



Computational Aircraft Prototype Syntheses

AIM Programming – Sensitivities

For ESP Rev 1.29

Bob Haimes

bob@geocentrictech.com or haimes@mit.edu

Geocentric Technologies LLC

Geometric Sensitivities

- Provides access to OpenCSM's sensitivity calculations
- Accomplished one Design Parameter at a time
- pyCAPS user has access through built-in *dataSets* named with the CSM Design Parameter – seen in *exercises/session14* by running `sensitivity.py`.

Output Values – derivatives

- The CAPS Value structure has slots to store derivative information if available from the analysis at-hand
- This should be populated when the value is set

AIM Helper Functions

Sensitivity Access

- provides useful functions for the AIM programmer
- gives access to CAPS Object data
- provides a dynamically loadable writer interface for dealing with large meshes
- note that all function names begin with `aim_`
- prototypes defined in `$ESP_ROOT/include/aimUtil.h`:
- if any of these functions are used, then the library must be included (`libaimUtil.a/aimUtil.lib`) in the AIM so/DLL build

Define Parameter Associated with Sensitivities

```
icode = aim_setSensitivity(void *aimInfo, const char *GIname,  
                           int *irow, int *icol)
```

aimInfo the AIM context

GIname the pointer to the string that matches the *Geometry Input* Parameter name

irow the parameter row to use – 1 bias

icol the parameter column to use – 1 bias

icode integer return code

Notes:

- 1 **aim_newTess** must have been invoked sometime before calling this function to set the tessellations for the Bodies of interest
- 2 Call **aim_setSensitivity** before call(s) to **aim_getSensitivity**.

Get Sensitivities based on Tessellation Components

```
icode = aim_getSensitivity(void *aimInfo, ego tess, int ttype,  
                           int index, int *npts, double **dxyz)
```

aimInfo the AIM context

tess the EGADS Tessellation Object

ttype topological type – 0 - NODE, *Tessellation Sensitivities*: 1 - EDGE, 2 - FACE
Geometric Sensitivities: -1 - EDGE, -2 - FACE

index the index in the Body (associated with the tessellation) based on the *type* (bias 1)

npts the returned number of sensitivities (number of tessellation points)

dxyz a pointer to the returned sensitivities – 3*npts in length (*freeable*)

icode integer return code

Note:

- Call `aim_setSensitivity` before call(s) to `aim_getSensitivity`

Get Global Tessellation Sensitivities

```
icode = aim_tessSensitivity(void *aimInfo, const char *name,  
                           int irow, int icol, ego tess, int *npts,  
                           double **dxyz)
```

aimInfo the AIM context

name the pointer to the string that matches the *Geometry Input* Parameter name

irow the parameter row to use – 1 bias

icol the parameter column to use – 1 bias

tess the EGADS Tessellation Object

npts the returned number of sensitivities (number of global vertices)

dxyz a pointer to the returned sensitivities – 3*npts in length (*freeable*)

icode integer return code

Notes:

- 1 Used to get the tessellation sensitivities for the entire Tessellation Object
- 2 The number of points is the global number of vertices in the tessellation
- 3 This function does not require that `aim_setSensitivity` be called first

Set Step Size for Sensitivities

```
icode = aim_setStepSize(void *aimInfo, double step)
```

aimInfo the AIM context

step the step size used for subsequent AIM-based sensitivity calculations
minus indicates defaulting to CAPS, 0.0 is for analytic, positive sets the finite difference step size

icode integer return code

Get Step Size for Sensitivities

```
icode = aim_getStepSize(void *aimInfo, double *step)
```

aimInfo the AIM context

step the step size used for subsequent AIM-based sensitivity calculations
minus indicates defaulting to CAPS, 0.0 is for analytic, positive sets the finite difference step size

icode integer return code

These functions should only be used for debugging

Filling in the Value Structure

```
char    *sensVar;                // filled with WRT string
double  value_dot;              // filled with the derivative

/* only 1 derivative (WRT sensVar) */
val->nderiv = 1;                 // only 1 derivative of rank 1
AIM_ALLOC(val->derivs, val->nderiv, capsDeriv, aimInfo, status);
val->derivs[0].name = NULL;
val->derivs[0].deriv = NULL;

/* set what the first (only) derivative is with respect to */
AIM_STRDUP(val->derivs[0].name, sensVar, aimInfo, status);

/* the rank (i.e., area, volume, and mass all have rank 1) */
val->derivs[0].len_wrt = 1;

/* allocate storage for the derivative(s) */
length = val->length * val->derivs[0].len_wrt;
AIM_ALLOC(val->derivs[0].deriv, length, double, aimInfo, status);

/* fill it */
val->derivs[0].deriv[0] = value_dot;
```

Sensitivities

Setup the AIM to deal with sensitivities through `tankCalc`

- Select an `GeometryIn` or `AnalysisIn` to get the derivative WRT. This should be in the form of a string value.
- Get the geometry *dot* information via the routines listed above. The dot value for `AnalysisIn` sensitivities is 1.0 for the input of interest, 0.0 for all others
- Fill the derivative information into the `derivs` member of the appropriate `AnalysisOut` Value structure in *Post* or *CalcOutput*
- The type of any Output Value Structure that contains derivatives must be set to *DoubleDeriv* (not *Double*)

Note that in *exercises/session14* we will be looking at geometric sensitivities associated with *Bounds*

The simple analysis: `tankCalc`

Computes a fuel tank's mass with **AnalysisIn Sensitivities**

- Input file format – ASCII
 - An integer – 1
 - 4 doubles – `wallDensity`, `wallDensity_dot`, `fuelDensity` and `fuelDensity_dot`
 - 9 floating point numbers – 3 *xyzs* for 3 points for the triangle
There may be as many triangles as necessary, which should represent a closed volume
- Output file format – ASCII
 - 6 floating point numbers on a single line
 - `area`, `area_dot`, `volume`, `volume_dot`, `mass` and `mass_dot` of the fuel tank

The simple analysis: `tankCalc`

Computes a fuel tank's mass with **GeometryIn Sensitivities**

- Input file format – ASCII
 - An integer – 2
 - 4 doubles – `wallDensity`, `wallDensity_dot`, `fuelDensity` and `fuelDensity_dot`
 - 18 floating point numbers representing a single triangle
3 `xyzs` then 3 `xyz_dots` for the 3 points that support the triangle
There may be as many triangles as necessary, which should represent a closed volume
- Output file format – ASCII
 - 6 floating point numbers on a single line
 - `area`, `area_dot`, `volume`, `volume_dot`, `mass` and `mass_dot` of the fuel tank