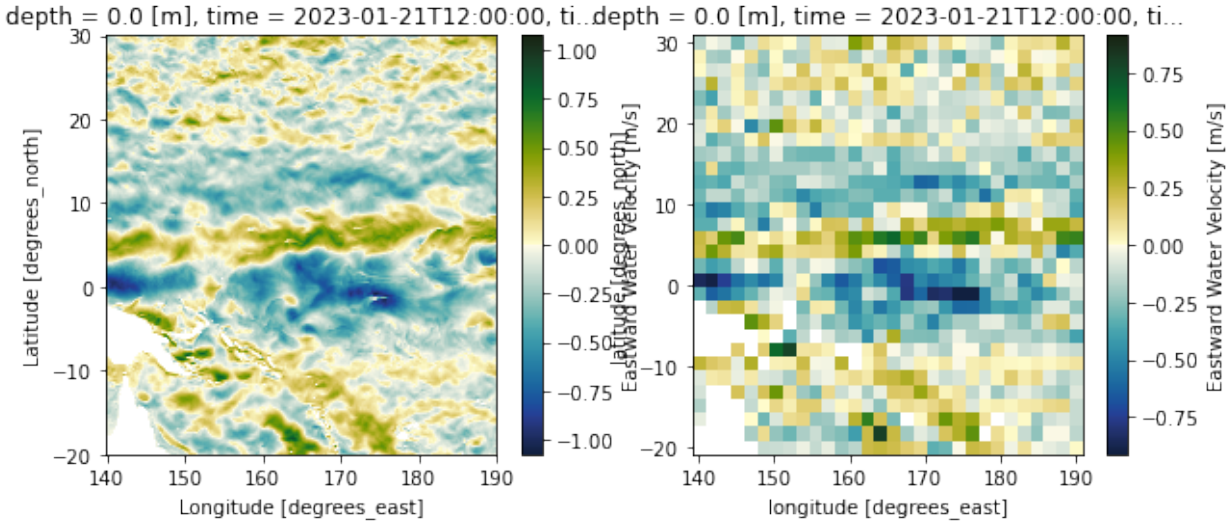
grid of known locations

```
# sel
longitude = ds.cf['X'][100::500]
latitude = ds.cf['Y'][100::500]
sel = dict(longitude=longitude, latitude=latitude)

# isel
iZ = None
iT = None
# isel = dict(Z=iZ)

da_out = ds.em.interp2d(longitude, latitude, iT=iT, iZ=iZ, locstream=False)

# check
da_check = ds.cf.sel(sel)#.cf.isel(isel)

assert np.allclose(da_out, da_check)
```

grid of new locations

```
# sel
longitude = np.linspace(ds.cf['X'].min(), ds.cf['X'].max(), 30)
latitude = np.linspace(ds.cf['Y'].min(), ds.cf['Y'].max(), 30)
sel = dict(longitude=longitude, latitude=latitude)

# isel
iZ = None
iT = 0
isel = dict(T=iT)

da_out = ds.em.interp2d(longitude, latitude, iT=iT, iZ=iZ, locstream=False)
# kwargs = dict(da, longitude=longitude, latitude=latitude, iT=T, iZ=Z)

# da_out = em.select(**kwargs)

# plot
cmap = cmo.delta
dacheck = ds.cf.isel(isel)

fig, axes = plt.subplots(1,2, figsize=(10,4))
dacheck.cmo.plot(ax=axes[0])
da_out.cmo.plot(ax=axes[1])
```

```
<matplotlib.collections.QuadMesh at 0x7fbfc349bee0>
```



1.2.3 POM

```
try:
    url = "https://www.ncei.noaa.gov/thredds/dodsC/model-loofs-agg/Aggregated_LOOFS_
↪Fields_Forecast_best.ncd"
    # url = ['https://opendap.co-ops.nos.noaa.gov/thredds/dodsC/LOOFS/fmrc/Aggregated_7_
↪day_LOOFS_Fields_Forecast_best.ncd']
    # ds = xr.open_mfdataset(url, preprocess=em.preprocess, chunks=None)
    ds= xr.open_dataset(url)
    ds = em.utils.preprocess_pom(ds, interp_vertical=False)
except OSError:
    import pandas as pd
    today = pd.Timestamp.today()
    url = [today.strftime('https://opendap.co-ops.nos.noaa.gov/thredds/dodsC/NOAA/LOOFS/
↪MODELS/%Y/%m/%d/glofs.loofs.fields.nowcast.%Y%m%d.t00z.nc'),
           today.strftime('https://opendap.co-ops.nos.noaa.gov/thredds/dodsC/NOAA/LOOFS/
↪MODELS/%Y/%m/%d/glofs.loofs.fields.nowcast.%Y%m%d.t06z.nc')]
    ds = xr.open_mfdataset(url, preprocess=em.preprocess)

ds = ds["zeta"].isel(time=slice(0,2)).load()
ds
```

```
<xarray.DataArray 'zeta' (time: 2, ny: 25, nx: 61)>
array([[[      nan,       nan,       nan, ...,       nan,       nan,
              nan],
        [      nan,       nan,       nan, ...,       nan,       nan,
              nan],
        [      nan,       nan, 0.5127469, ...,       nan,       nan,
              nan],
        ...,
        [      nan,       nan,       nan, ...,       nan,       nan,
              nan],
        [      nan,       nan,       nan, ...,       nan,       nan,
              nan],
        [      nan,       nan,       nan, ...,       nan,       nan,
              nan]],

       [[      nan,       nan,       nan, ...,       nan,       nan,
              nan],
        [      nan,       nan,       nan, ...,       nan,       nan,
              nan],
        [      nan,       nan, 0.5261942, ...,       nan,       nan,
              nan],
        ...,
        [      nan,       nan,       nan, ...,       nan,       nan,
              nan],
        [      nan,       nan,       nan, ...,       nan,       nan,
              nan],
        [      nan,       nan,       nan, ...,       nan,       nan,
              nan]]], dtype=float32)
Coordinates:
    lon       (ny, nx) float32 -79.79 -79.73 -79.67 ... -76.19 -76.13 -76.07
    lat       (ny, nx) float32 43.14 43.14 43.15 43.15 ... 44.22 44.22 44.22
```

(continues on next page)

(continued from previous page)

```
  * time      (time) datetime64[ns] 2018-01-01T00:59:45 2018-01-01T02:00:13.1...
    time_run  (time) datetime64[ns] 2018-01-01T00:59:45 2018-01-01T00:59:45
  * nx        (nx) int64 0 1 2 3 4 5 6 7 8 9 ... 51 52 53 54 55 56 57 58 59 60
  * ny        (ny) int64 0 1 2 3 4 5 6 7 8 9 ... 15 16 17 18 19 20 21 22 23 24
Attributes:
    units:          meters
    long_name:      Height Above Model Sea Level
    reference:      model_sea_level
    standard_name:  sea_surface_elevation
```

```
ds.cf
```

```
Coordinates:
- CF Axes: * X: ['nx']
           * Y: ['ny']
           * T: ['time']
             Z: n/a

- CF Coordinates:   longitude: ['lon']
                    latitude: ['lat']
                  * time: ['time']
                    vertical: n/a

- Cell Measures:   area, volume: n/a

- Standard Names:   forecast_reference_time: ['time_run']
                    latitude: ['lat']
                    longitude: ['lon']
                  * time: ['time']

- Bounds:   n/a
```

grid point

```
%%time

# Set up a single lon/lat location
j, i = 10, 10
longitude = float(ds.cf['longitude'][j,i])
latitude = float(ds.cf['latitude'][j,i])

# Select-by-index a time index and no vertical index (zeta has none)
# also lon/lat by index
Z = None
iT = 0
isel = dict(T=iT, X=i, Y=j)

da_out = ds.em.interp2d(longitude, latitude, iT=iT, iZ=Z)

# check work
```

(continues on next page)

(continued from previous page)

```
da_check = ds.cf.isel(isel)

assert np.allclose(da_out, da_check)
```

```
CPU times: user 143 ms, sys: 2.23 ms, total: 146 ms
Wall time: 145 ms
```

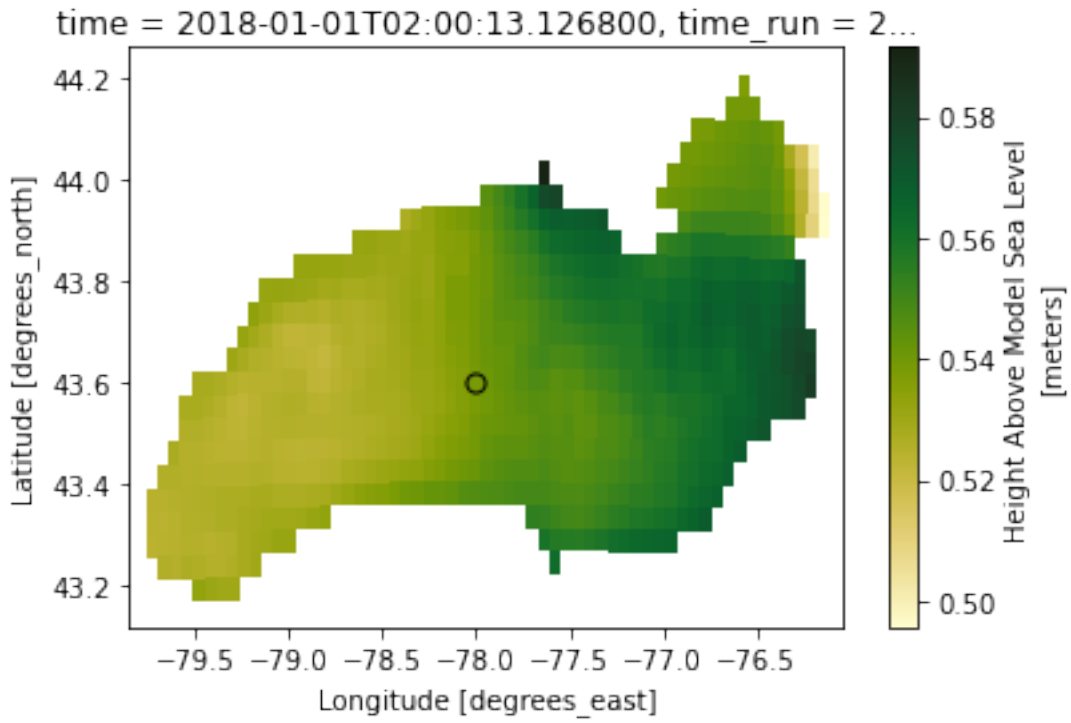
This is faster the second time the regridder is used — it is saved by the `extract_model` accessor and reused if the lon/lat locations to be interpolated to are the same.

not grid point

inside domain

```
# sel
longitude = -78.0
latitude = 43.6

# isel
iZ = None
iT = 1
isel = dict(T=iT)

da_out = ds.em.interp2d(longitude, latitude, iT=iT, iZ=iZ)

# plot
cmap = ds.cmo.seq
dacheck = ds.cf.isel(isel)
fig, ax = plt.subplots(1,1)
dacheck.cmo.cfplot(ax=ax, x='longitude', y='latitude')
ax.scatter(da_out.cf['longitude'], da_out.cf['latitude'], s=50, c=da_out,
           vmin=dacheck.min().values, vmax=dacheck.max().values, cmap=cmap, edgecolors='k
↪')
```

```
<matplotlib.collections.PathCollection at 0x7fbfc2c23a90>
```



points (locstream)

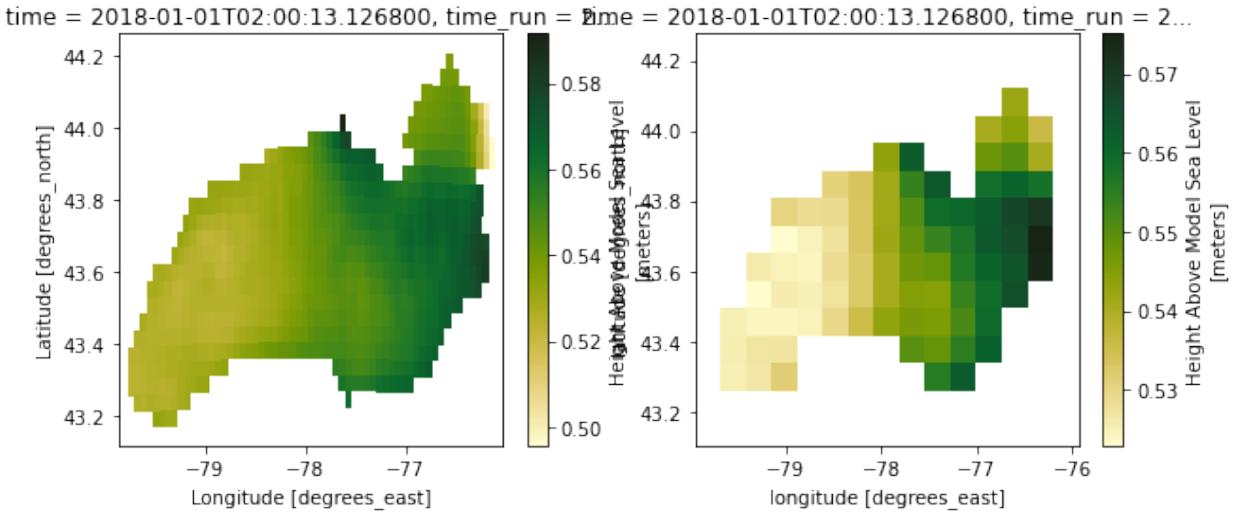
Unstructured pairs of lon/lat locations instead of grids of lon/lat locations, using `locstream`.

```
# sel
# this creates 12 pairs of lon/lat points that
# align with grid points so we can check the
# interpolation
longitude = ds.cf['longitude'].cf.isel(Y=20, X=slice(None, None, 10))
latitude = ds.cf['latitude'].cf.isel(Y=20, X=slice(None, None, 10))
sel = dict(X=longitude.cf['X'], Y=longitude.cf['Y'])

# isel
iZ = None
iT = 0
isel = dict(T=iT)

da_out = ds.em.interp2d(longitude, latitude, iT=iT, iZ=iZ, locstream=True)

# check
da_check = ds.cf.isel(isel).cf.sel(sel)

assert np.allclose(da_out, da_check, equal_nan=True)
```

grid of new locations

```
# sel
longitude = np.linspace(ds.cf['longitude'].min(), ds.cf['longitude'].max(), 15)
latitude = np.linspace(ds.cf['latitude'].min(), ds.cf['latitude'].max(), 15)

# isel
iZ = None
iT = 1
isel = dict(T=iT)

da_out = ds.em.interp2d(longitude, latitude, iT=iT, iZ=iZ, locstream=False, extrap=False,
↪ extrap_val=np.nan)

# plot
cmap = cmo.delta
dacheck = ds.cf.isel(isel)

fig, axes = plt.subplots(1,2, figsize=(10,4))
dacheck.cmo.cfplot(ax=axes[0], x='longitude', y='latitude')
da_out.cmo.cfplot(ax=axes[1], x='longitude', y='latitude')
```

```
<matplotlib.collections.QuadMesh at 0x7fbf82a61cd0>
```



1.3 Subsetting with `extract_model`

```
# Setup custom styling for this jupyter notebook
from IPython.core.display import HTML
from pathlib import Path

def css_styling():
    buf = Path('./css/custom.css').read_text()
    style_header = f'<style>\n{buf}\n</style>\n'
```

(continues on next page)

(continued from previous page)

```
    return HTML(style_header)
css_styling()
```

```
<IPython.core.display.HTML object>
```

```
from matplotlib import pyplot as plt

def figure(*args, figsize=(18, 8), facecolor='white', **kwargs):
    """Return a new figure."""
    return plt.subplots(*args, figsize=figsize, facecolor=facecolor, **kwargs)
```

```
from matplotlib import tri
from matplotlib import patches
import xarray as xr
import netCDF4 as nc4
import numpy as np
import extract_model as em
from extract_model.grids.triangular_mesh import UnstructuredGridSubset
```

```
# A helper function to make plotting variables from unstructured data easier.
def plot_unstructured_variable(ds, varname, triangulation_varname='nv', xvarname='x',␣
↪yvarname='y', vmin=None, vmax=None, facecolor='white', figsize=(24, 8), buf=None,␣
↪bbox=None, fig=None, ax=None):
    """Plot variable from an unstructured grid."""
    x = ds[xvarname][:].to_numpy()
    y = ds[yvarname][:].to_numpy()
    nv = ds[triangulation_varname][:].to_numpy().T - 1
    triang = tri.Triangulation(x, y, nv)
    x_vertices = x[nv]
    xmin = np.min(x_vertices.flatten())
    xmax = np.max(x_vertices.flatten())
    y_vertices = y[nv]
    ymin = np.min(y_vertices.flatten())
    ymax = np.max(y_vertices.flatten())
    C = ds[varname][:]
    if fig is None and ax is None:
        fig, ax = plt.subplots(facecolor=facecolor, figsize=figsize)
    cbar = ax.tripcolor(triang, C)
    fig.colorbar(cbar)
    if buf is None:
        buf = 0.05
    ax.set_xlim([xmin-buf, xmax+buf])
    ax.set_ylim([ymin-buf, ymax+buf])
    ax.triplot(triang, color='gray', linewidth=0.2)
    ax.grid(linewidth=0.5)
    ax.set_title(ds[varname].long_name)
    ax.set_xlabel(ds[xvarname].long_name)
    ax.set_ylabel(ds[yvarname].long_name)
    if bbox is not None:
        rect = patches.Rectangle((bbox[0], bbox[1]), bbox[2] - bbox[0], bbox[3] -␣
↪bbox[1], linewidth=2, edgecolor='r', facecolor='none')
```

(continues on next page)

(continued from previous page)

```
        # Add the patch to the Axes
        ax.add_patch(rect)
    return fig, ax
```

1.3.1 Lake Erie Operational Forecast System (LEOFS) FVCOM

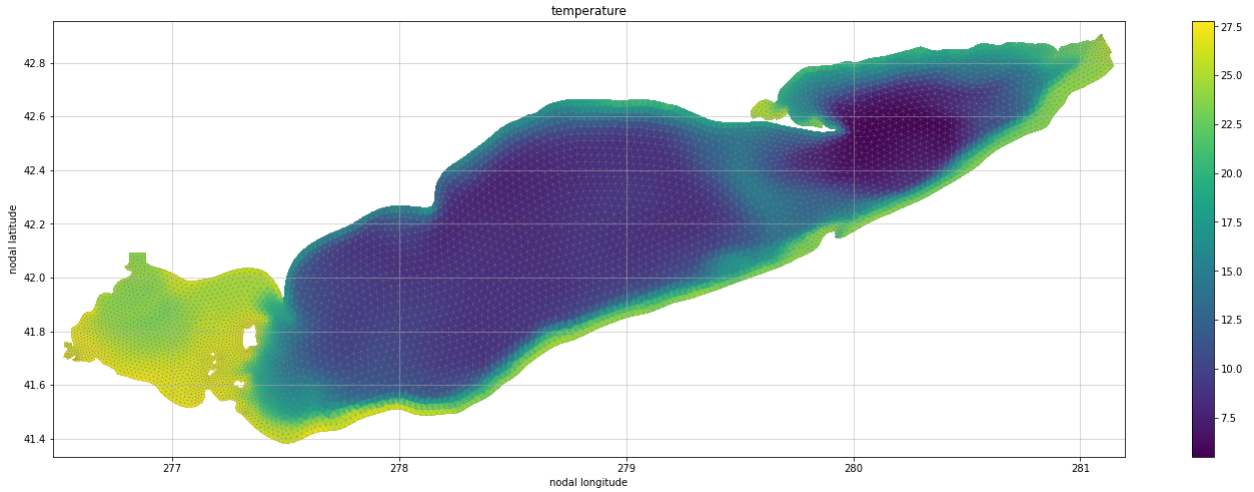
Initialize the dataset object, but use our engine so that we can load FVCOM data into an xarray Dataset.

```
url = 'https://www.ncei.noaa.gov/thredds/dodsC/model-leofs/2022/07/nos.leofs.fields.n006.
↪20220720.t00z.nc'
```

```
# We use a custom engine to support loading unstructured grids into xarray. The
# current version of xarray doesn't support coordinate variables having multiple
# dimensions, and technically it's not allowed according to the NUG and I think
# it's forbidden by CF as well. The custom version engine titled
# triangularmesh_netcdf allows clients to rename variables after the data
# arrays are constructed but before the xarray Dataset is assembled. This
# allows us to load the unstructured model output and rename the z coordinate
# variables siglay and siglev to sigma_layers and sigma_levels respectively.

# We also drop the Itime and Itime2 variables which are redundant to the time
# coordinate variable. These variables also interfere with cf-xarray and
# decoding times.

# MOST importantly, we specify the chunk size to be 1,N along the time
# coordinate. Without specifying a good chunk size, operations will take
# prohibitively long, and by default xarray makes extremely poor choices for
# chunksizes on unstructured grids, or at the very least with FVCOM.

ds = xr.open_dataset(url,
                     engine='triangularmesh_netcdf',
                     decode_times=True,
                     preload_varmap={'siglay': 'sigma_layers', 'siglev': 'sigma_levels'},
                     drop_variables=['Itime', 'Itime2'],
                     chunks={'time':1})
```

```
plot_unstructured_variable(ds.isel(time=0, siglay=-1), 'temp', xvarname='lon', yvarname=
↪'lat', figsize=(24,8))
```

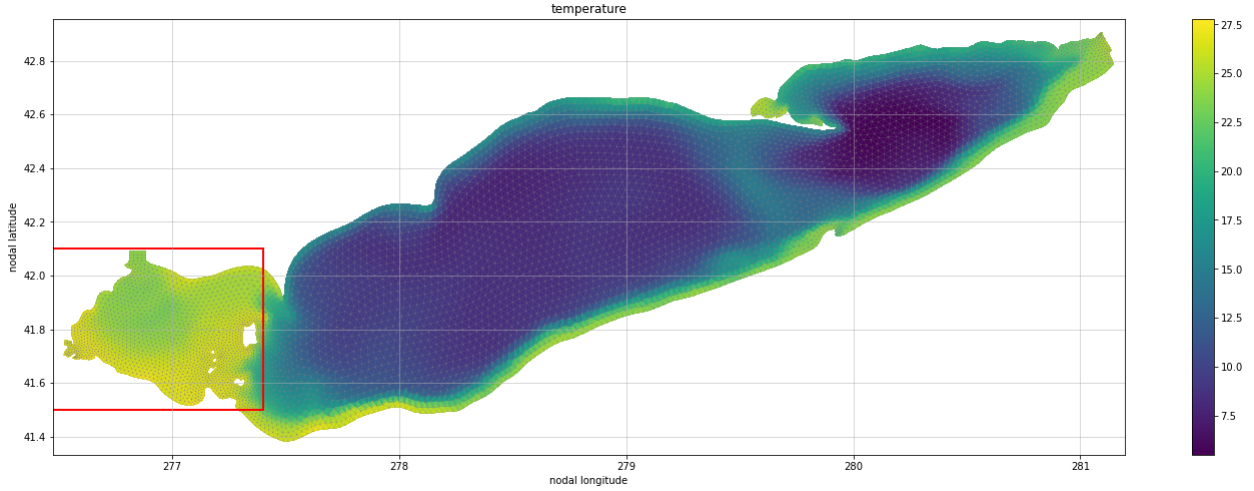
```
(<Figure size 1728x576 with 2 Axes>,
 <AxesSubplot: title={'center': 'temperature'}, xlabel='nodal longitude', ylabel='nodal␣
↪latitude'>)
```



Now consider that we desire to view only the southwestern portion of Lake Erie.

```
bbox = (276.4, 41.5, 277.4, 42.1)
plot_unstructured_variable(ds.isel(time=0, siglay=-1), 'temp', xvarname='lon', yvarname=
↪'lat', bbox=bbox, figsize=(24,8))
```

```
(<Figure size 1728x576 with 2 Axes>,
 <AxesSubplot: title={'center': 'temperature'}, xlabel='nodal longitude', ylabel='nodal␣
↪latitude'>)
```



Subsetting the dataset is simple with the `ds.em.sub_bbox` method.

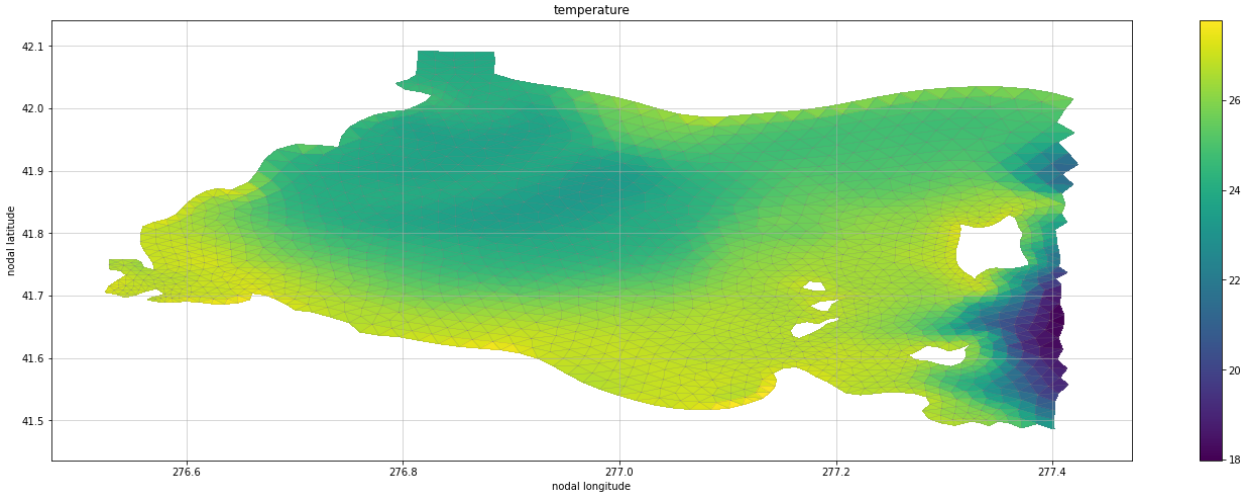
```
%%time
ds_ss = ds.em.sub_bbox(bbox=bbox)
```

```
CPU times: user 2.82 s, sys: 153 ms, total: 2.97 s
Wall time: 3.62 s
```

Plotting the new dataset we can see that only the relevant cells from the model output are available, any cells wholly external to the region of interest are discarded from the subsetted dataset.

```
plot_unstructured_variable(ds_ss.isel(time=0, siglay=-1), 'temp', xvarname='lon',␣
↪yvarname='lat', figsize=(24,8))
```

```
(<Figure size 1728x576 with 2 Axes>,
 <AxesSubplot: title={'center': 'temperature'}, xlabel='nodal longitude', ylabel='nodal␣
↪latitude'>)
```



Now that we have subsetted the data to a region of interest we can write that data to disk.

```
%%time
ds_ss.to_netcdf('/tmp/fvcom-subset.nc')
```

```
CPU times: user 2.28 s, sys: 255 ms, total: 2.53 s
Wall time: 18.7 s
```

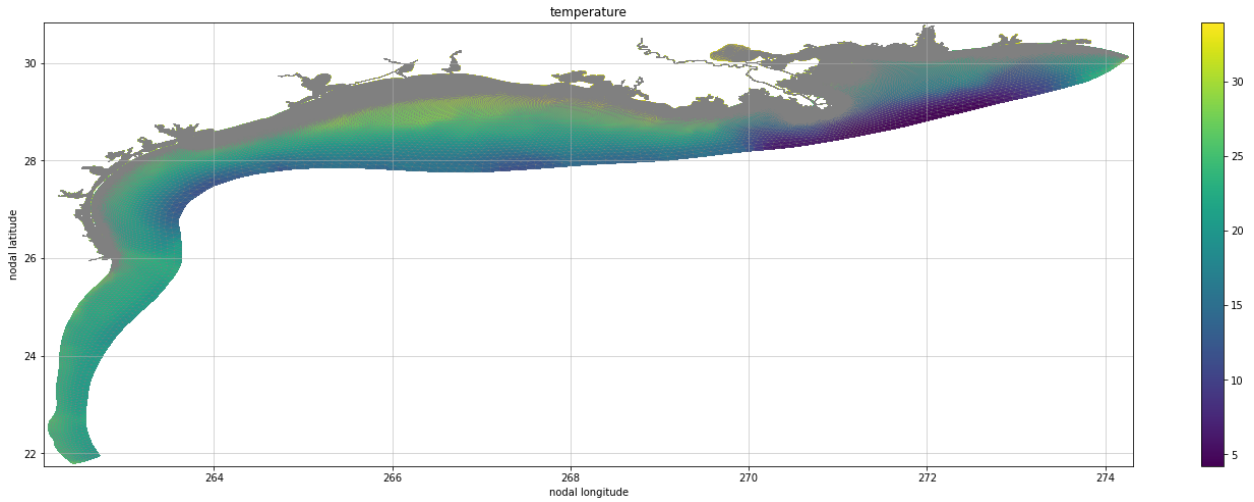
1.3.2 Northern Golf of Mexico Operational Forecast System (NGOFS2) FVCOM

```
url = 'https://www.ncei.noaa.gov/thredds/dodsC/model-ngofs2-files/2022/07/nos.ngofs2.
↪fields.n006.20220725.t03z.nc'
```

```
ds = xr.open_dataset(url, engine='triangularmesh_netcdf', decode_times=True, preload_
↪varmap={'siglay': 'sigma_layers', 'siglev': 'sigma_levels'}, drop_variables=['Itime',
↪'Itime2'], chunks={'time':1})
```

```
plot_unstructured_variable(ds.isel(time=0, siglay=-1), 'temp', xvarname='lon', yvarname=
↪'lat', figsize=(24,8))
```

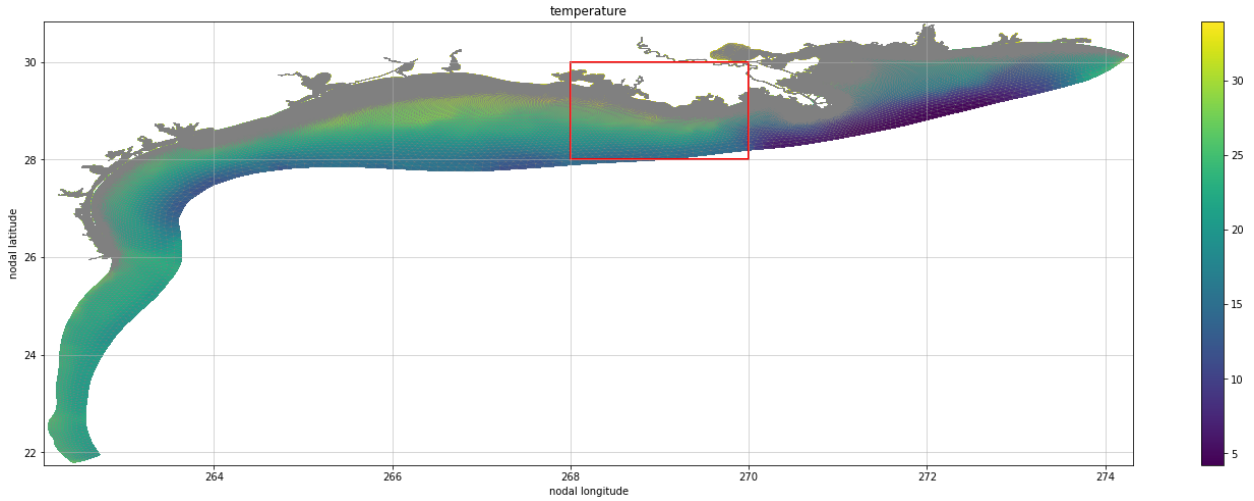
```
(<Figure size 1728x576 with 2 Axes>,
 <AxesSubplot: title={'center': 'temperature'}, xlabel='nodal longitude', ylabel='nodal␣
↪latitude'>)
```



Consider that we want to subset onto a region around southern Louisiana.

```
bbox = (268, 28, 270, 30)

plot_unstructured_variable(ds.isel(time=0, siglay=-1), 'temp', xvarname='lon', yvarname=
↪'lat', bbox=bbox, figsize=(24,8))
```

```
(<Figure size 1728x576 with 2 Axes>,
 <AxesSubplot: title={'center': 'temperature'}, xlabel='nodal longitude', ylabel='nodal␣
↪latitude'>)
```

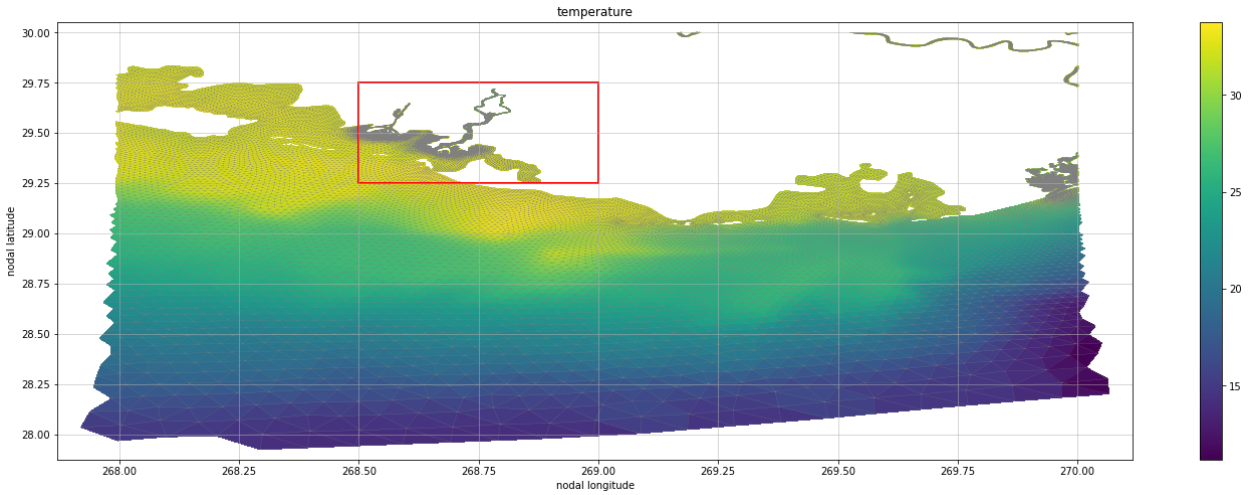



```
%%time
ds_ss = ds.em.sub_bbox(bbox=bbox, model_type='FVCOM')
```

```
CPU times: user 1.77 s, sys: 551 ms, total: 2.32 s
Wall time: 8.9 s
```

```
bbox = (268.5, 29.25, 269, 29.75)
plot_unstructured_variable(ds_ss.isel(time=0, siglay=-1), 'temp', xvarname='lon',␣
↪yvarname='lat', bbox=bbox, figsize=(24,8))
```

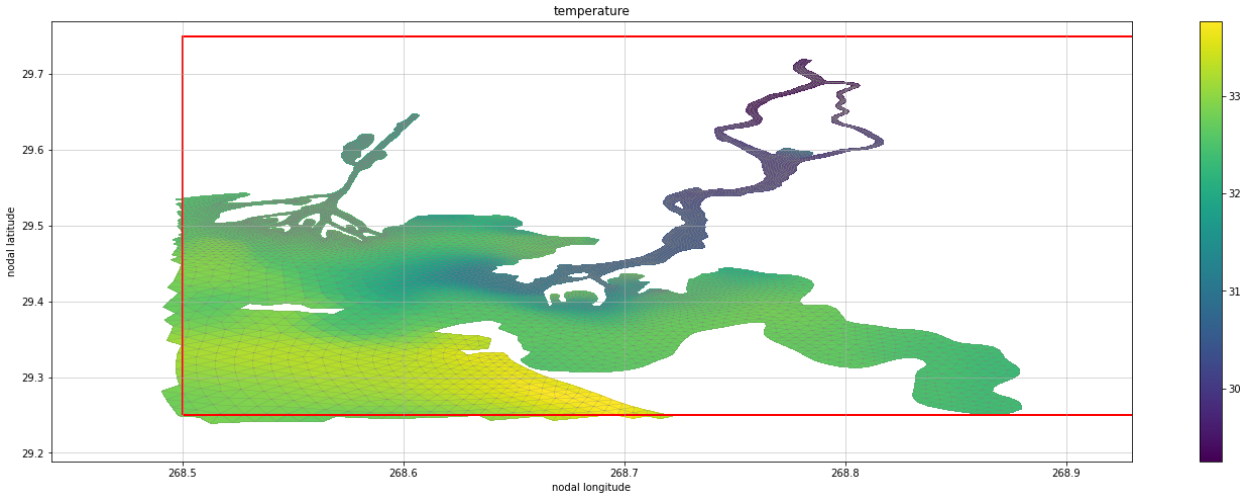
```
(<Figure size 1728x576 with 2 Axes>,
 <AxesSubplot: title={'center': 'temperature'}, xlabel='nodal longitude', ylabel='nodal␣
↪latitude'>)
```




```
ds_ss = ds_ss.em.sub_bbox(bbox=bbox)
```

```
plot_unstructured_variable(ds_ss.isel(time=0, siglay=-1), 'temp', xvarname='lon',␣
↪yvarname='lat', bbox=bbox, figsize=(24,8))
```

```
(<Figure size 1728x576 with 2 Axes>,
 <AxesSubplot: title={'center': 'temperature'}, xlabel='nodal longitude', ylabel='nodal␣
↪latitude'>)
```



1.3.3 Subsetting Columbia River Estuary Operational Forecast System (CREOFS) SELFE

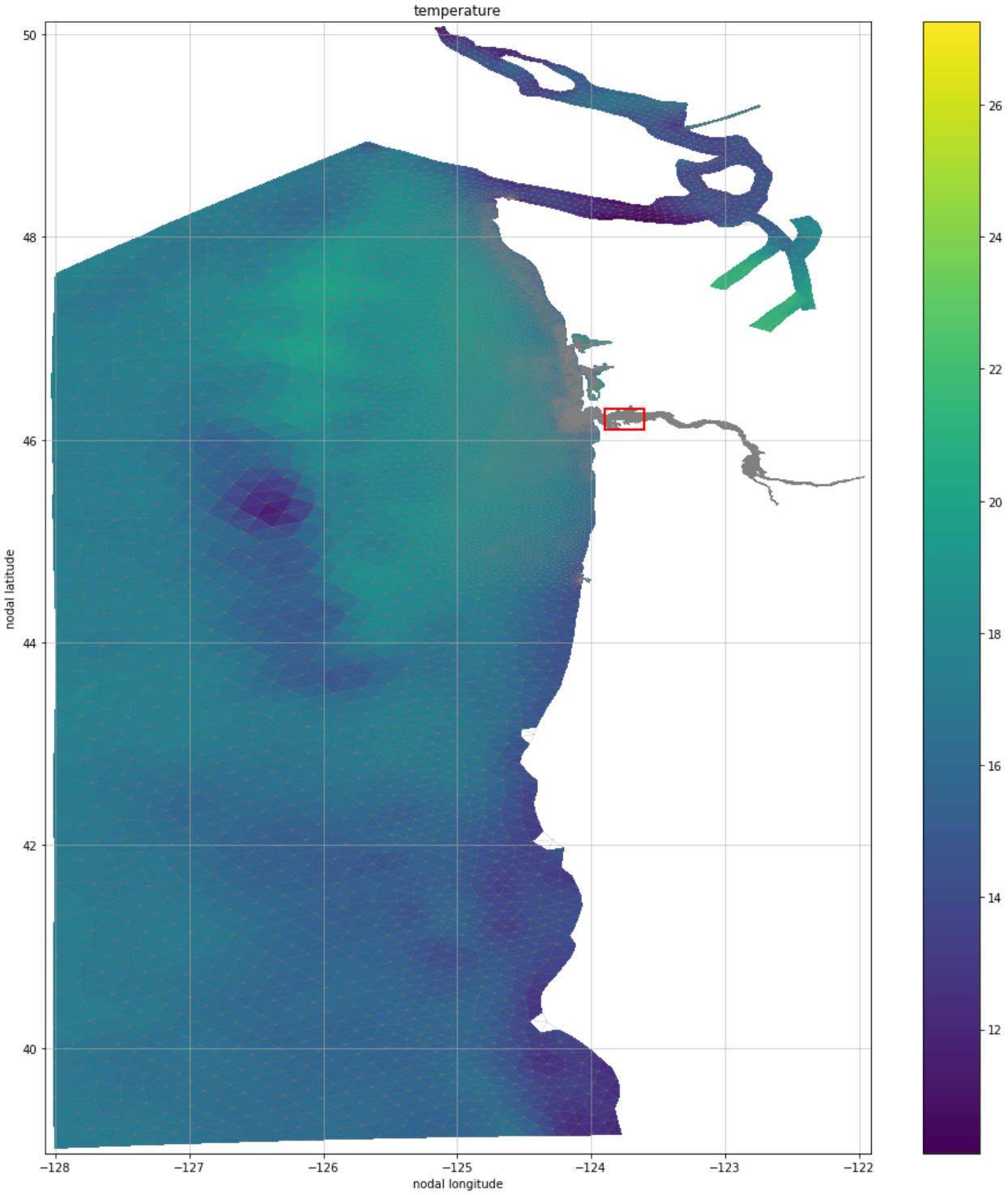
```
url = 'https://www.ncei.noaa.gov/thredds/dodsC/model-creofs-files/2022/07/nos.creofs.
↪fields.n006.20220728.t09z.nc'
```

```
ds = xr.open_dataset(url, chunks={'time': 1})
```

Looking at the western portion of the Columbia River, we find a section of interest.

```
bbox = (-123.9, 46.1, -123.6, 46.3)
plot_unstructured_variable(ds.isel(time=0, nv=-1), 'temp', triangulation_varname='ele',␣
↪xvarname='lon', yvarname='lat', bbox=bbox, figsize=(16,18))
```

```
(<Figure size 1152x1296 with 2 Axes>,
 <AxesSubplot: title={'center': 'temperature'}, xlabel='nodal longitude', ylabel='nodal␣
↪latitude'>)
```



Subsetting is the same as before.

```
%%time
ds_ss = ds.em.sub_grid(bbox)
```

```
CPU times: user 128 ms, sys: 7.54 ms, total: 135 ms
```

(continues on next page)

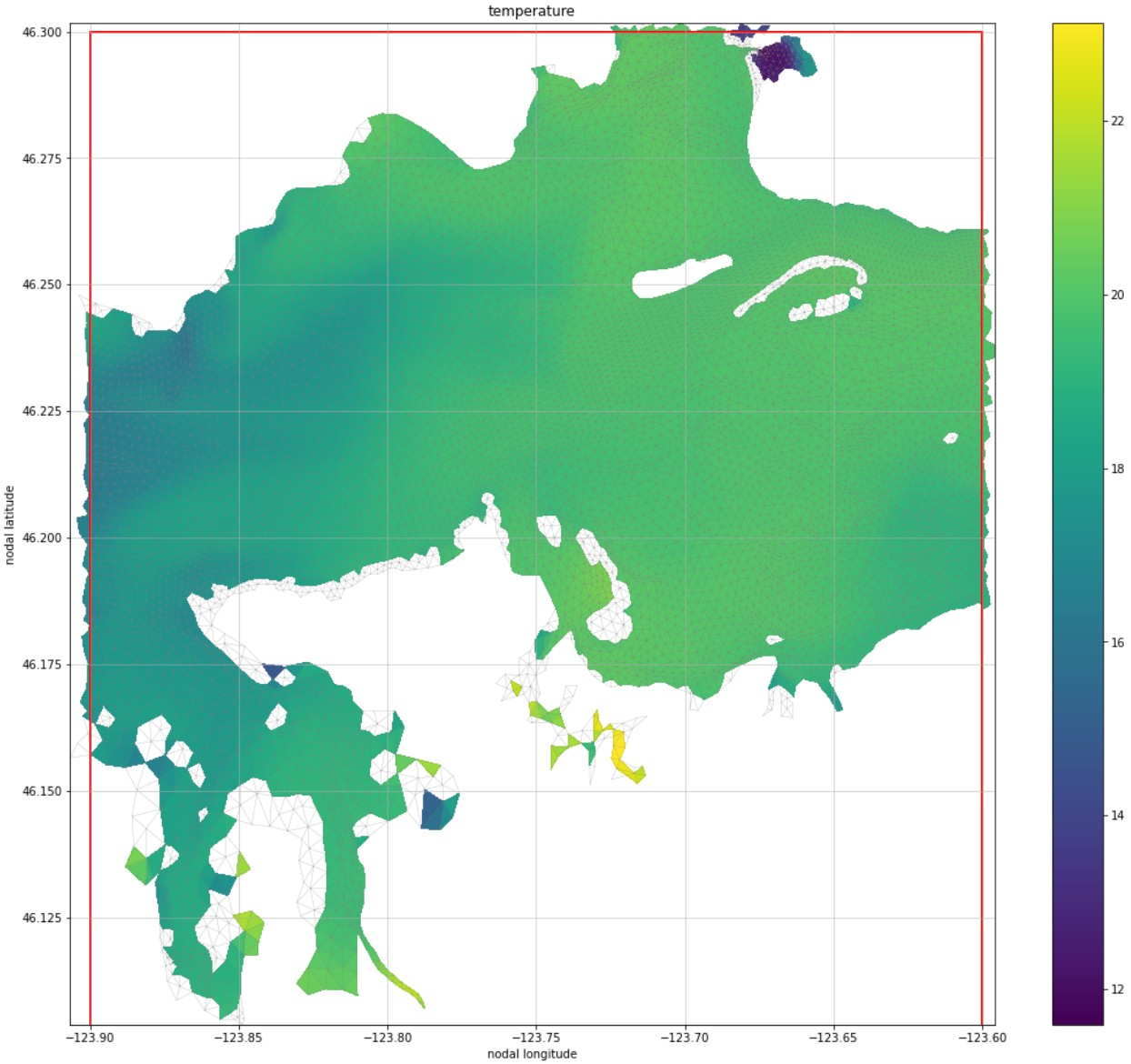
(continued from previous page)

```
Wall time: 137 ms
```

Plotting the subsetted data we see only the region of interest.

```
plot_unstructured_variable(ds_ss.isel(time=0, nv=-1), 'temp', triangulation_varname='ele
↪', xvarname='lon', yvarname='lat', buf=0, bbox=bbox, figsize=(18,16))
```

```
(<Figure size 1296x1152 with 2 Axes>,
 <AxesSubplot: title={'center': 'temperature'}, xlabel='nodal longitude', ylabel='nodal␣
↪latitude'>)
```



1.4 Time Series Extraction

```
import xarray as xr
import cf_xarray
import extract_model as em
import pandas as pd
from glob import glob
import matplotlib.pyplot as plt
import cmocean.cm as cmo

# For this notebook, it's nicer if we don't show the array values by default
xr.set_options(display_expand_data=False)
xr.set_options(display_expand_coords=False)
xr.set_options(display_expand_attrs=False)
```

```
<xarray.core.options.set_options at 0x111e59ac0>
```

1.4.1 Example model to use

```
# !wget https://www.ncei.noaa.gov/thredds/fileServer/model-ciofs-files/2022/03/nos.ciofs.
↪fields.n001.20220301.t12z.nc
# !wget https://www.ncei.noaa.gov/thredds/fileServer/model-ciofs-files/2022/03/nos.ciofs.
↪fields.n001.20220301.t18z.nc
```

```
# Structured: CIOFS: ROMS Cook Inlet model
# get some model output locally
# loc1 = glob('nos.ciofs.*.nc')
# ds1 = xr.open_mfdataset([loc1], drop_variables="ocean_time", preprocess=em.preprocess).
↪sel(time=slice("2022-03-01T07", "2022-03-01T08"))
loc1 = "https://www.ncei.noaa.gov/thredds/dodsC/model-ciofs-agg/Aggregated_CIOFS_Fields_
↪Forecast_best.ncd"
ds1 = xr.open_dataset(loc1, drop_variables="ocean_time")
ds1 = em.preprocess(ds1, kwargs={"interp_vertical": False})
ds1 = ds1.sel(time=slice("2022-03-01T07", "2022-03-01T08"))
ds1

# # Unstructured: CREOFS: SELFE Columbia River model
# today = pd.Timestamp.today()
# loc2 = [today.strftime('https://opendap.co-ops.nos.noaa.gov/thredds/dodsC/NOAA/CREOFS/
↪MODELS/%Y/%m/%d/nos.creofs.fields.n000.%Y%m%d.t03z.nc'),
#         today.strftime('https://opendap.co-ops.nos.noaa.gov/thredds/dodsC/NOAA/CREOFS/
↪MODELS/%Y/%m/%d/nos.creofs.fields.n001.%Y%m%d.t03z.nc')]
```

1.4.2 Demo code

Select time series from nearest point

Use a DataArray or a Dataset, but keep in mind that when there are multiple horizontal grids (like there are for ROMS models), you will need to specify which grid's longitude and latitude coordinates to use. The API is meant to be analogous to that of selecting with `xarray` using `.sel()`.

This functionality uses `xoak`.

```
da1 = ds1['temp']
lon0, lat0 = -151.4, 59  # cook inlet
```

For any of the following results, access the depth values with

```
[output].cf['vertical'].values
```

2D lon/lat

The first request will take longer than a second request would because the second request uses the index calculated the first time.

```
%%time
output = da1.em.sel2d(lon_rho=lon0, lat_rho=lat0).squeeze()
output
```

```
CPU times: user 608 ms, sys: 15.1 ms, total: 623 ms
Wall time: 624 ms
```

```
<xarray.DataArray 'temp' (time: 2, s_rho: 30)>
...
Coordinates: (7)
Attributes: (9)
```

```
%%time
output = da1.em.sel2d(lon_rho=lon0, lat_rho=lat0).squeeze()
output
```

```
CPU times: user 4.02 ms, sys: 1.92 ms, total: 5.95 ms
Wall time: 4.24 ms
```

```
<xarray.DataArray 'temp' (time: 2, s_rho: 30)>
...
Coordinates: (7)
Attributes: (9)
```

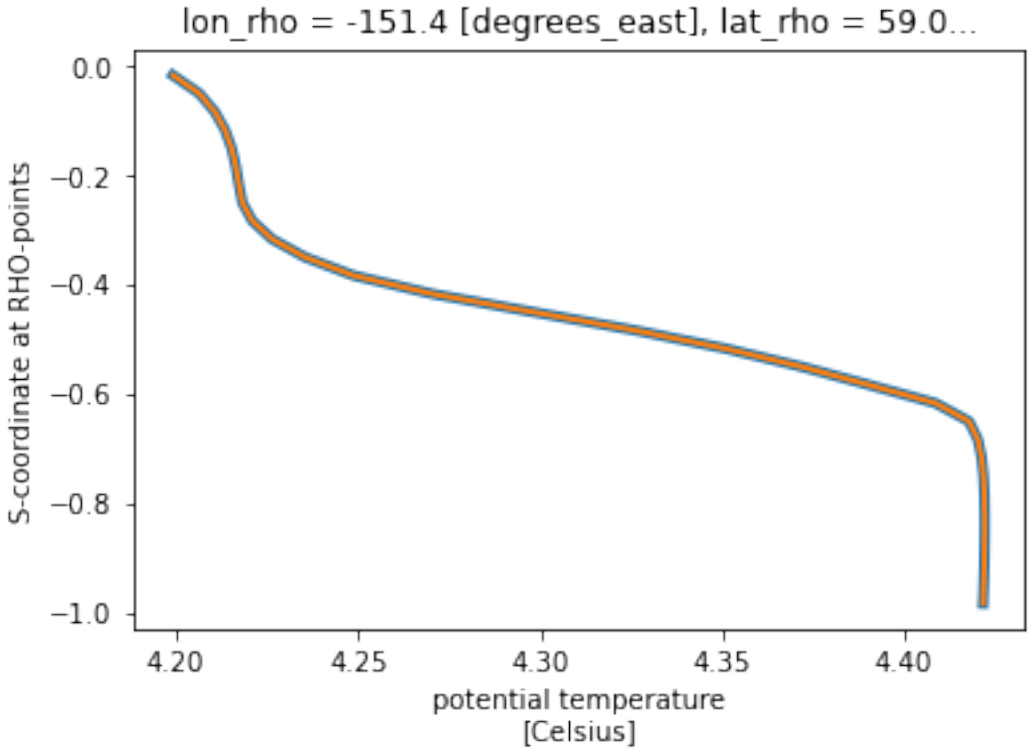
Access the associated indices:

```
j, i = int(output.eta_rho.values), int(output.xi_rho.values)
```

Profile for first time matches:

```
output.cf.isel(T=0).cf.plot(y='vertical', lw=4)
da1.cf.isel(X=i, Y=j, T=0).cf.plot(y='vertical', lw=2)
```

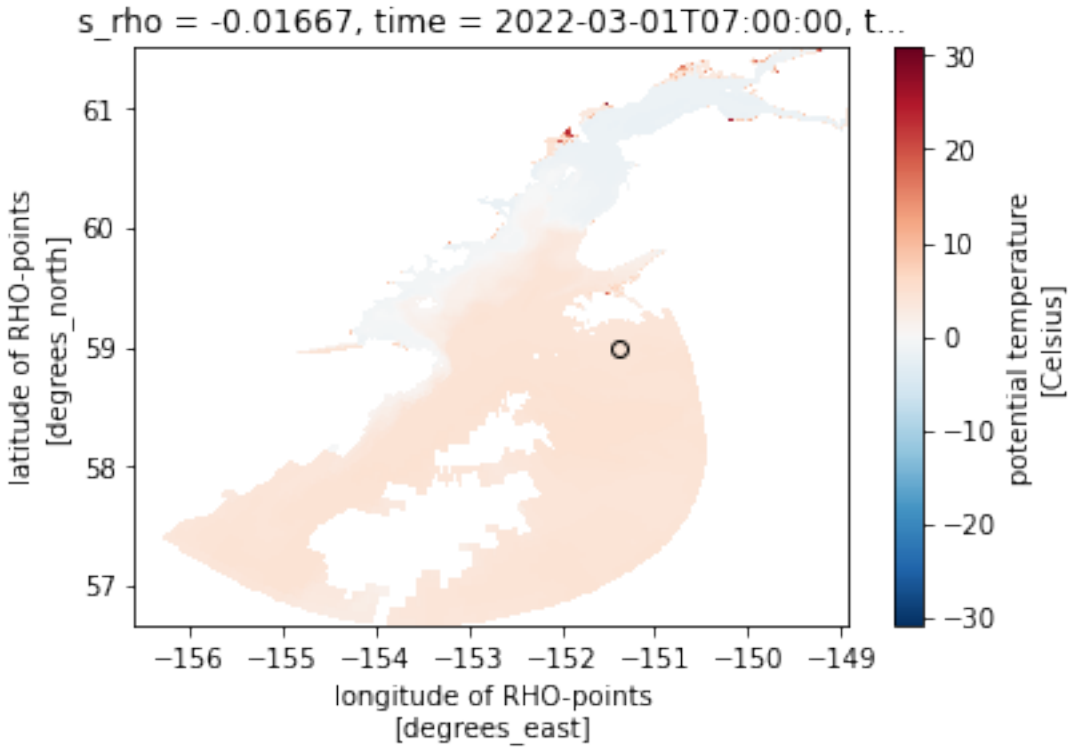
```
[<matplotlib.lines.Line2D at 0x170a7e190>]
```



Surface value for first time matches map:

```
mappable = da1.cf.isel(T=0, Z=-1).cf.plot(x='longitude', y='latitude')
vmin, vmax = mappable.get_clim()
plt.scatter(lon0, lat0, c=output.cf.isel(T=0, Z=-1).values, cmap=mappable.cmap,␣
↪vmin=vmin, vmax=vmax, edgecolors='k')
```

```
<matplotlib.collections.PathCollection at 0x170a5ed00>
```



To retrieve the values:

`output.values`

```
output.values
```

```
array([[4.4214993, 4.421684 , 4.4218173, 4.421909 , 4.4219575, 4.421951 ,
        4.4218597, 4.4216237, 4.421123 , 4.4200873, 4.4176564, 4.408428 ,
        4.3901873, 4.371695 , 4.350696 , 4.325884 , 4.297998 , 4.270254 ,
        4.248803 , 4.2355056, 4.226392 , 4.2209353, 4.218213 , 4.2172427,
        4.216388 , 4.2152452, 4.213435 , 4.2106075, 4.206378 , 4.199108 ],
       [4.434471 , 4.4342895, 4.4340544, 4.4337626, 4.4333944, 4.4329157,
        4.4322686, 4.4313636, 4.4300494, 4.428064 , 4.4249015, 4.4193425,
        4.4073486, 4.377332 , 4.3381915, 4.306614 , 4.281716 , 4.261284 ,
        4.2448688, 4.2351127, 4.2280526, 4.222328 , 4.2180495, 4.216049 ,
        4.214684 , 4.2133946, 4.2117043, 4.209222 , 4.2054753, 4.1987357]],
      dtype=float32)
```

To retrieve the associated depths:

`output.cf['vertical'].values`

```
output.cf['vertical'].values
```

```
array([-0.98333333, -0.95      , -0.91666667, -0.88333333, -0.85      ,
       -0.81666667, -0.78333333, -0.75      , -0.71666667, -0.68333333,
       -0.65      , -0.61666667, -0.58333333, -0.55      , -0.51666667,
       -0.48333333, -0.45      , -0.41666667, -0.38333333, -0.35      ,
       -0.31666667, -0.28333333, -0.25      , -0.21666667, -0.18333333,
```

(continues on next page)

(continued from previous page)

```
       -0.15      , -0.11666667, -0.08333333, -0.05      , -0.01666667])
```

3D lon/lat/Z or iZ

Return model output nearest to lon, lat, Z value. `z_rho` has two values because the depth changes in time.

```
out = da1.em.sel2d(lon_rho=lon0, lat_rho=lat0).squeeze()
out
```

```
<xarray.DataArray 'temp' (time: 2, s_rho: 30)>
...
Coordinates: (7)
Attributes: (9)
```

```
out.em.selZ(depths=-40)
```

```
<xarray.DataArray 'temp' (time: 2)>
4.421 4.434
Coordinates: (7)
Attributes: (9)
```

Return model output nearest to lon, lat, at index iZ in Z dimension.

```
da1.em.sel2d(lon_rho=lon0, lat_rho=lat0).cf.isel(Z=-1)
```

```
<xarray.DataArray 'temp' (time: 2, loc: 1)>
...
Coordinates: (7)
Dimensions without coordinates: loc
Attributes: (9)
```

Interpolate time series at exact point

```
da1 = ds1['salt']
lon0, lat0 = -152, 58
lons, lats = [-151, -152], [59,58]
```

2D lon/lat

1 lon/lat pair

```
%%time
output = da1.em.interp2d(lon0, lat0)
output
```

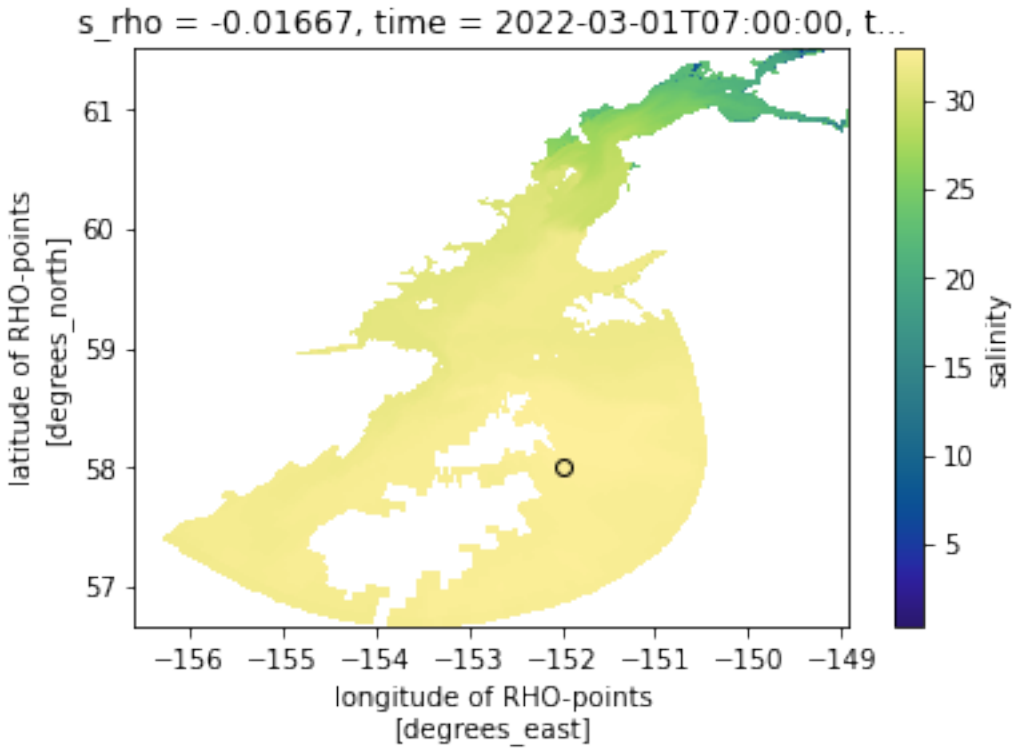
```
CPU times: user 8.95 s, sys: 2.08 s, total: 11 s
Wall time: 30.1 s
```

```
<xarray.DataArray 'salt' (time: 2, s_rho: 30)>
32.94 32.94 32.94 32.94 32.94 32.94 ... 32.87 32.87 32.87 32.87 32.87 32.87
Coordinates: (5)
Attributes: (8)
```

Surface value for first time matches map:

```
cmap=cmo.haline
mappable = da1.cf.isel(T=0, Z=-1).cf.plot(x='longitude', y='latitude', cmap=cmap)
vmin, vmax = mappable.get_clim()
plt.scatter(lon0, lat0, c=output.cf.isel(T=0, Z=-1).values, cmap=cmap, vmin=vmin,␣
↪vmax=vmax, edgecolors='k')
```

```
<matplotlib.collections.PathCollection at 0x170c1bac0>
```



To retrieve the values:

`output.values`

```
output.values
```

```
array([[32.93971 , 32.939648, 32.939594, 32.93954 , 32.939472, 32.939392,
        32.939293, 32.93917 , 32.93898 , 32.93863 , 32.937885, 32.936424,
        32.933247, 32.92608 , 32.918026, 32.910328, 32.903812, 32.898037,
        32.892227, 32.88485 , 32.877373, 32.875435, 32.875843, 32.87596 ,
        32.875977, 32.875988, 32.875996, 32.875996, 32.87599 , 32.875977],
       [32.938396, 32.93832 , 32.938255, 32.93819 , 32.938118, 32.93803 ,
        32.93791 , 32.93775 , 32.937523, 32.937138, 32.9364  , 32.934933,
```

(continues on next page)

(continued from previous page)

```
        32.931953, 32.92531 , 32.917496, 32.90954 , 32.90272 , 32.896683,
        32.89059 , 32.882954, 32.8751  , 32.87288 , 32.87316 , 32.873245,
        32.873253, 32.873257, 32.873257, 32.873253, 32.873245, 32.873226]],
      dtype=float32)
```

To retrieve the associated depths:

`output.cf['vertical'].values`

```
output.cf['vertical'].values
```

```
array([-0.98333333, -0.95      , -0.91666667, -0.88333333, -0.85      ,
       -0.81666667, -0.78333333, -0.75      , -0.71666667, -0.68333333,
       -0.65      , -0.61666667, -0.58333333, -0.55      , -0.51666667,
       -0.48333333, -0.45      , -0.41666667, -0.38333333, -0.35      ,
       -0.31666667, -0.28333333, -0.25      , -0.21666667, -0.18333333,
       -0.15      , -0.11666667, -0.08333333, -0.05      , -0.01666667])
```

multiple lon/lat pairs

```
%%time
da1.em.interp2d(lons, lats)
```

```
CPU times: user 8.4 s, sys: 1.72 s, total: 10.1 s
Wall time: 27.6 s
```

```
<xarray.DataArray 'salt' (time: 2, s_rho: 30, lat: 2, lon: 2)>
32.49 32.05 32.95 32.94 32.49 32.05 ... 32.78 32.87 31.97 32.03 32.78 32.87
Coordinates: (5)
Attributes: (8)
```

3D: lon, lat, iZ

Return model output interpolated to lon, lat, Z value.

```
da1.em.interp2d(lon0, lat0, Z=-40)
```

```
<xarray.DataArray 'salt' (time: 2)>
nan nan
Coordinates: (5)
Attributes: (8)
```

Return model output interpolated to lon, lat, at index iZ in Z dimension.

```
da1.em.interp2d(lon0, lat0, iZ=-1)
```

```
<xarray.DataArray 'salt' (time: 2)>
32.88 32.87
Coordinates: (5)
Attributes: (8)
```

Note that it is not currently possible to interpolate in depth when there are both multiple times and locations.

If uncommented, the following cell will return:

> NotImplementedError: Currently it is not possible to interpolate in depth with more than 1 other (time) dimension.

```
# da1.em.interp2d(lons, lats, Z=-40)
```

1.5 API

extract_model	Main file for this code.
utils	Utilities to help extract_model work better.
accessor	This is an accessor to xarray.
sorting	Solutions for sorting functions.
model_type	Class definition for the ModelTypes enum class.
grids.triangular_mesh	Algorithms and utilties for triangular meshes.
xr.triangular_mesh_netcdf	Backend which explicitly supports data defining a triangular mesh.

1.5.1 extract_model.extract_model

Main file for this code. The main code is in *select*, and the rest is to help with variable name management.

Functions

interp_multi_dim(da[, da_out, ...])	Interpolate input DataArray to output DataArray using xESMF.
make_output_ds(longitude, latitude[, locstream])	Given desired interpolated longitude and latitude, return points as Dataset.
sel2d(var[, mask, use_xoak, return_info])	Find the value of the var at closest location to inputs, optionally respecting mask.
sel2dcf(var[, mask, return_info])	Find nearest value(s) on 2D horizontal grid using cf-xarray names.
selZ(var, depths)	Select nearest point in depth.
select(da[, longitude, latitude, T, Z, iT, ...])	Extract output from da at location(s).

1.5.2 extract_model.utils

Utilities to help extract_model work better.

Functions

calc_barycentric(x, y, xs, ys)	Calculate barycentric weights for npts
filter(ds, standard_names[, ...])	Filter Dataset by variables
guess_model_type(ds)	Returns a guess as to which model produced the dataset.
interp_with_barycentric(da, ixs, iys, lam)	
naive_subbox(ds, bbox[, dask_array_chunks])	Perform subsetting directly using dimension slicing.
order(da)	Reorder var to typical dimensional ordering.
sub_bbox(da, bbox[, other, drop, ...])	Subset DataArray in space.
sub_grid(ds, bbox[, dask_array_chunks, ...])	Subset Dataset grids.
tree_query(lon_coords, lat_coords, ...[, k])	Set up and query BallTree for k nearest points

1.5.3 extract_model.accessor

This is an accessor to xarray. It is basically a convenient way to use the extract_model functions, and has bookkeeping in the background where possible. No new functions are available only here; this connects to functions in other files.

Classes

emDataArrayAccessor(da)	DataArray accessor.
emDatasetAccessor(ds)	Dataset accessor.

1.5.4 extract_model.sorting

Solutions for sorting functions.

Functions

index_of_sorted(haystack, needle)	Return an array of indexes for each value in values found in haystack.

1.5.5 extract_model.model_type

Class definition for the ModelTypes enum class.

Classes

ModelType(value)	Supported Models.

1.5.6 extract_model.grids.triangular_mesh

Algorithms and utilties for triangular meshes.

Classes

`UnstructuredGridSubset`()	A class for subsetting unstructured grids.

1.5.7 extract_model.xr.triangular_mesh_netcdf

Backend which explicitly supports data defining a triangular mesh.

Classes

`TriangularMeshNetCDF4BackendEntrypoint`()	A custom Backend for xarray to support loading unstructured grids with a triangular mesh.
`TriangularMeshNetCDF4StoreEntrypoint`()	A custom StoreBackendEntrypoint that enables clients to rename variables.

1.6 What's New

1.6.1 v1.4.0 (November 6, 2023)

- small changes so that using xESMF as horizontal interpolator works

1.6.2 v1.3.0 (October 11, 2023)

- Incorporated *locstreamT* and *locstreamZ* to standardize the options available to user in *em.select()*.
- added more tests by featuretype to cover more area

1.6.3 v1.2.2 (October 5, 2023)

- Improved the processing of ROMS model output

1.6.4 v1.2.1 (September 22, 2023)

- ROMS preprocessing checks for 3D instead of just 4D data variables now to update their coordinates to work better with cf-xarray. Also coordinates could say "x_rho" and "y_rho" instead of longitudes and latitudes in which case they are changed to say the coordinates

1.6.5 v1.2.0 (September 13, 2023)

- Improvements to interpolation

1.6.6 v1.1.4 (January 27, 2023)

- fixed docs to run fully
- had to switch back to pre-compiled notebooks to get docs to fully work

1.6.7 v1.1.3 (January 23, 2023)

- Updated the docs so that the notebooks are compiled with *myst-nb*.
- Updated pull_request_template.md.
- some changes to get release to work correctly (unlisted versions)

1.6.8 v1.1.0 (January 19, 2023)

Two main changes in *sel2d* / *sel2dcf*:

- a mask can be input to limit the lons/lats from the DataArray/Dataset that are used in searching for the nearest point, in case the nearest model point is on land but we still want a valid model point returned.
- incorporating changes from xoak that optional return distance of the model point(s) from the requested point(s).

1.6.9 v1.0.0 (December 9, 2022)

- Simplified dependencies
- Now available on PyPI!

1.6.10 v0.9.0 (September 26, 2022)

- An optional subsetting option that enables subsetting directly on the target dataset's dimensions. For remote datasets, this ensures that remote requests ask for minimal slices. *em.sub_grid(..., naive=True)*.
- Adds *preload* argument for unstructured grid subsetting. Radically improves xarray resolution times after subsetting.

1.6.11 v0.8.1 (August 16, 2022)

- Support for SELFE datasets is now incorporated into *em.sub_grid()* *em.sub_bbox()* and *em.filter()*.
- Support for numba and numpy implementations of *index_of_sorted()*.

1.6.12 v0.8.0 (August 3, 2022)

- *extract_model* has a backend that will support reading in FVCOM model output which has previously not been possible when using *xarray* without dropping the vertical grid coordinates.
- *em.sub_bbox()* supports subsetting FVCOM model output.
- A new jupyter notebook demonstrating subsetting of FVCOM model output is now available in docs.
- *em.sub_grid()* supports subsetting FVCOM model output.
- *em.filter()* will not discard any unstructured coordinate information in the auxiliary coordinate variables.

1.6.13 v0.7 (July 22, 2022)

- *em.sel2d()* now uses *xoak* to find the nearest neighbor grid point on ND grids. Due to this change, *em.argsel2d()* doesn't exist anymore. Note that vertical functionality that was previously in *em.sel2d()* is now in *em.selZ()*.
- Provide more options in *em.filter()* for keeping different coordinates in a *Dataset*.
- Improvement to unit test setup.
- *em.preprocess()* will implicitly assign horizontal coordinates longitude and latitude for POM datasets, even if the data do not specify *coordinates* attributes explicitly.

PYTHON MODULE INDEX

e

INDEX

E

M