Indoor Model Aerodynamic Studies Using Python

Roie R. Black

# 

# Introduction

In the new Data Science profession, researchers are collecting large sets of data, often from Internet sources, and analyzing that data using an amazing array of Python tools. In my [*Math Majik*](https://rblack42.github.io/math-magik) project, which I presented in the 2021 edition of the NFFS Symposium (Black, 2021), I showed a bit of Python code that I used to estimate the model weight and center of gravity location for an indoor model design created using *OpenSCAD*. In this article, we will extend that work by adding tools that will help analyze the design and work toward estimating flight times for a design as well. I think you will be surprised at how easy data analysis can be if you use the modern tools that are hugely popular in today’s research world.

Rather than continue analyzing the *Limited Penny Plane* design I showed previously, I am going to base this current study on Gary Hodson’s Model of the Year *Wart-A6* (Hodson, 2010). Gary is a fellow club member in the [*Heart of America Free Flight Association*](https://kcfreeflight.org/), and he has graciously provided flight data and many conversations on his model and his record-setting flights with his design.

A picture containing windmill

Description automatically generated

I started this study after reading “*A Method for Predicting Indoor Model Duration*” by Doug McLean in the 1976 Symposium (Mclean, 1976). I also ran into a related study by Walter Erbach who produced a simple program that calculated the power required for an indoor model to maintain level flight. (Erbach, 1990).

Although predicting flight time, is a worthy goal, this present study is just a start toward that goal. My main purpose in presenting this study now is to demonstrate the power of modern tools very popular in the world of Data Science and show you how easy it is to get interesting results on your personal computer.

We will work through several aspects of aerodynamic research aimed at predicting flight times, introducing tools and techniques as we proceed. I will focus mainly on the aerodynamics of the airframe here, leaving details about the propeller and rubber motor to a later effort.

Let’s start with something simple, and introduce a powerful Python tool most programmers have never seen.

# Hacklinger’s Equation

One of the key assumptions in Doug’s method was based on an extensive study of indoor model aerodynamics conducted by Max Hacklinger (Hacklinger, 1964). After conducting mant flight experiments, Max presented this equation to estimate flight times for indoor models:

Where **He** is a constant “energy height” which Max set at 900 meters, **Wr** is the weight of the motor, is the average torque over the flight, and is the average flight prop speed (RPM) for the flight.

My first question was simple: Could this even remotely give us usable estimates. From Gary’s record flight, I have the launch torque, flight time, motor weight, and the number of turns expended during his record flight. I do not have an estimate of the torque averaged over the flight. I wonder what Hacklinger’s formula will tell me. Let’s plug in some numbers and see. (No, I am not going to do this manually, I am a computer geek! Time for some Python):

Text

Description automatically generated

I used the actual flight time in this code to calculate the average prop speed. I had to guess the average torque for the flight. Otherwise, this is just Hacklinger’s equation at work. The predicted time looks promising, so maybe we are on to something.

The listing above is a screenshot of my [*Jupyter Notebook*](https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/examples_index.html) page where I developed the material presented in this article.

*Jupyte*r is a very popular Python application thatcreates a web server on your local computer and launches your web browser giving you an environment where you can write text or code in a series of “cells”. Each page you create becomes part of your “notebook” where you document your research. The text cells are marked up using a simple notation called [*Markdown*](https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/Working%20With%20Markdown%20Cells.html). Code cells can be written in one of several different programming languages, but the default language is Python. As you write you can process each kind of cell to create either nice-looking documentation with equations, figures and anything else you need to explain what is going on or code you can run on the spot. You see the results immediately. If you make a mistake in your code or text, you can correct and reprocess with a mouse click. This is a great environment for doing quick analyses and experimenting with ideas. I use it to test snippets of code before pasting them into a real Python program.

What you create is something called *reproducible science* where anyone with access to your *Jupyter Notebook* pages can generate the exact same results. As part of this article, I have set up a copy of this study on a free cloud service called [*Binder*](https://jupyter.org/binder) that will let you work through this analysis using just your web browser. No setup required! Details are in the appendix.

What was that powerful Python tool I mentioned earler? If you look closely at the code above, you will see something interesting. The *import* lines let your code access tools from the named “package”. A package is a collection of Python cmponents you can use in your own code once the package is loaded on your system. There are tons of really useful packages available to Python programmers, many of which come with Python itself. In the first line, I am accessing the Python **pint** tool that lets you attach units to your numbers. **pint** will manage unit conversions for you automatically. This is a huge benefit when doing engineering calculations since you never really need to worry about entering ounces when you meant grams! (Ask NASA about units gone wrong on one of their Mars lander missions!) To use pint, you. Create a *UnitRegistry* which knows all about standard units and how to manage them. We use that registry gadget to assign units to our data, as you can see in the example code. Once set up, you never have to worry about what units you use, **pint** will handle conversions to make sure everything is consistent.

As it stands, I am not really happy with Hacklinger’s equation. There is no mention of the airplane at all. We could be flying anything with a rubber motor attached, which is hardly realistic. McLean attacked this issue by adding in a bit more real aerodynamics, so the results made more sense.

In the sections that follow, I will introduce several additional Python analysis tools, and present a bit more theory based on work I found by Walter Erbach, Bud Tenny, and others. I am still looking for adequate supporting data, so the results of this study are currently a work in progress.

# Basic Aerodynamics Research

Aerodynamics is the study of how fluids move, usually over or through some object. There are two approaches to studying this topic: experimental and theoretical. In the first, we use real fluids and try to probe the motion of the fluids to get measurements you can study. This is difficult since the act of probing can influence the flow.

In the theoretical approach, we turn to mathematics to try to understand what is going on. Long before computers were even developed, a lot of theoretical research into aerodynamics was going on.

In my first assignment in the Air Force, I worked with an amazing team of scientists who were developing techniques for using computers to generate data from the mathematics of fluid motion. We got access to the fastest supercomputers on the planet to do this work. Our research was called *Basic Research*, since it focused more on developing the tools needed, rather than trying to study some specific problem. One principle of this kind of research was very important to our work. In order to gain confidence in our new tools, we needed to demonstrate that they accurately modeled the real world by comparing our numerical results with real experimental data. The tools we were developing are now part of *Computational Fluid Mechanics*, which has revolutionized the design of just about anything involved with moving fluids – like Air!

While I would love to validate the research we are doing on indoor model airplane flight with real data, that is difficult. Not many researchers have worried about small vehicles flying at extremely slow speeds. That is changing with new interest in micro-vehicles, so some real-world data is beginning to appear.

Obviously, we are not going to power up a supercomputer to do this research. Instead, I am going to limit my work to a typical home computer. What amazes me is that code I wrote back in the mid-1970s in FORTRAN still runs on my Macbook laptop today. A typical 10-minute run on a mainframe back then runs in about 5 seconds today on my laptop. That is impressive, but supercomputers have become insanely powerful! I got a tour of a modern supercomputer recently that used 175,000 high-power chips to solve some extremely complex problems! Someday I want to run my code on that beast!

## Python Analytical Tools

All tools I use in this study can be freely installed on any home computer. With one exception, all the tools used in this study are written in *Python*. The one exception is *XFoil*, which was written in FORTRAN, but I am driving that program using Python! I will not cover installation details here. Check the project GitHub website at <https://rblack42.github.io/nffs-2022-symposium-live> for more information. I will introduce each tool as we work through example calculations.

I do not expect readers to be programmers, so the code listings may be confusing. Unfortunately, space limitations in this article prevent me from showing code the way I actually teach beginning programmers to write it in my classes. I have shortened up names so lines fit this publication. If you are really interested in reading the code, I suggest navigating to the project website and looking at the real code produced for this project. But, be warned that I intend to clean that code up after this article goes to the publisher, so details may be different.

## Experimental Data

We begin this study by looking at some experimental data on indoor model flights found in previous editions of the NFFS Symposium.

### Bud Tenny’s Flight Data

Bud Tenny, conducted an extensive study of indoor model flight paths in his 1968 Symposium article (Tenny, 1968) and presented a graphical technique for estimating flight time. Bud managed to get some nice data from actual flights of indoor models. Not much detail is provided on the model making the flights, but Bud shows torque curves, flight path data, and rpm data for the test flights. Let’s explore his findings and use Python to see what we can do with them.

Bud presented his results as a set of graphs in his symposium article. Staring at those plots is nice, but we need the data behind the plots to do much work with these results. The article included a bit of tabulated results, but most of the results are just graphs, so we seem to be stuck. Or are we?

I found a nice free desktop application called *WebPlotDigitizer* (Rohatgi, 2022)that lets you take an image of a plot and digitize it with a mouse on your computer. Once done, you can save your curve in a standard comma-separated variable file that you could load into your favorite spreadsheet. I am going to read that file using Python.

#### Display the Graph Image

Here is how we embed the original graph image in the notebook page you are writing:

A picture containing text

Description automatically generated

Chart, line chart

Description automatically generated

### Capturing the Data

In digitizing this curve, I zoomed the image up to full-screen, then used *SnagIt*, a screen capture program I have used for years, to capture the image. Digitizing with *WebPlotAnalyzer* involves defining the axis values to set scaling factors, then using the mouse to pick a series of points along the curve. The data produced is then saved as a CSV file ready to load into a *Jupyter* notebook page. To make the loading process easier, I created a Python function that takes a file name as a parameter and returns the curve data as two lists, one holding **x** values, and the other **y** values. Splitting these values this way makes things much easier to process with other Python tools we will see soon. Here is the loading function:

Text

Description automatically generated

The first two lines shown provide access to two Python tools that make this seemingly complicated task much easier! The **numpy** package provides powerful tools for manipulating data sets of all kinds and sizes. We only have a few points to deal with here, but **numpy** can handle huge sets of data! The **csv** package handles loading the CSV file. It can also be used to create CVS files from data you generate in your code.

Basically, the **get\_points** function code above reads the named CSV file line by line. I saves each point into two **numpy** arrays holding **x** and **y** coordinates of the points you digitized on the image.

*Jupyter* remembers calculations you set up in previous cells, so you can build up your calculations and test your code in short chunks. It is far easier to correct things this way. After we process a cell with a Python function, we can use that function in code we write in cells that come later.

#### Plotting the Data

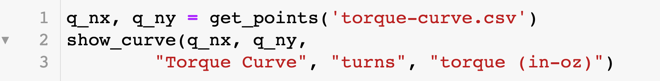
What does my digitized data look like? I am not especially interested in looking at a pile of numbers. How about a nice graph of my own? Time for another Python tool:

Text

Description automatically generated

The **matplotlib** tool provides plotting capabilities that help you visualize your data. Although we will only show simple plots here, **matplotlib** can generate very fancy displays of three-dimensional data if needed.

The display function created above accepts two point coordinate lists, a title, and labels for each axis and displays the result. This is a quick and easy way to see your data rather than looking at all those ugly numbers! Here is how we use these new functions:



Chart, line chart

Description automatically generated

The curve is flipped, since **matplotlib** thinks the numbers on the x-axis should increase. I can fix that later. In ancient times, researchers had to generate graphs like this manually!

I used these routines to process the propeller speed and flight path height graphs from Bud’s article. Here are those curves:

Chart

Description automatically generated

## Chart Description automatically generated

### Processing the Data

Once we have the data in digital form, we can do some amazing things. However, the data currently is just a pile of numbers. We can turn those numbers into a Python function by using a curve fitting scheme. Here is some code that will turn our numbers into a cubic spline function that we can use to get a data at any point along the curve:

Text

Description automatically generated

**scipy** is yet another useful Python tool that provides a set of standard mathematical operations you can use. In this case, we get a function back that we can use to get a value from the curve from any input value along the x-axis. The function will even return values beyond the range of your input data, but doing this extrapolation is risky. You might not like the results.

We create a curve fit function by passing the curve points to our fitting routine:

Text

Description automatically generated

Here, we flipped the values so they show turned expended, not turns left.

Now we can generate a new plot using as many points as we like:

Text

Description automatically generated

In this snippet, I create a sample set of 50 time values evenly distributed between the end point **x** values from my data, then feed that set of numbers to the curve fit function **q\_n** generated from the original data points. **numpy** gives back a new set of points that I can plot. It looks the same as the figure above, so I will skip showing the result.

I can use the curve fit function, which normally gives you torque for an input turn count, to get a turn count for some input torque. Here is an example of how this is done when we know the launch torque and want to find out what turn value on the torque curve would produce that torque:

Graphical user interface, text

Description automatically generated with medium confidence

Now, let’s find the vertical velocity of the model by differentiating the height function. This is a numerical differentiation since we do not really have access to an equation defining the curve.

Text

Description automatically generated

In this snippet, I created a new set of X values for the time axis. The next line takes the derivative of our function and returns a new function we can use. Finally, **numpy** applies the new function to our set of time samples and returns a set of velocity values.

Here is the graph:

Chart, line chart

Description automatically generated

This plot looks a bit choppy. The raw digitized data was not smooth, and we see that here. We will look at a way to “smooth” the data later.

Bud included some tabulated data on vertical speed. Let's see how this curve compares with his data:

Text

Description automatically generated

I entered Bud’s tabulated data here, then created a plot showing both the new curve and the tabulated data (shown as a “scatter” plot). I am happy with the results:

Chart, line chart

Description automatically generated

### Propeller Power

Let’s try something a bit more complicated. We have shown how to generate functions that represent our input data. I want to see the propeller power curve, but I do not have that data available. What I do have is propeller speed as a function of time, and torque as a function of turns. If I integrate the propeller speed function I can get turns as a function of time, then pass that into the torque function to get torque as a function of time. I can combine the results to get the power I want using this formula:

Where **n** is the propeller speed, and **Q** is the torque. Here is the code that does this math, again numerically:

Text

Description automatically generated

This looks complicated, but it follows the process described above. The “anitderivative’ is the opposite of the derivative, which basically gives us the result we are after. Here is the final curve

Chart

Description automatically generated

Along the way to this result, I produced the **ft\_t** function that produces turns as a function of time, which I used to generate a sample set of 50 points. Looking at that data, I was able to figure out how many turns were in the motor when it landed. Bud indicated that the test flight launched at 0.2 inch-oz of torque, which my function said was 1240 turns, and landed with about 500 turns remaining. This analysis said the flight used 749 turns, so it would have landed with 1240-749 = 491 turns remaining! Not bad!

## Experimental Airfoil Data

To analyze the Wart’s flights, we need some airfoil data. The wart uses a simplex airfoil for the wing and stab. Looking for suitable experimental data for this shape did not turn up anything useful. However, I was able to find some data for another common airfoil, the circular arc.

### Circular Arc Aerodynamics

Research into the flight of insects (Okamoto & Ebina, 2016) produced some test data that seems appropriate for this study. Here is a sample of that data.

Chart

Description automatically generated

Digitizing these curves produced more lumpy curves. Since I plan on creating a library of airfoil data for testing designs, I wrote a data management routine that lets me select an airfoil by name, then load coefficient data needed for analysis work. Here is an example of using that code:

Graphical user interface, text

Description automatically generated with medium confidence

That last line returns three Python curve-fit functions, one for each aerodynamic coefficient we need.

After digitizing data from this figure, here is the lift coefficient curve I produced:

Chart, line chart

Description automatically generated

This curve needs a bit of help. Another tool in the **scipy** package can help by smoothing the data.

### Data Smoothing

It is a fact of life that when you digitize data using your mouse, you will not produce the smooth curve you might be trying to capture. I discovered that trying to digitize small images with WebPlotDigitizer contributed to this problem. While you digitize points, you get an expanded view of the curve that helps you locate points more precisely. With a small image the the selection point jumped from pixel to pixel and I was not able to home in on the spot I was after. Blowing the images up to a higher full screen resolution helped, but the curves were still jittery. Time to try some data smoothing.

Basically, the smoothing we will use looks at a small set of points near each point on your curve and tries to fit a simple mathematical curve through that sample that has a minimum error from the point being considered. This is called a “least-squares” technique. Python s**cipy** has a routine that does this work.

Text

Description automatically generated

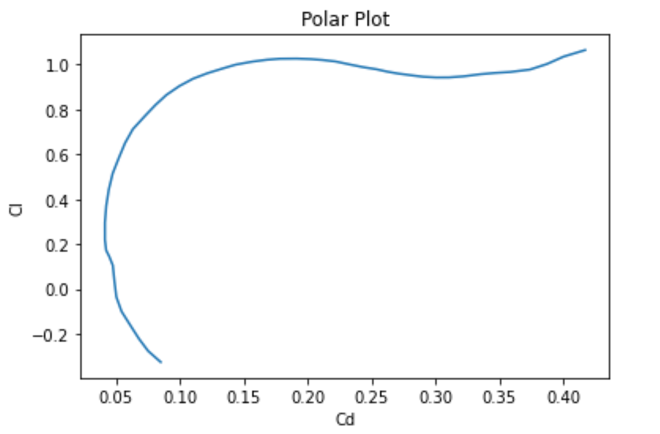
### Here is the result:

Chart, line chart

Description automatically generated

This looks better, but it is up to you to decide if you want to use this technique. I used this smoothing to clean up my bad digitizing work. Note that the smoothing operation did not smooth the function we generated, it smoothed the data produced by that function. If you want a smoothed function, you need to refit the smoothed data points. Good thing computers are fast!

From the new curve fit functions produced from these data sets, I created a Cl/Cd polar plot for this airfoil:



Next, let’s look at the overall airframe we will be studying.

# Simplified Indoor model

Here is the general layout of a typical indoor model.

Chart

Description automatically generated with medium confidence

This diagram also shows the forces acting on the model. The lift and drag forces are assumed to act at the aerodynamic centers of the lifting surfaces. For simplicity, these centers are located at the quarter-chord point on both surfaces. I will be locating all points shown in his diagram using actual dimensional data from Gary’s Wart plan in the example calculations.

### Locating the CG

The location of the center of gravity is important in our analysis, but that location is not shown in Gary’s plan. However, he does provide weight data for the major model components. A little Python code and we can calculate the CG using these formulas:

Where **Wi** is a set of component weights, and **xi,yi** are the corresponding locations of each component’s CG measured from any convenient reference point. I used the nose of the model for that point.

Here are the component weights:

Text

Description automatically generated

And, here are the associated arms we need:

Text

Description automatically generated

Finally, here is the code that calculates the CG:

Text

Description automatically generated

Using this scheme, my calculations produced:

CG is at x=4.05 inch, y=0.139 inch

Which puts the CG about an inch behind the trailing edge of the wing, right where Gary says it should be.

# Aerodynamic Forces

To take the next step in analyzing this model, we need to figure out the aerodynamic forces that are generated by the lifting surfaces. We have airfoil data we can use as a start. However, to use that data to generate the forces acting on the model, we need to know how fast the model will be flying. How can we figure that out?

## Level Flight

In level flight, all four major forces are balanced: lift equals weight, and thrust equals drag. We also need to balance the moments produced by those forces about the center of gravity. In stable level flight, the sum of all those moments must equal zero.

Once you launch the model, it will seek out a flight configuration where the forces try to balance. If you have not adjusted the trim settings properly, your airplane might stall, or head to the floor. Gary constructed the Wart with the wing aligned with the motor stick at 0 degrees incidence and the stab set at about -2 degrees incidence. The model certainly did not fly at zero degrees angle of attack, but what angle did it assume? That angle has nothing to do with the angle the airplane assumed during the climb. The angle we are talking about here is the angle relative to the actual model flight path through the air. During the climb, the model may head up at 30 degrees, but if you watch, the model is angled up some amount as well. The angle of attack during the climb is the difference between those two angles, which is not 30 degrees. Since the lift and drag forces are determined by that angle of attack, we again seem to be stuck in our calculations. How can we figure this out?

In McLean’s method, he sidestepped this whole issue and set the lift coefficient of the wing at 1.0, then calculated the stabilizer lift coefficient needed to make the moments sum to zero. He ignored the drag forces in his calculations. Those assumptions certainly simplify the calculations, but again, we are leaving out the actual model.

Let’s try a different approach that uses available airfoil data

### Balancing Lift and Weight

Walter Erbach published research into the power needed to maintain level flight in a 1990 Symposium article (Erbach, 1990). He included code that implemented his technique in Basic for home users of the Commodore-64. (That should tell you something about how old this study was!) Walt originally wrote his code in FORTRAN and translated it into Basic. I converted his Basic code to Python and verified his results. That code is included in this project on my GitHub account, but we will not explore it here. Instead, we will consider how he came up with his results.

Walter used a model configuration much like the simplified design I am looking at here. He used a different airfoil, a McBride-B7. He found lift and drag coefficient data in an old Frank Ziac Yearbook. Walt’s code generated an angle of attack survey, and he processed “terrifying yards of tabulated data” by hand into a set of nice graphs showing what was going on.

His calculations were based on finding the model configuration that would produce enough lift to balance the model weight. He assumed that both wing and stab contributed to the total lift in proportion to their surface areas. Given the model weight, he was able to determine the lift coefficients required to balance that weight. This gave him the required angle of attack for the airfoils. He then used the definition of the lift coefficient to calculate a flight velocity.

Once he knew the forward velocity of the model, Walt calculated the required power needed to maintain that lift by calculating the drag force. The required power had to balance this drag.

Walt used the same airfoil on both the wing and stab, and I have experimental data for an airfoil I can use as well.

## Aerodynamic Coefficients

The aerodynamic coefficients for an airfoil are defined as follows:

Text

Description automatically generated

From this equation, we can find the lift:Text

Description automatically generated

Similarly, we get these equations:

Text

Description automatically generated

And:

Text

Description automatically generated

In these equations, **S** is the surface area of the wing or stab, **c** is the chord. **ρ** is the density of the air and **u** is the flight velocity.

To calculate the forces, we need data on the air we will be flying through!

### Standard Atmosphere

Rather than rely on local measurements of the properties of air, most researchers use a model atmosphere so they can compare results. We will use data from the 1976 Standard Atmospheric Model, which you can find at the [Standard Atmosphere Calculator](https://www.digitaldutch.com/atmoscalc/). The website at that link lets you set your elevation and it presents you with standard values for the important properties we will need.

The Python **fluids** package implements this model, so we do not have to copy and paste numbers from the web. However, that package does not handle the units for those numbers, so I created a wrapper routine that adds units and returns four basic properties for any specified elevation:

* Temperature
* Pressure
* Density
* Dynamic Viscosity

Text

Description automatically generated

The code I wrote is in a package in my project. The first line provides access to that code. Th **Air** function is the place where property data can be accessed. If you look closely, I sent **u** as a parameter to the **Air** function. This is my **pint** *UnitRegistry* created earlier. **pint** only works properly if all code in a project uses the same registry.

Here is the output:Text

Description automatically generated

We now have all the air property data needed to do a sample calculation. As I was pondering this next step, I decided to try to get data for the actual Wart airfoils. Another free tool can solve this problem!

## Generating Airfoil Data

Generating airfoil data using computer-based tools is a common issue. Fortunately, there are many tools we can use. Unfortunately, many of those tools are written in other programming languages. For this study, I used a program that is very popular among model airplane designers: *XFoil*. This program was originally developed by Mark Drela at MIT as part of the [Daedalus Project](https://en.wikipedia.org/wiki/MIT_Daedalus) (Drela, 2022) that built a man-powered airplane that flew over 72 miles in 1988!

A picture containing sky, outdoor, plane, aircraft

Description automatically generated

This sure looks like an overgrown indoor model to me!

*XFoil* was written in *FORTRAN* and released into the public domain. The program can be compiled on any system with a modern *FORTRAN* compiler. I use the Free Software Foundation’s **gfortran** compiler on my systems.

I found an interesting project on GitHub created by engineers at the by the DARcorporation (DARcorporation, 2022) that packages *XFoil* so it can be run from a Python program. Getting that running was not something I recommend to beginners, but I am working on that. I will show the results of my work here and refer you to the project website for updates.

In order to use *XFoil*, we need to create airfoil data files describing the airfoil we want to study. The Wart uses a simplex airfoil, so I created some code that generates this airfoil in a form suitable for processing with *XFoil*.

The basic shape was generated with code I found in an Excel spreadsheet (author unknown). Here is that code:

Text

Description automatically generated

The thin line produced by this code is not suitable for use in *XFoil*. Instead, I made it slightly thick and added a round leading edge and a tapered trailing edge. These modifications are based on work from a Master’s Thesis by Michael Reid (Reid, 2006). Michael explored thin airfoils with a reflexed trailing edge using *XFoil*. Here is an example of the final airfoil I created:

A picture containing text, screenshot, weapon

Description automatically generated

Using this shape and the Python wrapper, I was able to generate aerodynamic coefficient data for the Reynolds numbers we are interested in and at angles of attack suitable for this study. Here is an example lift coefficient curve created with *XFoil*:

Chart, line chart

Description automatically generated

Sadly, we cannot validate this curve with real experimental data, but it will still be useful in our study.

### Erbach’s Level Flight Speed

With better aerodynamic coefficient data nowavailable from *XFoil*, let's try Erbach's calculations to get an initial flight speed.

I used the smoothed curve fit system to generate the coefficients for the wing and stab for the model flying at 5 degrees angle of attack as a test. I will not show that code here. Instead, here is the final calculation of flight speed:

Text

Description automatically generated

This code gives this flight speed:

3.3150814453086537 mile\_per\_hour

This is a little faster than Gary estimates for his model. It is also not a valid speed, since we have missed some considerations in our calculations. However, this number can be used to get an initial estimate of the Reynolds Number for our model.

### Reynolds Number

An important quantity in aerodynamic work is the *Reynold's Number*, a non-dimensional value that relates the viscous forces to the inertial forces working on a surface. This number is commonly used to characterize the type of airflow a vehicle might experience. For our indoor models, this number will be low, meaning that the flow near the surface of our flying surfaces should remain laminar. That means we do not need to worry about turbulence near the surface which greatly simplifies analysis.

The definition of this number is:

Text

Description automatically generated with low confidence

Where **L** is a reference length, usually the mean chord of the surface, and **μ** is the dynamic viscosity of the air.

Here is the code that calculates this number:

Graphical user interface, text, application

Description automatically generated

It is nice to see that **pint** found this to be a non-dimensional number!

When designing a model from scratch, we do not know what its flight speed will be, so picking a Reynolds number for analysis is a guessing game. The airfoil data I used here was for Re=3000, but that flight speed estimate indicated it should be closer to Re=4500. Obviously, we could home in on a number that matches, but we are not done with the analysis yet! We do not know if the model can actually fly at this speed. We need to add in the moment calculations and see what they say!

### Erbach’s Moment Calculations

Erbach set the wing and stab incidence, then conducted an angle of attack survey to generate the needed data. At each angle of attack, his code calculates the required level flight speed, then the lift and drag forces for the wing and stab. He then calculates the resulting pitching moment using these forces. He plotted the pitching moment curve looking for the zero point, which is where the model would fly level.

Since we are doing these calculations numerically, I set up a coordinate transformation system that calculates moments in a coordinate system centered on the center of gravity and rotated so the body is located at the desired angle of attack. I packaged this code in a simple routine that takes the CG location and the rotation angle we want as parameters, together with the coordinates of a where a force will be applied. The routine returns the position of that point in the transformed coordinate system.

Here is the code that generates the basic aerodynamic forces:

Text

Description automatically generated

Now, we calculate the moment arms at the specified angle of attack, using the coordinate transformation function.

Text

Description automatically generated

Finally, we calculate the moments produced by these forces:

**Text

Description automatically generated with medium confidence**

The signs of these terms are chosen to conform to the convention that positive moment results in the nose pitching up.

We still need to add the airfoil aerodynamic pitching moment contributions:

Text

Description automatically generated

According to these calculations, the final moment is

-0.8581620444255844 gram inch

This says the model will be pitching nose down in this configuration, so it will not be flying at this angle!

Erbach conducted an angle of attack survey to find the angle of attack where the model would be stable with a pitching moment of zero. We can do such a survey easily with our current data. I wrapped up all the code shown above into a **moment** procedure that takes an angle of attack as a parameter. It returns the moment and level flight velocity. Here is the code that does the survey:

Text

Description automatically generatedand here is a plot of the result:

Chart, line chart

Description automatically generated

According to this graph, our model will be happy at about 4 degrees angle of attack.

At this angle of attack, the velocity would be:

3.4701931619694575 mile\_per\_hour

This speed is still too high. Our lift and drag forces are still wrong. The actual wing does not generate lift based on just two-dimensional aerodynamic coefficients. We need to consider what happens with a full three-dimensional wing.

## Induced Drag

When air flows over a lifting surface several interesting things happen. One is the formation of two vortices that appear behind the wing tips. Although it is common to talk about the formation of these vortices as a result of the pressure difference between the upper and lower surfaces, their formation is more complicated than that. (For a good discussion of this, see [Mclean (2005)](http://www.smartcockpit.com/docs/Wingtip_Devices.pdf).

A side effect of those vortices is a downward deflection of the airstream behind the wing that influences the effective angle of attack of the stabilizer. This downward flow is called *Downwash*.

In initial designs, the downwash angle can be estimated using this equation:

**Text

Description automatically generated**

Where Aw is the aspect ratio of the wing. The effective angle of attack of the stabilizer is reduced by the downwash angle.

We can include this effect in our calculations by using the downwash angle to adjust the stabilizer angle of attack. I modified the moment function used above to adjust the stab angle of attack and ran the survey again. Here is the new plot:

Chart, line chart

Description automatically generated

Now, the model needs to be flying at about nine degrees angle of attack to generate a zero pitching moment. The new flight speed is:

2.9389394686987895 mile\_per\_hour

Or

4.310444554091559 foot/second

We are closer to the speeds Gary estimated for his airplanes. We can easily find the power required to maintain this flight speed, but I will not include that here.

To go further in this analysis, we need to start considering the propeller, which is complex enough to warrant a separate study. (I have already started to work with *QProp* (Drela N. , 2022), another program developed by XFoil creator Mark Drela. We will see what that program has to offer in a later effort.

# Next Steps

We have reached a good stopping point for this article. My intent was not to produce a high-quality assessment of the Wart design but rather to demonstrate how you can easily use Python tools to do such an analysis. I am not done with this work, and my website will be updated as I add more analysis techniques to this project.

have found several open-source tools from the world of *Computational Fluid Dynamics* that might help in analyzing a complete model. My plan is to apply available CFD techniques to this study to help predict performance using only the design data I generate using *OpenSCAD*, and theory backed up by some solid analytical tools. All of this work will be run on my laptop and tested on Windows and Linux to ensure that anyone can use the same tools to conduct their own studies. Everything I produce will be available on my *GitHub* account.

# Conclusion

If you are interested in doing any analytical research on aspects of our hobby, I think learning a bit about Python *and Jupyter* can significantly improve your work. I am quite pleased with how easily you can experiment with ideas and document your successes and failures for future reference using the tools I have presented here. To get started, you can use this study as a guide in doing your own research work. You can easily download a copy of my entire study from *GitHub* with a single command and have everything installed on your system, assuming you have some basic tools installed. Remember, if you just want to browse my code, head to: <https://github.com/rblack42/nffs-2022-symposium-live>.

I welcome comments, criticisms, and suggestions that might help improve this effort. Understanding flight has been a life-long passion for me, and I have no intention of setting that aside. I am already working on the next step in this Math-Magik project.

You can contact me at roie.black@gmail.com.

## Appendix

### Brief Guide to Using Jupyter

*Jupyter* is an interesting tool. It is a great place to play and even learn how to program. In this short introduction I am assuming that you have managed to install J*upyter* on your system.

Launching *Jupyter* is usually done from the command line, which may not be familiar to many of you. This simple environment is available on all systems and presents an interface we older programmers grew up in. No graphics, no mouse, just typing in commands and getting text results on a boring black and white window. To get to this environment on a PC just enter **CMD** in the search box at the lower left of your screen, then select the *Command Prompt* choice that will pop up. From there you just use simple commands to navigate to the folder you want to use for your project. More details on navigating the command prompt are at the project website, so I will not cover that here.

Move to the folder where your project is located and type **jupyter notebook**. After a few seconds, your web browser should launch and you will see a view of that project folder.

This is what mine looks like:

Graphical user interface, text, email

Description automatically generated

If you look closely (I know it is small), you will see files with a **.ipynb** extension. These are notebook pages I created for this study. Clicking on one of those will open that page. To start a new page, you select New, then Python3 from the menu near the top right.New pages are untitled by default. You can rename them using the File menu.

Basically, you enter either code or text into blocks on a page. When you are in a block, the screen shows a green line on the left edge of the block. The type of block is displayed on the menu at the top. You can change the type if necessary. If the block has already been processed the green line will not appear. Double-clicking in the block will show you the unprocessed text. You should select only code or markdown as a start. In code blocks you type in Python code lines. (Look up any tutorial on Python to get started). In *Markdown* blocks you just type in text. Lines will automatically fold as you get to the end of the block. Paragraphs are set off using blank lines. There are special formatting marks you can add to style the text in interesting ways. I recommend looking at the pages I created as a start or look for *Markdown tutorials* online.

Once you have a block set up the way you want, type *Shift-Enter* to process that block and create a new block below (or move to the next block if one is already there.

You can add images, and equations, but we will not cover that here. This is just a quick guide to get you started. Google *Jupyter Turorial* to learn more.

### Accessing The Online Study

If you want to play with the project code without needing to install anything on your personal computer, you can browse to a live copy of the *Jupyter* Notebook pages I created. Open up your web browser and navigate to <https://mybinder.org/v2/gh/rblack42/nffs-2022-symposium-live/HEAD>. This page will take a while to load up because the **mybinder.org** site creates a virtual machine for you as you connect, then installs all the tools I used in my development. Finally, it copies my project code into this virtual machine and starts up a web server presenting you with a live copy of my project pages. You can play with this as you like, but all changes you make (if any) will be lost when you close your browser. You are working on a private copy of my project code, not the real code. This is almost the same environment I use on my laptop!

When the page finally loads you will see a navigation panel in the left side of the page. Select the **book** folder to start exploring my notebooks. The entire set of pages here have been processed into a static (not interactive) website that is available from <https://rblack42/nffs-2022-symposium-live>.

# Bibliography

Black, R. (2021). Designing an Indoor model Using OpenSCAD. *NFFS Symposium*, 54-60.

DARcorporation. (2022). *xfoil-python*. Retrieved from https://github.com/DARcorporation/xfoil-python

Drela, M. (2022, 02 10). *XFoil - SUbsonic Airfoil Development System*. Retrieved from XFoil: https://web.mit.edu/drela/Public/web/xfoil/

Drela, N. (2022). *QProp - Propeller/Windmil Analysis and Design*. Retrieved from http://web.mit.edu/drela/Public/web/qprop/

Erbach, W. (1990). Calculating Indoor Flight. *NFFS Symposium*, 79-85.

Hacklinger, M. (1964). Theoretical and Experimental Investigation of Indoor Flying Models. *Journal of the Royal Aeronautical Society*.

Hodson, G. (2010). Gary Hodson's Wart-A6. *NFFS Symposium*, 172-173.

Mclean, D. (1976). A Methof for Predicting Indoor Model Duration. *NFFS Symposium*, 54-60.

Okamoto, M., & Ebina, K. (2016). Effectiveness of Large-Camber Circular Arc Airfoil at Very Low Reynolds Numbers. *Transactions of the Japan Society of Aeronautical and Space Sciences*, 295--304.

Reid, M. (2006). *Thin/Cambered/Reflexed Airfoil Development for Micro-Air Vehicles at Reynolds Numbers from 60,000 to 150,000.* Rochester Institute of Technology Master's Thesis.

Rohatgi, A. (2022). Retrieved from WebPlotAnalyzer: https://automeris.io/WebPlotDigitizer/

Tenny, B. (1968). CHoice of Rubber Motor for Low Ceiling Indoor. *NFFS Symposium*, 77-82.

# Biography

A person holding a toy

Description automatically generated with low confidence

In 1956, I was on a quest to figure out how airplanes fly. I was 10 years old!

I was already building model airplanes, mostly control-line Sterling models. That year Paul Poberezeny, founder of the *Experimental Aircraft Association*, published an article in *Mechanics Illustrated* that showed how to build a real airplane at home: the *Baby Ace*. I discovered that building models could lead to building full-sized airplanes.

As a kid growing up in Washington D.C., much of my spare time was spent wandering in the halls of the Smithsonian where many famous airplanes were on display. Shortly after Paul's article was published I found myself in the offices of the *American Aviation Historical Society* in the Arts and Industries Building of the Smithsonian. While there I was introduced to noneother than Paul Poberezeny himself, who was standing in the hallway with Dr.Paul Garber, Curator Emeritus of the *National Air and Space Museum*. Wow! I met two important men in the history of aviation, and Iwas just imagining a career in aviation!

My passion for flight took me through college where I earned degrees in Aerospace Engineering, then into my career as an officer in the USAF. My first assignment in the Air Force involved conducting research in the emerging field of *Computational Fluid Dynamics*. While working there, I was introduced to first-generation supercomputers, and my career shifted direction into Computer Science. I was next invited to teach at the Air Force's graduate engineering school. While this was going on, I was active building radio control models, and getting my Private and Commercial pilot’s licenses.

During most of my career, I was involved with advanced computers and programming languages. As my Air Force career neared its end, I was assigned as Deputy Director, then Director of a research supercomputer center in New Mexico. After retiring, I started a second career, again teaching Computer Science in Texas, and finally retired for good in 2018.

Since then, I have become a member of the *Heart of America Free Flight Association*. I am again active in building and flying rubber-powered free flight models. I am right back where I started all those years ago.

Currently, my wife and I are enjoying our retirement and finding time to pursue hobbies we love. My goal now is to continue my teaching, combining both of my career paths by building computer tools to assist in designing, building, and flying model airplanes. My goal is to help introduce another generation of folks to this fascinating world of flight.