



A Programmer's Training for ESP UDPs, UDFs and CAPS' AIMs The Engineering Sketch Pad – Rev 1.29

Bob Haimes

bob@geocentrictech.com or haimes@mit.edu

and

John F. Dannenhoffer, III

john@geocentrictech.com

Geocentric Technologies LLC

- You are a **C/C++** programmer
- You are here with a laptop containing:
 - **C & C++** (and optionally FORTRAN) compilers
 - **doxygen** (optional – for AIM documentation)
 - ESP Rev 1.29 (or a recent 1.30 Beta)
 - OpenCASCADE 7.8.1
from `HTTP://acdl.mit.edu/ESP` or a PreBuilt distribution
 - A functioning Python at 3.12.10 or higher
If from `HTTP://acdl.mit.edu/ESP` – Run the install script and make this the default Python in the shell/command-prompt used for ESP development
- ESP successfully built (and run) from source
- A desire to write an ESP UDF/UFD and/or an AIM and hopefully an idea for one that you would like to work on.

If any item above is not true – you do not belong here!

Ideally a software system should:

- Work for user, not the user working for the system
- Be based on a mental model that is *easy* for the user to grasp
- Never lose anything they have done (backward compatibility)
- Solve the user's **real** problem, not their stated request
- Be responsive to the changing needs of the users
- Never surprise the user
- Not be reverse-engineered
- Be tested thoroughly

This means that:

- The programmer *takes a hit* so that the user's life is easier

We do not always succeed!

ESP is:

- a parametric geometry creation and manipulation system designed to **fully support** the analysis and design of aerospace vehicles (**aCAD+**)
- a stand-alone system for the development of geometric models
- an API can be embedded into other software systems to support their geometric and process needs (e.g., CAPS)
- extensible so that users can add their own geometric *features*

ESP is not:

- a full-featured mechanical computer-aided design (**mCAD**) system
- a system to be used for creating “drawings”

CAPS is:

- a software system that allows for building complex workflows
- a system that supports multidisciplinary and multi-fidelity analyses with direct access to geometry and its attribution
- a system that simplifies inter-analysis communication
- an API that provides access to sensitivities (supports gradient-based optimization)
- software designed for aircraft design settings
- extensible so that additional analyses can be supported

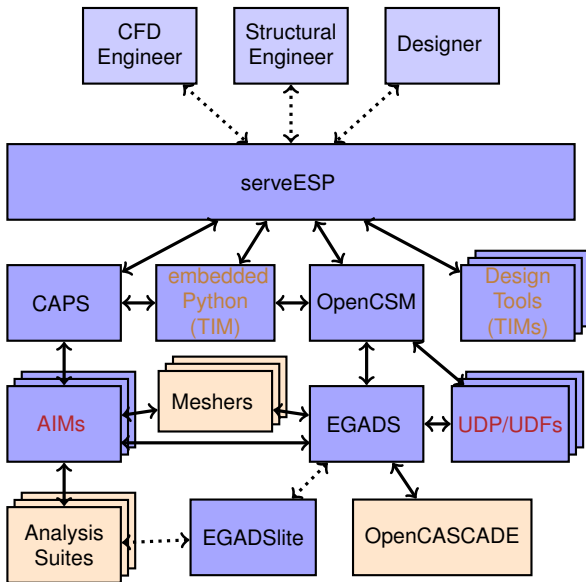
CAPS is not:

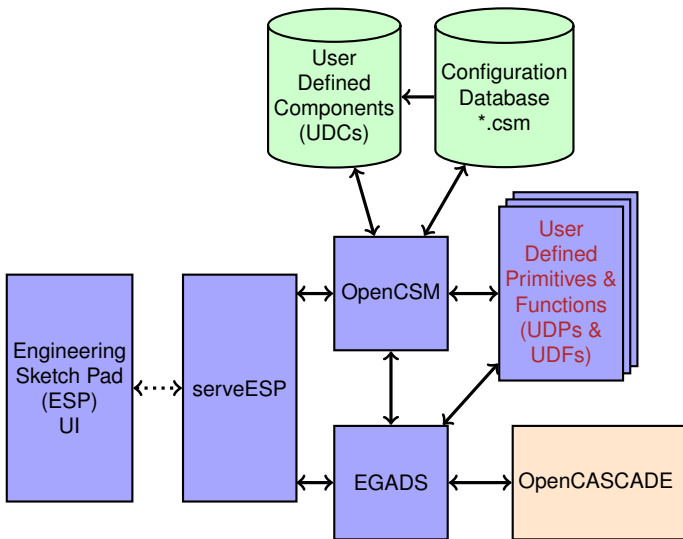
- an MDO Framework (but can support them)

Programmers can add functionality to ESP in the following ways:

- User-Defined Primitives (UDPs)
 - Create and returns an EGADS Body (usually a *Solid*) to be placed on the CSM stack
- User-Defined Functions (UDFs)
 - Has access to other Bodies on the CSM stack
 - Usually creates a new EGADS Body through some form of manipulation → result added back to the stack
- Analysis Interface Modules (AIMs)
 - Interface between the CAPS infrastructure and any Analysis & Meshing codes
 - Provides the ability to generate Analysis/Meshing inputs, perform execution and then access the results
 - AIM *Mesh Writers* are also handled in a similar manner

All are dynamically loaded at run-time





Tool Interface Modules are plugins to **serveESP**

- 2 Types:
 - Script driven
 - Interactive – needs JavaScript additions to the browser software
- Some TIMs:
 - MiniOpt – allows running of some NLOPT optimizers
 - Plotter – for line plots
 - PLUGS – Parametric Legacy Unstructured Geometry System
 - Pyscript – allows for the execution of Python in the server
 - SLUGS – Static Legacy Unstructured Geometry System
 - VspSetup – Interface to OpenVSP

TIMs will not be covered in this training:

- Requires an understanding of **serveESP**'s multi-threading
- JavaScript programming
- no automated way to merge the `js` file in the browser software



Build from source

Use the same shell/environment as was used for building ESP

Build from within a PreBuilt distribution

Make sure you:

- Have a C/C++ compiler and (*N*)*Make* available in your shell/command-prompt
- Set the environment →
 - MAC/Linux: Double-click on the ESP 1.29 desktop icon
Use the opened window for your development
 - Windows: Startup a command-prompt that supports Visual Studio

```
> cd %ESP_ROOT%      (the location of EngSketchPad)
```

```
> ESPenv.bat
```

During ESP Continuous Integration the following tasks are automatically performed

Static Analyzers

- `lint` for **C** code
- `scan-build` for **C** and **C++**
part of `llvm` (of which `clang` is a member)

Dynamic Analysis – run time

- `valgrind` – **LINUX** only
does not play well with default builds of Python
- `clang` or `gcc` with command-line option `-fsanitize=address`
must also include other libraries at link time

Doxygen

Must be available at the command-line for AIM documentation

- Linux: install from the Linux distribution
- Windows:
 - Download from www.doxygen.nl/downloads
 - Install
 - Add to the install point to PATH – can be done in a .bat file:

```
set PATH="C:\Program Files\doxygen\bin";%PATH%
```
- MAC:
 - Install from *homebrew* or www.doxygen.nl/downloads
 - Make sure the executable location is in your path

Built like a shared library

- Windows – a dynamically loaded library (.DLL)
- LINUX – shared object (.so)
- MAC – a *bundle* – we use the extension .so

Loaded at run-time

- Opened with `LoadLibrary / dlopen`
- Function pointers by name via `GetProcAddress / dlsym`
- Can have multiple *instances*, each may require its own *state data*
- Called by function pointer and *instance* via a dispatch table (i.e., name is only used to retrieve the pointer)
- Shared object/DLL not closed so that *valgrind* can report symbols

EGADS

- **API Reference:** `ESP_ROOT/doc/EGADS/egads.pdf`
- **Tutorial:** `ESP_ROOT/doc/EGADS/Tutorial/Tutorial.pdf`
with exercises/examples

OpenCSM

- **API Reference:** `ESP_ROOT/include/OpenCSM.h`
- **UDP/UDF:** Sessions 04 – 06 here

CAPS

- **CAPS API Reference:** `ESP_ROOT/doc/CAPS/CAPSapi.pdf`
- **AIM Development:** `ESP_ROOT/doc/CAPS/AIMdevel.pdf`

Note: Most of the relevant functions are described before there is an expectation of their use (i.e., these sessions can be considered *stand alone*).

Arguments in function signatures:

- When NOT described as *returned* (i.e., an input argument)
 - Usually a vector of the appropriate type or a C string
- When described as *filled*
 - A vector of the appropriate type and a length as long as described
- When described as *returned*
 - The calling routine places a reference to the returned data as the argument. This is usually accomplished by prepending an “&” to the appropriate variable with the matching type
- When described as *freeable*
 - Same as *returned* but the pointer has been allocated by the function and it is the responsibility of the calling routine to free the memory by the use of `EG_free` when appropriate
 - The statement: *the argument can be NULL so the Objects are not filled* means that having NULL in place of “&var” will not fill and return the pointer and you will have nothing that needs freeing

Queries the Objects in a Body

```
icode = EG_getBodyTopos(const ego body, ego ref, int oclass, int *ntopo, ego **topos);
```

- body** the Body Object
- ref** reference Topology Object or **NULL**. Sets the context for the returned Objects (i.e., all objects of the class **oclass** in the tree looking towards that class from **ref**) **NULL** starts from the **body** (for example all Nodes in the Body)
- oclass** is NODE, EDGE, LOOP, FACE or SHELL – must not be the same class as **ref** for EBODY can be EEDGE, ELOOPX, EFACE, ESHELL or the above
- ntopo** the returned number of Topology Objects
- topos** is a returned pointer to the vector of Objects, it is possible that an individual Object may be **NULL** (*freeable*)
Note: the argument can be **NULL** so the Objects are not filled
- icode** the integer return code

This allows for the traversal of the Topology *tree* by jumping levels and/or looking up the hierarchy.

Examples – C code

```
int stat, nface;
ego body, *faces;
```

```
body = ...
:
:
stat = EG_getBodyTopos(body, NULL, FACE,
                       &nface, &faces);
:
EG_free(faces);
```

Get all the Faces in a Body

```
int stat, nface;
ego body, edge;
```

```
body = ...
edge = ...
:
:
stat = EG_getBodyTopos(body, edge, FACE,
                       &nface, NULL);
:
:
```

The number of Faces touching an Edge in the Body

- Opportunity to provide anonymous (or named) immediate feedback for anything “not clear” (e.g. muddy)
- Ask questions about presentation material, previous exercises, point out errors, make suggestions, ...
- Questions will be answered at next session
- E-mail questions (post-training) to any member of the team

- 01 Plugin Overview (this session)
- 02 EGADS – Programmatic use of Geometry in the Plugins – 1/2
- 03 EGADS – Programmatic use of Geometry in the Plugins – 2/2
- 04 OpenCSM UDP Basics
- 05 OpenCSM Full-featured UDP
- 06 OpenCSM Full-featured UDF
- 07 CAPS' AIM Overview
- 08 AIM Software Structure
- 09 AIM Utility Functions & Tessellations
- 10 AIM Analysis Input, Execution and Output
- 11 AIM Sensitivities
- 12 AIMs and Units
- 13 Documenting AIMs
- 14 AIM Bounds & the AIM Discretization Structure
- 15 AIM Mesh Writing