AVL Analysis Interface Module (AIM) Manual

Ed Alyanak and Ryan Durscher
AFRL/RQVC

May 22, 2024

0.1 Introduction

0.1.1 AVL AIM Overview

The use of lower-dimensional design tools is clearly desirable in a multidisciplinary/multi-fidelity aero design optimization setting. This is the crux of the Computational Aircraft Prototype Syntheses (CAPS) program. In many ways describing geometry appropriate for AVL (the Athena Vortex Lattice) code is more cumbersome than higher fidelity codes that require an Outer Mold Line. The goal is to make a CAPS AIM (Analysis Input Module) that directly feeds input to AVL and extracts the output quantities of interest from AVL's execution. This needs to be consistent with a build description that is hierarchical and multi-fidelity. That is, the build description that generates the geometric data at this level can be further enhanced to produce the complete OML of the aircraft design under consideration. As for the geometric description, AVL requires airfoil section data specified at the appropriate locations that describe the *skeleton* of the aircraft. These sections when *lofted* as groups and finally *unioned* together builds the OML. Clearly, intercepting the state of the geometry before these higher-level operations are applied provides the data appropriate for AVL. This naturally constructs a hierarchical geometric view where a design can progress into higher fidelities and feedback can be achieved where we can go back to this level of description when need be.

An outline of the AIM's inputs and outputs are provided in AIM Inputs and AIM Outputs, respectively.

Details on the use of units are outlined in AIM Units.

Geometric attribution that the AIM makes use is provided in AIM Attributes.

The AVL AIM can automatically execute avl, with details provided in AIM Execution.

To populate output data the AIM expects files, "capsTotalForce.txt", "capsStripForce.txt", "capsStatbilityDeriv.txt", "capsBodyAxisDeriv.txt", and "capsHingeMoment.txt" to exist after running AVL (see AIM Outputs for additional information). An example execution for AVL looks like:

0.1.2 Assumptions

The AVL coordinate system assumption (X – downstream, Y – out the right wing, Z – up) needs to be followed.

Within **OpenCSM** there are a number of airfoil generation UDPs (User Defined Primitives). These include NACA 4 series, a more general NACA 4/5/6 series generator, Sobieczky's PARSEC parameterization and Kulfan's CST parameterization. All of these UDPs generate **EGADS** *FaceBodies* where the *Face*'s underlying *Surface* is planar and the bounds of the *Face* is a closed set of *Edges* whose underlying *Curves* contain the airfoil shape. In all cases, there is a *Node* that represents the *Leading Edge* point and one or two *Nodes* at the *Trailing Edge* – one if the representation is for a sharp TE and the other if the definition is open or blunt. If there are 2 *Nodes* at the back, then there are 3 *Edges* all together and closed, even though the airfoil definition was left open at the TE. All of this information will be used to automatically fill in the AVL geometry description.

The AVL Sections are automatically generated, one from each *FaceBody* and the details extracted from the geometry. The *FaceBody* must contain at least two edges and two nodes, but may contain any number of *Edges* otherwise. If the *FaceBody* contains more nodes, the node with the smallest **x** value is used to define the leading edge, the node with the largest **x** defines the trailing edge. The airfoil may have a single *Edge* that defines a straight blunt trailing edge. **Xle**, **Yle**, and **Zle**, are taken from the *Node* associated with the *Leading Edge*. The **Chord** is computed by getting the distance between the LE and TE (if there is a blunt trainling *Edge* in the *FaceBody* the TE point is considered the mid-position on that *Edge*). **Ainc** is computed by registering the chordal direction of the *FaceBody* against the X-Z plane. The airfoil shapes are generated by sampling the *Curves* and put directly in the input file via the **AIRFOIL** keyword after being normalized.

It should be noted that general construction in either **OpenCSM** or even **EGADS** will be supported as long as the topology described above is used. But care should be taken when constructing the airfoil shape so that a discontinuity (i.e., simply C^0) is not generated at the *Node* representing the *Leading Edge*. This can be done by splining the entire shape as one and then intersecting the single *Edge* to place the LE *Node*.

The rest of the information and options required to fill out the AVL geometry input file (**xxx.avl**) will be found in the attributes attached to the *FaceBody* itself. The conventions used will be described in the next section.

Also note that this first implementation is not intended to provide complete control over AVL. In particular, there is no mention above of the **BODY**, **DESIGN**, **CLAF**, or **CDCL** AVL keywords.

0.1.3 Examples

An example problem using the AVL AIM may be found at AVL AIM Examples, which contains example ∗.csm input files and pyCAPS scripts designed to make use of the AVL AIM. These example scripts make extensive use of the AIM Attributes, AIM Inputs, and AIM Outputs.

0.2 AVL AIM Examples

This example contains a set of ∗.csm and pyCAPS (∗.py) inputs that uses the AVL AIM. A user should have knowledge on the generation of parametric geometry in Engineering Sketch Pad (ESP) before attempting to integrate with any AIM. Specifically, this example makes use of Design Parameters, Set Parameters, User Defined Primitive (UDP) and attributes in ESP.

The follow code details the process in a ∗.csm file that generates three airfoil sections to create a wing. Note to execute in serveESP a dictionary file must be included "serveESP $ESP_ROOT/CAPSexamples/csmData/avl↩ Wing.csm"

The CSM script generates Bodies which are designed to be used by specific AIMs. The AIMs that the Body is designed for is communicated to the CAPS framework via the capsAIM string attribute. This is a semicolon-separated string with the list of AIM names. Thus, the CSM author can give a clear indication to which AIMs should use the Body. In this example, the list contains only the avlAIM:

```
attribute capsAIM     $avlAIM
```

Next we will define the design parameters to define the wing cross section and planform.

```
despmtr   thick     0.12      frac of local chord
despmtr   camber    0.04      frac of loacl chord
despmtr   area      10.0      Planform area of the full span wing
despmtr   aspect    6.00      Span^2/Area
despmtr   taper     0.60      TipChord/RootChord
despmtr   sweep     20.0      1/4 Chord Sweep
despmtr   washout  -5.00      deg (negative is down at tip)
despmtr   dihedral  4.00      deg
```

The design parameters will then be used to set parameters for use internally to create geometry.

```
set       span      sqrt(aspect*area)
set       croot     2*area/span/(1+taper)
set       ctip      croot*taper
set       dxtip     (croot-ctip)/4+span/2*tand(sweep)
set       dztip     span/2*tand(dihedral)
```

Finally, the airfoils are created using the User Defined Primitive (UDP) naca. The inputs used for this example to the UDP are Thickness and Camber. Cross sections are in the X-Y plane and are rotated to the X-Z plane. Reference quantities must exist on any body, otherwise AVL defaults to 1.0 for Area, Span, Chord and 0.0 for X,Y,Z moment References

```
# left tip
udprim    naca      Thickness thick     Camber    camber
attribute capsGroup    $Wing
attribute capsReferenceArea  area
attribute capsReferenceSpan  span
attribute capsReferenceChord croot
attribute capsReferenceX     croot/4
scale     ctip
rotatex   90        0         0
rotatey   washout   0         ctip/4
translate dxtip    -span/2    dztip
# root
udprim    naca      Thickness thick     Camber    camber
attribute capsGroup    $Wing
rotatex   90        0         0
scale     croot
# right tip
udprim    naca      Thickness thick     Camber    camber
attribute capsGroup    $Wing
scale     ctip
rotatex   90        0         0
rotatey   washout   0         ctip/4
translate dxtip     span/2    dztip
```

An example pyCAPS script that uses the above csm file to run AVL is as follows.

First the pyCAPS and os module needs to be imported.
```
import pyCAPS
import os
```

Next the ∗.csm file is loaded and design parameter is changed - area in the geometry. Any despmtr from the avl←Wing.csm file is available inside the pyCAPS script. They are: thick, camber, area, aspect, taper, sweep, washout, dihedral...
```
geometryScript = os.path.join("..","csmData","avlWing.csm")
myProblem = pyCAPS.Problem(problemName=workDir,
                           capsFile=geometryScript,
                           outLevel=args.outLevel)
myProblem.geometry.despmtr.area = 10.0
```

The AVL AIM is then loaded with:
```
myAnalysis = myProblem.analysis.create(aim = "avlAIM", name = "avl")
```

After the AIM is loaded the Mach number and angle of attack are set, though all AIM Inputs are available.
```
myAnalysis.input.Mach  = 0.5
myAnalysis.input.Alpha = 1.0
myAnalysis.input.Beta  = 0.0
wing = {"groupName"    : "Wing", # Notice Wing is the value for the capsGroup attribute
        "numChord"     : 8,
        "spaceChord"   : 1.0,
        "numSpanTotal" : 24,
        "spaceSpan"    : 1.0}
myAnalysis.input.AVL_Surface = {"Wing": wing}
```

Once all the inputs have been set, outputs can be directly requested. The avl analysis will be automatically executed just-in-time (AIM Execution).

Any of the AIM's output variables (AIM Outputs) are readily available; for example,
```
print ("CXtot  ", myAnalysis.output["CXtot" ].value)
print ("CYtot  ", myAnalysis.output["CYtot" ].value)
print ("CZtot  ", myAnalysis.output["CZtot" ].value)
print ("Cltot  ", myAnalysis.output["Cltot" ].value)
print ("Cmtot  ", myAnalysis.output["Cmtot" ].value)
print ("Cntot  ", myAnalysis.output["Cntot" ].value)
print ("Cl'tot ", myAnalysis.output["Cl'tot"].value)
print ("Cn'tot ", myAnalysis.output["Cn'tot"].value)
print ("CLtot  ", myAnalysis.output["CLtot" ].value)
print ("CDtot  ", myAnalysis.output["CDtot" ].value)
print ("CDvis  ", myAnalysis.output["CDvis" ].value)
print ("CLff   ", myAnalysis.output["CLff"  ].value)
print ("CYff   ", myAnalysis.output["CYff"  ].value)
print ("CDind  ", myAnalysis.output["CDind" ].value)
print ("CDff   ", myAnalysis.output["CDff"  ].value)
print ("e      ", myAnalysis.output["e"     ].value)
```

results in
```
CXtot   0.00061
CYtot   -0.0
CZtot   -0.30129
Cltot   -0.0
Cmtot   -0.19449
Cntot   -0.0
Cl'tot  -0.0
Cn'tot  -0.0
CLtot   0.30126
CDtot   0.00465
CDvis   0.0
CLff    0.30096
CYff    -0.0
CDind   0.0046467
CDff    0.0049692
e       0.967
```

Additionally, besides making a call to the AIM outputs, sensitivity values may be obtained in the following manner,
```
#sensitivity = myAnalysis.output["CLtot"].deriv("Alpha")
```

The avlAIM supports the control surface modeling functionality inside AVL. Trailing edge control surfaces can be added to the above example by making use of the vlmControlName attribute (see AIM Attributes regarding the attribution specifics). To add a **RightFlap** and **LeftFlap** to the previous example ∗.csm file the naca UDP entries are augmented with the following attributes.
```
# left tip
```

```
udprim    naca      Thickness thick     Camber    camber
attribute vlmControl_LeftFlap 80 # Hinge line is at 80% of the chord
...
# root
udprim    naca      Thickness thick     Camber    camber
attribute vlmControl_LeftFlap 80 # Hinge line is at 80% of the chord
attribute vlmControl_RightFlap 80 # Hinge line is at 80% of the chord
...
# right tip
udprim    naca      Thickness thick     Camber    camber
attribute vlmControl_RightFlap 80 # Hinge line is at 80% of the chord
...
```

Note how the root airfoil contains two attributes for both the left and right flaps.

In the pyCAPS script the AIM Inputs, **AVL_Control**, must be defined.

```
flap = {"controlGain"     : 0.5,
        "deflectionAngle" : 10.0}
 myAnalysis.input.AVL_Control = {"LeftFlap": flap, "RightFlap": flap}
```

Notice how the information defined in the **flap** variable is assigned to the vlmControl**Name** portion of the attributes added to the ∗.csm file.

0.3 AIM Attributes

The following list of attributes drives the AVL geometric definition. Each *FaceBody* which relates to AVL **Sections** will be marked up in an appropriate manner to drive the input file construction. Many attributes are required and those that are optional are marked so in the description:

- **capsReferenceArea** This attribute may exist on any *Body*. Its value will be used as the SREF entry in the AVL input.
- **capsReferenceChord** This attribute may exist on any *Body*. Its value will be used as the CREF entry in the AVL input.
- **capsReferenceSpan** This attribute may exist on any *Body*. Its value will be used as the BREF entry in the AVL input.
- **capsReferenceX** [Optional: Default 0.0] This attribute may exist on any *Body*. Its value will be used as the Xref entry in the AVL input.
- **capsReferenceY** [Optional: Default 0.0] This attribute may exist on any *Body*. Its value will be used as the Yref entry in the AVL input.
- **capsReferenceZ** [Optional: Default 0.0] This attribute may exist on any *Body*. Its value will be used as the Zref entry in the AVL input.
- **capsGroup** This string attribute labels the *FaceBody* as to which AVL Surface the section is assigned. This should be something like: *Main_Wing*, *Horizontal_Tail*, etc. This informs the AVL AIM to collect all *FaceBodies* that match this attribute into a single AVL Surface.
- **vlmControl"Name"** This string attribute attaches a control surface to the *FaceBody*. The hinge location is defined as the double value between 0 or 1.0. The range as percentage from 0 to 100 will also work. The name of the control surface is the string information after vlmControl (or vlmControl_). For Example, to define a control surface named Aileron the following are identical (*attribute vlmControlAileron 0.8* or *attribute vlmControl_Aileron 80*) . Multiple *vlmControl* attributes, with different names, can be defined on a single *FaceBody*.

  By default control surfaces with percentages less than 0.5 ($<$ 50%) are considered leading edge flaps, while values greater than or equal to 0.5 ($>=$ 50%) are considered trailing edge flaps. This behavior may be overwritten when setting up the control surface in "AVL_Control" (see AIM Inputs) with the keyword "leOrTe" (see Vortex Lattice Control Surface for additional details).

- **vlmNumSpan** This attribute may be set on any given airfoil cross-section to overwrite the number of spanwise horseshoe vortices placed on the surface (globally set - see keyword "numSpanPerSection" and "numSpan↩ Total" in Vortex Lattice Surface) between two sections. Note, that the AIM internally sorts the sections in ascending y (or z) order, so care should be taken to select the correct section for the desired intent.

Note: The attribute **avlNumSpan** has been deprecated in favor of **vlmNumSpan**

- **vlmSspace** This attribute may be set on any given airfoil cross-section in the range [-1 .. 1] specify the spanwise distribution function.

0.4 AIM Units

A unit system may be optionally specified during AIM instance initiation. If a unit system is provided, all AIM input values which have associated units must be specified as well. If no unit system is used, AIM inputs, which otherwise would require units, will be assumed unit consistent. A unit system may be specified via a JSON string dictionary for example: unitSys = "{"mass": "kg", "length": "m", "time":"seconds", "temperature": "K"}"

0.4.1 JSON String Dictionary

The key arguments of the dictionary are described in the following:

- **mass = "None"**
  Mass units - e.g. "kilogram", "k", "slug", ...

- **length = "None"**
  Length units - e.g. "meter", "m", "inch", "in", "mile", ...

- **time = "None"**
  Time units - e.g. "second", "s", "minute", ...

- **temperature = "None"**
  Temperature units - e.g. "Kelvin", "K", "degC", ...

0.5 AIM Inputs

The following list outlines the AVL inputs along with their default value available through the AIM interface.

- **Mach = 0.0**
  Mach number.
- **Alpha = NULL**
  Angle of attack [degree]. Either CL or Alpha must be defined but not both.
- **Beta = 0.0**
  Sideslip angle [degree].

- **RollRate = 0.0**
  Non-dimensional roll rate.
- **PitchRate = 0.0**
  Non-dimensional pitch rate.
- **YawRate = 0.0**
  Non-dimensional yaw rate.
- **CDp = 0.0**
  A fixed value of profile drag to be added to all simulations.
- **CLAF = NULL (double)**
  This scales the effective dcl/da of the section airfoil as follows:

  dcl/da = 2 pi (1 + CLaf t/c)

  where t/c is the airfoil's thickness/chord ratio. The intent is to better represent the lift characteristics of thick airfoils, which typically have greater dcl/da values than thin airfoils.

  A good 2D potential flow theory estimate for CLaf is 0.77.
  In practice, viscous effects will reduce the 0.77 factor to something less. Wind tunnel airfoil data or viscous airfoil calculations should be consulted before choosing a suitable CLaf value.

  This option is applied to all surface, and also be specified for individual surfaces using AVL_Surface Vortex Lattice Surface.
- **AVL_Surface = NULL**
  See Vortex Lattice Surface for additional details.
- **AVL_Control = NULL**
  See Vortex Lattice Control Surface for additional details.
- **CL = NULL**
  Coefficient of Lift. AVL will solve for Angle of Attack. Either CL or Alpha must be defined but not both.
- **Moment_Center = NULL, [0.0, 0.0, 0.0]**
  Array values correspond to the Xref, Yref, and Zref variables. Alternatively, the geometry (body) attributes "capsReferenceX", "capsReferenceY", and "capsReferenceZ" may be used to specify the X-, Y-, and Z-reference centers, respectively (note: values set through the AIM input will supersede the attribution values).
- **MassProp = NULL**
  Mass properties used for eigen value analysis Structure for the mass property tuple = ("Name", "Value"). The "Name" of the mass component used for documenting the xxx.mass file. The value is a JSON dictionary with values with unit pairs for mass, CG, and moments of inertia information (e.g. "Value" = {"mass" : [mass,"kg"], "CG" : [[x,y,z],"m"], "massInertia" : [[Ixx, Iyy, Izz, Ixy, Ixz, Iyz], "kg∗m2"]}) The components Ixy, Ixz, and Iyz are optional may be omitted. Must be in units of kg, m, and kg∗m^2 if unitSystem (see AIM Units) is not specified and no units should be specified in the JSON dictionary.
- **MassPropLink = NULL**
  Mass properties linked from structural analysis for eigen value analysis Must be in units of kg, m, and kg∗m^2 if unitSystem (see AIM Units) is not specified.
- **Gravity = NULL**
  Magnitude of the gravitational force used for Eigen value analysis. Must be in units of m/s^2 if unitSystem (see AIM Units) is not specified.
- **Density = NULL**
  Air density used for Eigen value analysis. Must be in units of kg/m^3 if unitSystem (see AIM Units) is not specified.
- **Velocity = NULL**
  Velocity used for Eigen value analysis. Must be in units of m/s if unitSystem (see AIM Units) is not specified.

0.6 AIM Execution

If auto execution is enabled when creating an AVL AIM, the AIM will execute avl just-in-time with the command line:

```
avl caps < avlInput.txt > avlOutput.txt
```

where preAnalysis generated the two files: 1) "avlInput.txt" which contains the input information and control sequence for AVL to execute and 2) "caps.avl" which contains the geometry to be analyzed.

The analysis can be also be explicitly executed with caps_execute in the C-API or via Analysis.runAnalysis in the pyCAPS API.

Calling preAnalysis and postAnalysis is NOT allowed when auto execution is enabled.

Auto execution can also be disabled when creating an AVL AIM object. In this mode, caps_execute and Analysis.↩ runAnalysis can be used to run the analysis, or avl can be executed by calling preAnalysis, system call, and pos↩ Analysis as demonstrated below with a pyCAPS example:

```
print ("\n\preAnalysis......")
avl.preAnalysis()
print ("\n\nRunning......")
avl.system("avl caps < avlInput.txt > avlOutput.txt"); # Run via system call
print ("\n\postAnalysis......")
avl.postAnalysis()
```

0.7 AIM Outputs

Optional outputs that echo the inputs. These are parsed from the resulting output and can be used as a sanity check.

- **Alpha** = Angle of attack.
- **Beta** = Sideslip angle.
- **Mach** = Mach number.
- **pb/2V** = Non-dimensional roll rate.
- **qc/2V** = Non-dimensional pitch rate.
- **rb/2V** = Non-dimensional yaw rate.
- **p'b/2V** = Non-dimensional roll acceleration.
- **r'b/2V** = Non-dimensional yaw acceleration.

Forces and moments:

- **CXtot** = X-component of total force in body axis
- **CYtot** = Y-component of total force in body axis
- **CZtot** = Z-component of total force in body axis
- **Cltot** = X-component of moment in body axis
- **Cmtot** = Y-component of moment in body axis
- **Cntot** = Z-component of moment in body axis
- **Cl'tot** = x-component of moment in stability axis

- **Cn'tot** = z-component of moment in stability axis
- **CLtot** = total lift in stability axis
- **CDtot** = total drag in stability axis
- **CDvis** = viscous drag component
- **CLff** = trefftz plane lift force
- **CYff** = trefftz plane side force
- **CDind** = induced drag force
- **CDff** = trefftz plane drag force
- **e =** Oswald Efficiency

Stability-axis derivatives - Alpha:

- **CLa** = z' force, CL, with respect to alpha.
- **CYa** = y force, CY, with respect to alpha.
- **Cl'a** = x' moment, Cl', with respect to alpha.
- **Cma** = y moment, Cm, with respect to alpha.
- **Cn'a** = z' moment, Cn', with respect to alpha.

Stability-axis derivatives - Beta:

- **CLb** = z' force, CL, with respect to beta.
- **CYb** = y force, CY, with respect to beta.
- **Cl'b** = x' moment, Cl', with respect to beta.
- **Cmb** = y moment, Cm, with respect to beta.
- **Cn'b** = z' moment, Cn', with respect to beta.

Stability-axis derivatives - Roll rate, p':

- **CLp'** = z' force, CL, with respect to roll rate, p'.
- **CYp'** = y force, CY, with respect to roll rate, p'.
- **Cl'p'** = x' moment, Cl', with respect to roll rate, p'.
- **Cmp'** = y moment, Cm, with respect to roll rate, p'.
- **Cn'p'** = z' moment, Cn', with respect to roll rate, p'.

Stability-axis derivatives - Pitch rate, q':

- **CLq'** = z' force, CL, with respect to pitch rate, q'.
- **CYq'** = y force, CY, with respect to pitch rate, q'.
- **Cl'q'** = x' moment, Cl', with respect to pitch rate, q'.

- **Cmq'** = y moment, Cm, with respect to pitch rate, q'.
- **Cn'q'** = z' moment, Cn', with respect to pitch rate, q'.

Stability-axis derivatives - Yaw rate, r':

- **CLr'** = z' force, CL, with respect to yaw rate, r'.
- **CYr'** = y force, CY, with respect to yaw rate, r'.
- **Cl'r'** = x' moment, Cl', with respect to yaw rate, r'.
- **Cmr'** = y moment, Cm, with respect to yaw rate, r'.
- **Cn'r'** = z' moment, Cn', with respect to yaw rate, r'.

Body-axis derivatives - Axial velocity, u:

- **CXu** = x force, CX, with respect to axial velocity, u.
- **CYu** = y force, CY, with respect to axial velocity, u.
- **CZu** = z force, CZ, with respect to axial velocity, u.
- **Clu** = x moment, Cl, with respect to axial velocity, u.
- **Cmu** = y moment, Cm, with respect to axial velocity, u.
- **Cnu** = z moment, Cn, with respect to axial velocity, u.

Body-axis derivatives - Sideslip velocity, v:

- **CXv** = x force, CX, with respect to sideslip velocity, v.
- **CYv** = y force, CY, with respect to sideslip velocity, v.
- **CZv** = z force, CZ, with respect to sideslip velocity, v.
- **Clv** = x moment, Cl, with respect to sideslip velocity, v.
- **Cmv** = y moment, Cm, with respect to sideslip velocity, v.
- **Cnv** = z moment, Cn, with respect to sideslip velocity, v.

Body-axis derivatives - Normal velocity, w:

- **CXw** = x force, CX, with respect to normal velocity, w.
- **CYw** = y force, CY, with respect to normal velocity, w.
- **CZw** = z force, CZ, with respect to normal velocity, w.
- **Clw** = x moment, Cl, with respect to normal velocity, w.
- **Cmw** = y moment, Cm, with respect to normal velocity, w.
- **Cnw** = z moment, Cn, with respect to normal velocity, w.

Body-axis derivatives - Roll rate, p:

- **CXp** = x force, CX, with respect to roll rate, p.
- **CYp** = y force, CY, with respect to roll rate, p.
- **CZp** = z force, CZ, with respect to roll rate, p.
- **Clp** = x moment, Cl, with respect to roll rate, p.
- **Cmp** = y moment, Cm, with respect to roll rate, p.
- **Cnp** = z moment, Cn, with respect to roll rate, p.

Body-axis derivatives - Pitch rate, q:

- **CXq** = x force, CX, with respect to pitch rate, q.
- **CYq** = y force, CY, with respect to pitch rate, q.
- **CZq** = z force, CZ, with respect to pitch rate, q.
- **Clq** = x moment, Cl, with respect to pitch rate, q.
- **Cmq** = y moment, Cm, with respect to pitch rate, q.
- **Cnq** = z moment, Cn, with respect to pitch rate, q.

Body-axis derivatives - Yaw rate, r:

- **CXr** = x force, CX, with respect to yaw rate, r.
- **CYr** = y force, CY, with respect to yaw rate, r.
- **CZr** = z force, CZ, with respect to yaw rate, r.
- **Clr** = x moment, Cl, with respect to yaw rate, r.
- **Cmr** = y moment, Cm, with respect to yaw rate, r.
- **Cnr** = z moment, Cn, with respect to yaw rate, r.

Geometric output:

- **Xnp** = Neutral Point
- **Xcg** = x CG location
- **Ycg** = y CG location
- **Zcg** = z CG location

Controls:

- **ControlStability** = a (or an array of) tuple(s) with a structure of ("Control Surface Name", "JSON Dictionary") for all control surfaces in the stability axis frame. The JSON dictionary has the form = {"CLtot":value,"CYtot"↩ :value,"Cl'tot":value,"Cmtot":value,"Cn'tot":value}
- **ControlBody** = a (or an array of) tuple(s) with a structure of ("Control Surface Name", "JSON Dictionary") for all control surfaces in the body axis frame. The JSON dictionary has the form = {"CXtot":value,"CYtot"↩ :value,"CZtot":value,"Cltot":value,"Cmtot":value,"Cntot":value}
- **HingeMoment** = a (or an array of) tuple(s) with a structure of ("Control Surface Name", "HingeMoment")
- **StripForces** = a (or an array of) tuple(s) with a structure of ("Surface Name", "JSON Dictionary") for all surfaces. The JSON dictionary has the form = {"cl":[value0,value1,value2],"cd":[value0,value1,value2]...}
- **EigenValues** = a (or an array of) tuple(s) with a structure of ("case #", "Array of eigen values"). The array of eigen values is of the form = [[real0,imaginary0],[real0,imaginary0],...]

0.8 Vortex Lattice Surface

Structure for the Vortex Lattice Surface tuple = ("Name of Surface", "Value"). "Name of surface defines the name of the surface in which the data should be applied. The "Value" can either be a JSON String dictionary (see Section JSON String Dictionary) or a single string keyword string (see Section Single Value String).

0.8.1 JSON String Dictionary

If "Value" is a JSON string dictionary (eg. "Value" = {"numChord": 5, "spaceChord": 1.0, "numSpan": 10, "space↩Span": 0.5}) the following keywords (= default values) may be used:

- **groupName = "(no default)"**
  Single or list of *capsGroup* names used to define the surface (e.g. "Name1" or ["Name1","Name2",...]. If no groupName variable is provided an attempted will be made to use the tuple name instead;

- **noKeyword = "(no default)"**
  "No" type. Options: NOWAKE, NOALBE, NOLOAD.

- **numChord = 10**
  The number of chordwise horseshoe vortices placed on the surface.

- **spaceChord = 1.0**
  The chordwise vortex spacing parameter.

- **numSpanTotal = 0**
  Total number of spanwise horseshoe vortices placed on the surface. The vorticies are 'evenly' distributed across sections to minimize jumps in spacings. numpSpanPerSection must be zero if this is set.

- **numSpanPerSection = 0**
  The number of spanwise horseshoe vortices placed on each section the surface. The total number of spanwise vorticies are (numSection-1)∗numSpanPerSection. The vorticies are 'evenly' distributed across sections to minimize jumps in spacings. numSpanTotal must be zero if this is set.

- **spaceSpan = 0.0**
  The spanwise vortex spacing parameter.

- **yMirror = False**
  Mirror surface about the y-direction.

- **CLaf = 0.0**
  This scales the effective dcl/da of the section airfoil as follows:

  dcl/da = 2 pi (1 + CLaf t/c)

  where t/c is the airfoil's thickness/chord ratio.

0.8.2 Single Value String

If "Value" is a single string the following options maybe used:

- (NONE Currently)

0.9 Vortex Lattice Control Surface

Structure for the Vortex Lattice Control Surface tuple = ("Name of Control Surface", "Value"). "Name of control surface defines the name of the control surface in which the data should be applied. The "Value" must be a JSON String dictionary (see Section JSON String Dictionary).

0.9.1 JSON String Dictionary

If "Value" is a JSON string dictionary (e.g. "Value" = {"deflectionAngle": 10.0}) the following keywords (= default values) may be used:

- **deflectionAngle = 0.0**
  Deflection angle of the control surface.

- **leOrTe = (no default)**
  Is the control surface a leading (= 0) or trailing (> 0) edge effector? Overrides the assumed default value set by the geometry: If the percentage along the airfoil chord is < 50% a leading edge flap is assumed, while >= 50% indicates a trailing edge flap.

- **controlGain = 1.0**
  Control deflection gain, units: degrees deflection / control variable

- **hingeLine = [0.0 0.0 0.0]**
  Alternative vector giving hinge axis about which surface rotates

- **deflectionDup = 0**
  Sign of deflection for duplicated surface

0.9.2 Single Value String

If "Value" is a single string, the following options maybe used:

- (NONE Currently)