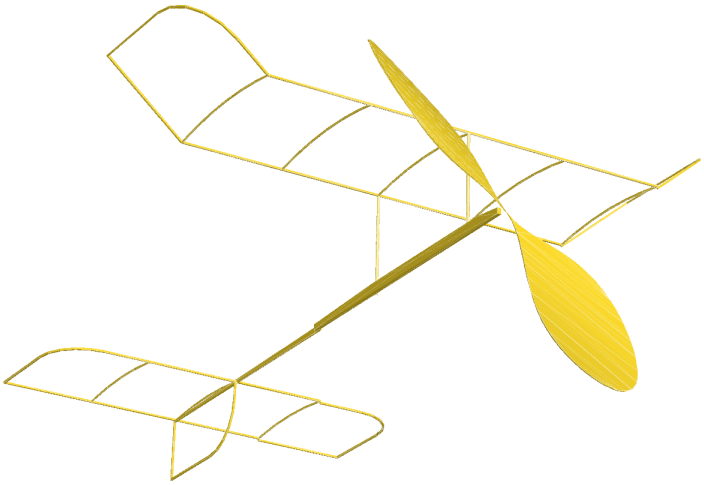
Designing an Indoor Model using OpenSCAD

Roie R. Black

January 27, 2021



1 Introduction

In case you have not noticed, 3D printers are becoming as common as the X-Acto knife in home shops these days. They let you, or your kids, build some amazing parts, some of which find their way into our model airplanes. If you look around on *Thingiverse* [**?**] where a lot of these 3d models can be found, you will discover that many of the files needed for 3D printing were generated using a neat open-source *Computer Aided Design* tool called *OpenSCAD*. I have used this tool for a number of projects, and I decided to see how it could help with my current indoor model building. In this article I will show how I developed a new design for a *Limited Pennyplane* model using *OpenSCAD*. I named the design *Math Magic* since I used a fair amount of math in its design.

OpenSCAD is not your usual CAD tool. It is a programmer's tool, meaning that you write fairly simple program code that describes your model. Then *OpenSCAD* processes that code to generate a visual representation of your design that you can see on your computer screen. Once you are happy with the design, *OpenSCAD* can export your design in file formats that could be post-processed as part of the path to a 3D printer. Unfortunately, we probably are not going to see competition ready 3D printed indoor models anytime soon, so we will not explore 3D printing in this article.

Do not be frightened off by having to write program code for *OpenSCAD*. Like anything new, it can be a bit intimidating for beginners, but I have worked on making the code needed simple enough that even non-technical folks should be able to generate useful

results.

This article is organized in three major sections. In *Part 1* I will explain the basic concepts used in *OpenSCAD* to set up a 3D design. This is not a complete tutorial, but covers enough to help you understand the rest of the article,

In *Part 2* we will work through the design on my *Limited Pennyplane* model. You will see some code here, but only enough to explain how I generated the design. Space limitations in this article prevent showing everything you need. All of the code and more detailed documentation on this project is available on the project website at **https://rblack42.github.io/math-magik** [**?**].

Finally, *Part 3* will show some analysis techniques I used to get a weight and balance assessment of my design. This is always a good thing to do before going flying! In this section, I will show a bit of Python code. *Python* is another programming language, commonly used in introductory programming courses. Kids in elementary school have managed to get going in programming using Python.

2 Part 1 - *OpenSCAD* Overview

2.1 Installing the tools

You will need to install two free programs to follow along with this design: *OpenSCAD* and *Python*.

2.1.1 Installing *OpenSCAD*

Installing *OpenSCAD* is pretty simple using instructions found on the program's website [**?**]. Basically, you download the installer for your system then run that program. Once it is installed, open it up and take a look at the basic interface. Figure 1 is what you should see.

There are three areas we will be using in this view:

- Editor - the left panel where you type in your code.
- Preview - the top right panel will be where your model is displayed.

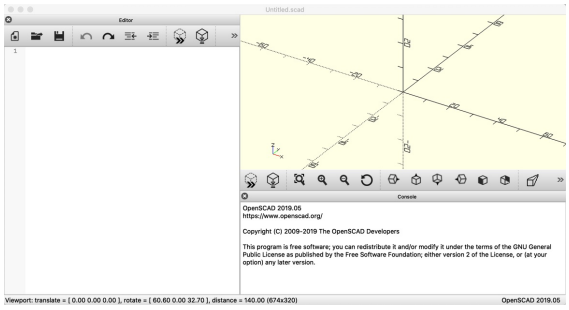


Figure 1: *OpenSCAD* interface

- Messages - the bottom right panel is where error messages will be shown when processing your code.

I will not attempt to show everything you need to know about the language *OpenSCAD* uses for describing a model. Instead, I will show fragments of code to give you a feel for what you need to write to design your model. The project website [**?**] has more details, as does the *OpenSCAD User Manual* [**?**].

I highly recommend that you print out a copy of the *OpenSCAD* "cheat-sheet" is available here: [**?**]. It will be a good reference as you see *OpenSCAD* code examples.

2.1.2 Installing Python

In the analysis part of this discussion, we will be using some *Python* programs to do some of our work. *Python* is another free tool, available for all platforms. It even comes pre-installed on some (sadly, not on PCs though). You should install this tool if you wish to follow along with this design. *Python* is really only needed for the analysis phase of your design work.

Navigate to the *Python* website [**?**] and download an installer for your system. *Python* does come with a simple editing tool, and there are many nicer development tools available. We will not explore those tools here. More details are on the *Math Magic* project website.

2.1.3 Command Line

Finally, some of the concepts in this article require running programs from the *command line*. Many of you have never seen this interface, although it is available on all systems. Basically, instead of clicking on some icon with your mouse to start up a program, you enter a line of text into the interface and tell the operating system what you want to do. I will only show a little of this here, more details are on the project website. You get to this interface by entering **cmd** in the Windows search box, or my opening the *Terminal* program on Mac systems. Linux users will open up a *shell* command window. We will only need to use this interface in the analysis of the design.

Ready to get started? Let's look at *OpenSCAD*!

2.2 Constructive Solid geometry

What does that even mean? Basically it means we construct bigger shapes using small set of primitive shapes, and a set of movement and combining operations.

The 3D shapes we will use include spheres, cylinders, and cubes. All of these shapes can be scaled and moved around using simple movement operations. We will also use some simple 2D shapes, like circles and rectangles, We will also use a more complex 2D shape called a polygon. 2D shapes have no thickness, but are useful when creating more complicated 3D shapes.

2.2.1 3D Primitive Shapes

For our first look at how you do things in *OpenSCAD*, here is a piece of code that shows the three basic 3D shapes:

Listing 1: *demo/demo1.scad*

```
cube();
translate([3,0,0])
  sphere();
translate([6,0,0])
  cylinder();
```

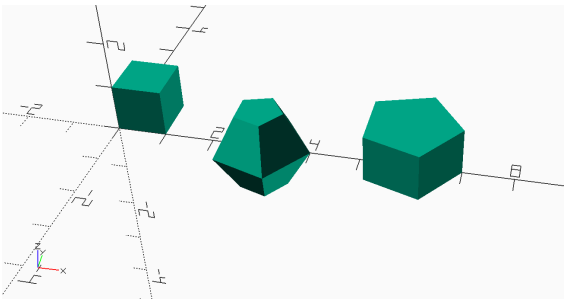


Figure 2: Demo 1

Figure 2 shows the result.

Each primitive shape is created at the origin. Cubes are created in the region where all three coordinates are positive. Spheres are created with the center of the sphere at the origin. The cylinder is centered along the **Z** axis. If you look closely, you will see a small representation of the coordinate directions at the lower left of this image.

We used a *translate* operation to move shapes aside so they do not overlap. The numbers inside square brackets control the distance we want to *translate* the following shape in the [**x,y,z**] directions. This bracketed group of numbers is is called a *vector* which we will use a lot in our work. The numbers *OpenSCAD* uses internally are in millimeters, but that is only important when we print things. I prefer to pretend those numbers are in our more familiar inches, and scale things when we generate printed output.

Notice how I indent code to show how things happen. This is strictly my style. *OpenSCAD* does not care about how you style your code. In this example, we *translate* the following *sphere* shape, then translate the following *cylinder* shape as well. The semicolon ends this command. Failing to put semicolons where they are needed is a common mistake when writing *OpenSCAD* code.

These shapes do not look quite right. The problem is that *OpenSCAD* generates approximations to the rounded shapes, using a set of small polygons to build up the model. If we make these polygons smaller, things look better. All we need to do to fix this is

change the code so it looks like this:

Listing 2: *demo/demo2.scad*

```
$fn=100;

cube();
translate([3,0,0])
  sphere($fn=100);
translate([6,0,0])
  cylinder($fn=100);
```

Figure 3 shows a much better result. The special variable **$fn** controls the resolution of rounded objects. Bigger numbers make things look smoother but it will take a bit longer to generate images on the screen.

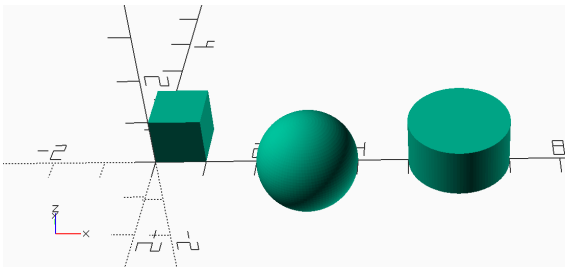


Figure 3: Demo 2

Some shapes are smart and can form different versions of themselves:

Listing 3: *demo/demo3.scad*

```
$fn=100;

cube([1,3,1]);
translate([4,0,0])
  sphere(r=2);
translate([8,0,0])
  cylinder(
    r1=1, r2=0.25);
```

Figure 4 shows a warped cube and cylinder. Spheres are not so smart, they stay spheres unless we warp them with external commands.

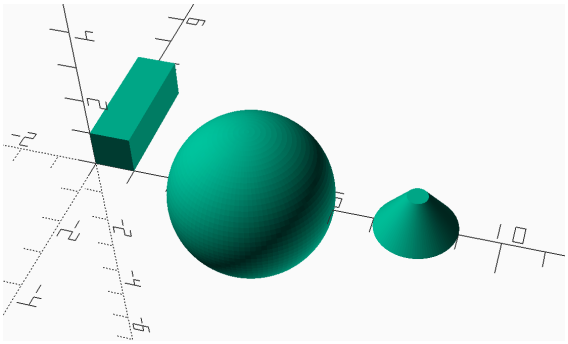


Figure 4: Demo 3

2.2.2 2D primitive Shapes

We will use a few 2D shapes in this design, including the circle and square, which act much like their 3D counterparts. A more interesting 2D shape we will use is the *polygon*.

Listing 4: *demo/polygon − demo1.scad*

```
triangle_points =[
  [0,0],[100,0],[0,100],[
  10,10],[80,10],[10,80]
];

triangle_paths =[
  [0,1,2],[3,4,5]
];

polygon(
  points=triangle_points,
  paths=triangle_paths,
  complexity=10
);
```

Here, we create two *variables* and set them equal to a list of vectors. 2D vectors have only 2 numbers, for the **X** and **Y** coordinate values. The first list defines a set of six points: three for the outer triangle, and three more for the inner triangle. The second list identifies *paths* meaning a continuous line that makes up a closed circuit, one for the outer triangle, and one for the inner triangle. The numbers refer to the position of vectors in the first list (programmers

count starting at zero!) That final **complexity=10** parameter is not important here. It helps the operation work properly.

I know this is a bit confusing, but we will not need much of this kind of code in our design work. Remember to try things and see what happens.

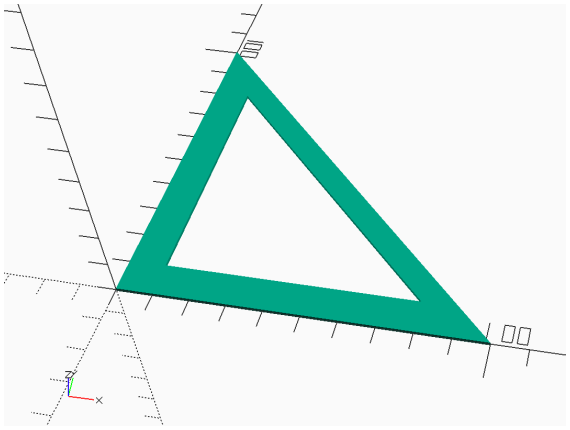


Figure 5: Polygon Demo 1

Figure 5 shows a 2D shape with no thickness, although *OpenSCAD* gives it enough of a thickness to show up on the screen.

subsubsectonExtrusions

We can use this 2D shape to create a 3D object by *extruding it* in the **Z** direction. This operation "pulls" the 2D shape upward to produce a 3D shape!

Listing 5: *demo/polygon − demo2.scad*

```
triangle_points =[
  [0,0],[100,0],[0,100],
  [10,10],[80,10],[10,80]
];
triangle_paths =[
  [0,1,2],[3,4,5]
];
linear_extrude(h=1)
    polygon(
      points = triangle_points,
      paths = triangle_paths,
      convexity=10
    );
```

Figure 6 definitely shows an interesting shape. We will use *extrusion* to make some parts that would be difficult to construct with just the basic primitive shapes.

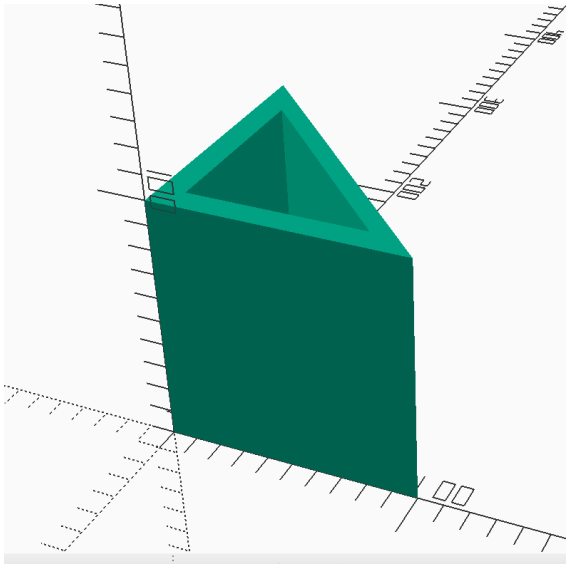


Figure 6: Polygon Demo 2

Obviously, we can form some interesting things with *OpenSCAD*.

2.2.3 Movement Operations

We saw the *translate* operation earlier. We can also *rotate* a shape. In this command we provide a vector of angles (in degrees) that we want to use to rotate the shape. Each number in the vector will be used to rotate the shape around the coordinate axis associated with that number. For instance **rotate([90,0,0])** will rotate the shape around the **X** axis ninety degrees. This operation uses the *right-hand rule*. If you want to rotate around the bf X axis, take your right hand and point the thumb in the direction of increasing **X** in your coordinate system. Your fingers "curl" around that axis in a positive direction.

Combining translations and rotations is done by writing both commands like this:

```
translate)[10,15,0])
  rotate[90,0,0])
    cube(1,1,5);
```

It helps to read this bottom up. We are creating a *cube* at the origin. We rotate it so it is aligned the way we want, then we translate that result to the position we have chosen. The semicolon at the end of this list ends the command. Notice that I indent so show what I want my code to do.

Be warned that you can swap the *translate* and *rotate* commands, but you might not get the result you expect. Rotations are applied to the shape as it is positioned when the command is processed. If you rotate after translating, The shape will swing a long way! In general, we will always do the rotations first when moving shapes around.

2.2.4 Combining Operations

In 3D modeling, we often want to combine shapes to create more complex shapes. That is important if we want to 3D print something. However, for most of our work in this design, we will not need to actually fuse two parts together, just position them correctly. However, there is one important combining operation we will use a lot. The **difference** operation actually takes two shapes that overlap, and slices off the part where they intersect. This operation is a combination of our knife and sandpaper block!

Let's try to taper a block of balsa that might represent a short spar!

Listing 6: *scad/spar − trim.scad*

```
trim_angle = atan2(1/8,5);

difference() {
cube([5,1/4,1/4]);
translate([0,-1/8,1/4])
  rotate([0,trim_angle,0])
    # cube([5.5,1/2,1/2]);
}
```

Figure 7shows what this looks like.

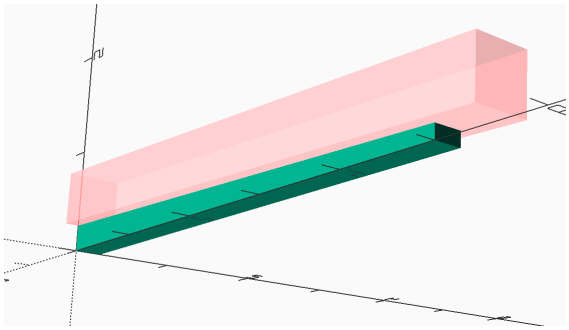


Figure 7: Spar Trimming block

This example shows some nice features of *OpenSCAD*. The first line calculates the angle we need to use to taper the spar to half its height. The **difference** operation takes the first shape inside those curly braces, in this case our basic spar, and uses the next shape, the trimming block properly aligned, to slice away where they intersect. That funny sharp character in front of the second cube makes *OpenSCAD* show you the trimming block for debugging. Normally, we leave this character off and the trimming block is invisible, and the intersection just evaporates. Like sanding blocks on steroids!

There are other combining operations we could use, but they will not show up much in our design work. The **union** operation combines two shapes into one, and the **intersection** operation takes two overlapping shapes an produces only that area where they intersect. We will use that last one in our analysis work in the last section.

2.2.5 Modules

OpenSCAD lets you package a number of operations in a *module* that you can activate later, one or more times. In fact those primitive shapes were all predefined *modules*. The module can have parameters, which makes this a powerful way to manage shapes that are similar, but differ depending on the parameters you specify. We will create a basic rib module for this model, and use parameters to control the exact rib we want.

All modules need to have a unique name in your code. The name you choose should help you remember what the module is all about. When you add parameters to a module, you define names for each parameter as well. These are surrounded by parentheses, with commas separating them if you have more than one. You can optionally provide a default value for each parameter by adding an equal sign followed by the default value you want. When you activate the module, you will provide actual values you want the module to use. You can just provide a sequence of numbers in the right order with commas separating them, or you can add the parameter name from the definition, an equal sign, then the new value you want. In this case, the order is not important, and you can leave off any parameters where you are happy with the default value. The rules for all of this are detailed in the *OpenSCAD User Manual* [**?**], so I will not go further in this discussion here.

2.2.6 Variables

All programs set up *variables* which are just a way we can name a number of some other data item. These variables can then be used in your code instead of the numbers. This makes it easy to modify something you use a bunch of times in your code. I tell my programming students to work hard at keeping real numbers out of their code. I can them *magic numbers* because after a while, you will see a number and have no idea why it is there.

We define a variable by picking a name for it, then adding an equal sign followed by the value you want to use. That value can just be a number, or you can calculate the value using some math and other variables. You will see enough of this ot get the idea, since my intent is not to teach programming here.

Variables can be defined in a master file that we can then use in any code we wish. We will go over how we do that later. These variables are called *global variables*. We can also set up variables inside of a module. These are called *local variables* and only code inside the module can use those.

We will see both kinds in this design work. Global variables will be used for things like **max_chord, and max_wing_span**. Local variables will be used when we need to store a number for use in the code we are writing. You saw an example in the code above where we calculated the **trim_angle**.

Successfully building 3D models involves visualizing what you want, then arranging simple shapes as needed and performing these three basic combining operations to generate the gadget you want! It takes practice! The more you experiment the better you will get! I encourage you to fire up *OpenSCAD* and type in this code. You will be better able to see how things work by doing this!

2.3 Code Management

We will be writing some program code in this design work. The code is not that complicated, at least for most things we will cover. However, if you are not a programmer, there are some details you need to be aware of.

First, you will be creating and managing a bunch of code files, which are just text files processed by one of our tools. I keep everything about a project in a directory (OK, folder for you PC folks) and create sub directories as needed to keep things organized. All of our tools will open up files on our system and process them to produce something we need. I have watched far too many beginners get confused by not keeping things organized and separate from other work you do on your system!

OpenSCAD lets you write your code in one file or in multiple files. All *OpenSCAD* code files have names ending with **.scad**. *Python* files all end with **.py**. You can see many such files by exploring the Internet. All of my code is kept on a public server used by millions of developers, called GitHub [**?**]. Many 3D projects, complete with the files needed to produce prints of those designs can be found on Thingiverse [**?**]. You can learn a lot by reading code you find on these sites.

I like to split up a design into multiple files in order to keep them short and focused on one just part of the design. If you split things up you will need to use either an *include* or a *use* line specifying the file

you want to access with the code in the present file. If you choose *include*, all that code in the second file will be processed as though it had been typed in the current file. On the other hand, using the *use* line only makes the names from the second file available in this one. The code in the second file will only be processed when those names are encountered in the current file.

A typical setup is to create a single file with dimensions you want every piece of code to be able to use. You *include* that file like so:

```
include <math-magik-data.scad>
```

2.3.1 The Design Process

Building a 3D model is a trial and error process. You type in or modify your code, then click on a command to process that code. You look at the preview window to see your model, and search the message area for hints about what went wrong. Non-programmers will find this a bit frustrating, but this takes practice to master, so do not get discouraged. My advice is to always take small steps. That limits the number of problems you face in getting things to work.

The best way to learn anything new is to experiment. Beginning programmers are always searching the Internet for solutions they can copy into their problems, but the only thing you actually learn when copying and pasting stuff is how to copy and paste. You will learn far more by typing in code yourself - at least until you get more proficient at this. Reading the code gives you a chance to really think about what is going on. There is nothing wrong with looking at code written by others. Many times studying that code will teach you how to better write your own code. I will show you enough code in this article to give you a feel for how you do things using *OpenSCAD*. The actual code I generated for this design is on the project Github account [**?**].

I always recommend building small files that generate one part of your overall design. Test that component until you are sure it looks like what you want. Then use that part in building other components. I like to work from small parts up to bigger assemblies, and that is how we will work through this design. Don't be afraid to fire up *OpenSCAD* and try things are you read this article. Of course, you should look at the project website to see all the code in greater detail.

Be warned that some of the code we will discuss ended up to complicated to show in this article. Instead, I will show the *OpenSCAD* screen display of more complex assemblies.

Now we can start work on our design!

3 Part 2 - LPP Design

3.1 Bottom-up Design

We could work through our design in a "top-down" manner, breaking up the model into major assemblies, then breaking down those assemblies into individual parts that need to be manufactured to complete the real model. However, using *OpenSCAD* we cannot see much until we have parts to build with. For that reason, we will begin with simple parts, then assemble them by properly positioning things to create assemblies. In the final step, we will put everything together so we can see our completed model! This all should make sense to an indoor model builder. You cut out the pieces you need, glue things together to form wings and stabilizers for instance, then assemble the final model using something like paper tubes perhaps.

We will start off by writing down the constraints we must obey to compete in the *Limited Pennyplane* class. I create a data file for this information, then add various other dimensions as I make design decisions. I can *include* this data file in any other file that needs this data.

As soon as we have enough information in hand to define the basic parts, we will begin building our CAD models. This will involve writing some simple code describing individual parts. Along the way, we will be defining other parameters like material thickness and overall parts dimensions. As much as possible dimensions will be derived from other data parameters defined in the design. For example, the rules

constrain the maximum chord for the wing. Once i know how thick my leading and trailing edges will be, I can calculate the chord of the rib I will place between those two parts. The constraints drive the process, as much as possible. We will document each design decision we make in the form of parameters, or math equations that are driven by other parameters in our design.

3.1.1 Parametric designs

Setting up the design in this way lets us adapt this design to create other designs. We might need to change the overall wingspan of the model, to compete in another class. We should be able to modify our constraining parameters, and let *OpenSCAD*generate a new model for the new rules. This is far better than simply scaling a PDF file so the wingspan is different.

A further benefit of this process is that you no longer confront plans that leave out critical dimensions. I do not like trying to figure out how to build some part by taking my digital calipers and measure things on a plan, or worse, a PDF printout, then trying to figure out the real dimensions! *OpenSCAD*forces you to completely define your model.

3.2 Design Constraints

The *Limited Pennyplane* class rules define a few constraints on dimensions for our model. Here is the start of my data file, where the important class dimensions are defined. All of these are *global variables.*:

Listing 7: *scad/math − magik − data.scad*

```
// LPP Constraints

max_wing_span = 18;
max_wing_chord = 5;
max_stab_span = 12;
max_stab_chord = 4;
max_overall_length = 20;
max_prop_diameter = 12;

minimum_weight = 3.1; // grams
```

What these constraints mean is that the model must fit in a box that measures **max_wing_span** wide by **max_length** long. There is no limit on how tall this box can be.

Furthermore, the wing must fit in a smaller box measuring **max_wing_span** by **max_wing_chord**. The stabilizer must fit in a similar box measuring **max_stab_span** by **max_stab_chord**. There are no constraints on where these boxes fit inside the outer box. The propeller is only limited by diameter, blade shape it up to the designer. However, the **max_length** constraint is measured from the forward-most point, usually on the propeller, to the aft-most point on the model. We could build a pusher, but I have not considered that idea.

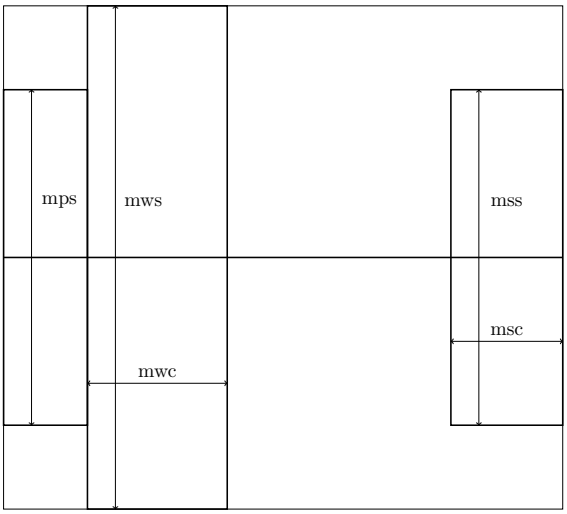


Figure 8: LPP Design Constraints

Note: The labels in this diagram are abbreviations for the names shown above. In my code I will use full names to improve readability of the code.

3.3 Building the Wing

Let's start off this design by building the wing.

Basic Wing Design

The class rules specify two constraints on the design of the wing. The projected span is limited to 18 inches, and the maximum chord is limited to five inches. Based on a survey of indoor models, we will use a flat, rectangular center section with wing tips angled upward, providing the needed dihedral for stability. We can define either the tip angle we want or the elevation of the tip. In either case, we can calculate the other item in our code.

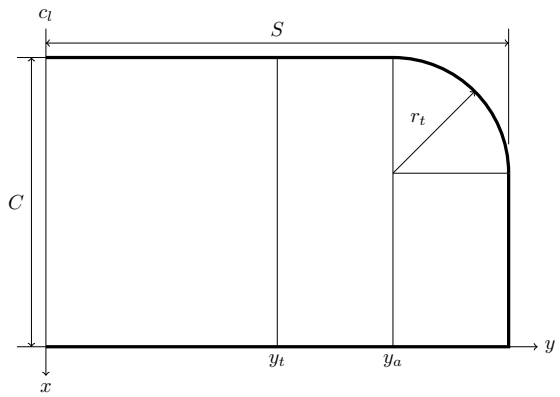


Figure 9: General Wing Geometry

Figure 9 shows the general layout of the wing and stab. (We can also use this for the fin, with minor tweaks.)

The tip uses another simple circular arc for the leading edge.

Figure 10 shows the dihedral configuration.

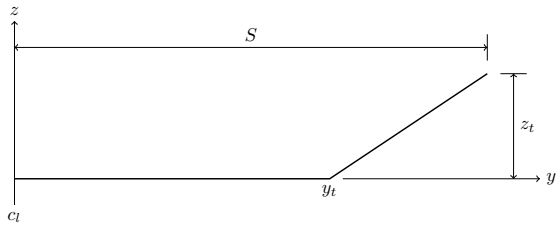


Figure 10: Basic Wing Dihedral Layout

3.3.1 Circular Arc Airfoils

Like many indoor model airplanes, we will use a simple circular arc airfoil. The airfoil is typically specified as an arc with a maximum height that is some percentage of the wing chord. So, given the chord, and thickness as a percentage value, we need to figure out the radius of the arc given these two values.

Since we start off with the chord and thickness specified as a percentage of the chord, we need to get rid of the percentage:

Given:

- **c** - the chord of the wing
- **T** - the camber as a percentage of **c**

$$t = Tc/100 \tag{1}$$

Here is a diagram showing what we are dealing with:

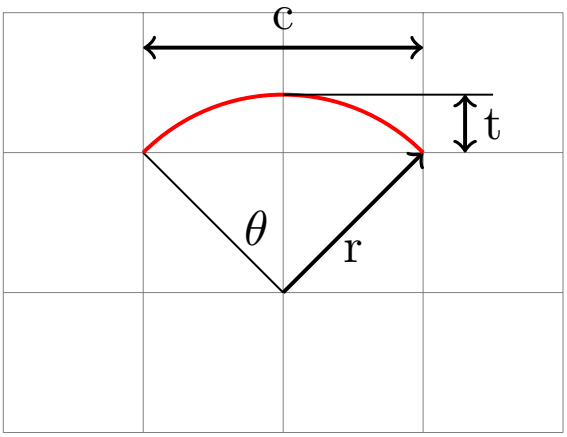


Figure 11: Circular arc Geometry

From this figure, we can write two equations:

$$r\sin(\theta) = \frac{c}{2} \tag{2}$$

$$t = r - r\cos(\theta) \tag{3}$$

The unknown variables are now:

- **r** - the radius of the circular arc
- θ - one half of the total angle swept by the arc

While we could drag out our old math books from high school, I would rather let my computer do the hard work. I did this math work using a cool *Python* package called *SymPy* [**?**]! This program manipulates math equations as symbols, like you used back in high school. Details on this are at the project website.

Here are the solutions:

$$\textbf{??} \tag{4}$$

The first solution is the **radius** we need to our code. The second one gives the half-angle swept by the arc for our rib. We do not really need to worry about that angle.

3.3.2 Wing Thickness function

In order to meet one of the design goals for this project, we need to come up with a formula that will give us the height of the wing at any point . Since we are using a circular arc airfoil, these equations get reasonably easy to figure out.

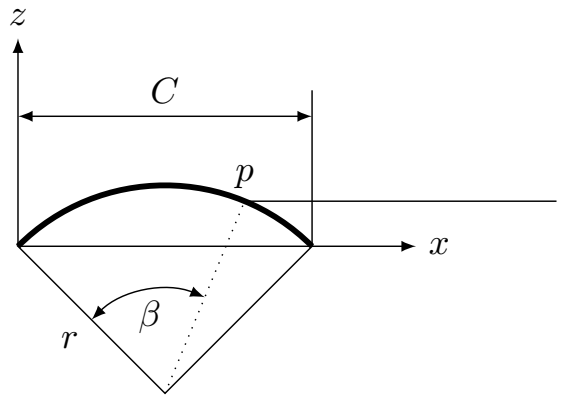


Figure 12: Wing Thickness Geometry

From figure 12 we can set up a few more equations.

Suppose we are interested in the height of our airfoil at any point along the x axis. From basic geometry, we come up with this equation

$$h = rcos\beta - (r - t) \tag{5}$$

We figured out the value for r in our earlier math work.

To figure out β we need to use the arcsin function.

$$x = C/2 - r\sin(beta) \tag{6}$$

Rearranging things we get:

$$\beta = \sin^{-1}(\frac{x - C/2}{r}) \tag{7}$$

That equation will give us the height of the wing at any point, depending on the camber, the local chord, and the value of x we use. The remaining problem is determining the chord at any point along the span of the wing.

The wing will have a nice constant airfoil across the inner portion. At the dihedral break, the height will taper off until it is flat at the tip. I will assume we can use a constantly decreasing camber across the tip, running from the center section airfoil to a flat tip. We need to come up with an equation to calculate the chord

The tip is a circular arc. From that we can use the general equation of or a circle centered at point c is:

$$(x - c_x)^2 + (y - c_y)^2 = r^2 \tag{8}$$

bit of rearranging gives us this result:

$$y = c_y - \sqrt{r^2 - (x - c_x)^2} \tag{9}$$

Using the height equation, and the chord equation, we have everything we need to figure out the height of the wing surface at any point. We will need that later.

3.3.3 The Rib Module

Now that we have our equations figured out, we can proceed with building a module that will generate a single rib. We will write the rib module so it takes four parameters to make using it for different chords easy,

```
module rib(
    chord = 5,
    camber = 6,
    rib_height = 1/16,
    rib_thickness = 1/32) {
  ...
}
```

I omitted the details here.

The **rib** module's job will be to generate a single rib sitting on the bf X axis with it's nose at the origin, and standing upright.

Notice that I defined default values for each of the parameters, If you are happy with those, you can generate a rib by writing **rib();**. You can change any of those by providing the parameter name, an equal sign, and a new value. So generating a rib with only a new chord value would look like this: **rib(chord=4);**. All other parameter default values still apply.

The code needed to generate a rib is a bit involved, so I will not show it here. Basically, it creates a cylinder using the radius we calculated from our formula, and a height equal to the rib thickness. Then it cuts out the center of that cylinder leaving a ring that will have the required rib stick height. Next, it slices off the unneeded part of the ring leaving just the rib. The final step rotates it upright, and aligns on the **X** axis, ready for use!

Here is an example rib:

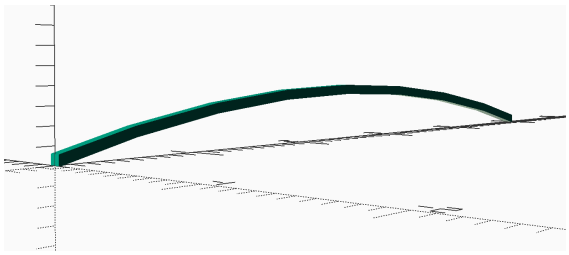


Figure 13: Basic rib

3.3.4 Wing Center Section

The leading and trailing edges of this model will be simple 1/16" balsa sticks. These are easy to model in *OpenSCAD* as really skinny cubes. The center section of the wing holds five equally spaced ribs. This code is pretty simple, so in the interest of keeping this article reasonable short, I will not show it here. All the code is placed in a simple **wing-center** module. Basically al it does is position the leading and trailing edge spars, then creates the five ribs, translating each into proper position. Figure 14 shows the result.

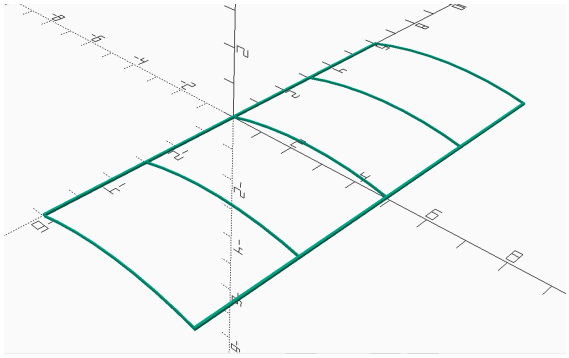


Figure 14: Wing Center Section

3.3.5 Tip Design

The tip is another rectangular section, except I decided to round off the leading edge corner. There are no ribs in this section, except for a flat rib at the tip. Here is the geometry I decided on:

The tip module uses a simple supporting module that generates the curved section. This will be a balsa stick formed over a template.

```
wing_right tip(
    span=3,
    chord=5,
```

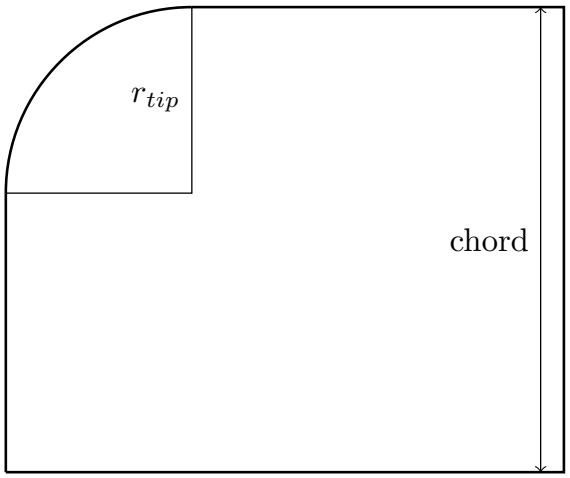


Figure 15: Tip Design

```
    radius=2
);
```

There is one interesting issue in the tip design. Many builders taper the leading and trailing edges so the tip is lighter. In this design, I decided to taper just the leading and trailing edge tip spars and make the circular arc and the tip thinner square stock. The puzzle is how to do this in *OpenSCAD*.

We saw how to taper a spare in our introduction to *OpenSCAD*. To get a double tapered spar, we just add another trimming block. Figure 16 shows the final tip section.

All we need to do now, is generate the opposite tip. We could duplicate all of the code we just set up, bu there is an easier way. *OpenSCAD* has an interesting operation called **mirror** that will create a mirror image of something. I used this feature to create the opposite tip.

All that remains is to add the dihedral. This is done by rotating the completed tip by the correct angle. If you specified the tip height, you need to calculate that angle using the **atan2** function we saw earlier.

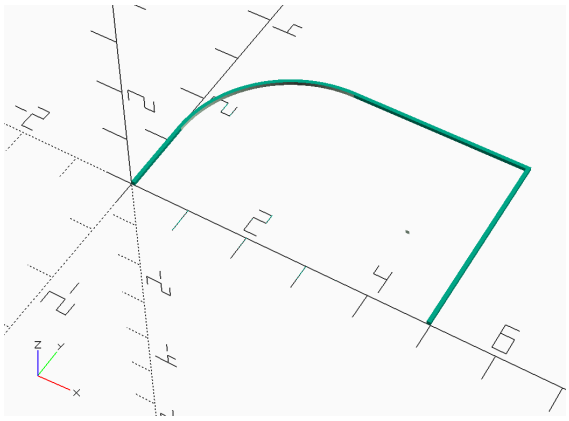


Figure 16: Wing Tip

3.3.6 Tip Templates

I kept this design simple on purpose. The only templates needed will be used for forming the ribs and the curved segments at the tips. Fortunately, *OpenSCAD* can help generate printed templates we can use! Here is the code used.

Listing 8: *demo/printer − test.scad*

```
use <wing-tip.scad>

left_margin=0.5;
bottom_margin=0.5;
printer_shift = 0.5;

projection(cut=false)
 scale([25.4,25.4,1])
   translate([
         left_margin,
         bottom_margin
           +printer_shift,
         0
    ])
      wing_tip();
```

The *scale* operation is used to scale the shape so it prints properly. Basically, we set up code that generates a tip section, then flatten it back into a 2D drawing. The operation that does this is called a *projection*. Once that is set up, we will ask *OpenSCAD*

to export this model as an SVG graphics file which can then be printed. I use my Chrome browser to do the printing on my home printer. Figure 17 shows what the template looks like in Chrome.

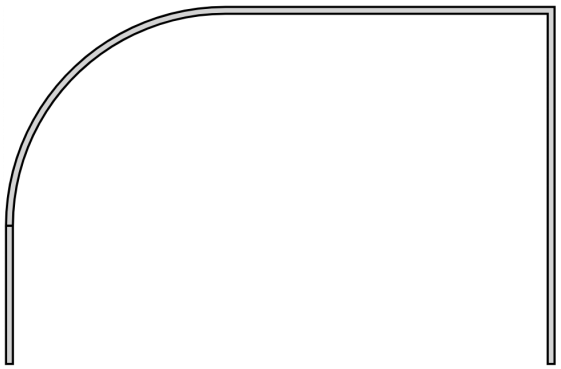


Figure 17: Tip Template

3.3.7 Wing Assembly

All we need to to to complete the wing is generate the center section, then generate the two wing tips and rotate them into place at the proper dihedral angle.

Figurefig:wing.pngFinal Wing shows the completed wing.

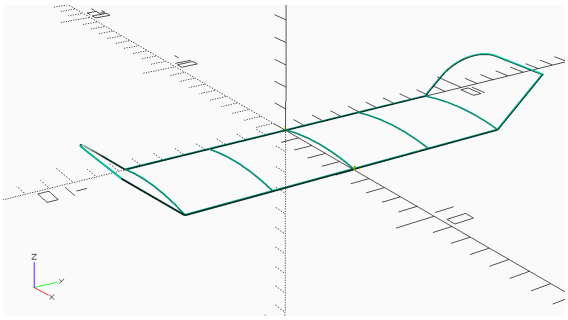


Figure 18: Wing Assembly

3.4 Stabilizer and Fin

The code we created to build the wing provides everything needed to build the stabilizer and fin.

3.4.1 stabilizer

The stab is identical to the wing, except in dimension and number of ribs. The modules used for the wing give us everything needed to build the stabilizer.

Figure 19 shows the result.

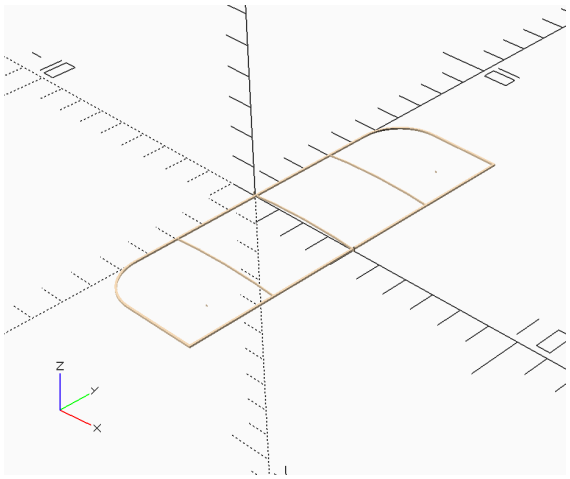


Figure 19: Stabilizer

3.4.2 Vertical Fin

The fin is a slightly different version of the tips. The only addition here is a square spar at the base of the fin. As we will see, we will mount the fin at an offset, and not glue it directly to the tail boom.

Figure 20 shows the fin.

3.5 Modeling the Covering

OpenSCAD does not provide a simple way to add covering to our design. However, I did find some code written by Justin Lin :[**?**] that can help. Justin created a module called *Bezier Surface* which takes an 3D matrix of control points and fits a smooth surface through all those points. We can then plot the smooth surface and give it a color so we can see it. I am using a blue glass color which is transparent so we can see the structure behind the surface.

Even better, we can submit this surface to the same

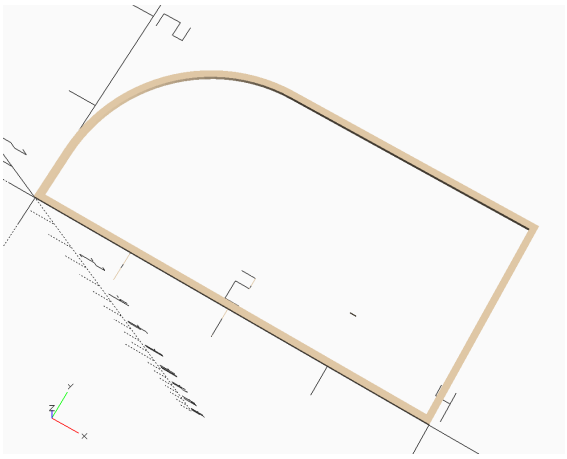


Figure 20: Vertical Fin

analysis we will be using to get weights and center of gravity data and add this new feature to our analysis.

This is where that wing surface function we figured out earlier will be used. I set up code that generates the needed surface points,, then used Lin's code to build a model of our covering. I made the covering thin enough so it will look right, and estimated the density needed to match real Mylar we might be using.

The code for this feature is a bit more involved, so I will refer you to the project website for details.

3.6 Motor Stick

With the flying surfaces set up, we can now turn to the fuselage parts. The most important of these is the motor stick, which supports everything else in the model. The motor stick must also bear the forces imposed by the wound up rubber motor that will power this craft.

We can set up the motor stick several ways, but I will use a *polygon* to define the basic shape, then use *linear_extrude* to generate the actual object.

Here is the basic layout we will use:

This design provides support for the front bearing

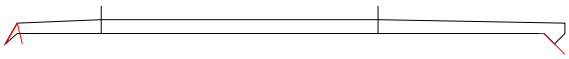


Figure 21: Basic Motor Stick

and the rear hook. The module only needs one parameter: the thickness of the stock you will be using for this part.

```
module motor_stick(thickness=1/8) {
  ...
}
```

The final shape is centered along the **X axis** with the bottom of the stick lying on that axis to make final assembly of the model easier.

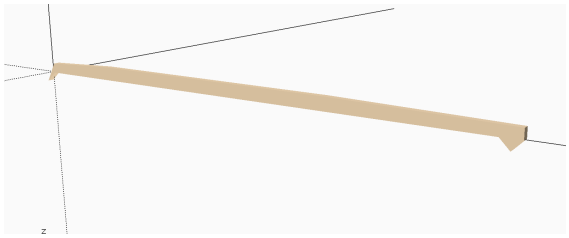


Figure 22: motor-stick.png

I added one new feature to this module, which will not show up in the printed article. You can ask *OpenSCAD* to color shapes using the *color* command. For this motor stick, I just added this code to the module:

```
module motor_stick*thickness=1/8) {
  color(WOOD_Balsa)
    rotate[(90,0,0)])
      translate([0,0,thickness/2])
  ...
}
```

I also added a single line at the top of the file:

```
include <colors.scad>
```

The **colors.scad** file is one I found online. I added my own color for balsa!

3.7 Tail Boom

After building the motor stick, the tail boom is something of a letdown. All we need here is a basic stick, but tapered from the front to the rear.

```
module tail_boom(
        thickness = 1/16,
        front_height = 3/16,
        rear_height = 1/16
        ) {
  ...
}
```

We will see this part when we assemble the airplane

3.8 Propeller

Now for an interesting component - the propeller. This is not just a flat part, it has an interesting shape, and figuring out how to model this thing took some time. In the end, I remembered how most of us build indoor propellers. We take a thin sheet of balsa, cut out the blade profile we want, then soak that blade for a while. Next, we tape it to the side of a round can at an angle and bake it. When it dries, it has a curvature that will work fairly well.

It struck me that I could generate the same shape by creating a polygon that represents the blade outline, then extrude that to form a very thick blade (bigger that the can!) I then build a hollow cylinder with a thickness that matches our desired blade thickness and slide the extruded blade into that cylinder. The *intersection* of these two shapes will leave us with a curved blade. Neat!

The shape of the blade seems to be a matter of taste. Many builders design blades that will provide for a flair so that the prop will have a higher pitch when the plane is launched with full torque from the motor. Moving the prop spar toward the trailing edge of the blade provides this flair.

I decided to generate a simple blade layout, with parameters that can be adjusted to give the flair you want. Figure 23 shows the general layout.

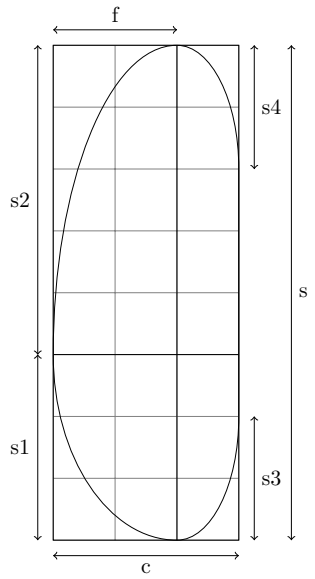


Figure 23: Prop Blade Layout

The prop spar is a tapered cylinder, something easy to generate with the *cylinder* shape. Parameters are provided so the size of the spar can be adjusted. i Figure 24 shows the final blade produced. Since the code is again a bot complicated, I will refer you t the project website.

3.9 Wire Parts

As I got to the end of this design, I wanted to finish it off by building the wire parts. I managed to find a few examples on Thingiverse that provided clues as to how I might do this. Unfortunately, the code is far too involved to present here. Instead, I will show one image from my experiments.

figure 25 shows an experiment. In this figure we see a basic propeller shaft and pieces of a wire bearing. The full code has this part finished off.

The wires are formed using very skinny cylinders. The curves in the wire are made using another extrusion operation provided with *OpenSCAD* called a **rotate_extrude**, which extrudes around a circular arc. The spring like part needed help from another

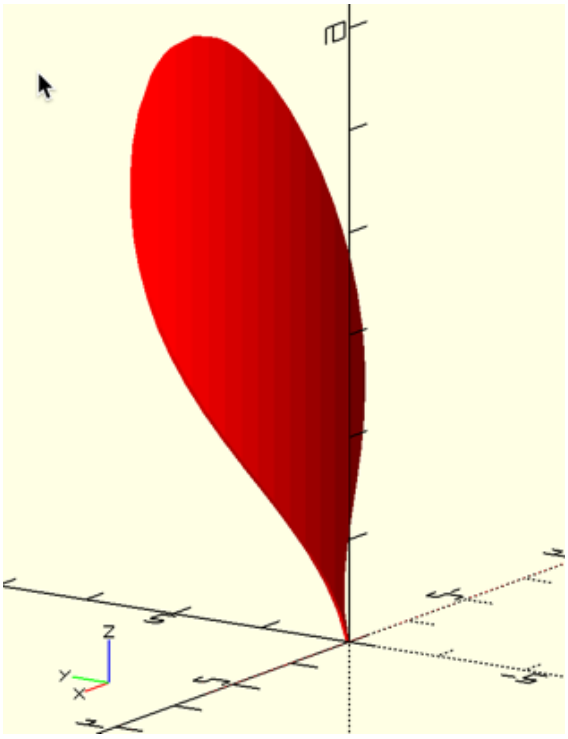


Figure 24: Prop Blade

piece of code found on the Internet. Getting this code running took a bit of work!

3.10 Final Assembly

Now that we have all of the major components defined, it is time to put things together and see the completed model.

3.10.1 Mounting Components

The wing will be mounted on top of two hard balsa posts using paper tubes. These wing posts are glued to the motor stock on the bottom and rounded at the top to slide into paper tubes that will be glued to the wing structure.

A similar arrangement is used to attach the stabilizer on top of the tail boom.

The rudder is glued to the bottom on the tail boom,

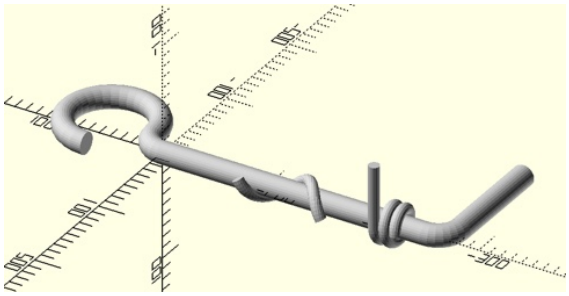


Figure 25: Simple Prop Shaft

but is offset to provide for a turn during flight. A small stick of balsa will be glued to the rear of the tail boom and the fin to provide needed support. Should this need adjusting, the rear attachment can be unglued and repositioned.

3.10.2 Paper Tubes

Paper tubes are formed on a mandrel using a strip of tissue soaked in thin glue and twisted around that mandrel. The resulting tube, after pulling off of the mandrel, will be stiff enough to provide the support needed. The posts must be carefully sanded to ensure a tight fit.

The code that creates a paper tube, it is just a very skinny hollow cylinder.

Figure **??** shows an image showing how the tubes are attached to both the wing and stabilizer:

3.10.3 Mounting Posts

The posts used to attach the wing and stabilizer are simple sticks of hard balsa with rounded tops that are sized for a tight fit in the paper tubes.

Since we need several mounting posts, the code that generates the post is placed in another module:

To attach the wing, we need to attach two posts to the motor stick. Figure **??** shows what they look like,

We use a very short post for the fin as well.

Both the stabilizer and fin mount on the tail boom.

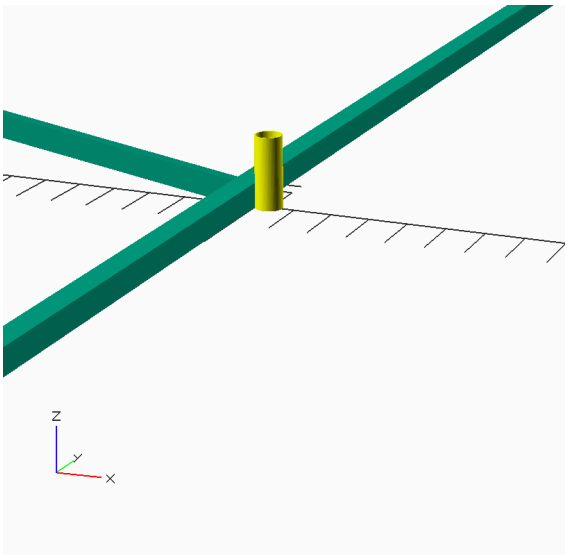


Figure 26: Wing Mount Tube

3.10.4 Stabilizer

The stabilizer is mounted on top of the tail boom, using small posts and paper tubes to allow for minor adjustments when flying.

3.10.5 Vertical Fin

The rudder is simple attached to the bottom of the tail boom, Since that side of the boom is canted by the trimming we performed earlier, we need to rotate the rudder slightly during positioning. However, we allow for left rudder attachment to provide a left turn flight path.

3.10.6 Adding the Propeller

We still need to set up the propeller. The wire parts need to be positioned, then the propeller translated into place. I rotated the prop a bit so the final image looks nicer. What you get us shown in the figure at the beginning of this article.

4 Part 3 -Weight and Balance Analysis

That part was very nice to see. However, we still have some work to do to complete this project. This time we will use *Python* and the *command line* to do our design analysis!

4.1 Generating STL Files

Many folks who use *OpenSCAD* generate *STL* files from their *OpenSCAD* designs. These files are used as part of the 3D printing process. Although I wish we could 3D print flyable model airplanes, that is not what we will do with the STL files in this project.

STL stands for *Standard Tessellation Library* [**?**]. Translated, that means an STL file is a list of triangular objects that describe the surface of a 3D object. These triangles cover the surface making a "water tight" approximation to the real surface. Triangles are guaranteed to be flat in the 3D space, and if they are tiny enough, they can be rendered to create a 3D display of the object, or sent to a 3D printer after suitable processing of all the triangles so the printer knows where the surface is located.

My purpose in introducing these triangles is simple. I want to know how much my model will weigh, and I want to know where the center of gravity will be when the design is constructed. Figuring out these two details is impossible using conventional building techniques: you build the model, weight it, and figure out the center of gravity manually. Computer geeks never do anything manually if they can get their computer to do the work.

I found some nice *Python* code that is all set up to figure out the volume of a 3D shape defined in an STL file, and return it's center of gravity location. It is simple enough to add in the predicted wood density for the design and come up with a weight estimate for each part. A little post-processing of all this data will give us some estimate of the total designs weight. Unfortunately, figuring out the weight of the glue is not so simple. For that we need to know where each glue joint will be and the surface area of the glue joint.

Fortunately, by designing our model using separate parts, this turns out to be fairly easy to do!

Knowing where each part will be placed in the final design, and the CG data for that part will let us predict the CG for the complete model as well!

Let's demonstrate this idea using a simple rib section.

4.1.1 Estimating Weights

I have worked to set up each major part you would need to create out of balsa, or other materials as a module we can activate separately. In this example, I will show how to estimate the weight of a single wing rib.

For this section, I will be using *OpenSCAD* as a *command-line* tool, something that may not be familiar to non-programmers. Details on this are on the project website, but basically, we will open up a window where we can type in commands to the operating system. This will involve running a **cmd.exe** on a Windows system, and opening up a *terminal* on Mac and Linux systems.

We will run *OpenSCAD* as a tool and will not open up the graphical user interface you normally see. In this mode, *OpenSCAD* will read a file, process it and generate an output file containing the results we want. In this example, we will be asking it to generate an STL file from a part definition *scad* file.

My **rib.scad** file is set up to display a single rib if you load it in OpenSCAD normally. The command I give to my Operating System to generate the *STL* file looks like this:

```
openscad rib.scad -o rib.stl
```

That is not so bad. If you check the directory where we ran this command (the one containing **rib.scad**, you will see a new file named **rib.stl**. You can examine this file, but all it contains is a bunch of definitions of the triangles needed to enclose a single rib.

Using the Python library to get the rib data is pretty easy. created a simple *Python* program to process this STL file. The program is named **getvolume.py**. I will not show the code here, but it is available on the project website.

This program is run from the command line as follows:

```
python getvolume.py rib.stl 4.5
```

This line tells the operating to run *Python* using my program code file. The next two parameters are the name of the part STL file followed by density of the part in pounds per cubit foot.

This will process the rib.scad file in the current directory, and print out the volume and CG information. It also prints the bounds of the box occupied by the shape, and the expected weight for the part.

Here is the output I saw from my test:

```
Processing rib with density 4.5
Shape bounds are:
  X: 0.0 <-> 5.0
  Y: -0.0156 <-> 0.0156
  Z: 0 <-> 0.362
Shape volume: 0.0099 cubic inches
Shape CG    : [2.5000 0.0000 2.2725e-1]
Estimated weight = 0.01157 grams
```

That is pretty cool, and the numbers look reasonable for this rib!

4.1.2 Glue Weight

We still need to figure out how much glue we will need. The scheme I came up with uses the *OpenSCAD* i*intersection* operation to give is an estimate of the glue surface.

It works like this. We take the two parts we want to glue together and slide one of them a tiny bit into the other part. Using the *intersection* operation will leave us with a tiny surface part that we can use as a model for the glue we will use. Of course, if you are a sloppy builder like me, this amount of c=glue might be off, but it is still a good way to get a weight estimate for the glue we will use.

I found some data on glue weights in a n article authored by Larry Coslick that I am using until I get some data of my own.

subsectionTransporting Your Model

4.2 Flight Box

Since indoor model airplanes are so extremely light and fragile, it is important that you give thought to how you are gong to transport the model. Most builders do not show up at a flying site with just one model. Often, they bring several for each event they want to fly. Many serious flyers build a nice carrying case for their models. I decided to design one for Math-Magik!

When I was first getting into indoor competitions back in my college days, I met Bud Tenny, who was the editor of a now-classic series of newsletters called *Indoor News and Views*. These newsletters provided a wealth of information on building indoor airplanes, and provided tips on how to fly them. Often there were sketches of simple boxes made out of cardboard, or foam board that looked like possibilities. However, I also have a copy of Ron Williams *Building and Flying Indoor Model Airplanes* [**?**] , and ran into the ECIM case, originally designed by the *East Coast Indoor Modelers* group. The design presented here is a version of that case:

As seen in figure 26, the box is made up of three sections. One holds a number of wings (four are shown here). One holds the stabilizers, and the middle section holds fuselages and propellers. The two end sections are hinged to the center section so it opens as seen above. Closed, this design measures 22"w x 9"h x 16"d. The outer sections are 6 inches deep and the center section ins 4 inches deep.

The sides are constructed out of 1/8" plywood and all corners are braced with 1/2" pine strips. The hinges are piano hinges that run the full height of the edges.

x Designing this box with OpenSCAD turns out to be a nice way to figure out how parts of the airplane will be transported and stored. The design of the airplane structure presented so far allows both the wing and stabilizer to be removed and stored separately. By attaching the wing posts to the motor stick and using paper tubes for the attachments to flying surfaces on top of those posts, the wings are not cluttered with structure and can be stacked neatly. The stabilizers also stack nicely.

Laying the body on its side lets those be stacked as well.

That leaves the propellers! They can be stacked separately in the space above the fuselage sections,.

When all of these items are placed in the box design, there is still room for another model. My current thinking is to adds an A-6 model or two to this box's payload!

I hope you see how handy a tool like *OpenSCAD* can be when you want to create things to support your hobby!

5 Conclusion

I hope I have shown you that *OpenSCAD* can be a powerful tool to add to your building. Yes, it may get you involved with programming, but that is not necessarily bad, You might discover that all kids in school are learning how to program computers these days.

Using fee tools like this is a great way to get started down this path. MY hope is that I can add come more detailed analytical tools to my project to get a better feel for how well this craft will fly. I did get degrees in Aerospace Engineering, but that was a long time ago.

Have fun trying this out, and please visit the project website. All of the details I could not fit into this article will be there, and anything else I come up with in the future.

If you need help with any of this, you can contact me at roie.black@gmail.com. I always welcome feedback on my projects!

6 Biography

In the Summer of 1955 I was delivering the evening newspaper in Falls Church, Virginia, when I rounded the corner of an apartment building and saw a man release the propeller on a rubber-powered model airplane. The plane circled in front of this man's home for several minutes, and magically landed where it had started. The airplane was a Henderson Gadfly, published in Model Airplane News that year. I was fascinated by that sight, and decided to figure out how the airplane managed to do that. I talked the man into giving me the plans he used to build the model, traced from the magazine. I still has those plans to this day!) Soon, a couple of my friends and I decided to start building model airplanes of our own. We all took a bus to downtown Washington, D.C (kids could do that back then), and joined the Academy of Model Aeronautics. We also joined the *Fairfax Model Associates* and began competing in a variety of events, mostly control line and gas free flight. At one meeting, Bill Bigge, an internationally known indoor model builder, was the guest speaker. I got my first look at a new form of model airplane. The indoor models Bill brought to the meeting were fascinating, and cheap enough even a kid with a limited allowance could build one. Bill became my mentor, and I managed to build an ornithopter and helicopter and set two national records! After almost getting a PhD in Aerospace Engineering from Virginia Tech, I spent 20 years as an officer in the USAF, then got a second Master's degree in Computer Science and spent another 17 years teaching college-level Computer Science. I finally retired for good in 2018, and moved with my wife to Kansas City, where I joined the *Heart of America Free Flight Association*, and again began flying model airplanes, this time focusing on rubber and electric powered outdoor free flight, and indoor events.When not building model airplanes, I am active in Amateur Radio and am currently authoring a book on Computer Architecture.

Contents

List of Figures

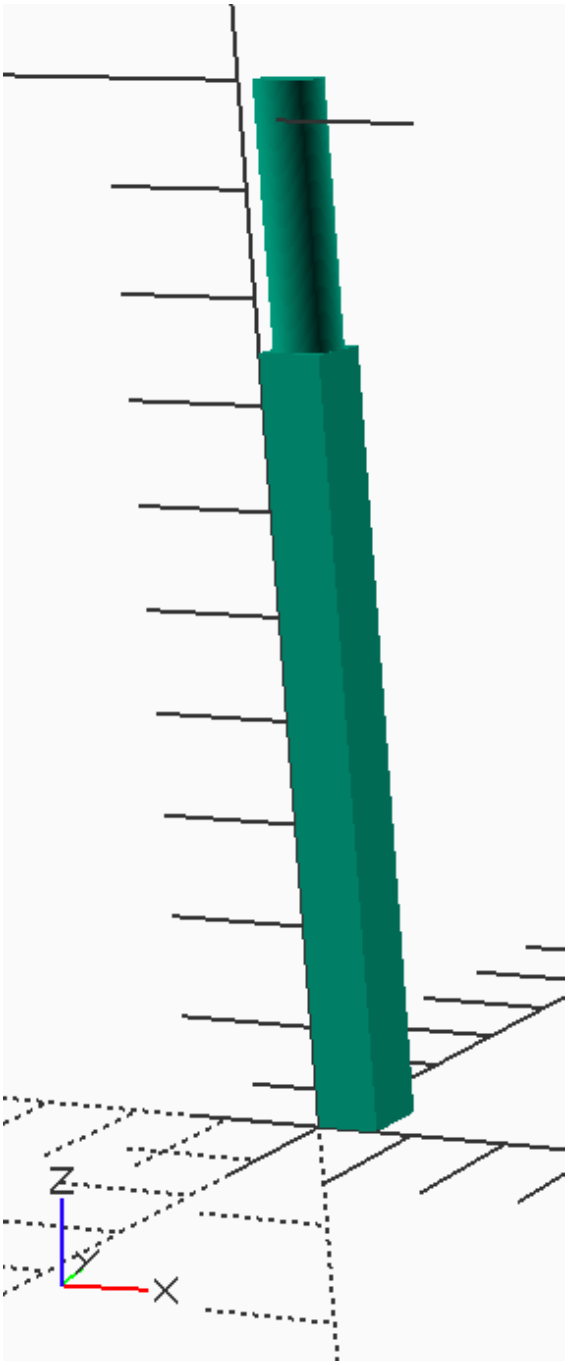


Figure 27: Mounting Post

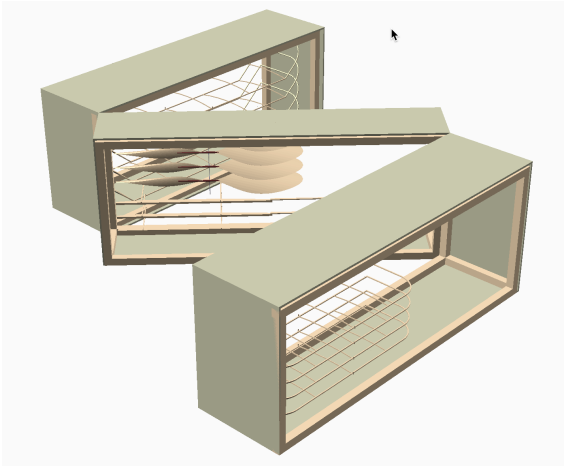


Figure 28: Flight Box