AIAA SciTech Forum
8-12 January 2024, Orlando, FL
AIAA SCITECH 2024 Forum

10.2514/6.2024-1315

An Overview of the Engineering Sketch Pad

John F. Dannenhoffer, III*
Aerospace Computational Methods Laboratory
Syracuse University, Syracuse, New York, 13244

The Engineering Sketch Pad (ESP) is a CAD-like system for the generation of geometric models for the analysis and design of complex configurations, such as aircraft. It is a feature-based, parametric solid modeler, that is freely available. A typical use of ESP is to generate the various views of a configuration that are required for multi-fidelity and multi-disciplinary models; these various views are typically linked to each other via ESP's rich attribution that are applied to the entities in the boundary representations. A unique feature of ESP is that the geometry creation process has been fully differentiated, so that one knows the sensitivity of every part of the configuration with respect to the user-defined design parameters.

Since its introduction in 2013, ESP has attracted a large user base of over 1000 users in more than 50 organizations and is used as the embedded geometry system in several mesh generation and analysis systems. Many of ESP's enhancements, such as a collaboration environment and its integration with the Computational Aerospace Prototype Syntheses (CAPS) system have been documented elsewhere; this paper provides an overview of those enhancements in one place.

Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315

The Engineering Sketch Pad (ESP)

Over the last decade, the Engineering Sketch Pad (ESP)[1] has been adopted by many organizations as the basis for the analysis of geometrically-complex configurations, such as aerospace vehicles. ESP is a geometry creation and manipulation system whose goal is to support the analysis methods used during the design process via the Computational Aerospace Prototype Syntheses (CAPS) program.[2,3] ESP's user interface runs in any modern web browser and its calculations are executed in a server-based backend program.

ESP is a solid modeler, which means that the construction process guarantees that models are realizable solids, with a watertight representation that is essential for mesh generators. Since some analyses require representations in terms of sheets or wires, ESP can support those too.

ESP's models are parametric, meaning that they are defined in terms of a feature tree (which can be thought of as the "recipe" for how to construct the configuration) and a set of user-defined design parameters that can be modified to generate families of designs.

Like all feature-based systems, ESP models start with the generation of primitives, which can either be one of the standard primitives (box, sphere, cone, cylinder, torus), or can be grown from a sketch as either an extrusion, body of revolution, or a blend of a series of sketches. In addition, ESP allows users to create their own primitives; for example, a series of airfoil generators are shipped with ESP. Primitives can be modified via transformations (translate, rotate, scale, mirror) and can have features such as fillets, chamfers, and hollows applied to them. Finally, bodies can be combined via boolean-like operators such as intersect, subtract, and union.

ESP maintains a set of global and local attributes on a configuration that are persistent through rebuilds. This association is essential in the support of multi-fidelity models (wherein the attributes can be used

*Associate Professor, Mechanical and Aerospace Engineering, AIAA Associate Fellow.

Copyright © 2024 by John F. Dannenhoffer, III. Published by the American Institute of Aeronautics and Astronautics, Inc., with permission.

Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315

to associate conceptually-similar parts in the various models) and multi-disciplinary models (wherein the attributes can be used to associate surface groups which share common loads and displacements). User-specified attributes are also used to mark faces and edges with information such as nominal grid spacings, material properties, or boundary conditions.

A key difference from ESP and all other available modeling systems is the ESP allows a user to compute the sensitivity of any part of a configuration with respect to any design parameter. Many of ESP's commands have been analytically "differentiated" or have used "operator overloading", making the computation of sensitivities efficient (since there is no need to re-generate the configuration) and accurate (since there is no truncation error associated with "differencing"). A few feature types still require the use of finite-differenced sensitivities, for which a mapping technique is used to ensure robustness.

As mentioned above, ESP is extensible, in that users can add their own user-defined primitives (UDPs) and user-defined functions (UDFs), both of which are written in C, C++, or FORTRAN and are compiled, using either a top-down or a bottom-up build process (or both). UDPs/UDFs are coupled into ESP dynamically at run time, meaning that organizations can encode their own definitions into ESP, and the UDP will be dynamically loaded at run time. Additionally, a user can write a user-defined component (UDC), which can be thought of as a "macro".

ESP models are defined in `.csm` files, which are human readable ASCII files that use a CAD-traditional stack-like process, but which also allows for looping (via patterns), logical (if/then) constructs, and error recovery via thrown/caught signals.

ESP's back-end (server) runs on a wide variety of modern compute platforms, including Windows, Linux, and MacOS. The server also allows for the addition of various tools, as described in the ESP extensions section.

ESP's user-interface (client) runs in most modern web browsers, including FireFox, Google Chrome, Safari, and chromium Edge.

ESP is an open-source project (using the LGPL 2.1 license) that is distributed as source, and is available from `acdl.mit.edu/ESP`.

ESP's Architecture

ESP has been developed in a multi-layered way, with clear definitions of the application programming interface (API) that is used to communicate with it. The layers are (starting at the bottom):

- **OpenCASCADE**[4] — this layer, which was developed by the OpenCASCADE Consortium, provides the geometric primitives and produces the Boundary representation (Brep);
- **EGADS**[5] — this layer provides an easy-to-use interface to OpenCASCADE and adds two very important capabilities, namely a rich, persistent attribution scheme, and a watertight tessellation (which can either be used to visualize the configuration or as the basis for solver-specific mesh generation). There is an evaluation-only version of EGADS, called EGADSlite,[6] which has a small memory footprint and which can easily be distributed through a high-performance computing (HCP) system;
- **OpenCSM**[7] — this layer adds the mechanisms necessary for feature-based parametric design, namely user-definable design parameters and a feature tree (which could be thought of as the "build recipe"). This layer also adds sensitivities to compute the change in any part of the configuration with respect to any design parameter. Such sensitivities are essential in order to perform optimal design via adjoints.
- **serveESP** — this layer is the system driver, responsible for translating user actions into calls to either OpenCSM or EGADS. In addition, this layer generates the graphical objects that can be displayed to the user.
- **ESP** — this layer controls the browser through which the user interacts. It also contains a sketcher, which allows users to interactively generate constrained sketches.

The bottom four layers (OpenCASCADE, EGADS, OpenCSM, and serveESP) run on a server, which can be either on Windows, Linux, or MacOS. Most of this is software is written in the C language, although there are a few parts of the program that are written in C++ or FORTRAN.

The top layer (ESP) runs on any modern web-browser (Google Chrome, Firefox, Safari, or chromium Edge) and is written largely in Javascript.

The interaction between the server and browser is handled via messaging, which includes both commands as well as the graphical representations.

Creating Geometry in ESP

Like all CAD systems, ESP can generate the BOX, SPHERE, CYLINDER, CONE, and TORUS standard primitives. In addition, ESP allows uses to create a cross-section (which is typically planar, but need not be) and then creates a solid body by EXTRUDE-ing, REVOLVE-ing (around an arbitrary axis), or RULE-ing/BLENDing a series of cross-sections. The difference between RULE and BLEND is that RULE does linear interpolations between the cross-sections, whereas BLEND creates a cubic spline. BLEND's cubic spline is by definition curvature-continuous (C2), although the user can make it only slope-continuous (C1) or value-continuous (C0) by repeating cross-sections. In addition, BLEND has the ability of creating rounded noses and/or tails (which are useful for fuselages) and can round-over wing tips.[8]

For aerospace applications, one has other shapes that are frequently used, such a airfoil shapes and super-ellipses. To accommodate these, ESP has the concept of a user-defined primitive (UDP), which is a user-supplied piece of code (in C, C++, or FORTRAN) that get dynamically loaded into the system at run time. The ESP distribution is shipped with over 60 UDPs.

Users can also extend ESP by writing macro-like code, called user-defined components (UDCs) (in the .csm language). These UDCs can either be include-type, in which case the UDC shares variables with its parent, or function-type, in which case the UDCs receives input values via arguments, returns output values, and has its own private variables; the writer of the UDC chooses its type. ESP ships with over 20 UDCs. (Note that the **Example Configurations** section below heavily uses UDCs of both type.)

Overview of the .csm Files

A .csm file is a ASCII file that contains definitions of the design and configuration parameters and the recipe needed to build the configuration. The .csm script used to create the bolt in Fig. 1 is:

```
 1: # bolt example
 2:
 3: # design parameters
 4: CFGPMTR   Nside     6      # number of sides
 5:
 6: DESPMTR   Thead     1.00   # thickness of head
 7: DESPMTR   Whead     3.00   # width     of head
 8: DESPMTR   Fhead     0.50   # fraction  of head that is flat
 9:
10: DESPMTR   Dslot     0.75   # depth of slot
11: DESPMTR   Wslot     0.25   # width of slot
12:
13: DESPMTR   Lshaft    4.00   # length   of shaft
14: DESPMTR   Dshaft    1.00   # diameter of shaft
15:
16: DESPMTR   sfact     0.50   # overall scale factor
17:
18: # make sure the number of side is even
```


Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315

```
19: IFTHEN    mod(Nside,2) NE 0
20:    MESSAGE the_number_of_sodes_must_be_even
21:    THROW   -999
22: ENDIF
23:
24: # head
25: BOX       0      -Whead/2  -Whead  Thead  Whead  2*Whead
26: PATBEG    iside  Nside/2-1
27:     BOX       0  -Whead/2  -Whead  Thead  Whead  2*Whead
28:     ROTATEX   360*iside/Nside
29:     INTERSECT
30: PATEND
31:
32: SET       Rhead  (Whead^2/4+(1-Fhead)^2*Thead^2)/(2*Thead*(1-Fhead))
33:
34: SPHERE    0             0   0    Rhead
35: TRANSLATE Thead-Rhead   0   0
36: INTERSECT
37:
38: # slot
39: IFTHEN    Dslot GT 0   AND   Wslot GT 0
40:    BOX       Thead-Dslot  -Wslot/2  -Whead  2*Thead  Wslot  2*Whead
41:    ATTRIBUTE _color $blue
42:    SUBTRACT
43: ENDIF
44:
45: # shaft
46: CYLINDER  -Lshaft   0   0   0   0   0   Dshaft/2
47: ATTRIBUTE _color $magenta
48: UNION
49:
50: SCALE     sfact
51:
52: END
```

Note the the line numbers are not in the script, but have been added here for explanatory purposes. The statements in a script are executed sequentially. The hash symbol (`#`) introduces a comment.

Line 4 defines a configuration parameter (CFGPMTR) and lines 6 through 16 define design parameters (DESPMTRs), which a user can set at run time, or which can be driven by an outside process, such as an optimizer. The difference between CFGPMTRs and DESPMTRs is that sensitivities can be computed with respect to DESPMTRs.

Lines 19 through 22 show an example of logic embedded in the script. These lines will send a MESSAGE to the user and THROW an error if `Nside` is not even.

Line 25 is the first statement that generates geometry: a BOX whose origin is at `(0, -Whead/2, -Whead)` and whose size is `Thead`× `Whead`×`2*Whead`. After this statement executes, this BOX is placed on the top of the "stack".

Line 26 contains an example of a loop (which in the CAD-world is called a pattern). The statements between lines 27 and 29 will be executed `Nside/2-1` times. Line 27 generates another BOX, which is placed on the top of the stack. The line 28 take the top BOX off the top of the stack and rotates it `60*iside/Nside` degrees about the X-axis, and places the rotated BOX on top of the stack. Line 29 INTERSECTs the two bodies on the top of the stack (the BOX created in line 25 and the rotated BOX produced in line 28, placing the results of the INTERSECTion on the stack. At the end of this loop (pattern), an n-sided body is all the

Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315

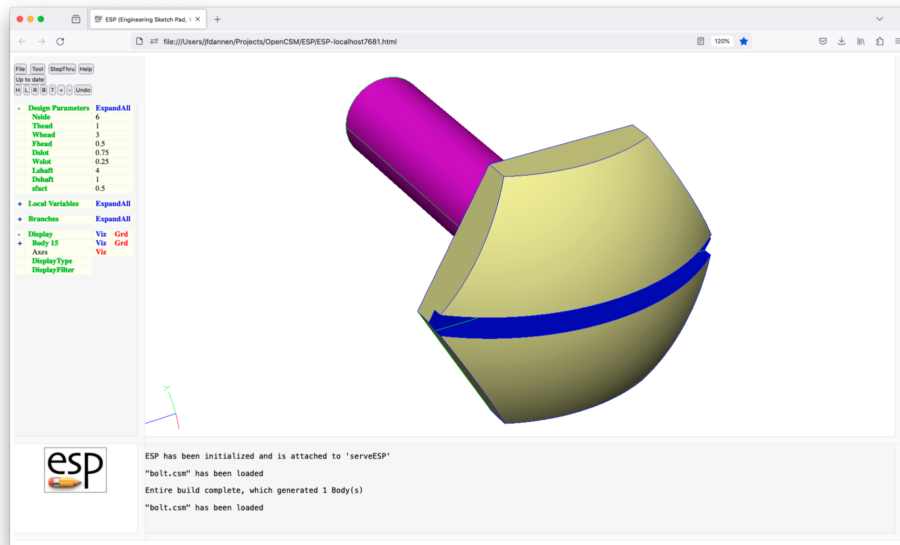


Figure 1. Bolt example

remains on the stack.

Line 32 is an example of setting a local variable (`Rhead`) to the results of the given expression. The syntax here is consistent with the expression rules in most modern computer languages. This is then used to define the SPHERE in line 34, which is TRANSLATEd, and INTERSECTed with the body that was left on the stack at the end of line 36.

In line 39, the script checks that both `Dslot` and `Wslot` are positive, and if they are it creates a blue BOX that is SUBTRACTed from the head (thereby making the slot).

Finally a shaft is created in line 46, colored magenta in line 47, and UNIONed with the head in line 48. The whole bolt is then SCALEd in line 50.

While some users struggle to get used to the concept of the stack, it is really no different that the way commercial CAD programs function, although the CAD systems do not call it a stack but instead rely on feature ordering. If you change the order of the features in a CAD program, you are essentially modifying their implied stack.

Example Configurations

Over the years, `ESP` has been applied to a wide variety of configurations,[9,10] such as those shown in Figs. 2 to 4. Each of these models is built from an `ESP` script that contained between 799 and 9615 commands. It has been observed that working with such large models can be very difficult, and this prone to errors. As a result, a rigorous process has been developed,[11] which organizes the whole script into three parts:

- a few header files that identify the various components (such as wing, fuselage, and nacelle) and various "views" (such a a vortex-lattice view, a CFD view, and a built-up-element view of the structure) that are available;
- a series of include-type user-defined components (UDCs) that build the basic parts of the various components (such as the outer mold line of the fuselage); and
- a series of include-type UDCs that the define the"views", which pull together the various components

Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315

needed for a specific analysis program.

In addition to breaking the overall script into pieces, the best-practice process described in the reference[11] discusses the importance of unit tests. This best practice clearly identifies the interactions between the various components (or disciplines), making the model easier to modify and extend.

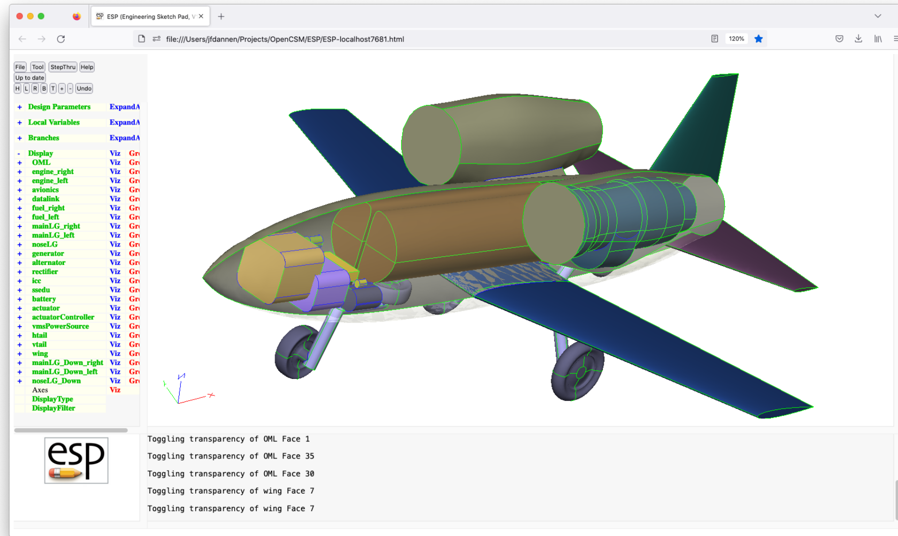


Figure 2. espRacer configuration generated with an ESP script containing 799 statements

Geometric and Tessellation Sensitivities

Unlike any commercially-available geometry-creating system, ESP can provide sensitives[12] with respect to the design parameters (DESPMTRs). The sensitivities come in two types:

- **geometric sensitivities** — these describe the motion normal to a surface (Face), perpendicular to an curve (Edge), or of a Node when the design parameter is changed; and
- **tessellation sensitivities** — these describe the motion that a grid point on a surface might have taken when the design parameter is changed.

The differences between these are described below.

The sensitivities in ESP are generally computed analytically, either by hand-differentiating the algorithms used to produce the geometry or via operator overloading in C++. In the few instances where the algorithm used (in OpenCASCADE) is unknowable (such as for a FILLET), finite differences are used. Obviously analytical derivative are preferable, both because they do not require that a perturbed configuration be created and because there is no need to pick a finite-difference step size. (Picking the finite-difference step size is difficult since picking one too large produces large truncation errors, whereas picking it too small makes it susceptible to round-off errors.)

For the Boolean operations (such as INTERSECTion or SUBTRACTion), ESP has a unique algorithm for finding the Edge sensitivities based solely upon the sensitivities of the adjoining Faces.[12] The consequence of this is that the expensive Boolean operations do not need to be re-executed when computing sensitivities, thus making the sensitivity process very rapid.

Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315

Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315

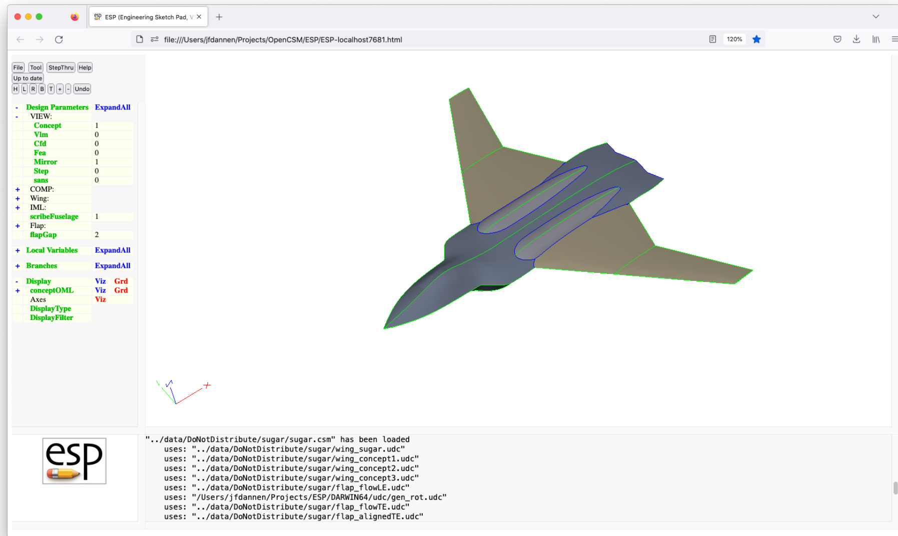
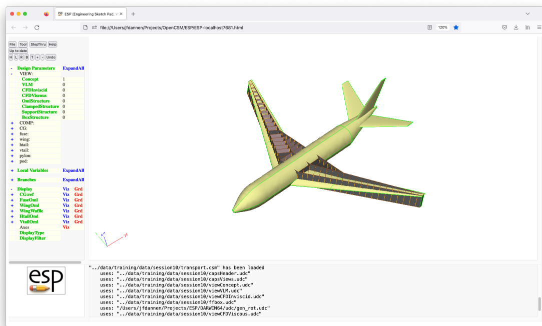
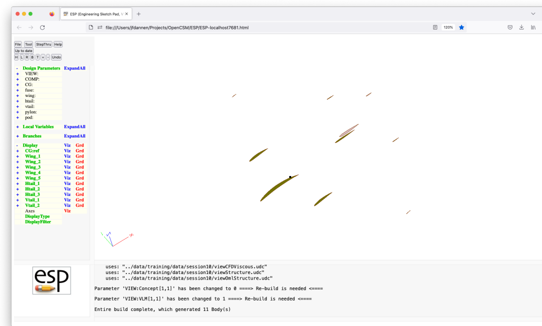


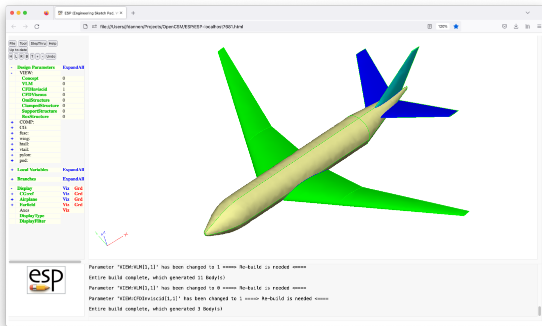
Figure 3. sugar configuration generated with an ESP script containing 9615 statements



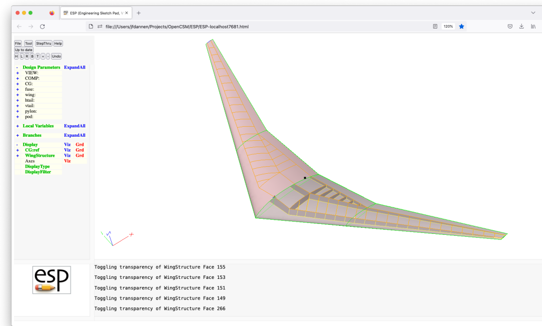
(a) conceptual view



(b) vortex-lattice view



(c) CFD view



(d) built-up-element view

Figure 4. model of transport generated with an ESP script containing 2320 statements

To understand the differences between geometric and tessellation sensitivities, consider the case of a cylinder, whose design parameters are `radius` and `length`. The geometric sensitivities on the rounded Faces with respect to the `radius` are `1.0` (i.e, the Faces grow outward) and the geometric sensitivities on the flat ends are zero, since there is no motion normal to the Face. Alternatively, the geometric sensitivities of the rounded Faces with respect to the `length` are zero (since there is no motion normal to the Face), whereas the faces on the ends have a geometric sensitivity of `1.0`. Note that in both cases, there is a discontinuity in the geometric sensitivities at the Edges that separate the rounded Faces from the ends. The geometric sensitivities are exact.

Now consider tessellation sensitivities for the same cylinder. Imagine that there was a tessellation (mesh) drawn on the surfaces of the cylinder. The tessellation sensitivities tell what the motion of the tessellation *might* be with respect to design parameter changes. The mesh points on all Faces of the cylinder move whenever the `radius` or `length` are changed. That is, if the `length` changes, the points on the rounded faces are dragged along the surface as the cylinder grows. The tessellation sensitivities do not exhibit discontinuity at the Edges.

The reason the word *might* is used above is because `ESP` does not know the mechanism that the tessellator uses when putting points on the surface, and thus in general does not exactly match the motion that would be obtained by regenerating the configuration with a perturbed parameter. To see this more clearly, if the tessellator spaces points evenly along the axial direction of the cylinder, then if the cylinder get longer, the movement of every point depends on its distance from the end. But if the tessellator prescribed the spacings near the ends of the cylinder and used a power-law stretching in the interior, then the points would move along the surface in a different way.

A study of the effect of this ambiguity in a design setting was conducted,[13] and it was found that as long as the tessellation gave the correct shape of the Face, there was no effect on either the final optimized solution or in the number of optimization iterations needed to get the result.

Using `ESP` for Geographically-dispersed Teams

In the post-Covid world, organizations are employing more geographically-dispersed teams than ever before. This can be burdensome when team member are working together to create a large model. While collaboration is possible via file sharing or screen sharing, none of the traditional design environments facilitate real-time collaboration. A similar problem in software development (programming) has been fixed with the introduction of pair-programming, wherein two programmers sit side by side, and work together. Pair programming has been shown to greatly improve the software development process, while only incurring a small cost penalty.

Other contemporary model-building systems claim to support collaboration, but do it in one of two ways:

- Cloud-based applications, wherein team member A creates a model and then publishes it in the cloud; team member B then gets the model from the cloud, add value to it, and publishes an updated model to the cloud; etc. In this way, multiple members can work on the same model, but can only so do sequentially; and
- Screen-sharing applications (like Zoom) allow one person to control and other to view the interactions with the modeling software. While better than nothing, these systems do not allow the collaborators to seamlessly change “who is in control”; therefore, there is really one user who can actively contribute.

`ESP`’s use of a web browser for the user interaction makes it ideally suited for a collaboration environment.[14] In particular, `ESP` has been architected such that more than one user can connect to a single session (which is running in the server). Therefore, multiple users in multiple locations can easily collaborate in real time. One user “has the ball”, and controls changes to the model; other users can see what the user with the ball is doing and have the option of either synchronizing their display with the user with the ball, or exploring the model independently. Unlike the screen-sharing approaches, the user with the ball can pass it to any other team member, at any time, therefore allowing every team member to contribute at the appropriate time.

Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315

During the summer of 2021, two Syracuse University students worked together for the summer to build models of balsa-wood aircraft models (Fig. 5). The students, who knew each other before the summer but had no `experience`, were located in different states. Their assessment of collaborating via `ESP` include the following advantages:

- the `.csm` scripts were stored in a common workspace (on the shared machine), and thus were accessible to both students at all times;
- since the `ESP` server was on the shared machine, the only software that they needed on their own computers were the browser-base code (`.html` and `.js` files). As a result, the fact that they had relatively slow personal computers was not a hinderance;
- the driver (user with the ball) could make any required changes and the results of the changes would be seen by both students almost immediately;
- this kept both users actively engaged in the process;
- the navigator tended to focus on big-picture (strategic) issues while the driver focused on typing appropriate commands, or tactical issues; and
- overall they felt that having two brains working on the same problem was beneficial.

The disadvantages that they noted included:

- users cannot simultaneously edit a script;
- there was only one version of the `.csm` script, so if a hard-to-find error was injected into the script, there was not a readily available backup.

Using Python to Automate Tasks

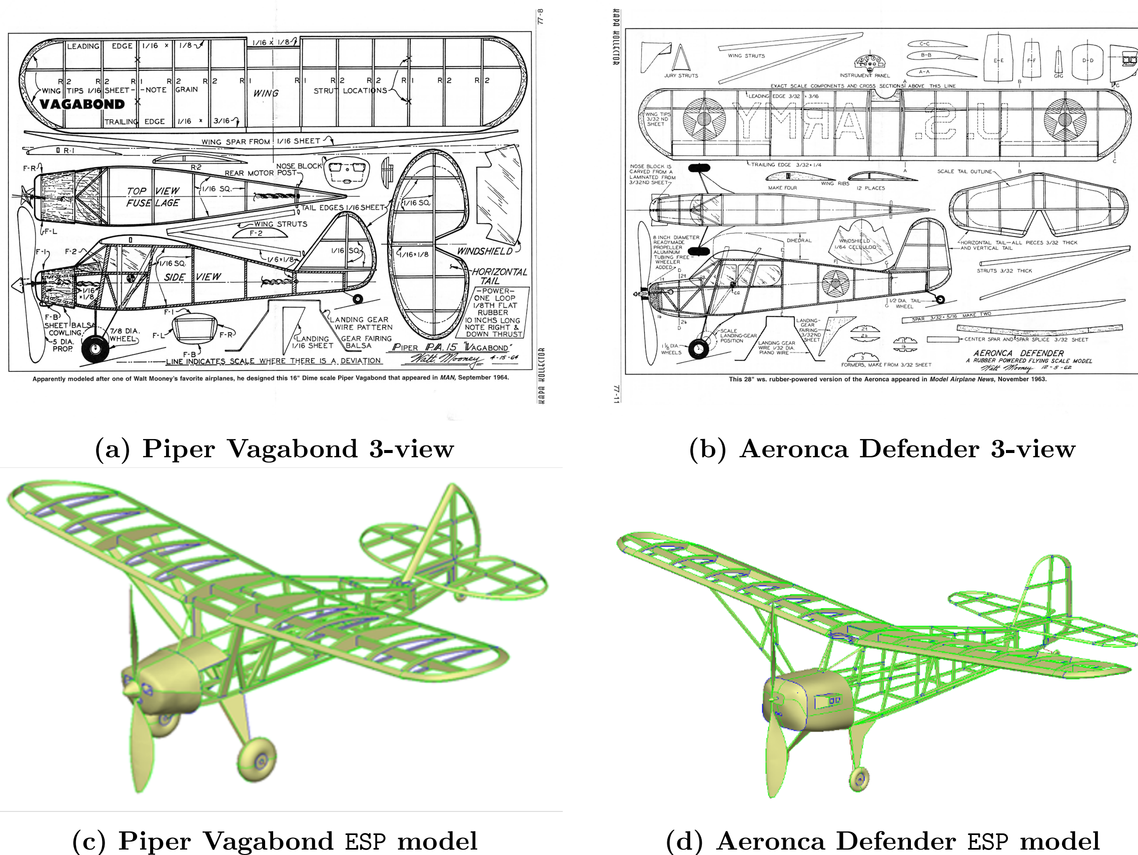
`ESP` includes an integrated Python interpreter that can be used to drive parameters and execution of a model. Over 70 `OpenCSM` commands are exposed to Python.

Using a Python script is best illustrated via an example: modify the fuselage of an aircraft so that the airplane's cross-sectional areas follow a prescribed area ruling. The python script to do it is:

```
##################################################################
#                                                                #
# areaRule.py -- adjust fuse:radius to satisfy Sears-Haack       #
#                                                                #
#              can be executed in either of two ways:            #
#                  serveESP filename   (such as areaRule1.csm)   #
#                      Tool->Pyscript areaRule.py                #
#              or                                                #
#                  python areaRule.py                            #
#                      filename        (such as areaRule1.csm)   #
#                                                                #
#              Written by John Dannenhoffer @ Syracuse University #
#                                                                #
##################################################################

from pyEGADS import egads
from pyOCSM  import ocsm
from pyOCSM  import esp
```


Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315

Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315



(a) **Piper Vagabond 3-view**

(b) **Aeronca Defender 3-view**

(c) **Piper Vagabond ESP model**

(d) **Aeronca Defender ESP model**

Figure 5. Models used by students in collaboration study.

```
import math
import os

#----------------------------------------------------------------------------#

# callback function
def pyMesgCB(text):
    print(" ")
    print("======= in pyMesgCB =======")
    print("    ", text.decode())
    print("===========================")
    return

# make a semi-colon-separated string from a list
def makeString(array):
    out = ""
    for i in array:
        out += str(i) + ";"
```

Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315

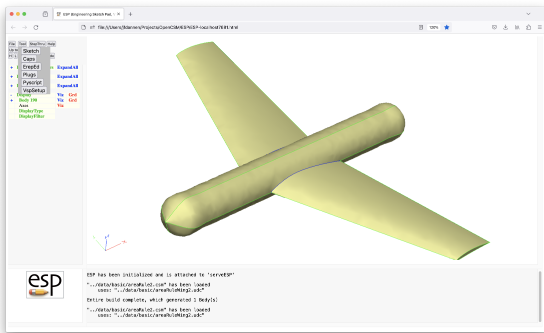
```
    return out

#--------------------------------------------------------------------------------#

# run quietly
ocsm.SetOutLevel(0)

# if we are running via serveESP, link to that MODL
try:
    modl = ocsm.Ocsm(esp.GetModl(esp.GetEsp("pyscript")))
    modl.RegMesgCB(pyMesgCB)
    print("==> getting MODL from ESP")

# an error means that we are probably running from the python prompt,
#    so get the filename from the user to create a new MODL
except ocsm.OcsmError:
    filename = ""
    while (".csm" not in filename):
        filename = input("Enter name of .csm file: ")
        if (not os.path.exists(filename)):
            print("\""+filename+"\" does not exist")
            filename = ""
    modl = ocsm.Ocsm(filename)
    print("==> making new MODL from \""+filename+"\"")

# check and build original MODL
modl.Check()
modl.Build(0, 0)

# get the pmtr indicies
ixloc   = modl.FindPmtr("fuse:xsect",    0, 0, 0)
iradius = modl.FindPmtr("fuse:radius",   0, 0, 0)
iarea   = modl.FindPmtr("aircraft:area", 0, 0, 0)

# get values from the MODL
nsect  = int(modl.GetValu(modl.FindPmtr("fuse:nsect",  0, 0, 0), 1, 1)[0])
length =     modl.GetValu(modl.FindPmtr("fuse:length", 0, 0, 0), 1, 1)[0]

xloc        = []
radius      = []
radius_lbnd = []
area        = []
for i in range(nsect):
    xloc.append(       modl.GetValu(ixloc,   i+1, 0)[0])
    radius.append(     modl.GetValu(iradius, i+1, 0)[0])
    radius_lbnd.append(modl.GetBnds(iradius, i+1, 0)[0])
    area.append(       modl.GetValu(iarea,   i+1, 0)[0])

    # find the maximum cross-sectional area
area_max   = 0
for i in range(nsect):
```

```
    if (area[i] > area_max):
        area_max = area[i]

# compute the sears-haack distribution
sears = []
for i in range(nsect):
    sears.append(area_max * math.pow(4 * xloc[i]/length * (1 - xloc[i]/length), 1.5))

# big iteration loop
niter = 10
for iter in range(niter+1):

    # show the current configuration
    if (iter > 0):
        esp.TimLoad("viewer", esp.GetEsp("pyscript"), "")
        esp.TimMesg("viewer", "MODL")
        esp.TimQuit("viewer")

    area   = []
    for i in range(nsect):
        area.append(modl.GetValu(iarea, i+1, 0)[0])

    # compute the rms error between the area and sears-haack
    rms = 0
    for i in range(nsect):
        rms += (area[i] - sears[i]) * (area[i] - sears[i])
    rms = round(math.sqrt(rms / nsect), 4)
    txt = "--> Iteration "+str(iter)+"  rms="+str(rms)
    print(txt)

    # show the area distribution
    esp.TimLoad("plotter", esp.GetEsp("pyscript"), "")
    esp.TimMesg("plotter", "new|"+txt+"|xloc|area|")
    esp.TimMesg("plotter", "add|"+makeString(xloc)+"|"+makeString(area)+"|k-+|")
    esp.TimMesg("plotter", "add|"+makeString(xloc)+"|"+makeString(sears)+"|g:|")
    esp.TimMesg("plotter", "show")
    esp.TimQuit("plotter")

    # check convergence
    if (rms < 0.01):
        print("--> converged")
        break
    elif (iter >= niter):
        print("--> out of iterations")
        break

    # compute new fuselage radii
    for i in range(nsect):
        radius[i] *= math.sqrt(sears[i] / area[i])
        if (radius[i] < radius_lbnd[i]):
            radius[i] = radius_lbnd[i]
```

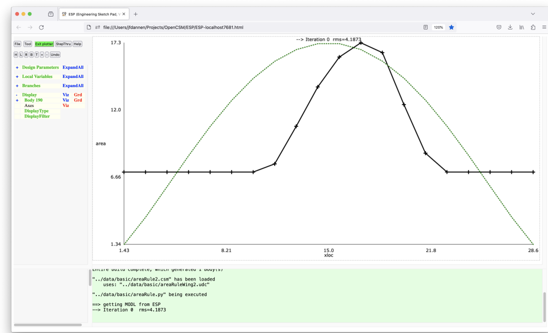

Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315

```
        modl.SetValuD(iradius, i+1, 0, radius[i])

    # rebuild with the new radii
    modl.Build(0, 0)

print("==> saving final DESPMTRs in \"./areaRule.despmtrs\"")
modl.SaveDespmtrs("./areaRule.despmtrs")
```

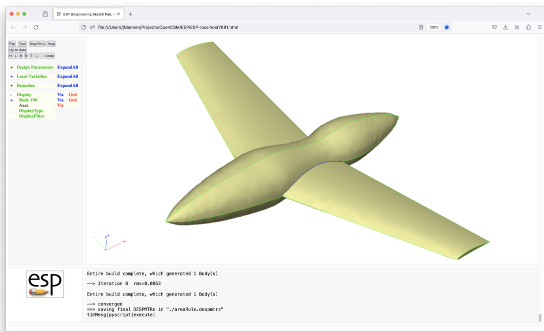
The results of this Python script is shown in Fig. 6. Part (a) of the figure is the original aircraft and part (b) is a plot of the airplane's cross-sectional area as a function of axial location, as well as the target Sears-Haack distribution. As this Python script executes, it stops after every iteration and shows the updated aircraft shape and area distributions. After only 8 iteration, the results shown in parts (c) and (d) were obtained. This demonstration also shows another of ESP's strengths: the ability to generate line plots as part of the process.



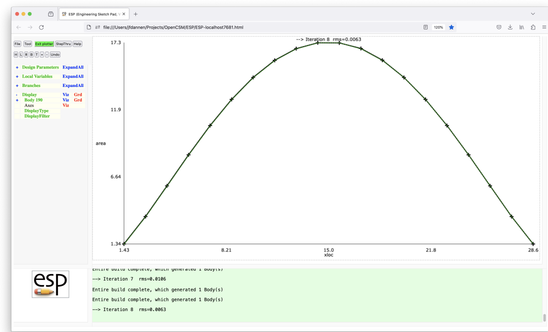
(a) original aircraft (with constant cross-section fuselage)



(b) original area distribution (solid black line) and target distribution (dotted green line)



(c) final aircraft (with optimize cross-section fuselage)



(d) final area distribution (solid black line) and target distribution (dotted green line)

Figure 6. Results of optimizing the area-rule for an aircraft.

ESP Extensions

There are several "tools" includes in ESP that provide a variety of additional capabilities:

Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315

- **Sketch**[15] — allows a user to make a constrained sketch in two dimensions. These sketches frequently serve a shapes that are EXTRUDEd, REVOLVEd, RULEd, or BLENDed;
- **Caps**[16] — allow a user access to the CAPS system, which offers coupling to a variety of mesh generators, flow solvers, and structural analyses;
- **ErepEd** — allows a user to logically combine Faces into a quilt (called an EFace - or Effective Face). This can the be used to make a single mesh over several faces;
- **Plugs**[17] — modifies the design parameters in a user-supplied model such that it fits the model, in a least-squares sense, to a cloud of points;
- **Plotter** — allows a user to display multiple line plots, with control over line and symbol types and colors;
- **Slugs**[18] — allows a user to build up a configuration by least-square fitting Faces (and Edges) to a cloud of points. Note that the resulting model is not parameterized;
- **Pyscript** — the Python interpreter (described previously); and
- **VspSetup** — allows a user to import an OpenVSP model into ESP. (This will be described in an upcoming publication.)

Availability and Training

ESP is an open-source project (using the LGPL 2.1 license) that is distributed as source, and is available from https://acdl.mit.edu/ESP.

That website contains:

- the latest released software, compiled for Windows, Linux, MacOS (both Intel and ARM);
- the full source for the latest release;
- a list of most of the ESP-related publications;
- copies of the slides and examples for the latest training, as well as recordings of the training; and
- an archive all previous versions (source only) as well as a beta release of the latest stable snapshot of the repository.

The full releases are made several times a year and are thoroughly tested and the documentation is updated. The beta release (in the archive section of the website) is tested, but the documentation may not be up to date.

The training sessions are available to everyone and are held roughly once a year. Send a note to jfdannen@syr.edu to get on the mailing list for the next training class.

Acknowledgment

The development of a system such as ESP is a team effort. The most notable contributer to ESP (other than the author of this paper) has been Bob Haimes (of the Massachusetts Institute of Technology), who was the project leader and key contributer to most of the ESP system since its inception, including EGADS and the base level of the CAPS system. More recently, Marshall Galbraith (also of MIT) has made significant contributions, mostly in the development of analysis interfaces to various mesh generators, and fluid and structures solvers in CAPS. Nitin Bhagat (of the University of Dayton Research Institute) has also been involved in many of ESP's development, serving as the main application developer, especially for multi-X

Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315

models. ESP has also been helped greatly by a series of project managers (and collaborators) at the Air Force Research Labs, including Ed Alyanak, Dean Bryson, Ryan Durscher, and Richard Snyder.

This work was supported by a variety of contracts and collaborative agreements, most recently the EnCAPS Project (AFRL Contract FA8650-20-2-2002): "EnCAPS: Enhanced Computational Prototype Syntheses".

References

[1]Haimes, R. and Dannenhoffer, J.F., "The Engineering Sketch Pad: A Solid-Modeling, Feature-Based, Web-Enabled System for Building Parametric Geometry", AIAA-2013-3073, June 2013.

[2]Alyanak, E., Durscher, R., Haimes, R., Dannenhoffer, J.F., Bhagat, N., and Allison, D., "Multi-fidelity Geometry-centric Multi-disciplinary Analysis for Design", AIAA-2016-4007, June 2016.

[3]Bryson, D.E., Haimes, R., and Dannenhoffer, J.F., "Toward the Realization of a Highly Integrated, Multidisciplinary, Multifidelity Design Environment", AIAA-2019-2225, January 2019.

[4]"Open CASCADE Technology", https://dev.opencascade.org/doc/overview.

[5]Haimes, R., and Drela, M., "On The Construction of Aircraft Conceptual Geometry for High-Fidelity Analysis and Design", AIAA-2012-0682, January 2012.

[6]Haimes, R., and Dannenhoffer, J.F., "EGADSlite: A Lightweight Geometry Kernel for HPC", AIAA-2018-1401, January 2018.

[7]Dannenhoffer, J.F., "OpenCSM: An Open-Source Constructive Solid Modeler for MDAO", AIAA-2013-0701, January 2013.

[8]Galbraith, M., and Haimes, R., "A Parametric G1-continuous Rounded Wing Tip Treatment for Preliminary Aircraft Design", AIAA-2022-1734, January 2022.

[9]Bhagat, N., and Alyanak, E., "Computational Geometry for Multifidelity and Multidisciplinary Analysis and Optimization", AIAA-2014-0188, January 2014.

[10]Dannenhoffer, J.F., and Haimes, R., "Generation of Multi-fidelity, Multi-discipline Air Vehicle Models with the Engineering Sketch Pad", AIAA-2016-1925, January 2016.

[11]Dannenhoffer, J.F., and Bhagat, N., "On Managing Geometric Models for Multi-component, Multi-disciplinary Analysis and Design", AIAA-2023-3599, June 2023.

[12]Dannenhoffer, J.F., and Haimes, R., "Design Sensitivity Calculations Directly on CAD-based Geometry", AIAA-2015-1370, January 2015.

[13]Dannenhoffer, J.F., and Haimes, R., "Using Design-Parameter Sensitivities in Adjoint-Based Design Environments", AIAA-2017-0139, January 2017.

[14]Dannenhoffer, J.F., and Bhagat, N., "Towards Modeling for Design: Using a Real-time Collaborative Environment in CAPS", AIAA-2022-2248, June 2022.

[15]Dixon, B.M, and Dannenhoffer, J.F., "Geometric Sketch Constraint Solving with User Feedback", AIAA-2013-0702, January 2013.

[16]Dannenhoffer, J.F., and Haimes, R., "An Integrated Design Environment for the Engineering Sketch Pad", AIAA-2023-0550, January 2023.

[17]Jia, P., and Dannenhoffer, J.F., "Generation of Parametric Aircraft Models from a Cloud of Points", AIAA-2016-1926, January 2016.

[18]Dannenhoffer, J.F., "The Creation of a Static BRep Model Given a Cloud of Points", AIAA-2017-0138, January 2017.

Downloaded by MIT Libraries on January 22, 2024 | http://arc.aiaa.org | DOI: 10.2514/6.2024-1315