Easier, Better, Faster, Shorter: Updates to grid-aware analysis with xgcm

Dianne Deauna | University of Hawai‘i at Mānoa | SIParCS intern | Project 8
Mentors: Anderson Banihirwe (NCAR), Julius Busecke (Columbia University), and Deepak Cherian (CGD)
I would like to acknowledge the ‘āina on which I am coming from you today, from the ‘ili ‘āina of Kau'wala'a, the ahupua'a of Mānoa, in the moku of Kona, on the mokupuni of O'ahu, in the pae‘āina of Hawai‘i. I recognize that her majesty Queen Lili‘uokalani yielded the Hawaiian Kingdom and these territories under duress and protest to the United States to avoid the bloodshed of her people, and that Hawai‘i remains an illegally occupied state of America. I further recognize that generations of Indigenous Hawaiians and their knowledge systems shaped Hawai‘i in sustainable ways that allow me to enjoy these gifts today. For this I am grateful as a guest, and I seek to support the varied strategies that the Indigenous peoples of Hawai‘i are using to protect their land and communities.

Adapted from: http://manoa.hawaii.edu/nhpol/language-option/pathways/auamo/
Ocean modeling is mostly representing the ocean as a lot of rectangular cubes.

- Allows for efficient integration of PDEs forwards through time
- Scalar quantity: temperature, Location: center
- Calculate average temperature along the x-axis: need distance from the center to the cell faces
Assigning velocity values to shifted locations within a grid cell makes calculations numerically efficient.

Vector quantity: u-velocity
Location: “eastern” face (shifted to the right relative to temperature)

Calculate average u-velocity along the x-axis: need distance from the cell face to the centers

Postprocessing ocean models require tools that can keep track of these distances for grid-aware operations.
In addition to distances, postprocessing tools also need to keep track of complex cell geometries.

Consider: Area along X,Y axis

Temperature and u-velocity areas are shifted in position and not necessarily equal to each other.
- Makes working with n-dimensional arrays (often provided as netCDF files) more efficient
- Labels raw arrays with dimensions, coordinates, and attributes

**xarray**

- General Circulation Model postprocessing with xarray
- Has sophisticated metric handling for staggered grid datasets
- Has built-in grid-aware operations such as average, integrate, etc.

**GCM**

*Python package*
**metrics**

*definition*

- Information about grid cell geometry in physical space
- Includes:
  - Distance along ‘X’, ‘Y’, or ‘Z’ axis,
  - Areas along (‘X’,’Y’), (‘Y’,’Z’), and (‘X’,’Z’),
  - Volume along (‘X’,’Y’,’Z’)
- Usually not explicitly defined in model outputs for all variables at all positions → there is a need for interpolation
Updated xgcm’s metric handling with three new methods

- **set_metrics()**
  - Enables overwriting of previously assigned metrics and allows for assigning multiple ones on the same axis but with different dimensions.

- **interp_like()**
  - Allows for the interpolation of a data array to the positions of another data array.

- **get_metric()**
  - Selects for the required metric for a data variable along a specified axis for grid-aware operations.
  - Incorporates *interp_like()* to allow for the automatic interpolation of missing metrics from available metric values on surrounding positions.
Interactive Jupyter notebook: bit.ly/xgcm_demo_siparcs2021
## Load data from an Earth System Model

**ds_subset**  
*xarray.Dataset*

- **Dimensions:**  
  
  *(lev: 75, time: 1980, x: 20, x_c: 20, y: 38)*  

- **Coordinates:**  
  
<table>
<thead>
<tr>
<th>Variable</th>
<th>Shape</th>
<th>Dtype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lat</td>
<td>(y, x)</td>
<td>float64</td>
<td>dask.array&lt;chunksize=(38, 20), meta=np.ndarray</td>
</tr>
<tr>
<td>lev</td>
<td>(lev)</td>
<td>float64</td>
<td>0.5058 1.556 ... 5.902e+03</td>
</tr>
<tr>
<td>lon</td>
<td>(y, x)</td>
<td>float64</td>
<td>dask.array&lt;chunksize=(38, 20), meta=np.ndarray</td>
</tr>
<tr>
<td>time</td>
<td>(time)</td>
<td>object</td>
<td>1850-01-16 12:00:00 ... 2014-12-...</td>
</tr>
<tr>
<td>areacello</td>
<td>(y, x)</td>
<td>float32</td>
<td>dask.array&lt;chunksize=(38, 20), meta=np.ndarray</td>
</tr>
<tr>
<td>lat_u</td>
<td>(y, x_c)</td>
<td>float64</td>
<td>dask.array&lt;chunksize=(38, 20), meta=np.ndarray</td>
</tr>
<tr>
<td>lon_u</td>
<td>(y, x_c)</td>
<td>float64</td>
<td>dask.array&lt;chunksize=(38, 20), meta=np.ndarray</td>
</tr>
</tbody>
</table>

- **Data variables:**  
  
<table>
<thead>
<tr>
<th>Variable</th>
<th>Shape</th>
<th>Dtype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thetao</td>
<td>(time, lev, y, x)</td>
<td>float32</td>
<td>dask.array&lt;chunksize=(4, 75, 38, 20), meta=np.ndarray</td>
</tr>
<tr>
<td>uo</td>
<td>(time, lev, y, x_c)</td>
<td>float32</td>
<td>dask.array&lt;chunksize=(3, 75, 38, 20), meta=np.ndarray</td>
</tr>
</tbody>
</table>

Interactive Jupyter notebook:  
Create a grid object using xgcm which contains all information

```python
from xgcm import Grid
grid = Grid(
    ds_subset,
    coords={
        'X': {'center': 'x', 'right': 'x_c'},
        'Y': {'center': 'y', 'right': 'y_c'},
        'Z': {'center': 'lev'},
    },
    periodic=False,
    boundary='extend',
    metrics={('X', 'Y'): 'areacello'}
)
grid.metrics
```

{frozenset({'X', 'Y'}): <xarray.DataArray 'areacello' (y: 38, x: 20)>
  dask.array<getitem, shape=(38, 20), dtype=float32, chunksize=(38, 20), chunktype=numpy.ndarray>
  Dimensions without coordinates: y, x
  Attributes:
  cell_methods: area: sum
description: Cell areas for any grid used to report ocean variables...
history: none
long_name: Grid-Cell Area
online_operation: once
standard_name: cell_area
units: m2}
Calculating area-weighted temperature is straightforward...

```python
mean_sst = grid.average(sst,['X','Y'])
mean_sst.plot()
```

```
[<matplotlib.lines.Line2D at 0x7f507e949460>]
```

![Graph showing time series data with labels](image)
...but not for area-weighted u-velocity (old version of xgcm)

```python
import xgcm
xgcm.__version__

'0.5.1'

mean_uo = grid.average(uo,['X','Y'])
mean_uo.plot()
```

```
KeyError: "Unable to find any combinations of metrics for array dims {'x_c', 'time', 'y'} and axes ['X', 'Y']"
```
Old way = lengthy code :(

```python
from xgcm import Grid

# Step 1: Create a grid object with the available metric
grid = Grid(
    ds_subset,
    coords={
        'X': {'center': 'x', 'right': 'x_c'},
        'Y': {'center': 'y', 'right': 'y_c'},
        'Z': {'center': 'lev'},
    },
    periodic=False,
    boundary='extend',
    metrics=[('X', 'Y'): 'areacello']
)

# Step 2: Interpolate the available metric to the desired variable
areacello_uo = grid.interp(ds_subset, areacello, ('X',))

# Step 3: Create a new grid object
grid_demo = Grid(
    ds_subset,
    coords={
        'X': {'center': 'x', 'right': 'x_c'},
        'Y': {'center': 'y', 'right': 'y_c'},
        'Z': {'center': 'lev'},
    },
    periodic=False,
    boundary='extend',
    metrics=[('X', 'Y'): 'areacello_uo']
)

# Step 4: Calculate the average and plot the time series
mean_uo_demo = grid_demo.average(uo, ['X', 'Y'])
mean_uo_demo.plot()
```

Interactive Jupyter notebook: bit.ly/xgcm_demo_siparcs2021
New way = easier, better, faster, and shorter!

```python
import xgcm
xgcm.__version__

'0.5.2.dev73+g6df944b'

mean_uo = grid.average(uo,['X','Y'])
mean_uo.plot()
```

Interactive Jupyter notebook: bit.ly/xgcm_demo_siparcs2021

(grid.average() calls get_metric() to find an existing metric [areacello] then automatically interpolates the missing one [areacello_uo] using interp_like())
set_metrics lets you assign values to grid objects

```python
# Step 1: Assign `areacello_uo` as a coordinate of subset so that you can assign it as a metric.
subset = ds_subset.assign_coords(areacello_uo=areacello_uo.reset_coords(drop=True).fillna(0))
# fill missing values with 0

# Step 2: Create an updated grid object
grid_updated = Grid(subset,
    coords={
        'X': {'center': 'x', 'right': 'x_c'},
        'Y': {'center': 'y', 'right': 'y_c'},
        'Z': {'center': 'z'},
    },
    periodic=False,
    boundary='extend',
)

# Step 3a: Assign `areacello_uo` as a metric.
grid_updated.set_metrics(('X', 'Y', 'areacello_uo'))

# Step 3b: Take note that with set_metrics you can assign multiple metrics on the same axes to your dataset as long as they have different dimensions.
grid_updated.set_metrics(('X', 'Y', 'areacello_uo'))

# Step 4: Double check if the metrics were assigned
grid_updated.metrics
```

Note: `set_metrics` gives you flexibility when assigning metrics, but it’s not required to use `grid.average()`
New updated xgcm = easier, better, faster, and shorter!

```python
import xgcm
xgcm.__version__

'0.5.2.dev73+g6df944b'

mean_uo = grid.average(uo,['X','Y'])
mean_uo.plot()

/srv/conda/envs/notebook/lib/python3.8/site-packages/xgcm/grid.py:1363: UserWarning: Metric at ('time', 'y', 'x_c') being interpolated from metrics at dimensions ('y', 'x'). Boundary value set to 'extend',
warnings.warn(
[<matplotlib.lines.Line2D at 0x7f36a3d26fa0>]
```

![Updated XGCM](image)

(heavy breathing)
Maráming salámat pô!
Interactive Jupyter notebook: bit.ly/xgcm_demo_siparcs2021
grid.average now uses two methods “under the hood”: interp_like and get_metric which can interpolate metrics.

**interp_like() inputs:**
- available metric
- variable you need the metric for

```python
areacello_uo = grid.interp_like(ds_subset.areacello, ds_subset.uo)
areacello_uo_getmetric = grid.get_metric(ds_subset.uo, ("x", "y"))
```

**get_metric() inputs:**
- variable you need the metric for
- axes for interpolation

```
/srv/conda/envs/notebook/lib/python3.8/site-packages/xgcm/grid.py:1363: UserWarning: Metric at ('time', 'lev', 'y', 'x_c') being interpolated from metrics at dimensions ('y', 'x'). Boundary value set to 'extend'.
warnings.warn(
```