IDWarp

MDO Lab

Mar 14, 2022

CONTENTS

`IDWarp` is a tool for performing mesh manipulation of 3-dimensional multi-block, overset and unstructured meshes.

CHAPTER

ONE

BUILDING

IDWarp depends on the follow libraries: - CGNS Library - PETSc - MPI

See the MDO Lab installation guide here for the supported versions and installation instructions.

All the core computations in IDWarp are coded in Fortran. It is therefore necessary to build this library before using IDWarp.

To see a list of architectures that IDWarp has been known to compile on run:

```
make
```

from the root directory.

Follow the instructions and copy the closest architecture file and attempt a build using

```
make
```

If everything was successful, the following lines will be printed to the screen (near the end):

```
Testing if module idwarp can be imported...
Module idwarp was successfully imported.
```

If you don't see this, it will be necessary modify the configure options in the config file.

It will most likely be necessary to modify the `CGNS_INCLUDE_FLAGS` and the `CGNS_LINKER_FLAGS` variables. After changes to the configuration file, run `make clean` before attempting a new build.

Lastly, to build the Python interface, go to the root directory and type:

```
pip install .
```

CHAPTER

TWO

TUTORIAL

This is a short tutorial explaining how to get started with IDWarp. We will start by presenting a simple stand-alone warping script and explaining the various parts.

```
from mpi4py import MPI
from idwarp import USMesh

options = {
  'gridFile':'../../input_files/o_mesh.cgns',
  'fileType':'cgns',
  'specifiedSurfaces':None,
  'symmetrySurfaces':None,
  'symmetryPlanes':[],
  'aExp': 3.0,
  'bExp': 5.0,
  'LdefFact':1.0,
  'alpha':0.25,
  'errTol':0.0001,
  'evalMode':'fast',
  'useRotations':True,
  'zeroCornerRotations':True,
  'cornerAngle':30.0,
  'bucketSize':8,
}

# Create the mesh object
mesh = USMesh(options=options, comm=MPI.COMM_WORLD)

# Extract all coordinates
coords0 = mesh.getSurfaceCoordinates()

# Modify the coordinates as required
newCoords = coords0.copy()
for i in range(len(coords0)):
    newCoords[i, :] *= 1.1

# Reset the newly computed surface coordinates
mesh.setSurfaceCoordinates(newCoords)

# Actually run the mesh warping
mesh.warpMesh()
```

(continues on next page)

(continued from previous page)

```
# Write the new grid file.
mesh.writeGrid('warped.cgns')
```

The first two lines of code imports mpi4py and the mesh warping module

```
from mpi4py import MPI
from idwarp import USMesh
```

The next chunk lists the options for warping. The options are explained in *Options*.

```
options = {
  'gridFile':'../../input_files/o_mesh.cgns',
  'fileType':'cgns',
  'specifiedSurfaces':None,
  'symmetrySurfaces':None,
  'symmetryPlanes':[],
  'aExp': 3.0,
  'bExp': 5.0,
  'LdefFact':1.0,
  'alpha':0.25,
  'errTol':0.0001,
  'evalMode':'fast',
  'useRotations':True,
  'zeroCornerRotations':True,
  'cornerAngle':30.0,
  'bucketSize':8,
}
```

Next we create the actual mesh object itself

```
# Create the mesh object
mesh = USMesh(options=options, comm=MPI.COMM_WORLD)
```

Note that we have explicitly passed in the MPI intracommunicator on which we want to create the object. If the 'comm' keyword argument is not given, it will default to MPI.COMM_WORLD. Therefor this example, mpi4py is not strictly required to be imported in the run script.

Next we request the surface coordinates from the mesh object. These will correspond to coordinates in the 'specifiedSurfaces' option.

```
# Extract all coordinates
coords0 = mesh.getSurfaceCoordinates()
```

coords0 is a numpy array of size (N,3). It is now up to the user to manipulate these coordinates however they wish for this example we simply loop over all coordinates and uniformly scale by a factor of 1.1:

```
new_coords = coords0.copy()
for i in range(len(coords0)):
    new_coords[i, :] *= 1.1
```

Once the new set of coordinates have been determined, return them to the mesh warping object with the following command.

```
# Reset the newly computed surface coordiantes
mesh.setSurfaceCoordinates(new_coords)
```

Note that the shape of 'new_coords' must be identical to the coords0 array that was originally provided by the warping. Next we run the actual mesh warp using

```
# Actually run the mesh warping
mesh.warpMesh()
```

And finally to produce an updated grid file we can write the grid:

```
# Write the new grid file.
mesh.writeGrid('warped.cgns')
```

The warped grid file 'warped.cgns' will contain all the boundary condition/connectivity/auxiliary information as the original cgns file. Only the coordinates are updated to their new positions.

CHAPTER

THREE

OPTIONS

Table 1: USMesh Default Options

Name	Type	Default value	Description
`gridFile`	str or NoneType	None	This is the grid file to use. It must always be specified. It may be a structured or unstructured grid file or it may be an OpenFOAM directory containing a mesh specification.
`fileType`	str	`CGNS`	Specify the type of grid file. • `CGNS`: CGNS file • `OpenFOAM`: OpenFOAM directory • `PLOT3D`: PLOT3D file
`specifiedSurfaces`	list or NoneType	None	This option is used to specify which surfaces are used to build the surface tree where deformations are to be specified. The default is None which will automatically use all wall-type surfaces in the grid file. For CGNS files this corresponds to the following boundary condtiions: `BCWall`, `BCWallViscous`, `BCWallViscousHeatFlux`, `BCWallViscousAdiabatic`, `BCWallInviscid`. For OpenFOAM files, all `patch` and `wall` surfaces are assumed by default. If a non-None value is given it should be list of families the use wants to use to generate the surface definition.
`symmetrySurfaces`	list or NoneType	None	This option is used to specify patch surfaces are used to determine symmetry planes. If None, IDwarp will

CHAPTER

FOUR

API

4.1 USMesh

class idwarp.UnstructuredMesh.**USMesh**(**args*, ***kwargs*)

This is the main Unstructured Mesh. This mesh object is designed to interact with an structured or unstructured CFD solver though a variety of interface functions.

Create the USMesh object.

Parameters

options [dictionary] A dictionary containing the options for the mesh movement strategies. THIS IS NOT OPTIONAL.

comm [MPI_INTRA_COMM] MPI communication (as obtained from mpi4py) on which to create the USMesh object. If not provided, MPI_COMM_WORLD is used.

debug [bool] Flag specifying if the MExt import is automatically deleted. This needs to be true ONLY when a symbolic debugger is used.

getCommonGrid()

Return the grid in the original ordering. This is required for the OpenFOAM tecplot writer since the connectivity is only known in this ordering.

getSolverGrid()

Return the current grid in the order specified by setExternalMeshIndices(). This is the main routine for returning the defomed mesh to the external CFD solver.

Returns

solverGrid, numpy array, real: The resulting grid. The output is returned in flatted 1D coordinate format. The len of the array is 3*len(indices) as set by setExternalMeshIndices()

getSurfaceCoordinates()

Returns all defined surface coordinates on this processor

Returns

coords [numpy array size (N,3)] Specified surface coordinates residing on this processor. This may be empty array, size (0,3)

getWarpGrid()

Return the current grid. This funtion is typically unused. See getSolverGrid for the more useful interface functionality.

Returns

warpGrid, numpy array, real: The resulting grid. The output is returned in flatted 1D coordinate format.

`getdXs`()

Return the current values in dXs. This is the result from a mesh-warp derivative computation.

Returns

dXs [numpy array] The specific components of dXs. size(N,3). This the same size as the array obtained with getSurfaceCoordiantes(). N may be zero if this processor does not have any surface coordinates.

`setExternalMeshIndices`(*ind*)

Set the indicies defining the transformation of an external solver grid to the original CGNS grid. This is required to use USMesh functions that involve the word "Solver" and warpDeriv. The indices must be zero-based.

Parameters

ind [numpy integer array] The list of indicies this processor needs from the common mesh file

`setSurfaceCoordinates`(*coordinates*)

Sets all surface coordinates on this processor

Parameters

coordinates [numpy array, size(N, 3)] The coordinate to set. This MUST be exactly the same size as the array obtained from getSurfaceCoordinates()

`setSurfaceDefinition`(*pts*, *conn*, *faceSizes*, *cgnsBlockID=None*)

This is the master function that determines the definition of the surface to be used for the mesh movement. This surface may be supplied from an external solver (such as SUmb) or it may be generated by IDWarp internally.

Parameters

pts [array, size (M, 3)] Nodes on this processor

conn [int array, size (sum(faceSizes))] Connectivity of the nodes on this processor

faceSizes [int Array size (N)] Treat the conn array as a flat list with the faceSizes giving connectivity offset for each element.

cgnsBlockID [dummy argument.] This argument is not used at all. It is here just to have the same API as the IDWarpMulti class.

`setSurfaceDefinitionFromFile`(*surfFile*)

Set the surface definition for the warping from a multiblock PLOT3D surface file

Parameters

surfFile [filename of multiblock PLOT3D surface file.]

`setSurfaceFromFile`(*surfFile*)

Update the internal surface surface coordinates using an external PLOT3D surface file. This can be used in an analogous way to setSurfaceDefinitionFromFile. The 'sense' of the file must be same as the file used with setSurfaceDefinitionFromFile. That means, the same number of blocks, with the same sizes, in the same order.

Parameters

surfFile: filename of multiblock PLOT3D surface file'

verifyWarpDeriv(*dXv=None*, *solverVec=True*, *dofStart=0*, *dofEnd=10*, *h=1e-06*, *randomSeed=314*)
Run an internal verification of the solid warping derivatives

warpDeriv(*dXv*, *solverVec=True*)
Compute the warping derivative (dXv/dXs^T)*Vec (where vec is the dXv argument to this function.

This is the main routine to compute the mesh warping derivative.

Parameters

dXv [numpy array] Vector of size external solver_grid. This is typically obtained from the external solver's dRdx^T * psi calculation.

solverVec [logical] Flag to indicate that the dXv vector is in the solver ordering and must be converted to the warp ordering first. This is the usual approach and thus defaults to True.

Returns

None. The resulting calculation is available from the getdXs()

function.

warpDerivFwd(*dXs*, *solverVec=True*)
Compute the forward mode warping derivative

This routine is not used for "regular" optimization; it is used for matrix-free type optimization. dXs is assumed to be the the peturbation on all the surface nodes.

Parameters

dXs [array, size Nsx3] This is the forward mode peturbation seed. Same size as the surface mesh from getSurfaceCoordinates().

solverVec [bool] Whether or not to convert to the solver ordering.

Returns

dXv [array] The peturbation on the volume meshes. It may be in warp ordering or solver ordering depending on the solverVec flag.

warpMesh()
Run the applicable mesh warping strategy.

This will update the volume coordinates to match surface coordinates set with setSurfaceCoordinates()

writeGrid(*fileName=None*)
Write the current grid to the correct format

Parameters

fileName [str or None] Filename for grid. Should end in .cgns for CGNS files. For PLOT3D whatever you want. It is not optional for CGNS/PLOT3D. It is not required for OpenFOAM meshes. This call will update the 'points' file.

writeOFGridTecplot(*fileName*)
Write the current OpenFOAM grid to a Tecplot FE polyhedron file. This is generally used for debugging/visualization purposes.

Parameters

fileName [str] Filename to use. Should end in .dat for tecplot ascii file.

4.2 MultiUSMesh

class idwarp.MultiUnstructuredMesh.**MultiUSMesh**(*CGNSFile*, *optionsDict*, *comm=None*, *dtype='d'*, *debug=False*)

This mesh object is designed to support independent deformation of multiple overset component meshes.

Create the MultiUSMesh object.

INPUTS:

CGNSFile: string -> file name of the CGNS file. This CGNS file should be generated with cgns_utils combine, so that the domain names have the appropriate convention. That is, domains will have the same name as their original files. Domains that share the same name will be grouped to make an IDWarp instance.

optionsDict: dictionary of dictionaries -> Dictionary containing dictionaries that will be used to initialize multiple IDWarp instances. The keys are domain names and the values are dictionaries of standard IDWarp options that will be applied to this domain. The domains of the full CGNS file that do not have a corresponding entry in optionsDict will not be warped. For instance, if the CGNS file has the domains wing.00000, wing.00001, and wing.00002 associated with a wing mesh that we want to warp, then optionsDict should have an entry for 'wing'.

Ney Secco 2017-02

getSolverGrid()

Return the current grid in the order specified by setExternalMeshIndices(). This is the main routine for returning the deformed mesh to the external CFD solver.

Returns

solverGrid, numpy array, real: The resulting grid. The output is returned in flatted 1D coordinate format. The len of the array is 3*len(indices) as set by setExternalMeshIndices()

Ney Secco 2017-02

getSurfaceCoordinates()

Returns all defined surface coordinates on this processor, with the Solver ordering

Returns

pts [numpy array size (N,3)] Specified surface coordinates residing on this processor. This may be empty array, size (0,3)

Ney Secco 2017-02

getWarpGrid()

Return the current grids. This function is typically unused. See getSolverGrid for the more useful interface functionality.

This only returns the nearfield meshes.

Returns

volNodesList, list of 1D numpy arrays, real: These are the local volume nodes (in a flat 1D array)

of each instance. That is, volNodesList[i] has the volume nodes stored in the local proc for

the i-th IDWarp instance.

numCoorTotal: The total number of coordinates, across all procs and IDWarp instances.

Ney Secco 2017-02

`getdXs()`

Return the current values in dXs. This is the result from a mesh-warp derivative computation. Note that the same steps used in this function are done at warpDeriv, so in theory you could save time by just saving the output of warpDeriv.

Here we accumulate all seeds coming from each IDWarp instance

Returns

dXs [numpy array] The specific components of dXs. size(N,3). This the same size as the array obtained with getSurfaceCoordiantes(). N may be zero if this processor does not have any surface coordinates. These can be, for instance, reverse AD seeds.

Ney Secco 2017-03

`setExternalMeshIndices(ind)`

Set the indices defining the transformation of an external solver grid to the original CGNS grid. This is required to use USMesh functions that involve the word "Solver" and warpDeriv. The indices must be zero-based.

Parameters

ind [numpy integer array[3*n] where n is the number of nodes stored in the proc.] The list of indicies this processor needs from the common mesh file. ind[3*i:3*i+3] represents the position of each coordinate of the i-th ADflow node in the CGNS file. ind has 0-based indices.

Ney Secco 2017-02

`setSurfaceCoordinates(pts)`

Sets all surface coordinates on this processor, with pts given in solver ordering

Parameters

pts [numpy array, size(N, 3)] The coordinate to set. This MUST be exactly the same size as the array obtained from getSurfaceCoordinates()

Ney Secco 2017-02

`setSurfaceDefinition(pts, conn=None, faceSizes=None, cgnsBlockIDs=None, distTol=1e-06)`

This is the master function that determines the definition of the surface to be used for the mesh movement. This surface may be supplied from an external solver (such as ADflow) or it may be generated by IDWarp internally.

Parameters

pts [array, size (M, 3)] Nodes on this processor

conn [int array, size (sum(faceSizes))] Connectivity of the nodes on this processor

faceSizes [int Array size (N)] Treat the conn array as a flat list with the faceSizes giving connectivity offset for each element.

cgnsBlockIDs [int Array size (N)] Block ID, in CGNS ordering, that contains each element.

distTol: Distance tolerance to flag that a given surface node does not belong to the current IDWarp surface definition in the current proc.

Ney Secco 2017-02

`warpDeriv(dXv, solverVec=True)`

Compute the warping derivative (dXv/dXs^T)*Vec (where vec is the dXv argument to this function.

This is the main routine to compute the mesh warping derivative.

Parameters

dXv [numpy array] Vector of size external solver_grid. This is typically obtained from the external solver's dRdx^T * psi calculation.

solverVec [logical] Flag to indicate that the dXv vector is in the solver ordering and must be converted to the warp ordering first. This is the usual approach and thus defaults to True.

Returns

dXs [numpy array] The specific components of dXs. size(N,3). This the same size as the array obtained with getSurfaceCoordiantes(). N may be zero if this processor does not have any surface coordinates. These can be, for instance, reverse AD seeds.

Ney Secco 2017-03

`warpDerivFwd`(*dXs*)

Compute the forward mode warping derivative

This routine is not used for "regular" optimization; it is used for matrix-free type optimization. dXs is assumed to be the the peturbation on all the surface nodes.

Parameters

dXs [array, size Nsx3] This is the forward mode peturbation seed. Same size as the surface mesh from getSurfaceCoordinates().

solverVec [bool] Whether or not to convert to the solver ordering.

Returns

dXv [array] The peturbation on the volume meshes. It may be in warp ordering or solver ordering depending on the solverVec flag.

Ney Secco 2017-03

`warpMesh`()

This calls the mesh warping method for each IDWarp instance.

This will update the volume coordinates internally in each instance.

Ney Secco 2017-02

`writeGrid`(*baseName=None*)

Write the grids of each instance

Parameters

baseName [str or None] a base namve that will be used to generate filenames for all instance mesh files.

Ney Secco 2017-03

CHAPTER

FIVE

LIMITATIONS

- Cannot be used where you have 2 symmetry planes e.g. in a 2D case. Use pywarp instead.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX