GEOCUBIT USER MANUAL

Emanuele Casarotti

February 10, 2011

Contents

1 WHY

GEOCUBIT is a python library wrapping around the CUBIT Python Interface and it aims to facilitate the meshing process in some common problems in seismic wave propagation.
In particular, it is focused on the meshing requests of SPECFEM3D and it is helpful for some tedious tasks as:

- Creation of geophysical surfaces and volumes (ex. topography).
- Mesh of layered volumes with hexahedral.
- Creation of an anisotropic mesh suitable for cases where some alluvial basin (or slow velocity zones) are present.
- It can be used as serial or parallel process. The parallel meshing capabilities are fundamental for large geophysical problems (ex. mesh of Southern California using SRTM topography).

GEOCUBIT can be use inside the graphical interface of CUBIT (i.e. as python object in the script tab) or as unix command. We refer to the CUBIT help appendix for more information about the python interface.

2 FIRST STEPS

2.1 Requirements

The minimum requirements for using GEOCUBIT are:

- CUBIT 12.2
- numpy 1.0+
- python 2.5 (strictly! it depends on the cubit library that refers to this version of python)

and for using the parallel meshing capabilities:

- pyMPI

2.2 Installation

For installing, download the code and type in the GEOCUBIT directory:

```
python2.5 setup.py install
```

Check that the following variables are set:

```
CUBITDIR=/usr/local/CUBIT
CUBITLIB=$CUBITDIR/bin:$CUBITDIR/structure:$CUBITDIR/components
PYTHONPATH=$CUBITDIR/components/cubit:$CUBITDIR/structure:$CUBITDIR/bin
LD_LIBRARY_PATH=$CUBITDIR/bin
PATH=$CUBITDIR/bin:$CUBITDIR/components/cubit
```

3 USING GEOCUBIT AT COMMAND LINE:

3.1 UTILITIES

- checking the configuration of the libraries and dependencies:

```
GEOCUBIT.py -chklib
```

- checking the parameter file:

```
GEOCUBIT.py -chkcfg -cfg=[file]
```

3.2 CREATE GEOMETRY

SURFACES

- creating acis surfaces using a parameter file:

```
GEOCUBIT.py -build_surface -cfg=[filename]
```

- creating a surface from regular ascii grid or ascii lines that define a "skin":

```
GEOCUBIT.py -surface=[file] (-regulargrid=[opts] | -skin=[opts])
```

VOLUMES

- **serial** command: creating cubit volumes using a parameter file:

```
GEOCUBIT.py -surface=[file] (-regulargrid=[opts] | -skin=[opts])
```

- **parallel** command: creating cubit volumes using a parameter file:

```
mpirun -n [numproc] pyMPI GEOCUBIT.py -build_volume -cfg=[file]
```

  In this parallel application, the volume is separated in N slices, each one is assigned at a single process and one file for each slice is created (see Figure 1 as example).

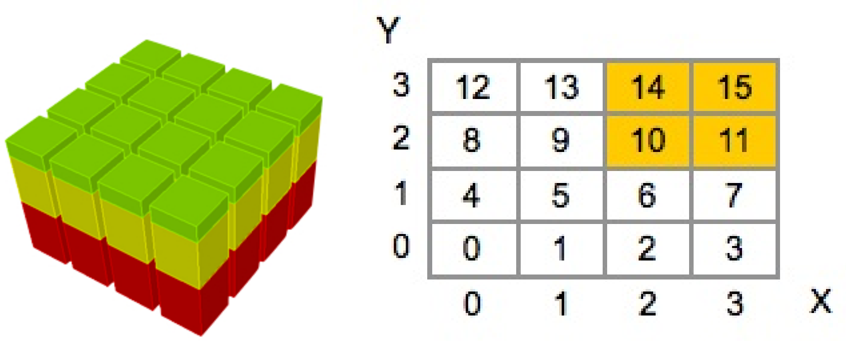
3.3 MESHING

- **serial** command: building a volume and meshing it.

```
GEOCUBIT.py -build_volume -mesh -cfg=[file] (-id_proc=n)
```

- **serial** command : meshing a volume

```
GEOCUBIT.py -mesh -cfg=[file] (-id_proc=[n, def=0])
```



(a) (b)

Figure 1: a) Volume divided in slices. b) Parallel map of the IDs for volume in Figure 1(a) divided in (16 slices, 4 along x-axis and 4 along y-axis).

note: without the –build_volume flag the script recall an old "geometry_vol_[id_proc].cub" file

- **parallel** command: meshing a volume from a parameter file

```
mpirun -n [nproc] pyMPI GEOCUBIT.py -build_volume -mesh
-cfg=[file]
```

3.4 FINALIZING AND EXPORTING

It is possible to collect, merge, set the absorbing boundary conditions and to export the resulting mesh in a format readable by SPECFEM3D.

- Collecting some cubit files, setting the absorbing boundary conditions and merging in a single free mesh cubitfile

```
GEOCUBIT.py -collect -merge -meshfiles=[files] -cpux=N -cpuy=N
(-rangecpux=[cpuxmin,cpuxmax] -rangecpuy=[cpuymin,cpuymax])
```

  `cpuy, cpux` set the number of cpus used for creating the mesh files; `rangecpux` and `rangecpuy` set the range of slices that are used in the mesh.
  Following the example in Figure 1, the parameters `-cpux=4 -cpuy=4 -rangecpux=1,2 -rangecpuy=2,3` select the slices with id 10 11 14 15. Only these slices are going to be collect and merged with the appropriated absorbing boundary conditions.

- Collecting a single free mesh cubitfile and refine the hex inside some curves (ex. alluvial basin, Figure 2)

```
GEOCUBIT.py -collect -meshfiles=[list of files]
-curverefining=[file]
```

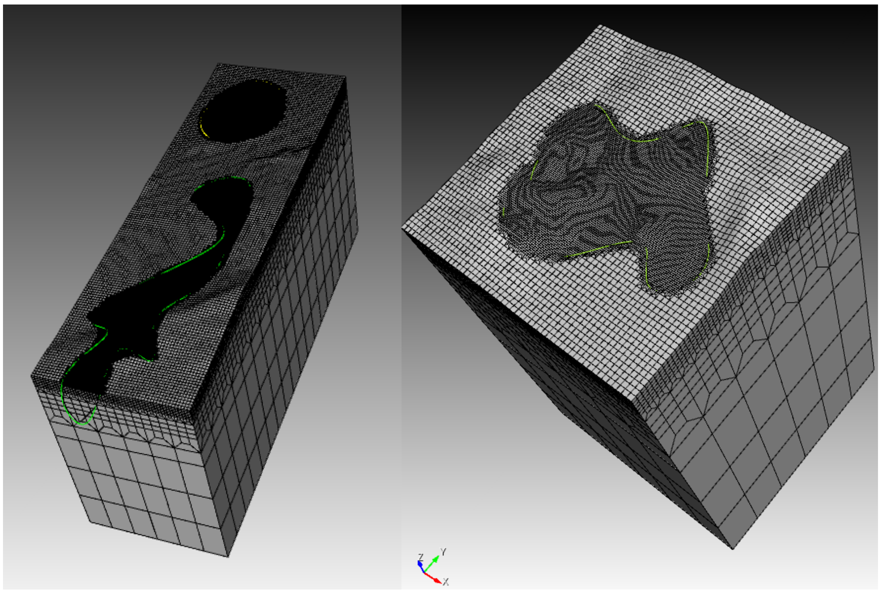


Figure 2:

note: the curves must be stored in acis format (sat) and must be closed.

- Exporting a cubit mesh file in a SPECFEM3D mesh

```
GEOCUBIT.py -export2SPECFEM3D=[output directory]
-meshfiles=[filename] (-listblock=[list of cubit blocks]
-listflag=[list of specfem3d flags])
```

If required, it is possible to assign personalized flags to each block using –listblock and the correspondent listflag (for example: –listblock=3,4 –listflag=1,-1)

4 GEOCUBIT and CUBIT GRAPHICAL INTERFACE

In the python script tab of CUBIT GUI, type:

`>f="volume.cfg"` → *store the name of the configuration file, see next section*

`>from geocubitlib import volumes` → *load the geocubit modules*
`>from geocubitlib import mesh_volume`
`>from geocubitlib import exportlib`

`volumes.volumes(f)` → *create the volumes, the parameters are stored in the cfg file*
`mesh_volume.mesh(f)` → *mesh the volume*

`exportlib.collect()` → *set the boundary conditions, see SPECFEM3D manual*
`exportlib.e2SEM(outdir="./output")` → *save the mesh in the SPECFEM3D format*

See the media/iterative.mov for a live example.

5 EXAMPLES of CONFIGURATION FILES

In the current section a few configuration files will be described. Please, see the files in the examples directory of the GEOCUBIT distribution for reference.

5.1 GENERAL OPTIONS

A cfg file has a format similar to ini windows files. They are divided in `[sections]`. There is no order in the position or in the name of the sections.
Generally, the parameters that control the general options are in the first section, called `[cubit.options]` and `[simulation.cpu_parameters]`. For example:

`[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*
`echo_info=off` → *turn on/off the cubit error journaling*
`working_dir=tmp` → *set the working directory*
`output_dir=output` → *set the output directory*
`save_geometry_cubit = True` → *true if it saves the geometry files*
`save_surface_cubit = False` → *true if it saves the surfaces in a cubit files*
`export_exodus_mesh = True` → *true if it saves the mesh also in exodus format. The default is the cubit format (cub), that contains both the geometry and the meshing information. In case of complex geometry, the cub file could be enormous and a more light exodus file become important.*

[simulation.cpu_parameters]
nodes = 1 → *number of nodes/process*

note: Usually the *_info options are turn off by default for improving the performances.

5.2 CREATING A TOPOGRAPHIC SURFACE

See stromboli.cfg, execute with: `GEOCUBIT.py --build\_surface --cfg=./example/stromboli.cfg.`

GEOCUBIT is able to create a topographic surface based upon the CUBIT command **create skin curve** and **create surface net** (Figure 3). See the CUBIT manual for details.

[geometry.surfaces] nsurf = 2 → *number of surfaces that will be created*

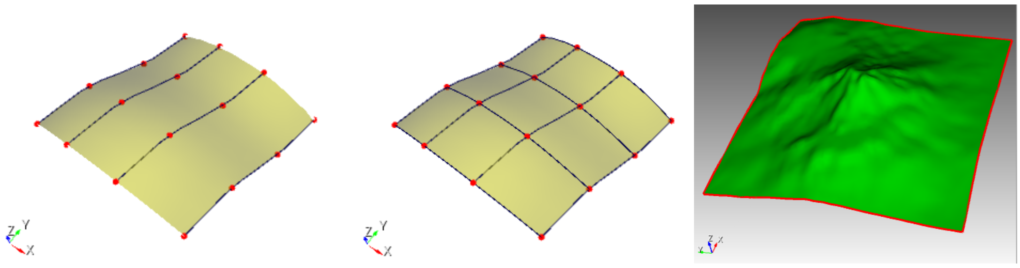
[surface1.parameters]
name=example/data/stromboli.xyz → *the name of the file defining the surface*
surf_type=skin → *the type of surface:* skin *, the file is a sequence of parallel points that span the surface in the order describer by the following direction parameters* regular_grid *the points are structured distritibuted. see Cubit Manual for details(skin surface and net surface)*
unit_surf= utm → *unit of the points,* utm *or* geo *(geographical) coordinates*
directionx = 0 → *if* surf_type=skin*:* directionx *and* directiony *define how the points span the surface*
directiony = 1
step = 1 → *the script creates a vertex each* step *points: in this case* step=1 *means a vertex for all the points*

[surface2.parameters] → *note the different number of surface in the section name*
name=./examples/surfaces/topo.dat → *the name of the file defining the surface*
surf_type=regular_grid → *the type of surface:* skin *, the file is a sequence of parallel points that span the surface in the order describer by the following direction parameters* regular_grid *the points are structured distritibuted. See see Cubit Manual for details (skin surface and net surface)*
unit_surf= geo → *unit of the points,* utm *or* geo *(geographical) coordinates*
nx=5 → *In case of* regular_grid: nx *and* ny *are the number of points along x and y direction.*
ny=5

5.3 CREATING and MESHING A VOLUME

see volume.cfg, for executioning: `GEOCUBIT.py --build\_volume --mesh --cfg=./example/volume.cfg`

The example is for a volume with topography with dimension of the hexahedra increasing with depth by means of a refinement layer (Figure 4).



(a) (b) (c)

Figure 3: GEOCUBIT creates a topographic surface in 2 ways: a) skin topography, `directionx=0` and `directiony=1`, b) topography from a net of structured distritibuted points, `nx=4` and `ny=4`, c) Topography of Stromboli volcano (Italy) using the net surface method.

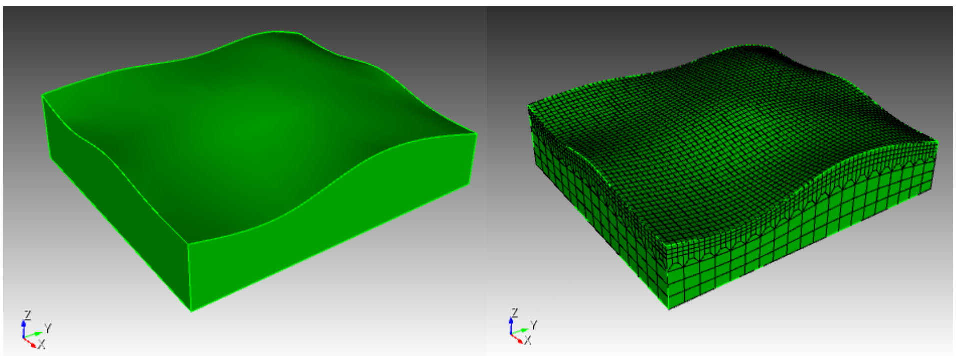


Figure 4: Mesh of a simple volume

```
[geometry.volumes]
volume_type = layercake_volume_ascii_regulargrid_regularmap
latitude_min = 13.879
latitude_max = 14.279
longitude_min = 40.619
longitude_max = 40.969
nx = 5
ny = 5
unit = geo

[geometry.volumes.layercake]
nz = 2
bottomflat = True
depth_bottom = -7000
filename = ./example/topo.dat
geometry_format=ascii
```

`nz` is the number of the horizontal surfaces in the volume (in this case: topography and bottom).
`bottomflat` is True if the bottom is flat.
`depth_bottom` is the depth of the bottom.
`filename` is a list of files. Each one defines a surface (in this case there is only the topography file in a regular_grid format).
`geometry_format` is set to ascii since the definition of the surfaces comes from ascii files (structured xyz points).

```
[meshing]
map_meshing_type=regularmap
iv_interval=5, → if only one value is present, append the comma
size=2000
or_mesh_scheme=map
ntripl=1
tripl=2, → if only one value is present, append the comma
smoothing=False
```

The `meshing` section contains the parameters request for the meshing process.
`map_meshing_type` set the meshing scheme and is regularmap by default (other schemes are in preliminary phase).
`iv_interval` set the number of "horizontal" hex sheets for each layer. `size` is the dimension of hexahedral (horizontally)
`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 tripling layer in the mesh (in this case 1).
`tripl` means in this case that the the refinement layer is located at the second surface (the topography). The surfaces are ordered from the bottom (surface 1) to the top (surface 2).
`smoothing` performes the smoothing command in Cubit.

5.4 LAYERED VOLUME and MESH FOR CENTRAL ITALY (PARALLEL)

In the example abruzzo.cfg, the layers are 2. For execution type: `mpirun -n 150 GEOCUBIT.py -build_volume -mesh -cfg=./example/abruzzo.cfg`.

Comparing the previous cfg files, there are few modification:

```
...
nz = 3
...
filename = example/data/moho_int.xyz,example/data/topo.xyz
...
iv_interval=8,8 (one value for each layer)
...
refinement_depth=1,
```

The volume has `nz=3` interfaces: bottom, moho, and topography. The bottom is flat and his z-coordinate position is set by `depth_bottom`. The name of the files that storage the data for the other interfaces are listed in `filename`. The interfaces defines 2 geological layers. Each layer has `iv_interval=8` "horizontal" hex sheets. There is only a refinement and it is set in the `refinement_depth=1` hex sheet, i.e just below the topography (`refinement_depth=8` means just below the moho).

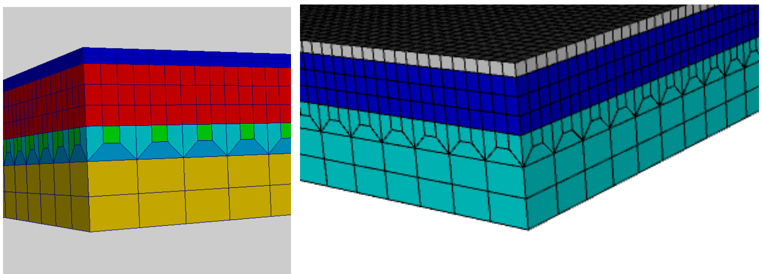
5.5 CREATION OF A REGULAR MESH

In the example grid.cfg, the layers are 3. For execution type: `GEOCUBIT.py -build_volume -mesh -cfg=./example/grid.cfg`.

In this example we create a simple layercake box `geometry_format=regmesh` with 3 layers at depth defined by `zdepth=-7000,-3000,-600,0`. The initial mesh has hex with horizontal size `size=2000` and the numbers of vertical hex sheets is iv_interval=3,1,1 respectively for the bottom, middle and top layer. We include a refinement layer (`ntripl=1`) at `tripl=2,` interface (the second from the bottom). Since the refinement occurs on the vertical direction, the vertical dimensions of the hexes in the top layer is too small and the quality of the mesh decreases. `coarsening_top_layer=True` remesh the top layer and the number of vertical hex sheet in the vertical is defined by `actual_vertical_interval_top_layer=1`. (see Figure 5)

5.6 TODO: MESH for SOUTHERN CALIFORNIA

Mesh with high number of hexes and high resolution topography. Mesh in Parallel.
TO DO



(a) (b)

Figure 5: Regular mesh with one tripling layer, a) mesh with coarsening_top_layer=False, b) mesh with `coarsening_top_layer=True`, the top layer is remeshed with `actual_vertical_interval_top_layer=1` vertical hex sheet.

5.7 TODO MESH for SANTA MONICA OVERTRUST

Using part of the mesh for Southern California, stitching together several slices.
TO DO

5.8 TODO Grouping the hexes with different period resolved

TO DO

6 TODO SPECFEM3D MESH FORMAT

TO DO

7 TODO NOTES ABOUT THE DIMENSION OF THE HEXES, THE MINIMUM PERIOD RESOLVED AND THE TIME STEP OF THE SIMULATION

TO DO