A Framework for Tuning Multirotor Flight Control Systems with Accelerated Genetic Algorithms

Brandon Braun
brandonbraun653@gmail.com

Abstract—**Manually tuning the flight control system of a multirotor to respond exactly as a pilot desires can be a time consuming task. In the scope of this paper, the multirotors in question are limited to those typically associated with quadcopter racing drones. These are highly dynamic systems that have traditionally required the pilot to manually tune the PID based control loops for Roll, Pitch, and Yaw. While there are general guidelines available to instruct the pilot on how to perform this procedure efficiently, they only provide rough approximations because large variations exist in system dynamics between one multirotor and the next. This is due primarily to the various myriad of possible combinations of motors, propellers, battery specs, ESCs, aerodynamics, and many other factors. Extracting the best performance usually is only possible after spending many hours (and batteries) fine tuning the control loop. This paper aims to provide pilots a possible solution to this problem by taking advantage of the Genetic Algorithm for an offline speed-up of the tuning process.**

I. Introduction

One of the most frustrating aspects of building a multirotor is tuning the flight control system, most commonly PID based, to perform within the pilot's specifications. In the case of pilots who fly highly dynamic racing quadcopters, the performance tolerances are tight and difficult to achieve. This problem is compounded by the complex system dynamics that arise when combining various types of motors, propellers, air-frames, and other components typically found on a multirotor. There is a vast array of possible selections for each component, each with slightly different characteristics, that make it difficult to design a "one size fits all" tuning approach. There are simply too many complex variables at play. From here on out, a generic multirotor system will be referenced interchangeably with drone and quadcopter.

Standard flight control software running on the quadcopter allow the pilot to configure many parameters that influence unique flight characteristics, but none are as important as the PID coefficients governing the roll, pitch, and yaw control loops. Tuning these values are what consume the majority of the pilot's time. Depending upon how experienced the pilot is with the tuning process, this could take up to several hours to fully hone the system to desired specs. The two primary bottlenecks in this process are battery life and the rate at which PID coefficients can be adjusted. The former is relatively fixed while the latter can be optimized heavily through software. In the case of this work, the tackled problem is to develop a Genetic Algorithm based software tool that increases the rate at which PID coefficients can be tested while at the same time converging upon pilot given performance constraints. The unique contributions of the developed approach are as follows:

1) Real Time Performance

The primary idea behind this software is for the pilot to have the capability to tune their multirotor significantly faster than traditional approaches. This requires that the GA be capable of quickly converging to a good enough solution. Due limited on-board battery capacity, solutions must be found within a few seconds or less in order to test many configurations within the duration of the handful of flights available to most pilots.

2) Multipath Feedback

The full version of this software tool is capable of incorporating actual performance data from a test flight in real time in addition to any user provided mathematical model. This enables the possibility for full abstraction of the complex system dynamics present in the multirotor and removes the need to develop unique models for various components, such as motors or props. In short, the proposed solution should be able to tune virtually any multirotor, within the limits of physics.

3) Abstracted & Configurable Architecture

No two multirotors are the same and some are more difficult to tune than others. Because of this, a single GA approach that may work well on one system may not work so well on a second system with different components. In anticipation of this likely problem, the solution architecture was designed from the ground up with the capability to adapt the algorithm at each step. This allows the pilot to select the best approach for their system and, if necessary, provide code to use their own methodology.

II. Literature Review

When searching for information on how this class of problem, GA optimization of PID control loops, has been tackled in the past, it quickly becomes apparent that there are not many variations on the approaches used. The earliest papers emerged in the 1990's from Wang [6] and Mitsukura et al. [7]. Both of their works are representative of how the majority of later authors structure their variants on GA based PID tuning. The idea is that given some system plant function with an associated PID control loop and unknown Kp, Ki, Kd constants, it is possible to tune the system using GA. An example of such a setup from [6] is shown in Figure 1:

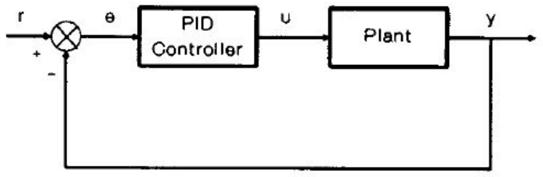

Figure 1. Classical PID Control Loop

Later authors tend to take minor variants of this structure and investigate how well the GA performs with specific test cases. In the paper by Meena and Devanshu [3], the authors compare performance of the classical Zieger-Nichols PID tuning method to the GA with good results. Gupta et al. applies GA to plant systems that have time delays [1], Kwok and Sheng solve optimal tuning of a robot arm PID control loop [2], and Jie and Ding optimize the dynamic response of an airplane engine [4]. Recently, Kumar et al. developed an approach that utilizes a multi-objective version of the GA to tune a PID control system [5].

What is immediately apparent from a literature search is 1) successful tuning of PID control loop systems with a Genetic Algorithm is possible and can actually perform quite well, 2) only specific, well defined test cases have been evaluated, 3) nearly all approaches require use of a full mathematical model of the system to be tuned, and 4) no attempt has been made, to the authors knowledge, to apply a parallel version of the GA to this tuning problem.

This leads to the impression that there is a gap of knowledge and/or a lack of interest for further development of this particular topic. The author would like to posit that perhaps more work in this area could be accomplished if a generalized framework existed that is capable of testing out many GA approaches while simultaneously supporting highly detailed and highly abstracted system dynamics. As such, most of the work done in this project is new from a process perspective.

III. Detailed Solution Analysis

Due to the perceived lack of work done in literature regarding more generalized GA based tuning methods, it is believed that an entirely new approach is needed to tackle this problem. As a test case, a framework is attempted to be developed that will be capable of tuning multirotors under a wide range of possible system dynamics while at the same time catering the software to both the well educated engineer or researcher as well as to the layman pilot. It is an ambitious goal to say the least. What follows in the next few sections is a description of how such a possible framework could be set up.

A. Pilot Tuning Without GA

As mentioned previously, not many pilots have the skill to optimally tune PID control loops quickly. This results in the pilot being forced to take a trial and error approach to tuning their system for peak performance. This process might look like the following:

1) *Apply a chosen set of PID parameters*
2) *Fly a test course and evaluate performance manually*
3) *Replace the battery*
4) *Determine a new set of PID values to test*
5) *Repeat (1) – (4) until tuned to pilot satisfaction*

There are several disadvantages to the trial and error approach. First, it requires that the pilot use many batteries. The Lithium Polymer batteries most commonly used by pilots are not cheap and as such, most only own 2-5 packs. Each pack lasts between two to three minutes, thus limiting the total time spent by the pilot trying out possible Kp, Ki, Kd tuning settings. In addition to this, each battery takes between 30 to 45 minutes to recharge before being capable of powering another flight. Second, it requires that the pilot have intimate knowledge of PID control systems and how changing various tuning parameters can affect the entire system performance. In a broad sense, most pilots do not have this kind of knowledge. The standard approach, which may or may not work well for a possible drone configuration, is to follow general rules of thumb in order to tune the system. Third, with a larger number of flights needed to tune properly, there is an increased chance that the pilot will crash the multicopter and break a component. Most of the components that are likely to break in a crash are not cheap, thus a longer time spent tuning yields an increased risk of financial expense. In a worst-case scenario, an inexperienced pilot could apply incorrect PID tuning values, cause a hard crash, and break critical components such as the motor or flight controller electronics. Such an event would require the pilot to wait several weeks before new parts come in.

While many of these disadvantages are naturally present when flying a drone, there is an increased risk of causing an occurrence, simply because of improper tuning making it more difficult to pilot. A better tuning method is needed.

B. Pilot Tuning With Offline GA

The proposed solution is to utilize an engineering version of the GA in order to remove the pilot's guesswork when trying to figure out optimal PID settings. Ideally, if the GA algorithm has an accurate model of the drone dynamics, it should be able to quickly find a usable solution. It is imperative in the tuning cycle that any work done by the GA is in an offline manner. In other words, the GA is not present in the actual control loop. This would not work for many reasons, but the most pressing concern is the overall loop response time. Including the GA in the control loop would reduce it to a discrete-time system with a sampling/control rate of several seconds, at a minimum. No drone would ever be flyable in these conditions. Therefore, the tuning process with an offline GA would look similar to the following:

1) *Fly a test course with an initial set of PID values*
2) *Upload recorded flight data wirelessly to a remote computer hosting the GA tuning software*
3) *While the software is running, change the drone battery*

4) *After a new solution is found, wirelessly upload the new PID settings*
5) *Repeat (1) - (4) until tuned to pilot satisfaction*

While the steps here are very similar to those in section (A), there is one major difference. New PID settings are found and then uploaded to the drone all within the time span taken to change the battery, and without intervention by the pilot. This feature is what makes it possible to improve the rate at which a drone's control system can be optimized.

C. Selection of System Model

The performance of such an optimization will however be highly dependent upon the system model. If the model incorrectly represents the dynamics of the drone, there is little chance of intelligently producing optimized PID values. Unfortunately this problem is not trivial, even for those who have experience with the complex mathematics needed to produce an accurate enough model. In response to this problem, the GA software endeavors to support several different kinds of models, each of which cater to a specific kind of pilot.

The first model type is a full mathematical description in state space form. This can appear as a linear time invariant/variant or nonlinear time invariant/variant system, generalized as:

$$x[k+1] = Ax[k] + Bu[k] \quad (1)$$
$$y[k] = Cx[k] + Du[k] \quad (2)$$

This option is for the most advanced users and will likely only be used by researchers or engineering types. The common pilot will not need to venture into this territory however.

The second model type is far more user friendly in that the model will not need to be described directly and the pilot's only involvement is by flying the drone. Instead of a state space model, a Neural Network (NN) will be used to learn the drone dynamics over repeated flights, given recorded input control commands and output of the flight control system. Then, during operation of the GA software, a range of inputs and various Kp, Ki, Kd values can be given to the NN and the output successfully approximated. The goal is to remove the need to meticulously model all the various combinations of components that change the drone response characteristics. Obviously this is far easier said than done.

D. Selection of Fitness Function

The selection of the fitness function for this proposed system is currently extremely similar to the multi-objective optimization idea proposed by Kumar et al. [?], where the GA is attempting to tune the classical performance characteristics of a system response to a given input, usually a step function. They are:

1. *Rise Time*
2. *Percent Overshoot*
3. *Settling Time*
4. *Steady State Error*

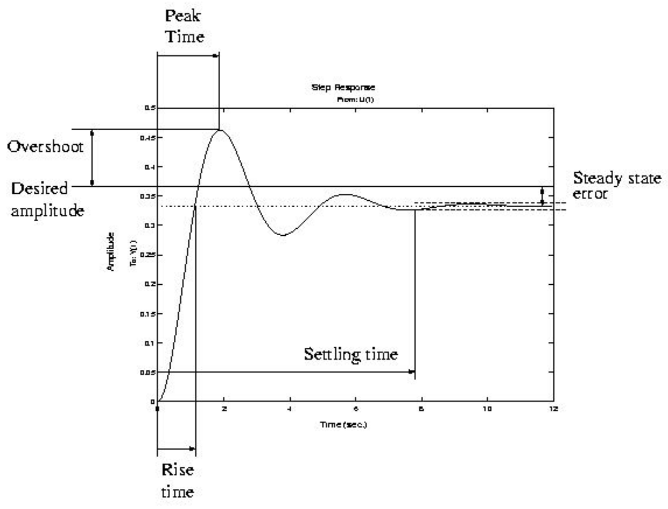

Figure 2. Example Step Response Characteristics

Typically these performance characteristics are competing against each other in that by trying to raise the performance of one parameter, it inadvertently causes one or more other parameter performance to degrade.

Currently, the user is allowed to specify a custom value for each of the above mentioned performance characteristics as a target goal for the GA to tune the system to. After evaluation of the system model, each chromosome is given a subscore for each parameter on the range of 0.0 to 1.0 based on how well it met a user defined goal. A score of 0.0 indicates the solution is far away from the goal and a score of 1.0 indicates the goal was satisfied. Once all the subscores have been assigned, they are averaged together to produce one global score that characterizes the overall fit of that particular chromosome.

E. Convergence Criteria

One of the unique aspects of this engineering version of the GA is that the algorithm is not allowed to continue on for a long time period to find the best possible solution. While most of the approaches found in literature are concerned with how well a specific type of algorithm performs in finding the global minimum or maximum in a slew of benchmark tests, this paper is concerned primarily with finding a good enough solution that meets user defined criteria in a specified time window.

The user is able to specify several kinds of convergence criteria. For convenience they are listed here briefly and dealt with more carefully in the Software Architecture section. Currently the supported criteria are based on 1) a good solution has been found, 2) a generation limit has been reached, or optionally 3) a time limit has expired.

IV. Software Architecture

The software architecture is of critical importance for this endeavor due to the rather unique goals it must accomplish. To reiterate, the two most difficult challenges facing the software design is real-time performance and the ability to dynamically reconfigure the Genetic Algorithm at each step. The real time performance is necessary so that the drone pilot is not spending large amounts of down time from flying. Due to the relative nature of defining "real-time", it is taken in this context to mean under 10-15 seconds. This allows the pilot time to land from a test flight and swap batteries without any noticeable hindrance from the GA software. The ability for dynamic reconfiguration is deemed important due to the inherent problems of hard coding a single algorithmic approach. It is well known throughout the academic community that a single "elite" GA solution capable of solving every problem to satisfaction does not exist. It would be extremely limiting, frustrating, and arrogant of the author if this software were only capable of implementing a single approach.

This section will walk through the structure of version 1.0 of the software, starting from a high level overview and then working in towards a reasonable amount of detail. The sections will be handled in the following manner:

A. Choice of Language
B. Acceleration of the GA
C. Global Configuration Options
D. Main Algorithm Pseudo Code
E. Population Initialization
F. Model Evaluation
G. Fitness Function
H. Population Filtering
I. Parent Selection
J. Breeding
K. Mutation
L. Convergence Checking

While there are a perhaps overwhelming number of sections, it is hoped that the reader will find it useful for explaining the thought behind the software design. The goal for such detail is to allow the reader to go to the actual source code and grasp at a high level why certain sections were written in a specific manner. In addition, it is hoped that future researchers will benefit by not having to reinvent the wheel and can use this work as a stepping stone towards an improved system.

A. Choice of Language

The primary choice of language is C++ because of its speed advantages over other languages like Python, Java, or Matlab. While this does create some difficulties in terms of lack of pre-built features into the language for ease-of-use, the powerful advantage of being able to manipulate data at a low level, close to the CPU, is highly desirable. The trade off is that the code is more dense and perhaps more complicated to read, but offers significantly increased performance gains and allows the developer to tweak the code to suit his or her needs. In experimental software tools, this kind of flexibility is crucial.

B. Acceleration of the GA

The genetic algorithm has many opportunities to be accelerated. Typically this occurs when a relatively complex operation must be applied to each individual population member before the program continues. Rather than run these complex operations one after the other, it is desirable to run as many in parallel as possible. Most frequently this occurs during the model evaluation and fitness function calculation steps. More information on how this is specifically achieved for each of these steps is described in their respective sections below.

From a hardware perspective, algorithmic speed up is achieved by splitting out tasks to multiple threads on the CPU. In the case of the 8 core Ryzen 1700X processor that was used in creating this software, there are a total of 16 threads that can be utilized to run in parallel at any given time. Within reason, the more threads that are available for parallelism, the greater the potential speed up of the algorithm. The total speed up achieved is also highly dependent upon the number of members in the population. If there is a small number, the benefit of using parallel execution is not very high. Multi threaded performance nearly approximates single threaded performance in this case. However, if the population is large, multi threaded execution is the clear winner over single-threaded. In some early test cases, there was observed at least a 10x speed up, sometimes even greater. For the extremely large population size it might be beneficial to use a GPU for computation in addition to using a CPU. The primary bottlenecks in this case would be GPU single threaded execution time and memory transfer rates between CPU and the GPU. This is an area open to further research.

The primary tool used to create and manipulate threads is the C++ Boost library. Normally a developer would need to specify unique methods for handling multi-threading depending upon which operating system is being used. Boost removes that problem and provides an API for use on Linux, Windows, or Mac. This helps enormously with porting of the GA tuner software to multiple platforms and simplifies development of further research.

C. Global Configuration Options

There are a large number of global configuration options available to the end user to help customize the runtime operation of the genetic algorithm software. These options range from as simple as logging and verbosity of the program to all the way to changing how the algorithm runs at each fundamental step.

The most commonly used functionalities will be those to modify how the algorithm executes at each step. Using a set of enumerations, the programmer can specify a specific methodology to apply in the setup and configuration stage of the software. For example, in the Parent Selection step, the current methodologies for selecting which chromosomes will mate are supported:

1. *GA_SELECT_RANDOM*

2. *GA_SELECT_RANKED*
3. *GA_SELECT_ROULETTE*
4. *GA_SELECT_STOCHASTIC_SAMPLING*
5. *GA_SELECT_TOURNAMENT*
6. *GA_SELECT_ELITIST*

These are all very popular methodologies for parents selection in literature. Should the user find that one particular methodology is not working well for a particular problem, all that must be done is simply switch out a variable name to select a different option. If the currently supported options do not work, novel new methods can be added easily by following certain architectural rules and data structures. This is a powerful advantage over many of the other hard coded algorithms which require rewriting the base software in order to implement new features.

D. Main Algorithm Pseudo Code

The main algorithm is extremely simple despite all the complexities described thus far. It takes the form of the classical GA but with one additional step: Population Filtering. This new feature is described in a later section. The pseudo code is displayed below:

```
initializePopulation();
while (currentStatus == GA_OK)
{
    evaluateModel();
    evaluateFitness();
    filterPopulation();
    selectParents();
    breedGeneration();
    mutateGeneration();
    checkConvergence();
}
```

The currentStatus variable is used to indicate whether or not another iteration of the GA needs to be completed. It can be modified at any time by the functions listed to report on how the algorithm is progressing. With this core structure in mind, the steps will now be broken down and discussed individually.

E. Population Initialization

The population initialization step is primarily concerned with generating a full set of randomized chromosomes for each population member. The particular representation choice for the chromosomes is both as a floating point real number and as a mapped unsigned 32 bit integer. In this stage of the algorithm, the chromosomes are in their floating point form and a total of 3 unique Kp, Ki, Kd values are given to each population member.

The unique numbers are generated from a uniform real Mersenne-Twister engine over a range of values specified by the user to the main GA interface.

F. Model Evaluation

Depending upon the type of model being used, there are different methods for evaluation. Common to both types of models is a PID controller that handles generation of a proper control signal given a reference and a population member's Kp, Ki, Kd values.

If given a state space form, the model is simulated in response to some kind of input using a numerical integration solver. The input could be a Step, Ramp, Quadratic, or even custom function. If given a trained neural network, the model is simulated using a pre-recorded combination of control inputs that mimic those actually performed when flying the drone. The output data for either system is then stored in a container for processing by the fitness function.

G. Fitness Function

The fitness function for this approach involves two main steps. 1) Calculate the performance metrics from the simulation results data in the Model Evaluation step. 2) Assign a fitness score.

In the first step, a custom algorithm was developed to determine what the rise time, percent overshoot, settling time, and steady state error was of the simulation data. The methodology is based on capturing the points where slope changes sign and on the application of several logical tests.

In the second step, a fitness score is assigned based on an averaging approach. Given a set of known performance characteristics from simulation data and a set of known user performance goals, a fitness value is assigned in proportion to how well the goal was met. If the characteristic falls within the user's goal and tolerance specification, it is given a perfect fit score, or 1.0. However, if the characteristic falls outside of the user's goal and tolerance specification, the fitness score is derated by the equation:

$$fe^{-n} \tag{3}$$

where f is the perfect fit value of 1.0 and n is the absolute error between the goal and characteristic. In practice this has been found to work well.

H. Population Filtering

Population filtering is a new step being introduced to the GA algorithm that is intended to model some of the more unfortunate aspects of natural selection: survival of the fittest. At each iteration, the population will be thinned out by a random percentage. The idea is to capture the effects of deaths due to events like natural disasters, old age, predators, etc that would normally have an impact in real evolutionary biology. Population members are selected by how well they performed during the last fitness evaluation. If one member scores very well, there is a low probability that their genetic material will be selected to die off. Conversely, if one scores poorly, there is a much higher chance of death. By doing this, it is possible to more quickly get rid of genetic data that does not help converge to a good solution. Because the population size is fixed for this version of GA, the removed members must be replaced. This is done in a similar manner to the Population Initialization step.

I. Parent Selection

The parent selection methodology is performed based on a user given parameter. This and the Breeding and Mutation sections below take full advantage of the dynamic reconfiguration capabilities of the architecture in order to implement many different popular mechanisms from literature. If desired, the user is able to add their own methodology to the list. Because entire papers could be written on each of the methodologies listed only their names will be given in the following sections. Currently, the supported methods for parent selection are:

1. GA_SELECT_RANDOM
2. GA_SELECT_RANKED
3. GA_SELECT_ROULETTE
4. GA_SELECT_STOCHASTIC_SAMPLING
5. GA_SELECT_TOURNAMENT
6. GA_SELECT_ELITIST

J. Breeding

The breeding methodology is performed based on a user given parameter. Currently the supported methods are:

1. GA_BREED_SIMPLE_CROSSOVER
2. GA_BREED_DYNAMIC_CROSSOVER
3. GA_BREED_FIXED_RATIO_CROSSOVER
4. GA_BREED_SIMULATED_BINARY_CROSSOVE R

K. Mutation

The mutation methodology is performed based on a user given parameter. Currently the supported methods are:

1. GA_MUTATE_BIT_FLIP
2. GA_MUTATE_ADD_SUB

The Bit Flip method uses the mapped unsigned integer form of chromosome to mutate individual bits. The Add Sub method uses the real floating point form of the chromosomes to mutate through addition and subtraction of a small number ε.

L. Convergence Checking

In the convergence checking section, there are a total of three different operations that can be performed. First, the solutions from that iteration are filtered in order to figure out which population member produced the best results. When the best solution is found, it is pushed into a holding container for further processing by an elitist algorithm, if enabled. In addition, an option can be enabled for these results to be displayed to the user to allow for feedback on how the algorithm is progressing. Second, the algorithm is deemed converged when a certain iteration limit has been met. The user can set a predefined threshold how many iterations the algorithm is allowed to perform before exiting, regardless of whether or not a good solution has been found. Third, the algorithm will exit based on whether or not the global fitness score of the highest performer exceeds a predefined value, currently hard coded to 0.9 or 90% fitness.

V. Performance Results & Discussion

Before the results can be presented, an accurate definition of what high performance actually means in the context of this application is needed. The most important characteristics here are 1) how quickly the GA can converge to a well fit solution, and 2) how reliable it is at finding a solution each time a tuning run is performed. Because there is not a large amount of time spent trying to find the absolute best set of solutions for a given system, generating an optimal pareto front and associated diversity metric is not considered to be too important for the first iteration of the software. The side effect of this focus is that the class of solutions typically found over many trials may tend to fall into a more constrained set.

A. Testing Methodology

Because most of the time for this project was dedicated towards establishing a solid framework from which to build more complex capabilities, only a simple test case with the state space model type has been considered thus far to demonstrate that the system can indeed tune properly. The particular model was chosen nearly at random and does not actually describe the physical properties of a specific system, however the matrix values were tuned slightly to allow a PID controller to find an optimal step response in the range that one might expect from a drone. The specific model is shown below:

$$x[k+1] = \begin{bmatrix} -8.202 & -2.029 \\ -0.149 & -3.250 \end{bmatrix} x[k] + \begin{bmatrix} 1.14 \\ -1.23 \end{bmatrix} u[k] \quad (4)$$

$$y[k] = [1\ 0]x[k] \quad (5)$$

Using the above model, several tests were performed. First, an optimal result was generated using the Matlab PID Tuner Toolbox as a way of creating a simple benchmark. Then, the single threaded and multithreaded versions of the GA were tested by tuning three PID control system loops per run (representing roll, pitch, and yaw) under population sizes of 5, 10, 25, 50, 100, and 250. For each population size, the two versions were allowed to run 30 times to gain an idea of average performance.

B. Benchmark Solution

The benchmark solution was generated using the PID Tuner Toolbox in Matlab. Using the same model as in (4) and (5), the toolbox was able to find an optimal solution to the step response as shown in Figure 3. The given criteria were:

1. Rise time < 0.25 sec
2. Settling Time < 1 sec
3. Overshoot < 10%
4. Steady State Error < 0.05

The associated solution's PID tuning values were:

Kp = 12.05, Ki = 68.38, Kd = 0.0

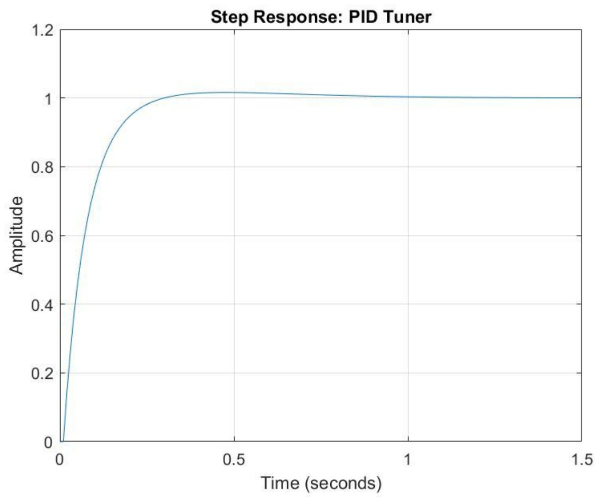

Figure 3. Matlab PID Tuner Toolbox Benchmark

C. GA Single Threaded Performance

Single threaded mode for this software is only used to demonstrate the advantages of using multi threading to speed up the GA, thus not much time will be given to discussion here. In all cases, the single threaded mode will have worse performance in terms of time needed to find a solution. For example, here are the results for this mode:

Population Size	**Avg. Execution Time (s)**	**Avg. Fitness**
5	1.0237	0.7811
10	1.8373	0.8152
25	4.4750	0.7981
50	7.7093	0.8141
100	13.6810	0.8102
250	28.4992	0.8350

Table 1. Single Threaded Performance Results

What is interesting to note is the suboptimal fitness values for the best performers out of each trial. This will be discussed further in the multithreaded performance section.

D. GA Multi Threaded Performance

The multi threaded mode had a surprising increase in performance. Due to the capability of the algorithm to be massively parallelized, significant speed ups were expected. The best case observed occurred when the the population size was 25. This yielded a performance gain of ~8x. Other population sizes were in the range of 7-8x. The full testing results are shown in Table 2.

Population Size	**Avg. Execution Time (s)**	**Avg. Fitness**
5	0.1233	0.9377
10	0.2356	0.9227
25	0.5372	1.0000
50	1.0427	0.9978
100	1.9117	1.0000
250	3.7852	1.0000

Table 2. Multi Threaded Performance Results

While the performance speed up is excellent given the time available to develop the software, there is a curious difference between the single threaded and multithreaded versions. The multithreaded version routinely finds a perfect or near perfect solution while the single threaded version peaks at ~0.8. Ideally this should not be happening due to how the software has been constructed, however this could be due to how the random number generator engine produces new numbers.

The particular random number generator engine used for this project is the Mersenne-Twister (mt19937) supplied by the C++ stdlib header. In the the multi-threaded version, the engine is routinely recreated in the many threads that are spawned. In the single threaded version, a single instance of the engine is used at length before a new one is ever created. Perhaps the observed average lower performance of the single threaded version is due to some kind of issue in how truly random the engine is after being used at length. This is untested at the moment, but appears to be a likely candidate.

E. Example Solutions

Figures 4-6 below demonstrate three randomly selected solutions generated during a trial run of the multithreaded version of the software. All three plots meet the user given performance criteria from the benchmark solution.

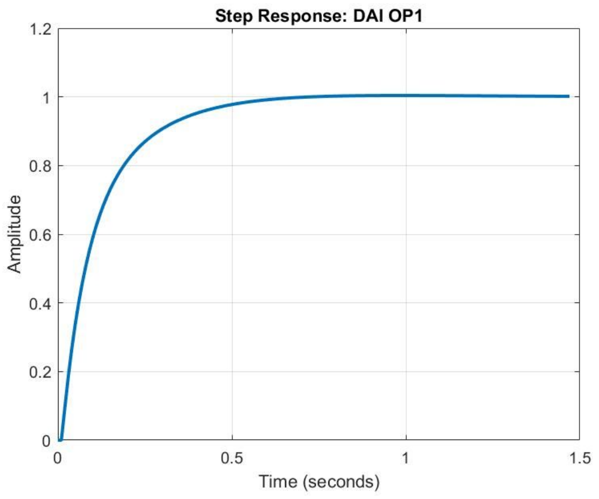

Figure 4. Optimizer 1 Example Solution

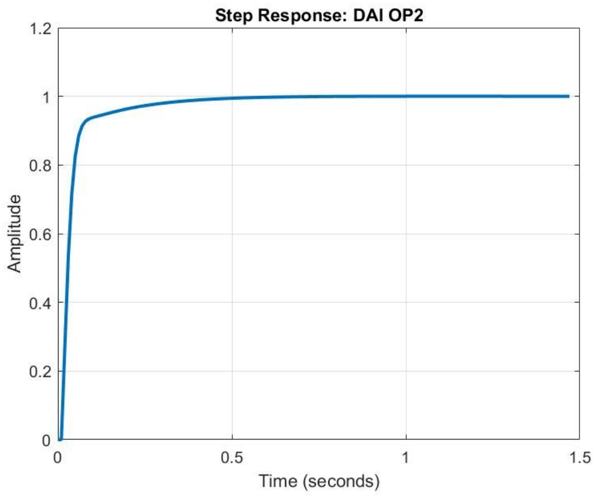

Figure 5. Optimizer 2 Example Solution

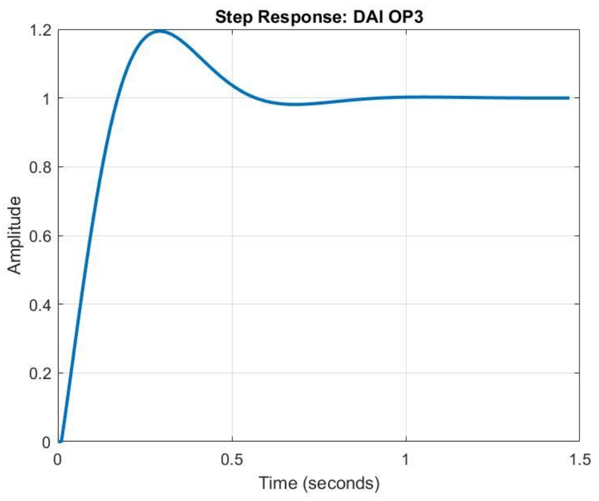

Figure 6. Optimizer 3 Example Solution

VI. Conclusions & Future Research

In conclusion, a framework for using the Genetic Algorithm to tune multirotor PID control loops within severely limited time constraints was presented. By utilizing a parallelized version of the GA that takes advantage of CPU multithreading, good solutions to a simple state space test problem were found while meeting user defined criteria and time limits. This demonstrates that the concept behind the full idea described in the Introduction is feasible with some work.

Because the presented version of software was created in the span of two months, there are still many opportunities for improvements. The current code needs optimizing for reduced run time and more widespread usage of threading. Several new features are planned to be added, including at a minimum:

1. Neural Network Models with TensorFlow
2. Simulation of Custom Control Inputs
3. Wireless Link between Drone and GA
4. Cloud Interface with Amazon Web Services

External to new features, the algorithm needs to be tested over a wider range of system dynamics to verify that good solutions are not found just within a specific subset of dynamics.

References

[1] A. Gupta, S. Goindi, G. Singh, H. Saini and R. Kumar, "Optimal Design of PID Controllers for Time Delay Systems Using Genetic Algorithm and Simulated Annealing," in *International Conference on Innovative Mechanisms for Industry Applications*, 2017.

[2] D. Kwok and F. Sheng, "Genetic Algorithm and Simulated Annealing for Optimal Robot Arm PID Control," in *The IEEE Conference on Evolutionary Computation*, Hong Kong, 1994.

[3] D. Meena and A. Devanshu, "Genetic Algorithm Tuned PID Controller for Process Control," in *International Conference on Inventive Systems and Control*, 2017.

[4] L. Jie, F. Ding and G. Bo, "Parameters Optimization of Aeroengine PID Controller Based on Genetic Algorithms," in *International Forum on Computer Science-Technology and Applications*, 2009.

[5] P. Kumar, J. Raheja and S. Narayan, "Design of PID Controllers Using Multiobjective Optimization with GA and Weighted Sum Objective Function Method," *International Journal of Technical Research,* vol. 2, no. 2, 2013.

[6] P. Wang and D. Kwok, "Auto-Tuning of Classical PID Controllers Using an Advanced Genetic Algorithm," in *International Conference on Industrial Electronics, Control, Instrumentation, and Automation*, 1992.

[7] Y. Mitsukura, T. Yamamoto and M. Kaneda, "A Genetic Tuning Algorithm of PID Parameters," in *International Conference on Systems, Man, and Cybernetics*, Orlando, 1997.