



Computational Aircraft Prototype Syntheses

AIM Programming

AIM Software Structure

For ESP Rev 1.29

Bob Haimes

bob@geocentrictech.com or haimes@mit.edu

Geocentric Technologies LLC

Execution Driven by Dependency

- All CAPS Objects have a serial number – an unsigned 64 bit integer (`uint64_t`). The *active* SN starts at 1 when the Problem Object is initialized.
- The SN is set to zero when an Object is created indicating it is unfilled and *dirty*.
- When any Object is changed the *active* SN is incremented and set in that Object. This is automatically accomplished by setting/updating a Value.
- When data is requested in CAPS the dependencies for the *owner* of that information are examined. If any have a SN greater than the owner's SN then this part of the dependency tree is *dirty*.
- *Links* are followed and override the any AnalysisIn data. *Links* are followed for any FieldIn data (which can cause a data transfer).
- Each connection is recursively examined and then execution is forward triggered to *clean* the state. This is done by incrementing the *active* SN and setting it in the Object executed.

The **capsValue** Structure is simply the data found within a CAPS Value Object. `aimInputs` and `aimOutputs` must fill the structure with the *type*, *form* and optionally *units* of the data. `aimInputs` also sets the default value(s) in the *vals* member. The structure's members listed below must be filled (most have defaults).

Value Type – no default

The value *type* can be one of:

```
enum capsvType {Boolean, Integer, Double, String, Tuple, Pointer, DoubleDeriv, PointerMesh};
```

Notes:

- 1 The `Pointer/PointerMesh` types are only supported at the AIM level to communicate between AIMS. Linkages should be used from *AnalysisOut* to *AnalysisIn* to make the connection. The `units` member of the Value Structure must match for a successful link.
- 2 `DoubleDeriv` is a `Double` with optional derivatives (`GeometryOut`, `AnalysisOut` & `AnalysisDynO` only)

The tuple structure

```
typedef struct {  
    char *name;                /* the name */  
    char *value;               /* the value for the pair */  
} capsTuple;
```

Shape of the Value – 0 is the default

dim can be one of:

- 0 scalar only
- 1 vector or scalar
- 2 scalar, vector or 2D array

Value Dimensions – 1 is the default

nrow and *ncol* set the dimension of the Value. If both are 1 this has a `scalar` shape. If either *nrow* or *ncol* are one then the shape is `vector`. If both are greater than 1 then this represents a 2D array of values.

Other enumerated constants

```
enum capsFixed      {Change, Fixed};  
enum capsNull       {NotAllowed, NotNull, IsNull, IsPartial};  
enum capstMethod    {Copy, Integrate, Average};
```

Varying Length – the default is “Fixed”

The member *lfixed* indicates whether the length of the Value is allowed to change.

Varying Shape – the default is “Fixed”

The member *sfixed* indicates whether the *shape* (i.e., *dim*) of the Value is allowed to change.

Can Value be NULL? – the default is “NotAllowed”

The member *nullVal* indicates whether the Value is or can be **NULL**
Options are found in `enum capsNULL`

A Note on String Storage

Multiple Strings are not stored as a list of pointers, but as a contiguous block of memory where each individual string is zero terminated.

capsValue Member Usage Notes

The AIM has complete access to the Value structure but should only modify the contents of AnalysisIn, AnalysisOut and AnalysisDynO Values.

- *sfixed & dim*

If the shape is “Fixed” then *nrow* and *ncol* must fit that shape. For example, if the shape is vector then both *nrow* and *ncol* cannot be greater than 1.

- *lfixed & nrow/ncol*

If the length is “Fixed” then all updates of the Value(s) must match in both *nrow* and *ncol* (which presumes a “Fixed” shape).

- *nullVal & nrow/ncol*

nrow and *ncol* should remain at their values even if the Value is **NULL** to maintain the dimension (and possibly length) when “Fixed”. To indicate a **NULL** all that is necessary is to set *nullVal* to “IsNull”. The actual allocated storage can remain in the *vals* member or set to **NULL**.

- Use AIM_ALLOC to allocate any memory required for the *vals* member.

- After the Value has been created *sfixed* and *lfixed* must not be changed!

From \$ESP_ROOT/include/capsTypes.h:

```
/*
 * structure for derivative data w/ CAPS Value structure
 *   only used with "real" (double) data and
 *   only with GeometryOut, AnalysisOut or AnalysisDynO Value Objects
 */

typedef struct {
    char    *name;           /* the derivative with respect to */
    int     len_wrt;         /* the number of members in the derivative
                             w.r.t. Value Object */
    double *deriv;           /* the derivative values
                             - capsValue.length*len_wrt long */
} capsDeriv;

/*
 * structure for CAPS object -- VALUE
 */

typedef struct {
    int     type;            /* value type -- capsvType */
    int     length;          /* number of values */
    int     dim;             /* the dimension */
    int     nrow;            /* number of rows */
    int     ncol;            /* the number of columns */
}
```

```

int          lfixed;          /* length is fixed */
int          sfixed;          /* shape is fixed */
int          nullVal;         /* NULL handling */
int          index;           /* index into collection of Values */
int          pIndex;          /* DESPMTR index */
int          gInType;         /* 0 -- DESPMTR (or not GeomIn), 1 -- CFGPMTR,
                               2 -- CONPMTR */

union {
  int         integer;        /* single int -- length == 1 */
  int         *integers;      /* multiple ints */
  double      real;           /* single double -- length == 1 */
  double      *reals;         /* multiple doubles */
  char        *string;        /* character string (no single char) */
  capsTuple   *tuple;         /* tuple (no single tuple) */
  void        *AIMptr;        /* AIM pointer(s) */
} vals;

union {
  int         ilims[2];       /* integer limits */
  double      dlims[2];       /* double limits */
} limits;

void         *lims;           /* per element limits [2*length*sizeof()] */
char         *units;          /* the units for the values */
char         *meshWriter;     /* the mesh writer (linked AnalysisIn) */
capsObject   *link;           /* the linked object (or NULL) */
int          linkMethod;      /* the link method */
int          *partial;        /* NULL or vector/array element NULL handling */
int          nderiv;          /* the number of derivatives */
capsDeriv    *derivs;         /* the derivatives associated with the Value */
double       *stepSize;       /* Finite Difference step size - DESPMTR only */
} capsValue;

```

Initialization Information for the AIM

```
icode = aimInitialize(int qFlag, const char *uSys, void *aimInfo,
                    void **instStore, int *major, int *minor,
                    int *nIn, int *nOut, int *nFields,
                    char ***fnames, int **franks, int **fInOut)
```

qFlag -1 indicates a query and not a new analysis instance (0 or greater)

uSys a pointer to a character string declaring the unit system – can be **NULL**

aimInfo the AIM context – **NULL** if **qFlag == -1**

instStore a returned pointer to a block of memory to be associated with this AIM instance may be returned as **NULL** if no AIM state data is required

major the returned AIM major version number

minor the returned AIM minor version number

nIn the returned number of Inputs (minimum of 1)

nOut the returned number of possible Outputs

nFields the returned number of fields to responds to for DataSet filling

fnames a returned pointer to a list of character strings with the field/DataSet names †

franks a returned pointer to a list of ranks associated with each field †

fInOut a returned pointer to a list of field flags (FIELDIN - input, FIELDOUT - output) †

icode integer return code – see \$ESP_ROOT/include/capsErrors.h

Return Analysis Inputs

```
icode = aimInputs(void *instStore, void *aimInfo, int index,  
                  char **ainame, capsValue *defval)
```

- instStore** the AIM *instance* storage – **NULL** if called from caps_getInput
- aimInfo** the AIM context – **NULL** if called from caps_getInput
- index** the Input index [1-nIn]
- ainame** a returned pointer to the returned Analysis Input variable name
- defval** a pointer to the filled default value(s) and units – any allocated memory will be freed
- icode** integer return code

Return Analysis Outputs

```
icode = aimOutputs(void *instStore, void *aimInfo, int index,  
                   char **aoname, capsValue *form)
```

- instStore** the AIM *instance* storage – **NULL** if called from caps_getOutput
- aimInfo** the AIM context – **NULL** if called from caps_getOutput
- index** the Output index [1-nOut]
- aoname** a returned pointer to the returned Analysis Output variable name
- form** a pointer to the Value Shape & Units information – to be filled
any actual values stored are ignored/freed
- icode** integer return code

Set or Update the AIM's Internal State

```
icode = aimUpdateState(void *instStore, void *aimInfo,  
                        capsValue *inputs)
```

instStore the AIM *instance* storage

aimInfo the AIM context (used by the Utility Functions)

inputs the complete suite of Analysis inputs (nIn in length)

icode integer return code

Notes: This function is always called first in the execution sequence (before `aimDiscr`, `aimPreAnalysis` or `aimPostAnalysis`). It should not write into the Analysis directory.

Parse Input data & Generate Input File(s)

```
icode = aimPreAnalysis(const void *instStore, void *aimInfo,  
                        capsValue *inputs)
```

instStore the AIM *instance* storage

aimInfo the AIM context (used by the Utility Functions)

inputs the complete suite of Analysis inputs (nIn in length)

icode integer return code

Execute Analysis – Optional

```
icode = aimExecute(const void *instStore, void *aimInfo, int *state)
```

instStore the AIM *instance* storage

aimInfo the AIM context (used by the Utility Functions)

state the returned status (0 - done, 1 - running) – currently unused

icode integer return code

Note: if this function exists it is an indication that the AIM can auto-execute. Otherwise the execution is driven explicitly and errors are raised if data is requested from the *dirty* AIM instance.

Processing after the Analysis is run

```
icode = aimPostAnalysis(void *instStore, void *aimInfo, int restart,  
                        capsValue *inputs)
```

instStore the AIM *instance* storage

aimInfo the AIM context (used by the Utility Functions)

restart restart state (0 - normal, 1 - restart invocation)

inputs the complete suite of Analysis inputs – for restart (nIn in length)

icode integer return code

Note: this function gets called by `caps_postAnalysis`, implicitly during `caps_execute`, during lazy execution (if auto-exec) and while restarting (only for the last invocation of an instance) to populate any internal state information (and should not write into the Analysis directory).

Free up any memory the AIM has stored

```
void aimCleanup(void *instStore)
```

instStore the block of memory associated with a particular instance

Note:

- Called a number of times, once for each instance

Calculate/Retrieve Output Information

```
icode = aimCalcOutput(void *instStore, void *aimInfo, int index,  
                      capsValue *val)
```

instStore the AIM *instance* storage

aimInfo the AIM context (used by the Utility Functions)

index the Output index [1-nOut] for this single result

val a pointer to the capsValue data to fill – CAPS will free any allocated memory

icode integer return code

Note:

- Called in a *lazy* manner, only when the output is needed (and after the Analysis is run)

MACROs defined in `$ESP_ROOT/include/aimUtil.h`:

Function Status MACRO

```
AIM_STATUS(void *aimInfo, int status, ...)
```

aimInfo the AIM context

status return status from a function

... printf type format string and data

Notes:

- 1 Tracks file, line, and function name backtrace information – if **status** < CAPS_SUCCESS
- 2 Includes “goto cleanup” if **status** < CAPS_SUCCESS

Pseudo Code Examples

```
status = myfunc1(aimInfo, arg1, arg2);  
AIM_STATUS(aimInfo, status)
```

```
status = myfunc2(aimInfo, arg1, arg2);  
AIM_STATUS(aimInfo, status, "myfunc2 args %d, %d", arg1, arg2)
```

Function Status MACRO with NOTFOUND exception

```
AIM_NOTFOUND (void *aimInfo, int status, ...)
```

aimInfo the AIM context

status return status from a function

... printf type format string and data

Notes:

- 1 Tracks file, line, and function name backtrace information – if **status** < CAPS_SUCCESS and **status** != CAPS_NOTFOUND and **status** != EGADS_NOTFOUND
- 2 Includes “goto cleanup” if the check fails

Pseudo Code Examples

```
status = myfunc1(aimInfo, arg1, arg2);  
AIM_NOTFOUND(aimInfo, status)
```

```
status = myfunc2(aimInfo, arg1, arg2);  
AIM_NOTFOUND(aimInfo, status, "myfunc2 args %d, %d", arg1, arg2)
```

ANALYSISIN Error Message MACRO

```
AIM_ANALYSISIN_ERROR(void *aimInfo, enum index, const char *format,  
                    ...)
```

aimInfo the AIM context

index index of ANALYSISIN

format printf format string

... printf data

Note: Tracks file, line, and function name backtrace information

Pseudo Code Examples

```
mach = inputs[Mach-1].vals.real;  
if (mach < 0) {  
    AIM_ANALYSISIN_ERROR(aimInfo, Mach,  
                        "Mach = %f must be >= 0\n", mach);  
    status = CAPS_BADVALUE;  
    goto cleanup;  
}
```

Error Message MACRO

```
AIM_ERROR(void *aimInfo, const char *format, ...)
```

aimInfo the AIM context
format printf format string
... printf data

Note: Tracks file, line, and function name backtrace information

Message Add Line MACRO

```
AIM_ADDLINE(void *aimInfo, const char *format, ...)
```

aimInfo the AIM context
format printf format string
... printf data

Pseudo Code Examples

```
status = aim_getBodies(aimInfo, &nBody, &bodies);
AIM_STATUS(aimInfo, status)

If (nBody != 1) {
    AIM_ERROR(aimInfo, "Only one body expected, but nBody = %d", nBody);
    AIM_ADDLINE(aimInfo, "This aim can only work with one body");
    status = CAPS_BADVALUE;
    goto cleanup;
}
```

Warning Message MACRO

```
AIM_WARNING(void *aimInfo, const char *format, ...)
```

aimInfo the AIM context

format printf type format string

... printf data

Notes:

- 1 Tracks file, line, and function name backtrace information
- 2 Use AIM_ADDLINE to add additional lines

Pseudo Code Examples

```
status = aim_getBodies(aimInfo, &nBody, &bodies);  
AIM_STATUS(aimInfo, status)
```

```
If (nBody > 1) {  
    AIM_WARNING(aimInfo, "Only one body will be used, but nBody = %d", nBody);  
    AIM_ADDLINE(aimInfo, "This aim only uses one body");  
}
```

Informational Message MACRO

```
AIM_INFO(void *aimInfo, const char *format, ...)
```

aimInfo the AIM context

format printf type format string

... printf data

Notes:

- 1 Tracks file, line, and function name backtrace information
- 2 Use AIM_ADDLINE to add additional lines

Remove Error Message

```
aim_removeError(void *aimInfo)
```

aimInfo the AIM context

Pseudo Code Example

```
status = myfunc3(aimInfo, arg1, arg2);  
if (status == CAPS_BADVALUE) {  
    aim_removeError(aimInfo);  
    /* Resolve CAPS_BADVALUE error */  
} else {  
    AIM_STATUS(aimInfo, status);  
}
```

Memory Allocation MACROs

AIM_ALLOC(**void** *ptr, **size_t** size, **type**, **void** *aimInfo, **int** status)

AIM_REALL(**void** *ptr, **size_t** size, **type**, **void** *aimInfo, **int** status)

ptr pointer assigned allocation (must be **NULL** for AIM_ALLOC)

size number of **type** allocations

type data type for the allocation

aimInfo the AIM context

status function return status

Notes:

- 1 Tracks file, line, and function name backtrace information
- 2 Includes “goto cleanup” on error and sets **status** = EGADS_MALLOC

Free Memory

AIM_FREE(**void** *ptr)

ptr frees pointer memory and sets **ptr** = **NULL**

String Duplication MACRO

```
AIM_STRDUP(char *ptr, const char *str, void *aimInfo, int status)
```

ptr pointer assigned allocation (must be **NULL**)

str string for duplication

aimInfo the AIM context

status function return status

Notes:

- 1 Tracks file, line, and function name backtrace information
- 2 Includes “goto cleanup” on error and sets **status** = EGADS_MALLOC

Enum Name Creation MACRO

```
char *AIM_NAME(enum Name)
```

Name enumeration

Notes: Converts enumeration Index “Name” to a string and returns a duplicate string

Memory & Error Handling

```
AIM_NOTNULL(char *ptr, void *aimInfo, int status)
```

ptr pointer checked

aimInfo the AIM context

status function return status

Notes: If **ptr** == **NULL**, sets **status** = CAPS_NULLVALUE and then “goto cleanup”

Pseudo Code Example

```
enum aimInputs {  
    Mach = 1,          /* index is 1-based */  
    NUMINPUT = Mach    /* Total number of inputs */  
};  
...  
if (index == Mach) {  
    *ainame = AIM_NAME(Mach);  
    ...  
}  
AIM_NOTNULL(*ainame, aimInfo, status);
```

```
#include <string.h>
#include <math.h>
#include "aimUtil.h"

//#define DEBUG

enum aimInputs
{
    InputVariable = 1,          /* index is 1-based */
    num,
    Mach,
    Mesh_Format,
    Table,
    NUMINPUT = Table           /* Total number of inputs */
};

enum aimOutputs
{
    sqrtNum = 1,               /* index is 1-based */
    NUMOUT = sqrtNum           /* Total number of outputs */
};

typedef struct {
    int nBody;
    ego *tess;
} aimStorage;
```

```
int aimInitialize(int inst, const char *unitSys, void *aimInfo,
                 void **instStore, int *major, int *minor, int *nIn, int *nOut,
                 int *nFields, char ***fnames, int **franks, int **fInOut)
{
    int      status;
    aimStorage *aimStore = NULL;

#ifdef DEBUG
    printf("\n myAIM/aimInitialize  instance = %d  unitSys = %s!\n",
          inst, unitSys);
#endif

    /* specify the rev of this AIM */
    *major = 1;
    *minor = 0;

    /* specify the number of analysis inputs and outputs */
    *nIn  = NUMINPUT;
    *nOut = NUMOUT;

    /* return if "query" only */
    if (inst == -1) return CAPS_SUCCESS;

    /* specify the field variables this analysis can generate and consume */
    *nFields = 0;

    /* setup our AIM specific state */
    AIM_ALLOC(aimStore, 1, aimStorage, aimInfo, status);
    *instStore = aimStore;
```

```
/* populate our instance storage */
aimStore->nBody = 0;
aimStore->tess = NULL;

return CAPS_SUCCESS;

cleanup:
return status;
}

int aimInputs(void *instStore, void *aimInfo, int index,
              char **ainame, capsValue *defval)
{
    int i, status = CAPS_SUCCESS;
    capsTuple *tuple = NULL;

#ifdef DEBUG
    printf(" myAIM/aimInputs instance = %d index = %d!\n",
           aim_getInstance(aimInfo), index);
#endif

    /* fill in the required members based on the index */
    if (index == InputVariable) {
        *ainame = AIM_NAME(InputVariable);
        defval->type = Boolean;
        defval->vals.integer = false;
    } else if (index == num) {
        *ainame = AIM_NAME(num);
        defval->type = Double;
        defval->vals.real = 8.0;
    }
}
```

```
} else if (index == Mach) {
    *ainame          = AIM_NAME(Mach); // Mach number
    defval->type      = Double;
    defval->>nullVal   = IsNull;
    defval->units     = NULL;
    defval->lfixed    = Change;
    defval->dim       = Scalar;

} else if (index == Mesh_Format) {
    *ainame          = AIM_NAME(Mesh_Format);
    defval->type      = String;
    defval->lfixed    = Change;
    AIM_STRDUP(defval->vals.string, "AFLR3", aimInfo, status);

} else if (index == Table) {
    /* an example of filling in a Tuple */
    *ainame = AIM_NAME(Table);

    AIM_ALLOC( tuple, 3, capsTuple, aimInfo, status);
    for (i = 0; i < 3; i++) {
        tuple[i].name = NULL;
        tuple[i].value = NULL;
    }
    AIM_STRDUP( tuple[0].name , "Entry1", aimInfo, status);
    AIM_STRDUP( tuple[1].name , "Entry2", aimInfo, status);
    AIM_STRDUP( tuple[2].name , "Entry3", aimInfo, status);
    AIM_STRDUP( tuple[0].value, "Value1", aimInfo, status);
    AIM_STRDUP( tuple[1].value, "Value2", aimInfo, status);
    AIM_STRDUP( tuple[2].value, "Value3", aimInfo, status);
```

```
    defval->type      = Tuple;
    defval->dim        = Vector;
    defval->nrow       = 1;
    defval->ncol       = 3;
    defval->vals.tuple = tuple;
}

AIM_NOTNULL(*ainame, aimInfo, status);

cleanup:
    return status;
}

int aimUpdateState(void *instStore, void *aimInfo, capsValue *inputs)
{
    int status;

    printf("\n");
#ifdef DEBUG
    printf(" myAIM/aimUpdateState      instance = %d!\n",
           aim_getInstance(aimInfo));
#endif

    status = aim_newGeometry(aimInfo);
    printf(" myAIM/aimUpdateState aim_newGeometry = %d!\n", status);

    return status;
}
```

```
int aimPreAnalysis(const void *instStore, void *aimInfo, capsValue *inputs)
{
    int          i, n, status = CAPS_SUCCESS;
    const char *name;
    double      mach;
    capsValue   *val;

#ifdef DEBUG
    printf("\n myAIM/aimPreAnalysis instance = %d!\n", aim_getInstance(aimInfo));
#endif

    /* look at the CSM design parameters */
    printf("\n   GeometryIn:\n");
    n = aim_getIndex(aimInfo, NULL, GEOMETRYIN);
    for (i = 0; i < n; i++) {
        status = aim_getName(aimInfo, i+1, GEOMETRYIN, &name);
        if (status != CAPS_SUCCESS) continue;
        status = aim_getValue(aimInfo, i+1, GEOMETRYIN, &val);
        if (status == CAPS_SUCCESS)
            printf("      %d: %s %d (%d,%d)\n", i+1, name,
                val->type, val->nrow, val->ncol);
    }

    /* look at the CSM output parameters */
    printf("\n   GeometryOut:\n");
    n = aim_getIndex(aimInfo, NULL, GEOMETRYOUT);
    for (i = 0; i < n; i++) {
        status = aim_getName(aimInfo, i+1, GEOMETRYOUT, &name);
        if (status != CAPS_SUCCESS) continue;
        status = aim_getValue(aimInfo, i+1, GEOMETRYOUT, &val);
        if (status == CAPS_SUCCESS)
            printf("      %d: %s %d (%d,%d)\n", i+1, name,
                val->type, val->nrow, val->ncol);
    }
}
```

```
/* write out input list of values */
if (inputs != NULL) {
    printf("\n    AnalysisIn:\n");
    for (i = 0; i < NUMINPUT; i++) {
        status = aim_getName(aimInfo, i+1, ANALYSISIN, &name);
        AIM_STATUS(aimInfo, status);
        printf("        %d: %s %d (%d,%d) %s\n", i+1, name,
            inputs[i].type, inputs[i].nrow, inputs[i].ncol, inputs[i].units);
    }
    mach = inputs[Mach-1].vals.real;
    if (mach < 0) {
        AIM_ANALYSISIN_ERROR(aimInfo, Mach, "Mach number must be >= 0\n");
        AIM_ADDLINE(aimInfo, "Negative mach = %f is non-physical\n", mach);
        status = CAPS_BADVALUE;
        goto cleanup;
    }
}
printf("\n");

cleanup:
    return status;
}

int aimExecute(const void *instStor, void *aimInfo, int *state)
{
#ifdef DEBUG
    printf(" myAIM/aimExecute instance = %d!\n", aim_getInstance(aimInfo));
#endif

    *state = 0;
    return CAPS_SUCCESS;
}
```

```
int aimPostAnalysis(void *instStore, void *aimInfo, int restart, capsValue *inputs)
{
#ifdef DEBUG
    printf(" myAIM/aimPostAnalysis instance = %d!\n", aim_getInstance(aimInfo));
#endif

    return CAPS_SUCCESS;
}

int aimOutputs(void *instStore, void *aimInfo, int index, char **aoname, capsValue *form)
{
    int status = CAPS_SUCCESS;
#ifdef DEBUG
    printf(" myAIM/aimOutputs instance = %d index = %d!\n",
        aim_getInstance(aimInfo), index);
#endif

    if (index == sqrtNum) {
        *aoname = AIM_NAME(sqrtNum);
        form->type = Double;
    } else {
        printf(" myAIM/aimOutputs: Unknown index = %d!\n", index);
        status = CAPS_BADINDEX;
    }

    return status;
}
```

```
int aimCalcOutput(void *instStore, void *aimInfo, int index, capsValue *val)
{
    int          status = CAPS_SUCCESS;
    double       myNum;
    capsValue    *myVal;

#ifdef DEBUG
    const char *name;

    status = aim_getName(aimInfo, index, ANALYSISOUT, &name);
    printf(" myAIM/aimCalcOutput instance = %d  index = %d %s %d!\n",
           aim_getInstance(aimInfo), index, name, status);
#endif

    if (index == sqrtNum) {
        /* default return */
        val->vals.real = -1;

        /* get the input num */
        status = aim_getValue(aimInfo, num, ANALYSISIN, &myVal);
        if (status != EGADS_SUCCESS) {
            printf("ERROR:: aim_getValue -> status=%d\n", status);
            goto cleanup;
        }

        if ((myVal->type != Double) || (myVal->length != 1)) {
            printf("ERROR:: aim_getValue -> type=%d, len = %d\n",
                   myVal->type, myVal->length);
            status = CAPS_BADTYPE;
            goto cleanup;
        }
    }
}
```

```
    myNum = myVal->vals.real;
    val->vals.real = sqrt(myNum);

} else {

    status = CAPS_BADVALUE;

}

cleanup:
    return status;
}

void aimCleanup(void *instStore)
{
    aimStorage *aimStore = NULL;
#ifdef DEBUG
    printf(" myAIM/aimCleanup!\n");
#endif

    aimStore = (aimStorage *) instStore;
    if (aimStore == NULL) return;
    AIM_FREE(aimStore->tess);
    AIM_FREE(aimStore);
}
```

```
import pyCAPS
from pyOCSM import esp

# Instantiate our CAPS problem "myProblem"
print("Loading file into our Problem")
myProblem = pyCAPS.Problem(problemName="myExample", capsFile="case.csm")

# Load our session aim
myAIM = myProblem.analysis.create(aim = "myAIM")

# Get current value of our first input
value = myAIM.input.num
print("Default num =", value)
myAIM.input.num = 16.0
value = myAIM.input.num
print("Current num =", value)

# AIM autoExecutes

# Get outputs
myGeometry = myProblem.geometry
value = myGeometry.outpmtr["pVol"].value
print(" ")
print("parentVolume =", value)
assert(abs(value - 34.6415248071731) < 1.e-7)
value = myGeometry.outpmtr["cVol"].value
print("childVolume =", value)
assert(abs(value - 7.296559648508515) < 1.e-7)

value = myAIM.output.sqrtNum
print("sqrtNum =", value)
assert(value == 4.0)
print(" ")
```

- Build the AIM and run it from the *examples* directory
- Note the *test* directory and execute the tests via `make test`
- Edit `myAIM.c` and set the *define* `DEBUG`
- Run it again and note the execution sequence
- In the Python script ask for *sqrNum* before the Volumes
Does this new execution sequence make sense?
- Change the value of *Mach* at the end of `session08.py` and ask for *sqrNum*. Does the geometry get rebuilt?
- Add your own Analysis input(s) and set a default*
- Play around with additional Analysis output(s)*

* Note: When building up an AIM it is a good idea to start with the bare minimum of inputs and outputs. This way you can get things up and running and later incrementally add additional functionality.