Hello, my name is JT Klenke, I’m an undergraduate at Cornell University majoring in computer science. I'm very interested in a summer internship in an entrepreneurial environment. I have experience coding in Java, Python, OCaml, and C/C++. I’m a fast learner and a hard worker eager to take on new challenges. By summer my completed CS coursework at Cornell will include Object Oriented Programming and Data Structures, Functional Programming, and Algorithms. Here is a link to my personal portfolio website: [jtklenke.github.io](http://jtklenke.github.io/)

About Me

I’m an undergraduate studying computer science in the College of Engineering at Cornell University. I enjoy math and physics and spend my free time flying drones, taking pictures, and throwing pottery. I’m part of the computer science team of Cornell’s Aerial Robotics club which competes in the annual International Aerial Robotics Competition. Additionally, I work with Misha Padidar as a research assistant in Cornell’s Applied Math department optimizing the coil shape for stellarator fusion reactors.

My Projects

Evolutionary algorithm assisted antenna design

I wrote an evolutionary algorithm in Python to optimize the antenna shape for communications antennas, the final design is to be used on F-5 Navy training jets. The plate antenna operates from 108-400MHz, the evolutionary algorithm designed antenna is about twice as good as the old design and exceeds the specification by 25%. Here is the paper I wrote on the process: Evolutionary Algorithm Designed Broadband Plate Antenna for F-5 Vertical Stabilizer.

The first few challenges of this project are about representation and fitness. First is to decide what makes a good antenna, since the specification is clear that the VSWR needs to be below a certain value over a certain range I defined fitness as the average VSWR over the range 108-400MHz. Second, I must decide on a representation for the antenna, this representation matters a lot because it will determine what “similar” antennas are from an evolutionary perspective, ie. two antennas where one mutation can make either one. Ideally, similar antennas from an evolutionary perspective have similar characteristics, namely fitness. I decided the representation would likely be a matrix but left the specifics until later.

After researching existing antennas, I figured that I would try to recreate some of them and see if I could get similar results using the windows version of 4nec2, the Python version necpp and a real world sheet of metal. The chosen antenna design was a [scimitar](https://en.wikipedia.org/wiki/Scimitar_antenna) antenna like was used in the Apollo program. I wanted to get a list of points on the path of the scimitar that I could later feed into necpp. Using the [patent](https://patents.google.com/patent/US3015101A/en) I parameterized the two curves (k e^(at) cos t, k e^(at) sin t) and wrote some code to generate and plot the points. Which for a specified number of segments returned this:

(image points)

Next, I wanted to test that antenna in necpp, to do that I first needed to familiarize myself with the necpp library. I found some example code from the [PyPi Website](https://pypi.org/project/necpp/). It simulates a vertical pole and calculates the impedance for the frequency 34.5 MHz.

(code ex from pypi)

The library works with cards, a holdover from actual punch cards, that are used to define aspects about the simulation. For example, the nec\_fr\_card allows me to change some frequency information about the simulation. After getting that to work, I tried simulating multiple connected wires, with the goal of eventually putting together a scimitar geometry. When I ran that though, I got an error, "index 2 is out of bounds for axis 0 with size 2" or sometimes just a crash. The lack of good documentation and examples made debugging difficult, but eventually I figured out the problem was with the segment count. I brute forced my way to a segment count that worked for the shape I was trying, which would be good enough for now. Later, I found out that the segment count must be high enough so that the largest segment of a wire is smaller than 20% of the wavelength. Basically, divide the speed of light by the highest frequency, divide that by 5 and make sure the wire has enough segments so that they are all smaller than that value. From there I could calculate the impedance for a specific frequency for any geometry I wanted, the final code looked like this:

(code impedance func)

Which returned the real and imaginary components of impedance for a certain frequency (freq) for a given list of points (pts). Impedance could then be turned into VSWR using [this](https://chemandy.com/calculators/return-loss-and-mismatch-calculator.htm) equation. To properly evaluate an antenna, I need a way to find the VSWR for multiple frequencies. Originally, I called the impedance functions for every frequency in the range, but this was needlessly slow because the program had to rebuild the geometry and environment each time it tested a frequency. By changing the frequency card (fr\_card) I could calculate the impedance of multiple frequencies and then return them as a list.

Finally, I could test out the scimitar. To get better results, I moved the tip of the scimitar to (0,0) and the wide part of the scimitar off the ground plane. To fit in the available space, I scaled the scimitar of the patent slightly. Here is the antenna I tested and the resulting VSWR graph:

(images scimitar and vswr graph)

The scale is in meters and orange line marks a VSWR of 2.5, which is the goal for the whole frequency range.

Clearly, the antenna as-is was not good enough for the job, but it was possible with the right set of parameters it could work. Before subjecting it to an evolutionary algorithm, I had to choose some parameters that would determine the specifics of an individual scimitar antenna. The 4 that I chose were a horizontal scaling coefficient, inner and outer curve values, and scale. I set boundary conditions for the antenna so that they would be a reasonable size (not good ones though because the output antenna ended up being about a meter long). Next, I modified the existing code to test the scimitar geometry and output its score as a sum of all the VSWRs in the chosen range to optimize, this was the cost of an organism.

Then I just had to implement the evolutionary algorithm. The core steps in an evolutionary algorithm are evaluation, selection, and variation. A random sample is generated and then these three steps are iterated as many times as necessary. Evaluation is the process of determining the most fit individuals (or least costly). Selection is the process of destroying the least fit individuals. And finally, while there are many different possible implementations of variation, the basic goal is to replace the lost population with combinations of the successful organisms with some random variation. As an example of variation, if an organism is represented by a binary number, the first half of one and the second half of another successful organism could be combined. Mutation could be implemented by flipping some number of bits in the new organism. I took the best half of organisms according to the cost and averaged random individuals to create the new population, additionally I added or subtracted small numbers to act as mutation. After running it for 500 generations with a population of 500 the result looked like this:

(image)

which gave a VSWR graph that looked like this:

(image)

where the y-axis is 0-50 and the x-axis is 118 to 400MHz. After that I tried a few other optimization strategies. First, putting a 3 times weight on all frequencies higher than 250MHz which yielded this antenna:

(image)

Which gave this VSWR graph (same axis):

(image)

I wanted to see if real versions of the antenna would give similar VSWR plots as the simulated ones. After drawing out the scimitar shapes on some paper, gluing tinfoil to that paper and cutting out the shapes, I tested the geometries. It turns out tinfoil gives the same VSWR values as a metal plate of the same shape, antennas are just made out of .032" aluminum so they don't melt. Outlines of wire also (supposedly). give the same VSWR as a filled in sheet which should mean that nec correctly simulates plate antennas. The actual sweep of the second antenna looked like this:

(image)

With x-axis from 0-1400MHz. The shape is similar in the range 118-400 but the values are considerably lower in the real world. This is probably fine because if the shape of the graph is similar then an optimized version will still be optimized in the real world and just have a better VSWR.

The scimitar seemed unlikely to work well enough to meet the requirements for a few reasons. First, all the antennas were too big but still gave VSWRs not close to the goal. Also, scimitars are usually used for narrow high frequency ranges and are seemingly can’t work well in lower ranges as I'd hoped. While there are a few options for existing designs to optimize (a U like design, multiple vertical wires, scimitar variants and trapezoidal shapes) I really wanted to see if a generalized antenna evolver would work.

I started by defining a rectangle that bounded where points could be, about 25cm tall and 60cm wide and defined a shape that would be on every antenna a 2cm by 1cm triangle where the antenna would be attached (with an alligator clip for testing but eventually soldered). Then, I defined an individual as a collection of points (a 2 by N matrix to represent the x and y position of N points) in the order that they would be connected by wires which would outline a shape. For example, an individual with a matrix of points like this:

(matrix)

Which gave this outline:

(image)

Notice the small triangle at the bottom of the antenna, that's the preset shape that allows the antenna to be attached, every matrix has the points (0,0) and (0.01 , 0.01) at the beginning and a similar set of points at the end to define that shape. All I had to do was make some small modifications so that the testing code would take in the new representation. There's one big problem, my code can’t really handle geometry errors. If two wires cross, it breaks, if two wires are too close together, it breaks, and if the wires are in certain configurations, it returns negative VSWRs (it breaks). Luckily, all the necpp functions return an int, 0 if it ran without errors, and some other number if there was an error (different numbers for different errors). I can find negative VSWRs because the real component of the impedance is negative. I use that result to check if there are any errors when calling a function and if there are, I give the broken antenna a high VSWR which should make it worse than any working antenna. I didn't have to do this for the scimitars because I set the limits on the individuals such that they would all give working shapes.

After running this new implementation in Google Collab, the program would eventually crash, without error. I figured that the problem was that I was running out of memory but after running it a few more times that didn’t seem to be the problem. Since memory didn’t seem to be the problem, I guessed that some antenna configuration might cause necpp to get caught in an infinite loop or crash. To see if that was the case, I added some print statements to print out all the “individuals” in a generation, unsurprisingly, this made the program too slow, and it never encountered the problem. While Google Colab had been great for a few reasons, it supported necpp and was faster than my laptop, it seemed that I would have to leave it behind. After downloading Oracle VM and installing Ubuntu, necpp is Linux only, I copied over my code, replaced print statements with writing the to a text file, and ran the program. After a few hours, the program stopped, just like it had in Colab.

To see if it was one of the antenna shapes that made it break, I got rid of the code that made random individuals at the start and replaced it with code that read in the saved generation from the text file. It ran fine. So, it wasn’t the specific antenna shapes that made it break. I had no idea what else to try, after a while, the code just stops. I noticed that the more times I load in the generations after it stops, the shorter it runs for, so obviously there is some dependence on what the generation looks like, but I still don’t know why the code is broken.

Luckily, in testing that theory, I had also built a workaround. I could just run the code until it stopped, exit the program, and restart the program loading in the saved generation. If I did that a few times then it would be as if the program had run longer. This also isn’t as much of a problem as I’d feared because individuals stop improving after a while. Here is the graph of how the best individual scores as a function of generation, I looked a few of these graphs and they all follow the same pattern.

(image: best individual)

Also, some runs seem to get stuck early on with a bad design, in which case I just restart the program, the graph for those runs looks like this.

(image other best)

Interestingly, while the graph of the best individual’s core follows a roughly exponential decay, the average score of a generation is very differently shaped. Again, this is an example of just one run but all of them take on a very similar shape.

(image: averages)

After running it about 6 times (with about 3 “restarts” per) I collected the best individual from each run. I then selected the two best scoring designs to test in the real world. They looked remarkably similar:

(images: irl antennas)

They both had a generally trapezoidal shape, took up the maximum height and had a extended part on the right top size. This seemed good, if the algorithm had converged on similar shapes over multiple independent executions, that would suggest it was optimal in some way.

I cut them out using the same technique I had used to test the scimitars, thick paper glued to tinfoil cut out with a CNC-like cutting machine. Then I taped them to the test stand, attached them with an alligator clip and put them on the tail, the testing setup looked like this:

(image jet)

The first antenna we tested yielded this as its VSWR scan, the red and green lines are two trials, with essentially identical results.

(image vswr sweep)

For reference this is the VSWR scan on the old design, the highlighted section is the same range as the previous image.

(image vswr sweep old)

Way better, and even more important, better than spec. The other design was alright, but not as good. In the end, the final design was cut out of aluminum and tested again, just to make sure, with identical results. That antenna was then shipped off to a third party who verified the results and additionally did some analysis on the directionality of the signal, which showed that the new design was no more directional than the old design. This project shows the significant potential of evolutionary design, with relatively little domain knowledge I was able to design a fixed antenna that outperformed the old design and exceeded specification. Afterwards, I wrote a paper, Evolutionary Algorithm Designed Broadband Plate Antenna for F-5 Vertical Stabilizer, that goes through the requirements, methods, and results in additional detail.

Timelapse photo creation

Inspired by this (https://www.ted.com/talks/stephen\_wilkes\_the\_passing\_of\_time\_caught\_in\_a\_single\_photo?language=en) Ted Talk by Stephen Wilkes, I wanted to create a version of these photos procedurally. Stephen Wilkes creates them manually, painstakingly picking out the most interesting subjects and times of day, manually blending it together to create his images. I wanted to simplify the process, using frames from a timelapse video, and procedurally blending over time to create a similar effect.

The first method that I wanted to try was to take columns of pixels from different frames of the timelapse. This would be easy to implement and give me a rough idea of how a more advanced method would look. I would specify a width that the final image would be as well as the number of columns of pixels I would take from each frame. From that, the final image could be constructed. The result is bands of pixels from separate frames arranged next to each other.

It's surprisingly tricky to find 24h time-lapses of interesting things with still cameras on YouTube (most are of the night skies, which stays relatively similar throughout the video, have moving cameras, have watermarks, or are of boring subjects.) I found two that work well enough for experimentation but suffer from one of the above issues: [30 Days Timelapse at Sea | 4K | Through Thunderstorms, Torrential Rain & Busy Traffic](https://www.youtube.com/watch?v=AHrCI9eSJGQ&) and [24 Hour Time Lapse](https://www.youtube.com/watch?v=x1Os8NcSrbI). The first has a watermark and text plus I found out everything happens too fast for this method to work effectively, the second is just a boring subject and the total number of frames is a bit too short to do everything I wanted (925 frames) but was great for testing because there's a good span in time but not much changes frame to frame. The last video I found was [Timelapse Los Angeles / Santa Monica Beach California](https://www.youtube.com/watch?v=Ew7N65F6oYc). This was a good video with a still camera and interesting subject over a long enough time that there was good variation but unfortunately, the video is super short. There are only 616 frames and only 470 of those are useful (the rest are black or darkened from a fade in and fade out).

To get the frames of the I downloaded the YouTube videos using one of many online tools, they're kind of sketchy if your careful to only download the video they seem to be fine, then I used VLC Media Player to get frames. Later I would streamline this process using the Python library youtube-dl to download YouTube videos and FFmpeg to get all the frames as .jpg images.

I then used Image from PIL to get pixel data from each frame. My final code simply creates a black image of the appropriate width and height and then filles it in with pixels in some nested loops, the final image is then showed. The result of the first attempt is a little underwhelming but shows some promise and some flaws to be remedied. Here it is on the 30 Days at Sea and the 24 Hour videos.

(image)

This has one pixel (column) per frame and a width of 1200 pixels.

(image)

This one is 925 pixels wide and 2 pixels per frame.

There are a few obvious issues to this method, first, the width has to be manually capped by the number of pixels per frame and the length of the video second, it just doesn't look that great, if more than 1 or 2 pixels are used in each band, they become extremely obvious. Below is an example with 4 pixels per frame and you can clearly see the bands start to appear.

(image)

While I think this is a good proof of concept that can make interesting images, it's not a great method.

For my second iteration I would generalize the process by taking pixels from different frames based on a more arbitrary function. For example, the distance of a pixel from the center would decide what frame that pixel would be taken from. This solved the issue of manually setting the width and number of pixels per column, the final image would have the same dimensions as a frame from the video and the pixel distribution would be decided by the function. I ended up having a scaling coefficient to the function determined by the maximum value of the function and the size of the “useful frames” of the video. There’s still some manual parameter tuning. but I suppose this whole project is more art than science anyway. Doing this for every pixel should create a picture that would smoothly blend over the length of the video.

After implementing this more general form, the time to build an image skyrocketed from just a few seconds to an hour at which point I ended the program. What was taking so long? It turns out that while my first implementation only opened each image once (400-1,000 images), the new one opened a new image for every single pixel, about 2 million. This was obviously a complete waste of time. Fixing it required splitting the process into two steps, first figure out what pixels would use what frames, second opening each image once and copying the pixels from that frame over. This sped it up immensely and the code was back to running under a minute. Eventually I decided it would be easiest to get all the pixel data from every frame, store that in a numpy array and then simply grab the data when I needed it. I folded this array building process into the separate code that included the video download and frame extraction. After running this code with the center distance function on the Santa Monica Beach video I got this image.

(image)

Much more interesting. However, after finishing that implementation, I noticed that the resulting pictures had noticeably rough transitions between pixels.

(image)

To fix this, I decided to use a weighted average of a pixel’s value over some time, this should soften the transition essentially by “blurring” the photo over the time axis. I was hoping this would mimic the effect of long exposure photography. This new method took significantly more computer power, while the original version sampled just 1 value per pixel the new version would sample 64 pixels and then perform a weighted average, which took additional time.

To speed up the new version I looked to parallelize the computation using the multiprocessing library. I further optimized the code by precomputing the Gaussian distribution (which is what I used as the weighted average values), pre-retrieving the image pixel values and storing them in a np array, and doing all three color channels at once using a matrix multiplication rather than doing the dot-product on a per channel basis, after that I was able to get the time to build an image down to around 10 minutes on my laptop, still slower than I’d like but not excruciatingly slow.

(image)

There are still some noticeable artifacts that I suspect come from YouTube's video compression being compounded by averaging over many frames, especially around the ferris wheel. Additionally, with this new technique some of the color detail and contrast is lost in the ferris wheel which is something I might revisit. I'd like to take some of my own timelapse video and use this tool and see what results I can get.

A\* pathfinding

As part of my work on the motion planning team of Cornell’s UAV club I implemented and benchmarked a few different implementations of A\* and related algorithms in Java and C.

The objective was to find the shortest path between two points on an NxM grid where some grid points are marked as obstacles. I first implemented Dijkstra’s shortest path algorithm which would be extremely slow for the given use case but was simple to implement and helped me understand the core algorithm behind A\*. Unsurprisingly, Dijkstra’s algorithm was slow, especially on large grids.

With a better understanding of the algorithm, I moved on to implementing A\*. Deciding that now was as good a time as any, I also created some test cases using JUnit after all if it’s not tested, its broken, and wouldn’t you know, it was broken. After some significant debugging I realized my mistake, as an optimization I had only created an instance of the point class when I “discovered” a point, what I had done wrong was to instantiate a new point even if it had already been discovered. Honestly, with such a major oversight I was shocked how few test cases it failed. To fix this I used Java’s HashMap to store all the discovered points and only created one if it didn’t exist.

After testing with JUnit and benchmarking A\* I was surprised at how slow it was. Looking for ways to speed it up I found an algorithm that takes advantage of the regularity of the grid called jump point search (JPS) which is said to be significantly faster. The original idea for JPS comes from this (<http://users.cecs.anu.edu.au/~dharabor/data/papers/harabor-grastien-aaai11.pdf>) paper by Daniel Harabor and Alban Grastien but I found this (<https://zerowidth.com/2013/a-visual-explanation-of-jump-point-search.html>) website by Nathan Witmer to give a great visual intuition behind the algorithm. Unfortunately, for almost every obstacle placement I found JPS to be much slower than traditional A\* and for very large grids it would run into the recursion limit.

Still looking for a faster implementation I tried to optimize by original A\* implementation. I did so first by writing my own priority queue implementation that supported an ‘increase priority’ method and second by condensing the representation of a grid point. Originally, I had a grid point class that stored the g-cost, f-cost, parent point and x and y coordinates of the point. While this was simple and readable, it came with all the overhead of a Java class, which for many points added up. I was able to reduce all the information down so that it was able to fit into a single long (a 64-bit number) then I built a class to abstract that structure away from the core A\* algorithm. This helped quite a bit but was still slower than I would have liked.

I decided to code the algorithm in C. It would be a good opportunity to refresh some basic C and should yield a faster implementation. I decided that I would start by implementing a linked list which I used to store the final path. I also started by implementing the priority queue as a linked list, just iterating over the whole list, and then returning and splicing out the “minimum” element. I knew this would be very slow (O(n) as opposed to O(lg n) from a binary heap) but it would let me debug the A\* algorithm, give me some extra practice in C and give me an idea as to the speed of C compared to Java. Additionally, instead of using a hash map to store the discovered points I just used an array with all of the potential points, while this would be very memory inefficient it would be very fast and easily implemented.

After getting that implementation working, I noticed that it was actually significantly slower than Java, originally, I just attributed that to the slow priority queue but even after I modified the Java priority queue to loop over every element it was still slower. I looked for ways to speed up the C code and realized that there were a few obvious things that were slowing it down. First, I cut down some unnecessary malloc calls I had, second, I used additional space to store some values that I had previously been recalculating, most importantly, I used O flags, O2 sped up the code by almost 4x.

Finally, with some guarantee that it would be faster than Java, I implemented the priority queue as a binary heap. In the end, while it took more significantly more time to code than Java it ended up being substantially faster. I was able to get the code to run anywhere from 4-100x faster than the best Java implementation.

Tribology internship