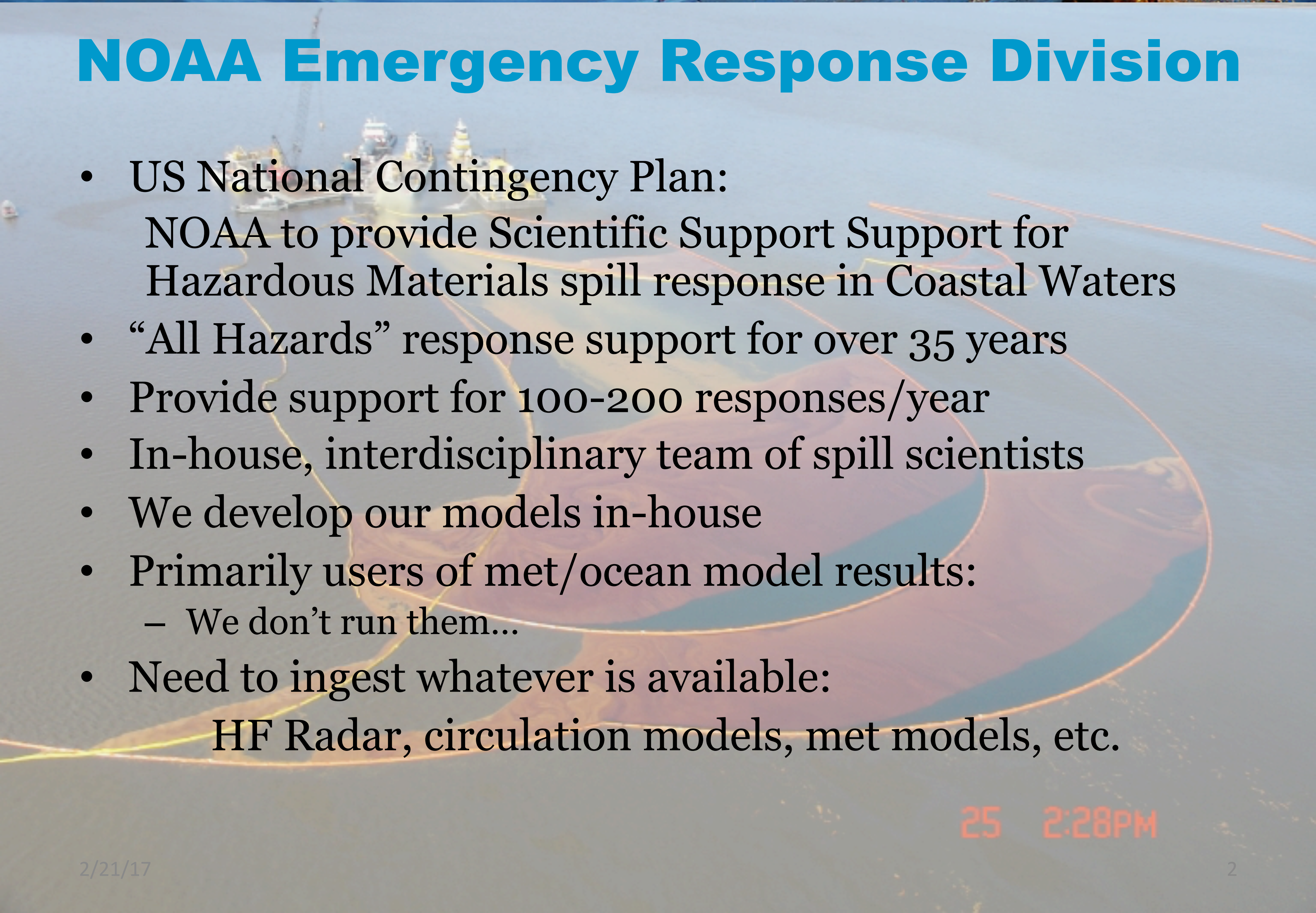
NOAA |Office of Response and Restoration | Emergency Response Division

"`gridded`": Multi-grid Data Analysis and Visualization with Python.

AMS 2017
Seventh Symposium on Advances in Modeling and Analysis Using Python

Jan 24, 2016
Christopher H. Barker
Emergency Response Division
National Oceanic and Atmospheric Administration

NOAA Emergency Response Division



- US National Contingency Plan:
  NOAA to provide Scientific Support Support for Hazardous Materials spill response in Coastal Waters
- "All Hazards" response support for over 35 years
- Provide support for 100-200 responses/year
- In-house, interdisciplinary team of spill scientists
- We develop our models in-house
- Primarily users of met/ocean model results:
  - We don't run them…
- Need to ingest whatever is available:
  HF Radar, circulation models, met models, etc.

GNOME

General **N**OAA **O**perational
Modeling **E**nvironment

- General particle tracking model – but primarily used for Oil Spill Modeling.
- Drivers:
  - Ocean, Coastal and Estuarine circulation models
  - Met models (surface winds)
  - Gridded data (i.e. National Digital Forecast Database)
- Largely written in Python
- **We're tired of writing custom code for each model!**

https://github.com/NOAA-ORR-ERD/PyGnome

The Problem

- There are a lot of models out there
- Models calculate and present their results on a particular grid system.
- Different models use different systems
  - Regular grids
  - Curvilinear grids
  - Unstructured grids
  - Staggered grids
- In order to use, visualize, analyze or compare model results, you need to understand each model's grid system (and it can be pretty complex to deal with them)

The Solution

Those of us that work with the results from multiple modeling systems need:

- Standardized file formats
- Libraries / APIs for working with these results that abstract out the differences.

The Standards

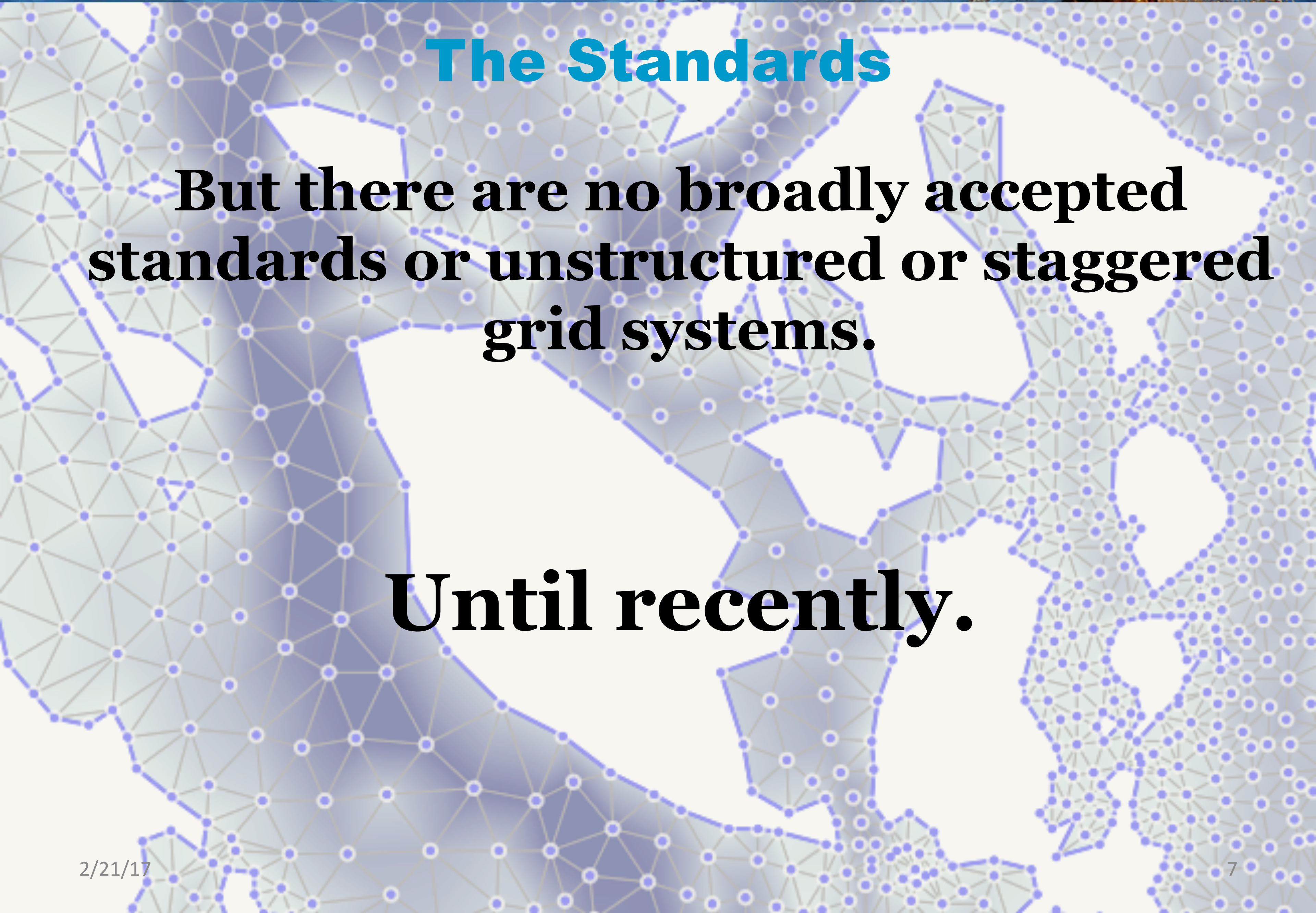
For Meteorological and Oceanographic work, the granddaddy of standards is:

NetCDF Climate and Forecast (CF) Metadata Conventions

http://cfconventions.org/

If your data is CF-compliant, there are great tools that make it easy to work with that data:

In Python: iris, xarray, etc.



The Standards

But there are no broadly accepted standards or unstructured or staggered grid systems.

Until recently.

UGRID + SGRID

UGRID: For unstructured grids, a standard has been developed over many years -- beginning with a meeting at Unidata in 2006

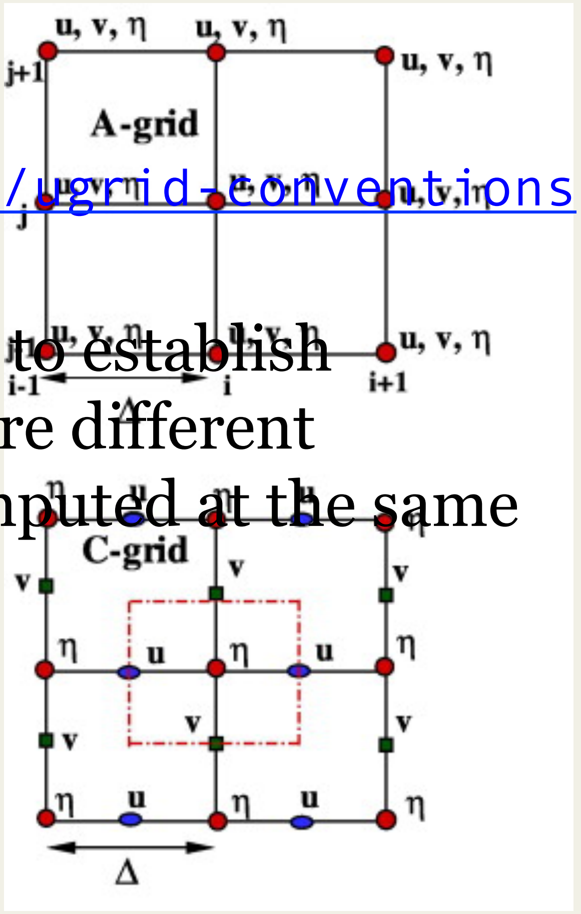
Community effort -- recently reached 1.0

http://ugrid-conventions.github.io/ugrid-conventions

SGRID is a more recent effort, seeking to establish conventions for "staggered grids", where different components of parameters are not computed at the same location on the grid.

Also a community effort:

http://sgrid.github.io/sgrid/



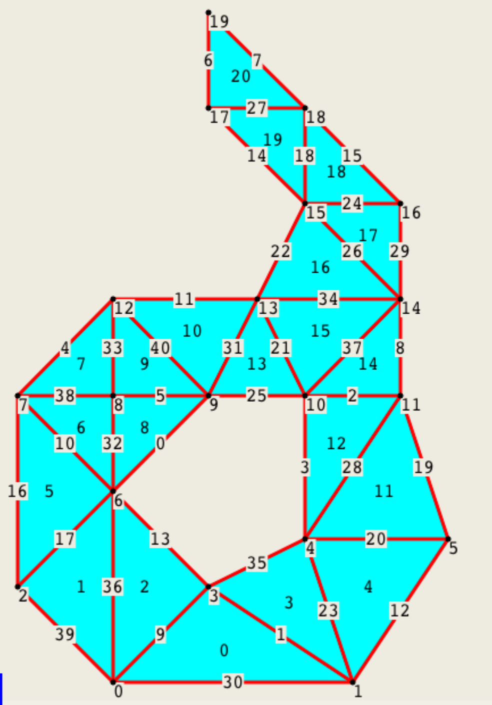
pyugrid

Python package for that supports the UGRID data model.

Provides reading/writing of UGRID-compliant datasets.

Facilitates grid:

- Manipulation
- Navigation
- Access data associated with the grid
- Interpolation



https://github.com/pyugrid/pyugrid

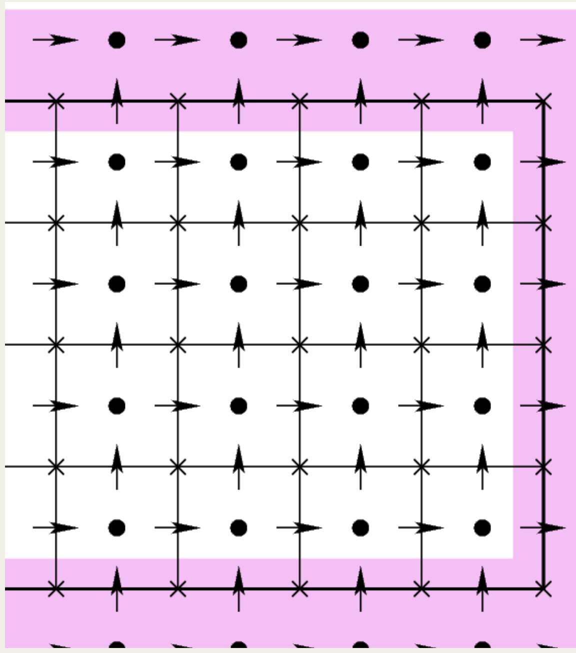
pysgrid

Python package for that supports the SGRID data model.

Provides reading/writing of UGRID-compliant datasets.

Facilitates grid:

- Manipulation
- Navigation
- Access data associated with the grid
- Interpolation



https://github.com/sgrid/pysgrid

gridded

Notice the similarity – Why have multiple packages / APIs for the same thing?

The goal of `gridded` is to provide a single API, so users can do:

Analysis – Computation – Visualization -- Inter-comparison

Without having to know what grid the results are on.

Kind of like an iris or xarray that understands the complex grid structure.

https://github.com/NOAA-ORR-ERD/gridded

Structure

Loading data:

- From netcdf file
- From OpenDAP url
- From already open `netCDF4 Dataset`
- From raw data collected from anywhere else

Can be as simple as:

```
import gridded
ds = gridded,Dataset(“a_netcdf_file.nc”)
```

This gets you a Dataset object you can work with

What's in a Dataset?

Create a Dataset:

```
In [8]: ds = Dataset("COOPS_NGOFS.nc")
```

It has a Grid object (Unstructured in this case)

```
In [10]: ds.grid
Out[10]: <gridded.grids.Grid_U at 0x110a902e8>
```

And a dict of variable objects:

```
In [11]: ds.variables.keys()
Out[11]: dict_keys(['nbe', 'u', 'v'])
```

What’s in a Grid?

Topology Info:

```
In [18]: ds.grid.nodes
array([[-97.15014648,  25.83448029],
       [-97.13427734,  25.83549118],
       ...,
       [-89.85437012,  29.99418068]])

In [19]: ds.grid.faces
array([[  171,    170,      0],
       [    1,    170,    172],
       ...,
       [90265, 90266, 90252],
       [90254, 90253, 90266]], dtype=int32)
```

But you generally don’t need to know about that.

What's in a Grid?

Lots of utilities for manipulating the grid

```
grid.build_boundaries
grid.build_face_face_conectivity
...
```

Cell finding and interpolation:

```
grid.locate_nodes(points)
grid.locate_faces(points)
grid.interpolate_var_to_points(points)
```

What's a Variable?

```
In [24]: u = ds.variables['u']
```

The actual data array (could be lazy-loaded netcdf variable):

```
In [25]: u.data[:]
array([[ 0.1068607 ,  0.12801492,  0.12656225, ...,
0.00156309,
```

Any attributes (pulled from netcdf variable)

```
In [26]: u.attributes
Out[26]:
{'long_name': 'Eastward Water Velocity',
 'mesh': 'fvcom_mesh',
 'standard_name': 'eastward_sea_water_velocity’,
```

What's a Variable?

A `Variable`'s data can be 2, 3, or 4D:

Each `Variable` has:

- A data array
- A `Grid` object that provides location / interpolation in the horizontal
- A `Time` object for interpolation in time
- A `Depth` object for interpolation in the vertical
    - `Depth` supports z and sigma coordinates

Interpolation:

```
Variable.at(points, time, extrapolate=False)
```

(`points` can be 2D or 3D)

(bi-linear or nearest neighbor for now)

Two "views" on the data

The whole data set:

- A `Dataset` represents a grid and a bunch of data on that grid.
- Essentially what is usually in a netcdf file
- Can be loaded from and saved to a file as a unit.

A single variable:

- Each `Variable` represents one field of one physical value:
    - Temperature, salinity, velocity
    - (there is vector Variable object, too)
- You can work with just a Variable, and the grid, etc. will be handled under the hood.

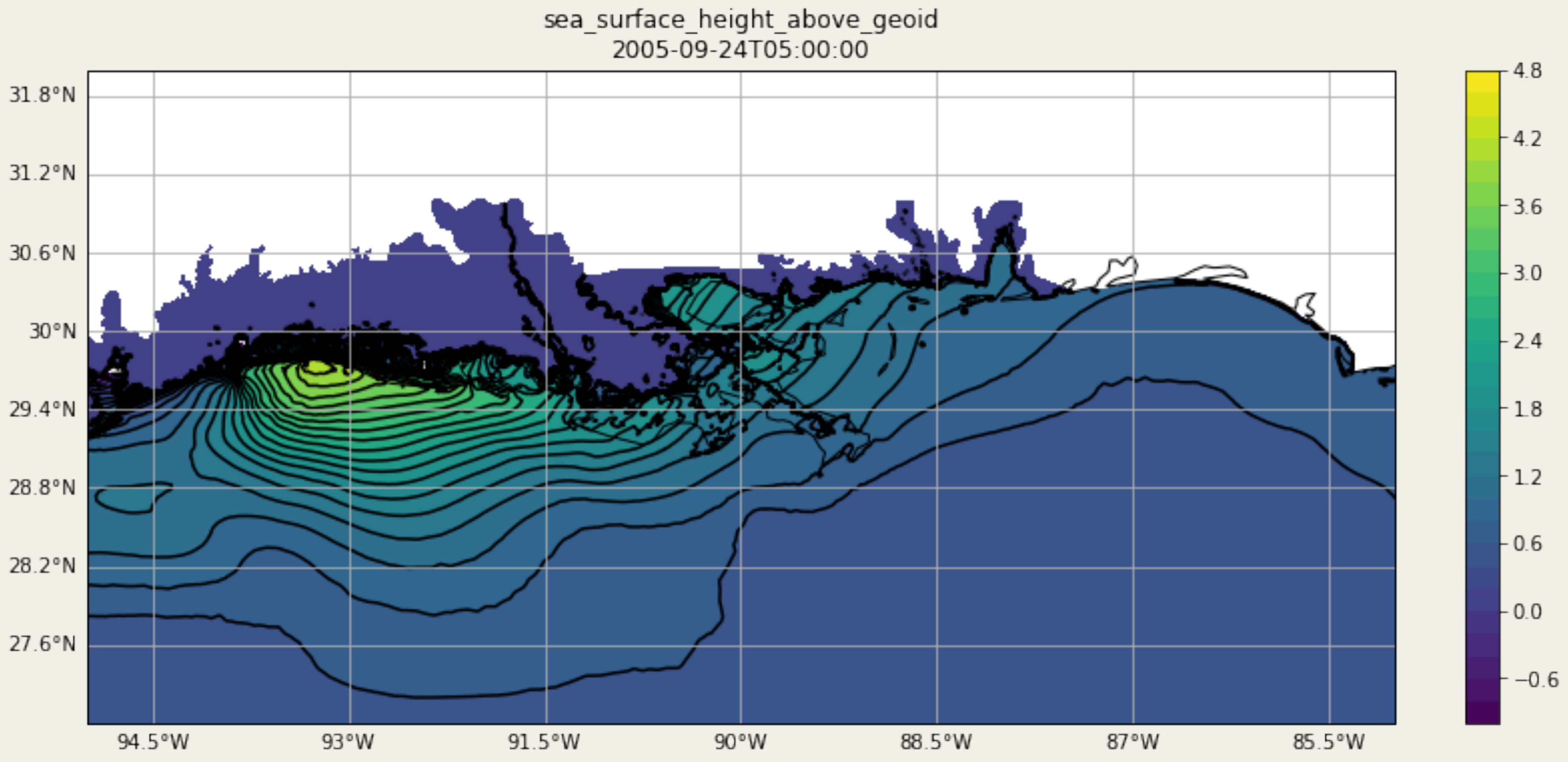
Performance

All operations designed for performance:

- All operations are numpy vectorized
- Data is lazy-loaded from file or OPenDAP url (often only what is required)
- Variables on the same grid share a Grid object
- Cell-finding uses high-performance cell_tree2d (C++)
- Grids “memoize” cell locations – so multiple queries for the same location don’t have to be repeated.

This is all code you don’t want to keep re-writing!

You can plot, etc. with whichever tools you like: Iris, Cartopy, Matplotlib



https://github.com/NOAA-ORR-ERD/gridded/blob/master/examples/UGRID_plotting_COMT.ipynb

Beyond Plotting

Plotting is the easy part:

- You know where the data are
- You just have to access the values

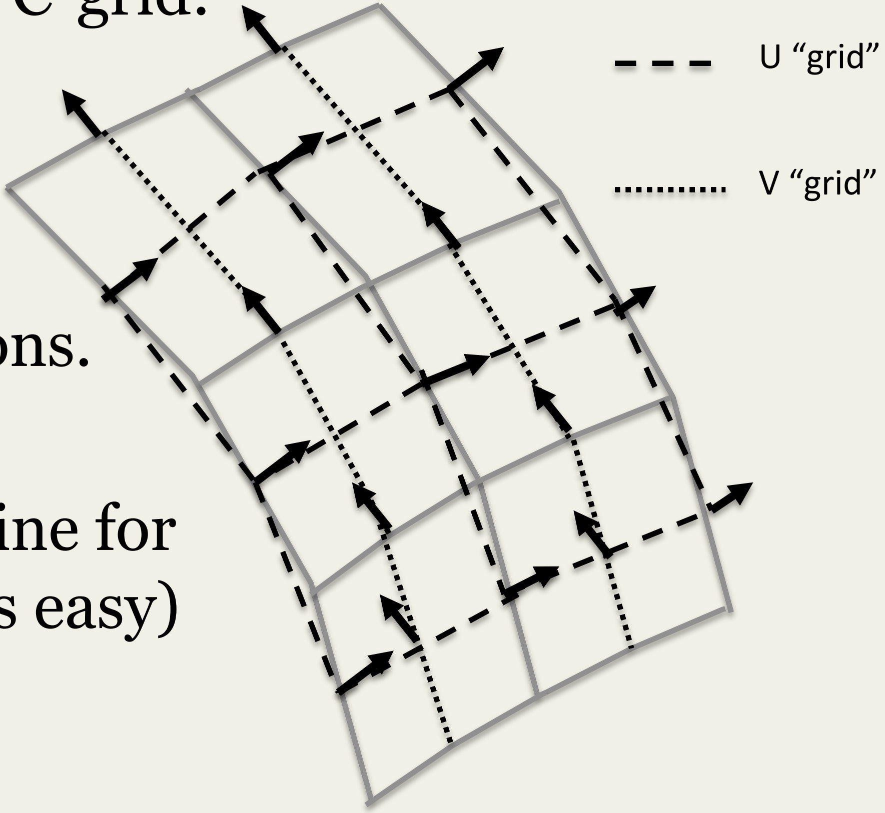
To do analysis or model comparison, or….

- You need to access data at a given point (in world coordinates)
- You may need to interpolate:

  not trivial on arbitrary quads…

Velocity on an SGRID

Example: ROMS and the Arakawa C-grid:



U and V at different locations on the cell:

Natural no flow boundary conditions.

Often interpolate to cell centers--fine for visualization (`gridded` makes this easy)

`gridded.sgrid`: interpolates U,V on their own "grids", puts them together and rotates them appropriately

Finding What Cell a Given point is in

Non-trivial: (826,866 cells)

```
%timeit ug.locate_faces((-93.5, 30.0), 'simple')
1 loops, best of 3: 10.9 s per loop
```

```
%timeit ug.locate_faces((-93.5, 30.0), 'celltree')
100000 loops, best of 3: 7.98 µs per loop
```

Celltree2d: A highly optimized data structure for locating cells in unstructured meshes:

```
https://github.com/NOAA-ORR-ERD/cell_tree2d/
```

Particle Tracking

To do particle tracking, you need to:

- Figure out where the particles are in the grid.
- Find the values of the velocities at that location:
  - Interpolate in space
  - Interpolate in time
  - Interpolate in depth ?
- Maybe multiple times – for RK methods

Each grid type requires specialized code:

- Single API provided by `gridded`

What if your file isn't compliant?

Utilities to help you load non-compliant files:

```
topology_names = {'nodes_lon': 'lon',
                  'nodes_lat': 'lat',
                  'faces': 'ele'
                  }
ds = Dataset.load_from_varnames(infilename, topology_names)
```

Now you have a "proper" `Dataset` object. Add variables if you like:

And save it out to get a fully compliant UGRID or SGRID netcdf file:

```
ds.save(outfilename, format=netcdf4)
```

Status

Fork me on GitHub

- Lots of working/tested code from:
  - pyugrid
  - pysgrid
  - py_gnome
- Code is semi-merged into gridded
- Interpolation works in horizontal and time
  - Vertical needs works

Future Work

Fork me on GitHub

- Iron out the API
- Documentation!
- PyPi and conda packages
- Custom plotting (Cartopy…)
- Subsetting grids ?
- Re-gridding ?
- Other interpolation methods ?
- Integrate with other packages: WRF Python?
- More of the xarray API?

Fork me on GitHub

Conclusions

The first step is metadata standards:

- Use **CF** and **UGRID** and **SGRID** so others can use your results.

`gridded` makes it easier to work with, and generate, compliant files.

`gridded` provides a model-agnostic analysis platform.

Join us on gitHub!

https://github.com/NOAA-ORR-ERD/gridded