baseClasses

MDO Lab

Oct 06, 2022

PROBLEM CLASSES

The `baseClasses` repository contains common definitions for aerodynamic, structural and aerostructural solvers, as well as classes defining standard ways of describing aerodynamic, structural and aerostructural problems for use in those classes.

CHAPTER

ONE

INSTALLING

The package may be installed by pip from PyPI, as:

```
pip install mdolab-baseclassees
```

Or by Conda from the `conda-forge` channel, as:

```
conda install -c conda-forge mdolab-baseclasses
```

Or from source, by cloning the repo and then running (from the repo root):

```
pip install .
```

1.1 AeroProblem

class baseclasses.**AeroProblem**(*name*, ***kwargs*)

The main purpose of this class is to represent all relevant information for a single aerodynamic analysis. This will include the thermodynamic parameters defining the flow condition and the reference quantities for normalization.

There are several different ways of specifying thermodynamic conditions. The following describes several of the possible ways and the appropriate situations.

'mach' + 'altitude'
: This is the preferred method for specifying flight conditions. This is suitable for all aerodynamic analysis codes, including aerostructural analysis. The 1976 standard atmosphere is used to compute P and T. We then compute $\rho = P/RT$. The remaining quantities are computed with *baseclasses.AeroProblem._updateFromM()*. The resulting Reynolds number depends on the scale of the mesh.

'mach' + 'reynolds' + 'reynoldsLength' + 'T':
: Used to precisely match Reynolds numbers. The remaining quantities are computed with *baseclasses.AeroProblem._updateFromRe()*.

'V' + 'reynolds' + 'reynoldsLength' + 'T':
: Used to precisely match Reynolds numbers for low-speed cases. The remaining quantities are computed with *baseclasses.AeroProblem._updateFromRe()*.

'mach' + 'T' + 'P':
: Any arbitrary temperature and pressure. The inputs are first used to compute $\rho = P/RT$. The remaining quantities are then computed with *baseclasses.AeroProblem._updateFromM()*.

'mach' + 'T' + 'rho':
: Any arbitrary temperature and density. The inputs are first used to compute $P = \rho RT$. The remaining quantities are then computed with *baseclasses.AeroProblem._updateFromM()*.

'mach' + 'P' + 'rho':
Any arbitrary density and pressure. The inputs are first used to compute $T = P/\rho R$. The remaining quantities are then computed with *baseclasses.AeroProblem._updateFromM()*.

'V' + 'rho' + 'T'
Generally for low-speed specifications. The inputs are first used to compute $P = \rho RT$. The remaining quantities are then computed with *baseclasses.AeroProblem._updateFromV()*.

'V' + 'rho' + 'P'
Generally for low-speed specifications. The inputs are first used to compute $T = P/\rho R$. The remaining quantities are then computed with *baseclasses.AeroProblem._updateFromV()*.

'V' + 'T' + 'P'
Generally for low-speed specifications. The inputs are first used to compute $\rho = P/RT$. The remaining quantities are then computed with *baseclasses.AeroProblem._updateFromV()*.

The combinations listed above are the **only** valid combinations of arguments that are permitted. Furthermore, since the internal processing is based (permanently) on these parameters, it is important that the parameters given on initialization are sufficient for the required analysis. For example, if only the Mach number is given, an error will be raised if the user tries to set the 'P' (pressure) variable.

For our compressible RANS solver, ADflow, the inputs from `AeroProblem` are the dimensional freestream values M, P, T, γ, ρ, R_{gas}, Sutherland's law constants S, T_{ref}, μ_{ref}, and the Prandtl number Pr. The non-dimensionalized inputs used in the actual ADflow CFD computations are derived from these inherited inputs.

All parameters are optional except for the `name` argument which is required. All of the parameters listed below can be acessed and set directly after class creation by calling:

```
<aeroProblem>.<variable> = <value>
```

An attempt is made internally to maintain consistency of the supplied arguments. For example, if the altitude variable is set directly, the other thermodynamic properties (rho, P, T, mu, a) are updated accordingly.

Parameters

name
[str] Name of this aerodynamic problem.

evalFuncs
[iterable object containing strings] The names of the functions the user wants evaluated with this aeroProblem.

mach
[float. Default is 0.0] Set the Mach number for the simulation

machRef
[float. Default is None] Sets the reference Mach number for the simulation.

machGrid
[float. Default is None] Set the Mach number for the grid.

alpha
[float. Default is 0.0] Set the angle of attack in degrees.

beta
[float. Default is 0.0] Set the side-slip angle in degrees.

altitude
[float. Default is 0.0] Set all thermodynamic parameters from the 1976 standard atmosphere. The altitude must be given in meters.

phat
[float. Default is 0.0] Set the rolling rate coefficient

qhat
[float. Default is 0.0] Set the pitch rate coefficient

rhat
[float. Default is 0.0] Set the yawing rate coefficient

degPol
[integer. Default is 0] Degree of polynomial for prescribed motion. ADflow only

coefPol
[array_like. Default is [0.0]] Coefficients of polynomial motion. ADflow only

degFourier
[integer. Default is 0] Degree of Fourier coefficient for prescribed motion. ADflow only

omegaFourier
[float. Default is 0.0] Fundamental circular frequency for oscillatory motion. ADflow only

cosCoefFourier
[array_like. Default is [0.0]] Coefficients for cos terms

sinCoefFourier
[array_like. Default is [0.0]] Coefficients for the sin terms

P
[float.] Set the ambient pressure

T
[float.] Set the ambient temperature

gamma
[float. Default is 1.4] Set the ratio of the specific heats in ideal gas law

reynolds
[float. Default is None] Set the Reynolds number

reynoldslength
[float. Default is 1.0] Set the reference length for the Reynolds number calculations

areaRef
[float. Default is 1.0] Set the reference area used for normalization of lift, drag, etc.

chordRef
[float. Default is 1.0] Set the reference length used for moment normalization

spanRef
[float. Default is 1.0] Set reference length for span. Only used for normalization of p-derivatives

xRef
[float. Default is 0.0] Set the x-coordinate location of the center about which moments are taken

yRef
[float. Default is 0.0] Set the y-coordinate location of the center about which moments are taken

zRef
[float. Default is 0.0] Set the z-coordinate location of the center about which moments are taken

momentAxis
[iterable object containing floats.] Default is [[0.0, 0.0, 0.0], [1.0, 0.0, 0.0]] Set the reference axis for non-x/y/z based moment calculations

englishUnits
[bool] Flag to use all English units: pounds, feet, Rankine etc.

solverOptions
[dict] A set of solver specific options that temporarily override the solver's internal options for this aero problem only. It must contain the name of the solver followed by a dictionary of options for that solver. For example `solverOptions={'adflow':{'vis4':0.018}}`. Currently, the only solver supported is 'adflow' and must use the specific key 'adflow'.

Notes

See *baseclasses.FluidProperties* for more parameters that can be set.

Examples

```
>>> # DPW4 Test condition (metric)
>>> ap = AeroProblem('tunnel_condition', mach=0.85, reynolds=5e6,␣
↪reynoldsLength=275.8*.0254, T=310.93, areaRef=594720*.0254**2, chordRef=275.8*.
↪0254, xRef=1325.9*0.0254, zRef=177.95*.0254)
>>> # DPW4 Flight condition (metric)
>>> ap = AeroProblem('flight_condition', mach=0.85, altitude=37000*.3048,␣
↪areaRef=594720*.0254**2, chordRef=275.8*.0254, xRef=1325.9*0.0254, zRef=177.95*.
↪0254)
>>> # Onera M6 Test condition (Euler)
>>> ap = AeroProblem('m6_tunnel', mach=0.8395, areaRef=0.772893541, chordRef=0.
↪64607, xRef=0.0, zRef=0.0, alpha=3.06)
>>> # Onera M6 Test condition (RANS)
>>> ap = AeroProblem('m6_tunnel', mach=0.8395, reynolds=11.72e6, reynoldsLength=0.
↪64607, areaRef=0.772893541, chordRef=0.64607, xRef=0.0, zRef=0.0, alpha=3.06,␣
↪T=255.56)
>>> # NACA0009 hydrofoil (0.9m semi-span) sailing condition (hacked for␣
↪incompressible flow and viscosity)
>>> # R=461.9 for water vapor, but we can lower it to get a higher Mach number
>>> # Hack to get the dynamic viscosity of water, TSuthDim must equal T for this to␣
↪work!
>>> ap = AeroProblem("hydrofoil", areaRef=0.243, alpha=6, chordRef=0.27, T=288.15,␣
↪V=17, rho=1025, xRef=0.18, yRef=0.0, zRef=0.0, evalFuncs=["cl","cd","lift","drag",
↪"cavitation","target_cavitation"], R=100, muSuthDim=1.22e-3, TSuthDim=288.15)
```

_updateFromM()

Update the full set of states from M, T, rho with the following steps:

1. $a = \sqrt{\gamma RT}$
2. Compute $\mu(T)$ from Sutherland's law.
3. $V = Ma$
4. $Re/L = \rho V/\mu$
5. $\nu = \mu/\rho$

6. $q = 0.5\rho V^2$

_updateFromRe()

Update the full set of states from Re, T, and either V or M with the following steps:

1. $a = \sqrt{\gamma RT}$
2. Compute $\mu(T)$ from Sutherland's law.
3. $V = Ma$ or $M = V/a$
4. $\rho = \frac{Re\mu}{VL}$
5. $P = \rho RT$
6. $q = 0.5\rho V^2$

_updateFromV()

Update the full set of states from V, T, rho with the following steps:

1. $a = \sqrt{\gamma RT}$
2. Compute $\mu(T)$ from Sutherland's law.
3. $\nu = \mu/\rho$
4. $q = 0.5\rho V^2$
5. $M = V/a$
6. $Re/L = \rho V/\mu$

addDV(*key*, *value=None*, *lower=None*, *upper=None*, *scale=1.0*, *name=None*, *offset=0.0*, *dvOffset=0.0*, *addToPyOpt=True*, *family=None*, *units=None*)

Add one of the class attributes as an 'aerodynamic' design variable. Typical variables are alpha, mach, altitude, chordRef, etc. An error will be given if the requested DV is not allowed to be added.

Parameters

key
[str] Name of variable to add. See above for possible ones

value
[float. Default is None] Initial value for variable. If not given, current value of the attribute will be used.

lower
[float. Default is None] Optimization lower bound. Default is unbonded.

upper
[float. Default is None] Optimization upper bound. Default is unbounded.

scale
[float. Default is 1.0] Set scaling parameter for the optimization to use.

name
[str. Default is None] Overwrite the name of this variable. This is typically only used when the user wishes to have multiple aeroProblems to explictly use the same design variable.

offset
[float. Default is 0.0] Specify a constant offset of the value relative to the actual design variable. This is most often used when a single aerodynamic variable is used to change multiple aeroProblems. For example, if you have three aeroProblems for a multiPoint analysis with Mach numbers of 0.84, 0.85 and 0.86, and you want all three to change by the same amount, you could do this:

```
>>> ap1.addDV('mach',..., name='centerMach', offset=-0.01)
>>> ap2.addDV('mach',..., name='centerMach', offset= 0.00)
>>> ap3.addDV('mach',..., name='centerMach', offset=+0.01)
```

The result is a single design variable driving three different Mach numbers.

dvOffset
[float. Default is 0.0] This is the offset used to give to pyOptSparse. It can be used to re-center the value about zero.

addToPyOpt
[bool. Default True.] Flag specifying if this variable should be added. Normally this is True. However, if there are multiple aeroProblems sharing the same variable, only one needs to add the variables to pyOpt and the others can set this to False.

units
[str or None. Default None] Physical units of the variable

Examples

```
>>> # Add alpha variable with typical bounds
>>> ap.addDV('alpha', value=2.5, lower=0.0, upper=10.0, scale=0.1)
```

addVariablesPyOpt(*optProb*)

Add the current set of variables to the optProb object.

Parameters

optProb
[pyOpt_optimization class] Optimization problem definition to which variables are added

evalFunctions(*funcs*, *evalFuncs*, *ignoreMissing=False*)

Evaluate the desired aerodynamic functions. It may seem strange that the aeroProblem has 'functions' associated with it, but in certain instances, this is the case.

For an aerodynamic optimization, consider the case when 'mach' is a design variable, and the objective is ML/D. We need the mach variable explictly in our our objCon function. In this case, the 'function' is simply the design variable itself, and the derivative of the function with respect the design variable is 1.0.

A more complex example is when 'altitude' is used for an aerostructural optimization. If we use the Breguet range equation is used for either the objective or constraints we need to know the flight velocity, 'V', which is a non-trivial function of the altitude (and Mach number).

Also, even if 'altitude' and 'mach' are not parameters, this function can be used to evaluate the 'V' value for example. In this case, 'V' is simply constant and no sensitivties would be calculated which is fine.

Note that the list of available functions depends on how the user has initialized the flight condition.

Parameters

funcs
[dict] Dictionary into which the functions are saved

evalFuncs
[iterable object containing strings] The functions that the user wants evaluated

evalFunctionsSens(*funcsSens*, *evalFuncs*, *ignoreMissing=True*)

Evaluate the sensitivity of the desired aerodynamic functions.

Parameters

funcsSens
[dict] Dictionary into which the function sensitivities are saved

evalFuncs
[iterable object containing strings] The functions that the user wants evaluated

`setBCVar`(*varName*, *value*, *familyName*)

set the value of a BC variable on a specific variable

`setDesignVars`(*x*)

Set the variables in the x-dict for this object.

Parameters

x
[dict] Dictionary of variables which may or may not contain the design variable names this object needs

`updateInternalDVs`()

A specialized function that allows for the updating of the internally stored DVs. This would be used for, example, if a CLsolve is done before the optimization and that value needs to be used.

1.2 FluidProperties

`class` baseclasses.`FluidProperties`(***kwargs*)

Parameters

gamma
[float (default = 1.4)] Set the ratio of the specific heats.

Pr
[float (default = 0.72)] Set the Prandtl number.

R
[float (default = 287.055 J / kg / K)] The specific gas constant. By default we use air.

SSuthDim
[float (default = 110.55)] The Sutherland temperature for Sutherland's Law.

muSuthDim
[float (default = 1.716e-5)] The viscosity at the reference temperature for Sutherland's Law. If you want to directly specify a viscosity for your fluid (e.g., running a case in water), a hack to achieve this is to set *muSuthDim* to the desired viscosity and *T* to *TSuthDim*. By doing so, *mu* will be equal to *muSuthDim*.

TSuthDim
[float (default = 273.15)] The reference temperature for Sutherland's Law.

`updateViscosity`(*T*)

Compute the dynamic viscosity using Sutherland's Law

1.3 StructProblem

class baseclasses.**StructProblem**(*name*, *loadFile=None*, *loadFactor=None*, *evalFuncs=None*)

The main purpose of this class is to represent all relevant information for a structural analysis. This will include information defining the loading condition as well as various other pieces of information.

Parameters

name
[str] Name of this structural problem

loadFile
[str] Filename of the (static) external load file. Should be generated from either ADflow or Tripan.

Examples

```
>>> sp = StructProblem('lc0', loadFile='loads.txt')
```

addDV(*key*, *value=None*, *lower=None*, *upper=None*, *scale=1.0*, *name=None*)

No design variable functions yet.

Parameters

key
[str] Name of variable to add. See above for possible ones

value
[float. Default is None] Initial value for variable. If not given, current value of the attribute will be used.

lower
[float. Default is None] Optimization lower bound. Default is unbonded.

upper
[float. Default is None] Optimization upper bound. Default is unbounded.

scale
[float. Default is 1.0] Set scaling parameter for the optimization to use.

name
[str. Default is None] Overwrite the default auto-generated name of this variable.

addVariablesPyOpt(*optProb*)

Add the current set of variables to the optProb object.

Parameters

optProb
[pyOpt_optimization class] Optimization problem definition to which variables are added

evalFunctions(*funcs*, *evalFuncs*, *ignoreMissing=False*)

No current functions

Parameters

funcs
[dict] Dictionary into which the functions are save

evalFuncs
[iterable object containing strings] The functions that the user wants evaluated

`evalFunctionsSens`(*funcsSens*, *evalFuncs*, *ignoreMissing=False*)

Evaluate the sensitivity of the desired functions

Parameters

funcsSens
[dict] Dictionary into which the function sensitivities are saved

evalFuncs
[iterable object containing strings] The functions that the user wants evaluated

`setDesignVars`(*x*)

Set the variables in the x-dict for this object.

Parameters

x
[dict] Dictionary of variables which may or may not contain the design variable names this object needs

1.4 AeroStructProblem

`class` baseclasses.`AeroStructProblem`(*ap*, *sp*, ***kwargs*)

The main purpose of this class is to represent all relevant information for a coupled aero-structural analysis. To this end, it maintains a reference to an AaeroProblem and a StructProblem.

Parameters

ap
[AeroProblem class instance] An instance of the AeroProblem class defining the aerodynamic part of the problem.

sp
[StructProblem class instance] An instance of the StructProblem class defining the structural part of the problem

`addVariablesPyOpt`(*optProb*)

Add the current set of variables to the optProb object.

Parameters

optProb
[pyOpt_optimization class] Optimization problem definition to which variables are added

`evalFunctions`(*funcs*, *evalFuncs*)

Evaluate functions of the AP and SP.

`evalFunctionsSens`(*funcsSens*, *evalFuncs*)

Evaluate the sensitivity of the desired functions

Parameters

funcsSens
[dict] Dictionary into which the function sensitivities are saved

evalFuncs
[iterable object containing strings] The functions that the user wants evaluated

setDesignVars(*x*)

Set the variables in the x-dict for this object.

Parameters

x
[dict] Dictionary of variables which may or may not contain the design variable names this object needs

1.5 MissionProblem

class baseclasses.**MissionProblem**(*name*, ***kwargs*)

Mission Problem Object:

This mission problem object should contain all of the information required to analyze a single mission. A mission problem is made of profiles. All profiles in a given mission problem must use consistent units.

Parameters

name
[str] A name for the mission

evalFuncs
[iteratble object containing strings] The names of the functions the user wants evaluated for this mission problem

Initialize the mission problem

addProfile(*profiles*)

Append a mission profile to the list. update the internal segment indices to correspond

addVariablesPyOpt(*pyOptProb*)

Add the current set of variables to the optProb object.

Parameters

optProb
[pyOpt_optimization class] Optimization problem definition to which variables are added

checkForProfileDVs()

Check if design variables have been added to this mission.

evalDVSens(*stepSize=1e-20*)

Evaluate the sensitivity of each of the 4 segment parameters (Mach, Alt) with respect to the design variables

getAltitudeCons(*CAS*, *mach*, *alt*)

Solve for the altitude at which CAS=mach

getAltitudeConsSens(*CAS*, *mach*, *alt*, *stepSize=1e-20*)

Solve for the altitude sensitivity at which CAS=mach

getNSeg()

return the number of segments in the mission

getSegments()

return a list of the segments in the mission in order

`setDesignVars`(*missionDVs*)

Pass the DVs to each of the profiles and have the profiles set the DVs

Parameters

missionDVs
[dict] Dictionary of variables which may or may not contain the design variable names this object needs

`setUnits`(*module*)

Set the units and the gravity constant for this mission.

`class` baseclasses.`MissionProfile`(*name*, *englishUnits=False*)

Mission Profile Object:

This Mission Profile Object contain an ordered set of segments that make up a single subsection of a mission. Start and end points of each segment in the profile are required to be continuous.

Initialize the mission profile

`addSegments`(*segments*)

Take in a list of segments and append it to the the current list. Check for consistency while we are at it.

`getSegmentParameters`()

Get the 4 segment parameters from each of the segment it owns Order is [M1, h1, M2, h2]

`setDesignVars`(*missionDVs*)

Set the variables for this mission profile

Parameters

missionDVs
[dict] Dictionary of variables which may or may not contain the design variable names this object needs

`class` baseclasses.`MissionSegment`(*phase*, ***kwargs*)

Mission Segment Object:

This is the basic building block of the mission solver.

Parameters

phase
[str] Segment type selector valid options include

`addDV`(*paramKey*, *lower=-1e+20*, *upper=1e+20*, *scale=1.0*, *name=None*)

Add one of the class attributes as a mission design variable. Typical variables are mach or velocity and altitude An error will be given if the requested DV is not allowed to be added .

Parameters

dvName
[str] Name used by the optimizer for this variables.

paramKey
[str] Name of variable to add. See above for possible ones

value
[float. Default is None] Initial value for variable. If not given, current value of the attribute will be used.

lower
[float. Default is None] Optimization lower bound. Default is unbonded.

upper
[float. Default is None] Optimization upper bound. Default is unbounded.

scale
[float. Default is 1.0] Set scaling parameter for the optimization to use.

name
[str. Default is None] Overwrite the name of this variable. This is typically only used when the user wishes to have multiple aeroProblems to explictly use the same design variable.

Examples

```
>>> # Add initMach variable with typical bounds
>>> seg.addDV('initMach', value=0.75, lower=0.0, upper=1.0, scale=1.0)
```

determineInputs()

Determine which of the four parameters (h, M, CAS, TAS) are inputs, which can be updated directly by the DV. For each end, there should be two inputs. At this point, the two beginning inputs should already be determined during initalization or by the MissionProfile.

propagateParameters()

Set the final V,M,h base on initial values and segType.

setMissionData(*module*, *segTypeDict*, *engTypeDict*, *idx*, *segIdx*)

set the data for the current segment in the fortran module

setParameters(*value*, *paramType*, *isInitVal*)

Design variable handling, where 'initMach' will be of paramType='Mach' and isInitVal=True, and the finalMach will be automatically adjusted if needed. Also determines if the previous or next segment will be affect as well

1.6 WeightProblem

class baseclasses.**WeightProblem**(*name*, *units*, ***kwargs*)

Weight Problem Object:

This Weight Problem Object should contain all of the information required to estimate the weight of a particular component configuration.

Parameters

name
[str] A name for the configuration

units
[str] Define the units that this weight problem will use. This set of units is transferred to all components when the are added to the weight problem. It is assumed that all user defined parameters provided to the components are in this unit system. Each component converts the user provided inputs from this unit system to the one used internally to perform calculations and then converts the output back to the user defined system.

evalFuncs
[iteratble object containing strings] The names of the functions the user wants evaluated for this weight problem

Initialize the mission problem

addComponents(*components*)

Append a list of components to the interal component list

addConstraintsPyOpt(*optProb=None*)

Add the linear constraints for each of the fuel cases.

Also add non-linear constraints that all of the fuel cases have a total TOW less than MTOW

Parameters

optProb
[pyOpt_optimization class] Optimization problem definition to which variables are added

addFuelCases(*cases*)

Append a list of fuel cases to the weight problem

addVariablesPyOpt(*optProb*)

Add the current set of variables to the optProb object.

Parameters

optProb
[pyOpt_optimization class] Optimization problem definition to which variables are added

getFuelCase(*caseName*)

Get the fuel case object associated with the caseName.

Parameters

caseName
[str] Name of the fuel case to return

getVarNames()

Get the variable names associate with this weight problem

resetFuelCase()

reset the fuel weight for this case.

setDVGeo(*DVGeo*)

Set the DVGeometry object that will manipulate this object. Note that pyWeight_problem doesn't **strictly** need a DVGeometry object set, but if optimization is desired it is required.

Parameters

dvGeo
[A DVGeometry object.] Object responsible for manipulating the constraints that this object is responsible for.

Examples

```
>>> wp.setDVGeo(DVGeo)
```

setDesignVars(*x*)

Set the variables in the x-dict for this object.

Parameters

x
[dict] Dictionary of variables which may or may not contain the design variable names this object needs

`setFuelCase`(*case*)

loop over the components and set the specified fuel case

Parameters

case
[fuelCase object] The fuel case to set

`setSurface`(*surf*)

Set the surface this configuratoin will use to perform projections for various components.

Parameters

surf
[pyGeo object or list] This is the surface representation to use for projections. If available, a pyGeo surface object can be used OR a triangulated surface in the form [p0, v1, v2] can be used. This triangulated surface form can be supplied from pyADflow or from pyTrian.

Examples

```
>>> CFDsolver = ADFLOW(comm=comm, options=aeroOptions)
>>> surf = CFDsolver.getTriangulatedMeshSurface()
>>> wp.setSurface(surf)
>>> # Or using a pyGeo surface object:
>>> surf = pyGeo('iges',fileName='wing.igs')
>>> wp.setSurface(surf)
```

`writeMassesTecplot`(*filename*)

Get a list of component keys based on inclusion and exclusion

Parameters

filename: str
filename for writing the masses. This string will have the .dat suffix appended to it.

`writeProblemData`(*fileName*)

Write the problem data to a file

`writeSurfaceTecplot`(*fileName*)

Write the triangulated surface mesh used in the weight_problem object to a tecplot file for visualization.

Parameters

fileName
[str] File name for tecplot file. Should have a .dat extension.

`writeTecplot`(*fileName*)

This function writes a visualization file for the components that have coordinates. All currently added components with coords are written to a tecplot file. This is useful for publication purposes as well as determine if the constraints are *actually* what the user expects them to be.

Parameters

fileName
[str] File name for tecplot file. Should have a .dat extension.

1.7 BaseSolver

class baseclasses.**BaseSolver**(*name*, *category*, *defaultOptions={}*, *options={}*, *immutableOptions={}*, *deprecatedOptions={}*, *comm=None*, *informs={}*, *checkDefaultOptions=True*, *caseSensitiveOptions=False*)

Abstract Class for a basic Solver Object

Solver Class Initialization

Parameters

name
[str] The name of the solver

category
[dict] The category of the solver

defaultOptions
[dict, optional] The default options dictionary

options
[dict, optional] The user-supplied options dictionary

immutableOptions
[set of strings, optional] A set of immutable option names, which cannot be modified after solver creation.

deprecatedOptions
[dict, optional] A dictionary containing deprecated option names, and a message to display if they were used.

comm
[MPI Communicator, optional] The comm object to be used. If none, serial execution is assumed.

informs
[dict, optional] A dictionary of exit code: exit message mappings.

checkDefaultOptions
[bool, optional] A flag to specify whether the default options should be used for error checking. This is used in cases where the default options are not the complete set, which is common for external solvers. In such cases, no error checking is done when calling `setOption`, but the default options are still set as options upon solver creation.

caseSensitiveOptions
[bool, optional] A flag to specify whether the option names are case sensitive or insensitive.

`getModifiedOptions()`

Prints a nicely formatted dictionary of all the modified solver options to the stdout on the root processor

`getOption`(*name*)

Default implementation of getOption()

Parameters

name
[str] Name of option to get. Not case sensitive

Returns

value
[varies] Return the current value of the option.

`pp`(*obj*, *flush=True*)

This method prints `obj` (via pprint) on the root proc of `self.comm` if it exists. Otherwise it will just print `obj`.

Parameters

obj
[object] Any Python object to be printed

flush
[bool] If True, the stream will be flushed.

`printModifiedOptions`()

Prints a nicely formatted dictionary of all the current solver options that have been modified from the defaults to the root processor

`printOptions`()

Prints a nicely formatted dictionary of all the current solver options to the stdout on the root processor

`setOption`(*name*, *value*)

Default implementation of setOption()

Parameters

name
[str] Name of option to set. Not case sensitive.

value
[varies] Value to set. Type is checked for consistency.

1.8 AeroSolver

`class` baseclasses.`AeroSolver`(*name*, *category*, *defaultOptions={}*, *options={}*, *immutableOptions={}*, *deprecatedOptions={}*, *comm=None*, *informs={}*)

Abstract Class for Aerodynamic Solver Object

AeroSolver Class Initialization

`addFamilyGroup`(*groupName*, *families*)

Add a custom grouping of families called groupName. The groupName must be distinct from the existing families. All families must in the 'families' list must be present in the CGNS file.

Parameters

groupName
[str] User-supplied custom name for the family groupings

families
[list] List of string. Family names to combine into the family group

`checkAdjointFailure`(*aeroProblem*, *funcsSens*)

Take in a an aeroProblem and check for adjoint failure, Then append the fail flag in funcsSens. Information regarding whether or not the last analysis with the aeroProblem was sucessful is included. This information is included as "funcsSens['fail']". If the 'fail' entry already exits in the dictionary the following operation is performed:

funcsSens[‘fail’] = funcsSens[‘fail’] or <did this problem fail>

In other words, if any one problem fails, the funcsSens[‘fail’] entry will be True. This information can then be used directly in multiPointSparse. For direct interface with pyOptSparse the fail flag needs to be returned separately from the funcs.

Parameters

aeroProblem
[pyAero_problem class] The aerodynamic problem to to get the solution for

funcsSens
[dict] Dictionary into which the functions are saved.

`checkSolutionFailure`(*aeroProblem*, *funcs*)

Take in a an aeroProblem and check for failure. Then append the fail flag in funcs. Information regarding whether or not the last analysis with the aeroProblem was sucessful is included. This information is included as “funcs[‘fail’]”. If the ‘fail’ entry already exits in the dictionary the following operation is performed:

funcs[‘fail’] = funcs[‘fail’] or <did this problem fail>

In other words, if any one problem fails, the funcs[‘fail’] entry will be True. This information can then be used directly in multiPointSparse. For direct interface with pyOptSparse the fail flag needs to be returned separately from the funcs.

Parameters

aeroProblem
[pyAero_problem class] The aerodynamic problem to to get the solution for

funcs
[dict] Dictionary into which the functions are saved.

`getForces`(*group_name*)

Return the set of forces at the locations defined by getSurfaceCoordinates

`getResNorms`()

Return the inital,starting and final residual norms for the solver

`getResidual`()

Return the reisudals on this processor.

`getSolution`()

Retrieve the solution dictionary from the solver

`getStateSize`()

Return the number of degrees of freedom (states) that are on this processor

`getStates`()

Return the states on this processor.

`getSurfaceCoordinates`(*group_name*)

Return the set of surface coordinates cooresponding to a Particular group name

`getTriangulatedMeshSurface`(*groupName=None*, ***kwargs*)

This function returns a trianguled verision of the surface mesh on all processors. The intent is to use this for doing constraints in DVConstraints.

Returns

surf
[list] List of points and vectors describing the surface. This may be passed directly to DVConstraint setSurface() function.

`globalNKPreCon`(*in_vec*)

Precondition the residual in in_vec for a coupled Newton-Krylov Method

`printFamilyList`()

Print a nicely formatted dictionary of the family names

`resetFlow`()

Reset the flow to a uniform state

`setDVGeo`(*DVGeo*, *pointSetKwargs=None*)

Set the DVGeometry object that will manipulate 'geometry' in this object. Note that <SOLVER> does not **strictly** need a DVGeometry object, but if optimization with geometric changes is desired, then it is required.

Parameters

DVGeo
[A DVGeometry object.] Object responsible for manipulating the geometry.

pointSetKwargs
[dict] Keyword arguments to be passed to the DVGeo addPointSet call. Useful for DVGeometryMulti, specifying FFD projection tolerances, etc.

Examples

```
>>> CFDsolver = <SOLVER>(comm=comm, options=CFDoptions)
>>> CFDsolver.setDVGeo(DVGeo)
```

`setMesh`(*mesh*)

Set the mesh object to the aero_solver to do geometric deformations

Parameters

mesh
[MBMesh or USMesh object] The mesh object for doing the warping

`setStates`(*states*)

Set the states on this processor.

`setSurfaceCoordinates`(*coordinates*, *groupName=None*)

Set the updated surface coordinates for a particular group.

Parameters

coordinates
[numpy array] Numpy array of size Nx3, where N is the number of coordinates on this processor. This array must have the same shape as the array obtained with getSurfaceCoordinates()

groupName
[str] Name of family or group of families for which to return coordinates for.

solveAdjoint(*objective*, **args*, ***kwargs*)

Solve the adjoint problem for the desired objective functions.

objectives - List of objective functions

totalAeroDerivative(*objective*)

Return the total derivative of the objective with respect to aerodynamic-only variables

totalSurfaceDerivative(*objective*)

Return the total derivative of the objective at surface coordinates

writeTriangulatedSurfaceTecplot(*fileName*, *groupName=None*, ***kwargs*)

Write the triangulated surface mesh from the solver in tecplot.

Parameters

fileName
[str] File name for tecplot file. Should have a .dat extension.

groupName
[str] Set of boundaries to include in the surface.

1.9 Testing

class baseclasses.**BaseRegTest**(*ref_file*, *train=False*, *comm=None*)

The class for handling regression tests.

Parameters

ref_file
[str] The name of the reference file, containing its full path.

train
[bool, optional] Whether to train the reference values, or test against existing reference values, by default False

comm
[MPI communicator, optional] The MPI comm if testing in parallel, by default None

check_arch
[bool, optional] Whether to check and set the appropriate PETSc arch prior to running tests, by default False. Note this option does not currently work.

add_metadata(*metadata*)

Add a metadata entry to the reference file, which is not used when checking reference values.

Parameters

metadata
[dict] The dictionary of metadata to add

assert_allclose(*actual*, *reference*, *name*, *rtol*, *atol*, *full_name=None*)

This is basically a wrapper on numpy.testing.assert_allclose with a generated error message

get_metadata()

Returns the metadata

Returns

dict
The metadata stored in the reference file

`par_add_norm`(*name*, *values*, ***kwargs*)

Add the norm across values from all processors.

Parameters

name
[str] The name of the value

values
[ndarray] The array to be added. This must be a numpy array distributed over self.comm

****kwargs**
See `getTol` on how to specif tolerances.

`par_add_sum`(*name*, *values*, ***kwargs*)

Add the sum of sum of the values from all processors.

Parameters

name
[str] The name of the value

values
[ndarray] The array to be added. This must be a numpy array distributed over self.comm

****kwargs**
See `getTol` on how to specif tolerances.

`par_add_val`(*name*, *values*, ***kwargs*)

Add value(values) from parallel process in sorted order

Parameters

name
[str] The name of the value

values
[ndarray] The array to be added. This must be a numpy array distributed over self.comm

****kwargs**
See `getTol` on how to specif tolerances.

`readRef`()

Read in the reference file on the root proc, then broadcast to all procs

`root_add_dict`(*name*, *d*, ***kwargs*)

Only write from the root proc

Parameters

name
[str] The name of the dictionary

d
[dict] The dictionary to add

****kwargs**
See `getTol` on how to specif tolerances.

root_add_val(*name*, *values*, ***kwargs*)

Add values but only on the root proc

Parameters

name
[str] the name of the value

values
[[type]] [description]

root_print(*s*)

Print a message on the root proc

Parameters

s
[str] The message to print

writeRef()

Write the reference file from the root proc

baseclasses.testing.**getTol**(***kwargs*)

Returns the tolerances based on kwargs. There are two ways of specifying tolerance:

1. pass in `tol` which will set `atol = rtol = tol`
2. individually set `atol` and `rtol`

If any values are unspecified, the default value will be used.

Parameters

atol
[float] absolute tolerance, default: 1E-12

rtol
[float] relative tolerance, default: 1E-12

tol
[float] tolerance. If specified, `atol` and `rtol` values are ignored and both set to this value

Returns

rtol
[float] relative tolerance

atol
[float] absolute tolerance

baseclasses.testing.**require_mpi**(*func*)

A decorator to skip tests unless `mpi4py` is available

Examples

```
@require_mpi
def test_mpi4py(self):
    print(self.comm.rank)
```

1.10 Regression Testing Example

baseclasses.BaseRegTest provides a framework for creating regression tests. It stores data in a JSON file format during *training* and compares test results against this stored data during *testing*. Here, we will go through a short example of how to integrate this into Pythons's `unittest` framework, specifically when used with `testflo`.

In this example, a regression test for a function called `sampling.polynomial` will be created. This function returns a `numpy` array of values that are spaced between a start and end point according to a polynomial distribution. The test function is structured as follows:

```
class TestSampling(unittest.TestCase):
    def test_polynomial(self, train=False):
        ref_file = os.path.join(baseDir, "ref/test_polynomial.ref")
        with BaseRegTest(ref_file, train=train) as handler:
            s = sampling.polynomial(0, 1, 100)
            handler.root_add_val("test_polynomial - Sample from Polynomial:", s, tol=1e-
↪10)
```

The key difference from a typical unit test is the optional `train` flag. When this flag is false then *baseclasses.BaseRegTest.root_add_val()* will check the reference file for the value associated with the given key and compare it to the array `s`. When train is set to true, *root_add_val()* will instead store the current value of `s` in the reference file with the given key. The `with` keyword is used to make sure that during training mode, the reference file is updated correctly at the end.

To quickly make the reference data for multiple regression tests we can create a corresponding train function:

```
def train_polynomial(self, train=True):
    test_polynomial(self, train=train)
```

Then, when running `testflo` we can specify to run all of the training functions using the `-m` flag.

```
testflo -m train_*
```

Normally, when `testflo` is run, it looks for all functions that begin with `test_`. The `-m` flag allows us to specify a different prefix. In this case, by using the `train_` prefix on all of the training functions we can run them all at once with the above command. Once the reference files are created, just calling `testflo` will run all the regression tests.

1.11 Utilities

class baseclasses.utils.**CaseInsensitiveDict**(**args*, ***kwargs*)

Python dictionary where the keys are case-insensitive. Note that this assumes the keys are strings, and indeed will fail if you try to create an instance where keys are not strings. All common Python dictionary operations are supported, and additional operations can be added easily. In order to preserve capitalization on key initialization, the implementation relies on storing a dictionary of mappings, which are used to check any new keys against existing keys and compare them in a case-insensitive fashion.

Warning: This container preserves the initial capitalization, such that any operation which operates on an existing entry will not modify it. This means that for example `__setitem__()` will NOT update the original capitalization.

Attributes

data
[dict] The equivalent case-sensitive dictionary. This stores the actual values.

map
[dict] Dictionary of mappings between the lowercase representation and the initial capitalization.

class baseclasses.utils.**CaseInsensitiveSet**(**args*, ***kwargs*)

Python set where the elements are case-insensitive. Note that this assumes the elements are strings, and indeed will fail if you try to create an instance where elements are not strings. All common Python set operations are supported, and additional operations can be added easily. In order to preserve capitalization on key initialization, the implementation relies on storing a dictionary of mappings which are used to check any new keys against existing keys and compare them in a case-insensitive fashion.

Warning: This container preserves the initial capitalization, such that any operation which operates on an existing entry will not modify it. This means that *add()* and *update()* will NOT update the original capitalization.

Attributes

data
[set] The equivalent case-sensitive set.

map
[dict] Dictionary of mappings between the lowercase representation and the initial capitalization.

add(*item*)

Add an element.

discard(*item*)

Remove an element. Do not raise an exception if absent.

issubset(*other*)

We convert both to regular set, and compare their lower case values

update(*d*)

Just call *add()* iteratively

exception baseclasses.utils.**Error**(*message*)

Format the error message in a box to make it clear this was a explicitly raised exception.

class baseclasses.utils.**SolverHistory**(*includeIter=True*, *includeTime=True*)

The SolverHistory class can be used to store and print various useful values during the execution of a solver.

NOTE: The implementation of this class contains no consideration of parallelism. If you are using a solverHistory object in your parallel solver, you will need to take care over which procs you make calls to the SolverHistory object on.

Create a solver history instance

Parameters

includeIter
[bool, optional] Whether to include the history's internal iteration variable in the history, by default True

includeTime
[bool, optional] Whether to include the history's internal timing variable in the history, by default True

addMetadata(*name*, *data*)

Add a piece of metadata to the history

The metadata attribute is simply a dictionary that can be used to store arbitrary information related to the solution being recorded, e.g solver options

Parameters

name
[str] Item name/key

data
[Any] Item to store

addVariable(*name*, *varType*, *printVar=False*, *valueFormat=None*, *overwrite=False*)

Define a new field to be stored in the history.

Parameters

name
[str] Variable name

varType
[Type] Variable type, i.e int, float, str etc

printVar
[bool, optional] Whether to include the variable in the iteration printout, by default False

valueFormat
[str, optional] Format string valid for use with the str.format() method (e.g "{:17.11e}" for a float or "{:03d}" for an int), only important for variables that are to be printed. By default a predefined format for the given *varType* is used

overwrite
[bool, optional] Whether to overwrite any existing variables with the same name, by default False

`getData()`

Get the recorded data

Returns

dict
Dictionary of recorded data

`getIter()`

Get the current number of iterations recorded

Returns

int
Number of iterations recorded

`getMetadata()`

Get the recorded metadata

Returns

dict
Dictionary of recorded metadata

`getVariables()`

Get the recorded variables

Returns

dict
Dictionary of recorded variables

`printData(`*iters=None*`)`

Print a selection of lines from the history

Each line will look something like this:

```
|      0  |  1.000e-01  |      -84     |   2.87098307489e-01  |      ...      |
```

Parameters

iters
[int or Iterable of ints, optional] Iteration numbers to print, by default only the last iteration will be printed

`printHeader()`

Print the header of the iteration printout

The header will look something like this:

```
+-------------------------------------------------------------------------...------+
|  Iter   |    Time     |  Random Int  |     Random Float     |      ...      |
+-------------------------------------------------------------------------...------+
```

`reset(`*clearMetadata=False*`)`

Reset the history to its initial state.

Parameters

clearMetadata
[bool, optional] Whether to clear the metadata too, by default False

`save`(*fileName*)

Write the solution history to a pickle file

Only the data dictionary is saved

Parameters

fileName
[str] File path to save the solution history to, file extension not required, will be ignored if supplied

`startTiming`()

Record the start time of the solver

This function only needs to be called explicitly if the start time of your solver is separate from the first time the *write* method is called.

`write`(*data*)

Record data for a single iteration

Note that each call to this method is treated as a new iteration. All data to be recorded for a single solver iteration must therefore be recorded in a single call to this method.

Parameters

data
[dict] Dictionary of values to record, with variable names as keys

`writeFullVariableHistory`(*name*, *values*)

Write the entire history of a variable in one go

This function should be used in the case where your solver already handles the recording of variables during a solution (e.g ADflow) but you want to use a SolverHistory object to facilitate writing it to a file

Parameters

name
[str] Variable name

values
[Iterable] Values to record, will be converted to a list

`baseclasses.utils.getPy3SafeString`(*string*)

Accepts a string and makes sure it's converted to unicode for python 3.6 and above

`baseclasses.utils.pp`(*obj*, *comm=None*, *flush=True*)

Parallel safe printing routine. This method prints `obj` (via pprint) on the root proc of `self.comm` if it exists. Otherwise it will just print `obj`.

Parameters

obj
[object] Any Python object to be printed

comm
[MPI comm] The MPI comm object on this processor

flush
[bool] If True, the stream will be flushed.

`baseclasses.utils.readJSON`(*fname*, *comm=None*)

Reads a JSON file and return the contents as a dictionary. This includes a custom NumPy reader to retrieve NumPy arrays, matching the *writeJSON()* function.

Parameters

file_name
[str] The file name

comm
[mpi4py.MPI.Comm, optional] The communicator over which this function is called. If supplied, only the root proc will be used for file IO.

References

This is based on this stack overflow answer

baseclasses.utils.**readPickle**(*fname*, *comm=None*)

This is a parallel pickle.load function, which is performed on the root proc only. Error checking is necessary to provide py2 compatibility.

Parameters

fname
[str] The pickle file name

comm
[mpi4py.MPI.Comm, optional] The communicator over which this function is called. If supplied, only the root proc will be used for file IO.

Returns

obj
[The object stored in the pickle file]

baseclasses.utils.**redirectIO**(*f_out*, *f_err=None*)

This function redirects stdout/stderr to the given file handle.

Parameters

f_out
[file] A file stream to redirect stdout to

f_err
[file] A file stream to redirect stderr to. If none is specified it is set to *f_out*

baseclasses.utils.**redirectingIO**(*f_out*, *f_err=None*)

A function that redirects stdout in a with block and returns to the stdout after the *with* block completes. The filestream passed to this function will be closed after exiting the *with* block.

Here is an example of usage where all adflow output is redirected to the file *adflow_out.txt*:

```
>>> from baseclasses.utils import redirectIO
>>> print("Printing some information to terminal")
>>> with redirectIO.redirectingIO(open("adflow_out.txt", "w")):
...     CFDSolver = ADFLOW(options=options)
...     CFDSolver(AeroProblem(**apOptions)
>>> print("Printing some more information to terminal")
```

Parameters

f_out
[file] A file stream that stdout should be redirected to

f_err
[file] A file stream to redirect stderr to. If none is specified it is set to *f_out*

baseclasses.utils.**writeJSON**(*fname*, *obj*, *comm=None*)

Write an object to a JSON file. This includes a custom NumPy encoder to reliably write NumPy arrays to JSON, which can then be read back via *readJSON()*.

Parameters

fname
[str] The file name

obj
[dict or ndarray] The object to be written to JSON

comm
[mpi4py.MPI.Comm, optional] The communicator over which this function is called. If supplied, only the root proc will be used for file IO.

baseclasses.utils.**writePickle**(*fname*, *obj*, *comm=None*)

Parallel pickle writing function, only performs operations on the root proc

Parameters

fname
[str] The pickle file name

obj
[any object which can be pickled by Python] The object to be pickled

comm
[mpi4py.MPI.Comm, optional] The communicator over which this function is called. If supplied, only the root proc will be used for file IO.

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

D

E

T

U

W