2D Meshing Example with PyLaGriT

Author: Charles Abolt (cabolt@lanl.gov; chuck.abolt@beg.utexas.edu)
Date: June 16, 2017

Modified from Dylan's example *arctic_2D_new* on GitHub

This example will demonstrate how to create a mesh for 2D ATS simulations using data from an elevation transect. The output mesh is in Exodus II format and composed of hexahedral volumes. Optionally, the mesh can be partitioned for parallel simulations, creating a set of *.par* files. Horizontal distance along the transect is represented on the x-axis, and elevation is represented on the z-axis. The cells also have a small, constant width along the y-axis. The mesh is created using topography from a low-centered ice wedge polygon near Prudhoe Bay, Alaska. Three soil materials (moss, peat, and mineral soil) and ice wedges are included.

The main steps are:

1 – Create an initial layer that reflects surface topography.

2 – Copy the layer several times, and stack the copies to create the soil profile.

3 – Create ice wedges.

4 – Define face sets and dump results into exodus file and ATS xml file.

5 – Partition the mesh for simulations run on a cluster.

1 – Create a base layer that reflects topography

A good way to get topographic data into the work environment is to import it from a csv file. In this example, horizontal distance (in meters) along the transect is loaded from the vector *s* and elevation (masl) is loaded from the vector *z* in the file *transectNWSE.csv*.

First, a flat mesh object in the x-y plane is created using PyLaGriT's *gridder* method. In the following lines, the object *lg* is created as an interface to the LaGriT program. The flat mesh object *top* is then created, with a width of 0.25 m in the y-direction.

```
from pylagrit import PyLaGriT
import numpy as np

lg = PyLaGriT()

# create base layer, with x matching s from the csv file
x = np.linspace(0., 29.75, 29.75/0.25 + 1)
y = [0., 0.25]
top = lg.gridder(x, y, elem_type='quad', connect=True)
```

If you are using a .pylagritrc file where the 'lagrit_exe' is defined, the syntax above will work (i.e., `lg = PyLaGriT()`). Otherwise, you will need to specify the location of your LaGriT executable as `lg = PyLaGriT(lagrit_exe=<path_to_your_lagrit_exe>)`.

By default, vectors specifying the spatial coordinates of the nodes in a mesh object are stored in the attributes *xic*, *yic*, and *zic*. In the following lines, a new attribute is created to store the original y-coordinates of *top*. All of the y-coordinates are then temporarily reassigned to zero, which will be important later when topography is projected.

```
top.addatt('y_save', type='vdouble', rank='scalar')
top.copyatt('yic', 'y_save')
top.setatt('yic', 0)
```

Next, a second mesh object, *surf_pts*, is created using the PyLaGriT method *points* and the elevation data from *transectNWSE.csv*. The z-coordinates in *zic* are saved in a copy, then reassigned to zero, so that the new mesh object overlaps with *top*.

```
# Read in top elevations
d = np.genfromtxt("transectNWSE.csv", delimiter=",", names=True)
surf_pts = lg.points(x=d['s'], z=d['z'], elem_type='quad')
surf_pts.addatt('z_save', type='vdouble', rank='scalar')
surf_pts.copyatt('zic', 'z_save')
surf_pts.setatt('zic', 0.)
```

The next lines demonstrate why the y- and z- coordinates of both mesh objects have been assigned to zero. The mesh object method *interpolate_voronoi* is used to copy elevation data from *surf_pts* to temporary storage in the attribute *z_val* in *top*. This step requires that the two mesh objects overlap – however, the node locations along the x-axis do not need to be identical, as they are in the example. The real y- and z- coordinates are then reassigned to *top*.

```
top.addatt('z_val', type='vdouble', rank='scalar')
top.interpolate_voronoi('z_val', surf_pts, 'z_save')
top.copyatt('y_save', 'yic')
top.copyatt('z_val', 'zic')
```

Finally, the materials of the nodes and the volumes in *top* are assigned the value *1*, representing moss. Node materials by default are stored in the attribute *imt*, while element materials are stored in *itetclr*. The information stored in *top* is dumped into the file *tmp_lay_peat_top.inp* and the mesh object *surf_pts* is deleted.

```
top.setatt('imt',1)
top.setatt('itetclr',1)
top.dump('tmp_lay_peat_top.inp')
surf_pts.delete()
```

2 – Copy the layer several times, and stack the copies to create the soil profile.

The work flow of the second step involves copying the mesh object *top*, and sequentially reassigning it to lower elevations. Each time it is assigned to a lower elevation, the data in the object are dumped into a new *inp* file. Each of these *inp* files represents an interface between layers of soil with either different thicknesses or different soil materials. Throughout the process, lists are created describing the number of cells and the soil material between each pair of interfaces, and the names of the interfaces themselves. These lists will be used to assemble all of the layers into a single mesh object.

First, *top* is copied to an identical object named *layer*, and the lists *stack_files* and *matids* are initialized with the name of the *inp* file and the material ID from the surface of the moss.

```
layer = top.copy()
stack_files = ['tmp_lay_peat_top.inp']
matids = [1]
```

Next, two cell layers of moss, each 1cm thick, are added just below the surface of the mesh. The mesh object method *math* is used to reassign *layer* to an elevation two centimeters below the surface—representing the lower interface of this zone—and the information from *layer* is dumped into the file *tmp_lay1.inp*. The name of this *inp* file is appended to the list *stack_files*, and the entry *1* is appended to *matids*, representing moss. Additionally, a new list, *nlayers*, is initialized with the value *1*. At each iteration, the value that will be appended to *nlayers* is equal to the number of cell layers in the newly defined zone minus 1.

```
# Add (2) 1 cm thick moss layers
layer.math('sub',0.01*2,'zic')
layer.dump('tmp_lay1.inp')
stack_files.append('tmp_lay1.inp')
nlayers = [1]
matids.append(1)
```

In the following lines, this process is repeated, but with six layers of a second material, peat, being added. Note the new material ID (2), and the new entry appended to nlayers (5). Each layer is 2 cm thick, meaning the mesh thus far is mantled with 14cm total of organic soil.

```
# Add (6) 2 cm thick peat layers
layer.math('sub',0.02*6,'zic')
layer.dump('tmp_lay2.inp')
stack_files.append('tmp_lay2.inp')
nlayers.append(5)
matids.append(2)
```

Next, a series of layers of increasing thickness is added below the organic soil. Each new group of layers is assigned the material ID *3*, representing mineral soil. Note that each zone of cells of a given thickness is separated from its neighbors by an *inp* file. Note also that the bottom *inp* file, representing the bottom boundary of the mesh, is assigned a flat elevation of 29 m (~50m below the ground surface), rather than reflecting topography.

```
# Add (6) 2 cm thick layers
layer.math('sub',0.02*6,'zic')
layer.dump('tmp_lay2.inp')
stack_files.append('tmp_lay2.inp')
nlayers.append(5)
matids.append(2)

# Add (8) 2 cm thick layers
```

```
layer.math('sub',0.02*8,'zic')
layer.dump('tmp_lay3.inp')
stack_files.append('tmp_lay3.inp')
nlayers.append(7)
matids.append(3)

# Add (15) 5 cm thick layers
layer.math('sub',0.05*15,'zic')
layer.dump('tmp_lay4.inp')
stack_files.append('tmp_lay4.inp')
nlayers.append(14)
matids.append(3)

# Add (15) 10 cm thick layers
layer.math('sub',0.1*15,'zic')
layer.dump('tmp_lay5.inp')
stack_files.append('tmp_lay5.inp')
nlayers.append(14)
matids.append(3)

# Add (15) 1 m thick layers
layer.math('sub',1*15,'zic')
layer.dump('tmp_lay6.inp')
stack_files.append('tmp_lay6.inp')
nlayers.append(14)
matids.append(3)

# Add (15) 2 m thick layers
layer.math('sub',2.*15.,'zic')
layer.dump('tmp_lay7.inp')
stack_files.append('tmp_lay7.inp')
nlayers.append(14)
matids.append(3)

# Add the bottom layer, and make the bottom boundary flat
layer.setatt('zic',29.)
layer.dump('tmp_lay_bot.inp')
stack_files.append('tmp_lay_bot.inp')
nlayers.append(0)
matids.append(3)
```

Finally, an empty mesh object, *stack*, is initialized, and the lists *stack_files*, *nlayers*, and *matids* are used as arguments in the mesh object method *stack_layers.* This method is used in combination with *stack_fill* to create a single mesh object of hexahedral volumes, with the correct material ID assigned to each layer. Importantly, for this step to work, each of the lists must be reversed before being passed to *stack_layers.*

```
stack_files.reverse()
nlayers.reverse()
matids.reverse()
stack = lg.create()
stack.stack_layers(stack_files,nlayers=nlayers,matids=matids,
flip_opt=True)
stack_hex = stack.stack_fill()
```

3. Create ice wedges

Thus far, a hexahedral mesh object representing the soil profile at the site has been created. Two ice wedges still need to be added, beneath the troughs of the polygon. To create them, two sets of points representing the ice wedges are created, then used as pointers to define which cells must have their material IDs reassigned.

The following python function creates a set of points for an ice wedge that in cross section is shaped like an upside down triangle. The triangle is defined by the x- and z-coordinates of its left, bottom, and right vertices. The points output by the function are spaced as in a grid, characterized by lengths dx and dz. Specifically, the x-, y-, and z- coordinates of two identical grids in the x-z plane, one at y=0.05 and another at y=0.2, are returned.

```
def iceWedgePoints( vtxL, vtxB, vtxR, dx, dz ):
    xnodes = np.arange( vtxL[0], vtxR[0], dx )
    znodes = np.arange( vtxB[1], vtxR[1], dz )
    xg, zg = np.meshgrid(xnodes, znodes)
    xg = xg.flatten(); zg = zg.flatten()
    m1 = (vtxB[1] - vtxL[1])/(vtxB[0] - vtxL[0])
    b1 = vtxL[1] - m1*vtxL[0]
    m2 = (vtxR[1] - vtxB[1])/(vtxR[0] - vtxB[0])
    b2 = vtxR[1] - m2*vtxR[0]
    idx = [ (zg[i] > m1*xg[i]+b1) & (zg[i] > m2*xg[i]+b2)  for i in
          range(len(zg)) ]
    iwx = xg[np.array(idx)]; iwz = zg[np.array(idx)]
    iwy = np.concatenate( (np.ones(iwx.size)*0.05, np.ones(iwx.size)*0.2) )
    iwx = np.tile(iwx, 2); iwz = np.tile(iwz, 2)
    return iwx, iwy, iwz
```

In the lines that follow, this function is used to define sets of points for two ice wedges. Note that the arguments vtxL, vtxB, and vtxR each comprise a coordinate pair passed to the function, and that dx and dz are chosen to be smaller than the horizontal and vertical dimensions of the cells which will be reassigned as ice.

For each ice wedge, the coordinates returned by the function are used as arguments in the PyLaGriT method *points* to define a new mesh object. The method *eltset_object*, from the mesh object *stack_hex*, is then called, using the new ice wedge mesh object as an argument. This method returns an element set (*i.e.*, a pointer to specific cells in *stack_hex*) that defines the

cells of the main mesh object intersecting the ice wedge. The pointer is then used to reassign the material ID of the ice wedge to *4*, representing ice.

```
iw1x, iw1y, iw1z = iceWedgePoints( [3.25, 78.3], [3.875, 75.0],
    [4.5, 78.3], 0.125, 0.01 )
iw1pts = lg.points(iw1x, iw1y, iw1z, connect=True)
iw1 = stack_hex.eltset_object(iw1pts)
iw1.setatt('itetclr', 4)

iw2x, iw2y, iw2z = iceWedgePoints( [18.5, 78.4], [19.125, 75.1],
    [19.75, 78.4], 0.125, 0.01 )
iw2pts = lg.points(iw2x, iw2y, iw2z, connect=True)
iw2 = stack_hex.eltset_object(iw2pts)
iw2.setatt('itetclr', 4)
```

4. Define face sets and dump results into Exodus file and ATS xml file

The mesh object *stack_hex* now contains cells representing three types of soil materials as well as ice wedges. Before it can be dumped to an Exodus file, the set of faces corresponding to each boundary of the model must be identified and labeled, so that boundary conditions may be applied in ATS. The mesh object method *create_boundary_facesets* performs this task by creating an *avs* file defining each boundary and returning an ordered dictionary (*fs* in the example) with their names.

```
fs = stack_hex.create_boundary_facesets(base_name='faceset_bounds',
     stacked_layers=True,reorder=True)
```

Note that the dictionary output by *create_boundary_facesets* lists the *avs* files defining the bottom, top, right, back, left, and front faces.

```
In [2]: fs
Out[2]:
OrderedDict([('bottom', faceset_bounds_bottom.avs),
             ('top', faceset_bounds_top.avs),
             ('right', faceset_bounds_right.avs),
             ('back', faceset_bounds_back.avs),
             ('left', faceset_bounds_left.avs),
             ('front', faceset_bounds_front.avs)])
```

The mesh object and face set information are now ready to be dumped into an Exodus II file, using the *dump_exo* method, with the names of the *avs* files defining the face sets passed as an argument.

```
stack_hex.dump_exo('transectNWSE.exo',facesets=fs.values())
```

Finally, the mesh object method *dump_ats_xml* is used to generate region specifications for the mesh, which can be pasted into the xml for an ATS simulation. In addition to the name of

the xml file to be generated and the path to the mesh file, the method takes dictionaries of the material IDs and the boundary names as arguments. Note in the following lines that four zeros are appended to the end of each material ID, and boundaries are listed in the same order as they appear in the ordered dictionary *fs*, although their exact names do not need to be preserved. Note also that the mesh file is specified by the relative path from the directory in which ATS will be run.

```
matnames = {10000:'computational domain moss',
            20000:'computational domain peat',
            30000:'computational domain upper mineral',
            40000:'computational domain ice wedge'}
facenames = {1:'bottom face',
             2:'surface',
             3:'front',
             4:'right',
             5:'back',
             6:'left'}

stack_hex.dump_ats_xml('transectNWSE_mesh.xml',
'../../mesh/transectNWSE.exo',matnames=matnames,facenames=facenames)
```

If you plan to partition your mesh to run on a cluster, you can call the *dump_ats_xml* method a second time, replacing the *.exo* filename passed as an argument with the base name you intend to give your partitioned mesh files (generally ending with *.par*). You may also wish to alter the path to a new directory where the *par* files will be stored.

```
stack_hex.dump_ats_xml('transectNWSE_parmesh.xml',
'../../mesh/4/transectNWSE.par',matnames=matnames,facenames=facenames)
```

5. Partition the mesh for simulations on a cluster

If you intend to partition your mesh, you will create an additional set of files, usually with a *.par* extension. Partitioning is performed outside of PyLaGriT and the python environment, using the utility *meshconvert* found under the *AMANZI_TPLS_DIR* environmental variable. If the *AMANZI_TPLS_DIR* is not defined on your system, it will be the location where you have installed the Amanzi third party libraries.

The utility is executed using *mpirun*, and the flag *–n* indicates the number of partitions created. The last two arguments indicate the name of the *.exo* file that will be partitioned, and the base of the *.par* files to be created.

```
$ mpirun -n 4 $AMANZI_TPLS_DIR/bin/meshconvert --partition=y
       --partition-method=2 transectNWSE.exo transectNWSE.par
```

The set of files output by *meshconvert* is numbered sequentially.

```
$ ls transectNWSE.par*
```

```
transectNWSE.par.4.0  transectNWSE.par.4.1  transectNWSE.par.4.2
transectNWSE.par.4.3
```

Meshing is now complete. To check your results, view your *.exo* file in Paraview, and compare with Figure 1 on the next page. After Paraview opens, hit the 'Apply' button on the left, and set the view direction with the z-axis pointing up and the x-axis pointing right.

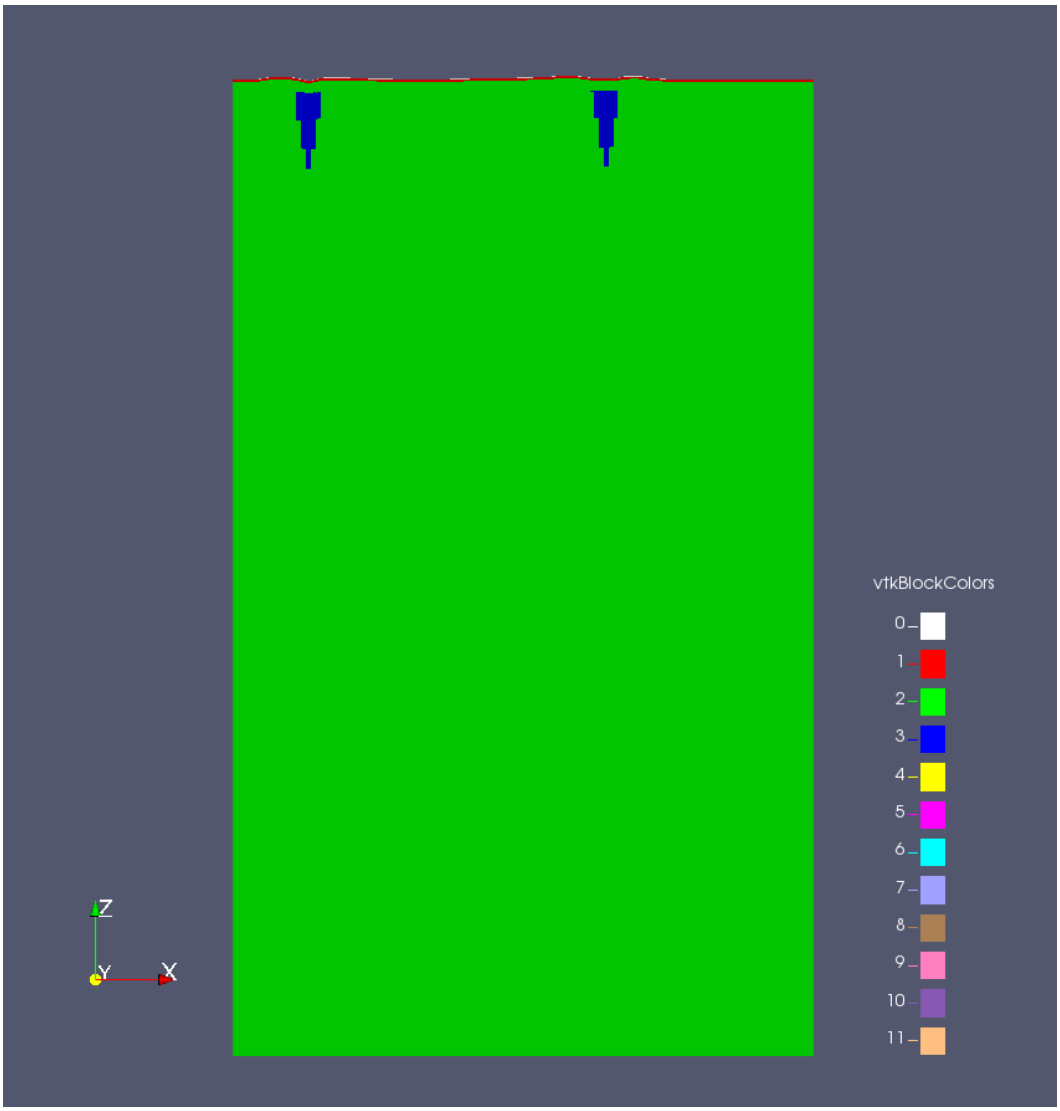

Figure 1. The mesh file *transectNWSE.exo*, seen in Paraview. Moss and peat layers are white and red bands near the surface. Mineral soil is green, and ice wedges are blue.