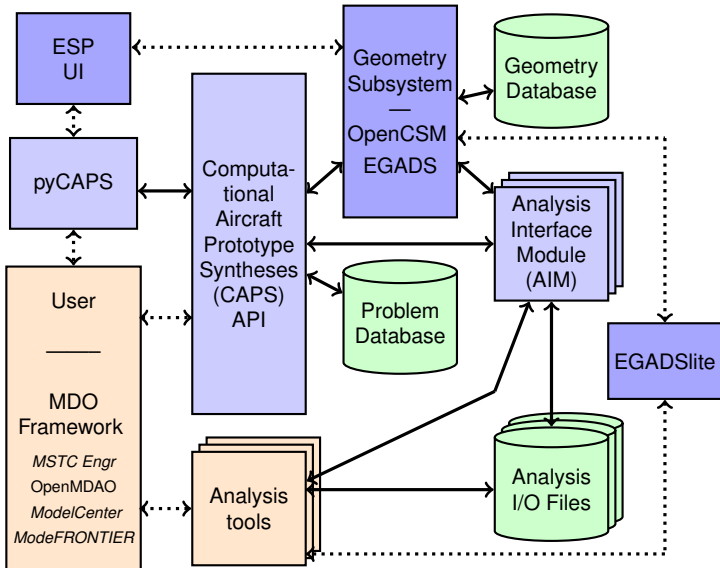




# Computational Aircraft Prototype Syntheses: The CAPS API for ESP Rev 1.22

Bob Haimes   Marshall Galbraith  
[haimes@mit.edu](mailto:haimes@mit.edu)   [galbramc@mit.edu](mailto:galbramc@mit.edu)  
Aerospace Computational Design Lab  
Massachusetts Institute of Technology

Note: Sections in **red** are changes in CAPS from Revision 1.21.



## Changing Thrusts Beginning at Rev 1.19

CAPS was originally designed to run concurrently with an MDO framework. This has turned out to be rarely the method of execution. In addition there were always issues in restarting from where the runs left off (due to the amount of state info stored in AIMs, the difficulty in getting to the correct place in the control program and the scattering of files). Also if MDO frameworks are not used, then additional execution support is required within the CAPS environment. So the enhancements include:

- Restarting runs the same script (or control program) *recycling* previous data.
- AIM reload. The AIMs ended up maintaining too much internal *state*, which made restarting almost impossible (requiring either rerunning or writing out the state). The AIMs need recasting not to hold on to extraneous data.
- A file structure where the *Problem Database* contains all of the *Analysis I/O Files* (seen in the block diagram on the previous slide).
- Better support for Analysis execution, which embraces asynchronous CAPS running when the Analysis is not run directly in the AIM.
- More emphasis on tracking data and decisions during the session.
- Enhanced handling of derivatives from both geometry construction and analysis output.
- Removal of Value Object of Value Objects.

## Variable Dimension GeometryIn Value Objects

Now that OpenCSM supports the ability to change the size of its *Design* and *Configuration Parameters* (GeometryIn Value Objects), this complicates dealing with derivatives associated with these inputs. This is because the meaning and use of rows and columns are now malleable. There are now internal *slots* for derivatives with respect to GeometryOut Value Objects, which are internally *registered* when `caps_getDot` is called. This is done via specifying which row/column is in play. The same is true for DataSet Objects, which request sensitivity information.

Note that when a changing a GeometryIn Value Object that effects the size of other GeometryIn Value Objects:

- 1 You can get which other GeometryIn Value Objects are effected when calling `caps_setValue` (see `nGIVAL` and `GIVALS`).
- 2 Any GeometryOut Value *slots* associated with changed size GeometryIn Objects are invalidated and removed. These would need to get reregistered if still needed.
- 3 Any DataSets associated with the changed-size GeometryIn Value Objects are also removed and need to be reinstated if still required.

## 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 bypassed by the *linkage* connection to another Value (or *DataSet*) Object of the same (*sub*)*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
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 correct capitalization



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”.

## Filtering within AIM Code

Each AIM can adopt it's own filtering scheme for down-selecting how to use each Body it receives. The “capsIntent” string is accessible to the AIM, but it is for information only.

## 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 to disambiguate which AIM instance should receive a given Body targeted for the AIM. 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. The attribute “capsIntent” is a semicolon-separated list of keywords. If the string to `caps_makeAnalysis` is **NUL**L, all Bodies with a “capsAIM” attribute that matches the AIM name are given to the AIM instance.

## capsLength

This string Attribute must be applied to an EGADS Body to indicate the length units used in the geometric construction.

## capsBound

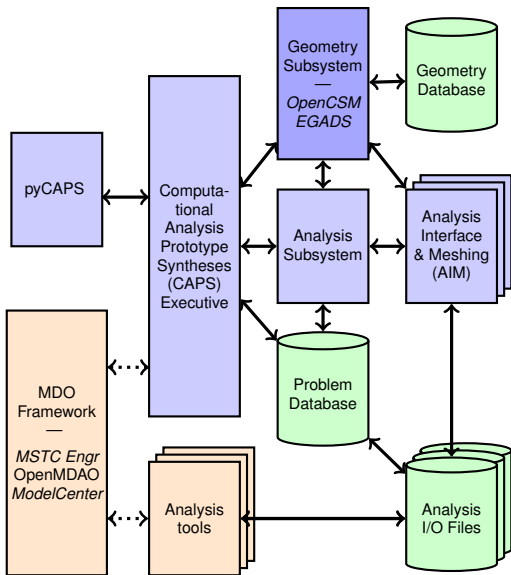
This string Attribute must be applied to EGADS BRep Objects to indicate which CAPS Bound(s) are associated with the geometry. A entity can be assigned to multiple Bounds by having the Bound names separated by a semicolon. Face examples could be “Wing”, “Wing;Flap”, “Fuselage”, and etc.

Note: Bound names should not cross dimensional lines.

## capsGroup

This string Attribute can be applied to EGADS BRep Objects to assist in grouping geometry into logical sets. A geometric entity can be assigned to multiple groups in the same manner as the capsBound attribute.

Note: CAPS does not internally use this, but is suggested of classifying geometry.



#### Setup (or read) the Problem:

- Initialize Problem with *csm* (or *static*) file  
GeomIn and GeomOut parameters
- Specify *mission* parameters
- Make Analysis instances  
AnalysisIn and AnalysisOut params
- Create *Bounds*, *VetrexSets* & *DataSets*
- Establish linkages between parameters

#### Run the Problem:

- Adjust the appropriate parameters
- Regenerate Geometry (if *dirty* – *lazy*)
- Call for Analysis Input file generation
- AIM Execute runs each *solver*
- Inform CAPS that an Analysis has run  
fills AnalysisOut, AnalysisDynO Objects & *DataSets* (*lazy*)
- Generate *Objective Function*

## CAPS Execution Phases

CAPS has 4 fundamental modes for starting the 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 directory. The initialization can either be from a CSM, geometry file, an OpenCSM or EGADS Model.
- Continuation – This occurs when CAPS has not fully completed a *phase* either do to an interruption or not reaching `caps_close` (where the *phase* is marked as completed). In this case the CAPS application or pyCAPS script can be run from the beginning, but *recycling* of results is used to quickly get to the position where the *phase* terminated.
- Starting from a completed *phase*. There are options for ignoring the deletion markers on Objects, read-only and reloading the CSM file.

This is controlled by the Problem Object's initialization using `caps_open`.

## CAPS Directory Structure

At the top level **prName** (of `caps_open`) you will find *phase* sub-directories. Note that *Scratch* is not as protected as the others.

In each *phase* subdirectory you may see:

- `capsCSMFiles` – A directory containing the CSM/UDC files used for reloading the geometry – must include the file *capsCSMLoad*.  
Can be generated by a call to `caps_phaseNewCSM` (see page 18).
- `capsRestart.cpc` – A CSM saved state file – or –
- `capsRestart.egads` – An EGADS saved geometry file (for nonparametric runs).
- `capsRestart` – This subdirectory contains the CAPS restart data.
- `capsClosed` – An indication that the *phase* has been closed (`caps_close` has been called marking completion).
- `capsLock` – A flag that another application is using this subdirectory.
- `AIMnames` – any number of directories each related to an AIM instance in the running CAPS Problem.

## CAPS Modes of Analysis Execution

There are 3 different ways that Analyses can be executed:

- **Manual** – This is the default mode. It requires a call to `caps_preAnalysis` (page 59), the execution of the solver (use `caps_system` – page 59 if the execution is performed via the command line) and then `caps_postAnalysis`, see page 60.
- **By the AIM** – If the AIM can execute the Analysis (noted by the return argument **exec** from either `caps_queryAnalysis` – page 52 or `caps_makeAnalysis` – page 56) use `caps_execute` (page 53) to perform “pre”, “exec” and “post”.
- **Automatic** – Again if the AIM does the Analysis execution (see above) and the flag **exec** was set for *auto-exec* on input when instantiating the AIM using `caps_makeAnalysis` (page 56) then the Analysis is triggered automatically when data associated with the AIM is retrieved. This happens during invocations of `caps_getData` (page 66) or `caps_getValue` (page 34).

The Python CAPS API is built on `ctypes` and mirrors the C/C++ API

Methods have similar names and arguments, which are ordered consistently when possible (optional arguments are placed last)

C-arrays with strides are `lists` of `tuples`

Import statement: `from pyCAPS import caps`

Main API classes: `caps.capsObj` `caps.c_capsObj`

- `c_capsObj` is a `ctypes` struct for C function arguments of C `capsObj` type
- Python class `capsObj` wrap a `c_capsObj` and implement CAPS API
  - The wrapped `c_capsObj` is automatically deleted when a Python class is created from a `caps` method

Note: this is not pyCAPS but mostly a one-to-one wrapping of the C/C++ API

See `$ESP_ROOT/doc/CAPS/html/pyCAPShtml/index.html` or

`$ESP_ROOT/doc/CAPS/pdf/pyCAPS.pdf` for the full pyCAPS documentation



## Get CAPS revision

```
caps_revision(int *major, int *minor)  
imajor, iminor = caps.revision()
```

**major** the returned major revision

**minor** the returned minor revision number

## Check State of CAPS Problem Phase

```
icode = caps_phaseState(const char *prNm, const char *phNm, int *bts)  
bts = caps.phaseState(prNm, phNm)
```

**prNm** the path ending with the CAPS problem name

**phNm** the queried *phase* name (**NULL** is equivalent to *Scratch*)

**bts** the returned state (additive): 1 – locked, 2 – closed, 4 – no capsRestart directory

**icode** the integer return code

## Setup for new Phase changing the CSM file

```
icode = caps_phaseNewCSM(const char *prName, const char *phName,  
                        const char *csm)  
    caps.phaseNewCSM(prName, phName, csm)
```

**prName** the path ending with the CAPS problem name

**phName** the new *phase* name

**csm** the CSM file to use in the new *phase* – for caps\_open **flag** = 5

**icode** the integer return code

The above functions may be called before CAPS *proper* is started via the invocation of caps\_open

## Open CAPS Problem Phase

```
icode = caps_open(const char *prName, const char *phName, int flag,
                  void *ptr, int outLevel, capsObj *problem,
                  int *nErr, capsErrs **errs)
```

```
problem = caps.open(prName, phName, flag, ptr, outLevel=1)
```

**prName** the path ending with the CAPS problem name

if exists the stored data initializes the problem, otherwise the directory is created

**phName** the current *phase* name (**NULL** is equivalent to *Scratch*)

**flag** 0 – **ptr** is a filename, 1 – **ptr** is an OpenCSM Model Structure, 2 – **ptr** is a Model **ego**,

3 – **ptr** is the starting *phase* name, 4\* – continuation (**ptr** can be **NULL**),

5 – **ptr** is the starting *phase* name with reloading of the CSM/UDC files †,

6 – **ptr** is the starting *phase* name but does not remove Objects marked for deletion,

7 – Open the existing **phName** in read-only mode (**ptr** can be **NULL**)

**ptr** input path/filename (**flag** == 0) – based on file extension:

\*.csm initialize the project using the specified OpenCSM file

\*.egads initialize the project based on the static geometry

– or – pointer to OpenCSM/EGADS Model – left open after caps\_close

**outLevel** 0 - minimal, 1 - standard (default), 2 - debug

**problem** the returned CAPS problem Object

**nErr** the returned number of errors generated – 0 means no errors

**errs** the returned CAPS error structure – **NULL** with no errors

**icode** the integer return code

Notes: \* A continuation can only occur on the same setup as initialized (ESP rev, version of OpenCASCADE and machine architecture)

† These files must be placed in the *capsCSMFiles* subdirectory (of the empty Phase directory) before calling caps\_open

## Specify a Call-back for Broken Links

```
icode = caps_brokenLink(void (*callBack) (capsObj problem, capsObj obj,  
                                     enum capstMethod tmethod, char *name,  
                                     enum capsstType stype))
```

**callBack** the function to be called when links are found to be broken – or –  
**NULL** to remove an existing call-back

**problem** the almost complete reloaded Problem Object

**obj** is the existing Object that has lost its link (either source or target –  
see **stype**)

**tmethod** the transfer method used for the broken link

**name** the name of the lost Value Object

**stype** the subtype of the lost Value Object

**icode** the integer return code

This is only needed if `caps_open` is invoked with **flag** as 5 or there are deleted Parameter Value Objects, Bounds and/or Analysis Objects (note that this must be called before `caps_open`).

If there are existing links that are broken due to the changes in the objects then the function **callBack** is invoked for each broken link during `caps_open`.

Note that this is not *thread safe* for multi-thread/multi-Problem situations. If you wish to have different call-backs per Problem initialization you will need to ensure the calls to `caps_open` are sequential.

## Do not use CAPS signal handling

```
caps_externSignal()
```

Must be called before `caps_open`. Calling program is responsible for invoking `caps_rmLock()` on any abort, which deletes the `capsLock` file.

## Get Problem root

```
icode = caps_getRootPath(const capsObj problem, const char **fullPath)  
fullPath = problem.getRootPath()
```

**problem** the input CAPS Problem Object

**fullPath** the file path to find the root of the Problem/Phase directory structure  
if on Windows it will contain the drive

**icode** integer return code

Note: All other uses of *path* is relative to this point.

## Close CAPS Problem

```
icode = caps_close(capsObj problem, int complete, const char *phName)  
del problem or problem.close(complete = 0, phName = None)
```

- problem** the input CAPS problem is written to disk and closed; memory cleanup is performed
- complete** -1 – remove the *phase*, 0 – the *phase* is left open, 1 – the *phase* is completed
- phName** Phase Name of the Scratch phase is closed as complete
- icode** the integer return code

Notes: If caps\_open was initialized with an OpenCSM or EGADS Model, it is left open. All Analyses must be past *Post* to be complete.

## Information about an Object

```
icode = caps_info(capsObj object, char **name, enum capsoType *otype,  
                  enum capssType *stype, capsObj *link,  
                  capsObj *parent, capsOwn *last)
```

```
name, otype, stype, link, parent, last = object.info()
```

**object** the input CAPS Object

**name** the returned Object name pointer (if any)

**otype** the returned Object type: Problem, Value, Analysis, Bound, VertexSet, DataSet

**stype** the returned subtype (depending on **otype**)

**link** the returned linkage Value Object (**NULL** – no link)

**parent** the returned parent Object (**NULL** for a Problem or an Attribute generated User Value)

**last** the returned last owner to *touch* the Object

**icode** integer return code, can be 1 indicating the Object is marked for deletion

## Number of Children in a Parent Object

```
icode = caps_size(capsObj object, enum capsObjType type,  
                 enum capsObjType subtype, int *size, int *nErr,  
                 capsErrors **errs)
```

```
size = object.size(type, subtype)
```

**object** the input CAPS Object

**type** the data type to size: Bodies, Attributes, Value, Analysis, Bound, VertexSet, DataSet

**subtype** the subtype to size (depending on type)

**size** the returned size

**nErr** the returned number of errors generated – 0 means no errors

**errs** the returned CAPS error structure – **NULL** with no errors

**icode** integer return code

## Mark an Object for Deletion

```
icode = caps_markForDelete(capsObj object)  
object.markForDelete()
```

**object** the Object to be deleted in the next Phase

Note: only Value Objects of subtype Parameter, Analysis and Bound Objects may be deleted! Value Objects of subtype User are automatically removed at Phase closure.

**icode** integer return code



## Get Child by Index

```
icode = caps_childByIndex(capsObj object, enum capsoType type,  
                          enum capssType sty, int ind, capsObj *child)  
child = object.childByIndex(type, sty, ind)
```

**object** the input parent Object

**type** the Object type to return: Value, Analysis, Bound, VertexSet, DataSet

**sty** the subtype to find (depending on type)

**ind** the index [1-size]

**child** the returned CAPS Object

**icode** integer return code

## Get Child by Name

```
icode = caps_childByName(capsObj object, enum capsoType type,  
                        enum capssType stype, const char *name,  
                        capsObj *child, int *nErr, capsErrs **errs)  
child = object.childByName(type, stype, name)
```

**object** the input parent Object

**type** the Object type to return: Value, Analysis, Bound, VertexSet, DataSet

**stype** the subtype to find (depending on type)

**name** a pointer to the index character string

**child** the returned CAPS Object

**nErr** the returned number of errors generated – 0 means no errors

**errs** the returned CAPS error structure – NULL with no errors

**icode** integer return code

## Set Verbosity Level

```
icode = caps_outLevel(capsObj problem, int outLevel)  
oldOutLevel = problem.outLevel(outLevel)
```

**problem** the CAPS problem object

**outLevel** 0 - minimal, 1 - standard (default), 2 - debug

**icode** the integer return code / old outLevel

## Get Body by index

```
icode = caps_bodyByIndex(capsObj obj, int index, ego *body,  
                        char **unit)  
body = obj.bodyByIndex(index)
```

**obj** the input CAPS Problem or Analysis Object

**index** the index [1-**size**] – see caps\_size, page 24

**body** the returned EGADS Body Object ([egads.ego](#))

**units** pointer to the string declaring the length units – **NULL** for unitless values

**icode** integer return code

## Get Error Information

```
icode = caps_errorInfo(capsErrs *errors, int eindex, capsObj *errObj,  
                      int *eType, int *nLines, char ***lines)
```

```
lines = errors.info()
```

**errors** the input CAPS Error structure

**eindex** the index into **errors** (1 bias)

**errObj** the offending CAPS Object

**eType** the returned error type (CINFO, CWARN, CERROR or CSTAT)

**nLines** the returned number of comment lines to describe the error

**lines** a pointer to a list of character strings with the error description

**icode** integer return code

## Free Error Structure

```
icode = caps_freeError(capsErrs *errors)  
del errors
```

**errors** the CAPS Error structure to be freed

**icode** integer return code

## Write Geometry Parameter File

```
icode = caps_writeParameters(const capsObj problem, char *fileName)  
    problem.writeParameters(fileName)
```

**problem** the input CAPS Problem Object

**fileName** the name of the parameter file to write

**icode** integer return code

Note: This outputs an OpenCSM Design Parameter file.

## Read Geometry Parameter File

```
icode = caps_readParameters(const capsObj problem, char *fileName)  
    problem.readParameters(fileName)
```

**problem** the input CAPS Problem Object

**fileName** the name of the parameter file to read

**icode** integer return code

Note: This reads an OpenCSM Design Parameter file and overwrites (makes *dirty*) the current state for the GeometryIn Values in the file.

## Write out Geometry

```
icode = caps_writeGeometry(capsObj obj, int flag, const char *fName,  
                           int *nErr, capsErrs **errs)  
obj.writeGeometry(fileName, flag = 1)
```

**obj** the input CAPS Problem/Analysis Object

**flag** the write flag: **0** – no additional output, **1** – also write Tessellation Objects for EGADS output (only for Analysis Objects)

**fName** the name of the file to write – typed by extension (case insensitive):

iges/igs – IGES File

step/stp – STEP File

brep – OpenCASCADE File

egads – EGADS file (which includes attribution)

**nErr** the returned number of errors generated – **0** means no errors

**errs** the returned CAPS error structure – **NULL** with no errors

**icode** integer return code

**Note:** The *EGADS Tessellation Objects* used by the Analysis Object are written in the EGADS output file along with the geometry of the Bodies.

## Get History of an Object

```
icode = caps_getHistory(capsObj obj, int *nhist, capsOwn **hist)  
hist = obj.getHistory()
```

**obj** the input CAPS Object

**nhist** the returned length of the history list

**hist** the returned pointer to the list of History entities (**nhist** in length)

**icode** integer return code

## Set the Intent Phrase for History tracking

```
icode = caps_intentPhrase(capsObj problem, int nLines, char **lines)  
problem.intentPhrase(lines)
```

**problem** the CAPS Problem Object to set the phrase

**nLines** the number of comment lines to describe the intent phrase  
can be 0 to unset any phrase

**lines** a pointer to a list of character strings with the description  
can be **NULL** if **nLines** is 0

**icode** integer return code

## Get Owner Information

```
icode = caps_ownerInfo(const capsObj problem, const capsOwn owner,  
                       char **phase, char **pname, char **pID,  
                       char **userID, int *nLines, char ***lines,  
                       short *datetime, CAPSLONG *sNum)  
pname, pID, userID, lines, datetime, sNum = owner.info()
```

**problem** the CAPS Problem Object

**owner** the input CAPS Owner structure

**phase** the returned Phase Name when this entry was generated (can be **NULL**)

**pname** the returned pointer to the process name

**pID** the returned pointer to the process ID

**userID** the returned pointer to the user ID

**nLines** the returned number of comment lines to describe the intent phrase

**lines** a returned pointer to a list of character strings with the description

**datetime** the filled date/time stamp info – 6 in length:  
year, month, day, hour, minute, second

**sNum** the sequence number (always increasing)

**icode** integer return code



## Create A Value Object

```

icode = caps_makeValue(capsObj problem, const char *vname,
                      enum capsType stype, enum capsVType vtype,
                      int nrow, int ncol, const void *data,
                      int *partial, const char *units, capsObj *val)
val = problem.makeValue(vname, stype, data)

```

**problem** the input CAPS Problem Object where the Value to to reside

**vname** the Value Object name to be created

**stype** the Object subtype: Parameter or User

**vtype** the value data type:

0	Boolean	2	Double	4	String Tuple
1	Integer	3	String		

**nrow** number of rows

**ncol** number of columns – *Value length = nrow \* ncol*

**data** pointer to the appropriate block of memory

must be a pointer to a contiguous block of memory for strings (each zero terminated)

must be a pointer to a *capsTuple* structure(s) when **vtype** is a Tuple

**partial** integer vector/array containing specific *ntype* indications

**units** string pointer declaring the units for **vtype 2** – **NULL** for unitless values

if **vtype** is **3** and **units** is “PATH” – slashes are converted automatically

**val** the returned CAPS Value Object

**icode** integer return code

## Retrieve Values

```
icode = caps_getValue(capsObj val, enum capsvType *vtype, int *nrow,
                      int *ncol, const void **data,
                      const int **partial, const char **units,
                      int *nErr, capsErrs **errs)
```

```
data = val.getValue()
```

**val** the input Value Object

**vtype** the returned data type:

0	Boolean	2	Double	4	String Tuple	6	Double w/ Deriv
1	Integer	3	String	5	AIM pointer		

**nrow** returned number of rows

**ncol** returned number of columns – *Value length = nrow \* ncol*

**data** a filled pointer to the appropriate block of memory (**NULL** – don't fill)

Can use `caps_childByIndex` (page 25) to get Value Objects

**partial** a returned integer vector/array containing specific `ntype` indications

**NULL** is returned except for `ntype` is 'partial' – filled with 'not NULL' or 'is NULL'

**units** the returned pointer to the string declaring the units

if **vtype** is 3 and **units** "PATH" – slashes are converted automatically

**nErr** the returned number of errors generated (Analysis Out) – 0 means no errors

**errs** the returned CAPS error structure (Analysis Out) – **NULL** with no errors

**icode** integer return code

Use the structure *capsTuple* when casting **data** if a Tuple (4)

## Reset A Value Object

```
icode = caps_setValue(capsObj val, enum capsvType vtype, int nrow,
                     int ncol, const void *data, const int *partial,
                     const char *units, int *nErr, capsErrs **errs)
```

```
val.setValue(data)
```

**val** the input CAPS Value Object (not for GeometryOut, AnalysisOut or AnalysisDynO)

**vtype** the data type:

0	Boolean	2	Double	4	String Tuple
1	Integer	3	String	5	AIM pointer

**nrow** number of rows

**ncol** number of columns – *Value length* = **nrow** \* **ncol**

**data** pointer to the appropriate block of memory used to reset the values; must point to a contiguous block of memory for *Value length* strings (each zero terminated)

**partial** an integer vector/array of *Value length* containing specific **ntype** indications ignored for length = 1 or **ntype** is 'NULL invalid' – may be NULL if non-NULL **ntype** is set to 'partial' – must be filled with 'not NULL' or 'is NULL' See caps\_getValueProp

**units** the string declaring the units for **data**

**nErr** the returned number of errors generated (Geometry In) – 0 means no errors

**errs** the returned CAPS error structure (Geometry In) – NULL with no errors

**icode** integer return code

## Get Valid Value Range

```
icode = caps_getLimits(capsObj val, capsvType *vtype,
                      const void **limits, const char **units)
```

See the Appendix on Limits

**val** the input Value Object  
**vtype** the data type:  
     -2 **Doubles** | -1 **Integers** | 1 **Integer** | 2 **Double**  
**limits** an returned pointer to a block of memory containing the valid range  
     [2 or 2\*nrow\*ncol\*sizeof(vtype) in length] – or – **NULL** if not yet filled  
**units** a string units of the limits  
**icode** integer return code

Note: use caps\_getValue or caps\_getValueSize if **nrow** and/or **ncol** are needed.

## Retrieve the Value's Size

```
icode = caps_getValueSize(capsObj val, int *nrow, int *ncol)
nrow, ncol = val.getValueSize()
```

**val** the input Value Object  
**nrow** returned number of rows  
**ncol** returned number of columns – *Value length = nrow \* ncol*  
**icode** integer return code

Note: this does not possibly initiate auto-execution like caps\_getValue.

## Set Valid Value Range

```
icode = caps_setLimits(capsObj val, capsvType vtype, void *limits,
                      const char *units, int *nErr, capsErrs **errs)
```

See the Appendix on Limits

- val** the input Value Object (only for the User & Parameter subtypes)
- vtype** the data type of the limits pointer:
 

-2	Doubles		-1	Integers		1	Integer		2	Double
----	---------	--	----	----------	--	---	---------	--	---	--------
- limits** a pointer to the appropriate block of memory which contains the minimum and maximum range allowed (2 or 2\*nrow\*ncol in length)
- units** a string units of the limits
- nErr** the returned number of errors generated – 0 means no errors
- errs** the returned CAPS error structure – NULL with no errors
- icode** integer return code

Note: use caps\_getValue or caps\_getValueSize if nrow and/or ncol are needed.

## Get Value Properties

```
icode = caps_getValueProps(capsObj val, int *dim, int *gInType,
                           enum capsFixed *lfix, enum capsFixed *sfix,
                           enum capsNull *ntype)
```

```
dim, pmtr, lfix, sfix, ntype = val.getValueProps()
```

**val** the input Value Object

**dim** the returned dimensionality:

**0** scalar only

**1** vector or scalar

**2** scalar, vector or 2D array

**gInType** the returned type: **0** – GeometryIn type → OCSM\_DESPMTR (or not GeomIn),

**1** – GeometryIn type → OCSM\_CFGPMTR,

**2** – GeometryIn type → OCSM\_CONPMTR

**lfix** **0** – the length(s) can change, **1** – the length is fixed

**sfix** **0** – the Shape can change, **1** – Shape is fixed

**ntype** **0** – NULL invalid, **1** – not NULL, **2** – is NULL, **3** – partial NULL

**icode** integer return code

## Set Value Properties

```
icode = caps_setValueProps(capsObj val, int dim, enum capsFixed lfix,  
                           enum capsFixed sfix, enum capsNull ntype,  
                           int *nErr, capsErrs **errs)
```

```
val.setValueProps(dim, lfix, sfix, ntype)
```

**val** the input Value Object (only for the User & Parameter subtypes)

**dim** the dimensionality:

**0** scalar only

**1** vector or scalar

**2** scalar, vector or 2D array

**lfix** **0** – the length(s) can change, **1** – the length is fixed

**sfix** **0** – the Shape can change, **1** – Shape is fixed

**ntype** **0** – NULL invalid, **1** – not NULL, **2** – is NULL

**nErr** the returned number of errors generated – **0** means no errors

**errs** the returned CAPS error structure – **NULL** with no errors

**icode** integer return code

## Units conversion

```
icode = caps_convertValue(capsObj val, double inVal,  
                           const char *inUnit, double *outVal)  
val.convertValue(inVal, inUnit)
```

**val** a Value Object

**inVal** the source value to be converted

**inUnit** the pointer to the string declaring the source units

**outVal** the returned converted value in the units of the **val** Value Object

**icode** integer return code

## Free memory in Value Structure

```
caps_freeValue(capsValue *value)  
del value
```

**value** a pointer to the Value structure to be cleaned up



## Transfer Values

```
icode = caps_transferValues(capsObj src, enum capstMethod tmethod,  
                           capsObj dst, int *nErr, capsErrs **errs)  
dst.transferValues(tmethod, src)
```

**src** the source input Value Object (not for Tuple vtypes) – or –  
DataSet Object

**tmethod** 0 – copy, 1 – integrate, 2 – weighted average – (1 & 2 only for DataSet **src**)

**dst** the destination Value Object to receive the data

Notes:

- Must not be GeometryOut, AnalysisOut or AnalysisDynO
- Shapes must be compatible
- Overwrites any Linkage

**nErr** the returned number of errors generated – 0 means no errors

**errs** the returned CAPS error structure – **NULL** with no errors

**icode** integer return code

## Establish Linkage between Value Objects

```
icode = caps_linkValue(capsObj link, enum capstMethod tmethod,  
                      capsObj trgt, int *nErr, capsErrs **errs)  
trgt.linkValue(link, tmethod)
```

**link** linking Value Object (not for AnalysisDynO, User subtype or Tuple vtype)  
– or – DataSet Object

**tmethod** 0 – copy, 1 – integrate, 2 – weighted average – (1 & 2 only for DataSet **link**)

**trgt** the target Value Object which will get its data from **link**

Notes:

- Must not be GeometryOut, AnalysisOut or AnalysisDynO
- Shapes must be compatible
- **link** = **NULL** – removes any Linkage

**nErr** the returned number of errors generated – 0 means no errors

**errs** the returned CAPS error structure – **NULL** with no errors

**icode** integer return code

Note: circular linkages are not allowed!

## Set the Finite-Difference Step Size

```
icode = caps_setStepSize(capsObj val, const double *sizes)
    val.setStepSizeSize(sizes) [See the Appendix on StepSize]
    val the input CAPS Value Object (GeometryIn/DESPMTR types only)
    sizes the FD step sizes for each Value member (nrow*ncol in length)
    a zero indicates use analytic derivatives; can be NULL – set all to zero
    icode integer return code
```

## Get the Finite-Difference Step Size

```
icode = caps_getStepSize(capsObj val, const double **sizes)
    sizes = val.getStepSizeSize() [See the Appendix on StepSize]
    val the input CAPS Value Object (GeometryIn/DESPMTR types only)
    sizes the returned FD step sizes for each Value member (nrow*ncol in length)
    a zero indicates use analytic derivatives; can be NULL – all set to zero
    icode integer return code
```

Should only be used for debugging purposes or if OpenCSM uses Finite Differences and the default step is poor for the design parameter at-hand

## Get a list of Derivatives available

```
icode = caps_hasDeriv(capsObj val, int *ndot, char ***names,  
                     int *nErr, capsErrs **errs)  
names = val.hasDeriv()
```

**val** the input CAPS Value Object (*DoubleDeriv* type only)

**ndot** the returned length of the number of dots available

**names** the returned pointer to the list of derivative names (**ndot** in length – freeable)

**nErr** the returned number of errors generated – **0** means no errors

**errs** the returned CAPS error structure – **NULL** with no errors

**icode** integer return code

*DoubleDeriv* types only exist for GeometryOut and certain AnalysisOut as well as AnalysisDynO Value Objects

## Get Derivative values

```
icode = caps_getDeriv(capsObj val, const char *name, int *len,  
                     int *len_wrt, double **deriv, int *nErr,  
                     capsErrs **errs)
```

```
deriv = val.getDeriv(name)
```

**val** the input CAPS Value Object (*DoubleDeriv* type only)

**name** the input name of the derivative

**len** the returned rows of the deriv (the length of the Value Object)

**len\_wrt** the returned columns of deriv (length of the w.r.t. Value Object)

**deriv** the returned pointer to the derivative information (**len** x **len\_wrt** in length)

**nErr** the returned number of errors generated – **0** means no errors

**errs** the returned CAPS error structure – **NULL** with no errors

**icode** integer return code

*DoubleDeriv* types only exist for GeometryOut and certain AnalysisOut as well as AnalysisDynO Value Objects.

For 2D Value Object or w.r.t. Value Object the indexing is flattened where the column index has no stride (i.e.  $irow \times ncol + icol$ ).

## Convert value between units

See Appendix Python Units

```
icode = caps_convert(int count, const char *inUnit, double *inVal,  
                    const char *outUnit, double *outVal)
```

**count** length of **inVal** and **outUnit** arrays

**inUnit** a string representing the units of **inVal**

**inVal** the input values to be converted

**outUnit** a string representing the desired units of **outVal**

**outVal** the output values in units of **outUnit** (may be same pointer as **inVal**)

**icode** integer return code

## Multiply units

See Appendix Python Units

```
icode = caps_unitMultiply(const char *unitL, const char *unitR,  
                        char **outUnit)
```

**unitL** a input string representing units

**unitR** a input string representing units

**outUnit** a string representing the resulting units from multiplying **unitL** and **unitR**

**icode** integer return code

## Divide units

See Appendix Python Units

```
icode = caps_unitDivide(const char *unitL, const char *unitR,  
                        char **outUnit)
```

**unitL** a input string representing units

**unitR** a input string representing units

**outUnit** a string representing the resulting units from dividing **unitL** and **unitR**

**icode** integer return code

## Raise units

See Appendix Python Units

```
icode = caps_unitRaise(const char *unit, int power, char **outUnit)
```

**unit** a input string representing units

**power** power to raise **unit**

**outUnit** a string representing the resulting units from raising **unit** to **power**

**icode** integer return code

## Invert units

See Appendix Python Units

```
icode = caps_unitInvert(const char *unit, char **outUnit)
```

**unit** a input string representing units

**outUnit** a string representing the resulting units from inverting **unit**

**icode** integer return code

## Offset units

See Appendix Python Units

```
icode = caps_unitOffset(const char *unit, double off, char **outUnit)
```

**unit** a input string representing units

**off** offset to apply to **unit**

**outUnit** a string representing the resulting units from offsetting **unit** by **off**

**icode** integer return code

## Valid unit string

See Appendix Python Units

```
icode = caps_unitParse(const char *unit)
```

**unit** a input string representing units

**icode** integer return code (CAPS\_SUCCESS if valid, CAPS\_UNITERR otherwise)



## Valid unit conversion

See Appendix Python Units

```
icode = caps_unitConvertible(const char *unitL, const char *unitR)
```

**unitL** a input string representing units  
**unitR** a input string representing units  
**icode** integer return code (CAPS\_SUCCESS **unitL** is convertible to **unitR**, CAPS\_UNITERR otherwise)

## Unit comparison

See Appendix Python Units

```
icode = caps_unitCompare(const char *unitL, const char *unitR,  
                        int *compare)
```

**unitL** a input string representing units  
**unitR** a input string representing units  
**compare** signed difference between **unitL** and **unitR**  
**icode** integer return code

## Get Attribute by name

```
icode = caps_attrByName(capsObj object, char *name, capsObj *attr)  
attr = object.attrByName(name)
```

**object** any CAPS Object

**name** a string referring to the Attribute name

**attr** the returned User Value Object  
will be deleted at the end of the *phase*

**icode** integer return code

## Get Attribute by index

```
icode = caps_attrByIndex(capsObj object, int index, capsObj *attr)  
attr = object.attrByIndex(index)
```

**object** any CAPS Object

**index** the index (bias 1) to the list of Attributes

**attr** the returned User Value Object – Attribute name is the Value Object name  
will be deleted at the end of the *phase*

**icode** integer return code

## Set an Attribute

```
icode = caps_setAttr(capsObj object, const char *name, capsObj attr)
object.setAttr(attr, name=None)
```

**object** any CAPS Object

**name** a string referring to the Attribute name – **NULL**: use name in **attr**  
Note: an existing Attribute of this name is overwritten with the new value

**attr** the Value Object containing the attribute  
2D arrays and Tuples are not supported; 1D arrays will have rows only

**icode** integer return code

## Delete an Attribute

```
icode = caps_deleteAttr(capsObj object, char *name)
object.deleteAttr(name)
```

**object** any CAPS Object

**name** a string referring to the Attribute to delete  
**NULL** deletes all attributes attached to the Object

**icode** integer return code

## Query Analysis – Does not ‘load’ or create an object

```
icode = caps_queryAnalysis(capsObj problem, const char *aname,  
                           int *nIn, int *nOut, int *exec)  
nIn, nOut, execute = problem.queryAnalysis(aname)
```

**problem** a CAPS Problem Object

**aname** the Analysis (and AIM plugin) name

**nIn** the returned number of Inputs

**nOut** the returned number of Outputs

**exec** returned execution flag: **0** – no execution, **1** – aimExecute exists (can auto-exec)

**icode** integer return code

Note: this causes the the DLL/Shared-Object to be loaded (if not already resident)

## Execute Geometry Build or Analysis

```
icode = caps_execute(capsObj object, int *status, int *nErr,  
                    capsErrs **errors)  
object.execute()
```

- object** the Analysis or Problem Object  
a *Geometry*-only regen is forced when this is a Problem Object  
for an Analysis Object that has `aimExecute` this runs `aimPreAnalysis`,  
`aimExecute` and `aimPostAnalysis`
- status** the returned status (**0** – done, **1** – running)  
currently unused – always returns **0**
- nErr** the returned number of errors generated – **0** means no errors
- errors** the returned CAPS error structure – **NULL** with no errors
- icode** integer return code

## Get Bodies

```
icode = caps_getBodies(capsObj aobj, int *nBody, ego **bodies,  
                      int *nErr, capsErrs **errs)  
bodies = aobj.getBodies()
```

- aobj** the Analysis Object
- nBody** the returned number of EGADS Body Objects that match the Analysis' intent
- bodies** the returned pointer to a list of EGADS Body/Node Objects (length – **nBody**)
- nErr** the returned number of errors generated – **0** means no errors
- errors** the returned CAPS error structure – **NULL** with no errors
- icode** integer return code

## Get Tessellations

```
icode = caps_getTessels(capsObj aobj, int *nTessel, ego **tessels,  
                      int *nErr, capsErrs **errs)  
tessels = aobj.getTessels()
```

- aobj** the Analysis Object
- nTessel** the returned number of EGADS Tessellation Objects
- tessels** the returned pointer to a list of EGADS Tessellations (length – **nTessel**)
- nErr** the returned number of errors generated – **0** means no errors
- errors** the returned CAPS error structure – **NULL** with no errors
- icode** integer return code

## Query Analysis Input Information

```
icode = caps_getInput(capsObj problem, const char *aname, int index,  
                      char **ainame, capsValue *default)
```

Not implemented yet

**problem** a CAPS Problem Object

**aname** the Analysis (and AIM plugin) name

**index** the Input index [1-nIn]

**ainame** a pointer to the returned Analysis Input variable name (use EG\_free to free memory)

**default** a pointer to the filled default value(s) and units – use caps\_freeValue to cleanup

## Query Analysis Output Information

```
icode = caps_getOutput(capsObj problem, const char *aname, int index,  
                      char **aoname, capsValue *form)
```

Not implemented yet

**problem** a CAPS Problem Object

**aname** the Analysis (and AIM plugin) name

**index** the Output index [1-nOut]

**aoname** a pointer to the returned Analysis Output variable name (use EG\_free)

**form** a pointer to the Value Shape & Units information – returned  
use caps\_freeValue to cleanup

## Create a new Analysis Object

```
icode = caps_makeAnalysis(capsObj problem, const char *aname,  
                          const char *name, const char *uSys,  
                          char *intent, int *exec, capsObj *analysis,  
                          int *nErr, capsErrs **errs)  
analysis = problem.makeAnalysis(aname, name, uSys=None,  
                                intent=None, execute=1)
```

**problem** a CAPS Problem Object

**aname** the Analysis (AIM plugin) name

**name** the unique supplied name for this instance (can be **NULL**)

**uSys** pointer to string describing the unit system to be used by the AIM (can be **NULL**)  
see specific AIM documentation for a list of strings for which the AIM will respond

**intent** the *intent* character string used to pass Bodies to the AIM, **NULL** – no filtering

**exec** the execution flag: On input **0** – no auto-exec, **1** – allow for auto-exec

On output **0** – no AIM execution, **1** – aimExecute exists

**analysis** the resultant Analysis Object

**nErr** the returned number of errors generated – **0** means no errors

**errors** the returned CAPS error structure – **NULL** with no errors

**icode** integer return code

Note: If **exec** is returned as **1** then `aimPreAnalysis`, `aimExecute` and `aimPostAnalysis` automatically run when `caps_execute` (page 53) is called. When **exec** is input (and output) as **1** the analysis can run in a *lazy* manner when there is a request for an AIM output or data transfer.



## Initialize Analysis from another Analysis Object

```
icode = caps_dupAnalysis(capsObj from, const char *name, capsObj *obj)  
obj = from.dupAnalysis(name)
```

- from** an existing CAPS Analysis Object
- name** the name of the duplicate Analysis Object
- obj** the resultant Analysis Object
- icode** integer return code

## Get Dirty Analysis Object(s)

```
icode = caps_dirtyAnalysis(capsObj obj int *nAobj, capsObj **aobjs)  
aobjs = obj.dirtyAnalysis()
```

- obj** a CAPS Problem, Bound or Analysis Object
- nAobjs** the returned number of *dirty* Analysis Objects
- aobjs** a returned pointer to the list of *dirty* Analysis Objects (*freeable*)
- icode** integer return code

Note: Listed from most *stale* to most recent – the order in which to execute.

## Get Info about an Analysis Object

```
icode = caps_analysisInfo(capsObj aobj, char **dir, char **uSys,
                          int *major, int *minor, char **intent,
                          int *nfields, char ***fnames, int **frank,
                          int **fInOut, int *exec, int *status)
dir, uSys, major, minor, intent, fnames, franks, fInOut,
execute, status = aobj.analysisInfo()
```

- aobj** the input Analysis Object
- dir** a returned pointer to the string specifying the directory for file I/O  
**name** (or **aname** augmented with the instance number) of caps\_makeAnalysis
- uSys** returned pointer to string describing the unit system used by the AIM (can be **NULL**)
- major** the returned AIM major version number
- minor** the returned AIM minor version number
- intent** the returned pointer to the *intent* character string used to pass Bodies to the AIM
- nfields** the returned number of fields for DataSet filling
- fnames** a returned pointer to a list of character strings with the field/DataSet names
- frank** 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)
- exec** returned execution flag: **0** – no AIM execution, **1** – aimExecute exists, **2** – auto-exec
- status** **0** – up to date, **1** – *dirty* Analysis inputs, **2** – *dirty* Geometry inputs  
**3** – both Geometry & Analysis inputs are *dirty*, **4** – new geometry,  
**5** – *post Analysis* required, **6** – Execution & *post Analysis* required
- icode** integer return code

## Generate Analysis Inputs

```
icode = caps_preAnalysis(capsObj analysis, int *nErr, capsErrs **errs)  
analysis.preAnalysis()
```

- analysis** the Analysis Object – use `caps_execute` (page 53) for *auto-exec* Objects  
Also use `caps_execute` to perform a *Geometry*-only regen
- nErr** the returned number of errors generated – 0 means no errors
- errs** the returned CAPS error structure – **NULL** with no errors
- icode** integer return code

## Execute the Command Line String

```
icode = caps_system(capsObj aobj, const char *rpath, const char *cmd)  
analysis.system(cmd, rpath=None)
```

- aobj** the Analysis Object
- rpath** the relative path from the Analysis' directory or **NULL** (in the Analysis path)
- cmd** the command line string to execute
- icode** integer return code

Notes:

- 1 only needed when explicitly executing the appropriate analysis solver (*i.e.*, not using the AIM)
- 2 should be invoked after `caps_preAnalysis` and before `caps_postAnalysis`
- 3 this must be used instead of the OS *system* call to ensure that journaling properly functions

## Mark Analysis as Run

```
icode = caps_postAnalysis(capsObj analysis, int *nErr,  
                          capsErrs **errors)  
analysis.postAnalysis()
```

**analysis** the Analysis Object – use `caps_execute` (page 53) for *auto-exec* Objects

**nErr** the returned number of errors generated – **0** means no errors

**errors** the returned CAPS error structure – **NULL** with no errors

**icode** integer return code

Note: this clears all Analysis Output Objects to force reloads/recomputes

## Create a Bound

```
icode = caps_makeBound(capsObj problem, int dim, const char *bname  
                      capsObj *bound)
```

```
bound = problem.makeBound(dim, bname)
```

**problem** the CAPS Problem Object

**dim** the dimensionality of the Bound (1 – 3)

**bname** the character string associated with “capsBound” attribute on bodies

**bound** the returned new CAPS Bound Object

## Get Information about a Bound

```
icode = caps_boundInfo(capsObj bound, enum capsState *state, int *dim,  
                      double *plims)
```

```
state, dim, plims = bound.boundInfo()
```

**bound** the CAPS Bound Object

**state** the returned Bound state:

-1 Open

0 Empty & Closed

1 single BRep entity

2 multiple BRep entities

-2 multiple BRep entities – Error in reparameterization!

**dim** the returned dimensionality of the Bound (1 – 3)

**plims** the filled parameterization limits (2 values when **dim** is 1, 4 when **dim** is 2)

## Make a VertexSet

```
icode = caps_makeVertexSet(capsObj bound, capsObj analysis,  
                           const char *vname, capsObj *vset,  
                           int *nErr, capsErrs **errs)
```

```
vset = bound.makeVertexSet(analysis, vname=None)
```

**bound** an input *open* CAPS Bound Object

**analysis** the Analysis Object (**NULL** – Unconnected)

**vname** a character string naming the VertexSet (can be **NULL** for a Connected VertexSet)

**vset** the returned VertexSet Object

**nErr** the returned number of errors generated – **0** means no errors

**errs** the returned CAPS error structure – **NULL** with no errors

## Get Info about a VertexSet

```
icode = caps_vertexSetInfo(capsObj vset, int *nGpts, int *nDpts,  
                           capsObj *bound, capsObj *analysis)
```

```
nGpts, nDpts, bound, analysis = vset.vertexSetInfo()
```

**vset** the VertexSet Object

**nGpts** the returned number of *Geometry* points in the VertexSet

**nDpts** the returned number of point *Data* positions in the VertexSet

**bound** the returned associated Bound Object

**analysis** the returned associated Analysis Object (**NULL** – Unconnected)

## Fill an Unconnected VertexSet

```
icode = caps_fillUnVertexSet(capsObj vset, int npts, double *xyzs)  
vset.fillUnVertexSet(xyzs)
```

**vset** the input Unconnected VertexSet Object  
**npts** the number of points in the VertexSet  
**xyzs** the point positions (3\***npts** in length)  
**icode** integer return code

## Close a Bound

```
icode = caps_closeBound(capsObj bound)  
bound.closeBound()
```

**bound** an input *open* CAPS Bound Object to close  
**icode** integer return code

## Output a VertexSet for Plotting/Debugging

```
icode = caps_outputVertexSet(capsObj vset, const char *filename)  
vset the VertexSet Object  
filename the VertexSet filename (should have the extension “.vs”)
```

The CAPS application **vVS** can be used to interactively view the file generated by this function.

**This is now deprecated because CAPS viewing has been integrated!**

## DataSet Naming Conventions

- Multiple DataSets in a Bound can have the same Name
- Allows for automatic data transfers
- One *source* (from either *FieldOut* or *User Methods*)
- Reserved Names:

DSet Name	rank	Meaning	Comments
xyz	3	<i>Geometry</i> Positions	
xyzd	3	<i>Data</i> Positions	Not for vertex-based discretizations
param*	1/2	t or [u,v] data for <i>Geometry</i> Positions	
paramd*	1/2	t or [u,v] for <i>Data</i> Positions	Not for vertex-based discretizations
<i>GeomIn</i> *	3	Sensitivity for the <i>Geometry</i> Input <i>GeomIn</i>	can have [ <i>irow</i> , <i>icol</i> ] in name

\* Note: not valid for 3D Bounds



## Create a DataSet

```
icode = caps_makeDataSet(capsObj vset, const char *dname,
                        enum capsftype ftype, int rank,
                        capsObj *dset, int *nErr, capsErrs **errs)
dset = vset.makeDataSet(dname, dmethod, rank=0)
```

- vset** the VertexSet Object – associated Bound must be *open*
- dname** a pointer to a string containing the name of the DataSet (i.e., *pressure*)
- ftype** the type of data field: (FieldIn, FieldOut, GeomSens, TessSens, User)
- rank** the rank of the data for a User field (e.g., 1 – scalar, 3 – vector), ignored otherwise
- dset** the returned DataSet Object
- nErr** the returned number of errors generated – 0 means no errors
- errs** the returned CAPS error structure – **NULL** with no errors

## Get DataSet Information

```
icode = caps_dataSetInfo(capsObj dset, enum capsftype *ftype,
                        capsObj *link, enum capsdMethod *dmethod)
ftype, link, dmethod = dset.dataSetInfo()
```

- dset** the input DataSet Object
- ftype** the returned type of data field: (FieldIn, FieldOut, BuiltIn, GeomSens, TessSens, User)
- link** the returned linked DataSet Object (for **ftype** of FieldIn) – can be **NULL**
- dmethod** the returned linked DataSet Method (only valid for **ftype** of FieldIn)

## Get Data from a DataSet

```
icode = caps_getData(capsObj dset, int *npt, int *rank, double **data,  
                    char **units, int *nErr, capsErrs **errs)
```

```
data = dset.getData()
```

**dset** the DataSet Object

**npt** the returned number of points in the DataSet

**rank** the returned rank of the data (e.g., 1 – scalar, 3 – vector)

**data** the returned pointer to the data (**rank**\***npts** in length)

**units** the returned pointer to the string declaring the units

**nErr** the returned number of errors generated – 0 means no errors

**errs** the returned CAPS error structure – NULL with no errors

**icode** integer return code

## Establish Linkage between DataSet Objects

```
icode = caps_linkDataSet(capsObj link, enum capsdMethod dmethod,  
                        capsObj trgt, int *nErr, capsErrors **errs)  
trgt.linkDataSet(link, dmethod)
```

- link** linking DataSet Object, must be FieldOut
- dmethod** 0 – Interpolate, 1 – Conserve
- trgt** the target DataSet Object which will get its data from **link**, must be FieldIn or User
- nErr** the returned number of errors generated – 0 means no errors
- errs** the returned CAPS error structure – **NULL** with no errors

## Initialize DataSet for cyclic/incremental startup

```
icode = caps_initDataSet(capsObj dset, int rank, double *startup,  
                        int *nErr, capsErrors **errs)  
dset.initDataSet(startup)
```

- dset** the DataSet Object (Field type must be FieldIn)
- rank** the rank of the data (e.g., 1 – scalar, 3 – vector)
- startup** the pointer to the constant *startup* data (**rank** in length)
- nErr** the returned number of errors generated – 0 means no errors
- errs** the returned CAPS error structure – **NULL** with no errors

Note: invocations of `caps_getData` and `aim_getDataSet` will return this data (and a length of 1) until properly filled.

## Get DataSet Objects by Name

```
icode = caps_getDataSets(capsObj bound, const char *dname, int *nobj,  
                        capsObj **dsets)
```

**dsets** = bound.getDataSets(dname)

**bound** an input CAPS Bound Object

**dname** a pointer to a string containing the name of the DataSet

**nobj** the returned number of Objects with the name

**dsets** a returned pointer to the list of DataSet Objects (*freeable*)

## Put User Data into a DataSet

```
icode = caps_setData(capsObj dset, int nverts, int rank, double *data,  
                   const char *units, int *nErr, capsErrs **errs)
```

**dset.setData(data)**

**dset** the DataSet Object

**nverts** the number of points in data – must match declared **npts**

**rank** the rank of the data – must match declared **rank** (e.g., 1 – scalar, 3 – vector)

**data** a pointer to the data (**rank\*nverts** in length)

**units** the pointer to the string declaring the units

**nErr** the returned number of errors generated – 0 means no errors

**errs** the returned CAPS error structure – **NULL** with no errors

## Get Triangulations for a 2D VertexSet

```
icode = caps_getTriangles(capsObj vst, int *nGtris, int **Gtris,
                          int *nGsegs, int **Gsegs, int *nDtris,
                          int **Dtris, int *nDsegs, int **Dsegs)
Gtris, Gsegs, Dtris, Dsegs = vst.getTriangles()
```

**vst** the input CAPS Connected VertexSet Object

**nGtris** the returned number of *Geometry*-based Triangles

**Gtris** the returned pointer to a list of indices (bias 1) referencing *Geometry*-based points (3\***nGtris** in length) – *freeable*

**nGsegs** the returned number of *Geometry*-based element mesh segments

**Gsegs** the returned pointer to a list of indices (bias 1) referencing *Geometry*-based points (2\***nGsegs** in length) – *freeable*

**nDtris** the returned number of *Data*-based Triangles (0 if discretization is vertex based)

**Dtris** the returned pointer to a list of indices (bias 1) referencing *Data*-based points (3\***nDtris** in length) – *freeable*

**nDsegs** the returned number of *data*-based element mesh segments

**Dsegs** the returned pointer to a list of indices (bias 1) referencing *data*-based points (2\***nDsegs** in length) – *freeable*

**icode** integer return code

CAPS_SUCCESS	0	CAPS_SHAPEERR	-322
CAPS_BADRANK	-301	CAPS_LINKERR	-323
CAPS_BADDSETNAME	-302	CAPS_MISMATCH	-324
CAPS_NOTFOUND	-303	CAPS_NOTPROBLEM	-325
CAPS_BADINDEX	-304	CAPS_RANGEERR	-326
CAPS_NOTCHANGED	-305	CAPS_DIRTY	-327
CAPS_BADTYPE	-306	CAPS_HIERARCHERR	-328
CAPS_NULLVALUE	-307	CAPS_STATEERR	-329
CAPS_NULLNAME	-308	CAPS_SOURCEERR	-330
CAPS_NULLOBJ	-309	CAPS_EXISTS	-331
CAPS_BADOBJECT	-310	CAPS_IOERR	-332
CAPS_BADVALUE	-311	CAPS_DIRERR	-333
CAPS_PARAMBNDERR	-312	CAPS_NOTIMPLEMENT	-334
CAPS_NOTCONNECT	-313	CAPS_EXECERR	-335
CAPS_NOTPARMTRIC	-314	CAPS_CLEAN	-336
CAPS_READONLYERR	-315	CAPS_BADINTENT	-337
CAPS_FIXEDLEN	-316	CAPS_NOTNEEDED	-339
CAPS_BADNAME	-317	CAPS_NOSENSITVITY	-340
CAPS_BADMETHOD	-318	CAPS_NOBODIES	-341
CAPS_CIRCULARLINK	-319	CAPS_JOURNAL	-342
CAPS_UNITERR	-320	CAPS_JOURNALERR	-343
CAPS_NULLBLIND	-321	CAPS_FILELINKERR	-344

## The Population of the VertexSets

Bounds needed to be fully populated (i.e., the VertexSets need to be filled for all analyses) before they can be used. This is due to the requirement to have all points available to ensure that there is a single UV space (either by construction or by re-parameterization). As a result, the meshing information for an AIM maybe required prior to calling the `aimPreAnalysis`.

The VertexSets are filled with calls the AIM to fill the `aimDiscr` structure (basically the VertexSet), which means the meshing information must be available via a link or generated in `aimDiscr`.

NOTE: An analysis AIM that supports `aimDiscr` and also generates meshes “on the fly” must be able to generate meshes and call `aim_newTess` from either `aimDiscr` or `aimPreAnalysis` (whenever and wherever the mesh gets generated).

## Fluid/Structure Interaction Pseudocode

```
caps_makeAnalysis egadsTess aim -> msobj
caps_makeAnalysis TetGen aim -> mfobj
caps_makeAnalysis fluids aim -> fobj
caps_makeAnalysis structures -> sobj
caps_makeBound "srf" -> bobj
caps_makeVertexSet(bobj, fobj) -> vfobj
caps_makeVertexSet(bobj, sobj) -> vsobj
caps_makeDataSet(vfobj, "Pressure", FieldOut) -> dpfobj
caps_makeDataSet(vsobj, "Pressure", FieldIn ) -> dpsobj
caps_makeDataSet(vsobj, "Displace", FieldOut) -> ddsobj
caps_makeDataSet(vfobj, "Displace", FieldIn ) -> ddfobj
caps_linkDataSet(dpfobj, Conserve, dpsobj)
caps_linkDataSet(ddsobj, Conserve, ddfobj)
caps_initDataSet(ddfobj, 3, zeros)          /* Note #1 */
caps_closeBound(bobj)

caps_execTue(msobj)                        /* generate structures mesh */
caps_execute(mfobj)                       /* generate fluids mesh */

for (iter = 0; iter < nIter; iter++) {
    caps_preAnalysis(fobj)
    /* execute fluids analysis */
    caps_postAnalysis(fobj)

    caps_preAnalysis(sobj)
    /* execute structures analysis */
    caps_postAnalysis(sobj)
}
```



## Pseudocode Notes

The fluids AIM requires the “Displace” values during its “pre” phase, just as the structural analysis AIM requires “Pressure” (i.e., loads) during its “pre” phase to fill in all the inputs.

- 1 `caps_initDataSet` gets called to set the first displacement data to zeros, in that no structural analysis will have been run at start, but is needed by the fluids.
- 2 The lines in red and will mark Analysis *dirty* when the DataSet is filled.

caps_analysisInfo	58	caps_getInput	55	caps_phaseState	17
caps_attrByIndex	50	caps_getLimits	36	caps_postAnalysis	60
caps_attrByName	50	caps_getOutput	55	caps_preAnalysis	59
caps_bodyByIndex	27	caps_getRootPath	21	caps_queryAnalysis	52
caps_boundInfo	61	caps_getStepSize	43	caps_readParameters	29
caps_brokenLink	20	caps_getTessels	54	caps_revision	17
caps_childByIndex	25	caps_getTriangles	69	caps_rmLock	21
caps_childByName	26	caps_getValueProps	38	caps_setAttr	51
caps_close	22	caps_getValue	34	caps_setData	68
caps_closeBound	63	caps_getValueSize	36	caps_setLimits	37
caps_convertValue	40	caps_hasDeriv	44	caps_setStepSize	43
caps_convert	46	caps_info	23	caps_setValueProps	39
caps_dataSetInfo	65	caps_initDataSet	67	caps_setValue	35
caps_deleteAttr	51	caps_intentPhrase	31	caps_size	24
caps_dirtyAnalysis	57	caps_linkDataSet	67	caps_system	59
caps_dupAnalysis	57	caps_linkValue	42	caps_transferValues	41
caps_errorInfo	28	caps_makeAnalysis	56	caps_unitCompare	49
caps_externSignal	21	caps_makeBound	61	caps_unitConvertible	49
caps_execute	53	caps_makeDataSet	65	caps_unitDivide	47
caps_fillUnVertexSet	63	caps_makeValue	33	caps_unitMultiply	46
caps_freeError	28	caps_makeVertexSet	62	caps_unitInvert	47
caps_freeValue	40	caps_markForDelete	24	caps_unitOffset	48
caps_getBodies	54	caps_open	19	caps_unitParse	48
caps_getData	66	caps_outLevel	27	caps_unitRaise	47
caps_getDataSets	68	caps_outputVertexSet	63	caps_vertexSetInfo	62
caps_getDeriv	45	caps_ownerInfo	32	caps_writeGeometry	30
caps_getHistory	31	caps_phaseNewCSM	18	caps_writeParameters	29

## Python API get/set Limits

The CAPS C API for `caps_getLimits` and `caps_setLimits` does not provide the size of the limits array, and instead C API developers are supposed to combine these calls with `caps_getValueSize` if the size of the Value Object is unknown. However, **Python** lists require the size of the array. Hence, the CAPS Python API for `caps_getLimits/caps_setLimits` requires two C API calls which are journaled. Since this breaks the one-to-one nature of the bindings described here, the following functions are used instead. Implementation of these functions are akin to the C functions that follow.

## Get Valid Value Range

```
limits = val.getLimitsSize()
```

**val** the Value Object

**limits** a returned list containing the valid range

## Set Valid Value Range

```
val.setLimitsSize(limits)
```

**val** the Value Object

**limits** the list of limits

```
int
caps_setLimitsSize(capsObj object, enum capsvType vtype, int nrow, int ncol,
                  void *limits, const char *units, int *nErr, capsErrs **errs)
{
    int status, valnrow, valncol;

    status = caps_getValueSize(object, &valnrow, &valncol);
    if (status != CAPS_SUCCESS) return status;

    /* check the shape of the limits array to make sure it matches the value */
    if (limits != NULL) {
        if ((nrow == 1) && (ncol == 1)) {
            /* A single value can be assigned to all entries */
            vtype = abs(vtype);
        } else if (valnrow == 1 || valncol == 1) {
            if ((nrow != 1) && (ncol != 1) &&
                (valnrow*valncol > 1)) return CAPS_SHAPEERR;
            if (nrow*ncol != valnrow*valncol) return CAPS_SHAPEERR;
            vtype = -abs(vtype);
        } else {
            if (nrow != valnrow) return CAPS_SHAPEERR;
            if (ncol != valncol) return CAPS_SHAPEERR;
            vtype = -abs(vtype);
        }
    }

    return caps_setLimits(object, vtype, limits, units, nErr, errs);
}
```

```
int
caps_getLimitsSize(const capsObj object, enum capsvType *vtype, int *nrow,
                  int *ncol, const void **limits, const char **units)
{
    int status;

    if (ncol == NULL) return CAPS_NULLVALUE;
    if (nrow == NULL) return CAPS_NULLVALUE;

    status = caps_getLimits(object, vtype, limits, units);
    if (status != CAPS_SUCCESS) return status;

    if (*vtype > 0) {
        *nrow = 1;
        *ncol = 1;
    } else {
        status = caps_getValueSize(object, nrow, ncol);
        if (status != CAPS_SUCCESS) return status;
        *vtype = -*vtype;
    }

    return status;
}
```

## Python API get/set StepSize

The CAPS C API for `caps_getStepSize` and `caps_setStepSize` does not provide the size of the sizes array, and instead C API developers are supposed to combine these calls with `caps_getValueSize` if the size of the Value Object is unknown. However, **Python** lists require the size of the array. Since this breaks the one-to-one nature of the bindings described here, the following functions are used instead. Implementation of these functions are akin to the C functions that follow.

## Get the Finite Difference Step Size

```
sizes = val.getStepSizeSize()
```

**val** the input CAPS Value Object (*GeometryIn*/DESPMTR types only)

**sizes** the FD step sizes for each Value member (**nrow** of **ncol** tuples)  
a zero indicates use analytic derivatives; can be **None** – set all to zero

## Set the Finite Difference Step Size

```
val.setStepSizeSize(sizes)
```

**val** the input CAPS Value Object (*GeometryIn*/DESPMTR types only)

**sizes** the returned FD step sizes for each Value member (**nrow** of **ncol** tuples)  
a zero indicates use analytic derivatives; can be **None** – all set to zero

```
int
caps_setStepSizeSize(capsObj object, int nrow, int ncol, double *sizes)
{
    int status, valnrow, valncol;
    /* check the shape of the sizes array to make sure it matches the value */
    if (sizes != NULL) {
        status = caps_getValueSize(object, &valnrow, &valncol);
        if (status != CAPS_SUCCESS) return status;
        if (nrow != valnrow) return CAPS_SHAPEERR;
        if (ncol != valncol) return CAPS_SHAPEERR;
    }
    return caps_setStepSize(object, sizes);
}

int
caps_getStepSizeSize(const capsObj object, int *nrow, int *ncol, const double **sizes)
{
    int status;
    if (ncol == NULL) return CAPS_NULLVALUE;
    if (nrow == NULL) return CAPS_NULLVALUE;
    status = caps_getStepSize(object, sizes);
    if (status != CAPS_SUCCESS) return status;
    if (*sizes == NULL) {
        *nrow = *ncol = 0;
    } else {
        status = caps_getValueSize(object, nrow, ncol);
        if (status != CAPS_SUCCESS) return status;
    }
    return status;
}
```

## Python API Units

This **Python** API uses the exposed CAPS unit manipulation functions to be consistent with internal use of units in CAPS. Similar to the Pint<sup>1</sup> Python package, this API defines these classes:

`caps.Unit`

`caps.Quantity`

where `caps.Unit` defines a unit which can be manipulated with standard operator, and `caps.Quantity` represents a value with units. This API is designed to work with these classes as the C API uses the optional **units** string. The best way to extract a value from a `caps.Quantity` is to divide it out by its units.

### Unit Manipulation

```
kg = caps.Unit("kg")  
m  = caps.Unit("m")  
s  = caps.Unit("s")
```

```
Newton = kg*m/s**2
```

### Value from Quantity

```
m = caps.Unit("m")  
ft = caps.Unit("ft")  
  
q = 10 * m      # Make a Quantity  
  
assert(10 == q/m)  
assert(10 == q.value())  
assert(q.convert(ft).value() == q/ft)
```

<sup>1</sup><https://pint.readthedocs.io>