



Computational Aircraft Prototype Syntheses

AIM Programming – Overview

For ESP Rev 1.28

Bob Haines

bob@geocentrictech.com or haines@mit.edu

Geocentric Technologies LLC

Geocentric Technologies, LLC will be taking over the stewardship of EPS (and CAPS) from **MIT**

This is a reflection of software transitioning from an academic *research* project to *mature commercial* products with support.

Note that the software will remain under an *Open Source* license and hence continue to be freely available.

This entails (post ESP Rev 1.28) the following:

- Change of hosting website to: <http://geocentrictech.com/ESP>
- Opening up a portal so that external sources can contribute plugins
- Moving most of the current AIMs to that portal, this will strongly effect how CAPS is distributed

This transition is initially funded by an Air Force SBIR

Requirements for contributed AIMs & Mesh Writers:

- A directory with the top-level name of the contribution, for example: `nameAIM` or `nameWriter`
- The source, Makefile and NMakefile (as will be seen in the upcoming sessions)
- A Windows *def* file
- An optional *example* subdirectory – contains illuminating plugin executions that are driven by CSM and Python scripts, as well as supporting files for the execution
- A *test* subdirectory
Unit and regression tests to support continuous integration and acceptance
Executed by issuing “make test” (or “nmake -f NMakefile test”) at the top-level
- A *docs* subdirectory – the contents of which is still under discussion

There will be methods for users to grab plugins from the portal, but the details of this have not been finalized.

The CAPS API and AIMs

- This is MUCH harder than most software testing because the full execution depends on third-party software, which may change and is out of our control
- How do you test the results of surface and volume mesh generation as well as (and in conjunction with) solver execution?
 - Mesh counts depend on the geometry and floating point predicates!
 - Solver integrated quantities can hide issues!
 - Is this the responsibility of CAPS?
- AIMs will now require testing and will not be accepted without
- This should be done in Python in one of two ways:
 - 1 Use Python's testing framework (see `exercises/session06`)
 - 2 By Python script(s) that contain *asserts* (see `exercises/session07`)

Note: both sessions use the same AIM.

CAPS Execution

The Design workflow may require a long running and complex process involving a mix of multi-fidelity and multidisciplinary analyses:

- May need to manage the complexity
- Get to some point and ask *what if* questions
- Store away (checkpoint) the state of the Design at various times

CAPS Design execution breaks the process into *phases*

- One *phase* is a “stepping stone” to the next
- Allows for “branching” from a completed *phase* to multiple new *phases* (like repositories — but with no merging)
- Each *phase* is typically driven by a different CAPS app or pyCAPS script
- Can robustly get from *phase* to *phase*

CAPS *phases*

In most cases:

- The first *phase* builds the objects
- The subsequent *phases* discover or use the existing objects

At all times the current object state is mirrored on disk:

- Uses directory structure to mimic the object hierarchy
- Writes objects when updated in binary (which are usually small)

In addition, CAPS API function's output are *journalled* to disk

End result:

- Execution can be paused, interrupted or encounter an error and continue later
- AIM post-Analysis is re-executed during continuation when last invocation is reached to reestablish any internal storage

CAPS Execution Modes

CAPS has 4 modes for starting a session:

- Scratch – This is for development (and not production). It will remove any existing data in the *Scratch* directory of the Problem's path
- Initial – This *phase* is started by a call to `caps_open` that points to a nonexistent *phase* subdirectory. The initialization can either be from a geometry file or an OpenCSM or EGADS Model.
- Continuation* – This occurs when CAPS has not finished a *phase* either do to an interruption or not reaching `caps_close` (that can mark the *phase* complete). In this case the CAPS application or pyCAPS script can be run from the beginning, but reading results from the *journal* is used to quickly get to the position where the *phase* terminated.
- Starting from a completed *phase*

* works best when most of the computation is controlled by CAPS
must be same ESP rev & architecture (due to the chaotic nature of meshing)

CAPS Directory Structure

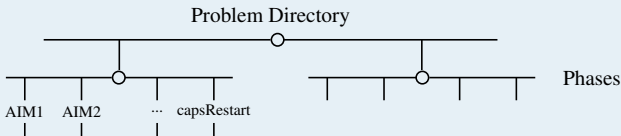
At the top specified directory level you will find *phase* subdirectories

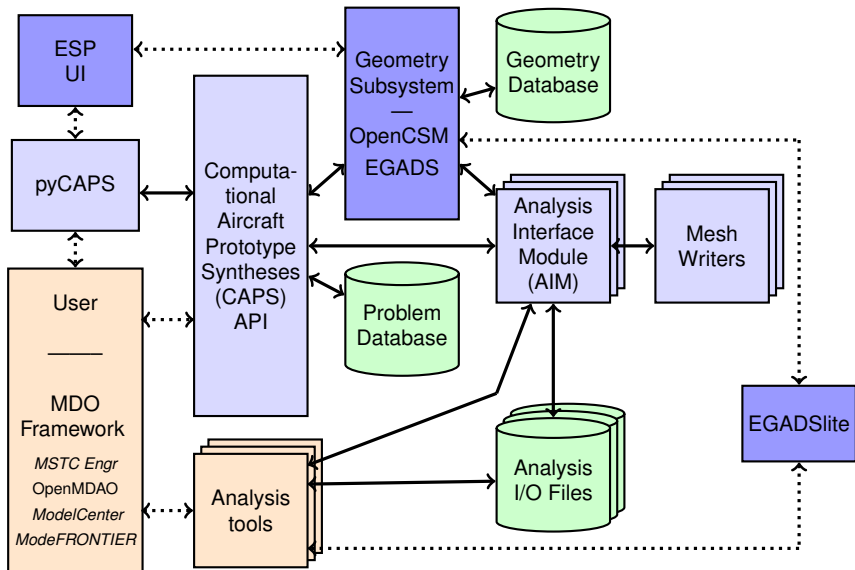
In each *phase* subdirectory you may see:

- capsRestart.cpc – a CSM saved state file – or – capsRestart.egads – an EGADS file (for nonparametric runs)
- capsRestart – subdirectory that contains the CAPS restart data
- capsClosed – the *phase* has been closed (caps_close has been called to mark completion)
- capsLock – an indication that another application is executing in this subdirectory
- AIMnames – subdirectories each related to an AIM instance in the running CAPS Problem/Phase

Notes:

- *Scratch* phase (no name specified) is not as protected as the others
- CAPS Problem directory or individual *phase* subdirectories can be copied and used elsewhere





Object-based Not *Object Orientated*

- Like **egos** in EGADS
- Pointer to a C structure – allows for a function-based API
- Treated as *blind pointers* (i.e., not meant to be dereferenced)
Header info is used to determine how to dereference the *pointer*
- API C Functions
 - Returns an **int** error code or CAPS_SUCCESS
 - Usually have one (or more) input Objects
 - Can have an output Object (usually at the end of the argument list)
- Can interface with multiple compiled languages
pyCAPS sits on-top of the C API

See \$ESP_ROOT/doc/CAPSapi.pdf

```
/*
 * defines the owning information
 */
typedef struct {
    int      index;           /* intent phrase index -- -1 no intent */
    char     *pname;          /* the process name -- NULL from Problem */
    char     *pID;            /* the process ID -- NULL from Problem */
    char     *user;           /* the user name -- NULL from Problem */
    short    datetime[6];     /* the date/time stamp */
    CAPSLONG sNum;            /* the CAPS sequence number */
} capsOwn;
```

```
/*
 * defines the CAPS object
 */
typedef struct capsObject {
    int      magicnumber;           /* must be set to validate the object */
    int      type;                 /* object type */
    int      subtype;              /* object subtype */
    int      delMark;              /* delete mark */
    char     *name;                /* object name */
    egAttrs  *attrs;               /* object attributes */
    void     *blind;               /* blind pointer to object data */
    void     *flist;               /* freeable list */
    int      nHistory;             /* number of history entries */
    capsOwn  *history;             /* the object's history */
    capsOwn  last;                 /* last to modify the object */
    struct  capsObject *parent;
} capsObject;

typedef struct capsObject* capsObj;
```

Note: this is only for reference. AIM programming is outside of the CAPS Object handling. `blind` is a pointer to a CAPS data structure where AIM programming does have access.

Problem Object

The Problem is the top-level *container* for a single mission. It maintains a single set of interrelated geometric models, analyses to be executed, connectivity and data associated with the run(s), which can be both multi-fidelity and multidisciplinary. There can be multiple Problems in a single execution of CAPS and each Problem is designed to be *thread safe* allowing for multi-threading of CAPS at the highest level.

Value Object

A Value Object is the fundamental *data* container that is used within CAPS. It can represent *inputs* to the Analysis and Geometry subsystems and *outputs* from both. Also Value Objects can refer to *mission* parameters that are stored at the top-level of the CAPS database. The values contained in any *input* Value Object can be *linked* to another Value (or *DataSet*) Object of the same *shape*. Attributes are also cast to temporary (*User*) Value Objects.

Analysis Object

The Analysis Object refers to an instance of running an analysis code. It holds the *input* and *output* Value Objects for the instance and a directory path in which to execute the code (though no explicit execution is initiated). Multiple various analyses can be utilized and multiple instances of the same analysis can be handled under the same Problem.

Bound Object

A Bound is a logical grouping of BRep Objects that all represent the same entity in an engineering sense (such as the “outer surface of the wing”). A Bound may include BRep entities from multiple Bodies; this enables the passing of information from one Body (for example, the aero OML) to another (the structures Body).

Dimensionally:

- 1D – Collection of Edges
- 2D – Collection of Faces

VertexSet Object

A VertexSet is a *connected* or *unconnected* group of locations at which discrete information is defined. Each *connected* VertexSet is associated with one Bound and a single *Analysis*. A VertexSet can contain more than one DataSet. A *connected* VertexSet can refer to 2 differing sets of locations. This occurs when the solver stores it's data at different locations than the vertices that define the discrete geometry (i.e. cell centered or non-isoparametric FEM discretizations). In these cases the solution data is provided in a different manner than the geometric.

DataSet Object

A DataSet is a set of engineering data associated with a VertexSet. The rank of a DataSet is the (user/pre)-defined number of dependent values associated with each vertex; for example, scalar data (such as *pressure*) will have rank of one and vector data (such as *displacement*) will have a rank of three. Values in the DataSet can either be deposited there by an application or can be computed (via evaluations, data transfers or sensitivity calculations).

Object	SubTypes	Parent Object
capsProblem	Parametric, Static	
capsValue	GeometryIn, GeometryOut, Parameter, User	capsProblem, capsValue
capsAnalysis		capsProblem
capsValue	AnalysisIn, AnalysisOut, AnalysisDynO	capsAnalysis
capsBound		capsProblem
capsVertexSet	Connected, Unconnected	capsBound
capsDataSet	FieldOut, FieldIn, User, GeomSens, TessSens, Builtin	capsVertexSet

Body Objects are EGADS Objects (egos)

See `$ESP_ROOT/include/capsTypes.h` for the complete *defines*

Filtering the active CSM Bodies occurs at two different stages: once in the CAPS framework, and once in the AIMs. The filtering in the CAPS framework creates sub-groups of Bodies from the CSM stack that are passed to the specified AIM. Each AIM instance is then responsible for selecting the appropriate Bodies from the list it has received.

The filtering is performed by using two Body attributes: “capsAIM” and “capsIntent”.

CSM AIM targeting: “capsAIM”

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. For example, a body designed for a CFD calculation could have:

```
ATTRIBUTE capsAIM $su2AIM;fun3dAIM;cart3dAIM
```

CAPS AIM Instantiation: “capsIntent”

The “capsIntent” Body attribute is used when an AIM uses different Body types for different analyses. The attribute “capsIntent” is a semicolon-separated list of keywords. An argument to `caps_makeAnalysis` accepts a semicolon-separated list of keywords when an AIM is instantiated in CAPS/pyCAPS. Bodies from the “capsAIM” selection with a matching string attribute “capsIntent” are passed to the AIM instance. If the string to `caps_makeAnalysis` is **NULL**, all Bodies with a “capsAIM” attribute that matches the AIM name are given to the AIM instance.

- Hides all of the individual Analysis details (and peculiarities)
 - Individual plugin functions *translate* from the Analysis' perspective back and forth to CAPS
 - Provides a direct connection to BRep geometry and attribution through EGADS
- Outside the CAPS Object infrastructure
 - Use of C structures
 - AIM Utility library (with the *context* embedded in `aimInfo`)

Notes due to changes at Rev 1.19:

- 1 When an AIM function is invoked it is in the correct directory (as part of the Problem file structure).
- 2 There is no longer an explicit AIM parent/child relationship. This is accomplished via *linking* AnalysisOut Values of the *parent* to AnalysisIn Values of the *child*.
- 3 During restart only “Post” is executed at the last use of the AIM instance.
- 4 AIM specific storage is no longer indexed by the instance and is held internally.

Large Mesh IO

The meshing AIMs used to hold onto the grid information in memory and pass a pointer on to the CFD AIMs that write out a mesh file during solver preAnalysis.

In order to have the meshing AIM know what kind of file the downstream solver AIM requires, there needs to be information “passed” from the solver AIM to the meshing AIM. This is accomplished through the linkage itself. An AIM utility function has been added that returns the info for linked (solver) AIMs to aid in knowing what files to write in aimPostAnalysis. After the files have been written from PostAnalysis, the mesh memory needs to be freed up.

The mesh writer is specified by using the Value structure member: `meshWriter`. This is filled in the link target (solver) AnalysisIn Value Object, which will allow for the upstream (meshing) AIM to have knowledge about how the mesh is to be written. The string contains the name of the so/DLL to be loaded for the writing. The meshing AIM accesses this information via the AIM utility function `aim_writeMeshes`. See Session13.

Analysis Dynamic Value Objects

There are circumstances where you may not know *a priori* all of the AnalysisOut values of interest. In this case an AIM can generate outputs that have not been registered at initialization – AnalysisDynO Values.

- These are transient Objects and cannot be used in *linking*
- After successful AIM preAnalysis invocation, all existing Analysis Dynamic Output Objects that are stored in the instance are deleted (and those associated mirrored restart files)
- AIM postAnalysis is the only place where Analysis Dynamic Output Objects can be created (see `aim_makeDynamicOutput`)
- They should not be created (i.e., they will already exist) if the **restart** flag is set
- After successful postAnalysis (and not at restart), the created Dynamic Output Objects are given the serial number of postAnalysis (and are written for restart)
- If postAnalysis errors, any created Dynamic Output Objects are deleted

- An AIM plugin is required for each Analysis code at:
 - a specific *intent*
 - a specific *mode* (i.e., where the inputs may be different)
- AIMs can “talk” to each other
 - AIM outputs of one AIM instance can be linked to inputs of another AIM instance
 - Communication can be accomplished via pointers
- Dynamically loaded at runtime – extendibility and extensibility
 - Windows** Dynamically Loaded Libraries (name .dll)
 - LINUX** Shared Objects (name .so)
 - MAC** *Bundles*, CAPS uses the so file extension
- Plugin names must be unique – loaded by the name
- † indicates memory handled by CAPS in the AIM descriptions
i.e., CAPS will free these memory blocks when necessary

Registration (Session07):

- aimInitialize
- aimInputs
- aimOutputs
- aimCleanup

Analysis Execution (Session07):

- aimUpdateState
- aimPreAnalysis
- aimExecute – optional
- aimPostAnalysis
- aimCalcOutput

Data Transfers – all optional (Session12):

- aimDiscr
- aimFreeDiscrPtr
- aimLocateElement
- aimTransfer
- aimInterpolation & aimInterpolateBar
- aimIntegration & aimIntegrateBar

Analysis Execution Calling Sequences

AIM function execution happens automatically and are driven by what is requested and the *clean/dirty* state of the associated CAPS Objects

- `aimUpdateState` is always called before `aimDiscr`, `aimPreAnalysis` or `aimPostAnalysis`
- `aimDiscr` may be called before or after `aimPreAnalysis` or `aimPostAnalysis`
- `aimPreAnalysis` is always called before `aimExecute` or `aimPostAnalysis` (unless doing a restart/continuation)
- `aimPostAnalysis` is called right after `aimUpdateState` when CAPS is restarting (the *restart* argument is set), or if `aimPostAnalysis` is the first live function with continuation. See note on next page.

Only `aimPreAnalysis` and/or `aimPostAnalysis` (not at restart) should write to the Analysis directory.

A Note on Continuation and Journalling

- The internal state for each AIM instance must be able to be recovered by the AIM
- This means that the structure pointed to by `instStore` (described in the next session) needs to be refilled during the invocation of `aimPostAnalysis` with the **restart** flag
- This may require the AIM to write this data every time it is modified. This file should reside in the AIM's *path* and given a unique filename
- On restart this file should be read either in `aimUpdateState` or `aimPostAnalysis`

```
#
ifndef ESP_ROOT
$(error ESP_ROOT must be set -- Please fix the environment...)
endif
ifndef ESP_ARCH
$(error ESP_ARCH must be set -- Please fix the environment...)
endif
#
IDIR  = $(ESP_ROOT)/include
include $(IDIR)/$(ESP_ARCH)
LDIR  = $(ESP_ROOT)/lib

$(LDIR)/myAIM.so: myAIM.o $(LDIR)/libaimUtil.a
    $(CC) $(SOFLGS) -o $(LDIR)/myAIM.so myAIM.o \
        -L$(LDIR) -laimUtil -locsm -legads -ludunits2 -ldl -lm

myAIM.o: myAIM.c $(IDIR)/aimUtil.h $(IDIR)/capsTypes.h
    $(CC) -c $(COPTS) $(DEFINE) -I$(IDIR) myAIM.c

run: $(LDIR)/myAIM.so
    python session01.py

clean:
    -rm myAIM.o

cleanall: clean
    -rm $(LDIR)/myAIM.so

% make -or-
% make run
```

```
#
!IFDEF ESP_ROOT
!ERROR ESP_ROOT must be set -- Please fix the environment...
!ENDIF
#
IDIR = $(ESP_ROOT)\include
!include $(IDIR)\$(ESP_ARCH).$(MSVC)
LDIR = $(ESP_ROOT)\lib

$(LDIR)\myAIM.dll: myAIM.def myAIM.obj
    -del $(LDIR)\myAIM.dll $(LDIR)\myAIM.lib $(LDIR)\myAIM.exp
    link /out:$(LDIR)\myAIM.dll /dll /def:myAIM.def myAIM.obj \
        /LIBPATH:$(LDIR) aimUtil.lib ocsmlib egads.lib udunits2.lib
    $(MCOMP) /manifest $(LDIR)\myAIM.dll.manifest \
        /outputresource:$(LDIR)\myAIM.dll;2

myAIM.obj: myAIM.c $(IDIR)\aimUtil.h $(IDIR)\capsTypes.h
    cl /c $(COPTS) $(DEFINE) /I$(IDIR) myAIM.c

run: $(LDIR)\myAIM.dll
    python session01.py

clean:
    -del myAIM.obj

cleanall: clean
    -del $(LDIR)\myAIM.dll $(LDIR)\myAIM.lib $(LDIR)\myAIM.exp

> nmake -f NMakefile -or-
> nmake -f NMakefile run
```

- We will be incrementally building up functionality in two simple AIMs: `myAIM.c` & `theAIM.c`
- Most all of the AIMs in the ESP distribution can be used as examples/templates, but note:
 - Many of these have a dependency on a library of utility functions that has not been documented
 - Even if documented, the number of functions is overwhelming and would be difficult to include here
 - Also the desire is now to have isolated AIMs that can stand-alone and have few dependancies
- Nearly all sessions have one or more exercises that (hopefully) illuminate the additions to the AIM from the last session, and do some development
- It would be great if you are considering building an AIM and can find some of the *exercise* time to work on that!

The goal is to provide a template for AIM development, but this AIM is/will be self-contained and does not actually execute anything.

- `aimPreAnalysis` examines the inputs available but does not open and write an analysis input file.
- `aimExecute` does nothing (except indicate that the non-existent analysis has run).
- `aimPostAnalysis` also does not open and read an analysis output file, parse it, and fill in the `AnalysisOut` and `AnalysisDynO` outputs.
- Python scripts, which differ in each session, attempt to display the results of the added functionality.
- Because the same AIM name is used for most of these sessions, it is a good idea to end/start the session with `make cleanall` (or `nmake -f nmakefile cleanall`)

This AIM is also a template but actually runs a simple analysis.

- `aimPreAnalysis` examines the inputs available and writes an analysis input file.
- `aimExecute` sets up a command-line that gets passed to a shell to execute the analysis.
- `aimPostAnalysis` reads the analysis output file and stores away the results to make them available through calls to `aimCalcOutput`.
- Because the same AIM name is used for most of these sessions, it is a good idea to end/start the session with `make cleanall` (or `nmake -f nmakefile cleanall`)

In *exercises/session06*:

- Load `case.csm` into *serveESP*
 - Examine the geometry we will be using in these sessions
 - Note the attributes placed on Faces
- Examine the Python test script `session06.py`
- Build the AIM (in this case `myAIM`)
- Execute the script
 - `make run` – or –
 - `python session06.py` – or –
 - `serveESP session06.py`
- Note the *test* directory and execute the tests via `make test`
- For C++ programmers:
 - Examine the difference between `myAIM.c` and `myAIM.cpp`
 - Build with: `make -f cpp.make` –or– `nmake -f cpp.mak`
 - Use variants of these Makefiles for your exercises