YLVIS v4.2 software user manual

By Michaela Brchnelova, u0140651

,

Welcome to this simple guide to the YLVIS meshing program!

A pYthon-based consoLe finite-Volume meshIng Software, for generation of simple 2D and 3D quad-based and hexa-based domains and 3D spherical domains for computational fluid dynamics simulations.

This program is the intellectual property of KU Leuven. Please, cite it appropriately if used in published work. For questions, contact michaela.brchnelova@kuleuven.be (but first read the following pages).

This program has two modes of operation:

- Generation of simple blocked 2D and 3D quad based domains
- Generation of a spherical grid based on a surface mesh

Separate python scripts are used for the two modes. With each of them, there is a respective fortran executable (write.exe) and also the source code in case you wanted to create your own format. The procedure is two-fold:

1. First, create a native mesh file format using a python script
2. Transform this native format to GAMBIT neutral format with a fortran script

The reason why the second script is in fortran is due to the fact that the GAMBIT neutral format writing instructions are formulated in fortran, thus this is the most reliable way to produce a proper input file.

1. Simple blocked 2D and 3D quad based domains

The main purpose of the code is to create simple quad-based (or hexagonal-based) meshes in which the user is in complete control of mesh stretching and other properties. For this, the code `main.py` is used.

The name of the native file is saved in the variable “name”. Each mesh type (whether it is the entire mesh or a block, discussed later) has several `params` that must be defined:

- `block_type`: the dimension and type of mesh, e.g. `2D_rect_quad` or `3D_box_quad` (currently the only supported)
- refinement: that consists of the number of nodes in each dimension, the type of stretching in each dimension and the parameters for each stretching function if required
- `xmin, xmax, ymin, ymax (, zmin, zmax)`: minimum and maximum coordinates in each dimension

– `start_n`, `start_e`: these denote from which index should the indexing of the nodes and elements of the block start, can be easily obtained by taking the length of the nodes and elements array prior to adding of the block (or set to 0 in case of a single blocked mesh)

The stretching functions currently implemented are the following (s is spacing, N is number of elements, idx is the current cell index and a the parameter, $\zeta = \text{idx}/(N-1)$ unless stated otherwise):

– `uniform`

$$s = \zeta / I$$

– `negatsin`, type I (if $a = 1$):

$$\zeta = 198.0\text{idx}/(N-1)$$
$$s = -0.0213 + 0.0149\zeta - 0.000151\zeta^2 + 0.000000515\zeta^3$$
$$\zeta_{\max} = 198.$$
$$\min = -0.0213$$
$$\max = -0.0213 + 0.0149\zeta_{\max} - 0.000151\zeta_{\max}^2 + 0.000000515\zeta_{\max}^3$$
$$\text{range} = (\max - \min)$$
$$s = s - \min$$
$$s = s/\text{range}$$

– `negatsin`, type II (if $a = 2$):

$$s = -0.0199 + 2.41\zeta - 4.18\zeta^2 + 2.81\zeta^3$$
$$\min = -0.0199$$
$$\max = 1.0201$$
$$\text{range} = (\max - \min)$$
$$s = s - \min$$
$$s = s/\text{range}$$

– `negatsin`, type III (if $a \neq 1 \&\& a \neq 2$):

$$s = 0.00712 + 1.5\zeta + 2.53\zeta^2 - 16.0\zeta^3 + 22.7\zeta^4 - 10.7\zeta^5 + 0.966\zeta^6$$
$$\min = 0.00712$$
$$\max = 1.00312$$
$$\text{range} = (\max - \min)$$
$$s = s - \min$$
$$s = s/\text{range}$$

- `doubhyptng`

$$s = \frac{1}{2}\frac{1.0 + \tanh(a(\zeta / I - 1.0/2.0))}{\tanh(a/2.0)}$$

- `hyptng`

$$s = \frac{1.0 + \tanh(a(\zeta / I - 1.0))}{\tanh(a)}$$

- `douhbypsin`

$$s = \frac{1}{2}\frac{1.0 + \sinh(a(\zeta / I - 1.0/2.0)}{\sinh(a/2.0))}$$

- `hypsin`

$$s = \frac{1.0 + \sinh(a(\zeta / I - 1.0)}{\sinh(a))}$$

The negative sine distribution has three types, where the type can be selected by the stretching factor (setting it to 1, 2 or something else). These were determined by fitting negative sine distributions raised to different powers. The other functions follow the conventional definitions. The user can easily add their own stretching functions if they follow the principle in which the other functions are handled.

The code is run by simply invoking it, without any arguments:

```
python spherical_main.py [surface_ply_file] [output_file]
```

afterwards, the output file in the native format must be transformed to the GAMBIT file:

```
./write.exe 2d [input_file] [output_file]
```

for 2d domains, and:

```
./write.exe 3d [input_file] [output_file]
```

for 3d domains.

Keep in mind that the 3d domains (especially with blocking) were not yet extensively tested.

Without blocking, the original example given in the code can be simply followed and modified Blocking is also possible. Adding multiple blocks is shown in the commented-out portion of the code (follow the comments in the code). Here, however, the user must be cautious - with each block, re-indexing of the nodes must be done and connectivity adjusted. This is currently done through a double for loop searching for the nodes residing at the same position, which can be very computationally heavy for finer meshes. In case for finer meshes, for this reason, avoid blocking if possible.

For blocking, the following instructions must be used after adding each block:

1. Define the new block `params` (`refinement, boundaries, block_type`)
2. Add the block type by:

```
nodes_add, elements_add = add_block(block_type, params)
nodes = nodes + nodes_add
elements = elements + elements_add
```

3. Delete repeated nodes (on the boundaries of the touching blocks) by:

```
nodes, elements = delete_repeated_nodes(nodes, elements, tol)
```

   note here that the tolerance can be adjusted depending on the scale of the mesh you are building. This tolerance is used to determine whether two points overlap or not, so it should be smaller than the scale of the smallest mesh cell.

4. Recompute the starting node and element index if you wish to not overwrite the previous block:

```
start_n = len(nodes)
start_e = len(elements)
```

2. Spherical grid based on a surface mesh

For this functionality, use the `spherical_main.py` script in the `spherical` folder.

This mode requires an input spherical surface mesh that can be radially extended outward. This mesh must be in the .ply (Stanford) format to be legible. Such meshes can be easily generated by, for example, Blender. Some examples for icospheric meshes are included in the folder `example_surface_meshes`, including their blender setup files and the output ply formats read by the mesher. The input ply files can include double points or disconnected points - all of this is corrected for in the code.

The principle is simple - the surface connectivity (in the ply format) is preserved, but radially extended outwards. From the subsequent layers, the 3D elements are formed. Thus, at least one additional layer is needed, in which case we create a mesh of one layer of elements.

The radial spacing between the layers is prescribed in `ReturnSpacing_idx(i)`. Here, depending on which layer is added (`layer + 1 = i`), the radial distance can be defined. In this function, it is defined as an absolute radius, but also spacing can be used, in which case one should return `Rs[i]` instead of the spacing variable.

The number of layers that will be added can be independent of the Rs array and is defined in `n_layers` variable.

Keep in mind that the first layer is added separately, since the procedure is a bit different. Thus, for the first layer, define the spacing in the respective `delta_r` variable as commanded by the comments (read the comments, they are useful).

To start the program, follow the syntax:

```
python spherical_main.py [surface_ply_file] [output_file]
```

The arguments will be printed onscreen by the code just in case. In addition, also the spacing of the layers will be printed (apart from the very first layer).

Finally, set the overall output path (without the name) in the variable `path`.

Now, this procedure produces a native-formatted file. To convert this file to one that has the information formatted according to the GAMBIT neutral format, follow this syntax:

```
./write.exe pr [input_file] [output_file]
```

where the output file should have the ".neu" extension. The "pr" argument tells the code that prisms will be created. The boundaries will be marked `inlet` (the inner sphere) and `outlet` (the outer sphere), but this can be easily modified by the user. Once can either do this directly in the output file, or in the `write.f90` and recompiling.

Here, we only discussed generation of a prism mesh (that is why the `pr` argument) where the surface mesh is made out of triangles, but in the `quad_example` folder, one can find this procedure also for a quad-based surface mesh (thus, giving hexagonal elements). In that case, still do use the `pr` when invoking the fortran function, but use the correct executable (in the respective file) so that the fortran writer knows to write hexagonal elements.