Engineering Sketch Pad (ESP)

Training Session 2.3
CSM Language (2)

John F. Dannenhoffer, III
jfdannen@syr.edu
Syracuse University

Bob Haimes
haimes@mit.edu

Marshall Galbraith
galbramc@mit.edu

Massachusetts Institute of Technology

Overview

- Manipulating the Stack
  - GROUP
  - STORE, RESTORE
- Looping
  - PATBEG, PATBREAK, PATEND
- Logic
  - IFTHEN, ELSEIF, ELSE, ENDIF
- Signal Handling
  - THROW, CATBEG, CATEND
- User-defined Components (UDCs)
  - Include-style
  - Function-style

Manipulating the Stack (1)

- 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.
- Bodys can be grouped with the `GROUP` statement
  - all the Bodys back to the Mark (or the beginning of the Stack) are put into a single Group
  - some operations, such as the transformations, `ATTRIBUTE`, `STORE`, and `DUMP` operate on all Bodys in the Group simultaneously

Manipulating the Stack (2)

- The Group on the top of the Stack can be "popped" off the stack with a `STORE` command
  - if the `name` is alpha-numeric, the Group is stored in a named storage location
  - if the `name` is a dot (.), the Group is not stored (just popped off the Stack)
  - if the `name` is two dots (..), all the Groups back to the Mark are popped off the Stack (and not stored)
  - if the `name` is three dots (...), everything is popped off the Stack

Manipulating the Stack (3)

- Groups can be read from a named storage location and "pushed" onto the Stack with the `RESTORE` command
- The `RESTORE` command is considered a primitive, so its Attributes are put on all the Bodys and all their Faces

Patterns

- Patterns are like “for” or “do” loops
  - the Branches between the `PATBEG` and `PATEND` are executed a known number of times
  - at the beginning of each “instance”, the pattern number is incremented (from 1 to the number of copies)
  - one can break out of the pattern early with a `PATBREAK` statement
- Example pattern (indentation optional):

```
PATBEG      i     7
   SET      j     i-1
   BOX      j     0   0   1   1   1
   ROTATEX  j*10  0   0
PATEND
```

If/then (1)

- If/then constructs are used to make a choice within a `.csm` script
  - start with `IFTHEN` statement
  - has zero or more `ELSEIF` statements
  - has zero or one `ELSE` statement
  - has exactly one `ENDIF` statement
- The `IFTHEN` and `ELSEIF` statements have arguments
  - `val1` — an expression
  - `op1` — can be `lt`, `le`, `eq`, `ge`, `gt`, or `ne`
  - `val2` — an expression
  - `op2` — can be `or`, `xor`, or `and` (defaults to `and`)
  - `val3` — an expression (defaults to `0`)
  - `op3` — can be `lt`, `le`, `eq`, `ge`, `gt`, or `ne` (defaults to `eq`)
  - `val4` — an expression (defaults to `0`)

If/then (2)

- Example (indentation optional):

```
IFTHEN     a   eq   4   or   b   ne   2
   BOX     0   0    0   1    1   1
ELSEIF     c   eq   sqrt(9)
   BOX     2   2    2   2    2   2
ELSE
   BOX     3   3    3   3    3   3
ENDIF
```

Throw/catch (1)

- Throw/catch constructs are used to generate and react to signals (errors)
- Signals can be generated by
  - executing a `THROW` command
  - a run-time error encountered elsewhere (see "help" for more info)
- When a signal is generated, all Branches are skipped until a matching `CATBEG` statement is encountered
  - the signal is cancelled
  - processing continues at the statement following the `CATBEG`
- If a `CATBEG` statement is encountered when there is no pending signal (or the pending signal does not match the `CATBEG`)
  - all Branches up to, and including the matching `CATEND` statement, are skipped

Throw/catch (2)

```
 1: BOX      0 0 0 1 1 1

 2: THROW    -99

 3: SPHERE   0 0 0 1

 4: CATBEG   -98
 5:    SPHERE    0 0 0 2
 6: CATEND

 7: SPHERE   0 0 0 3

 8: CATBEG   -99
 9:    BOX         1 0 0 1 1 1
10: CATEND

11: CATBEG   -99
12:    SPHERE   0 0 0 4
13: CATEND

14: END
```

- `BOX` in line 1 is generated
- `SPHERE` in line 3 is skipped (since there is an active signal)
- `CATBEG/CATEND` in lines 4–6 are skipped (since they do not match `-99`)
- `SPHERE` in line 7 is skipped
- `BOX` in line 9 is generated
- `CATBEG/CATEND` in lines 11–13 are skipped (since the signal was cancelled when it was caught in line 8)

Special Note on Programming Blocks

- Programming Blocks are delineated by
  - `PATBEG` and `PATEND`
  - `IFTHEN`, `ELSEIF`, `ELSE`, and `ENDIF`
  - `SOLBEG` and `SOLEND`
  - `CATBEG` and `CATEND`
- Any programming Block can be nested fully within any other programming Block (up to 10 levels deep)

User-defined Components (UDCs)

- A UDC is a series of statements that are contained in a `.udc` file
- The statements in the UDC can be treated in two ways:
  - Include-style
    - statements within the UDC are simply processed as if they were included in the enclosing `.csm` or `.udc` file
    - the `.udc` file must start with an `INTERFACE . ALL` statement
    - Variables and Parameters in the `.udc` file have the same scope as its caller (that is, the UDC shares variables with its caller)
  - Function-style
    - Variables and Parameters in the `.udc` file have local scope (that is, the UDC's variable are private)
    - Variables in the UDC get values via `INTERFACE . IN` statements
    - The UDC can output some of its variables via `INTERFACE . OUT` statements

UDCs Shipped with **ESP**

- `biconvex` — generate a biconvex airfoil
- `boxudc` — similar to the box UDP
- `diamond` — generate a double-diamond airfoil
- `flapz` — cut a (deflected) flap in a Body
- `gen_rot` — general rotation with two fixed points
- `popupz` — pop up a part of the configuration
- `spoilerz` — pop up a spoiler
- `duct` — generate a duct
- `fuselage` — generate a fuselage
- `strut` — generate a strut (between a duct and wing)
- `wing` — generate a wing

Calling a UDC

- UDCs are called with a `UDPRIM` statement
- `$primtype` must start with a slash (`/`), dollar-slash (`$/`), or dollar-dollar-slash (`$$/`)
  - if `/`, then the UDC file is in the current working directory
  - if `$/`, then the UDC file is in the same directory as the `.csm` file
  - if `$$/`, then the UDC file is in `ESP_ROOT/udc`
- The `UDPRIM` statement can be preceded by one or more `UDPARG` statements
- name-value pairs are processed in order (with possible over-writing)

Writing a UDC

- Define the interface
  - input variables (with default values)
  - output variables (with default values)
  - dimensioned variables (which all default to 0)
- Add assertions to ensure valid inputs
- Make sure all "output" variables are assigned values

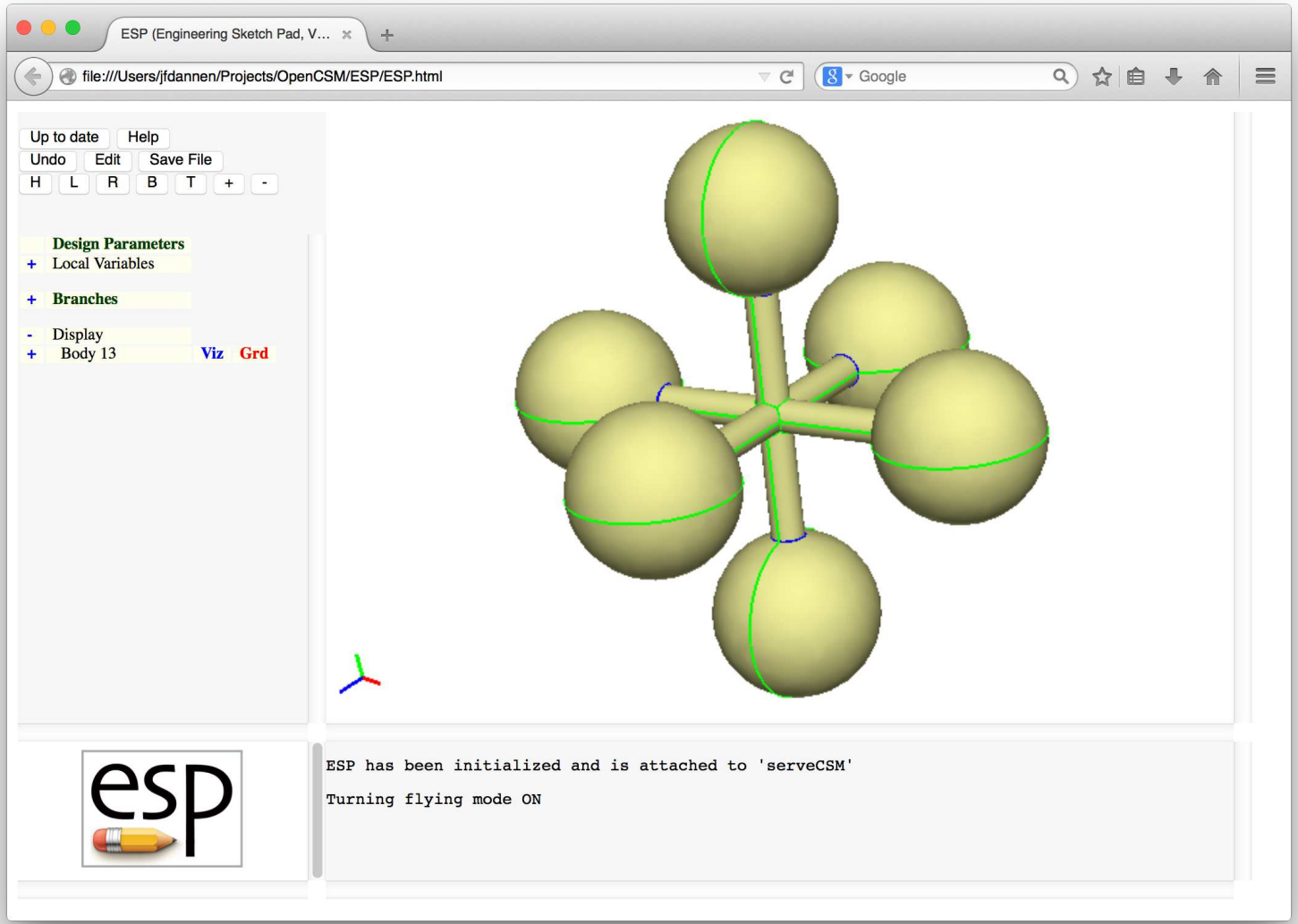
Example UDC — dumbbell.udc

```
# dumbbell

INTERFACE Lbar      in  0    # length of bar
INTERFACE Dbar      in  0    # diameter of bar
INTERFACE Dball     in  0    # diameter of balls
INTERFACE vol       out 0    # volume

ASSERT    ifpos(Lbar,1,0)   1
ASSERT    ifpos(Dbar,1,0)   1
ASSERT    ifpos(Dball,1,0)  1
SET       Lhalf    "Lbar / 2"

CYLINDER  -Lhalf  0  0  +Lhalf  0  0  Dbar
SPHERE    -Lhalf  0  0   Dball
UNION
SPHERE    +Lhalf  0  0   Dball
UNION

SET       vol    @volume

END
```

Example UDC — jack.csm

```
# jack

UDPARG  $/dumbbell  Lbar   5.0
UDPARG  $/dumbbell  Dball  1.0
UDPRIM  $/dumbbell  Dbar   0.2
SET     foo @@vol
STORE   dumbbell 0 1

RESTORE dumbbell
ROTATEY 90  0  0
UNION

RESTORE dumbbell
ROTATEZ 90  0  0
UNION

# show that vol was a local variable in .udc
ASSERT  ifnan(vol,1,0)  1
END
```

Example UDC — Jack



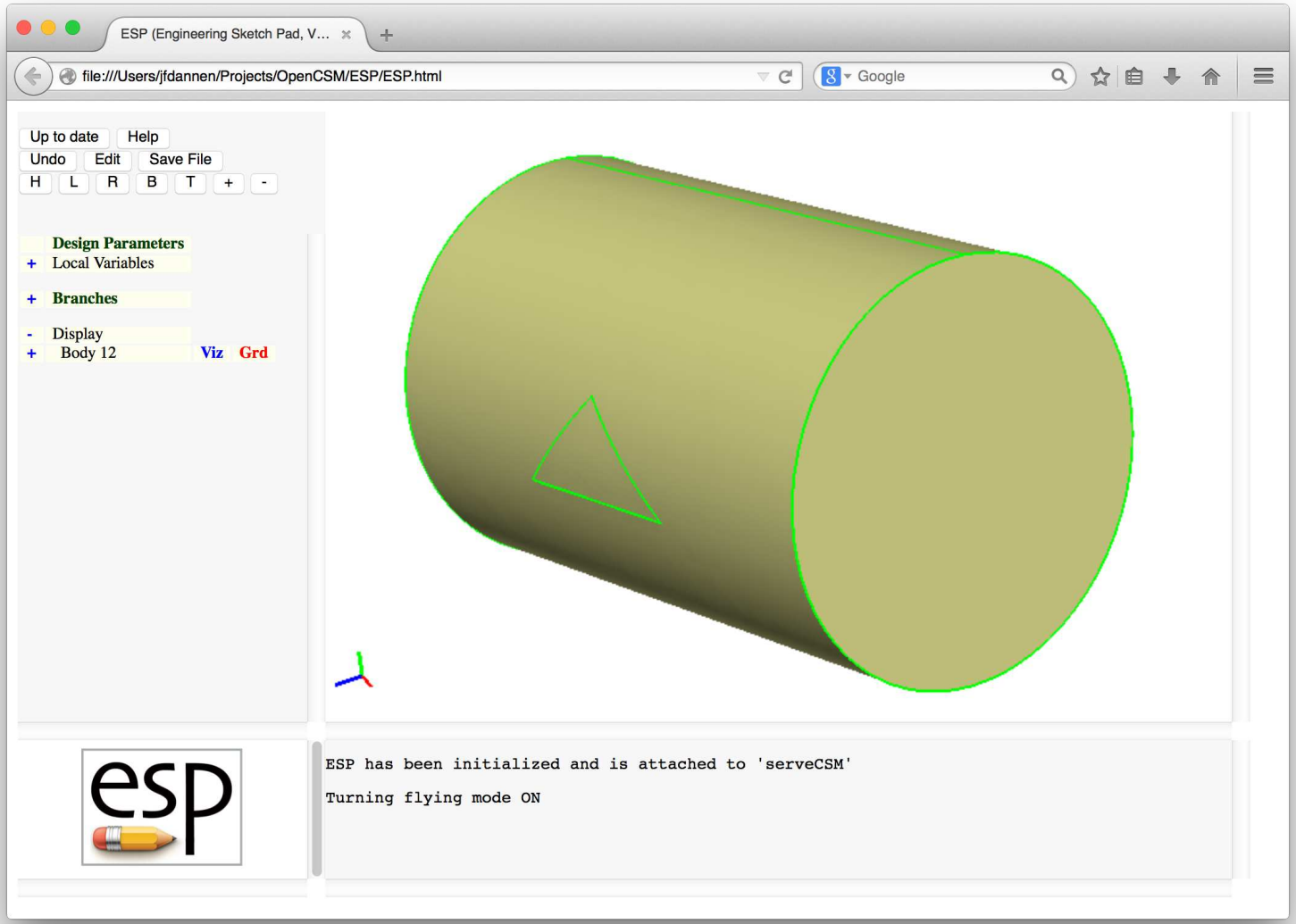
Example UDC — `cutter.udc`

```
# cutter

INTERFACE xx    in  0
INTERFACE yy    in  0
INTERFACE zbeg  in  0
INTERFACE zend  in  0

ASSERT    ifpos(xx.size-2,1,0)  1
ASSERT    ifzero(xx.size-yy.size,1,0)  1

SKBEG          xx[1]    yy[1]    zbeg
   PATBEG i xx.size-1
      LINSEG  xx[i+1]  yy[i+1]  zbeg
   PATEND
   LINSEG      xx[1]    yy[1]    zbeg
SKEND  1

EXTRUDE   0  0  zend-zbeg

END
```

Example UDC — scribeCyl.csm

```
# scribeCyl

DIMENSION xpoints   1  3
DIMENSION ypoints   1  3

SET       xpoints  "-1.;  1.;  .0;"
SET       ypoints  "-.5; -.5; +.5;"

CYLINDER  -3  0  0  +3  0  0  2
ROTATEX   90  0  0

UDPARG    $/cutter  xx    xpoints
UDPARG    $/cutter  yy    ypoints
UDPARG    $/cutter  zbeg  0
UDPRIM    $/cutter  zend  3
SUBTRACT

END
```

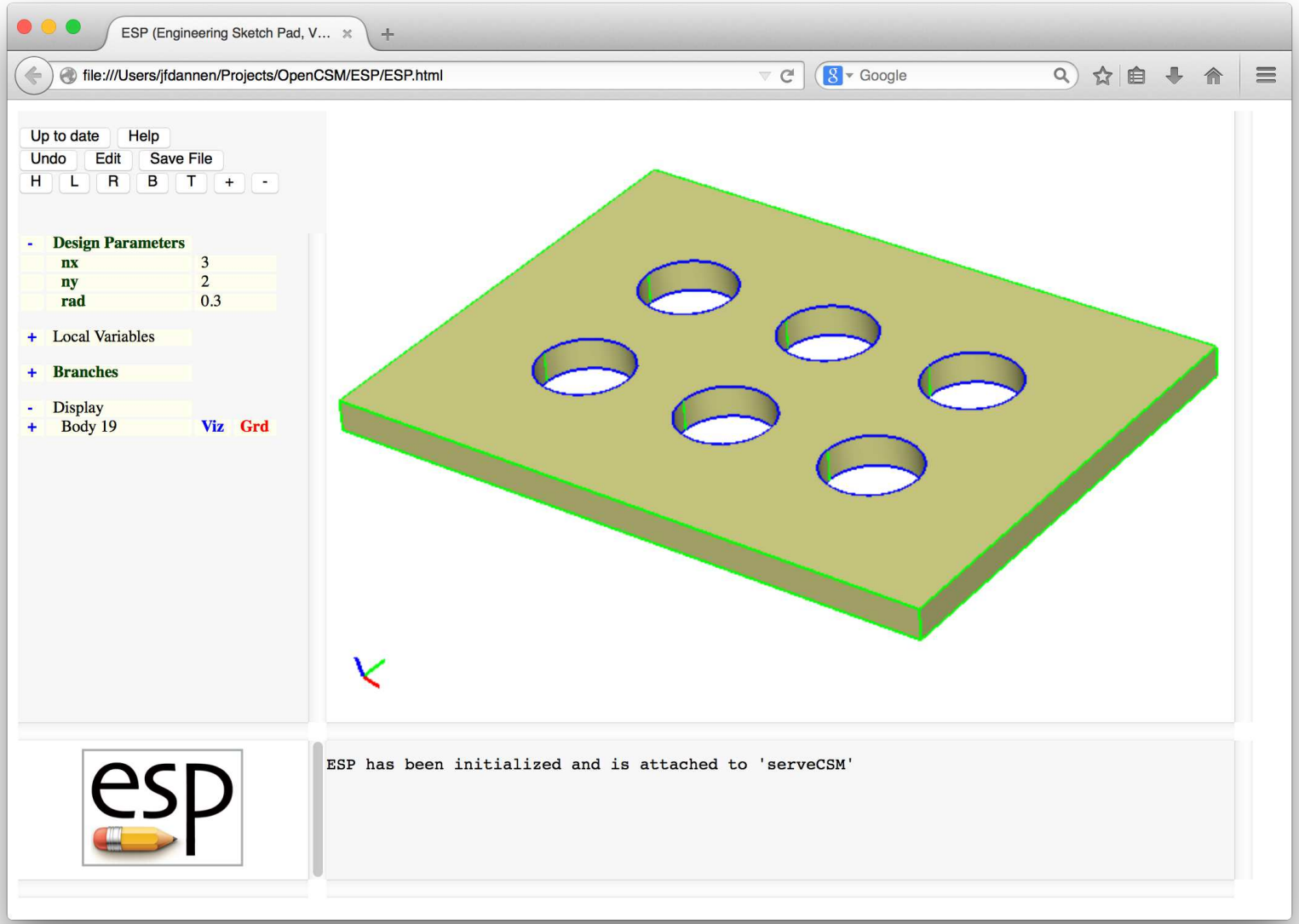
Example UDC — Scribed Cylinder



Hands-on Exercises

- Rectangular plate with holes
- Round plate with holes
- Reflected cone
- Files in `$ESP_ROOT/training/session2.3` will get you started

Rectangular Plate with Holes (1)



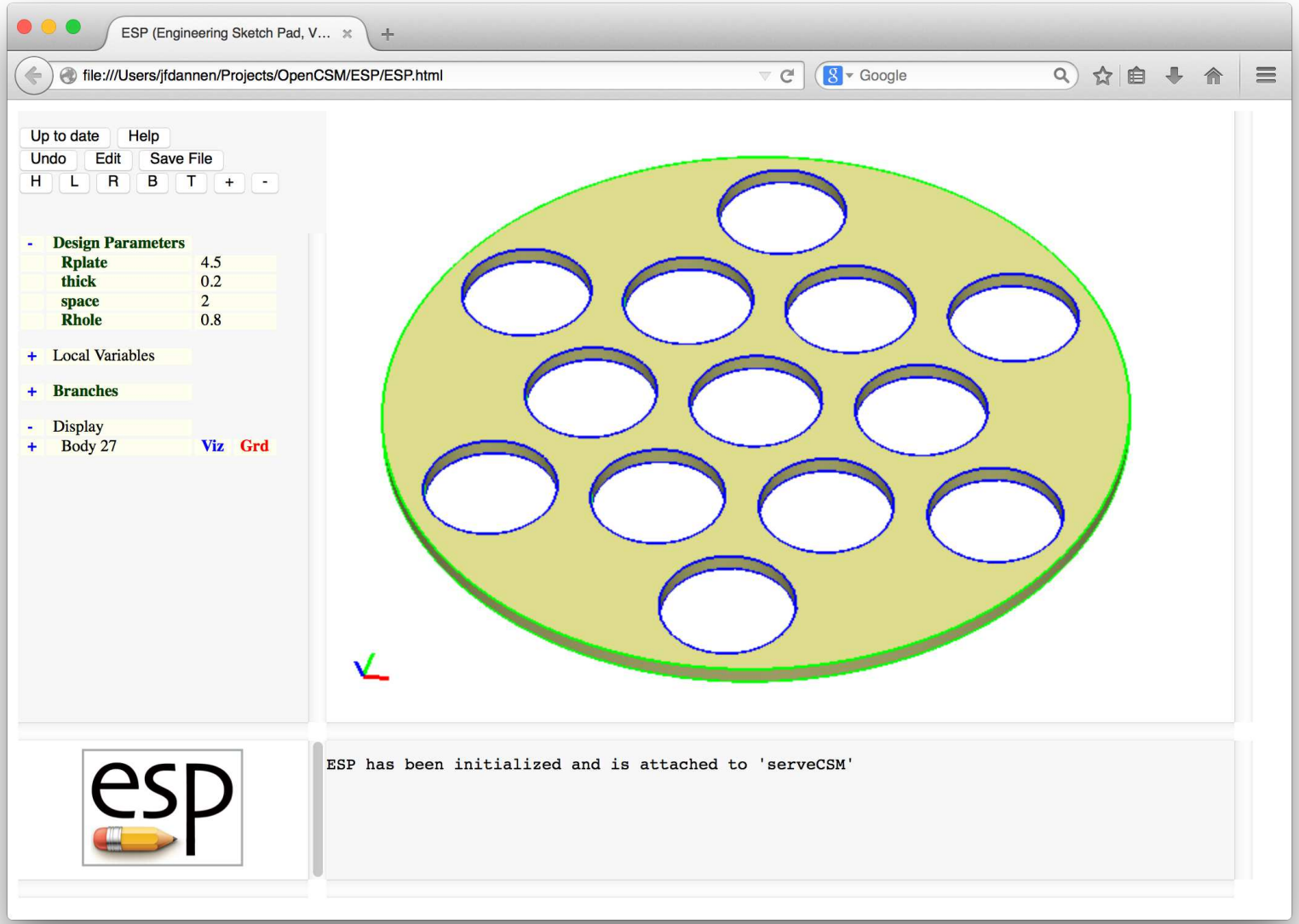
Rectangular Plate with Holes (2)

nx	number of holes in X-direction	3.00
ny	number of holes in Y-direction	2.00
rad	radius of each hole	0.30
	distance between hole centers	1.00

Rectanguler Plate with Holes (3)

- Can you make a single hole in the center of the plate?
- Can you change your solution to have the holes spaced so that they fill the plate?
- What if you make the radius of the hole too big?

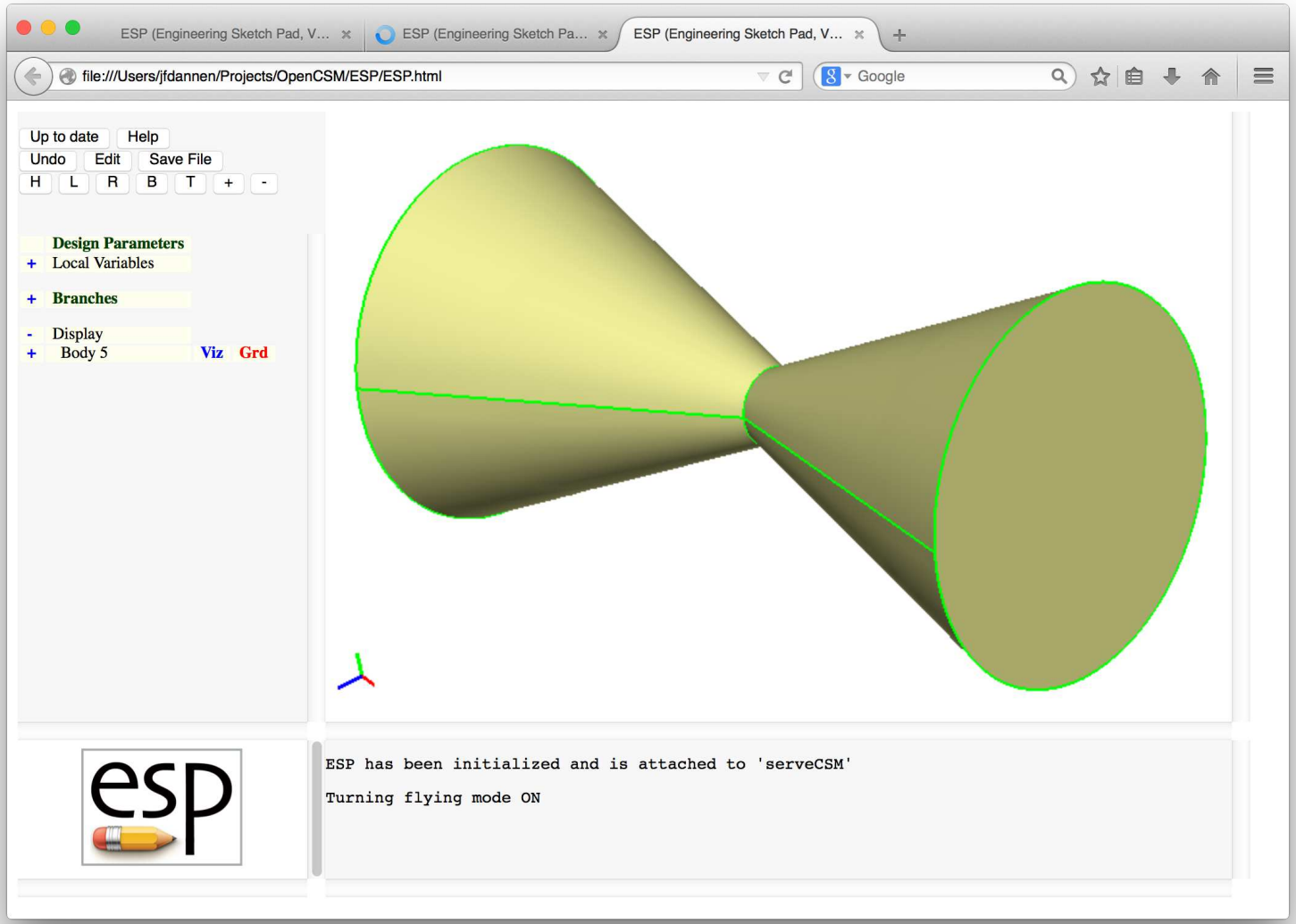
Round Plate with Holes (1)



Round Plate with Holes (2)

Rplate	radius or plate	4.50
thick	thickness of plate	0.20
space	distance between hole centers	2.00
Rhole	radius of holes	0.80
	number of holes selected automatically	

Reflected Cone (1)



Reflected Cone (2)

- Write `mirrorDup.udc` to
  - store a copy of the Body on the top of the stack
  - mirror the Body across a plane whose normal vector and distance from the origin are given
  - union the original and mirrored Bodys
- Apply `mirrorDup.udc` to a cone
  - cone base at $(5, 0, 0)$
  - cone vertex at $(0, 0, 0)$
  - cone diameter is 4
  - reflection across a plane at $x = 1$