Predicting Indoor Model Flight Times 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) [1] project, which I presented in the 2021 edition of the NFFS Symposium [2], I showed a bit of Python code that can be 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 help estimate flight times for a design as well. I think you will be surprised at how easy model 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* [3]. Gary is a fellow club member in the [*Heart of America Free Flight Association*](https://kcfreeflight.org/) [4], 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, outdoor object

Description automatically generated

(Image: i1-hodson-wart.jpg, “Hodson Wart A-6”)

I started this study after reading “*A Method for Predicting Indoor Model Duration*” by Doug McLean in the 1976 Symposium [4]. 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. [5].

Although predicting flight time, is a worthy 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. Most of this analysis will focus on 2D aerodynamics. We will leave a 3D study for another time.

If you have some experience with programming, the example code included here should be easy enough to follow. Those with no programming experience can find many tutorials on Python online. (Google “Python Cheatsheet” for some short overviews). All code shown in this article is available online for those wishing to go deeper into this project.

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 [6]. After conducting many 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

(lst1.png, “Hacklinger’s Time Estimate”)

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 [8] 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) [8] 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 earlier? 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* components you can use in your own code once the package is installed on your system. There are tons of useful packages available to *Python* programmers, many of which come with *Python* itself. In the second line in that listing, 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 your calculations needed 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 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. Let’s see what *Python* can offer in this kind of analysis.

## 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 teach beginning programmers to write their programs in my classes. 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.

## Atmospheric Properties

Since we are using pint to handle unit conversions, let’s see how we can easily get atmospheric properties for our calculations. In most research projects the *Standard Atmosphere* is used to define air properties. This allows researchers to share data without worrying about local weather conditions.

There is a convenient website at [*https://www.digitaldutch.com/atmoscalc/*](https://www.digitaldutch.com/atmoscalc/)where you can enter your site elevation and get air properties you can use. Fortunately, there is also a *Python* package named **fluids** that you can install as well. Unfortunately, **fluids** does not handle units, so I created a *Python* package named **StdAtm** that adds units for the properties we need in this study. I have created my own Python package for the code in this project named **mmtime**. Here is an example of using that wrapper:

Graphical user interface, text, application, email

Description automatically generated

(lst2.png, “Standard Air Properties”)

## 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 [7] 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 those flights, but Bud shows torque curves, flight path data, and rpm data for his 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 useful work with these results. The article included some 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* [8]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* (CSV) 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

(lst3.png)

And here is the image we will digitize:

Chart, line chart

Description automatically generated

(img2-tenny-torque-curve.png, “Tenny’s Torque Curve”)

### Capturing the Data

In digitizing this curve, I zoomed the image up to full screen, then used *SnagIt*, a screen capture program, 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 from the curve. 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

(lst4.png, “Load a CSV Curve Point File”)

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 receives the name of a CSV data file to process. It reads the file line by line and saves each point into **numpy** arrays holding **x** and **y** coordinates of the points you digitized on the image. It returns those two arrays for use in your code.

*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:

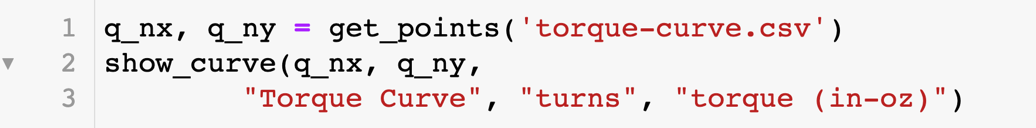
Text

Description automatically generated

(lst5.png, “Simple Plotting Routine”)

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 **show\_curve** display function created above accepts two point coordinate arrays, 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:



(lst6.png, “Using The Plot Routine”)

And here is the plot showing what we produced:

Chart, line chart

Description automatically generated

(img3-digitized-torque-curve.png, “Digitized Torque Curve”)

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

(img4-digitized-prop-speed.png, “Digitized Prop Speed Curve”)

Chart

Description automatically generated

(img5-digitized-flight-profile.png, “Digitized Flight Profile Curve”)

### 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 data at any point along the curve:

Text

Description automatically generated

(lst7.png, caption: “Curve Fit Routine”)

**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

(lst8.png, “Generating Torque Curve Fit”)

Here, the second line flips the turn values so they **ql\_nx** now is turns expended, not turns left. **numpy** makes processing lists of numbers easy!

**q\_n** is a new function we can use to get torque for any value of turns at any point alone the curve. With it, we can generate a new plot using as many points as we like:

Text

Description automatically generated

(lst9.png, “Display the New Torque Data”)

In this snippet, line 1 created a sample set of 50 turn values evenly distributed between the end point **q1\_nx** values from my data, then feeds 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

(lst10.png, “Find Turns for Given Torque”)

Now, let’s find the vertical velocity of the model by taking the derivative of the height function. This is a numerical differentiation since we do not really have access to an equation defining the curve. I followed the same process outlined above to get a curve fit function **fh\_t** for height as a function of time. The differentiation gives you back a new function which we saved as **fdh\_t**.

Text

Description automatically generated

(lst11.png, “Generating Vertical Velocity”)

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

(img6-vertical-velocity.png, “Generated Vertical Velocity Plot”)

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

(lst12.png, “Comparing Vertical Velocity Plot to Tenny’s Data”)

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

(img7-vertical-velocity-samples.png, “Validating Vertical Velocity Data”)

### 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:

Text, logo

Description automatically generated with medium confidence

(eq1-power-equation.png)

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

(lst13.png, “Generate Power vs Time”)

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

Chart

Description automatically generated

(img8-power-function.png, “Power vs Time”)

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 [9] produced some test data that seems appropriate for this study. Here is a sample of that data:

Chart

Description automatically generated

(img9-3%-arc.png, “3% Arc Experimental Data”)

Digitizing these curves produced more lumpy curves.

### 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 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

(lst14.png, “Smoothing the Lift Curve”)

### Here is the result:

Chart, line chart

Description automatically generated

(img10-smoothed-lift.png, “Smoothed Lift Coefficient Curve”)

This figure shows both the original data and the smoothed data. 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:

Chart

Description automatically generated

(img11-smoothed-polar.png, “Smoothed Polar Curve”)

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

# Simplified Indoor model

Now that we have a way to get aerodynamic data for our airfoils, we need to look at the force arrangement on a typical model. Here is the general layout of those forces:

Chart

Description automatically generated with medium confidence

(img12-simplified-model.png, “Simplified Indoor model”)

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:

Text

Description automatically generated

(eq2-cg-equations.png)

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

(lst15.png, “Wart Weight Data”)

And here are the associated arms we need:

Text

Description automatically generated

(lst16.png, “Wart CG Moment Arms”)

Finally, here is a Python routine that calculates the CG:

Text

Description automatically generated

(lst17.png, “CG Calculation Routine”)

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

McLean assumed that the energy used in climbing is about equal to the energy used in descending. Therefore, we can consider only level flight in doing our analysis. This simplification greatly eases our task in predicting flight times.

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.

In McLean’s method, he 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 this calculation. 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 [5]. He included code that implemented his technique, written 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 *McBride-B7* airfoil data he found 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 angle of attack 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.

## Aerodynamic Coefficients

The aerodynamic coefficients for an airfoil are defined using the following equations:

Text

Description automatically generated

(eq3-lift.png)

Text

Description automatically generated

(eq4-drag.png)

Text

Description automatically generated

(eq5-moment.png)

In these equations, **S** is the surface area of the wing or stab,  **c** is the chord length **ρ** is the density of the air, and **u** is the flight velocity. The aerodynamic coefficients are dimensionless numbers that vary with angle of attack and other factors we will need to consider.

As I was pondering the next step in this analysis, I decided to try to generate data for the actual Wart simplex airfoils. Another free tool can solve this problem!

## Generating New Airfoil Data

Generating airfoil data using computer-based tools is a common problem. 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) [13] that built a man-powered airplane that flew over 72 miles in 1988!

A picture containing sky, outdoor, beach, aircraft

Description automatically generated

(img15-daedalus.png, “Man-Powered Daedalus”)

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* [11] 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

(lst18.png, “Simplex Airfoil Function”)

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 [12]. 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

(img16-simplex airfoil.png, “Test Simplex Airfoil”)

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

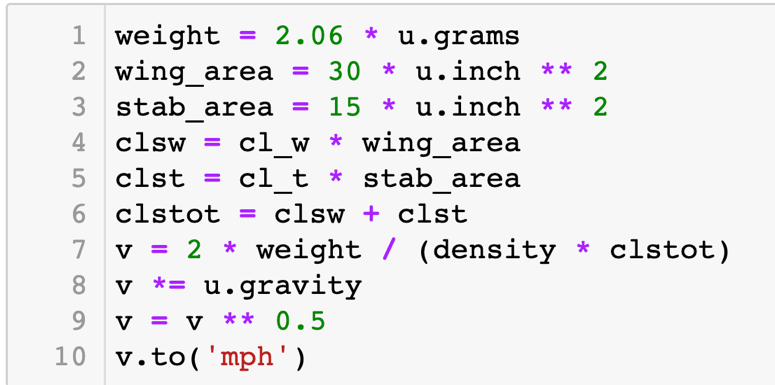
(img17-simplex-lift-curve.png, “Simplex Lift Curve”)

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 now available from *XFoil*, let's try Erbach's calculation scheme to get an initial flight speed for the Wart.

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:



(lst20.png, “Level Flight Speed Calculation”)

This code gives this flight speed:

3.630740612697587 mile\_per\_hour

This is a little faster than Gary estimated 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

(eq6-reynolds-number.png)

Where **L** is a reference length, usually the mean chord of the surface, **u** is the velocity, 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

(lst21.png, “Reynolds Number Calculation”)

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.

I used equations shown above to calculate the total moment for the model at a given angle of attack, then performed an angle of attack survey. I then used a simple search routine to locate the angle of attack that gives a zero moment. I will not show that code here since it is a bit long and does not introduce anything new. Instead, here is the plot generated by that survey:

Chart, line chart

Description automatically generated

(img18-moment survey.png)

From this survey, our model should be flying at about 4 degrees angle of attack. However, we have a bit more work to do.

McLean added in induced drag (and a few other elements) in his work to get more realistic drag values. We will include those next.

## 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 Doug McLean’s 2005 article on wing-tip design [16].

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

(eq7-downwash-angle.png)

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

As a first approximation, the induced drag due to the wing is given by:

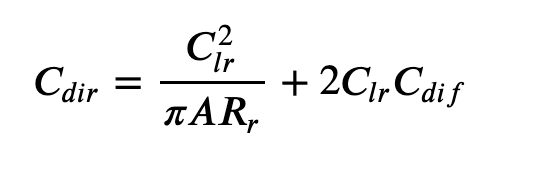
Diagram

Description automatically generated

(eq8-wing-downwash.png)

The induced drag due to the stab is given by:

(eq9-stab-downwash.png)



With these additions, we can calculate the total drag as follows:

Text

Description automatically generated

(eq10-total-drag.png)

As a first approximation in adding these effects, we will simply increase the drag force for both flying surfaces. A more complete 3D analysis will certainly give better results.

I packaged the code to perform this moment survey in a Python Function named **stability**. This routine returns the angle of attack for zero moment using the techniques we have shown previously. This code is a bit too long for this article, and the change in level flight velocity ws negligible, so I will skip showing those results.

From these results the model needs to be flying at about:

alpha = 3.836734693877551

The resulting flight speed is:

3.630740612697587 mile\_per\_hour

We are close to the speeds Gary estimated for his airplane.

The calculated total drag is:

0.004192078039260456 kilogram meter/second2

# A-6 Propeller Analysis

In the next step in our analysis, we need to find out if the propeller can deliver that forward velocity we just predicted we will need to maintain level flight.

Most studies of propellers found in textbooks assume that the propeller is designed in an optimal way. It has a proper blade airfoil, and the pitch is distributed in a logical manner. Unfortunately, all that goes out the window when we look at an A-6 propeller! The blade must be a flat plate and is attached to the prop spar at some fixed angle. No nice airfoil section, and no pitch distribution. This is one bad design, but it is easy to build! We are not even allowed to round off the edges of the blade! We are allowed to come up with some blade shape, so the chord distribution is something we can vary.

I used *WebPlotDigitizer* to digitize the *Wart* blade outline. Next, I created a flat plate section using the same approach shown for the simplex airfoil. That meant a round leading edge and a tapered trailing edge to help keep *Xfoil* happy. I was able to obtain lift and drag distributions from that work. The next challenge involves getting thrust and torque estimates for this blade.

## Actuator Disc Theory

Let’s start off by considering what a propeller actually does. Basically, it takes some mass of air arriving at the propeller surface and accelerates that air as it passed by the propeller. That acceleration creates a thrust force. Obviously, we do not get this thrust for free, we need to apply power to the propeller to accomplish the acceleration.

Simplifying this, we can think of the propeller as a disc through which the air passes. We can make this disc very thin, and simply view it as a magical thing that alters the properties of the air. Here is a simplified view of this *Actuator Disc:*

Diagram

Description automatically generated

(fig1-actuator-disc.png, “Actuator Disc”)

Specifically, there will be an increase in pressure as air passes through the disc. The lines approaching and leaving this disc represent a “stream tube” surrounding the air that passes through the disc. In the development that follows, I will skip the derivation of the equations. Detailed derivations (using *SymPy*) can be found on the project website.

If we assume that the density of the air is constant through this tube, the thrust produced by this pressure increase is given by:

Text

Description automatically generated

(eq11-pressure-thrust.png, “Pressure Thrust”)

Bernoulli says that the total pressure along a streamline is constant. This is true approaching the actuator disc, and leaving the disc, but not across the disc which injects a pressure increase. Total pressure is the sum of static pressure and dynamic pressure. From all of this, we get the following equation:

Text

Description automatically generated

(eq12-bernoulli-thrust.png, “Bernoulli Thrust”)

Equating these two equations, and simplifying leads to this result:

Text

Description automatically generated with low confidence

(eq13-disc-velocity.png, “Disc Velocity”)

Now let’s introduce the “inflow factor” a:

Text, logo

Description automatically generated with medium confidence

(eq14-inflow-factor.png, “Inflow Factor”)

The power supplied to the air flow is equal to the change in kinetic energy in the airstream. If the propeller is moving at the free stream velocity, the power needed to accomplish that is the thrust times the velocity. We can calculate the ratio of these two power values to calculate the *Froude Efficiency* of the actuator disc. This is an ideal efficiency, not something we can see in real life:

A picture containing text

Description automatically generated

(eq15-froude-efficiency.png, “Froude Efficiency”)

Now we are left with finding a way to calculate the inflow factor **a**. for that we need to look more closely at the aerodynamics of the propeller.

## Blade Element Theory

If we divide up the propeller blade into a series of small radial slices, we can examine the aerodynamics of each of these “elements” as a simple two-dimensional airflow and calculate the lift and drag forces for that element. Adding up these forces over all the elements will give us a first approximation for the thrust and drag for the blade. Multiply that value times the number of blades, and we have a value for the entire propeller. Of course, we need to consider other factors to get a better approximation, but this is a good first approximation.

In researching this theory, I found a nice reference [17] that included code that implements this technique using *MatLab*. I converted that code to *Python* for this study.

To use this code, we need the aerodynamic coefficients of the blade airfoil, which we examined earlier, and the pitch distribution which is a simple constant for the A-6 propeller. We also need the rotation speed (RPM) of the propeller, and the chord length of each element. Finally, we need to know the flight velocity of the propeller. All of these data values have been calculated earlier in our study. The code calculates the thrust and torque for the propeller and calculates a value for the inflow factor, meaning we can get estimates of the efficiency of the propeller as well.

The derivation of the equations for this code are a bit tedious, so I will refer you to the project website for that. Here are the input data values:

Text

Description automatically generated

(Listing 22: listing22.png, caption: “Blade Element Analysis Test Data”)

The code also needs lists of element radius and chord values as well.

Basically, the code uses the actuator disc equations and adds another factor that accounts for changes in angular momentum due to the blade rotation. It guesses values for these two factors, then iterates until all conservation laws are satisfied. It is possible for the scheme to fail to converge on a solution, but that was not an issue for the Wart propeller. Here is the final output produced:

Text

Description automatically generated with low confidence

(lst23.png, “A-6 Blade analysis – level flight”)

Unfortunately, this scheme produced about half of the thrust needed to balance the drag we estimated. Perhaps just picking the numerical average of the RPM is not a good idea. To get further in this study, we need to look at the source of power to our propeller!

# Rubber Motors

Probably the most difficult part of the indoor model analysis process is figuring out how to manage the rubber motor. Obviously, the motor is important, but there are many variables to consider.

Basically, we pick a strand of rubber from some batch we have acquired. That strand has a cross section we can measure with calipers. We cut the strand, then tie a knot to form a loop that has some final length, then lubricate it with some magic lotion. The final motor has a weight that we can easily measure. All of this is simple enough to document. The big issue is what kind of power will that motor supply to our model and how will that power vary over time as a flight proceeds.

Rubber is interesting material. Internally it is made up of long tangled molecules called polymers that are very happy if left alone. These molecules are all a bit different in their internal tangles causing two pieces of rubber strip to act differently even if they are identical in shape, dimensions, and weight.

When we pull on the rubber, those molecules straighten out causing them to get unhappy. They really want to go back to that tangled state. The internal temperature of the rubber goes up as we stretch, and it cools back down as those molecules return to their normal state. If we pull too hard, the molecules reach a point where they cannot return to normal. Way too hard, and they break down and, well, you know what happens!

The energy applied during stretching is stored in the rubber and released as the rubber returns to its normal state. You are in charge of the external energy application, the rubber is in charge of the return process. If the external temperature is high, rubber molecules straighten out by themselves, and it cannot be stretched out as far before reaching the failure point. The solution is to fly in cold places. Maybe that is why indoor flying is more of a winter, indoor, pastime.

When we wind up a piece of rubber, we are applying a torsion force. This causes molecules on one side of the band to stretch more than on the other side and internally causes different amounts of energy to be stored. The molecules in the band are continually trying to straighten out and the result of this battle is that evil knot! As we pack on more turns, more energy is stored, but also more knots form! The exact process we use to do our winding affects how much internal energy is stored and how and how knots form.

Phew! This rubber it complex stuff!

In Doug McLean's paper, we see this formula for the total energy we can get out of a rubber motor:

Text

Description automatically generated

(eq16-rubber-energy.png, “Rubber Energy”)

Where **𝐾𝑚** is a constant equal to 30,000 inches and 𝑤𝑚 is the motor weight. Basically, this formula says that the energy we can get out of a rubber motor is proportional to the weight of the motor. That will be a maximum amount we might get, since the other variables will determine the actual energy available for each flight. Let's see how much energy was available for Gary's record flight.

Graphical user interface, text, application

Description automatically generated

(lst24.png, “Wart Rubber Energy”)

That seems like a lot of energy. Unfortunately, we will not be able to get all of that energy into our ptopeller. Inefficiencies will enter into the picture.

# Maximum Winds

The bare weight of the motor may provide us with a measure of the possible energy we can get out of each motor, but we need to wind it up to get any useful power out of the motor. Don Slusarczyk {cite}slusarczyk2014 presented a simple formula that will tell us approximately how many turns we can pack into the motor.

Diagram

Description automatically generated  
(eq17-maximum-winds.png, “Maximum Winds”)

Where **𝐾** is a constant Don set to 8.5. **𝑚𝑙** is the motor length in inches, **𝑁𝑚𝑎𝑥** is the predicted breaking turn count, and **𝑤𝑚** is the motor weight in ounces.

Don provided a test case that will give us the correct value for **𝐾**, and the associated units we need:

Text

Description automatically generated

(lst25.png, “Calculating Maximum winds Constant”)

This calculation produces the constant and it also provides the proper units for that constant. Knowing that, we can set up a simple Python function that will take a motor weight and loop length and return the predicted maximum turn count for that motor:

Graphical user interface, text, application

Description automatically generated

(lst26.png, “Maximum Winds Function”)

Now, let's plug in Gary's record flight numbers:

Text

Description automatically generated

(lst27.png, “Wart Maximum Winds”)

Gary launched his record flight with 3660 turns, which looks reasonable using this function to calculate a maximum turn limit! He was probably safe, since measuring the loop length is often done after the knot is tied, and the exact length is a bit hard to get with the motor in a loop.

# Torque Curve

As mentioned earlier, as you wind up a motor, you are adding energy to the rubber. We can measure the force the rubber is applying back s it tries to return to its normal state on a torque meter. Once you stop winding and release the motor to do its own internal work, we can again measure this torque and we will find that the curve is different. This difference is called hysteresis, meaning that the torque delivered to the propeller at each turn count is lower from the winding torque at the same turn count value. Measuring this is difficult and data reflecting the delivered torque is difficult to find. Here is a diagram from Carl Bakay's article on rubber I found in volume 102 of INAV {cite}bakay2001rticle showing this effect:

Diagram

Description automatically generated

(img19-torque-curve.png, “Typical Torque Curve”)

This figure gives us a variety of parameters we might use to model a rubber motor. For flight analysis, we are not really interested in the load data, except that it certainly has an effect on the structure of the motor stick! The energy delivered to the propeller is what we want to examine. A simple approach is to assume a fixed percentage loss that accounts for the hysteresis effect. I have seen estimates ranging from 25-30 percent, so we can add this as another parameter for study.

For the purposes of this study, we will set up a simple analytical curve that models a rubber motor. From this figure, I will use simple polynomial curves for segments A and B and D above, and a straight line for segment C. Here is what my “analytical” motor looks like for now:

Chart, line chart

Description automatically generated

(img20-analytical-torque-curve.png, “Analytical Torque Curve”)

With this model, we can scale the curve using the maximum turn count and a maximum torque value the model builder can select. The parameters used to define the curve are something we can alter as part of a more detailed study, hopefully, using real data from tests of actual motors.

We have already presented a way to find that magic “average torque value” in our analysis of Bud Tenney’s flight data. The only remaining puzzle is finding the average propeller speed.

# Propeller Speed

This part is going to be a puzzle. We have a scheme that will tell us something about how the propeller might work if we spin is at some speed, but not one that will tell us how fast it will spin if we apply some torque. Or do we?

Increasing the RPM value by 10% in our original propeller analysis produced this result:

Thrust 0.004140032227728423 kilogram \* meter / second \*\* 2

This is very close to our original model drag calculation for level flight of:

Drag = 0.004192078039260456 kilogram meter/second2

So, the critical part here is coming up with that RPM value!

Gary Hodson provided me with a spread sheet with details of many of his flights with the Wart. I generated a plot of average propeller speed as a function of initial torque and cme p this this curve:

# Putting It All Together

We have worked through a lot of material. So how do we put everything together and come up with an estimated time?

1. Design your model, I would use OpenSCAD!
2. Pick your airfoil sections.
3. Get an estimate of the weight of the model.
4. Find the CG location and lift and drag force locations.
5. Generate airfoils data for XFoil
6. Estimate the Reynolds Number for your model
7. Generate Aerodynamic Coefficients using XFoil
8. Perform an Angle of Attack survey to find the level flight speed
9. Determine the required thrust value
10. Pick a planform for the propeller and pitch distribution
11. Using XFoil, generate aerodynamic coefficients for the prop
12. Use the Blade Element Model routine to find the RPM required for level flight
13. Pick a motor length and weight
14. Calculate maximum winds using Slusarczyk’s method
15. Pick a winding torque value.
16. Use the analytical torque function to get a torque as a function of turns curve.
17. Integrate this curve to find the average torque value
18. Use Hacklinger’s Equation to produce your time estimate!

Here is my final calculation for the Wart:

(lst28.png, “Final Time Calculation”)

Is this really going to work. I will be reviewing all of this work and using more sophisticated methods to improve this process. I am also collecting the code shown in these *Jupyter* pages so they can be more easily used to do these assessments.

There are some things here you need to figure out on your own. We will not get a magic formula for everything. However, once you pick these values the procedure will give you a time estimate. you can do a parametric assessment of those parameters and see how the time changes. Perhaps this will help optimize your choices. In the end, you will need to test your own motors and model settings and see how things compare with your analysis.

Hey! I am College Professor. This is your homework assignment!

# Next Steps

I have no doubt that there are holes in my analysis. If any of you readers have data from your own flights, especially motor data, I would love for you to send me that information. I need to validate many aspects of this work against experimental data, and I plan on doing some experimental work to do tis myself. For example, I am hooking my digital torque meter to a Raspberry Pi so I can display and record winding data in real time.

I 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. AI do all of this work will on my development laptops. I was a teacher for many years, so I have machines for every major operating system). I test things on Mac, 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

Remember, my intent in this study 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.

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.

It is amazing how easy it is to put together ideas and test things out in the *Jupyter Notebook* environment. You get immediate feedback on what works and what does not. Generating plots in a simple way makes it easy to see what you are generating.

To get started in your own research, you can use this study as a guide. 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 even though I am retired now, 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 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.github.io/nffs-2022-symposium-live>.

# Bibliography

|  |  |
| --- | --- |
| [1] | R. R. Black, "Math Magik: 3D Model Airplanes with OpenSCAD," 2021. [Online]. Available: https://rblack42.github.io/math-magik/. |
| [2] | R. Black, "Designing an Indoor model Using OpenSCAD," *NFFS Symposium,* pp. 54-60, 2021. |
| [3] | G. Hodson, "Gary Hodson's Wart-A6," *NFFS Symposium,* pp. 172-173, 2010. |
| [4] | Heart of America Free Flight Association, "Casey Free Flight," 2022. [Online]. Available: https://kcfreeflight.org/. [Accessed 27 May 2022]. |
| [5] | D. Mclean, "A Methof for Predicting Indoor Model Duration," *NFFS Symposium,* pp. 54-60, 1976. |
| [6] | W. Erbach, "Calculating Indoor Flight," *NFFS Symposium,* pp. 79-85, 1990. |
| [7] | M. Hacklinger, "Theoretical and Experimental Investigation of Indoor Flying Models," *Journal of the Royal Aeronautical Society,* 1964. |
| [8] | jupyter.org, "jupyter," 2022. [Online]. Available: https://jupyter.org/. [Accessed 27 May 2022]. |
| [9] | binder.org, "binder," 2022. [Online]. Available: https://mybinder.org/. [Accessed 27 May 2022]. |
| [10] | B. Tenny, "CHoice of Rubber Motor for Low Ceiling Indoor," *NFFS Symposium,* pp. 77-82, 1968. |
| [11] | A. Rohatgi, 2022. [Online]. Available: https://automeris.io/WebPlotDigitizer/. |
| [12] | M. Okamoto and K. Ebina, "Effectiveness of Large-Camber Circular Arc Airfoil at Very Low Reynolds Numbers," *Transactions of the Japan Society of Aeronautical and Space Sciences,* pp. 295--304, 2016. |
| [13] | M. Drela, "XFoil - SUbsonic Airfoil Development System," 10 02 2022. [Online]. Available: https://web.mit.edu/drela/Public/web/xfoil/. |
| [14] | DARcorporation, "xfoil-python," 2022. [Online]. Available: https://github.com/DARcorporation/xfoil-python. |
| [15] | M. Reid, "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, 2006. |
| [16] | N. Drela, "QProp - Propeller/Windmil Analysis and Design," 2022. [Online]. Available: http://web.mit.edu/drela/Public/web/qprop/. |

# 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.