DVPerturbation Notes

Irene Virdis, Mario Carta

January 15, 2019

1 Compulsory Modules

The following modules are needed in order to correctly run all the command listed in the section below.

```
import numpy as np
from DVPerturbation import *
import os
import math
```

2 Class CfgGenerator

The instance of this class is done by passing the inputs *bump* (the array which contains the bump entity reported as DV variable inside a configuration file) and the array *pos*, which specifies the positions of each bump; each position is referred to the non- dimensional chord of a blade. An example is reported below:

```
B = np.eye(6)*1E-03
z = np.zeros([1,6])
print '------------------------------'
print 'from trigger file:'
print z
B = np.vstack((z,B))
print 'the declared Bump matrix is:'
print B
print 'the position array is :'
pos = np.matrix([[0.0, 0.25], [0.0, 0.5],
        [0.0, 0.75], [1.0, 0.25],
 [1.0, 0.5], [1.0, 0.75]])
print pos
print '------------------------------'

obj = CfgGenerator( bump=B, pos=pos)
```

2.1 Method WriteDraft

This class represents a writing generator of configuration files for the SU2 simulations; the boundary conditions are limited to a 2D case and the are specified into lines 39-40 of the method. The outputs are runnable configuration files, automatically saved inside the directory in which the method is called. An example has been reported to show the syntax (given the lines previously written in section 2):

```
obj.WriteDraft()
```

2.2 Method RunADJ

This method will run the simulatons with the adjoint operator. Inputs are not necessary because a list of Configuration files will be automatically searched, to this aim it is necessary to use the method *WriteDraft* before the present method.

```
obj.WriteDraft()
obj.RunADJ()
```

The outputs are represented by a list of sub-folders, renamed *CONFIG*∗, which contain the output file for each i^th bump configuration.

2.3 ReadRes

This method will read the results and calculates the gradients starting from the file *forces breakdown*: it is necessary to firstly run the method *RunADJ* to obtain this kind of file.

3 Class MeshDeform

3.1 Method ExtractSurface

This method has been written to extract the coordinates of a surface, given its name as input; the expected file is an SU2 format. In the lines below an example has been reported: we want to extract the blade surface "airfoil" from the mesh file "mesh.su2"; the python script for the istance of the class and the correct use of this method should contains the following commads:

```
object = MeshDeform(mesh_name='mesh.su2')
object.ExtractSurface(surface_name='airfoil')
```

The method will automatically save into the folder a file renamed *blade-points.txt*, which contains:

column 0	column 1	column 2
index of point	X coordinate	Y coordinate

3.2 Method ExtractSurface

this method has been written to extract the coordinates of a surface, given the name as input. MAKE SURE THAT THE SURFACE NAME WILL BE WRITTEN IN LOWER OR UPPER CASE AS REPORTED INTO THE ORIGINAL MESH FILE.

```
obj = MeshDeform(mesh_name='mesh_RAE2822_turb.su2')
obj.ExtractSurface('AIRFOIL')
```

The output of the method is the file *blade-points.txt* which contains the coordinate of the blade, with the same order specified inside the original mesh file with SU2 format.

3.3 Method SortedBlade

This method has been written to associate a curvilinear coordinate to each point of the blade surface. The blade points which belong to the return statement of 'ExtractSurface' are sorted from the leading edge (with respect of the hypothesis that this section corresponds to the origin of the axis) The putput of the method is a file *blade-from-le.txt*, which contains only the coordiantes of the blade, sorted by the leading edge point (here the hypothesis is that the leading edge is on the point with minimum value of X).

```
obj = MeshDeform(mesh_name='mesh_RAE2822_turb.su2')
obj.SortedBlade('AIRFOIL')
```

3.4 Method TransformMesh

This method has been written to transform the mesh:

- the inputs for the manipulation of the geometry are optional, while the parameter 'surface' is compulsory.
- a scale action is applied with a constant factor 'scale'
- a translation action is imposed through an array [x,y,z]
- a rotation matrix is imposed by specifing the angle in degree.

  The output of the method is a file *transformed-mesh.txt*, which contains only the coordiantes of the blade. after a rotation, translation and a scale action; if the parameter for this three actions are not given, the default calculation is rotation of 0 degree, translation of (0,0) and scale of 1.0; the angle of rotation should be expressed with degree; the translation is expressed with the coordinates (X,Y) of the displacement and the scale is a constant factor for resizing the blade.)

```
obj = MeshDeform(mesh_name='mesh_RAE2822_turb.su2')
obj.TransformMesh('AIRFOIL', scale=2, translate=(0,0.1), rotate=30)
```

3.5 Method ApplyBump

This method has been written to apply a list of bump using an HickHenne function over a surface: the inputs are rappresented by the bump matrix *bump-array* and the string *surface* which specifies the name of the surface on which bumps will be applied. The following example is applied tothe mesh of a *RAE2822* airfoil, given the name *AIRFOIL* associated to the blade surface inside the su2 mesh.

```
b = np.matrix([[0.3, 0.01],[0.5, 0.002]])
obj = MeshDeform(mesh_name='mesh_RAE2822_turb.u2')
obj.ApplyBump(bump_array=b, surface='AIRFOIL')
```

The output file will be: *bump-new.txt* which contains the matrix of the applied bump (the first column represents the list of the non-dimensional curvilinear coordinate, while the second one is the associated bump entity); the second output file sis *TMP.txt*, a matrix with all the dimensional, non-dimensional eulerian and curvilinear coordinates, the perturbation after the application of the HicksHenne function and the final coodrdinates of the new blade; the last file, *surface-positions.dat* is the matrix of the new blade coordinates.

3.6 Method ApplyDEF

This method automatically applies the SU2 module *DEF*, starting from the configuration file specified as input. An example is reported below:

```
b = np.matrix([[0.3, 0.01],[0.5, 0.002]])
obj = MeshDeform(mesh_name='mesh_RAE2822_turb.su2')
obj.ApplyBump(bump_array=b, surface='AIRFOIL')
obj.ApplyDEF(config_file='baseline.cfg')
```

The final output is the mesh file *mesh-out.su2*

3.7 Method VerifyIntegrity

This method has been written to check the maximum displacement of the nodes after the usage of the DEF module. Originally it has ben thought to verify the periodic nodes and their final displacement.

```
b = np.matrix([[0.3, 0.01],[0.5, 0.002]])
obj = MeshDeform(mesh_name='mesh_RAE2822_turb.su2')
obj.ApplyBump(bump_array=b, surface='AIRFOIL')
obj.ApplyDEF(config_file='baseline.cfg')
obj.VerifyIntegrity(mesh_new='mesh_out.su2',
mesh_old='mesh_RAE2822_turb.su2',
bump_matrix=b)
```

The output have been writen to check the maximum displacement of the nodes inside a mesh after the application of the DEF module. The file *mesh-verify0.txt* contains only the coordinates of the specified surface of the original blade; the file *mesh-verify1.txt* contains the coordinates of the blade surface after the application of the DEF module; the file *corrected-mesh* and *full-corrected mesh.txt* contain the new coordinates of all the nodes (not only the blade) after the check of the maximum displacement ($2 * bump_entity$); *mesh-verified.su2* can be used for a simulation.