Genetic Algorithm Amplifier Biasing System (GAABS):

Genetic Algorithm for Biasing on Differential Analog Amplifiers

By

Sean Whalen

June 2018

Senior Project

Computer Engineering Department

California Polytechnic State University

San Luis Obispo

2018

Table of Contents

Abstract

This project looks into using a genetic algorithm to bias an amplifier to yield the largest gain. It looks to tackle the issue that analog amplifying circuits often are specifically setup for a particular input signal with a range of values, having fixed bias voltages, but this lacks an aspect flexibility in its applications. Using python, a script is run to utilize LTSpice to bias a bipolar junction transistor based differential amplifier. The script implements a genetic algorithm to continually search different potential biasing voltages, which completes when the gain is largest and unchanged for 15 consecutive generations. This specific circuit, a differential amplifier, takes 18-25 minutes to fully converge, getting gain values averaging around 650V/V, being able to converge within 3% of this point in every running of the algorithm.

Introduction

This project looks into using a genetic algorithm to bias an amplifier to yield the greatest value of gain, with the purpose of allowing for more widespread applicability these amplifiers. The system will apply a genetic algorithm to bias an amplifying circuit, using 2 bias voltages as the children of each generation; the algorithm is applied to the circuit in LTSpice, running through Python code, iterating until the best bias voltage pair is determined after 15 consecutive generations of being the strongest result. The genetic algorithm generates 20 pairs of children per generation, starting with randomly generated pairs; it performs bit-mutation on 10 pairs of children, crossover mutations on 4, throws out the worst 4, and keeps the best 2 to make up the next generation of 20 children. The algorithm grades on the greatest difference in voltage between 2 points on the output, as that is synonymous with the differentiation of the output, which is directly related to the amplifying factor. Gain can also be calculated by dividing the difference in output voltage from the difference in the input voltage over the same period of time. Results from each generation are printed to a file named 'Results.txt.'

Developing a self-biasing circuit is an attempt to better understand genetic algorithms and prove concepts of analog circuits. This project is at the intersection of computer science and electrical engineering, combining knowledge of complex circuitry and the logic for implementing the genetic algorithm in code; as a computer engineer. This project allows me to show proof of concepts I've learned during undergraduate studies at Cal Poly, while engineering a solution to a complex problem in the field of electronic circuits and computing algorithms.

Background

The purpose of this system is to bias an analog differential amplifier with the greatest increase in amplification from input to output through LTSpice, being ran via a set of python scripts. An analog amplifier takes an analog input waveform asnd modifies the maximum and minimum voltages of the signal, as well as the slope with which the voltage is changing at (analog amplifiers do not affect a signal's frequency). The change in amplification is typically measured in gain, the output voltage over a short range of time divided by the input voltage over a short range of time. This project measures amplification via the largest difference in 2 discrete output voltage points; the input voltage doesn't need to be taken into account, since running of the script will bias the circuit based off of a fixed input voltage.

Analog amplifiers utilize transistors to amplify a signal based on their voltage and current equations, and the particular arrangement of the transistors can increase or decrease the overall amplification factor. The amplifier in this project uses bipolar junction transistors, or BJTs. BJTs are transistors that allow current to flow through them based on the voltages on each node (base, collector, emitter). When a voltage is applied to the base node, the other nodes voltages will fluctuate with the base node. This project uses a differential amplifier, meaning it has 2 inputs; one input is a fixed waveform, and the other is a fixed DC voltage, with both being connected to the base of an NPN BJT. The collectors of these BJTs are connected to the collectors of PNP BJTs, giving the circuit an active load; their base voltages are equal to each other, which is one of the biasing voltages, VB, while the emitter is fixed at a VDD = 5V. Changing VB will change the current through those transistors, affecting the collector and emitter voltages, and thus affecting gain. A passive load, where the collectors of the NPNs would be connected to resistors, was not chosen because active loads, though more difficult to bias, can yield greater gain.

The circuit also utilizes a current tail, where 2 NPN BJTs are connected via their base nodes and their emitters tied to ground (0V). The collector of one BJT is connected to the emitters of the NPN BJTs with the base voltages set to the inputs to the circuit. The other tail BJT has its collector and base tied together, connected to a resistor; the other end of that resistor is the second biasing voltage, VDB. This voltage will affect the total current flowing through the tail, which will affect the current through the other transistors, which changes the gain of the circuit.

A genetic algorithm is an algorithm based off of human biological evolution, utilizing properties of natural selection and random mutation of inputs to come to an optimized solution [1]. Genetic algorithms work by creating sets of children (inputs to a system), testing them, scoring the result, and making changes to the children to create a new set of children, or a new generation. A key aspect of genetic algorithms is the blending of randomness and selection off of the scoring system; the algorithm in this project seeks to strike a balance between the two.

The majority of generating new children in the algorithm of this system is bit-based mutation, where a bit of the number can be flipped. Each of the sets of children are made up of floating-point values, 32-bit numbers, which are composed of a sign bit (positive = 0, negative = 1), an 8-bit exponent values (the sum of the real exponent and the 2's complement of the largest positive number able to be represented with that many bits), and a 23-bit mantissa (the bits to the right of the first 1 in binary representation of the number). Floating points are calculated by shifting the decimal of the binary mantissa by the real exponent bits left or right (if positive or negative), with the sign added in front at the end. The mantissa's bits are the values that get flipped.

The genetic algorithm attempts to find the best biasing voltages to bias the circuit to achieve the greatest gain, or a very close approximate, on every running of the script. Since different input signals will operate in a region of greatest gain using different biasing voltages, the algorithm must be flexible, and have a high success rate for finding the best set of biasing voltages for the specific amplifier (VB and VDB). The data flow of the genetic algorithm can be found in the appendix.

The project runs via LTSpice, a simulation tool used for predicting how a circuit would ideally behave in real life application. LTSpice provides a user interface that allows users to place circuit components in a schematic and simulate them graphically. LTSpice then generates a spice deck, or a set of connections between components described in text, and that is what's ran to get a simulation; the simulation creates set of discrete points, which are currents and voltages on each node and through each component, and that is what's graphically displayed. This project calls LTSpice to run via python script, which removes the graphical user interface from it, and returns a binary file with all the voltages and currents for each point in time.

Requirements and Specifications

The system is designed to work on any differential amplifier, operating with any input signal, as long as there are other biasing voltages that can be changed to affect the gain. The system should be able to run with LTSpice on a particularly named schematic file, which would need to be changed to run with a different schematic file. The two changing biasing voltages must be named “VB” and “VDB” so that the script can set these voltages in the spice deck; otherwise, the script would need to be rewritten to handle differently named voltages. The system should converge around roughly the same gain every run with the same input signal. Changing the input signal’s amplitude could change the greatest possible gain and changing the DC offset voltage could also have an effect. The script assumes the maximum voltage in the circuit is 5V and the minimum voltage is 0V; it generates children using a random number generator in a range between 0 and 5V. The script is not expected to run quickly; it takes time to collect all the data points for each simulation, run 20 simulations, and continue until the most highly graded child has been constant for 15 generations. The script shall run in Windows command line using python 2.7; other versions of a terminal will not work, and the script must be run in the same directory that python is saved in.

Design and Development

The idea for the project started as designing and building the digital circuit with the analog amplifier, where the digital component would bias the analog circuit to get the greatest possible gain. It was decided the amplifier would be a BJT based differential amplifier with an active load, and the digital circuit would attempt to work using a genetic algorithm to bias the voltage on the base of the active load PNPs. After planning and trying to work out how to do this entirely as a circuit, the decision was made that there wouldn't be enough time to either fabricate an integrated circuit from a schematic and manual layout using Cadence tools. Although the project would be reformatted, the project goal didn't change, using a genetic algorithm to bias an amplifier.

Since the manual layout and schematic design would take too long, energy was focused on looking into a way to write code to automate the layout process. System C, a subset of C++ that translates code describing a digital circuit into VHDL, appeared to be a viable path to take on creating the circuit. It was a good option at the time, as it still offered the chance to use Cadence tools get the circuit fabricated. Looking into System C for a while, more questions kept popping up about how the translation from C++ to VHDL would take place, and how everything would come together, code and simulating via Cadence. It was determined that another approach would have to be taken to do this, as the fabrication process would be too time intensive, given the remaining time left on the project was less that 10 weeks.

At this point, the idea of fabricating a circuit was scrapped for writing code to integrate with an analog circuit. LTSpice, a circuit simulation tool that electrical and computer engineers at Cal Poly are very familiar with, showed itself as being the most reasonable option. The project was then reformatted to writing a script that would run the analog amplifier in LTSpice, gather

information about the simulations, and use a genetic algorithm to bias the circuit to achieve the highest gain. Since biasing the circuit with one voltage would be too quick, a second voltage to bias was added as the voltage on the current sink.

The most grueling part of this project was going to be integrating LTSpice to run via command line and output valuable data that could be read out; fortunately, this process was already completed by someone's master's thesis, allowing for integration of this code; the original code was written by GitHub user 'joskvi' and was available via their GitHub account [2]. This code was a lot of python files that ran through command line to operate on a RC low-pass filter, being able to change the resistor and capacitor values. Using a MacBook Pro, it took a few days of trying to get the code to run until realizing that it only runs on Windows. A small obstacle, but one nevertheless, led to installing software to dual boot the MacBook with the Windows operating system. After setting up the correct execution paths for the script on Windows, everything worked as anticipated.

The next step was taking the script that worked on the filter and integrating it with the differential amplifier from the original plans. Drawing the schematic of it wasn't difficult, but what proved challenging was where to make every change in the code. The names of the parameters had to be changed, the correct values needed to be taken from the raw file (containing every value at every point in time), and the path of execution needed to be changed. After walking through the code and working to understand everything going on with it, making the adjustments wasn't too time intensive. Getting the differential amplifier to run was a milestone, but there was still a lot left to integrate.

The next step was getting the system to run for a specified number of children and storing all the values (VB, VDB, maximum difference in voltages between 2 points in time) from each

simulation. The first function call in the script would check if there was a parameter file existing, and it would change the values in LTSpice to the values in the file. This original function was discarded for the time being, in favor of a new function, that would automatically adjust the VB and VDB values for each simulation and run in a loop for 20 children, storing the max difference in consecutive voltage points. Getting all 20 differences meant the system could determine which pair of voltages yielded the greatest gain. After this, the genetic algorithm needed to be integrated.

Implementing the genetic algorithm took a long time, as there were many steps involved. First, the system need to be iterative, running the 20 children over and over until converging for a specified number of generations; this was just running a couple nested while loops. Adding the portion in which the code generated the new generation was very time consuming. After deciding on the types of manipulation to create the new generation, the functions had to be written. Each function takes in a certain number of VB and VDB values (depending on the function), performs the evolution, and returns and array of these values to be restored where the previous values were stored. These genetic functions wouldn't be able to perform their purpose if they operated on random ordered grades, so the voltages and the maximum difference values all needed to get sorted. Writing a sorting function took a while, as it works by taking in an array of all the maximum difference voltages and returning an array with the sorted order of the indices of the input array. The returned order array is used to sort the VB and VDB values, as well as the max diff array, so that the genetic functions can be performed on specific percentages of the population of each generation.

Testing allowed for optimizing how well the genetic functions were working in getting the gain of each running of the system to converge. Until getting consistent results, the algorithm

had to be adjusted; this meant, for example, changing what bits would be flipped (how much values would change from generation to generation), how many values to save across generations and not affect, and how many random new children to be generated every new generation. Tweaking these equations took a long time, as simulations couldn't be ran very quickly, but it was a lot of making adjustments and actively seeing how they would pay off. The end result is a system that converges to roughly the same gain every simulation, with the maximum difference in voltages between adjacent time measurements being well within 100mV on every run.

Testing

Testing for this project was making sure that everything ran accordingly and seeing the system converge consistently to the same value, and the testing process for this system is fairly straightforward. The script runs via the command ‘python run.py -r’ in Windows command line, which will run until the system converges. A lot of early testing was checking to make sure that the appropriate circuit was running or if it was iterating correctly. Later on, testing transitioned into looking at how well the genetic algorithm was working, by looking at each generation’s best child and the max difference value. The process of testing the system is very tedious, as each generation takes ~38 seconds to run, and waiting for the system to finish converging can take up to 25 minutes. Though information is posted about each child running in the command line, it’s difficult to extrapolate from the data until the system converges; much of the data displayed was manually entered into Excel sheets and information, such as variances and averages, can be calculated, along with graphs to display. Ideally this could’ve been performed automatically via the script, but there wasn’t enough time to integrate that functionality. Tests were performed on the amplifier with an input sine wave with amplitudes of 150mV and 100mV, with seeded (specifically chosen) initial children and random children, centered at 2.5V. Utilizing LTSpice through the application GUI, it can be confirmed that these are roughly the optimal values after sweeping all potential values for VB and VDB.

The next step for testing would be to test how well these voltages work on an actual, physical, differential amplifier. Unfortunately, there was not enough time to test this, so it can’t be confirmed that this simulation leads to correct results in real life application. It would be fair to assume that simulation would likely be roughly consistent to real life results, as LTSpice is a widely respected and used simulation tool.

Results and Discussion

Testing thoroughly showed that the system does converge with great consistency. Test runs for the 150mV amplitude sine wave converged within 3% of the average max difference of voltage values for every simulation. This convergence was consistent for seeded and random initial children values. Figures 1-4 below describe the convergence, and tables of data for each simulation can be found in the Appendix along with more graphs.

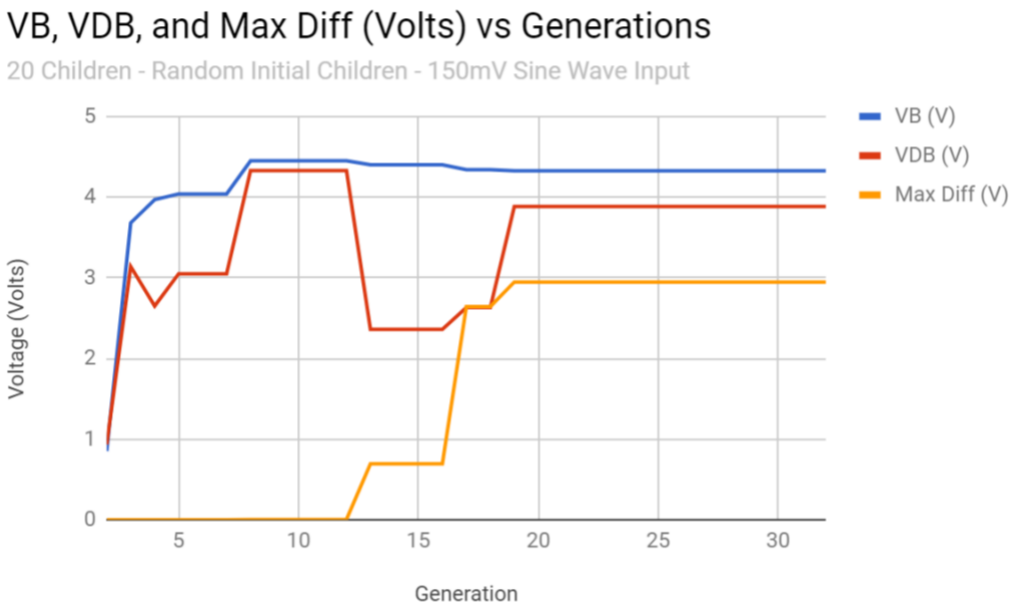

Figure 1: Graph of Voltages from Convergence of Gain – Random Children #1

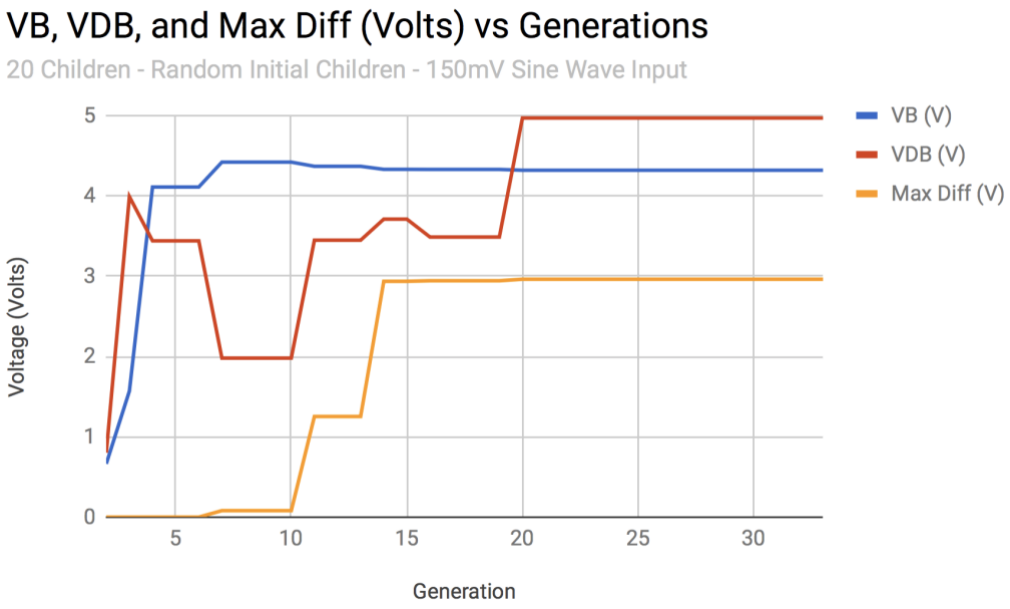

Figure 2: Graph of Voltages from Convergence of Gain – Random Children #2

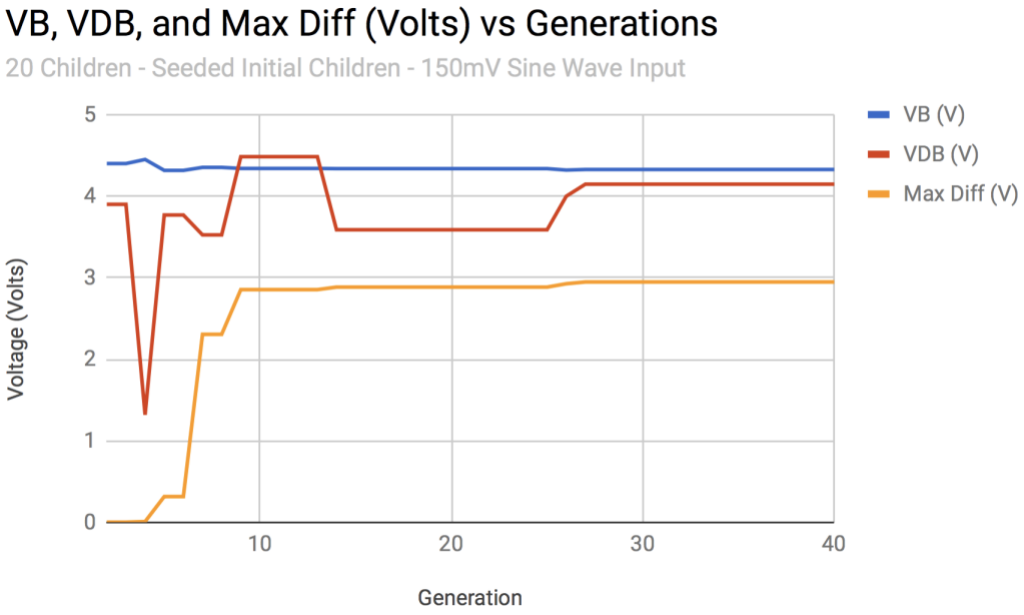

Figure 3: Graph of Voltages from Convergence of Gain – Seeded Children #1

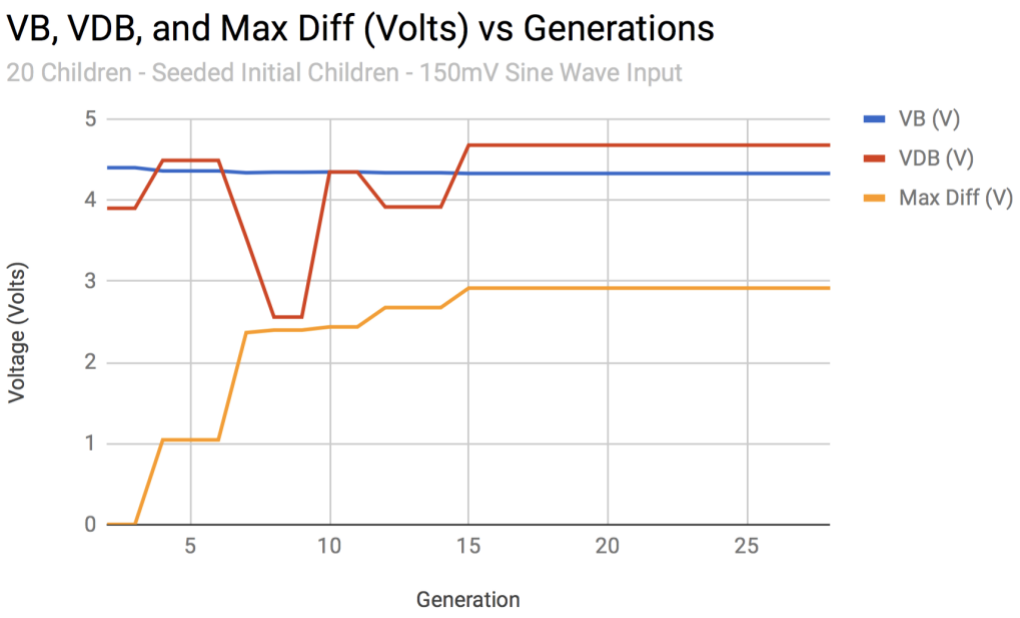

Figure 4: Graph of Voltages from Convergence of Gain – Seeded Children #2

Table 1: Averages and Variances of 150mV Sine Wave Input

	Max Diff (V)	VB (V)	VDB (V)	# Generations to Full Convergence
Average (Random)	2.930	4.326	4.435	29.67
Variance (Random)	2.35 E-3	91.0 E -6	0.146	49.57
Average (Seeded)	2.940	4.327	4.177	27.20
Variance (Seeded)	288 E -6	13.0 E -6	0.8999	9.360
Average (Overall)	2.934	4.326	4.274	28.13
Variance (Overall)	3.39 E -3	105 E -6	1.115	25.86

Figures 1-4 describe the evolution of the biasing voltages and the convergence of Max Diff, which is proportional to gain. In the seeded-child simulations, VB doesn't move around much, as one of the children is a relatively close solution for the particular amplifier. In comparison, the random-child simulations VB starts around 1; this is merely a coincidence, and the other figures and data tables in the appendix highlight that these just happened to be identical in starting points. VDB varies throughout much of the simulations, and this is explained more in

depth later on, but that's not an unreasonable result. Many of the simulations Max Diff values are right around 0V for a few simulations, and from there they either shoot up or slowly increase. This behavior is due to the genetic algorithm slowly optimizing on the children that yield better values. Once a good child is generated, the system reaches the best biasing child fairly after not within 10-15 generations for most simulations.

Looking at all the averages for the Max Diff value, it's evident that the averages (random, seeded, and overall) are very similar, and their variances are very small. The averages being close together indicates that, regardless of whether the initial children are random or seeded, they will converge to around the same gain; the small variances indicate that they will converge with high precision. This consistency indicates that the algorithm works very well at biasing to the maximum gain, which is the goal of the circuit.

The first biasing voltage, VB, has averages and that are all nearly equal and variances that are extremely small. This indicates, similarly to Max Diff, that VB will always converge to the same value. The second biasing voltage, VDB, converges to far less consistent values, with large variances, particularly for random and overall. Even though VDB fluctuates between different runs of the script, VB and Max Diff are extremely consistent; this shows that the value of VDB, though it affects the current through the tail, will have a more minor affect than VB in this arrangement. The idea that VB would have a greater impact on the gain than VDB makes sense. Looking deeper into this, VDB has an effect on the voltage of the base node of the NPN controlling the current of the current sink. VB is going to be biasing the PNP transistors, the active loads, which factors into the gain more significantly than the current through each side of the amplifier. Looking at the current equation of an NPN BJT, this makes sense:

$$I_C = I_S * e^{\wedge}(V_{BE}/V_T)$$

The saturation current, I_S, is going to be a constant for a given BJT model, and the same is true for the thermal voltage, V_T. If the main factors into this equation are I_C and V_{BE}, then a small change in the collector current will result in an extremely small change in the base-emitter voltage; a small change in this voltage is going to have a miniscule impact on any gain. The change in current will still affect the gain, but to such a small degree when compared to VB.

Conclusion

Using this genetic algorithm resulted in consistent convergence to maximum gain for differential amplifiers given a fixed input waveform. The variance of the maximum difference in voltage between 2 adjacent time points was 3.4 mV squared, which is extremely precise. This level of consistency in biasing the amplifier with the most gain is significant having implemented the system using a genetic algorithm. The gain the system was able to achieve was ~650V/V for a sine wave with a 150mV amplitude with a 2.5V offset and a frequency of 100kHz. The system was tested on other input waveforms and was able to achieve similar convergence. The second bias voltage VDB was less impactful on the overall gain, as it didn't converge to a consistent value, having a variance of 1.115V squared; this was due to its impact on the circuit being relatively minor in principle. However, this isn't a negative, as it didn't impact the goal of the project, biasing the amplifier for the largest possible gain. These results imply that it is possible to create systems to self-bias amplifying circuits, and that genetic algorithms are useful tools to find optimized solutions to difficult problems. The project would be considered a success as the desired solution was achieved.

References

[1] Garrison W. Greenwood and Andrew M. Tyrrell, *Introduction to Evolvable Hardware*. Hoboken, NJ: John Wiley and Sons, Inc, 2007.

[2] "joskvi/LTspice-cli: A command line tool for automating LTspice simulations," November, 2017. [Online]. Available: https://github.com/joskvi/LTspice-cli. [Accessed May 10, 2018].

Appendix

Senior Project GitHub Repository: https://github.com/slwhalen/SeniorProject

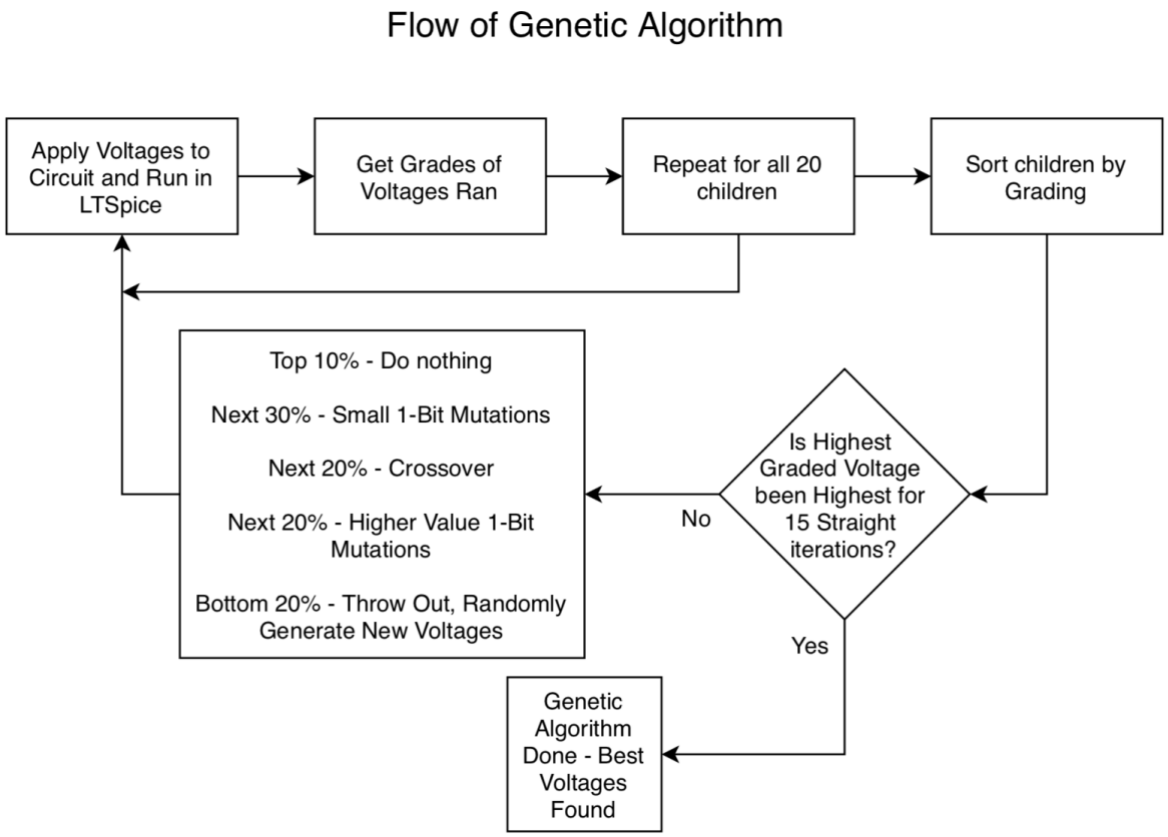

Figure 5: Flow of Genetic Algorithm

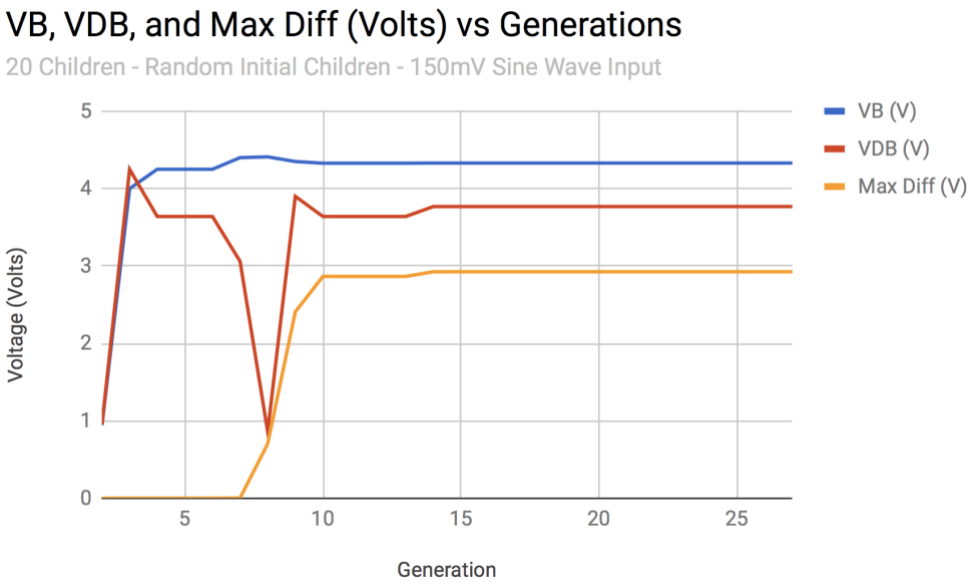

Figure 5: Graph of Voltages from Convergence of Gain – Random Children #3

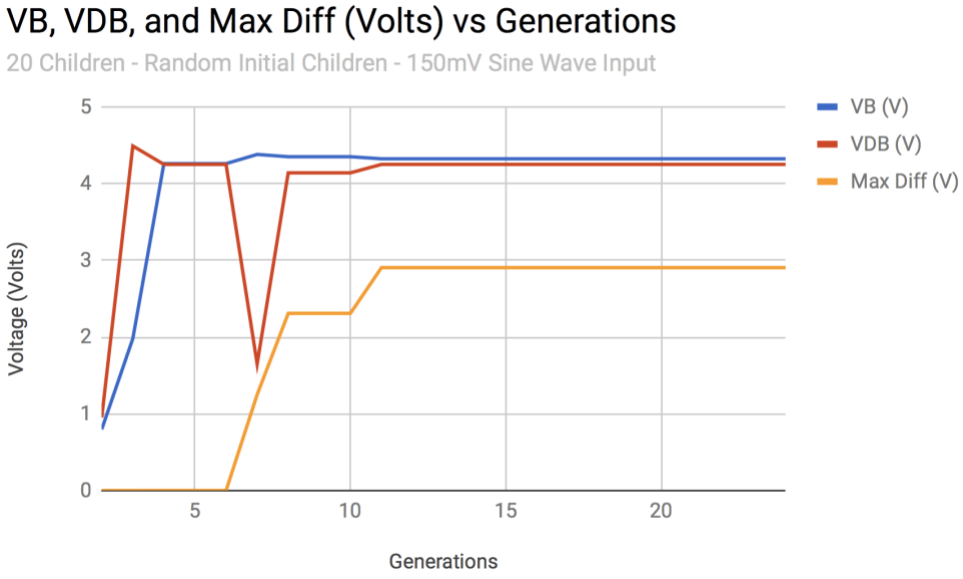

Figure 6: Graph of Voltages from Convergence of Gain – Random Children #4

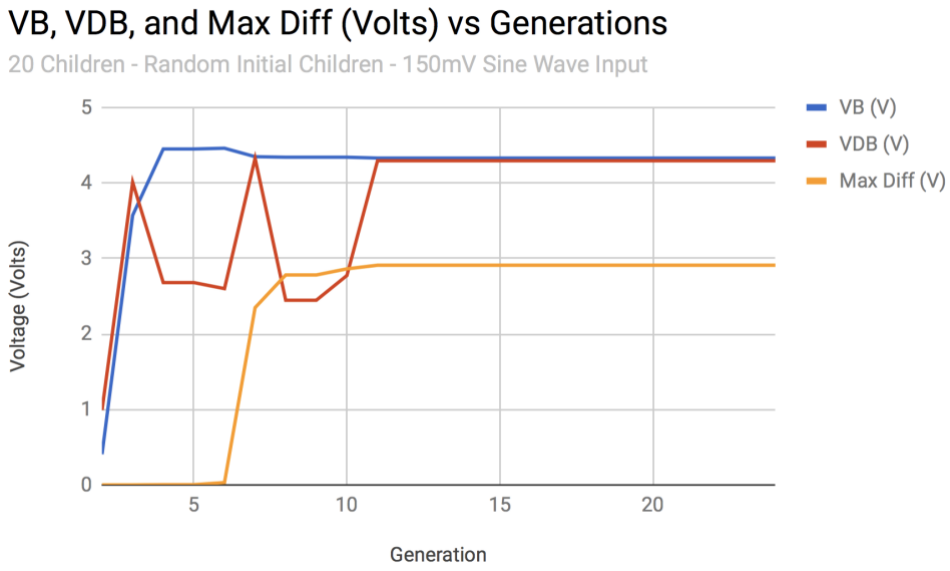

Figure 7: Graph of Voltages from Convergence of Gain – Random Children #5

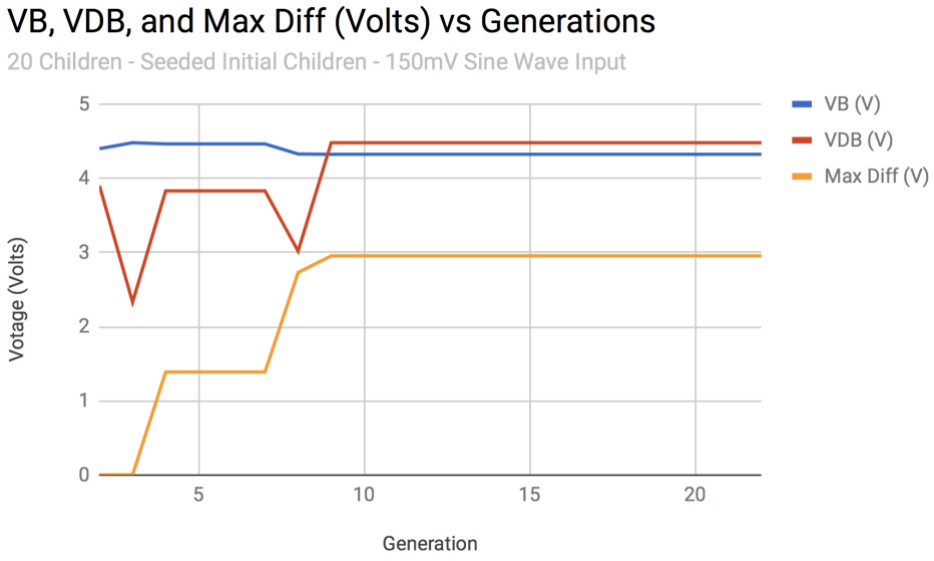

Figure 8: Graph of Voltages from Convergence of Gain – Seeded Children #3

Table 2: Best Child Data Points Corresponding to Graph of Seeded Children #1

VB (V)	VDB (V)	Max Diff (V)	Generation
4.4	3.9	0.002888	1
4.4	3.9	0.002888	2
4.36	4.49	1.045	3
4.36	4.49	1.045	4
4.36	4.49	1.045	5
4.3375	3.5375	2.367	6
4.3438	2.559	2.3984	7
4.3438	2.559	2.3984	8
4.347	4.347	2.4375	9
4.347	4.347	2.4375	10
4.3375	3.916	2.676	11
4.3375	3.916	2.676	12
4.3375	3.916	2.676	13
4.329	4.678	2.916	14
4.329	4.678	2.916	15
4.329	4.678	2.916	16
4.329	4.678	2.916	17
4.329	4.678	2.916	18
4.329	4.678	2.916	19
4.329	4.678	2.916	20
4.329	4.678	2.916	21
4.329	4.678	2.916	22
4.329	4.678	2.916	23
4.329	4.678	2.916	24
4.329	4.678	2.916	25
4.329	4.678	2.916	26
4.329	4.678	2.916	27
4.329	4.678	2.916	28

Table 3: Data Corresponding to Graph of Seeded Children #2

VB (V)	VDB (V)	Max Diff (V)	Generation
4.4	3.9	0.002888	1
4.4	3.9	0.002888	2
4.45	1.32	0.01005	3
4.316	3.769	0.3166	4
4.316	3.769	0.3166	5
4.353	3.525	2.306	6
4.353	3.525	2.306	7
4.34	4.484	2.854	8
4.34	4.484	2.854	9
4.34	4.484	2.854	10
4.34	4.484	2.854	11
4.34	4.484	2.854	12
4.3375	3.588	2.885	13
4.3375	3.588	2.885	14
4.3375	3.588	2.885	15
4.3375	3.588	2.885	16
4.3375	3.588	2.885	17
4.3375	3.588	2.885	18
4.3375	3.588	2.885	19
4.3375	3.588	2.885	20
4.3375	3.588	2.885	21
4.3375	3.588	2.885	22
4.3375	3.588	2.885	23
4.3375	3.588	2.885	24
4.32	3.999	2.926	25
4.327	4.147	2.948	26
4.327	4.147	2.948	27
4.327	4.147	2.948	28
4.327	4.147	2.948	29
4.327	4.147	2.948	30
4.327	4.147	2.948	31
4.327	4.147	2.948	32
4.327	4.147	2.948	33
4.327	4.147	2.948	34
4.327	4.147	2.948	35
4.327	4.147	2.948	36
4.327	4.147	2.948	37
Continuation of Same Data Point until Generation 40			

Table 4: Data Corresponding to Graph of Seeded Children #3

VB (V)	VDB (V)	Max Diff (V)	Generation
4.4	3.9	0.002888	1
4.48	2.33	0.003163	2
4.464	3.831	1.389	3
4.464	3.831	1.389	4
4.464	3.831	1.389	5
4.464	3.831	1.389	6
4.328	3.019	2.73	7
4.324	4.48	2.955	8
4.324	4.48	2.955	9
4.324	4.48	2.955	10
4.324	4.48	2.955	11
4.324	4.48	2.955	12
4.324	4.48	2.955	13
4.324	4.48	2.955	14
4.324	4.48	2.955	15
4.324	4.48	2.955	16
4.324	4.48	2.955	17
4.324	4.48	2.955	18
4.324	4.48	2.955	19
4.324	4.48	2.955	20
4.324	4.48	2.955	21
4.324	4.48	2.955	22

Table 5: Data Corresponding to Graph of Random Children #1

VB (V)	VDB (V)	Max Diff (V)	Generation
0.85	0.934	0.000004972	1
3.68	3.14	0.00004813	2
3.97	2.65	0.0000561	3
4.04	3.05	0.00007945	4
4.04	3.05	0.00007945	5
4.04	3.05	0.00007945	6
4.45	4.33	0.002712	7
4.45	4.33	0.002712	8
4.45	4.33	0.002712	9
4.45	4.33	0.002712	10
4.45	4.33	0.002712	11
4.4	2.36	0.6921	12
4.4	2.36	0.6921	13
4.4	2.36	0.6921	14
4.4	2.36	0.6921	15
4.34	2.63	2.643	16
4.34	2.63	2.643	17
4.327	3.883	2.947	18
4.327	3.883	2.947	19
4.327	3.883	2.947	20
4.327	3.883	2.947	21
4.327	3.883	2.947	22
4.327	3.883	2.947	23
4.327	3.883	2.947	24
4.327	3.883	2.947	25
4.327	3.883	2.947	26
4.327	3.883	2.947	27
4.327	3.883	2.947	28
4.327	3.883	2.947	29
4.327	3.883	2.947	30
4.327	3.883	2.947	31
4.327	3.883	2.947	32

Table 6: Best Child Data Points Corresponding to Graph of Random Children #2

VB (V)	VDB (V)	Max Diff (V)	Generation
0.944	0.956	0.000004162	1
4	4.25	0.0001066	2
4.25	3.64	0.0003741	3
4.25	3.64	0.0003741	4
4.25	3.64	0.0003741	5
4.4	3.06	0.003271	6
4.41	0.86	0.704	7
4.35	3.9	2.408	8
4.328	3.64	2.866	9
4.328	3.64	2.866	10
4.328	3.64	2.866	11
4.328	3.64	2.866	12
4.33	3.768	2.926	13
4.33	3.768	2.926	14
4.33	3.768	2.926	15
4.33	3.768	2.926	16
4.33	3.768	2.926	17
4.33	3.768	2.926	18
4.33	3.768	2.926	19
4.33	3.768	2.926	20
4.33	3.768	2.926	21
4.33	3.768	2.926	22
4.33	3.768	2.926	23
4.33	3.768	2.926	24
4.33	3.768	2.926	25
4.33	3.768	2.926	26
4.33	3.768	2.926	27

Table 7: Best Child Data Points Corresponding to Graph of Random Children #3

VB (V)	VDB (V)	Max Diff (V)	Generation
0.798	0.952	0.000004017	1
1.98	4.49	0.00004713	2
4.26	4.25	0.0005276	3
4.26	4.25	0.0005276	4
4.26	4.25	0.0005276	5
4.38	1.65	1.25	6
4.35	4.14	2.309	7
4.35	4.14	2.309	8
4.35	4.14	2.309	9
4.323	4.25	2.905	10
4.323	4.25	2.905	11
4.323	4.25	2.905	12
4.323	4.25	2.905	13
4.323	4.25	2.905	14
4.323	4.25	2.905	15
4.323	4.25	2.905	16
4.323	4.25	2.905	17
4.323	4.25	2.905	18
4.323	4.25	2.905	19
4.323	4.25	2.905	20
4.323	4.25	2.905	21
4.323	4.25	2.905	22
4.323	4.25	2.905	23
4.323	4.25	2.905	24

Table 8: Best Child Data Points Corresponding to Graph of Random Children #4

VB (V)	VDB (V)	Max Diff (V)	Generation
0.406	0.992	0.000004177	1
3.571	4.01	0.00005986	2
4.45	2.68	0.003081	3
4.45	2.68	0.003081	4
4.46	2.6	0.03093	5
4.348	4.326	2.348	6
4.341	2.446	2.781	7
4.341	2.446	2.781	8
4.341	2.774	2.862	9
4.33	4.294	2.91	10
4.33	4.294	2.91	11
4.33	4.294	2.91	12
4.33	4.294	2.91	13
4.33	4.294	2.91	14
4.33	4.294	2.91	15
4.33	4.294	2.91	16
4.33	4.294	2.91	17
4.33	4.294	2.91	18
4.33	4.294	2.91	19
4.33	4.294	2.91	20
4.33	4.294	2.91	21
4.33	4.294	2.91	22
4.33	4.294	2.91	23
4.33	4.294	2.91	24

Table 9: Best Child Data Points Corresponding to Graph of Random Children #5

VB (V)	VDB (V)	Max Diff (V)	Generation
0.664	0.802	0.000002217	1
1.57	3.99	0.00003973	2
4.11	3.44	0.0001173	3
4.11	3.44	0.0001173	4
4.11	3.44	0.0001173	5
4.42	1.98	0.08068	6
4.42	1.98	0.08068	7
4.42	1.98	0.08068	8
4.42	1.98	0.08068	9
4.368	3.448	1.253	10
4.368	3.448	1.253	11
4.368	3.448	1.253	12
4.33	3.71	2.936	13
4.33	3.71	2.936	14
4.329	3.487	2.943	15
4.329	3.487	2.943	16
4.329	3.487	2.943	17
4.329	3.487	2.943	18
4.319	4.969	2.962	19
4.319	4.969	2.962	20
4.319	4.969	2.962	21
4.319	4.969	2.962	22
4.319	4.969	2.962	23
4.319	4.969	2.962	24
4.319	4.969	2.962	25
4.319	4.969	2.962	26
4.319	4.969	2.962	27
4.319	4.969	2.962	28
4.319	4.969	2.962	29
4.319	4.969	2.962	30
4.319	4.969	2.962	31
4.319	4.969	2.962	32
4.319	4.969	2.962	33