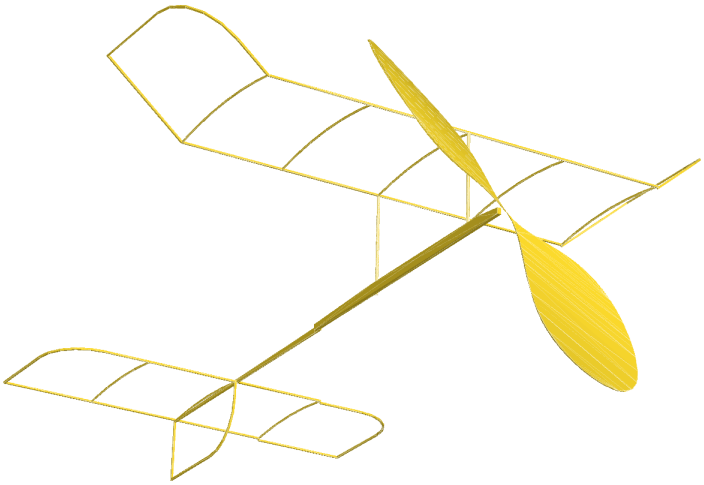
Designing an Indoor Model using OpenSCAD

Roie R. Black

January 25, 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* [**?**] 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*. As a retired Computer Science Professor, 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.

OpenSCAD is not your usual CAD tool. It is a programmers tool, meaning that you write fairly simple program code describing your model, then generate a visual representation of your design 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.

In this article I will show how I developed a new design for a *Limited Pennyplane* model using *Open-*

SCAD. 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. 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.

I have tried to keep the computer jargon to a minimum. Instead, I have created a way to generate *OpenSCAD* code that uses terms more familiar to model builders. Hopefully, this article will encourage you to try these techniques when you design your own model.

All of the code and more detailed documentation on this project is available on the project website at **https://github.com/rblack42/mathmagik** [**?**]. I will try to limit computer programming jargon, and present concepts in more familiar modeling terms.

2 Part 1 - *OpenSCAD* Overview

Installing *OpenSCAD* is pretty simple using instructions found on the project's website. Once it is installed, open it up and take a look at the basic interface:

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

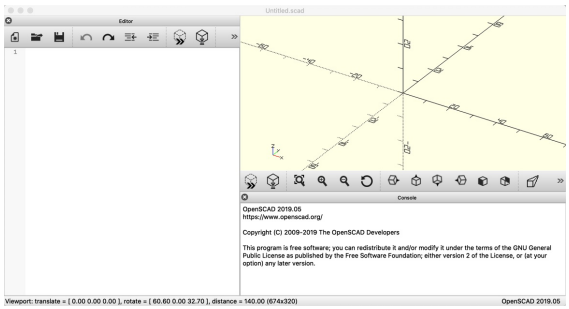


Figure 1: OpenSCAd interface

I will not try 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 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 programs are just text files. I like to use an editing tool designed for programmers to create my code. I have used a program called **gvim** for years as a professional software developer, but many other tools are available. All operating systems provide some form of editor that will do the job.

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.

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

3 OpenSCAD

Once you have *OpenSCAD* installed, open it up and take a look at the basic interface:

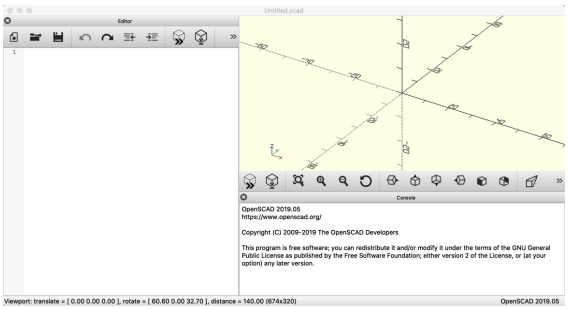


Figure 2: OpenSCAd interface

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

I will not try 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* [**?**].

As you work through this article, you might find the *OpenSCAD* "cheat-sheet" to be very handy. It is available here: [**?**].

OpenSCAD builds 3D models using a small set of primitive shapes, and a set of movement and combining operations to create more complex models.

3.1 Primitive Shapes

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. Openscad supports both 2D and 3D shapes. We will also be using some simple 2D shapes, like circles and rectangles, and more complex 2D shapes like a polygon. 2D shapes have no thickness, but are useful when creating complicated 3D shapes.

3.2 3D Primitives

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

Listing 1: *demo/demo1.scad*

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

Figure 3 shows the result.

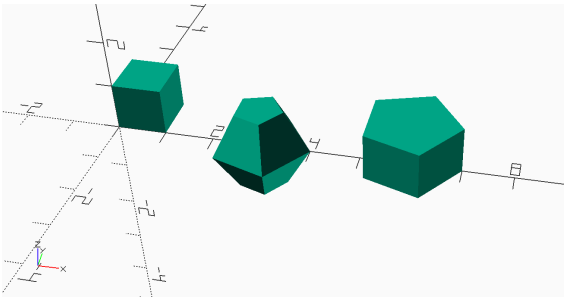


Figure 3: Demo 1

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.

Notice how I indent code to show how things happen. In this example, we *translate* the following *cube* shape. 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*

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

Figure 4 shows a much better result. The special variable *$fn* controls the resolution of rounded objects. Bigger numbers make things look smoother but cost of longer times to generate images on the screen.

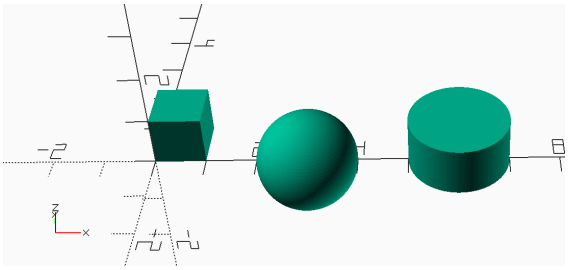


Figure 4: Demo 2

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

Listing 3: *demo/demo3.scad*

```
cube([1,3,1]);
translate([4,0,0])
  sphere(r=2, $fn=100);
translate([8,0,0])
  cylinder(
    r1=1, r2=0.25, $fn=100
  );
```

Figure 5 shows a warped cube and cylinder. Spheres are not so smart, they stay spheres unless we warp them with external commands.

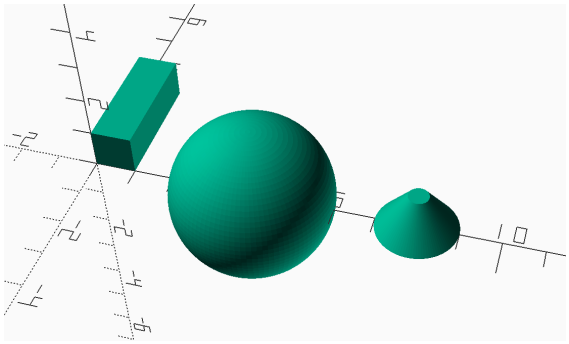


Figure 5: Demo 3

3.3 2D primatives

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(
```

```
  triangle_points,
  triangle_paths,
  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 **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.

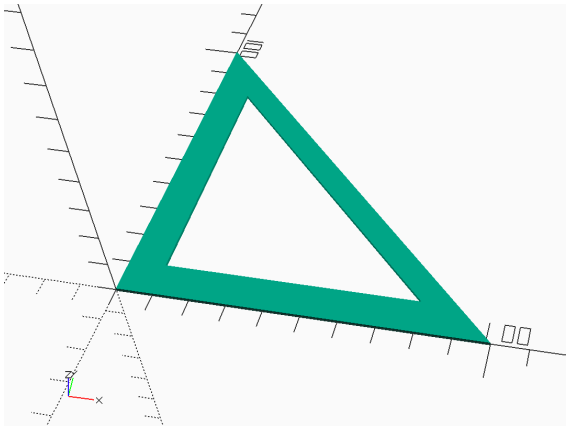


Figure 6: Polygon Demo 1

Figure 6 shows a 2D shape with no thickness, although *OpenSCAD* gives it enough of a thickness to show up on the screen.

We can use this 2D shape to create a 3D object by *extruding it* in the **Z** direction:

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 7 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.

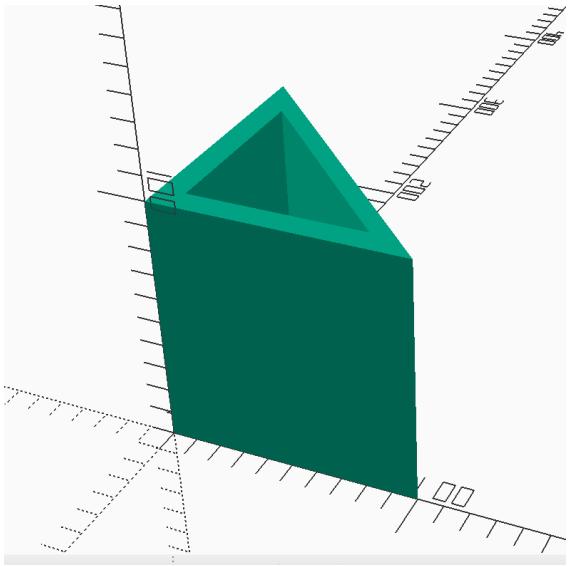


Figure 7: Polygon Demo 2

Obviously, we can form some interesting things with *OpenSCAD*. But things get even more interesting when we start combining multiple shapes to form more complex objects.

3.4 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!

3.5 Combining Operations

We form more complex objects by moving things around and combining them to form new objects. An example found in the *Wikipedia* article on CSG [**?**] demonstrates these operations.

Suppose you wanted to build something that looks like Figure 8.

We can form this shape using three cylinders, a sphere, and a cube. We use all three basic combining operations to construct the final shape.

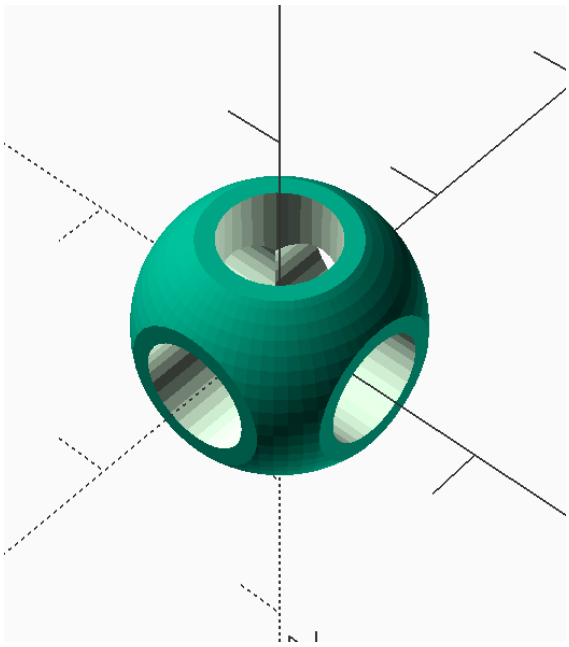


Figure 8: CSG Example Shape

Here is the OpenSCAD code used to generate this shape:

Listing 6: *demo/csg – demo.scad*

```
module core() {
  union() {
    cylinder(
      r=0.25,
      h=2,
      center=true,
      $fn=32
    );
    rotate([90,0,0])
      cylinder(
        r=0.25,
        h=2,
        center=true,
        $fn=32
      );
    rotate([0,90,0])
      cylinder(
```

```
        r=0.25,
        h=2,
        center=true,
        $fn=32
      );
  }
}

module round_cube() {
  intersection() {
    cube(
      [1,1,1],
      center=true
    );
    sphere(
      r=0.6,
      $fn=64,
      center=true
    );
  }
}

module part() {
  difference() {
    round_cube();
    core();
  }
}

core();
```

There is a point to be made here. We can move two objects together so they touch, like a rib to a spar, but we do not really need to join them together in this design work. Visually, things will look right, but the two objects remain separate. Joining them together to make a combined part would be important if we were going to 3D print the object. Since we do not have he technology to print with balsa (yet), I will not worry about combining the components of our design to create a single airplane object.

3.5.1 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 saw that when we showed "warped" shapes earlier. We will create a basic rib module for this model, and use parameters to control the exact rib we want.

All modules have a unique name in your code. The name you choose should help you remember what the module is all about. In this example, we are interested in the final **part** shape, which is constructed using the difference operation. This final module uses two supporting modules to build the part. You can write your code almost any way you like, but it is common to use spaces,indentations, and newlines to organize your code to make reading it easier. Also, we surround a sequence of individual operations inside of curly braces when needed. I always indent any code inside of these braces.

When you add parameters to a module, you define names for each one between the parentheses. Commas separate parameters 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 must 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.

3.5.2 Building the Example Shape

To build this part, we first set up three cylinders, aligned along each coordinate axis. The bf center parameter, sets each cylinder up with the origin of the coordinate system at the exact center of the cylinder. Remember, The **$fn=32** parameter is really only needed to make the cylinders actually look round.

Notice that all three of these cylinders occupy the same space. In the real world, we could not do that, but in our 3d modeling world this is common. We form the **union** of these three overlapping cylinders to form one merged shape.

The outer shell of our part is made up of the **intersection** of a sphere and a cube. We size the cube shape so it trims off six sides of the sphere where holes will end up. Finally, we use the **difference** operator to carve out the inside of the part, using our three-cylinder shape.

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!

3.6 *OpenSCAD* Basics

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 subdirectories as needed to keep things organized. Al 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**. P*Python* foles 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 od developers, called GitHub. 3D projects complete with the files needed to produce prints of those designs can be fon on Thiniverse. You can learn alot y 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>
```

3.6.1 Modules

Programmers like to group code related to one simple thing in a package *OpenSCAD* calls a *module*. I use modules to organize my design, creating them with names like **wing** or **rib so it is obvious what they are all about. If I create a rib module in one file, and my wing module in another file, I can** *use* **the rib module multiple times by adding this line at the top of the wing.scad file:**

```
use <rib.scad>
```

You will see a lot of this notation in the full project code files on the project website.

3.7 The Designing 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 design 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 highly 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.

4 Part 2 - LPP Design

5 The Design Process

Before we get into *OpenSCAD*, let's discuss how we will proceed with this design.

Since this is a new design, we will begin with the rules defining the constraints for this competition class. We will derive some basic parameters that will ensure our model conforms to the rules, something I want my design process to verify. We will proceed 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.

In using conventional CAD tools, you can start off with by creating a simple top-level drawing for your design and work out the details as you go. However, we cannot do that with *OpenSCAD*. Instead, we will start off by looking at the general layout of the model using some figures created using some of my professional authoring tools, which are not covered in this article. We will work through the design using these figures until we see what parts we need to create to build the full model.

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.

5.1 Parametric designs

Settnig up the design in this way lets us adapt this design to create other designs. We might need ot change the overall wingspan of the model, to meet 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 winspan is different.

A further benefit of this procss is that you no longer confront plans theat leave out critical dmensions. I do not like trying to figure out how to build some part by taking my digital calipers and measure int a plan, or worse, a PDF printout, then tryingot figure out the real dimnsions1 *OpenSCAD*forces you to completely define your model.

5.2 Coding the design

Programmers break down long complcated lists of instructions into more manageable parts. You are probably familiar with the concept of a trigomemetric sin— function from your geometry class. Someone created the logic that is used inside of that function and packed it in what we will be calling a *module—. when you need that function, you will refer to it by name, and the computer will activate the logic inside of that module ot produce the result you want. We will package all the code for creating a single part in such a [module*, then package the assembly of a number of pars into a single component in another [*module* as well. The top-level module will bring everything together to create the entire airplane.

I will need some "sketches" showing the basic layout of the model. Reather than present pencil sketches, I will sow layouts n the form of figures produced using some of my authoring tools. Those are not covered here, but are shown to define some of the details we need to know before actually proceeding witht he design.

Once we know what we want to build, we need to identify the major parts and assemblies we need to constrct to create a finished model. We will use *OpenSCAD*to creare those parts, and assemble them, just like we would if we were building a real model. Finally, we will analyze Designing a new model airplane usually starts with some sketches of some sort, so you can start to see how your design will look. You probably then progress to generating a plan of some sort. You might then refine your sketches by generating a more detailed plan using some form of computer tool. Finally, with a plan in hand, we proceed to constructing a prototype model to see how well it flies.

Sure, you could simply cut balsa, and glue things together just based on what looks "right", but in the end you need those plans, especially if your design seems to fly well!

It seems that we need plans for most projects. Good plans are pretty detailed, others barely show you anything but the general layout of the airplane. I have quite a collection of PDF files containing plans for indoor models I have considering building. However, since I want to build my own designs, I need to generate my own plans.

5.3 Generating Plans

Of course you can use a pencil and paper to generate your plans, but if you think you might want to publish your plan, you will need to use some form of *Computer Aided Design* tool to produce your final plan. Unfortunately many popular CAD tools are complex, and often too expensive for the average modeler.

Most CAD tools depend on using the mouse to maneuver components around. These tools do allow you to specify dimensions exactly, but often, details are left to the program to figure out as you "snap" parts together. The learning curve for these programs is very steep, but once you get the hang of things, they become much easier for you to use.

So why look at anything else?

5.4 Parametric Design

There is another way to create your plans. In this technique, you define some basic parame-

ters, like maximum wing span and chord, then use those dimensions to calculate the dimensions of other parts. Once you decide on a wing outline design, you calculate the sizes of each rib using those design parameters. If you need to design a similar model with different parameters, the recalculation of The local c

Having recently retired from teaching Computer Science, and finally getting back into model building, I decided to design a new indoor model for the *Limited Pennyplane* class. As part of the design process, I wanted to see that airplane in 3D even before I built the first prototype. I decided to use a different form of CAD tool: OpenSCAD [?], a tool designed for computer programmers!

While that description may discourage some folks from reading further, rest assured that this particular tool is simple enough that non-programmers can certainly master it. In fact, some teachers have successfully managed to get elementary school kids to use OpenSCAD to design simple 3D models.

OpenSCAD is an open-source (meaning free) 3D modeling program, available on all major platforms. It is commonly used by folks designing parts to be printed on 3D printers. What makes OpenScad different is how you generate the design. Instead of using your mouse to drag things around on the screen, you describe your model in a simple programming language. Formally, OpenSCAD uses something called *Constructive Solid Geometry* to construct your model, then gives you a visual interface you can use to examine your 3D model in detail.

I will only show example code from the project so you can get a feel for the design process. You are encouraged to explore the project website for much better documentation and complete source code. [?].

6 Design Constraints

The *Limited Pennyplane* class rules define a few constraints on dimensions for our model. Specifically we must honor these limits:

- max_wing_span - 18”
- max_wing_chord - 5”
- max_stab_span - 12”
- max_spab_chord - 4”
- max_length - Max Prop to Tail length = 20”
- max_prop_diameter - 12”

What this means 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.

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.

7 Building the Wing

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

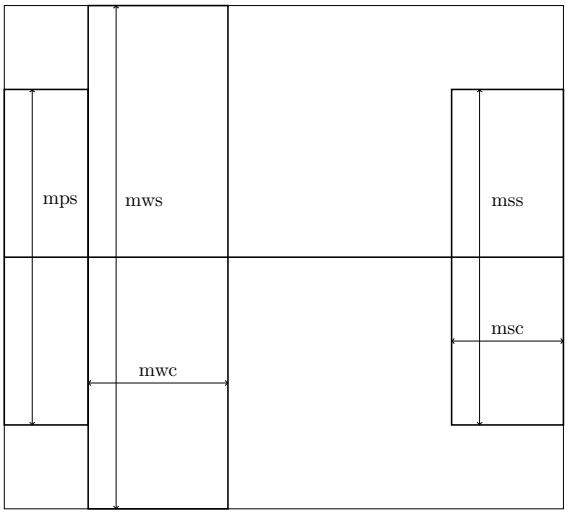


Figure 9: LPP Design Constraints

Basic Wing Layout

The class rules specify two constraints on the design of the wing. The projected span is limited to 18 inches, nd the maximum chord is limited to five inches. Based on a survey of indoor models, we will use a flat center section with wing tips angled upward, providing the needed dihedral for stability. Here is our starting geometry:

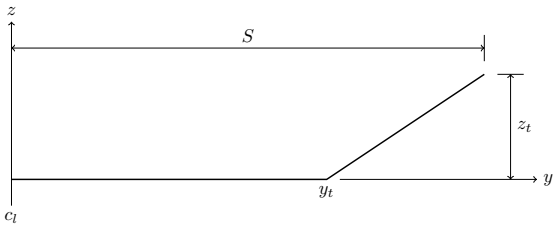


Figure 10: Basic Wing Dihedral Layout

The constraint in this figure is max_span. We are free to choose the other dimensions as long as we respect this limit.

7.1 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.

7.1.1 Arc Geometry

Since we start off with the chord and thickness specified as a percentage, 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:

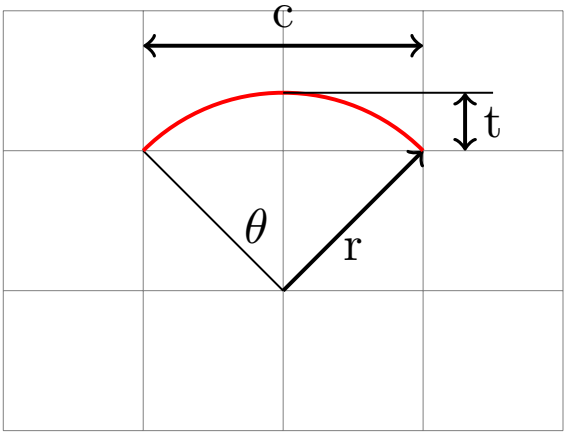


Figure 11: Circular arc Geometry

From this figure, we can write two equations:

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

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

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. Time for *SymPy* [?]!

7.1.2 SymPy

***SymPy* is a Python program that knows how to do a lot of math. It manipulates *symbols*, not numbers, and that is a lot of what you should have learned in a trigonometry classes.**

Here is how we get *SymPy* to solve our equations:

Listing 7: *python/circular − arc.py*

```
import sympy

r,c,t,theta = \
        sympy.symbols('r c t theta')

eq1 = 2 * r * sympy.cos(theta) - c
eq2 = r -r * sympy.sin(theta) -t

sol = sympy.solve([eq1,eq2],[r,theta])
print(sol)
```

Running this code, we find our solution:

$$\left[\left(c^2/8t + t/2,\ 2\,\mathrm{atan}\left(\frac{c-2t}{c+2t}\right)\right)\right] \tag{4}$$

***SymPy* actually displays those solutions like this:**

```
[(
  c**2/(8*t) + t/2,
  2*atan((c - 2*t)/(c + 2*t))
)]
```

It can also produce an output I can include in my documentation that looks much nicer, which is what you see above.

I asked *SymPy* to solve the two equations for the two *symbols* r and θ. The two solutions are shown. The first one is the radius we need to our code. We do not really need to worry about the angle part.

We will use these result to set up an *OpenSCAD* module that will build our ribs.

Warning! If there are any parents reading this, do not let your kids know about SymPy. On the other hand, if you cannot solve your kids math homework, you might try *SymPy* on the sly! YMMV!

7.2 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 uging a circular arc airfoil, these equations get reasonably easy to figure out. (If you have trouble following this math, ask your kid for help, or your grandkid!)

From figure ?? 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.**

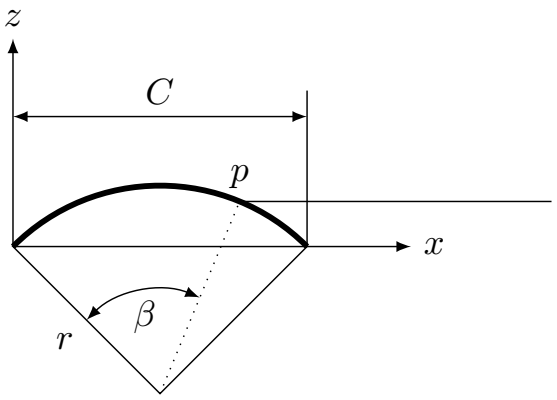


Figure 12: Wing Thickness Geometry

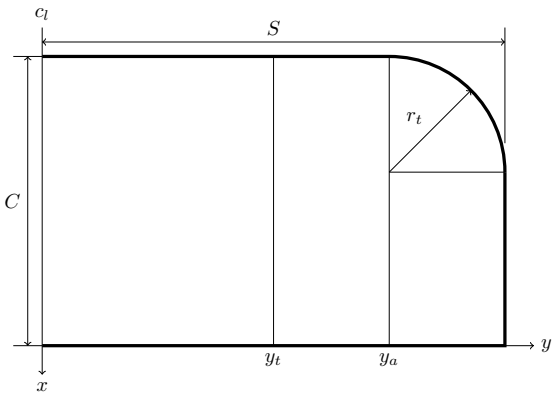


Figure 13: General Wing Geometry

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

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.

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

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 th tip. I will assume we can use a constantly decreasing camber across the tip, running from the inner airfoil value to zero at the tip. The functin we need to calculate the chord and camber at any point along the span liiks like this:

The general equation of or a circle centered at point c is:

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

This is simple enough to solve for y, but I did let *sympy* chew on it just to check:

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

These equations are being used in an OpenSCAD function when we need to figure out how high the wing is at any point.

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:

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

***OpenSCAD* supports *global variables*. This means you can define a variable at the top of a file, give it a value, then use that variable anywhere in the code. Unfortunately, that makes**

it hard to take your module to another program, since it depends on those variables. Writing your modules so they depend only on data they receive through parameters makes for better code.

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:

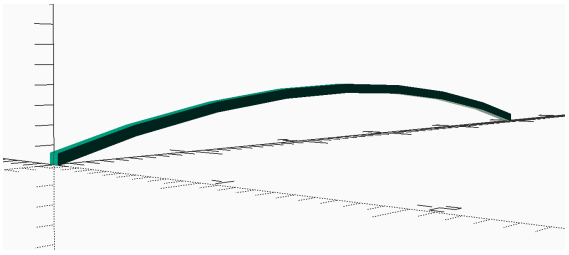


Figure 14: Basic rib

7.3 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 I packaged it in a simple module:

Here is this code:

Listing 8: *scad/wing_center.scad*

```
use <rib.scad>

module wing_spar(
    span,
    le_thickness,
    le_height) {
     cube([
      le_thickness,
      span,
      le_height]);
}

module wing_rib(
    chord,
    camber,
    thickness,
    height,
    le_thickness,
    te_thickness) {
  ribchord = chord
    - le_thickness
    - te_thickness;
  rib(
    ribchord,
    camber,
    thickness,
    height
  );
}

module wing_center(
    chord=5,
    span=12,
    nribs=5,
    camber=5,
    height=1/16,
    thickness=1/16) {

  // leading edge
```

```
  translate([0,-span/2,0])
    wing_spar(span, height, thicknes

  // place ribs
  for (n=[0:1:nribs-1]) {
    offset = -span/2 + span
        /(nribs-1) * n;
    translate(
      [thickness,offset,0]
    )
      wing_rib(
        chord - 2 * thickness,
        camber,
        thickness,
        height
      );
    }

  // trailing edge
    translate([
      chord  -2 * thickness,
      -span/2,
      0
    ])
      wing_spar(span, thickness, hei
}

wing_center();
```

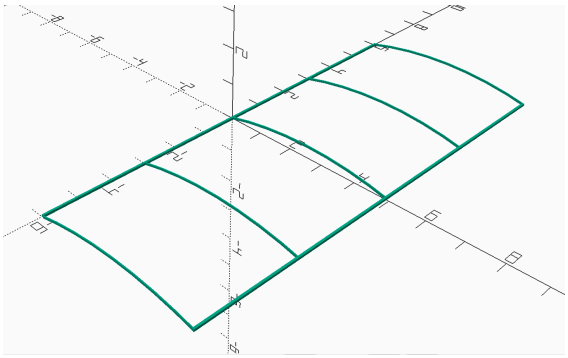


Figure 15: Wing Center Section

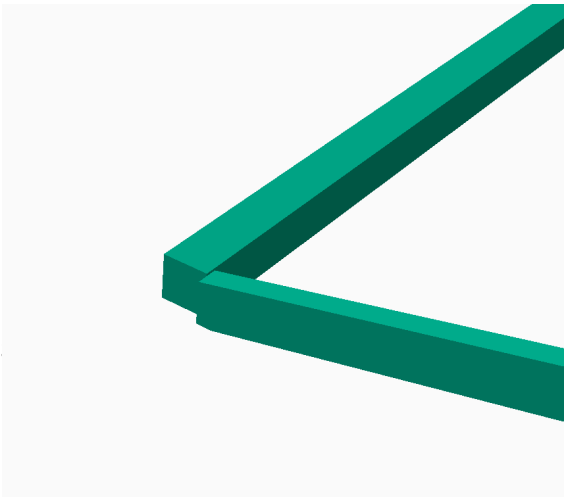


Figure 16: Tip Joint Detail

This code uses a loop to place the ribs. Details on all of these language constructs can be found in the *Users Manual* ??

Figure 15 shows the wing center section.

There is one detail in this image worth examining closer. Figure 17 shows the joint at the outer rib. This allows the tip section to mate at this joint.

Being able to zoom in on details like this is handy for making sure your design is clean.

7.4 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. By writing the tip as a module, I will be able to reuse this design on the stabilizer and the fin!

```
tip_section(span=3, chord=5, radius=2);
```

There is one interesting issue in the tip design.

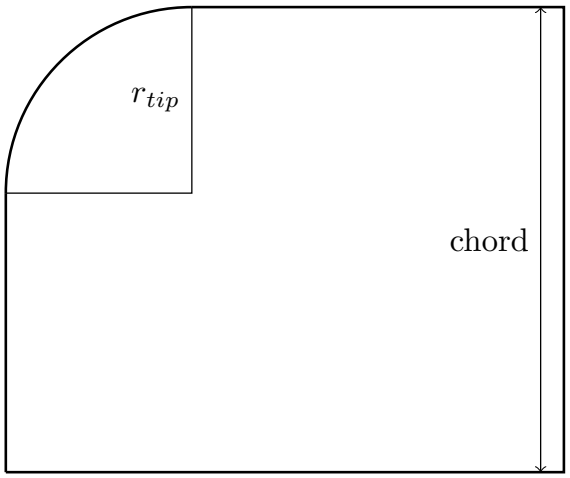


Figure 17: Tip Design

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.

It turns out to be easy. Once again, we use the *difference* operation. We generate a square strip the size we want, then set up a block longer than the strip and place it at a slight angle so we end up chopping off the unwanted material leaving a tapered strip. It is easy enough to do this twice to get the result we are after. This is sanding with no mess!

Figure 18 shows the basic idea.

Figure 19 shows the final tip section.

7.5 Tip Templates

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

Listing 9: *demo/printer – test.scad*

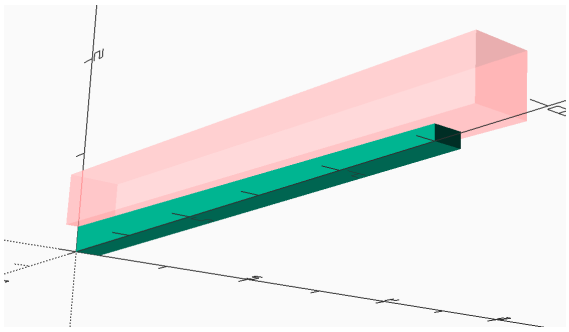


Figure 18: Trimming Block

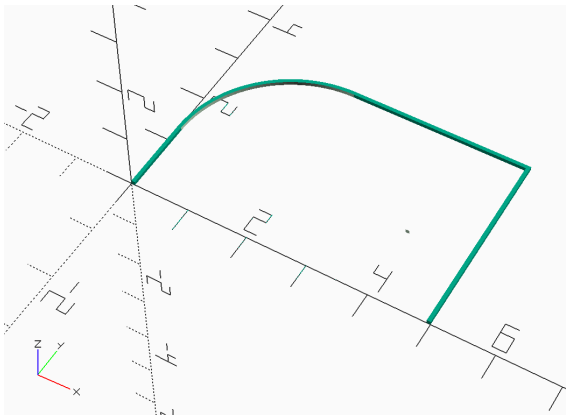


Figure 19: Wing Tip

```
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+prin
       wing_tip();
```

The *scale* operation is what sized the shape so it prints accurately. Most of the rest of the code just made adjustments so the template printed properly on my printer. These might need adjusting for different printers.

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 file which can be printed. I use my Chrome browser to do the printing on my home printer. Figure ?? shows what the template looks like in Chrome.

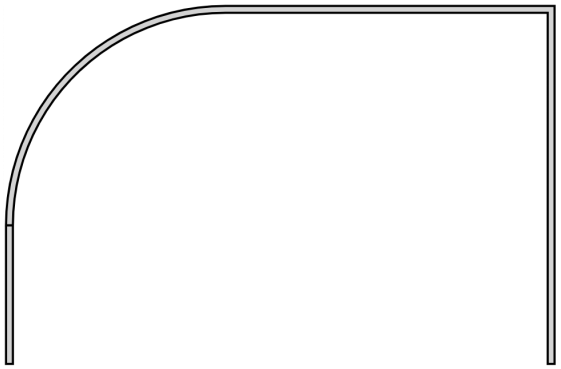


Figure 20: Tip Template

7.6 Wing Assembly

All we need to to to complete the wing is generate the center section, then generate the wing tips and rotate them into place at the proper dihedral angle. The only thing tricky here is generating the two tip sections.

Since the tip is flat, I can generate two of then, and rotate one 180 degrees so it will fit on the opposite side. I do need to do some minor translating to make sure things line up, but by now, this is getting easy!

Figurefig:wing.pngFinal Wing shows the completed wing.

8 Stabilizer and Fin

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

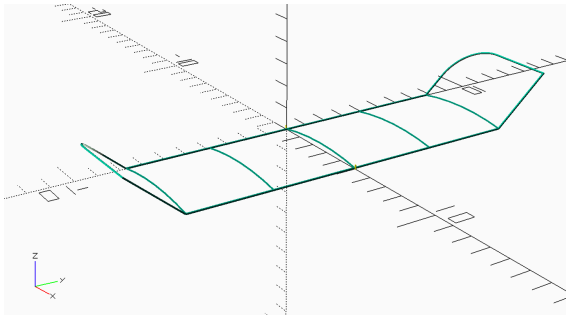


Figure 21: Wing Assembly

8.1 stabilizer

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

Figure 22 shows the result.

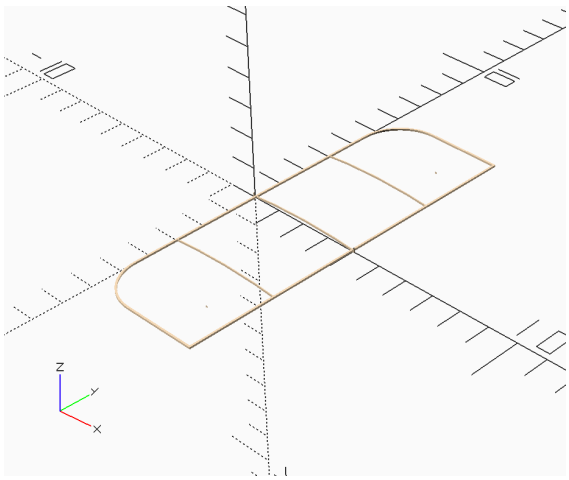


Figure 22: Stabilizer

8.2 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 di-

rectly to the tail boom.

Figure 23 shows the fin.

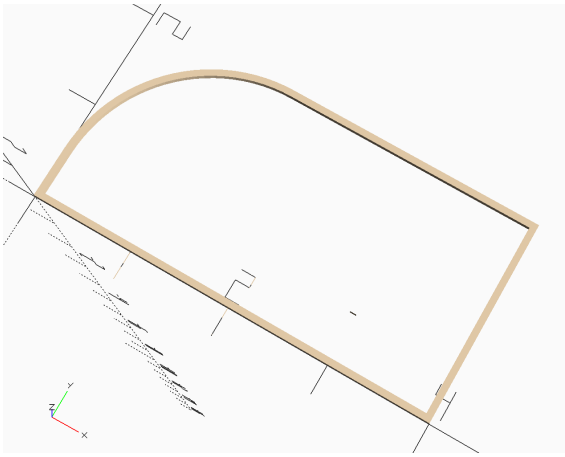


Figure 23: Vertical Fin

9 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:

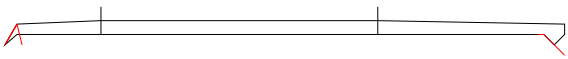


Figure 24: Basic Motor Stick

This design provides support for the front bearing 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.

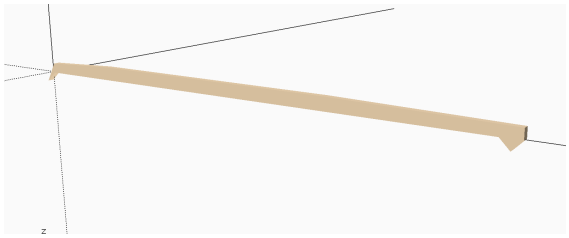


Figure 25: motor-stick.png

I added one new feature to this module. 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

```
\begin{lstlisting}
include <colors.scad>
```

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

10 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

11 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 code for this got fairly involved, so I will refer you to the project website for details.

```
module prop_blade(
        span =
    ) {
...
}
```

11.1 Blade Planform

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 26 shows the general layout.

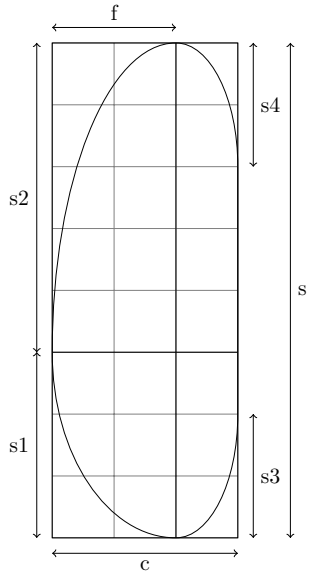


Figure 26: Prop Blade Layout

11.2 Prop Spar

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.

Here is what I ended up producing, after rotating it back upright.

12 Final Assembly

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

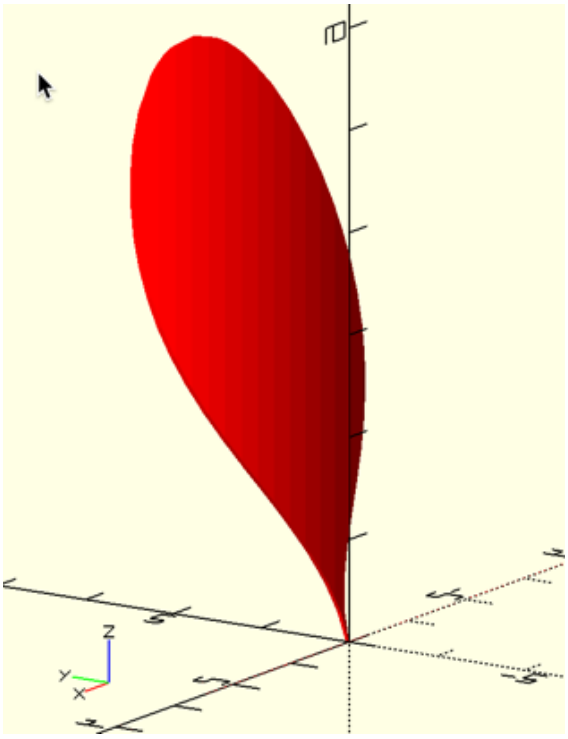


Figure 27: Prop Blade

see the completed model.

12.1 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, 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.

12.1.1 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:

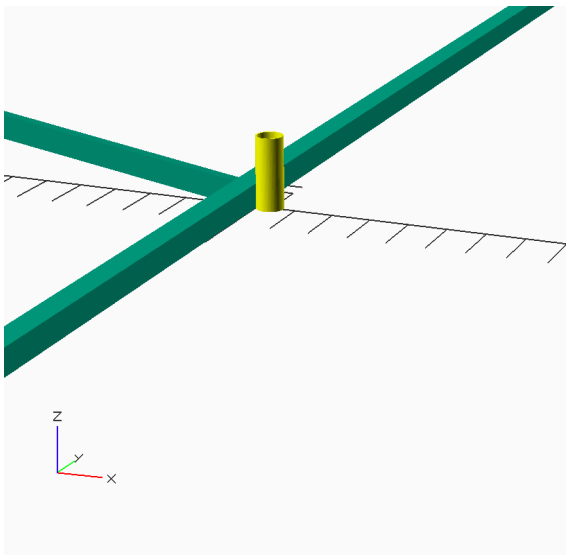


Figure 28: Wing Mount Tube

12.1.2 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:

12.2 Mounting Tail Group

Both the stabilizer and fin mount on the tail boom.

12.2.1 Stabilizer

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

To get started, we set up a few dimensions in the data file

12.2.2 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.

subsectionTransporting Your Model

13 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:

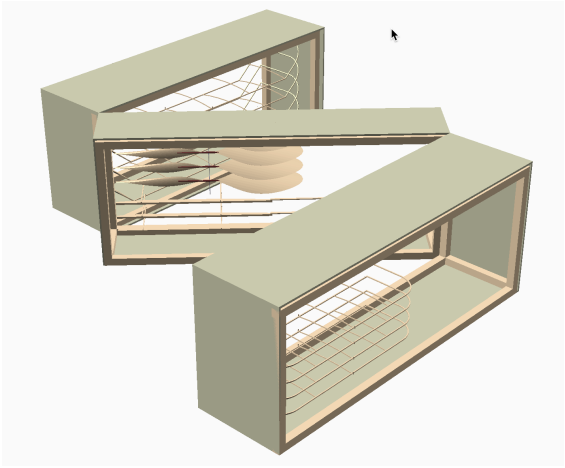


Figure 29: Flight Box

13.1 Basic Construction

As seen in figure 29, 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.

13.2 OpenSCAD Design

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!

14 Weight and Balance Analysis

14.1 Generating STL Files

Many folks who use *OpenSCAD* generate *STL* files from their design. 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 $that is a topic for another article$!

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 a nice Python library 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. I am working on that problem now, but am not ready to show any results yet.

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.

14.1.1 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!

At present, I am working on setting up a system to process all part files contained in the project directory so I can produce a report on the overall design.

15 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! I spent 20 years as an officer in the USAF, then 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

References

[] **Constructive solid geometry - Wikipedia, 2021. URL** https://en.wikipedia.org/wiki/Constructive{_}solid{_}geometry.

[] **OpenSCAD CheatSheet, 2021. URL** https://www.openscad.org/cheatsheet/.

[] **STL (file format) - Wikipedia, 2021. URL** https://en.wikipedia.org/wiki/STL{_}(file{_}format).

[] **R. Black. Math Magik LPP, 2021. URL** https://rblack42.github.io/math-magik-lpp/.

[] **S. Developers. SymPy, 2021. URL** https://www.sympy.org/en/index.html.

[] **M. Kintel. OpenSCAD - The Programmers Solid 3D CAD Modeller, 2021. URL** https://www.openscad.org/.

[] **R. Williams.** ***Building & Flying Indoor Model Airplanes.*** **Ron Williams, 2008. ISBN 978-0-615-20203-7.**