Grid*Pro* API – Schedule File

API Tutorial - 2

Scope of the tutorial

- The grid generation process is an iterative process in Grid*Pro*. In general the user will have to monitor the grid quality and stop the grid generator manually.
- However, if you approximately know the sweep count for the convergence, Grid*Pro* provides you option to stop after specific number of iterations.
- The iterative process can be steered, controlled and stopped by specifying parameters in the schedule file (extension *sch).
- By default the schedule file with default parameters is written every time the grid generation process is started.

Figure 1: Schedule file

Scope of the tutorial

- In this tutorial, we will discuss about
    1. How to write a schedule file using the API in python.
    2. Use the schedule file to run the grid generator and stop it.
- With the help of API, you can even control the grid generator based on quality which will be discussed in the next tutorial.

step5.final_topo
.sch

```
#This is a sample schedule file. You may edit it.
step 5: -c all 1.0 0 -C all 1.0 24 -r -S 300 -w
write -f blk.tmp
write -D 0 -f dump.tmp
```

Figure 2: Schedule file

Schedule File Terminology

step {}: Instructs how many times the sequence of steps have to be executed.

-c all 1.0 0: Boundary Clustering Specification.If the statement is added it checks for any boundary layer cluster parameters specified to the surface and executes it. For an Euler grid, the syntax has to be removed from the schedule file.

-C all 1.0 24 : Smoother Clustering Specification.This is used to control the internal clustering for cell smoothing.(Adjusted to give best results)

-r <x>: Readjust surfaces with radius = x. This is again parameter for the smoother which is adjusted to give best results. If no value is given, the default value 1 is taken into account.

-S <num> : Represents the number of sweeps per step.

```
step 5: -c all 1.0 0 -C all 1.0 24 -r -S 300 -w
```

Schedule File Terminology

-w <num> : Writes the output grid file every <num> sweeps. Without <num> it writes out after every step. For e.g in this case, it writes out every 300 sweeps.

```
write -f blk.tmp
```

-f blk.tmp : Writes the ASCII grid file to blk.tmp. To write the grid in binary format, a flag –B needs to be added before –f.

```
write -D 0 -f dump.tmp
```

-D <num> : Writes the grid as dump file. If num = 0 , writes in binary format for restart purpose, 2 for 2D grids in ASCII format, 3 for 3D grids in ASCII format. Default = 0.

-f dump.tmp : Writes the grid file to dump.tmp

Problem Definition

- Consider we have a topology file named as ‘step5.final_topo.fra’
- We need to write a python script to create a schedule file which runs for 10 steps with 300 sweeps for every step.
- And output the grid file to ‘output_grid.grd’ and dump file to ‘dump.tmp’.
- Also run ggrid for the input topology file.

Input

- ☐ Topology - step5.final_topo
- ☐ Step Count - 10
- ☐ Sweep Count - 300
- ☐ Output - output_grid.grd
- ☐ Dump - dump.tmp

Figure 3: Input Parameters

Code Snippet

```
# Import required modules
import gp_utilities
import os
#Main Function
if(__name__ == '__main__'):
    topo = gp_utilities.Topology()
    #Input Parameters
    input_topo_prefix = "step5.final_topo"
    step_count = 10
    sweep_count = 300
    #Writing Schedule File and Run Ggrid
    topo.write_schedule_file("{0}.sch".format(input_topo_prefix), step_count,
                             sweep_count, "blk.tmp", "dump.tmp")
    ggrid_command = "Ggrid {0}.fra".format(input_topo_prefix)
    os.system(ggrid_command)
```

```
step 10: -c all 1.0 0 -C all 1.0 24 -r -S 300 -w
write -f blk.tmp
write -D 0 -f dump.tmp
```

Figure 4: Output Schedule FIle

API Module

- Import the ‘gp_utilities’ module to access the GridPro’s API.

```
import gp_utilities
```

- ‘os’ module provides functions for interacting with the operating system. It is commonly used to execute a system command.

```
import os
```

- The grid generator is not exposed through the API at this moment. Hence it has to be executed as a system command.

API Module

- An object of the topology class has to be created. This object keeps track of surfaces, corners, groups and topology operations along with other state variables.
- Before writing out a schedule file, we can check for some errors like missing edge labels and block groups. This is not added to the API yet.
- If this module/source file is used as the main program, Python interpreter sets the special variable '__name__' to '__main__'

```
#Main Function
if(__name__ == '__main__'):
    topo = gp_utilities.Topology()
```

Import modules → Call the Topology Class → Get input parameters → Write schedule File → Run Ggrid

Input Parameters

- Next, let's define the commonly used input parameters described in fig 3.
- Here we are taking only the prefix of the topology file name. i.e. without the 'fra' extension.
- In this way it will be easier to set the name for schedule file since the prefix of the schedule file should be same as the fra file.

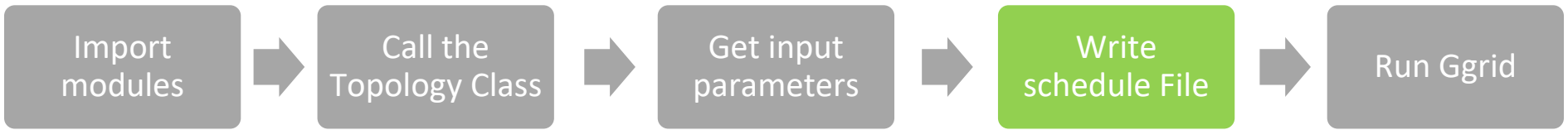
```
input_topo_prefix = "step5.final_topo"

step_count = 10

sweep_count = 300
```

Import modules → Call the Topology Class → **Get input parameters** → Write schedule File → Run Ggrid

Schedule File

- Syntax: `write_schedule_file ("schedule_file_name", no. of steps, sweeps per step, "output grid file name", "output dump file name").`
- For now, only these 5 parameters are exposed. Other advanced features are not added to the API yet, and the default values are used when needed.

```
topo.write_schedule_file("{0}.sch".format(input_topo_prefix), step_count,
sweep_count, "blk.tmp", "dump.tmp")
```



Ggrid

- Ggrid Command line Syntax: `Ggrid <topology file name>`
- It uses the associated schedule file to control and stop the grid generation process.
- Execute the system command to run the grid generating engine.

```
ggrid_command = "Ggrid {0}.fra".format(input_topo_prefix)

os.system(ggrid_command)
```

End of the Tutorial