**Visualizing ADCIRC outputs through GIS shape files or Google Earth (GE) KMZ files using Python**

This document describes the methodology used in kalpana.py for the creation of shape file and kmz files. The sequence of steps involved has been grouped under related sub-headings for better comprehension.

(The terms in italics refer to variable names in kalpana.py)

**Code Overview**

The kalpana.py script allows us to convert netCDF formatted ADCIRC output files and convert them to GIS shape files and KMZ files. The ADCIRC output parameters being considered are the following:

1. Bathymetry
2. Maximum Water levels (maxele.63.nc)
3. Maximum Wave Heights (swan\_HS\_max.63.nc)
4. Maximum Wind Velocity (maxwvel.63.nc)
5. Maximum Peak Period (swan\_TPS\_max.63.nc)
6. Water levels (fort.63.nc)
7. Wind Speeds (fort.74.nc)
8. Wave heights (swan\_HS.63.nc)

The code outlines the processes involved for generating polyline and polygon shape files for each of the above types of files. It also generates KMZ files for maximum water levels, wave heights and wind speeds and peak wave periods.

**Input data**

* 1. ADCIRC output file in netCDF format or the corresponding OPeNDAP url
  2. Appropriate palette files (.pal extension). The palette files included with the current package are water-level.pal, wavht.pal. These palette files are used for kml creation.
  3. To have some control over the symbology in which the shape files are visualized .mxd and .lyr files have been created outside the code. To package them with the shape file distribution a directory with the same name as the shape file has also been included. This directory contains a sample shape file, a .mxd and a .lyr file - all with the same file name. Every time a new shape file with the same name is created the previous one is overwritten. The .mxd and .lyr automatically references the current shape file (with the same file name as the .mxd/.lyr file) within the directory. Providing the .mxd and .lyr files is completely optional.

**Modules to be installed**

1. matplotlib

-an important python module for data visualization

1. shapely

-module which can be used to construct geometric objects like Points, Polygons and LineStrings

1. fiona

-to write .shp files

1. netCDF4

-to read and write .nc files

1. datetime

- to work with dates and times in python

1. time

- module used for time related functions such as calculating the run time

1. numpy

- powerful python module for scientific computing (here used primarily for working with ndimensional numpy arrays which are ideal for storing large amount of data)

1. collections

- used in this code for accessing OrderedDict (dict subclass that remembers the order in which entries were added whereas an ordinary dictionary does not do so)

1. simplekml

* used for writing kml file

The following link provides a useful repository of Windows binary installers for various python modules: <http://www.lfd.uci.edu/~gohlke/pythonlibs/>

**Defining Long/Lat Boxes for the whole domain**

While creating the kmz files for the entire domain of NC9 it was noticed that the largest polygon (corresponding to the lowest contour level) was incorrectly rendered. This was due to the number of vertices in its outer boundary being too high. It exceeded 31000 vertices which is the upper limit specified in Google Earth documentation for a polygon boundary to be rendered correctly.

As a solution to this issue the entire mesh domain is divided into a series of lat-long bins such that there are more bins in regions of higher mesh resolution. The idea is to plot contour levels and generate polygons for each of these sub domains separately and later combine them.

In the present code the entire domain is specified using *gdomain* = [50, 5,-60,-100] (i.e. North-Lat = 50o N, South-Lat = 5o N, East-Long = 60o W, West-Long =100oW). The user is then prompted to enter a local box (our area of interest which has a higher mesh resolution) where bins can be defined at smaller latitude intervals. For the NC9 mesh we have used a local box of: North-Lat = 36, South-Lat = 33.5, East-long = -60, West-Long = -100. This constituted the regions near and along the NC coast which very finely resolved in this mesh.

Within this local box different lat-long bins which are 0.5o wide in the N-S direction are created. The sign conventions used are as in Table1.

Table1 Sign Convention

|  |  |
| --- | --- |
| Latitude | N – positive, S - negative |
| Longitude | E – positive, W - negative |

A total of 7 lat long bins created as follows worked for NC9 and NC 9.99 meshes:

[[33.5, 5, -60, -100], [34.0, 33.5, -60, -100], [34.5, 34.0, -60, -100], [35.0, 34.5, -60, -100], [35.5, 35.0, -60, -100], [36.0, 35.5, -60, -100], [50, 36.0, -60, -100]]

It is to be noted the same procedure is applicable for the visualization of results of other complex meshes. However the local box has to be chosen appropriately. Also the bins are chosen assuming that the mesh is more finely resolved in the N-S direction within the local box. This approach may cause problems while applying to a coastline which has very fine features extending along the E-W direction as well. For such applications the procedure of selection of subdomains will have to be modified appropriately.

**Accepting User Input**

The user is asked to input the following:

* Name of the Storm (*storm*)
* ADCIRC output to be visualized (*filechoice*)
* Polyline or Polygon (*shape*)
* GIS Shape file or KML file(*vchoice*)
* Whether to combine subplots or plot the full domain directly (*domain*) (applicable for KMZ creation)
* Local box (*l*) (applicable for KMZ creation)
* buffer distance specified to ensure smooth overlap of the plots of adjacent long/lat boxes (*lonlatbuffer*)(applicable for KMZ creation)

lonlaatbuffer = 0 was found to work fine with the above defined bins

**Processing User Input**

Appropriate information is read into python variables depending upon the *filechoice*:

* Name of ADCIRC output file in netCDF format
* Name of the variable in the .nc file to be plotted (*vname*)
* Name of the palette file (*palettename*)
* Contour Levels to be plotted (*levels*)
* Name of Output Shape File (*outputname*)
* Name of Folder in Output KML file (*foldname*)

**Extracting data from the input file (ADCIRC output file in .nc format) as numpy arrays**

* Longitude (*lon*) and Latitude (*lat*) of every vertex in the entire domain
* Description of triangular elements (*nv*) as numpy arrays of 3 vertices for each triangle.
* ADCIRC output values (like maximum water elevation, maximum wave height, maximum wind speed etc.) stored against the corresponding variable name (*var*)
  + If the ADCIRC output file contains \_FILL VALUE (= -99999) *var* gets stored as a masked numpy array. The masked or redundant value is replaced by -100
  + Unit conversion is done if required
  + For better visualization the extreme values for each variable are readjusted
* Time step information (*time\_var*)
  + Used to extract *startdate*
  + The time step values are converted into ‘datetime’ objects and then into a string format to use while writing shapefiles later (used for time series output data)

**General Settings for Shape Files and KML files**

* For Shape files
  + Define Spatial Reference(*crs*) which specifies the ellipsoid , datum and projection used
  + Define OGR Driver (*driver*)
* For KML files
  + Create a simplekml object (*kml*)
  + Define a region (*reg*) with a specified lat-long box (*box*) and Level of Detail (LOD) (*lod*) (specifying the minimum and maximum pixels the kml object should occupy before it disappears)
  + Based on the number of contour levels specified, the colors in the chosen palette file are interpolated to obtain a list of hex color codes (*hex*). These are used as colors for polygon outline and fill while generating the kmz files.
  + Creation of colorbar based on the colors specified in *hex*
  + Screen Overlays are created to insert the logo and color bar into the kmz file

**Processing input data for every time step**

The following processes are repeated at every time step:

* Triangulation and contouring
  + If you are plotting directly for the full domain then the triangulation (*tri*) and contouring (*contour*) is done for the full domain
  + If plotting for subdomains a local mesh is created for each subdomain long-lat box. The procedure for deriving a local mesh from a global mesh is explained in APPENDIX A.

Triangulation and contouring is done separately for each local mesh using the vertex (*localx*, *localy*), element (*localelements*) and variable (*localvar*) information derived for each local mesh.

* Extracting information from the tricontour/tricontourf plot
  + *enumerate (contour.collections)* gives a list of contour levels (*colli*) in the form of indices and contours that store all the vertices lying in that contour in the form of path objects (*coll*). Each contour level may have more than one path.
  + We can access all the paths corresponding to a single contour level using *get\_paths()*.
    - For creating polylines the vertices stored in each of the paths can be extracted using *.vertices* and wrapped into a shapely LineString object
    - For creating polygons each of the paths can be converted into polygons (*polys* - a list of numpy arrays of vertices defined for each polygon) using *to\_polygons()*.
  + For writing shape files
    - The above set of polygons corresponding to each path is wrapped into a shapely Polygon object.
    - An empty ordered dictionary (*geoms*) is defined outside the time step loop and each time step is added as a key to this dictionary
    - A tuple consisting of the shapely Polygon object and the corresponding minimum and maximum contour levels is stored in *geoms* against the appropriate time step. This step is repeated for all the contour levels.

Thus *geoms* has the following form:

*{“timestep1”:[(Polygon1,min1,max1),(Polygon2,min2.max2),(….)…] “timestep2”:[ ( ….), (…)…..]……}*

* + For writing kml files
    - Define a new folder (*fol*) for every time step.
    - A multigeometry kml object (*multipol*) is defined for each contour level as a child of *fol.* A list of these multigeometry elements is created (*store*).
    - *polys* is converted to a list of tuples (*polys1*) where each tuple represents a polygon.
    - The outer and inner polygons in *polys1* are identified by calculating their signed area. If the area is positive it is identified as an outer and if it is negative it is identified as an inner polygon. A list of outer (*outer*) and inner (*inner*) polygons are made separately.
    - The list of outer polygons is traversed. For every outer polygon the inner polygons which fall within it are identified. To do this is we check if any one of the vertices on the boundary of an inner polygon falls within the outer polygon (no 2 polygons can intersect each other since they represent contour levels). A list of the indices of inner polygons which fall within an outer polygon is stored in a dictionary (*topo*) with the outer polygon number as keys.
    - For each contour level, each of the outer polygons with all its inner polygons are stored as a child element of the corresponding multi geometry object (*store[m]* where *m* refers to each contour level)
    - Visibility, fill, color, outline, description etc. is defined for each multigeometry object (*store[m]*).

1. **Defining schema and writing shape files**

* Defining the schema (*schema*) of each shape file includes stating the type of geometry i.e. Line String, Polygon etc., and the attribute properties (name of attribute and data type (int, float etc.) corresponding to each output shape file.
* A new shape file is opened using Fiona. The data stored in *geom* corresponding to each time step is written into this shape file.

1. **Creating the KMZ file**

* The simplekml object is saved as a .kml file.
* The kml file is zipped with the corresponding color bar file (ex: Colorbar-water-levels.png) and the logo (logo.png).
* Zipping the files is not included in kalpana.py. It can be done using the zipfile module of python.

**APPENDIX A – Creating local mesh from the global mesh for a specified lat/long box**

1. Define a subdomain using the bounding latitude (North and South) and longitude (East and West) values. These are defined as *LatN, LatS, LongE, and LongW* respectively.
2. Specify a buffer distance to have a smooth overlap of the plots from adjacent subdomains
3. Traverse through all the elements in the mesh. If any of the vertices of each element falls within a specified lat-long box then that element and all its vertices are considered to be within that lat-long box. The values at the corresponding indices (representing the element number or node number) in *includeele1* and *includevertex1* are updated to 1.
4. Traverse through all elements in the mesh. Check the value of *includevertex1* for all the vertices for each element. If the value is 1 for any of them update *includeele2* and *includevertex2* at the corresponding vertices to 1 like in Step 3.
5. Traverse through all the nodes in the mesh. Check the value of *includevertex2* for each node. If the value is 1 then consider that node to be in the new local mesh and include its latitude and longitude values in *locallat* and *locallon* respectively. Renumber the vertices that fall in the local mesh and create a look up table with the global and local node numbers of each vertex. Also store the value of the ADCIRC output variable corresponding to these vertices in *varlocal*.
6. Repeat step 5 for all the elements of the domain. For each element check if *includeele2* at the corresponding element number. If it is 1 then append a tuple containing the coordinates of each of its vertices to the list *localelem.*
7. Return *locallat, locallon, localelem, varlocal.*