Methodology for Ingest of a New Model into Ocean Cube

Introduction:

In developing the Ocean Cube application to interface with a new model, the main effort is to develop the routines to periodically retrieve the data and transform the incoming files to a standard netcdf file format such that the remaining ocean cube processing software recognizes and is capable of working with. Although there are likely changes needed to the remainder of the processing software and GUI that may result from model differences i.e., how often the model is run, number of TAUs, depth levels, etc., the focus of this document is the steps needed to retrieve and manipulate the data prior to calling the common ocean cube processing suite. This document will specifically address the development of a routine to incorporate the unstructured files produced by NOAA for both the Northern Gulf of Mexico Operational Forecast System (NGOFS2).

Data Source—NGOFS2:

NOAA maintains a web-based catalog that is updated as each the model runs are completed. The address of the catalog file can be obtained with a link:

https://opendap.co-ops.nos.noaa.gov/thredds/catalog/NOAA/NGOFS2/MODELS/YYYY/MM/DD/catalog.html

where “YYYY” is replaced with the year,

“MM” is replaced with the month, and

“DD” is replaced with the day.

Within this catalog web page is a list of links specified by file names. There are 3 types of files produced in this directory. The first is the nowcast/forecast at each station (we are not currently interested in these files). The second is the regular grid files that ocean cube routinely retrieves and processes with the “GetNGOFSDAP.py” python script running 4 times a day. The third is the unstructured data which is the non-gridded data produced at potentially higher resolution that the regular grid or structured data produced at the same time and location. The model files are represented by the following structure:

nos.XXX.SSS.NTTT.YYYYMMDD.tRRz.nc

where “SSS” is replaced with the file type either regulargrid (structured) or fields (unstructured),

"N" is replaced with either "n" for nowcast or "f" for forecast

"TTT" is replaced with TAU (or hour) value, (There are 3 values for nowcast: 000 ,003, or 006 and 13 values for forecast ranging from 000 to 048 in steps of 3)

"YYYY" is replaced with the year,

"MM" is replaced with the month,

"DD" is replaced with the day, and

"RR" is replaced with the valid or runtime hour (03, 09, 15, or 21).

The links on the data files take you to a THREDDS Data Server page specific for each file which contains 2 links of interest. The first is the OPENDAP link which takes one to a page which allows for a tailored download where parameters and ranges of vales can be chosen. The second link is a the HTTPServer link which allows for the entire file to be downloaded. The entire file sizes are as follows:

NGOFS2 regulargrid 660.1 MB per file,

NGOFS2 fields 636.2 MB per file,

By using the OPENDAP link, the file sizes of the data actually retrieved can be greatly reduced, particularly for the regulargrid where the latitude and longitude coordinates are sequentially spaced allowing for a geographic cutout on the server side. Size reduction for the fields data can also be reduced through the selection of only the parameters of interest. The OPENDAP link is used by the existing ocean cube model data retrieval software GetNGOFS.py for the download of regulargrid files. This document concentrates on the establishment of a new python script. "GetNGOFSField2.py" that will be used for the retrieval and manipulation of unstructured data contained in the …fields… files.

NOAA OPENDAP FIELD Query Form:

While the GetNGOFS2Field software will not use the query form directly, it is beneficial to view the form to understand the parameters available, dimensions f the parameters, and how the HTML query is formed. Figure 1 is the OPENDAP page that is presented when a link is made to a field file. NGOFS2 field files currently have 303,713 nodes and 569,404 elements (nele).

OPeNDAP Dataset Access Form

Action: Get ASCII | Get Binary | Show Help

Data URL: https://opendap.co-ops.nos.noaa.gov/thredds/dodsC/NOAA/NGOFS2/MOD

Global Attributes:

```
title: NGOFS2
institution: School for Marine Science and Technology
source: FVCOM_4.3
history: model started at: 29/04/2021   09:14
references: http://fvcom.smast.umassd.edu,
```

Variables:

☐ **nprocs**: 32 bit Integer

nprocs =

```
long_name: number of processors
```

☐ **partition**: Array of 32 bit Integers [nele = 0..569404]

nele:

```
long_name: partition
```

☐ **x**: Array of 32 bit Reals [node = 0..303713]

node:

```
long_name: nodal x-coordinate
units: meters
```

☐ **y**: Array of 32 bit Reals [node = 0..303713]

node:

```
long_name: nodal y-coordinate
units: meters
```

☐ **lon**: Array of 32 bit Reals [node = 0..303713]

node:

```
long_name: nodal longitude
standard_name: longitude
units: degrees_east
```

☐ **lat**: Array of 32 bit Reals [node = 0..303713]

node:

```
long_name: nodal latitude
standard_name: latitude
units: degrees_north
```

☐ **xc**: Array of 32 bit Reals [nele = 0..569404]

nele:

```
long_name: zonal x-coordinate
units: meters
```

☐ **yc**: Array of 32 bit Reals [nele = 0..569404]

nele:

```
long_name: zonal y-coordinate
units: meters
```

☐ **lonc**: Array of 32 bit Reals [nele = 0..569404]

nele:

```
long_name: zonal longitude
standard_name: longitude
units: degrees_east
```

☐ **latc**: Array of 32 bit Reals [nele = 0..569404]

nele:

```
long_name: zonal latitude
standard_name: latitude
units: degrees_north
```

☐ **siglay**: Array of 32 bit Reals [siglay = 0..39][node = 0..303713]

siglay: node:

```
long_name: Sigma Layers
standard_name: ocean_sigma/general_coordinate
positive: up
valid_min: -1.0
valid_max: 0.0
```

☐ **siglev**: Array of 32 bit Reals [siglev = 0..40][node = 0..303713]

siglev: node:

```
long_name: Sigma Levels
standard_name: ocean_sigma/general_coordinate
positive: up
valid_min: -1.0
valid_max: 0.0
```

☐ **h**: Array of 32 bit Reals [node = 0..303713]

node:

```
long_name: Bathymetry
standard_name: sea_floor_depth_below_geoid
units: m
positive: down
grid: Bathymetry_Mesh
```

☐ **nv**: Array of 32 bit Integers [three = 0..2][nele = 0..569404]

three: nele:

```
long_name: nodes surrounding element
```

☐ **iint**: Array of 32 bit Integers [time = 0..0]

time:

```
long_name: internal mode iteration number
```

☐ **time**: Array of 32 bit Reals [time = 0..0]

time:

```
long_name: time
units: days since 2019-01-01 00:00:00
format: defined reference date
time_zone: UTC
```

☐ **Itime**: Array of 32 bit Integers [time = 0..0]

time:

```
units: days since 2019-01-01 00:00:00
format: defined reference date
time_zone: UTC
```

☐ **Itime2**: **Array of 32 bit Integers [time = 0..0]**

time:

```
units: msec since 00:00:00
time_zone: UTC
```

☐ **Times**: **Array of Strings [time = 0..0]**

time:

```
time_zone: UTC
DODS:
  strlen: 26
  dimName: DateStrLen
```

☐ **zeta**: **Array of 32 bit Reals [time = 0..0][node = 0..303713]**

time: node:

```
long_name: Water Surface Elevation
units: meters
positive: up
standard_name: sea_surface_height_above_geoid
grid: Bathymetry_Mesh
```

☐ **nbe**: **Array of 32 bit Integers [three = 0..2][nele = 0..569404]**

three: nele:

```
long_name: elements surrounding each element
```

☐ **ntsn**: **Array of 32 bit Integers [node = 0..303713]**

node:

```
long_name: #nodes surrounding each node
```

☐ **nbsn**: **Array of 32 bit Integers [maxnode = 0..9][node = 0..303713]**

maxnode: node:

```
long_name: nodes surrounding each node
```

☐ **ntve**: **Array of 32 bit Integers [node = 0..303713]**

node:

```
long_name: #elems surrounding each node
```

☐ **nbve**: **Array of 32 bit Integers [maxelem = 0..7][node = 0..303713]**

maxelem: node:

```
long_name: elems surrounding each node
```

☐ **a1u**: **Array of 32 bit Reals [four = 0..3][nele = 0..569404]**

four: nele:

```
long_name: a1u
```

☐ **a2u**: **Array of 32 bit Reals [four = 0..3][nele = 0..569404]**

four: nele:

```
long_name: a2u
```

☐ **aw0**: **Array of 32 bit Reals [three = 0..2][nele = 0..569404]**

three: nele:

```
long_name: aw0
```

☐ **awx**: **Array of 32 bit Reals [three = 0..2][nele = 0..569404]**

three: nele:

```
long_name: awx
```

☐ **awy**: **Array of 32 bit Reals [three = 0..2][nele = 0..569404]**

three: nele:

```
long_name: awy
```

☐ **art2**: **Array of 32 bit Reals [node = 0..303713]**

node:

```
long_name: Area of elements around a node
```

☐ **art1**: **Array of 32 bit Reals [node = 0..303713]**

node:

```
long_name: Area of Node-Base Control volume
```

☐ **u**: **Array of 32 bit Reals [time = 0..0][siglay = 0..39][nele = 0..569404]**

time: siglay: nele:

```
long_name: Eastward Water Velocity
standard_name: eastward_sea_water_velocity
units: meters s-1
grid: fvcom_grid
type: data
```

☐ **v**: **Array of 32 bit Reals [time = 0..0][siglay = 0..39][nele = 0..569404]**

time: siglay: nele:

```
long_name: Northward Water Velocity
standard_name: Northward_sea_water_velocity
units: meters s-1
grid: fvcom_grid
type: data
```

☐ **tauc**: **Array of 32 bit Reals [time = 0..0][nele = 0..569404]**

time: nele:

```
long_name: bed stress magnitude from currents
note1: this stress is bottom boundary condtion on velocity field
note2: dimensions are stress/rho
units: m^2 s^-2
grid: fvcom_grid
```

☐ **omega**: **Array of 32 bit Reals [time = 0..0][siglev = 0..40][node = 0..303713]**

time: siglev: node:

```
long_name: Vertical Sigma Coordinate Velocity
units: s-1
grid: fvcom_grid
type: data
```

☐ **ww**: **Array of 32 bit Reals [time = 0..0][siglay = 0..39][nele = 0..569404]**

time: siglay: nele:

```
long_name: Upward Water Velocity
units: meters s-1
grid: fvcom_grid
type: data
```

☐ **temp**: **Array of 32 bit Reals [time = 0..0][siglay = 0..39][node = 0..303713]**

time: siglay: node:

```
long_name: temperature
standard_name: sea_water_temperature
units: degrees_C
grid: fvcom_grid
coordinates: time siglay lat lon
```

☐ **salinity**: **Array of 32 bit Reals [time = 0..0][siglay = 0..39][node = 0..303713]**

time: siglay: node:

```
long_name: salinity
standard_name: sea_water_salinity
units: 1e-3
grid: fvcom_grid
coordinates: time siglay lat lon
```

☐ **short_wave**: **Array of 32 bit Reals [time = 0..0][node = 0..303713]**

time: node:

```
long_name: Short Wave Radiation
units: W m-2
grid: fvcom_grid
coordinates: time lat lon
type: data
```

☐ **net_heat_flux**: **Array of 32 bit Reals [time = 0..0][node = 0..303713]**

time: node:

```
long_name: Surface Net Heat Flux
units: W m-2
grid: fvcom_grid
coordinates: time lat lon
type: data
```

☐ **uwind_speed**: **Array of 32 bit Reals [time = 0..0][nele = 0..569404]**

time: nele:

```
long_name: Eastward Wind Velocity
standard_name: eastward wind
units: meters s-1
grid: fvcom_grid
coordinates: time latc lonc
```

☐ **vwind_speed**: **Array of 32 bit Reals [time = 0..0][nele = 0..569404]**

time: nele:

```
long_name: Northward Wind Velocity
standard_name: northward wind
units: meters s-1
grid: fvcom_grid
coordinates: time latc lonc
```

☐ **atmos_press**: **Array of 32 bit Reals [time = 0..0][node = 0..303713]**

time: node:

```
long_name: Atmospheric Pressure
units: pascals
grid: fvcom_grid
coordinates: lat lon
type: data
```

☐ **wet_nodes**: **Array of 32 bit Integers [time = 0..0][node = 0..303713]**

time: node:

```
long_name: Wet_Nodes
grid: fvcom_grid
type: data
coordinates: time lat lon
mesh: fvcom_mesh
```

☐ **wet_cells**: **Array of 32 bit Integers [time = 0..0][nele = 0..569404]**

time: nele:

```
long_name: Wet_Cells
grid: fvcom_grid
type: data
coordinates: time latc lonc
mesh: fvcom_mesh
```

☐ **wet_nodes_prev_int**: **Array of 32 bit Integers [time = 0..0][node = 0..303713]**

time: node:

```
long_name: Wet_Nodes_At_Previous_Internal_Step
grid: fvcom_grid
type: data
coordinates: time lat lon
mesh: fvcom_mesh
```

☐ **wet_cells_prev_int**: Array of 32 bit Integers [time = 0..0][nele = 0..569404]

time: nele:

```
long_name: Wet_Cells_At_Previous_Internal_Step
grid: fvcom_grid
type: data
coordinates: time latc lonc
mesh: fvcom_mesh
```

☐ **wet_cells_prev_ext**: Array of 32 bit Integers [time = 0..0][nele = 0..569404]

time: nele:

```
long_name: Wet_Cells_At_Previous_External_Step
grid: fvcom_grid
type: data
```

For questions or comments about this dataset, contact the administrator of this server [Support] at: CO-OPS.USERSERVICES@noaa.gov

For questions or comments about OPeNDAP, email OPeNDAP support at: support@opendap.org

---

DDS:

```
Dataset {
    Int32 nprocs;
    Int32 partition[nele = 569405];
    Float32 x[node = 303714];
    Float32 y[node = 303714];
    Float32 lon[node = 303714];
    Float32 lat[node = 303714];
    Float32 xc[nele = 569405];
    Float32 yc[nele = 569405];
    Float32 lonc[nele = 569405];
    Float32 latc[nele = 569405];
    Float32 siglay[siglay = 40][node = 303714];
    Float32 siglev[siglev = 41][node = 303714];
    Float32 h[node = 303714];
    Int32 nv[three = 3][nele = 569405];
    Int32 iint[time = 1];
    Float32 time[time = 1];
    Int32 Itime[time = 1];
    Int32 Itime2[time = 1];
    String Times[time = 1];
    Float32 zeta[time = 1][node = 303714];
    Int32 nbe[three = 3][nele = 569405];
    Int32 ntsn[node = 303714];
    Int32 nbsn[maxnode = 10][node = 303714];
    Int32 ntve[node = 303714];
    Int32 nbve[maxelem = 8][node = 303714];
    Float32 a1u[four = 4][nele = 569405];
    Float32 a2u[four = 4][nele = 569405];
    Float32 aw0[three = 3][nele = 569405];
    Float32 awx[three = 3][nele = 569405];
    Float32 awy[three = 3][nele = 569405];
    Float32 art2[node = 303714];
    Float32 art1[node = 303714];
    Float32 u[time = 1][siglay = 40][nele = 569405];
    Float32 v[time = 1][siglay = 40][nele = 569405];
    Float32 tauc[time = 1][nele = 569405];
    Float32 omega[time = 1][siglev = 41][node = 303714];
    Float32 ww[time = 1][siglay = 40][nele = 569405];
    Float32 temp[time = 1][siglay = 40][node = 303714];
    Float32 salinity[time = 1][siglay = 40][node = 303714];
    Float32 short_wave[time = 1][node = 303714];
    Float32 net_heat_flux[time = 1][node = 303714];
    Float32 uwind_speed[time = 1][nele = 569405];
    Float32 vwind_speed[time = 1][nele = 569405];
    Float32 atmos_press[time = 1][node = 303714];
    Int32 wet_nodes[time = 1][node = 303714];
    Int32 wet_cells[time = 1][nele = 569405];
    Int32 wet_nodes_prev_int[time = 1][node = 303714];
    Int32 wet_cells_prev_int[time = 1][nele = 569405];
    Int32 wet_cells_prev_ext[time = 1][nele = 569405];
} NOAA/NGOFS2/MODELS/2021/04/29/nos.ngofs2.fields.f006.20210429.t09z.nc;
```

Figure 1

As seen from the query form, there are numerous parameters that may be selected but for the purposes of ocean cube the only parameters that are currently of interest are zeta (Water Surface Elevation), u (Eastward Water Velocity), v (Northward Water Velocity), temp (Water Temperature), and salinity. In addition, to position the data both horizontally and vertically we must retrieve Lon, Lat, LonC, LatC, siglay, h, nbve and ntve (I've included uwind_speed and vwind_speed for future development). The resulting DODS data query for NEGOFS field data will appear similar to the following ('%5B' and '%5D' are substituted for '['and ']' respectively):

https://opendap.co-ops.nos.noaa.gov/thredds/dodsC/NOAA/NGOFS2/MODELS/2021/04/29/nos.ngofs2.fields.f006.20210429.t09z.nc.dods?siglay%5B0:1:39%5D%5B184291:1:197582%5D,zeta%5B0:1:0%5D%5B184291:1:197582%5D,u%5B0:1:0%5D%5B0:1:39%5D%5B351586:1:377803%5D,v%5B0:1:0%5D%5B0:1:39%5D%5B351586:1:377803%5D,temp%5B0:1:0%5D%5B0:1:39%5D%5B184291:1:197582%5D,salinity%5B0:1:0%5D%5B0:1:39%5D%5B184291:1:197582%5D,uwind_speed%5B0:1:0%5D%5B351586:1:377803%5D,vwind_speed%5B0:1:0%5D%5B351586:1:377803%5D

Ocean Cube Toolkit Description:

Inputs:

The GetNGOFS2Field.py script program starts, as do all of the model retrieval software, with the gathering of parameters necessary for processing. Initially, there are the command line options, which among other things, point a set of area and model specific parameters contained in a file named cube.json. The command line options are as follows:

Python GetNGOFS2Field.py AREA_ID -dtg YYYYMMDDHH –launch

Where:

AREA_ID is a nave of a set of parameters stored in cube.json,

-dtg is followed by the model run date and time (HH is one of 03, 09, 15, 21). Argument -dtg is optional and if omitted the default will be to wait for the next model un to be generated based on the current time.

--launch is used to an optional parameter to indicate that upon completion of the download and manipulation of the model data, the python script CubeRun.py will be launched to complete the generation of the ocean cube web pages and accompanying graphics.

Typically for automatic regeneration of the web pages, crontab entries are made at the appropriate times as “python GetNGOFSField2.py GLP_NGOFS2 --launch”.

The next set of parameters are user configuration parameters stored for the area in the cube.json file. This file is stored in the cubeenv directory which is set up by the environmental variable “JSONPATH”. Several other environmental path variables are used and if they are not set, the defaults are taken. Here is the python code used to retrieve the paths needed and the defaults used:

```
if os.getenv('JSONPATH',None)==None:
    os.environ['JSONPATH']="/home/braud/cubeenv"
if os.getenv("NETCDFPATH",None)==None:
    os.environ['NETCDFPATH']='/home/braud/cubeenv/Netcdffiles'
if os.getenv("CUBE",None)==None:
    os.environ['CUBE']="/home/braud/Cube/CubeRun"
if os.getenv("WORK",None)==None:
    os.environ['WORK']="/home/braud/cubeenv/data"
```

Because in developing the unstructured model input, a major objective is to develop high resolution small areas for examination, a special 0.03-degree latitude by 0.03-degree longitude

area was established in the Mississippi Sound just west of the Gulfport harbor ship channel. The following entry was added to the cube.json file with the Area_ID of GLP_NGOFS2:


```
  {
    "STREAK_LENGTH": "0.025",
    "AREA_ID": "GLP_NGOFS2",
    "HEIGHTS_INC": "1",
    "TEMP_BOTTOM": "0",
    "DISTRIUBTION_REASON": "Admin/Ops",
    "TEMP_MAX": "100",
    "AREA_NAME": "Gulf of Mexico",
    "CURRENTS_BOTTOM": "0",
    "Y_AXIS_TICKS": "0.00125",
    "MODEL_AREA_ID": "",
    "DERIVED_FROM": "",
    "X_AXIS_TICKS": "0.00125",
    "MODEL": "NGOFS2",
    "X_AXIS_GRIDLINE": "0.0125",
    "CURRENTS": "1",
    "DISTRIBUTION_STATEMENT": "T",
    "HEIGHTS_CONTOUR": "1",
    "HEIGHTS_MIN": "-4",
    "TEMP": "1",
    "BOTTOM": "0",
    "Y_GRID_ANNO": "0.0125",
    "STREAK_SPACING": "0.005",
    "SALINITY": "1",
    "HEIGHTS_MAX": "4",
    "SALINITY_MIN": "30",
    "CURRENTS_INC": ".25",
    "SOUTH": 30.3,
    "CLASSIFIED_BY": "",
    "TEMP_MIN": "50",
    "SALINITY_BOTTOM": "0",
    "EAST": -89.1,
    "RESOLUTION": "0.5",
    "COASTLINE": "DEFAULT",
    "TEMP_INC": "5",
    "CLASSIFICATION": "UNCLASSIFIED",
    "FILL_LAND": "1",
    "WAVES": "0",
    "WINDS": "0",
    "SALINITY_INC": "2",
    "HEIGHTS": "1",
```

```
    "DEPTHS": "0,2,4",
    "TEMP_CONTOUR": "0",
    "DISTRIBUTION_OFFICE": "University of Southern Mississippi",
    "AREA_TITLE": "Mississippi Sound, Gulfport",
    "CURRENTS_MIN": "0",
    "ACTIVE": "1",
    "NORTH": 30.33,
    "TEMP_OVERLAY": "CURRENTS",
    "SALINITY_MAX": "40",
    "CURRENTS_CONTOUR": "0",
    "SALINITY_OVERLAY": "CURRENTS",
    "CHLOROPHYLL": "CURRENTS",
    "STREAK_KEY_MAX": "3",
    "WEST": -89.13,
    "X_GRID_ANNO": "0.0125",
    "SUBMIT_TIME": "20210302134232",
    "STRIDE": "3",
    "Y_AXIS_GRIDLINE": "0.0125",
    "CURRENTS_VECTORS": "0",
    "CURRENTS_MAX": "1",
    "SALINITY_CONTOUR": "0",
    "HEIGHTS_OVERLAY": "CURRENTS"
  },
```


Most of the parameters in this entry are used to generate the graphical images and have to do with chart boundaries, density of vector arrows, any overlays produced, and annotation. For the purpose of retrieving and modifying the NEGOFS field files the parameters used are:
"MODEL": "NEGOFS",
"NORTH": 30.33,
"SOUTH": 30.3,
"EAST": -89.8,
"WEST": -89.8, and
"DEPTHS": "0,2,4"

When used for regulargrid retrievals these parameters would be used to generate the exact dimensions of the data to be retrieved. When accessing the unstructured data, the entire parameter set must be retrieved and then filtered using these parameters to select the points and values that are within the selected area.

Processing:
Once all the needed information has been retrieved from the command line, environmental variables and cube.json file, a list of each of the needed filenames is constructed from the model run and TAUs using the template for filenames described above . An URL string pointing to the web page containing the file names is also generated. A sample URL name would be

"https://opendap.co-ops.nos.noaa.gov/thredds/dodsC/NOAA/NEGOFS/MODELS/2021/03/04/" and a sample corresponding file name would be" nos.negofs.fields.f002.20210304.t09z.nc". Since the normal mode of automatic download is to run GetNGOFSField2.py from a cron execution and to start it up prior to the model data actually being available, getNEGOFSField.py attempts to poll the web to retrieve the request header for the first file. This occurs once per minute for a period of up to 10 hours or until the returned request header has a status code of 200 indicating a successful retrieval. Once it has been established that at least one file from the model run exists, a multiprocessing task is set up to retrieve and manipulate each of the required model files. Each process is started by the "download" subroutine which is passed sufficient information to construct and execute the full DoDs query for each file. This is currently set up to process 10 files at a time but is adjustable by modifying the variable "nproc" within the "main" portion of the python script. At this point the greatest concern for proposing a 10-process limit is that downloading more than 10 large files simultaneously may put an undue strain on network bandwidth. As with the main program, care is taken that the file actually exists, so the download subroutine itself does not actually read the data. It constructs the full URL but only attempts to read a single "lat" (latitude value) for the file. It will continually poll for the file every minute until the request header status returned is the value of 200 indicating that the file is available. The subroutine convert_ngofs is then called to read the file and produce an ocean cube compatible netcdf file. The main program spawns the tasks for each file that is downloaded and waits for all tasks to complete. Once all the ocean cube netcdf files have been created form the downloaded files, the optional launch parameter is evaluated. If the launch option is included, the CubeRun.py program is invoked to process these files and generate the files needed by the oceancube web application. The source file for GetNGOFS2.py is included in the Appendix A.

Conclusion:

The software developed to demonstrate the methodology for incorporating a new model data source into the ocean cube web application. It mainly assumes that new data is arriving on a frequent time schedule (daily or multiple times daily) and handles the automation of processing these data for oceancube web update. The development was somewhat forced by changes made to the previously used NGOFS and NEGOFS data sources produced by NOAA and replaced in late March 2021 by the NGOFS2 data source. This program, however, specifically focused on the need to incorporate the unstructured data that is provided by NOAA that may be gridded at higher resolutions for larger scale areas. This is particularly important since the previous NOAA produced NEGOFS with 500-meter resolution was replaced by NGOFS2 with 1000-meter resolution for the regulargrid downloads. Using the NGOFS2 unstructued model data a cutout for a large-scale area (approximately 3km by 3km) is produced in the Mississippi Sound and is gridded to provide a 192 meter resolution. The web page found at http://oceancube.usm.edu/GLP_NGOFS2/GLP_NGOFS2.html is automatically produced as the NGOFS2 data is updated. An image of a web page output is provided in Appendix B.

Appendix A: Source Code Listing of GetNGOFS2.py

```
'''
Created on March 11, 2021

@author: James Braud
Description:
This program is a toolkit developed to read in the unstructured OPENDODS Field model
data produced 4 times daily
and distributed on a NOAA THREDDS for Northern Gulf of Mexico Ocean Forecast System
(NGOFS2). It filters the unstructured
data for a specified geographic area and list of standard depths.  It downloads the
necessary files and invokes a GMT
Mapping Toolkit (GMT) to convert each parameter at each depth to a regularly spaced
grid using the
nearneighbor algorithm. (GMT version 5 or later must be available and in the
execution path.)
Once all of the regularly spaced netcdf files have been produced, they are read in
and reassembled
into a new netcdf file in a format that is expected by the subsequent CubeRun.py
program that
will assemble and produce all the data needed for the Ocean Cube application.
Version:
1.0
ChangeLog:
04/21/21 Add comments JB
'''
import datetime
import os, sys
import time
from datetime import timedelta
import argparse
import json
import requests
import multiprocessing
import numpy as np
import math
import subprocess
from netCDF4 import Dataset
from pydap.client import open_dods
def dist(lat,lng,lat0,lng0):
    #
    #Distance formula for 2 latitude and longitude positions
    #
    x = lat - lat0
    y = (lng - lng0)*math.cos(lat0)
    return math.sqrt(x*x + y*y)

def ReturnJSONParameters (inputFile,searchField,searchItem):
    with open(inputFile) as data_file:
        data = json.load(data_file)
        fieldDict = []
        for item in data:
```

```
            for thing in item:
                if ( (thing.upper() == searchField.upper()) and (item[thing].upper()
== searchItem.upper()) ):
                    fieldDict.append(item)
    return(fieldDict)
def gengrid(lat,lon,d,values,dashR,dashI,ot):
    #
    #This routine calls the Generic Mapping Toolkit (GMT) to create regularly spaced
grid of the values
    #array using a minimum curvature spline algorithm on the latitude and longitude
positions of the input.
    #
    t="" #start with empty string
    #
    #Loop through the points and write it out as a string of Lon, lat and value
triplets
    #
    count=0
    for k in range(0,len(lat)):
        value=values[k][d]
        if not np.isnan(value):
            t+=str(lon[k])+" "+str(lat[k])+" "+str(value)+"\n"
        count+=1
    #
    #I've tested both gridding algorithms in GMT--They both use all of the processors
available up to 16
    #greenspline algorithm is considerable slower and gobbles up huge memory for
large number of points. Its processing is related
    #to the square of the incoming points.
    #surface is the more traditional algorithm
    #after reviewing the results for a small area,  151 points there was discernible
difference in the images created.
    #Large files (8000 points) crashed my system with only 16GB memory.
    #The NGOFS2 for the USM area requires 1GB of memory and has 51000 points.
    #I am choosing to use the surface algorithm
    #
    #call="gmt greenspline -R"+dashR+" -D2 -I"+dashI+" -G"+ot
    #call="gmt surface -R"+dashR+" -M4c -Ve -T0.025 -I"+dashI+" -G"+ot
    call="gmt nearneighbor -G" + ot + " -I"+dashI+" -S"+str(float(dashI)*10)+"d -
N8+m2 -R"+dashR
    print call
    #
    #Call the GMT spline interpolator
    #dashR contains the boundaries of the area
    #-D2 option is to 2D interpolate latitude and longitude positions
    #dashI is the resolution in the lat and lon direction (same resolution in
degrees)
    #-G is followed by the output file name
    #
    p = subprocess.Popen(call.split(),stdin=subprocess.PIPE)
    #
    #feed the string in as input to GMT
    #
    p.communicate(input=t.encode())[0]
    #p.wait()
```

```
def
convert_ngofs(outfilename,url,latitude,longitude,h,nodes,ele,north,south,east,west,de
pths):
    print "Working Directory is "+os.getcwd()+" Filename is "+outfilename
    #
    #open the .dods netcdf file and read in all the parameters needed
    #
    infile=open_dods(url)
    #
    #siglay for each node is an array of values with a range 0 to 1 as a
multiplication factor of h (bathy) to compute depths of parameters at this node.
    #
    siglay=np.array(infile['siglay'][:]) #layers at each node
    t=np.array(infile['temp'][:]) #water temperature at each node and siglay
    s=np.array(infile['salinity'][:]) #salinity at each node and siglay
    z=np.array(infile['zeta'][:]) #surface elevation or height
    u0=np.array(infile['u'][:]) #eastward water velocity component for each element
and siglev
    v0=np.array(infile['v'][:]) #northward water velocity component for each element
and siglev
    u2=np.array(infile['uwind_speed'][:]) #eastward water velocity component for each
element and siglev
    v2=np.array(infile['vwind_speed'][:]) #northward water velocity component for
each element and siglev
    #
    #Set up the new arrays to be generated
    #
    lat=[]
    lon=[]
    salinity=[]
    temp=[]
    surf_el=[]
    u=[]
    v=[]
    winds_u=[]
    winds_v=[]
    #
    #file uses 0-360 longitude so we convert to the what ocean cube expects (-180 to
180)
    #
    if east<0:east=360.+east
    if west<0:west+=360.
    for i in range(0,len(latitude)):
        #if latitude[i]< south or latitude[i]>north or longitude[i]<west or
longitude[i]>east:continue
        lat.append(latitude[i])
        #
        #store longitude in range -180 to 180 if the east or west boundaries require
it
        #
        lon.append(longitude[i])
        if longitude[i]<0:
            lon[-1]+=360.
        #
```

```
        #Make surf_el an array of arrays with only a 0 depth value to conform to
structures for the other parameters
        #
        z1=[]
        z1.append(z[0][i])
        surf_el.append (z1)
        #
        #initialize with surface values (siglev[0])--assume that layer[0] is the
surface for all parameters
        #
        s1=[]
        s1.append(s[0][0][i])
        t1=[]
        t1.append(t[0][0][i])
        #
        #compute the actual bathy of siglay[0].  The first standard depth value that
is to be interpolated to should always be larger than this.
        #
        lastd=-(h[i]*siglay[0][i])
        #
        #Since we cut out only the elements that are needed for the area using
OPenDODS, we need to adjust the element pointers by subtracting the start
        #
        nearest=ele[i]-min(ele)
        u1=[]
        u1.append(u0[0][0][nearest])
        v1=[]
        v1.append(v0[0][0][nearest])

        z1=[]
        z1.append(u2[0][nearest])
        winds_u.append (z1)
        z1=[]
        z1.append(v2[0][nearest])
        winds_v.append (z1)

        #
        #for the depths other than the surface, find the layers that the standard
depth lies between
        #This allows us to find an interpolation factor that will be applied to all
the parameter values at this location
        #
        if len(depths)>1:
            l=1 #l is used to index the standard depths a            for k in
range(1,siglay.shape[0]-1):
            for k in range(1,siglay.shape[0]-1):
                #
                #cycle through the layer depths until we get one greater than the
current standard depth.
                #
                nextd=-(h[i]*siglay[k][i])
                if nextd>=depths[l]:
                    #
                    #factor is the interpolation value used to adjust the level to
the nearest standard depth.
```

```
                    #It will be applied for salinity, temp, u and v
                    #
                    factor=(nextd-depths[l])/(nextd-lastd)
                    s1.append(s[0][k][i]-(s[0][k][i]-s[0][k-1][i])*factor)
                    t1.append(t[0][k][i]-(t[0][k][i]-t[0][k-1][i])*factor)
                    u1.append(u0[0][k][nearest]-(u0[0][k][nearest]-u0[0][k-
1][nearest])*factor)
                    v1.append(v0[0][k][nearest]-(v0[0][k][nearest]-v0[0][k-
1][nearest])*factor)
                    l+=1 #next standard depth index
                    if l>=len(depths):break
                lastd=nextd
            #
            #If the standard depth is deeper than all layers at this location--null
fill all parameter values
            #
            for j in range(l,len(depths)):
                s1.append(float("NaN"))
                t1.append(float("NaN"))
                u1.append(float("NaN"))
                v1.append(float("NaN"))
        salinity.append(s1)
        temp.append(t1)
        u.append(u1)
        v.append(v1)
    #
    #Determine estimate of grid spacing (resolution) based on the number of points in
the area. (Assumes uniform distribution)
    #
    d=math.sqrt(len(lat))+1
    x=(north-south)/d
    #
    #Write the resolution to a file so that it can be picked up and displayed
(make_html.py script) on the web page.
    #This file will be attempted to be read in, only if there is no "RESOLUTION"
parameter in cube.json for this area
    #
    if outfilename[-8:]=="_t000.nc":#only do this 1 time -- first file
        midlat=(north+south)/2
        londist=dist(midlat,east,midlat,east-x)
        resolution=int((x+londist)/2.*111111.) #convert the distance to meters
assuming 1 degree is about 111.111km
        resolution /=1000. #convert to kilometers--use 3 decimal places
        f=open("resolution.txt",'w')
        f.write (str(resolution))
        f.close()
    #
    #dashI is a parameter used by GMT indicating the resolution of the interpolater
    #
    dashI=str(x)
    #adjust east and north boundaries so that they are multiples of resoution
    #
    eastinc=int((east-west)/x)+1
    east+=eastinc*x
    northinc=int((north-south)/x)+1
```

```
    north+=northinc*x
    #
    #dashR is a parameter used by GMT to define the boundaries of an area
    #
    dashR=str(west)+"/"+str(east)+"/"+str(south)+"/"+str(north)
    #
    #create an array of parameters to send to gengrid (used to generate gridded files
for each parameter and at each depth)
    #
    outfile=[]
    values=[]
    lev=[]
    #
    #Surface elevation only has one call to gengrid
    #
    outfile.append("surf_el_0_"+outfilename)
    values.append(surf_el)
    lev.append(0)
    outfile.append("winds_u_0_"+outfilename)
    values.append(winds_u)
    lev.append(0)
    outfile.append("winds_v_0_"+outfilename)
    values.append(winds_v)
    lev.append(0)
    #
    #All other parameters will have a gengrid call for each standard depth
    #
    for i in range(0,len(depths)):
        outfile.append("salinity_"+str(i)+"_"+outfilename)
        values.append(salinity)
        lev.append(i)
        outfile.append("temp_"+str(i)+"_"+outfilename)
        values.append(temp)
        lev.append(i)
        outfile.append("u_"+str(i)+"_"+outfilename)
        values.append(u)
        lev.append(i)
        outfile.append("v_"+str(i)+"_"+outfilename)
        values.append(v)
        lev.append(i)

processes=[multiprocessing.Process(target=gengrid,args=(lat,lon,lev[index],values[ind
ex],dashR,dashI,outfile[index]))for index in range(0,len(outfile))]
    procs=[]
    nproc=1
    #
    #Initially start 10 (nproc) processes and continue to start all of the processes
as each one finishes.
    #
    for p in processes:
        while len(procs)>=nproc:
            for proc in procs:
                if not proc.is_alive():
                    procs.remove(proc)
            time.sleep(.1)
```

```
        p.start()
        procs.append(p)
    #
    #wait for all to complete
    #
    while len(procs)>=1:
        for proc in procs:
            if not proc.is_alive():
                procs.remove(proc)
        time.sleep(.1)
    #
    #Open and read the first file netcdf created by the GMT program (surface
elevation)
    #
    infile=Dataset(outfile[0],'r')
    lats=[]
    lons=[]
    #
    #The lats and lons are the same for all files--If surface is the gridding
algorithm, the variables will be y and x respectively
    #
    latname='Lat'
    lonname='Lon'
    try:
        latitude=infile.dimensions[latname]
    except:
        latname='y'
        lonname='x'
    latitude=infile.variables[latname][:]

    longitude=infile.variables[lonname][:]
    for i in range(0,infile.dimensions[latname].size):
        lats.append(latitude[i])
    for i in range(0,infile.dimensions[lonname].size):
        if longitude[i]>180:
            lons.append(longitude[i]-360.)
        else:
            lons.append(longitude[i])
    height = infile.variables['z'][:]  # The z values are the surface heights
    #
    #close and delete the file
    #
    infile.close()
    os.remove(outfile[0])
    #
    #Single level for surface winds
    #
    infile=Dataset(outfile[1],'r')
    winds_u = infile.variables['z'][:]  # The z values are the winds u component
    infile.close()
    os.remove(outfile[1])
    infile=Dataset(outfile[2],'r')
    winds_v = infile.variables['z'][:]  # The z values are the winds v component
    infile.close()
    os.remove(outfile[2])
```

```
    #
    #Now process all the files with depth layers
    #
    paramstart=3
    #
    #The requested depth layer may go deeper than the data, If that is the case (all
missing) no file is produced
    #create a all NaN array to fill in the data
    #
    nanarray=[]
    for i in range(0,len(lats)):
        newrow=[]
        for j in range(0,len(lons)):
            newrow.append(float('NaN'))
        nanarray.append(newrow)
    #
    #process the remaining files produced by the GMT program (one for each parameter
and depth)
    #
    for index in range(paramstart,len(outfile)):
        if os.path.exists(outfile[index]):
            infile=Dataset(outfile[index],'r')
            #
            #close and delete the file
            #
            z=infile.variables['z'][:]
            infile.close()
            os.remove(outfile[index])
        else: #if file is missing, use the NaN array
            z=nanarray
        i=(index-paramstart) % 4 #There are 4 parameters that are cycled through for
each depth
        if i==0:
            if index<paramstart+4: #if this is the first salinity file (surface)
initialize the numpy array.
                salinity=z  # extract/copy the data
            else: # add all other files as depth layers for this parameter
                salinity=np.concatenate((salinity,z),axis=0)
        elif i==1: #if this is the first temperature file (surface) initialize the
numpy array.
            if index<paramstart+4:
                temp=z  # extract/copy the data
            else: # add all other files as depth layers for this parameter
                temp=np.concatenate((temp,z),axis=0)
        elif i==2: #if this is the first water_u file (surface) initialize the numpy
array.
            if index<paramstart+4:
                water_u=z  # extract/copy the data
            else: # add all other files as depth layers for this parameter
                water_u=np.concatenate((water_u,z),axis=0)
        elif i==3: #if this is the first water_v file (surface) initialize the numpy
array.
            if index<paramstart+4:
                water_v=z  # extract/copy the data
```

```
            else: # add all other files as depth layers for this parameter
                water_v=np.concatenate((water_v,z),axis=0)
    #
    #numpy concatenate just adds the values to the array. numpy reshape will create
an array with a depth dimension
    #
    salinity=np.reshape(salinity,(len(depths),len(lats),len(lons)))
    temp=np.reshape(temp,(len(depths),len(lats),len(lons)))
    water_u=np.reshape(water_u,(len(depths),len(lats),len(lons)))
    water_v=np.reshape(water_v,(len(depths),len(lats),len(lons)))
    #
    #Ready to create a new netcdf file with all of the parameters and naming
conventions needed by the ocean cube application
    #
    outfile=Dataset(outfilename,"w")
    #
    #create the depth dimension and attributes
    #
    depthdim=outfile.createDimension('depth',len(depths))
    depthvar=outfile.createVariable('depth','f4',('depth',))
    depthvar.units="m"
    depthvar.long_name= "Depths of Standard Layer"
    depthvar.positive="down"
    mindepth=min(depths)
    maxdepth=max(depths)
    depthvar.actual_range=(mindepth,maxdepth)
    #
    #populate the depth values
    #
    depthvar[:]=depths
    #
    #create the lat dimension and attributes
    #
    latdim=outfile.createDimension('lat',len(lats))
    minlat=min(lats)
    maxlat=max(lats)
    latvar=outfile.createVariable('lat','f4',('lat',))
    latvar.units= "degrees_north"
    latvar.long_name= "Latitude in common grid"
    latvar.actual_range=(minlat,maxlat)
    #
    #populate the lat values
    #
    latvar[:]=lats
    #
    #create the lon dimension and attributes
    #
    londim=outfile.createDimension('lon',len(lons))
    minlon=min(lons)
    maxlon=max(lons)
    lonvar=outfile.createVariable('lon','f4',('lon',))
    lonvar.units="degrees_east"
    lonvar.long_name= "Latitude in common grid"
    lonvar.actual_range=(minlon,maxlon)
    #
```

```
    #populate the lon values
    #
    lonvar[:]=lons
    #
    #create the time dimension and attributes
    #
    times=[0]
    timedim=outfile.createDimension('time',1)
    timevar=outfile.createVariable('time','f4',('time',))
    #
    #populate the time value--a netcdf file is created for each hour or TAU--it will
only have one time value of 0
    #
    timevar[:]=times
    #
    #for each parameter set up the variables (with appropriate dimensions)
    #

variables=['surf_el','water_u','water_v','water_temp','salinity','winds_u','winds_v']
    for j in range(0,len(variables)):
        vname=variables[j]
        if vname=="surf_el": #surf_el has no depth dimension so create it separately
from other parameters

datavar=outfile.createVariable(vname,'f4',('time','lat','lon',),fill_value=float("NaN
"))
            #
            #numpy reshape is needed for all parameters to add a time dimension
            #
            height=np.reshape(height,(-1,len(lats),len(lons),))
            datavar[:]=height[:]
        elif vname=="winds_u":

datavar=outfile.createVariable(vname,'f4',('time','lat','lon',),fill_value=float("NaN
"))
            winds_u=np.reshape(winds_u,(-1,)+winds_u.shape)
            datavar[:]=winds_u[:]
        elif vname=="winds_v":

datavar=outfile.createVariable(vname,'f4',('time','lat','lon',),fill_value=float("NaN
"))
            winds_v=np.reshape(winds_v,(-1,)+winds_v.shape)
            datavar[:]=winds_v[:]
        else:

datavar=outfile.createVariable(vname,'f4',('time','depth','lat','lon',),fill_value=fl
oat("NaN"))
            if vname=="water_u":
                water_u=np.reshape(water_u,(-1,)+water_u.shape)
                datavar[:]=water_u[:]
            elif vname=="water_v":
                water_v=np.reshape(water_v,(-1,)+water_v.shape)
                datavar[:]=water_v[:]
            elif vname=="water_temp":
                temp=np.reshape(temp,(-1,)+temp.shape)
```

```
                datavar[:]=temp[:]
            elif vname=="salinity":
                salinity=np.reshape(salinity,(-1,)+salinity.shape)
                datavar[:]=salinity[:]
    #
    #All variable added--close the netcdf file and terminate this process
    #
    outfile.close
def
download(urlbase,name,filename,areaID,lat,lon,h,nodes,ele,north,south,east,west,depth
s,modeltype):
    if os.path.isfile(filename):os.remove(filename)
    #
    #try to retrieve file 10 times
    #
    trys=10
    maxelem="%5B0:1:8%5D"
    #
    #The different models have different dimensions
    #
    if modeltype=="NEGOFS":
        siglay="%5B0:1:19%5D"
    else:
        siglay="%5B0:1:39%5D"
    #
    #create a single time dimension
    #
    ocean_time="%5B0:1:0%5D"
    nele="%5B"+str(min(ele))+":1:"+str(max(ele))+"%5D"
    node="%5B"+str(min(nodes))+":1:"+str(max(nodes))+"%5D"
    #
    #try to retrieve file 10 times over 10 minutes
    #
    while not os.path.isfile(filename) and trys > 0:
        trys-=1
        if os.path.isfile(filename):continue #if file already exists we have probably
been through this already
        while True:
            #
            #retrieve1 Latitude value from file to make sure it is there
            #
            url=urlbase+name+".dods?lat"+ocean_time
            try:
                #res=requests.head(url)
                res=requests.get(url)
            except:
                print name,"Connection Refused"
                time.sleep(60)
                continue
            if(res.status_code == 200): break # good result
            #
            #Bad result: wait 60 seconds and try again
            #
            print "failed",url
            print res.status_code
```

```
            time.sleep(60)
        #
        #create the entire URL with all parameters and dimensions
        #
        allnele=ocean_time+siglay+nele
        allnode=ocean_time+siglay+node

finalurl=urlbase+name+".dods?siglay"+siglay+node+",zeta"+ocean_time+node+",u"+allnele
+",v"+allnele+",temp"+allnode+",salinity"+allnode+",uwind_speed"+ocean_time+nele+",vw
ind_speed"+ocean_time+nele
        print finalurl
        #
        #file to download exists, go download all parameters and convert
        #

convert_ngofs(filename,finalurl,lat,lon,h,nodes,ele,north,south,east,west,depths)
        #os.remove(name)
#
#
#This is where the execution starts
#
if __name__ == '__main__':
    #
    #Use today's date by default, but should allow for past dates on the command line
    #
    parser = argparse.ArgumentParser(description='Get NGOFS model data and optionally
launch cube software.')
    parser.add_argument('areaID',
                        help='values should match the name stored in "AREA_ID" field
of the cube.json file')
    parser.add_argument('-dtg',
                        help="Optional dateTime string in the form of YYYYMMDDHH.
Default is today's date.")
    parser.add_argument('--launch', dest='launch', action='store_true',
default=False,
                        help="Optional flag to launch processing of the newly
acquired data. Default is no launch.")
    args = parser.parse_args()
    if args.dtg != None:
        loaddate=time.strptime(args.dtg[0:8],"%Y%m%d")
        #
        #break dtg into components needed for the OPENDAP querry
        #
        year=args.dtg[0:4]
        month=args.dtg[4:6]
        hour=args.dtg[8:]
        day=args.dtg[6:8]
        #
        #Convert the time to a date for comparison
        #
        dtg=datetime.date(loaddate.tm_year,loaddate.tm_mon,loaddate.tm_mday)
    else: #if no dtg specified on the command line use current time as a start
        dtg=datetime.datetime.utcnow()
        if dtg.hour>23:dtg+=timedelta(hours=3) # Is the next model run crossing days
        #
```

```
        #set up for the next model run (4 times daily at 03, 09, 15 and 21)
        #
        hour="21"
        if dtg.hour<17:hour="15"
        if dtg.hour<11:hour="09"
        if dtg.hour<5:hour="03"
        year=str(dtg.year)
        month=str(dtg.month).zfill(2)
        day=str(dtg.day).zfill(2)

    areaID=args.areaID
    #
    #load needed environmental variables
    #
    if os.getenv('JSONPATH',None)==None:
        os.environ['JSONPATH']="/home/braud/cubeenv"
    if os.getenv("NETCDFPATH",None)==None:
        os.environ['NETCDFPATH']='/home/braud/cubeenv/Netcdffiles'
    if os.getenv("CUBE",None)==None:
        os.environ['CUBE']="/home/braud/Cube/CubeRun"
    if os.getenv("WORK",None)==None:
        os.environ['WORK']="/home/braud/cubeenv/data"
    print datetime.datetime.now()
    #
    #read the cube.json file to get needed information
    #
    imdHash = ReturnJSONParameters(os.getenv('JSONPATH')+'/cube.json', 'AREA_ID',
areaID)
    #
    if not imdHash:
        print "Could not find entry in "+os.getenv('JSONPATH')+'/cube.json for id
'+areaID
        sys.exit(1==0)
    #
    #Go to directory where the files will be placed. Create it if it does not exist
    #
    try:
        os.chdir(os.environ['NETCDFPATH'] + "/" + areaID)
    except:
        os.mkdir(os.environ['NETCDFPATH'] + "/" + areaID)
        os.chdir(os.environ['NETCDFPATH'] + "/" + areaID)
    #
    #Default time stride is 1 hour intervals
    #
    try:
        stride = int(imdHash[0]['STRIDE'])
    except:
        stride = 1 #default
    #
    #get the bounding box--west, east, south, and north
    #
    try:
        west = float(imdHash[0]['WEST'])-.001
    except:
```

```
        print "Could not find WEST entry in
"+os.getenv('JSONPATH')+'/'+os.getenv('AREAFILE')+'.json for id '+areaID
        sys.exit(1==0)
    try:
        east = float(imdHash[0]['EAST'])+.001
    except:
        print "Could not find EAST entry in
"+os.getenv('JSONPATH')+'/'+os.getenv('AREAFILE')+'.json for id '+areaID
        sys.exit(1==0)
    try:
        south = float(imdHash[0]['SOUTH'])-.001
    except:
        print "Could not find SOUTH entry in
"+os.getenv('JSONPATH')+'/'+os.getenv('AREAFILE')+'.json for id '+areaID
        sys.exit(1==0)
    try:
        north = float(imdHash[0]['NORTH'])+.001
    except:
        print "Could not find NORTH entry in
"+os.getenv('JSONPATH')+'/'+os.getenv('AREAFILE')+'.json for id '+areaID
        sys.exit(1==0)
    #
    #make sure the coordinates make sense
    #
    if (north <= south) or ((east <= west) and ((west-east) > 180.)): #allow for
crossing 180
        print "Invalid bounding rectangle South="+str(south)+" North="+str(north)+"
West="+str(west)+" East="+str(east)
        sys.exit(1==0)
    #
    #get the depths
    #
    try:
        dstr= imdHash[0]['DEPTHS'].split(',')
        depths=[]
        for d in dstr:
            depths.append(int(d))

    except:
        depths = [0,2,4,6,8,10,12,15,20,25] # default
    #
    #get the model (NGOFS or NEGOFS)
    #
    try:
        model = imdHash[0]['MODEL']
    except:
        model = "NGOFS2" #default
    if model=="NEGOFS":
        modelfile="negofs"
    elif model=="NGOFS":
        modelfile="ngofs"
    else:
        modelfile="ngofs2"
    #
```

```
    #If a file with a list of nodes and nearest elements exists for this area read it
in. Otherwise create it.
    #
    urlbase="https://opendap.co-
ops.nos.noaa.gov/thredds/dodsC/NOAA/"+model+"/MODELS/"+year+"/"+month+"/"+day+"/"

    name=[]
    filename=[]
    #
    #build a list of file names-- 000-006 nowcast and 001-048 forecast
    #
    for x in range (0,7,stride):
        name.append
("nos."+modelfile+".fields.n00"+str(x)+"."+year+month+day+".t"+hour+"z.nc")
        filename.append(year+month+day+hour+"_t"+str(x).zfill(3)+".nc")
    for x in range (stride,49,stride):
        name.append
("nos."+modelfile+".fields.f0"+str(x).zfill(2)+"."+year+month+day+".t"+hour+"z.nc")
        filename.append(year+month+day+hour+"_t"+str(x+6).zfill(3)+".nc")
    url=urlbase+name[len(name)-1]
    #
    #Try getting the whole file from the fileServer link (It shows up here first, but
if its here it will be on the OpenDODS server)
    #
    url=url.replace("dodsC","fileServer")
    waitforit=False
    maxtrys=600 #10 hours
    #
    #wait until a file shows up
    #
    while maxtrys>0:
        maxtrys-=1
        #Now see if we can get the data:  A lot of work goes on the server side to
slice the data so the response is not immediate
        print url
        #res=requests.get(url,stream=True)
        try:
            res=requests.head(url)
        except requests.exceptions.ConnectionError:
            print "Connection refused"
            time.sleep(60)
            continue
        print(res.status_code,datetime.datetime.utcnow())
        if(res.status_code == 200): break # good result
        #Bad result: wait 60 seconds and try again
        time.sleep(60)
        waitforit=True
    #
    #Exit if we try for more than 10 hours
    #
    if maxtrys==0:exit(1)
    res.close()
    #
    #If a file with a list of nodes and nearest elements exists for this area read it
in. Otherwise create it.
```

```
    #This is a one time deal to process the entire file (currently over 600MB for the
GOM) and filter to only those
    #nodes and elements in the requested area. A file (nodes.txt) is written out with
a list of node and element indexes so
    #that they will not be needed by subsequent file downloads.  When used with the
OpenDODs sever, this can result in a tremendous
    #savings in bandwidth an processing requirements. If the coordinates for an
AREA_ID change as defined in the cube.json
    #file, the node.txt file located in the cubeenv/Netcdffiles/(AREA_ID) directory
should be deleted so that a new one
    #is created.
    #
    lat=[]
    lon=[]
    h=[]
    nodes=[]
    ele=[]
    #
    #The latitude, Longitude, depth, node index and neaest element index do not
change from model run to model run
    #Download these one time for an area and write them to a file so that they will
not need to be downloaded again
    #
    if not os.path.exists("nodes.txt"):
        res=requests.get(url,stream=True)
        #filename=os.environ['NETCDFPATH']+'/'+areaID+"/"+name
        ofile=open(name[-1],'w+b')
        for chunk in res.iter_content(chunk_size=1000000):
            ofile.write(chunk)
        ofile.close
        res.close
        #
        #open the .dods netcdf file and read in all the parameters needed
        #
        infile=Dataset(name[-1])
        latitude = np.array(infile['lat'][:])  #latitude at each node
        longitude = np.array(infile['lon'][:])  #longitude at each node
        depth = np.array(infile['h'][:]) #bathymetry at each node
        ntve = np.array(infile['ntve'][:]) #number of surrounding elements for each
node (up to 8)
        nbve = np.array(infile['nbve'][:]) #indexes of the surrounding elements for
each node
        #
        #siglay for each node is an array of values with a range 0 to 1 as a
multiplication factor of h (bathy) to compute depths of parameters at this node.
        #
        latitudec = np.array(infile['latc'][:])  #latitude at each element
        longitudec = np.array(infile['lonc'][:])  #longitude at each element
        #
        #Set up the new arrays to be generated
        #
        #
        #file uses 0-360 longitude so we convert to the what ocean cube expects (-180
to 180)
        #
```

```
        if east<0:east=360.+east
        if west<0:west+=360.
        f=open("nodes.txt",'w')
        #
        #Geographically filter--make sure the point is within the area
        #
        for i in range(0,len(latitude)):
            if latitude[i]< south or latitude[i]>north or longitude[i]<west or
longitude[i]>east:continue
            lat.append(latitude[i])
            #
            #store longitude in range -180 to 180
            #
            if longitude[i]>180:
                lon.append(longitude[i]-360.)
            else:
                lon.append(longitude[i])
            h.append(depth[i])
            #
            #find the nearest element (snap the values to this node's location -- we
only have siglay for elements
            #no way to accurately determine depth given siglay with no h or
bathymetry value.)
            #
            #Start with the first element being the nearest and compute its distance
from the node
            #
            nearest=nbve[0][i]

shortest=dist(latitude[i],longitude[i],latitudec[nbve[0][i]],longitudec[nbve[0][i]])
            #
            #Cycle through the surrounding elements to find the closest--
            #
            for k in range(1,ntve[i]):
                #
                #compute the distance for this element
                #

d=dist(latitude[i],longitude[i],latitudec[nbve[k][i]],longitudec[nbve[k][i]])
                if d < shortest:
                    #
                    #This element is closer
                    #
                    shortest=d
                    nearest=nbve[k][i]
            nodes.append(i)
            ele.append(nearest)
            f.write(str(lat[-1])+","+str(lon[-1])+","+str(h[-1])+","+str(nodes[-
1])+","+str(ele[-1])+"\n")

    else: #after the 1st run for an area is made, this file as created with the above
code should be available
        f=open("nodes.txt")
        for line in f:
            fields=line.split(',')
```

```
            lat.append(float(fields[0]))
            lon.append(float(fields[1]))
            h.append(float(fields[2]))
            nodes.append(int(fields[3]))
            ele.append(int(fields[4]))
    f.close()
    processes = [multiprocessing.Process(target=download,
args=(urlbase,name[x],filename[x],areaID,lat,lon,h,nodes,ele,north,south,east,west,de
pths,model)) for x in range(0,len(name))]
    procs=[]
    nproc=10
    #
    #Initially start 10 (nproc) processes and continue to start all of the processes
as each one finishes.
    #
    for p in processes:
        while len(procs)>=nproc:
            for proc in procs:
                if not proc.is_alive():
                    procs.remove(proc)
            time.sleep(.1)
        p.start()
        procs.append(p)
    #
    #wait for all to complete
    #
    while len(procs)>=1:
        for proc in procs:
            if not proc.is_alive():
                procs.remove(proc)
        time.sleep(.1)
    #
    #All files have been created: At this point we should launch CubeRun.py
    #
    if args.launch:
        print "Launch Processing ",datetime.datetime.now()
        os.chdir(os.environ['CUBE'])
        call='python '+os.getenv('CUBE')+'/CubeRun.py '+year+month+day+hour+'
'+areaID+' > '+os.getenv('WORK')+"/"+dtg.strftime("%Y%m%d")+hour+areaID+'.log'
        print call
        os.system(call)
        #
        #wait until complete
        #
    print datetime.datetime.now()
```

Appendix B: Web page for high resolution NGOFS2 unstructured model data (http://oceancube.usm.edu/GLP_NGOFS2/GLP_NGOFS2.html)