CubitPython4SPEED Manual

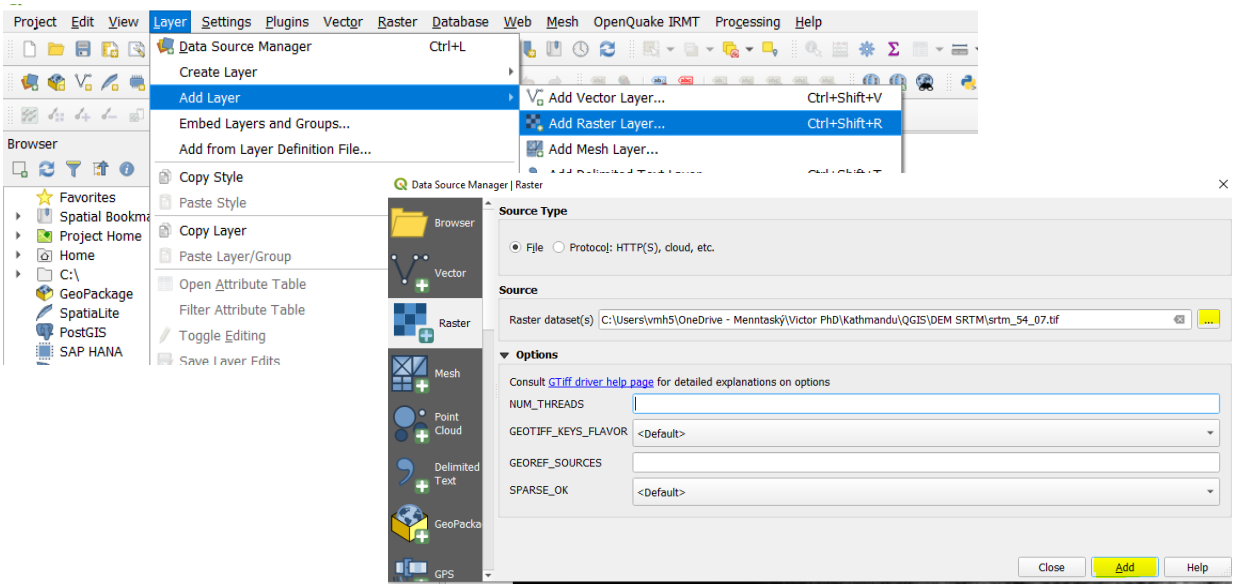
1 Topography (DEM)

The first step for setting up the model is to define the topography, if considered. You can download topography data (raster) from the SRTM DEM with 30 m (1 arc sec) resolution in the US and 90 m (3 arc sec) DEM in the rest of the world. It can be accessed from here: http://srtm.csi.cgiar.org/srtmdata/

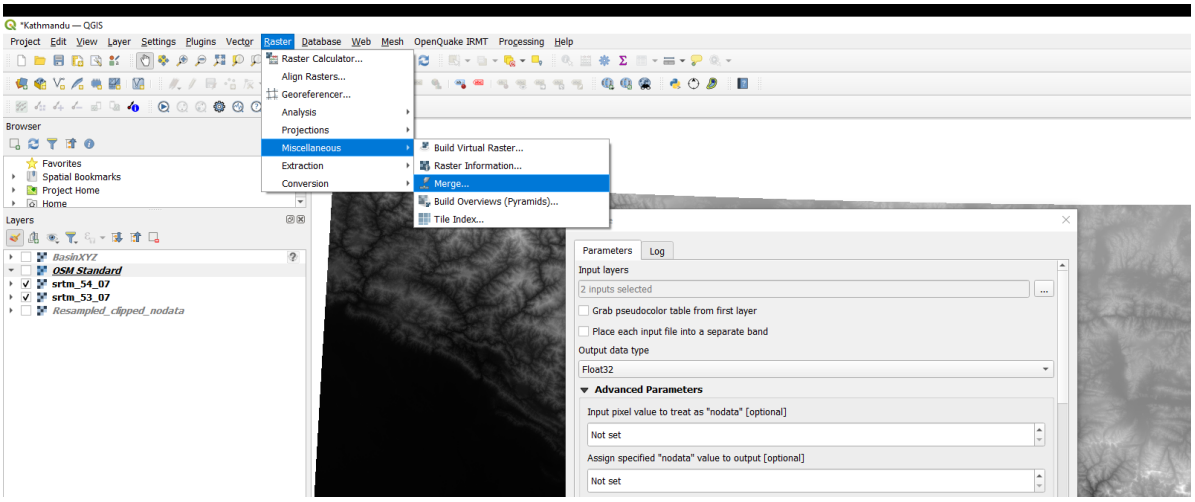
Depending on the country, local agencies might have more detailed models, e.g., (Landmælingar Íslands): https://atlas.lmi.is/mapview/?application=DEM for Iceland.

1.1 DEM manipulation in QGIS

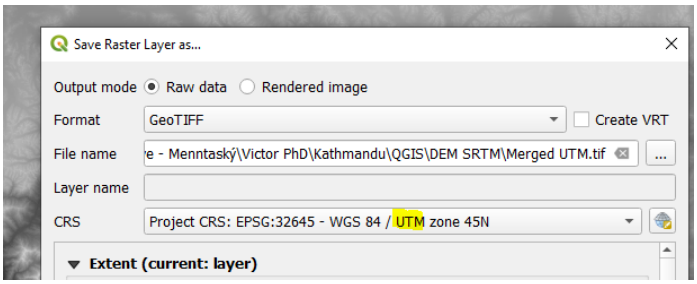
Import the DEM into QGIS: Layer-> Add Layer-> Select file-> Add



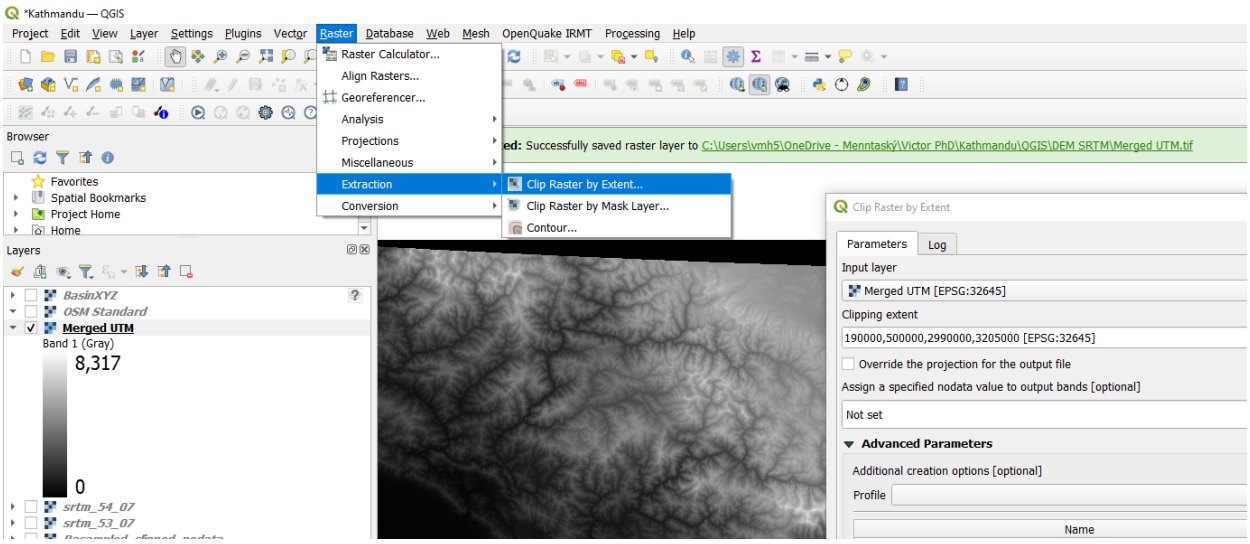
The whole region might be covered by more than one DEM file, in that case you need to merge them: Raster-> Miscellaneous-> Merge-> Select rasters -> Run



DEM from SRTM is in Geographical coordinates, but for SPEED with need projected coordinates (UTM), so you have to change its CRS (coordinate reference system):

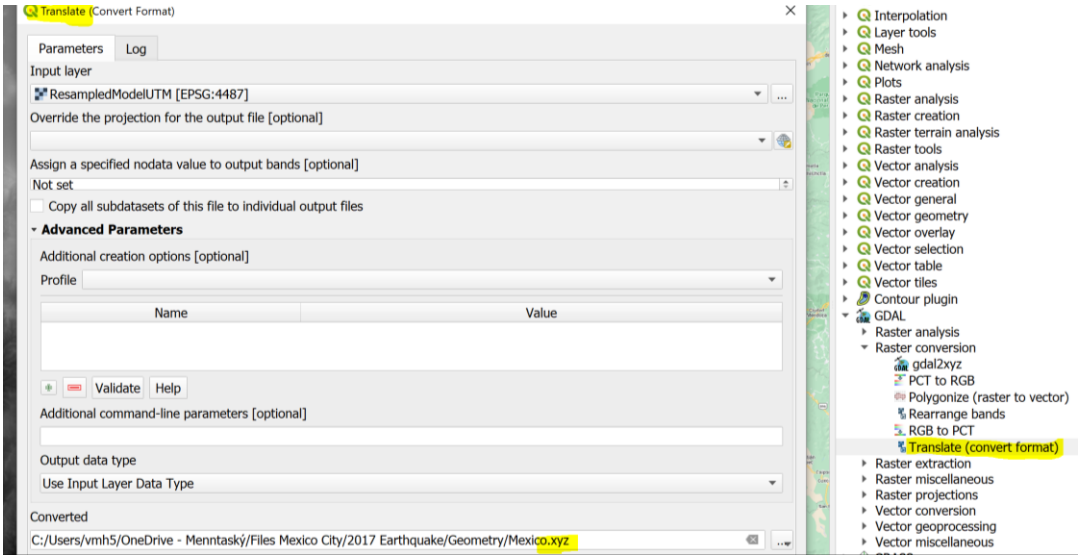


Clip the merged the DEM (take are of interest): Raster-> Extraction-> Clip Raster by Extent-> Select clipping extent -> Run



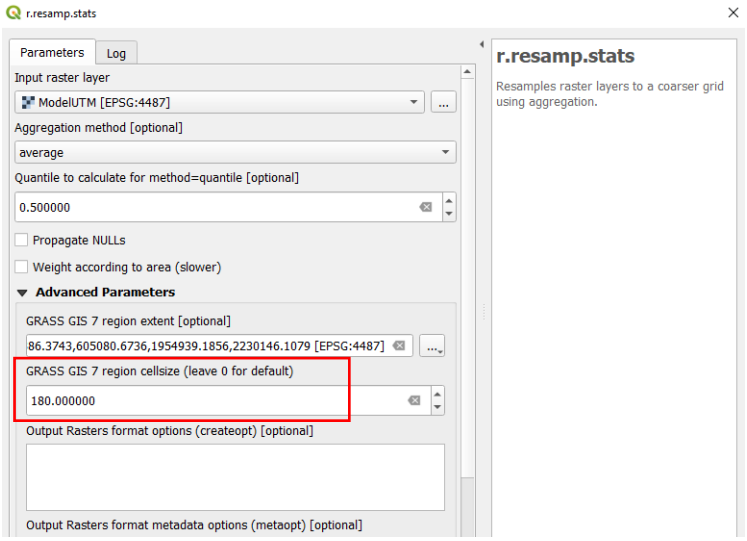
1.2 Export

Use the plugin GDAL-> Translate to export as XYZ (points):



No Data: Sometimes pixels of the raster don't have a value (No data), hence they are assigned a very large number in the Z coordinate. You should fix this before exporting, or post-process the output file to change these values.

NB: If the raster is too big to work with, you might need to reduce its size (resample) with GRASS-> r.resamp.stats (choose a bigger region cellsize):



1.3 Topography file for Cubit

The output from the plugin GDAL is a file ***.xyz** that contains the coordinates of the DEM, with one point (3 coordinates, x y and z) per row. It is ordered from West to East and from North to South. For reference, see the topo.dat file in the Files folder.

It is certainly not necessary to use QGIS to create the topography file, but if done somewhere else, it should have the characteristics mentioned in the previous paragraph (be an ordered grid).

2 CubitPython4SPEED

The meshing tool presented here is a modification of GEOCUBIT by Emanuele Casarotti (https://github.com/casarotti/GEOCUBIT--experimental/tree/master/geocubitlib). The routines have been modified to be compatible with Python3 and with the latest version of CUBIT. A free non-commercial license for CUBIT can be requested here: https://coreform.com/products/coreform-cubit/free-meshing-software/. Furthermore, it has been optimized for running in 1 CPU (GEOCUBIT is focused on parallel runs), and routines for the generation of the *input files* for SPEED are included.

The routines use the Python interface of the meshing software CUBIT. The interface provides an object-oriented structure that allows a developer to manipulate Cubit objects such as volumes, surfaces, etc. It also allows developers to create and manipulate as well as query geometry:

https://coreform.com/cubit_help/cubithelp.htm#t=python%2Fnamespace_cubit_interface.htm

Cubit must be included in the Environment variables – System variables – Path, e.g. : C:\Program Files\Coreform Cubit 2022.11\bin (according to your installation).

You should also install Python: https://www.python.org/downloads/

2.1 Configuration (input) file

The configuration cfg file has a format similar to ini windows files. It is divided in [sections]. There is no order in the position or in the name of the sections. The parameters that control the general options are in the first section, called [cubit.options] and [simulation.cpu_parameters]. For example (see *TopoExample.cfg* in the folder Files):

[cubit . options]

cubit_info =off turn on/ off the information of the Cubit command

echo_info =off turn on/ off the echo of the Cubit command

cubit_info =off turn on/ off the Cubit journaling

jou_info =off turn on/ off the Cubit journaling

working_dir =tmp set the working directory

output_dir = output set the output directory

geometry = True true for building the geometry (volumes)

onlysurface = True for creating just the top surface (topography)

meshing = True true if it meshes the geometry

merging = True true if it merges the meshed the geometry. By default, the routine creates vertical chunks that are disconnected from the rest of the chunks.

export_mesh = True true if it saves the mesh file in SPEED format.

disassociate = True true if after meshing each chunk it disassociates the mesh from the volumes and deletes the volumes. THIS SAVES A LOT OF RAM!

[simulation . cpu_parameters]

number_chunks_xi = 2 number of chunks in the X direction

number_chunks_eta = 2 number of chunks in the Y direction

start_chunk_xi = 0

start_chunk_eta = 2

end_chunk_xi = 0

end_chunk_eta = 2

The start and end chunk flags allow to work with the same *.xyz topography file, but use just a part of the domain to create the mesh or a surface mesh (e.g., the XYZ.out mesh covering just the basin area for using the not-honoring technique). By the default the start_chunk is set to 0, and the end_chunk equal to number_chunks.

```
[cubit.options]
cubit_info=on
echo_info=on
jou_info=off
jer_info=on
working_dir=tmp
output_dir=output
geometry = True
onlysurface = False
meshing = True
merging = True
disassociate = True
export_mesh = False

[simulation.cpu_parameters]
number_chunks_xi                    =2
number_chunks_eta                   =2
start_chunk_xi = 0
start_chunk_eta = 0
end_chunk_xi = 2
end_chunk_eta = 2

[geometry.volumes]
volume_type                 = layercake_from_ascii_regulargrid
latitude_min                = 10000
latitude_max                = 54000
longitude_min               = 5000
longitude_max               = 45000
nx = 5
ny = 5
unit                        = utm
# geo or utm

[geometry.volumes.layercake]
nz = 3
#included the bottom
bottomflat = True
depth_bottom = -10000
filename = Files\topo.dat,
geometry_format=ascii
zlayer = -10000,-6000
volumecreation_method = loft

[meshing]
map_meshing_type=regularmap
iv_interval=2,4
size=2000
or_mesh_scheme=map
ntripl=1
smoothing=False
tripl=3,
curve_refinement = False
curvename=Files/box.sat
curvename2=Files/curve.sat
num_curve_refinement = 2
block_firstlayer= True
```

[geometry.volumes]

latitude/longitude min and max are the limits of the topography grid file.

nx are the number of points in the grid file along the X direction.

ny are the number of points in the grid file along the Y direction.

unit = utm or **geo** are the units of the grid file and of the latitude/longitude values given above, utm units are preferred.

[geometry.volumes.layercake]

nz is the number of the surfaces in the volume (in this case: topography, one intermediate layer boundary and the bottom).

bottomflat is True if the bottom is flat.

depth_bottom is the depth of the bottom of the model.

filename is a list of files defining surfaces (in this case there is only the topography file).

geometry_format in this case is set to ascii since the definition of the surfaces comes from ASCII files (structured xyz points).

zlayer list with the depths of the interfaces from bottom to top, without including the topography

The [meshing] section contains the parameters requested for the meshing process:

map_meshing_type sets the meshing scheme and is regularmap by default (other schemes are in preliminary phase).

iv_interval sets the number of "horizontal" hex sheets for each layer. Since nz=3, we will have 2 layers vertically, so two values must be included here.

size is the dimension of the hexahedra (horizontally) **before refinement.**

or_mesh_scheme is the meshing scheme for the topography (map or pave are possible, see the Cubit manual for more information).

ntripl is the number of refinements (tripling) to be done.

tripl means in this case that the refinement layer is located at the third surface (the topography). The surfaces are ordered from the bottom (surface 1) to the top (surface 3). Keep in mind that after refinement the size of the hexahedra on top will be approximately **size/(3* ntripl).**

smoothing performs the smoothing command in Cubit.

block_firstlayer optional flag set by default to False. In case it's True, the uppermost sheet of hexahedra will be included in a different block.

When a **local** refinement (basin) is needed, the following lines are included in the input file

curve_refinement True for carrying out the refinement.

num_curve_refinement is the number of split iterations inside the curve.

curvename name of the **closed ACIS curve** inside of which the refinement will be done. This file must be present in the Files folder. The closed curve could be created in Rhino for example, using the polyline command:

```
Command: Polyline
Start of polyline ( PersistentClose=No ): 0,0,0
Next point of polyline ( PersistentClose=No  Mode=Line  Helpers=No  Undo ): 1000,0,0
Next point of polyline. Press Enter when done ( PersistentClose=No  Mode=Line  Helpers=No  Length  Undo ): 1000,1000,0
Next point of polyline. Press Enter when done ( PersistentClose=No  Close  Mode=Line  Helpers=No  Length  Undo ): 0,1000,0
Next point of polyline. Press Enter when done ( PersistentClose=No  Close  Mode=Line  Helpers=No  Length  Undo ): c
1 closed curve added to selection.
Command: _Export
Saved C:\Users\vmh5\Documents\CubitPython\Files\Box2.sat.
```

When the num_curve_refinement is set to 2, **curvename2** is also needed (it could the same as curvename). curvename2 should be contained inside curvename. In this case the first refinement will be done for the hexahedra inside curvename, and a second refinement will be done inside curvename2.

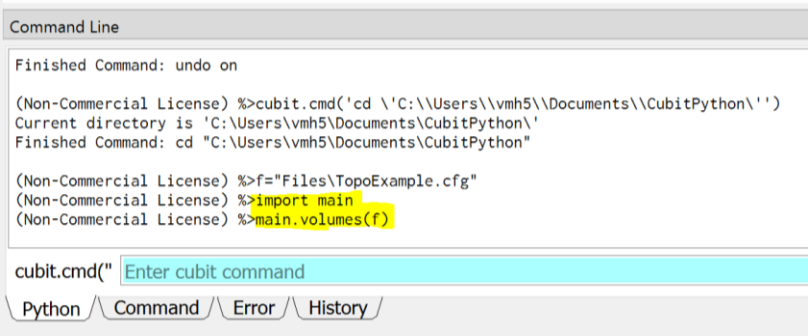
One important observation is that using the flag “disassociate” (RECOMMENDED), the use of RAM is drastically reduced. Disassociate the mesh means to keep just the meshed entities, hexahedral (elements) and quad (faces), without the underlying geometry. When using the flag merging, the mesh will always be disassociated, whether you used the flag disassociate or not.

3 Examples

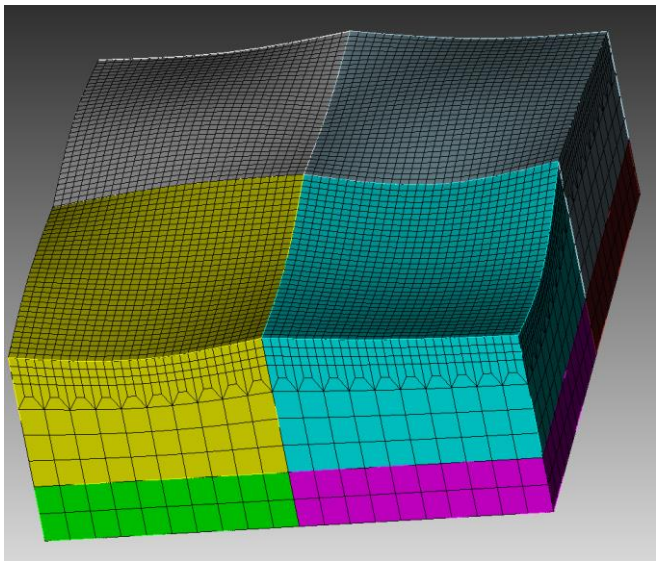
The input file for the first example is for the input file shown in the previous section: *TopoExample.cfg*

The meshing tool can be run directly from the Cubit GUI, you just have to type the following commands in the Python interpreter tab:

```
cubit.cmd('cd \'C:\\Users\\vmh5\\Documents\\CubitPython\\") <-location of the modules
f="Files/TopoExample.cfg" <-Input file located in the subfolder Files
import main
main.volumes(f) <- run main module
```

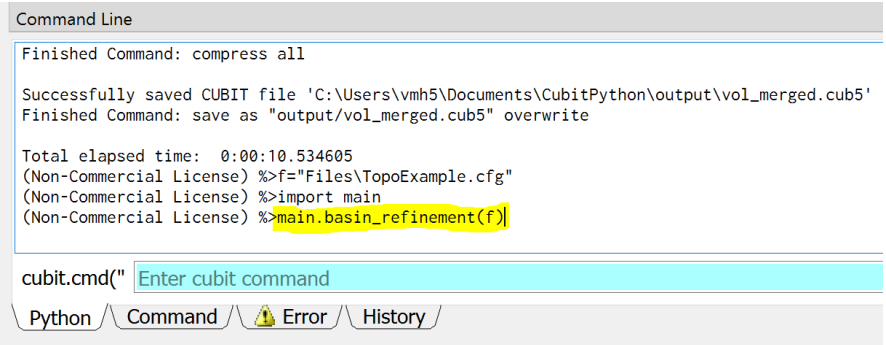


The output file is vol.cub5. If the flags merging and disassociate are deactivated, you should see the following output:

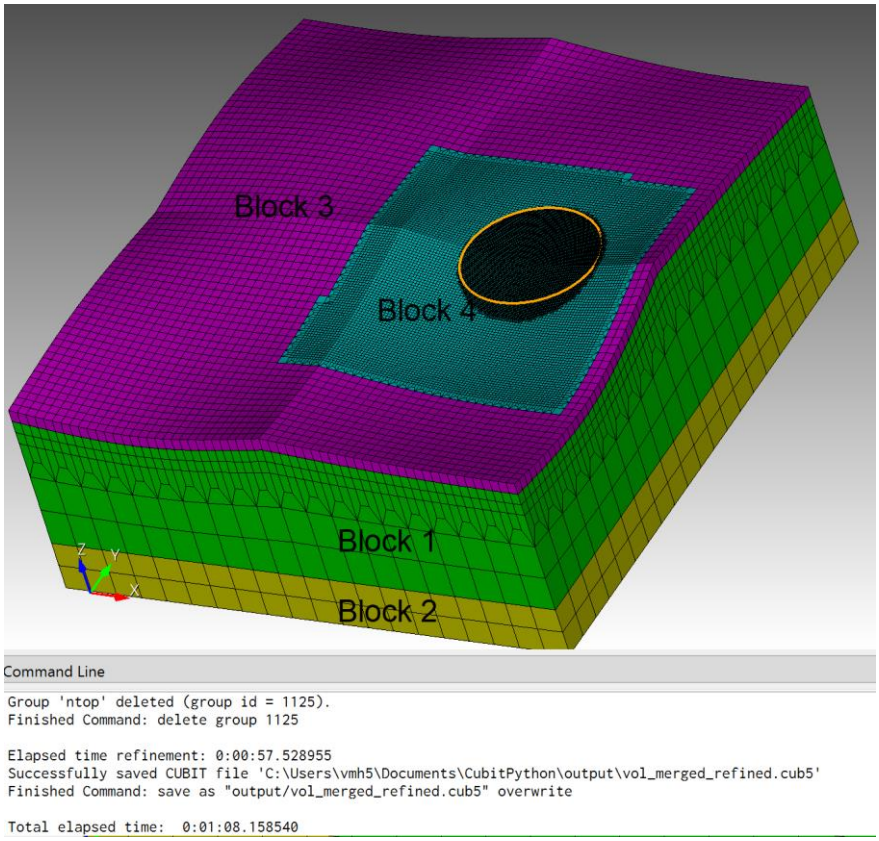


As you can see the domain was divided in 4 vertical chunks, 2 in the X direction and 2 in the Y direction.

When the flag merging=True, the output file vol.cub5 will be a free mesh disassociated from the volumes (whether you used the flag disassociate or not) with the same geometry but with the chunks merged (sharing nodes). Then, if you change the *curve_refinement* flag to true, you could run from the cubit GUI (having the vol_merged.cub5 file open) the basin_refinement routine separately:



The resulting file is *vol_merged_refined.cub5*, which is a free mesh with the necessary blocks for SPEED already defined. If you import box.sat and curve.sat (the input curves for the refinement), you will see that the local refinement was done for the hexes inside of them.



*The *curve_refinement* flag can be set to True from the beginning.

Blocks 1 to n represent the different n horizontal layers. They are numbered from top to bottom. In this case we have 2 since we included 3 interfaces (nz=3). Then, if the flag block_firstlayer= True, the next block (3 in this case) will be the first sheet of hexahedra. Finally, the next block (4

in this case) will contain the hexes that were refined. Usually, the basin will lie inside this last block, so the not-honoring technique should be applied to this block.

Block 100 includes the external faces for the absorbing boundary conditions.

Running from command prompt

This is more efficient and recommended for large meshes. You have to run a script similar to this one (*ExampleRunCubitPython.py*):

```
C: > Users > vmh5 > Documents > CubitPython > ExampleRunCubitPython.py > ...
1  import cubit
2  cubit.init(['cubit','-nojournal'])
3  cubit.reset()
4
5  cubit.cmd('cd \'C:\\Users\\vmh5\\Documents\\CubitPython\'')
6  f="Files\TopoExample.cfg"
7  # f="Files\Regmesh.cfg"
8  import main
9  main.volumes(f)
```

The first lines initialize Cubit, and the following lines are the same typed in the GUI. To run the script you simply change the directory to the one where the script is located, and then you run it with the command *python*:

```
C:\Users\vmh5>cd C:\Users\vmh5\Documents\CubitPython

C:\Users\vmh5\Documents\CubitPython>python ExampleRunCubitPython.py
[2023-11-01 11:45:56.594] [info] RLM session initialized
Creating new Workspace session
Initializing LLVM
Loading workspace command plugin
Starting volume creation


                 CCCCC    UU   UU   BBBBBB    IIII   TTTTTT
                CC   CC   UU   UU   BB   BB    II       TT
                CC        UU   UU   BB   BB    II       TT
                CC        UU   UU   BBBBBB     II       TT
                CC        UU   UU   BB   BB    II       TT
                CC   CC   UU   UU   BB   BB    II       TT
                 CCCCC     UUUUU    BBBBBB    IIII     TT  tm

                          MESH GENERATION TOOLKIT
```

Since the basin refinement is done after disassociation and merging, the refinement doesn't have to follow the smooth ACIS topographic surface, making the operation extremely faster. Each hex is simply split internally.

Regular mesh

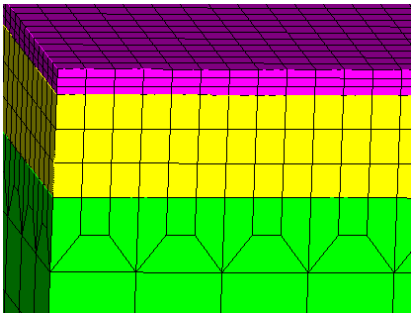
In the example *RegmeshExample.cfg*, a simple regular mesh for a "box" with flat interfaces is created. The flag for this case is **geometry_format=regmesh**. There are 3 layers with interfaces at depths defined by *zlayer=-10000,-3000,-600,0*. In this case **zlayer** has the same size of ***nz***, because it must include the depth of the uppermost interface. The initial mesh has hexes with horizontal size=2000 (before refinement) and the number of vertical hex sheets are *iv_interval=4,1,1* for the bottom, middle and top layers, respectively. A refinement layer (*ntripl=1*) is included at *tripl=2* interface (the second from the bottom). Since *iv_interval* refers to the intervals before refinements, after the refinement is done there will be 3 sheets of hexes in

the first layer of 600 m, and the vertical dimensions of these hexes will be 200 m. The flag *coarsening_top_layer=True* remeshes the top layer and the number of vertical hex sheet in the vertical is changed by *actual_vertical_interval_top_layer* (=1 in this case).

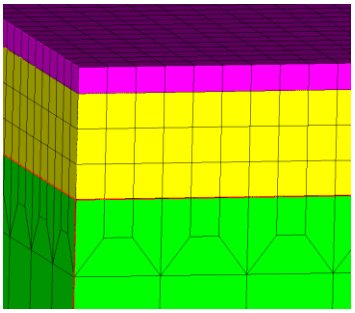
```
[simulation.cpu_parameters]
number_chunks_xi                        =1
number_chunks_eta                       =1
#
[geometry.volumes]
volume_type                     = layercake_from_ascii_regulargrid
latitude_min                    = 0
latitude_max                    = 20000
longitude_min                   = 0
longitude_max                   = 20000
unit                            = utm
# geo or utm

[geometry.volumes.layercake]
nz = 4
#included the bottom
bottomflat = True
depth_bottom = -10000
geometry_format=regmesh
zlayer=-10000,-3000,-600,0

[meshing]
map_meshing_type=regularmap
iv_interval=4,1,1
size=2000
or_mesh_scheme=map
ntripl=1
tripl=2,
smoothing=False
coarsening_top_layer= True
actual_vertical_interval_top_layer=1
```



No flag coarsening_top_layer



flag coarsening_top_layer=True

For these simple models (regmesh) there is no need to subdivide the domain in chunks, use 1 for *number_chunks*.

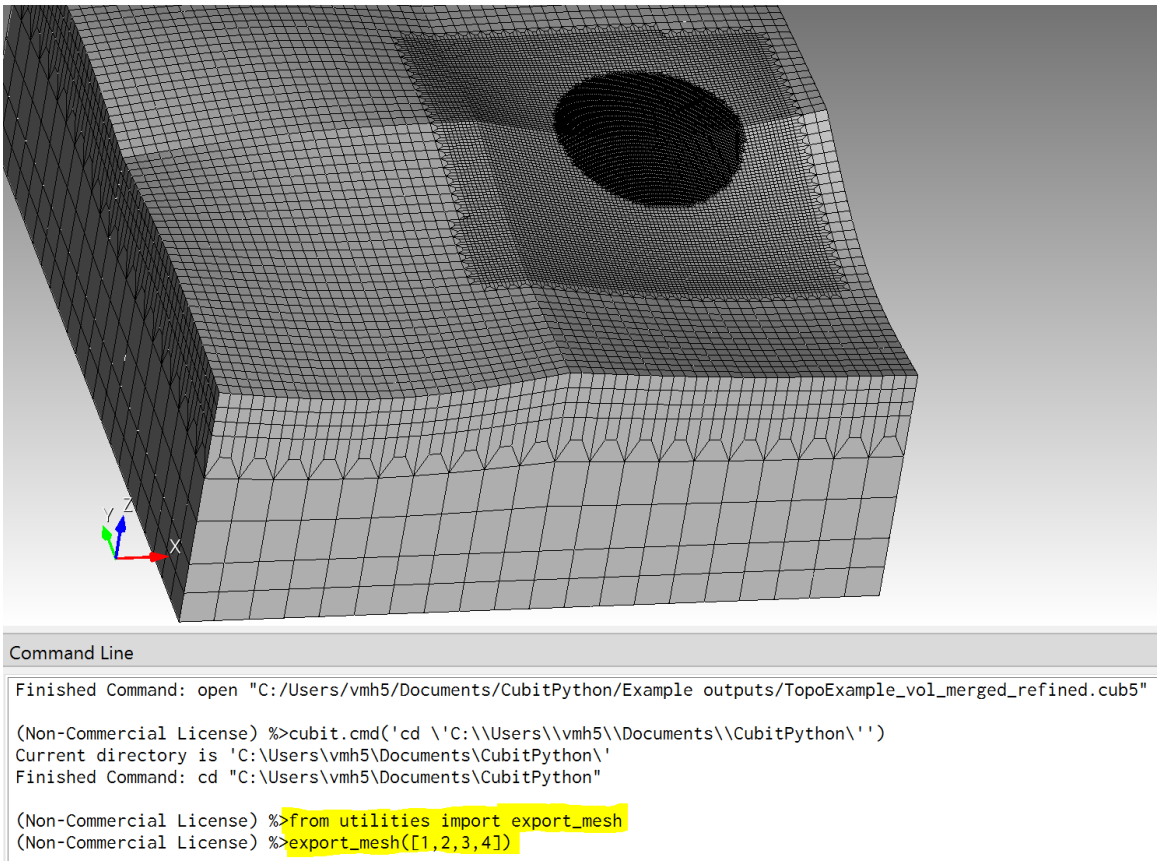
Running individual routines

You can run individual routines, such as merging, export_mesh, basin_refinement, etc. These routines are located either in the *utilities.py* or *main.py* files. You could for instance create the domain without basin refinement first (curve_refinement = False), and then test different options for refinement of the basin (curve_refinement = True in the cfg file), as it was shown before. See the example script: *ExampleRefinement.py*.

3.1 SPEED input files

Mesh file *.mesh

To get the *.mesh file you could either include the flag export_mesh=True, or use separately the routine *export_mesh([list hexahedral blocks])*, which is located in the *utilities* file:



The only requirement is that the hexahedral blocks were created using the command ***block n add hex in volume N***, instead of the classical command ~~***block n vol N***~~, where N is the number of the volume(s) to include in the block *n*. Moreover, the block for the ABC (absorbing boundary condition) should be created using the command ***block 100 add face in surface N***, where N are the numbers of the surfaces to include. This is already taken care of when using this tool to create the mesh. The output is a file named Meshfile.mesh.

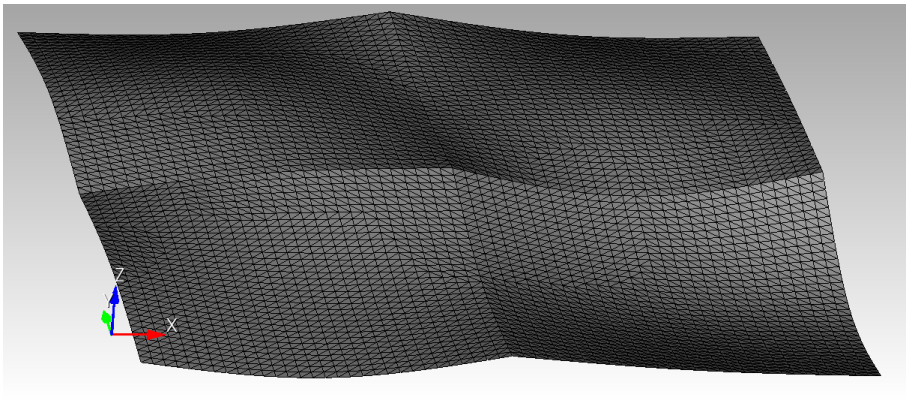
Topography file XYZ.out

When using SPEED's honoring technique, you will need a triangular mesh (XYZ.out file) of the part of the topographic surface covering your basin mesh. You can create it from the same topography file *.xyz using the flag **onlysurface = True**. If you use the same *number_chunks*, *size* and *ntripl* as for the hexahedral mesh, the resulting surface mesh will be very similar to the upper boundary of the hexahedral mesh. The approximate size of the triangle edges will be *size*/*ntripl*.

See the file *TopoSurfaceExample.cfg*. The flag export_mesh is already set to True, so when you run it you will get the XYZ.out file.

Since this mesh doesn't necessarily have to cover the whole domain, but just the part of the topographic surface covering your basin surface mesh (ALL.out), you could use the flags *start_chunk* and *end_chunk* to create the surface just for the part you are interested in.

The routine used to export the surface mesh is *export_surfacemesh(n)*, which is located in the *utilities* file. You could also use this routine separately:




```
Command Line
(Non-Commercial License) %>cubit.cmd('cd \'C:\\Users\\vmh5\\Documents\\CubitPython\'')
Current directory is 'C:\Users\vmh5\Documents\CubitPython\'
Finished Command: cd "C:\Users\vmh5\Documents\CubitPython"

(Non-Commercial License) %>from utilities import export_surfacemesh
(Non-Commercial License) %>export_surfacemesh(1)
exporting SPEED surface file...
```

Where the *(n)* is the block number of the mesh (1 in this case). The only requirement is that the block was created using the command ***block n add tri in surface N***, instead of the classical command ~~*block n surface N*~~, where N is the number of the surface.

The same individual routine could be used to export the basin **ALL.out** surface mesh. By default the name of the output file is XYZ.out, so you just need to rename it.

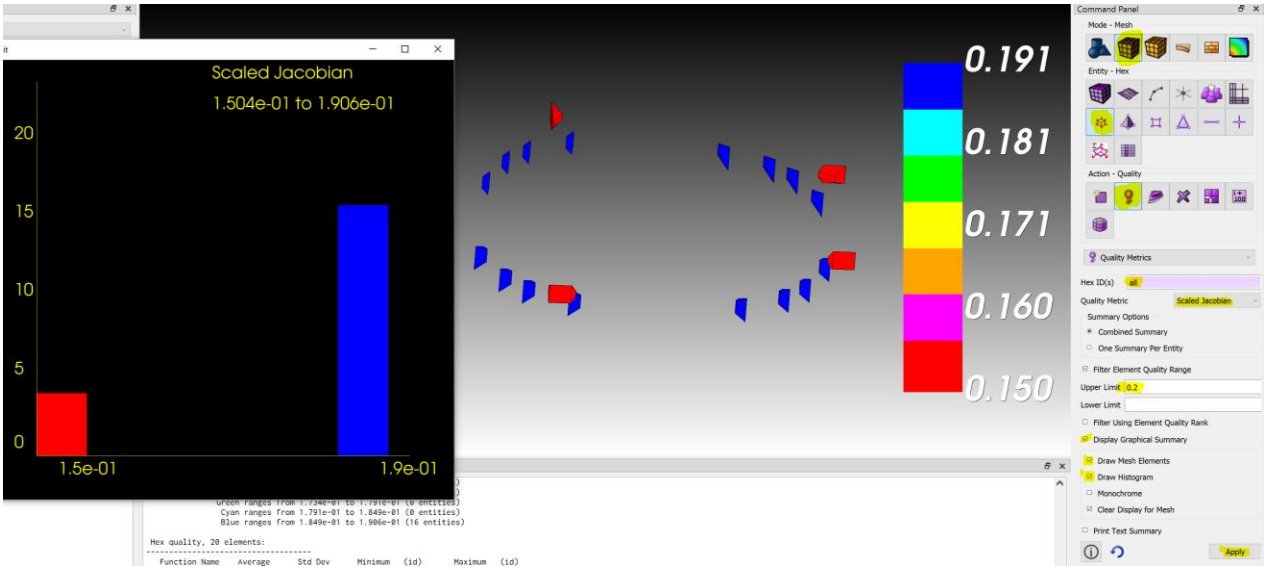
Monitors file LS.input

This file contains the coordinates of the monitor points where you would like to export the results from SPEED. It is common to export results at the ground surface, or for a vertical cross section. The flag **onlysurface = True** again could be used to create a mesh at the ground surface, and the nodes of this mesh taken as monitors. This surface could also be locally refined to increase the density of monitors in a certain region, see the file *TopoSurfaceLSExample.cfg*. Compared to TopoSurfaceExample.cfg, the flag export_ls was added, as well as the commands for the local refinement.

The routine used to export the surface mesh, located in the utilities file, is *export_LS(n,istart)*, where the *n* is the block number of the triangular mesh and *istart* is the starting numbering of the monitors (by default =1). *istart* is useful in the case you want monitors in more than 1 surface (e.g., ground surface + cross-section), so you can export different surfaces (with consecutive numbering) and then combine them in one file. You could also use this routine separately as in the previous examples.

3.2 Mesh quality

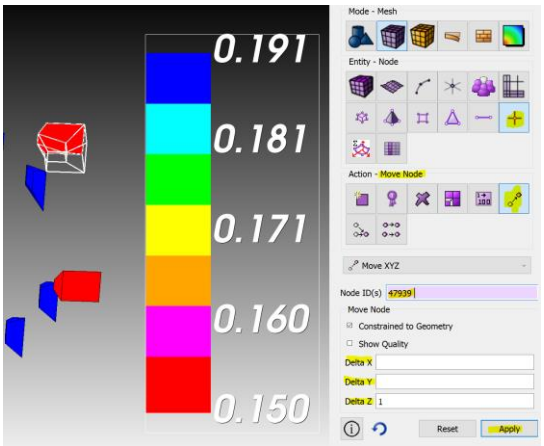
It's important to always check the mesh quality, you can do this from the Cubit GUI:



You can use the buttons or directly input a command similar to this one: *quality hex all scaled jacobian global high 0.2 draw histogram draw mesh*

Ideally all the hexes should have a scaled Jacobian at least larger than 0.2 to ensure numerical stability. If there are hexes with low quality and smoothing doesn't improve the mesh quality, an option is to manually move nodes trying to improve the shape of those hexes (closer to a cube):

*Meshes with elements with scaled Jacobians lower than 0.2 might still be stable. Depends on the specific case.



The command is: *node <list of nodes> move x <val> y <val> z <val>*