

# Engineering Sketch Pad (ESP)



## Training Session 4 CSM Language (1)

**John F. Dannenhoffer, III**

[jfdannen@syr.edu](mailto:jfdannen@syr.edu)  
Syracuse University

**Bob Haimes**

[haimes@mit.edu](mailto:haimes@mit.edu)  
Massachusetts Institute of Technology  
updated for v1.18

- Format of `.csm` file
- Special characters
- Numbers
- Parameters
  - Types
  - Names
  - Dimensions
  - Lower and Upper Bounds
- Expressions
  - Numeric
  - String
- Reading Help File
- CSM File Editor

- All configuration information is contained in `.csm` (or possibly `.udc`) files
  - `.csm` files are plain ASCII text that are readable by humans
  - because they are ASCII files, they can either be written directly by humans (using any text editor) or by other programs
- When you build a configuration using the ESP user interface, you are actually building a `.csm` file
- Using the interface can be effective for beginning users who are building small models
- Once a user gets experience with ESP, most of the models are created by “typing” a `.csm` directly

- The .csm file contains a series of statements.
- If a line contains a hash (#), all characters starting at the hash are ignored.
- If a line contains a backslash (\), all characters starting at the backslash are ignored and the next line is appended; spaces at the beginning of the next line are treated normally.
- All statements begin with a keyword (described below) and must contain at least the indicated number of arguments.
- The keywords may either be all lowercase or all UPPERCASE.
- Any CSM statement can be used in a .csm file except the INTERFACE statement.

- Blocks of statements must be properly nested. The Blocks are bounded by
  - PATBEG/PATEND
  - IFTHEN/ELSEIF/ELSE/ENDIF
  - SKBEG/SKEND
  - SOLBEG/SOLEND
  - CATBEG/CATEND
- Extra arguments in a statement are discarded. If one wants to add a comment, it is recommended to begin it with a hash (#) in case optional arguments are added in future releases.
- Any statements after an **END** statement are ignored.
  - hint: if debugging, consider **THROWing** an error instead to avoid unclosed Blocks
- All arguments must not contain any spaces or must be enclosed in a pair of double quotes (for example, "a + b").

- Parameters are evaluated in the order that they appear in the file, using MATLAB-like syntax (see 'Expression rules' below).
- During the build process, **OpenCSM** maintains a last-in-first-out (LIFO) "Stack" that can contain Bodys and Sketches.
- The .csm statements are executed in a stack-like way, taking their inputs from the Stack and depositing their results onto the Stack.
- The default name for each Branch is **Brch\_xxxxxx**, where **xxxxxx** is a unique sequence number.

#	introduces comment
"	ignore spaces until following "
\	ignore this and following characters and concatenate next line
<space>	separates arguments in .csm file (except between " and ")
0-9	digits used in numbers and in names
A-Z a-z	letters used in names
_ : @	characters used in names (see rule for names)
.	decimal separator (used in numbers), introduces dot-suffixes (in names)
,	separates function arguments and row/column in subscripts
;	multi-value item separator

( )	groups expressions and function arguments
[ ]	specifies subscripts in form [row,column] or [index]
{ } < >	characters used in strings
+ - * / ^	arithmetic operators
\$	as first character, introduces a string that is terminated by end-of-line or un-escaped plus, comma, or open-bracket
@	as first character, introduces @-parameters
,	used to escape comma, plus, or open-bracket within strings
!	if first character of implicit string, ignore \$! and treat as an expression
	cannot be used (reserved for OpenCSM internals)
~	cannot be used (reserved for OpenCSM internals)
&	cannot be used (reserved for OpenCSM internals)



- Start with a digit or decimal (.)
- Followed by zero or more digits and/or decimals (.)
- There can be at most one decimal in a number
- Optionally followed by an e, e+, e-, E, E+, or E-
- If there is an e or E, it must be followed by one or more digits
- If numbers are in a list, the elements are separated by a semicolon (;)

- Design Parameter

- values are declared in a DESPMTR statement
  - in .csm file or
  - in top-level include-type .udc file
- must contain one or more numbers (no strings)
- if multi-valued, must be first DIMENSIONED
- can contain lower- and upper-bounds, specified in LBOUND and UBOUND statements
- values are only visible at the top-level
- values can be changed by a call to `ocsmSetValu` or `ocsmSetValuD` (after `ocsmLoad` and before `ocsmBuild`)
- values can be read by call to `ocsmGetValu`
- sensitivities can be computed by a call to `ocsmSetVel` or `ocsmSetVelD`

- Configuration Parameter
  - values are declared in a `CFGPMTR` statement
    - in `.csm` file or
    - in top-level include-type `.udc` file
  - must contain one or more numbers (no strings)
  - if multi-valued, must be first `DIMENSIONED`
  - can contain lower- and upper-bounds, specified in `LBOUND` and `UBOUND` statements
  - values are only visible at the top-level
  - values can be changed by a call to `ocsmSetValu` or `ocsmSetValuD` (after `ocsmLoad` and before `ocsmBuild`)
  - values can be read by call to `ocsmGetValu`
  - sensitivities CANNOT be computed for Configuration Parameters

- Constant Parameter
  - values are declared in a `CONPMTR` statement
    - in `.csm` file
    - in top-level include-type `.udc` file
  - must contain only one number (no strings)
  - values are visible from any `.csm` or `.udc` file
  - values CANNOT be changed by a call to `ocsmSetValu` or `ocsmSetValuD`
  - sensitivities CANNOT be computed for Constant Parameters

- Local Variables
  - is created by a `SET`, `PATBEG` or `GETATTR` statement
  - can contain one or more numbers or a character string
  - if multi-valued, must first be `DIMENSIONED`
  - can be an `@`-parameter (described below)
  - are only usable in `.csm` or `.udc` file in which it was defined (unless the `.udc` file has `INTERFACE . ALL` in its preamble)
- Output Parameters
  - declared in a `OUTPMTR` statement
  - refers to any local variable whose value is available outside ESP (such as to `CAPS`)

	DESPMTR	CFGPMTR	CONPMTR	OUTPMTR	LOCALVAR
Can be vector or array of numbers	Y	Y	N	Y	Y
Can have a string value	N	N	N	Y	Y
Can be restricted by LBOUND or UBOUND	Y	Y	N	N	N
Scope	T	T	G	L	L
Defined during <code>ocsmLoad</code> or <code>ocsmLoadDict</code>	Y	Y	Y	N	N
Can be set via <code>ocsmSetValu(D)</code>	Y	Y	N	N	N
Defined and set during <code>ocsmBuild</code>	N	N	N	Y	Y
Can be read via <code>ocsmGetValu(S)</code>	Y	Y	Y	Y	Y*
Can find associated sensitivity	Y	N	N	N	N
Y*=Parameter index may be different for different builds scopes: T=top-level, G=global, L=local					

- General form is: `DIMENSION $pmtrName nrow ncol`
- Can only be applied once to a `DESPMTR` or `CFGPMTR`
- Cannot be applied to a `CONPMTR`
- When applied to an `OUTPMTR` or `LOCALVAR`
  - if the new size has fewer elements than the old size
    - the old values are copied to fill the new size
    - extra old elements are lost
  - if the new size has more elements than the old size
    - the old values are all copied
    - the last old value is copied into all the remaining new locations

- Start with a letter, colon (:), or at-sign (@)
- Contains letters, digits, at-signs (@), underscores (\_), and colons (:)
- Contains fewer than 32 characters
- Names that start with an at-sign cannot be set by a CONPMTR, DESPMTR, CFGPMTR, SET, or PATBEG statement
- When listed in ESP, are sub-grouped based upon the colons (:)



- If a name has a dot-suffix, a property of the parameter (and not its value) is returned

`x.nrow`    number of rows in `x`

`x.ncol`    number of columns in `x`

`x.size`    number of elements or characters in `x`

`x.sum`    sum of elements in `x`

`x.norm`    RMS norm of elements in `x`

`x.min`    minimum value in `x`

`x.max`    maximum value in `x`

- Example:

```
DIMENSION  myvar  2 3 1
DESPMTR    myvar  "1; 2; 3;\n
               4; 5; 6"
```

- `myvar.nrow` returns 2
- `myvar.sum` returns 21

- Basic format is: `name[irow,icol]` or `name[ielem]`
- Name must follow rules above
- `irow`, `icol`, and `ielem` must be valid expressions
- `irow`, `icol`, and `ielem` start counting at 1
- For 2D arrays, either `name[irow,icol]` or `name[ielem]` be used
- Values are stored across rows (`[1,1]`, `[1,2]`, ..., `[2,1]`, ...)

- Every time a Body gets created, or after a **SELECT** statement, readable local variables are set

	body	face	edge	node	<- last SELECT
@seltype	-1	2	1	0	selection type (0=node,1=edge,2=face)
@selbody	x	-	-	-	current Body
@sellist	-1	x	x	x	list of Nodes/Edges/Faces
@nbody	x	x	x	x	number of Bodys
@ibody	x	x	x	x	current Body
@nface	x	x	x	x	number of Faces in @ibody
@iface	-1	x	-1	-1	current Face in @ibody
@nedge	x	x	x	x	number of Edges in @ibody
@iedge	-1	-1	x	-1	current Edge in @ibody
@nnode	x	x	x	x	number of Nodes in @ibody
@inode	-1	-1	-1	x	current Node in @ibody
@igroup	x	x	x	x	group of current Body
@itype	x	x	x	x	0=NodeBody, 1=WireBody, 2=SheetBody, 3=SolidBody
@nbors	-1	x	-	x	number of incident Edges
@nbors	-1	-	x	-	number of incident Faces

@ibody1	-1	x	x	-1	first element of 'Body' Attribute in @ibody
@ibody2	-1	x	x	-1	second element of 'Body' Attribute in @ibody
@xmin	x	x	*	x	x-min of bounding box or x at beg of edge
@ymin	x	x	*	x	y-min of bounding box or y at beg of edge
@zmin	x	x	*	x	z-min of bounding box or z at beg of edge
@xmax	x	x	*	x	x-max of bounding box or x at end of edge
@ymax	x	x	*	x	y-max of bounding box or y at end of edge
@zmax	x	x	*	x	z-max of bounding box or z at end of edge
@length	0	0	x	0	length of edge
@area	x	x	0	0	area of face or surface area of body
@volume	x	0	0	0	volume of body (if a solid)
@xcg	x	x	x	x	location of center of gravity
@ycg	x	x	x	x	
@zcg	x	x	x	x	

@Ixx	x	x	x	0	centroidal moment of inertia
@Ixy	x	x	x	0	
@Ixz	x	x	x	0	
@Iyx	x	x	x	0	
@Iyy	x	x	x	0	
@Iyz	x	x	x	0	
@Izx	x	x	x	0	
@Izy	x	x	x	0	
@Izz	x	x	x	0	
@signal	x	x	x	x	current signal code
@nwarn	x	x	x	x	number of warnings
@edata					only set up by EVALUATE statement
@stack					Bodys in stack: 0=mark, -1=none
@version					version number

in above table:

- x -> value is set
- -> value is unchanged
- \* -> special value is set (if edge)
- 0 -> value is set to 0
- 1 -> value is set to -1

- Valid operators (in order of precedence):
  - ( ) parentheses, inner-most evaluated first
  - func(a,b) function arguments, then function itself
  - $\wedge$  exponentiation (evaluated left to right)
  - \* / multiply and divide (evaluated left to right)
  - + - add and subtract (evaluated left to right)

- Contains the sequence of characters starting after a dollar-sign(\$) and ending with a space, plus-sign (+), comma (,), or closed-parenthesis ())
- If escaped with an apostrophe ('), can contain a plus-sign ('+), comma (',) or closed-parenthesis ('))
  - for example:

```
$thisStringContainsAComma(',')  
returns thisStringContainsAComma(,)
```

- Can never contain a space
- Are parsed left-to-right, as is any expression
  - for example:

```
SET    one 1  
SET    mystr $thereIsA+one+$inThisString  
returns (in mystr) thereIsA1inThisString
```



<code>pi(x)</code>	$3.14159... * x$
<code>min(x,y)</code>	minimum of $x$ and $y$
<code>max(x,y)</code>	maximum of $x$ and $y$
<code>sqrt(x)</code>	square root of $x$
<code>abs(x)</code>	absolute value of $x$
<code>int(x)</code>	integer part of $x$ ( $3.5 \rightarrow 3$ , $-3.5 \rightarrow -3$ ) produces derivative=0
<code>nint(x)</code>	nearest integer to $x$ produces derivative=0
<code>ceil(x)</code>	smallest integer not less than $x$ produces derivative=0
<code>floor(x)</code>	largest integer not greater than $x$ produces derivative=0

<code>mod(a,b)</code>	modulus( $a/b$ ), with same sign as $a$ and $b \geq 0$
<code>sign(test)</code>	returns -1, 0, or +1
<code>exp(x)</code>	produces derivative=0
<code>log(x)</code>	exponential of $x$
<code>log10(x)</code>	natural logarithm of $x$
	common logarithm of $x$

<code>sin(x)</code>	sine of $x$	(in radians)
<code>sind(x)</code>	sine of $x$	(in degrees)
<code>asin(x)</code>	arc-sine of $x$	(in radians)
<code>asind(x)</code>	arc-sine of $x$	(in degrees)
<code>cos(x)</code>	cosine of $x$	(in radians)
<code>cosd(x)</code>	cosine of $x$	(in degrees)
<code>acos(x)</code>	arc-cosine of $x$	(in radians)
<code>acosd(x)</code>	arc-cosine of $x$	(in degrees)

<code>tan(x)</code>	tangent of $x$	(in radians)
<code>tand(x)</code>	tangent of $x$	(in degrees)
<code>atan(x)</code>	arc-tangent of $x$	(in radians)
<code>atand(x)</code>	arc-tangent of $x$	(in degrees)
<code>atan2(y,x)</code>	arc-tangent of $y/x$	(in radians)
<code>atan2d(y,x)</code>	arc-tangent of $y/x$	(in degrees)
<code>hypot(x,y)</code>	hypotenuse: $\sqrt{x^2 + y^2}$	
<code>hypot3(x,y,z)</code>	hypotenuse: $\sqrt{x^2 + y^2 + z^2}$	

Xcent(xa, ya, dab, xb, yb)

X-center of circular arc  
produces derivative=0

Ycent(xa, ya, dab, xb, yb)

Y-center of circular arc  
produces derivative=0

Xmidl(xa, ya, dab, xb, yb)

X-point at midpoint of circular arc  
produces derivative=0

Ymidl(xa, ya, dab, xb, yb)

Y-point at midpoint of circular arc  
produces derivative=0

seglen(xa, ya, dab, xb, yb)

length of segment  
produces derivative=0

<code>incline(xa,ya,dab,xb,yb)</code>	inclination of chord (in degrees) produces derivative=0
<code>radius(xa,ya,dab,xb,yb)</code>	radius of curvature (or 0 for linseg) produces derivative=0
<code>sweep(xa,ya,dab,xb,yb)</code>	sweep angle of circular arc (in degree) produces derivative=0
<code>turnang(xa,ya,dab,...           xb,yb,dbc,xc,yc)</code>	turning angle at b (in degrees) produces derivative=0
<code>dip(xa,ya,xb,yb,rad)</code>	acute dip between arc and chord produces derivative=0
<code>smallang(x)</code>	ensures $-180 \leq x \leq 180$

`val2str(num,digits)`

convert `num` to a string

`str2val(string)`

convert `string` to a number

`findstr(str1,str2)`

finds location of `str2` in `str1`  
(bias-1) or 0 if not found

`slice(str,ibeg,iend)`

substring of `str` from `ibeg`  
to `iend` (bias-1)

`path($pwd)`

returns present working directory

`path($csm)`

returns directory of current `.csm` file

`path($root)`

returns `$ESP_ROOT`

`path($file)`

returns name of `.csm` file

<code>ifzero(test,ifTrue,ifFalse)</code>	if <code>test = 0</code> , return <code>ifTrue</code> , else return <code>ifFalse</code>
<code>ifpos(test,ifTrue,ifFalse)</code>	if <code>test &gt; 0</code> , return <code>ifTrue</code> , else return <code>ifFalse</code>
<code>ifneg(test,ifTrue,ifFalse)</code>	if <code>test &lt; 0</code> , return <code>ifTrue</code> , else return <code>ifFalse</code>
<code>ifnan(test,ifTrue,ifFalse)</code>	if <code>test</code> is NaN, return <code>ifTrue</code> , else return <code>ifFalse</code>



STORE        \$name index=0 keep=0

use:        stores Group on top of Stack

pops:       any

pushes:     -

notes:      Sketch may not be open

             Solver may not be open

             \$name is used directly (without evaluation)

             previous Group in name/index is overwritten

             if \$name=.    then Body is popped off stack

                                 but not actually stored

             if \$name=.. then pop Bodys off stack back

                                 to the Mark

             if \$name=... then the stack is cleared

             if keep==1, the Group is not popped off stack

             cannot be followed by ATTRIBUTE or CSYSTEM

             signals that may be thrown/caught:

             \$insufficient\_bodys\_on\_stack

## Reading Help File (2)

- If argument starts with dollar-sign (\$), then the argument is assumed to be string, and the user does not need to prepend the argument with a dollar-sign (\$)
  - if an expression is given that should be evaluated (to a string value), prepend the argument with an exclamation point (!), as in:

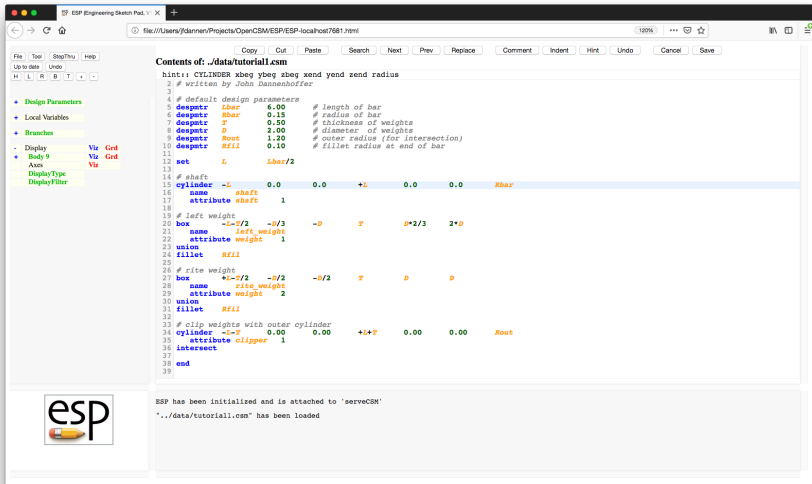
```
SET      i 10
STORE    !$ThisIsBody+i+$.
```

stores the Body in a location named `ThisIsBody10`.

- For arguments that are listed with an equal-sign (=), the value after the equal sign is the default value

# CSM File Editor (1)

- Started via the button **File**→**Edit**:



## CSM File Editor (2)

- Options (on top row) include:
  - **Copy** — copy highlighted text into paste-buffer
  - **Cut** — copy highlighted text into paste-buffer and remove it from the file
  - **Paste** — copy paste-buffer into `.csm` file at the cursor
  - **Search** — search for text (input is on top line)
  - **Next** — search for next occurrence
  - **Prev** — search for previous occurrence
  - **Replace** — replace one text string with another
  - ...

# CSM File Editor (3)

- Options on top row include:
  - ...
  - **Comment** — if first statement in highlighted region is not a comment, block comment the whole region. Otherwise, block un-comment the whole region
  - **Indent** — indent the highlighted region
  - **Hint** — provide a hint (on the top line) for the statement at the cursor
  - **Undo** — un-do the previous edit
  - **Cancel** — leave the editor (and lose your changes)
  - **Save** — save the file to disk. If there is only one file in the session, the configuration is also automatically re-built