SHACT — An Evolutionary Method for Hardware and Controller Tuning

Nathaniel Schmitz
Mechatronics Engineering
University of Waterloo
Waterloo, ON, Canada
naschmit@edu.uwaterloo.ca

Tai Sun
Mechatronics Engineering
University of Waterloo
Waterloo, ON, Canada
t25sun@edu.uwaterloo.ca

Lalit Lal
Mechatronics Engineering
University of Waterloo
Waterloo, ON, Canada
l2lal@edu.uwaterloo.ca

***Abstract*—Vehicle hardware and controller software are developed independently for complex control problems, but there is often a better solution if both are developed simultaneously. Neuroevolution using genetic algorithms (GAs) and artificial neural networks (ANNs) are often used to solve difficult control problems after the vehicle hardware has already been designed. Simultaneous Hardware and Controller Tuning (SHACT) is a new method derived from evolutionary algorithms (EAs) to optimize a complex system's hardware and control system simultaneously. This paper shows how Simultaneous Hardware and Controller Tuning (SHACT) can be used to evolve an ANN controller's weights and system hardware together to obtain more optimal solutions than optimizing them independently. SHACT is applied to a problem that demonstrates the power of developing hardware and controller together, and compares the results with existing approaches for controller design.**

***Index Terms*—genetic algorithm, neuroevolution, island-model, optimization, controller**

I. Introduction

This project focuses on the development and comparison of an unique evolutionary algorithm against other vehicle and controller tuning algorithms. Where other algorithms take a already completed physical design and optimizes the control algorithm around it, this approach tunes both hardware and software simultaneously. The hardware represents the physical properties such as center of mass and positioning of vehicle hardware and software being the control algorithm for the hardware. The problem with the current approach is that with the physical design already complete, there is now a limit on the controller design due to the physical constraints. Despite there being robust and complex tuning algorithms to control the vehicle with enough accuracy, this can be further improved. The leap forward is to develop a technique to design both hardware and software together to reach a tuned vehicle and controller combination that is not constrained by one or the other.

II. Related Works

There are several existing approaches that focus on the non-linear optimization of hardware and controller development. There are several approaches for hardware tuning, such as genetic algorithms and finite element modeling. There are also several approaches for controller tuning, such as neuroevolution or designing discrete controllers. Typically, the hardware and controller are independently developed and optimized using genetic algorithms.

A. Hardware Evolution

The first area of work involves tuning the hardware parameters using genetic algorithms; which is known as hardware evolution. There are various hardware parameters that can be tuned and designed using genetic algorithms, such as wire antennas [1], composite material structures [2], and Systems on Programmable Chips [3]. In these works, evolution was used as a strategy to speed up the development and diversity of solutions. It is interesting to note that typically, the search space for hardware evolution is smaller than the controller evolution, and as such, hardware solutions converge faster than controller solutions, which was also noted and addressed in the novel approach mentioned below.

B. Controller Evolution

The controller evolution is of more interest, since it directly impacts the ability of the vehicle to meet objectives such as target set-points and paths using actuation. There are two forms of controller development and evolution that have been explored in development of a novel approach: using genetic algorithms to either tune traditional control parameters such as PID controllers, or using genetic algorithms to train neural network parameters—weights and biases—which is known as neuroevolution.

The first use of genetic algorithms in controller design is in the area of traditional controllers. There are various traditional controllers used to date, such as Proportional (P) control, Proportional-derivative (PD) control and Proportional-integral-derivative (PID) control. Although the search space is smaller than most genetic algorithms for traditional controllers (P, PD, PID), which have at most three control elements in the chromosomes, the search space is still very complex. Using evolutionary algorithms to train PID controllers is an adaptive approach on classical control techniques, as seen in [4]. Simpler control mechanisms, such as a bang-bang controllers can also be optimized and fine-tuned using GAs, as seen in [5]. Recent, optimal control methods such as a Model Predictive Controller (MPC) is used to look ahead to determine the optimal next step based on a specific look ahead

distance. A genetic algorithm can be used to tune a MPC in order to optimize the look-ahead distance and other involved parameters, as seen in [6].

Moving forward in the controller evolution, recent developments use a neural networks as the controller of the vehicle, where there is a set of inputs, a hidden layer, and a set of outputs. More commonly known as neuroevolution, this allows the controller to be optimized for the specific type of of maneuvers required by the hardware, while simultaneously honoring the hardware limitations. There are various techniques that utilize genetic algorithms to optimize neural networks, specifically the weights and biases associated with them. Such techniques include Analog Genetic Encoding (AGE), Enforced SubPopulations (ESP), Simulated Annealing (SA), Symbiotic, Adaptive Neuro-Evolution (SANE), Neuroevolution of Augmenting Topologies (NEAT), seen in [7], [8], [9], [10], [11] respectively. All of these methods focus on training neural networks using genetic algorithms to optimize for a specific problem that has hardware already implemented or separately developed. The following approach, SHACT, considers the simultaneous optimization of hardware and controller (neural network) of a rocket, designed to meet a target point, using a genetic algorithm [12]. The idea involved with simultaneous development is to allow for multiple complementary solutions to develop, allowing the designer to choose which is best depending on a set of constraints. Of course, since design of chromosomes will involve multiple parameters (hardware and software tuning parameters), the search space will be larger and require more time to solve. This will be addressed in later sections as well.

Three approaches will be used in optimizing the uniquely encoded chromosomes with combined hardware and software parameters. Specifically, a Standard Genetic Algorithm (SGA) [13] will be tested, as well as a variant of the SGA known as the Island Genetic Algorithm (I-GA) [14], [15], [16], [17]. Finally a novel algorithm will be tested with this unique idea of encoding both hardware and software parameters into a single chromosome, labelled as SHACT—Simultaneous Hardware and Controller Tuning.

The SGA approach is fairly simple, where the overall performance of the rocket is determined with a single fitness function that evaluates each chromosome and its parameters. This is seen in many applications in literature, as GAs are problem independent. A parallel and variant approach of SGAs is Multi-Island SGAs, or I-GA, where diversity is encouraged since many sub-populations simultaneously develop in unique search spaces, periodically allowing for a top percentage of chromosomes in each population to migrate to every other sub-population. The migration is dictated using a migration policy, which includes the migration frequency (number of generations between migrations), and migration rate (percent of most fit chromosomes that migrate to every other sub-population). This diversity promotes a thorough search of the solution space while preventing the solution from getting stuck in a local extrema. There are several other genetic algorithm variants known as Genitor and CHC (Cross-generational elitist selection, Heterogeneous recombination, and Cataclysmic mutation) [18], [19].

III. SHACT

SHACT is an evolutionary algorithm that prioritizes evolving a general controller when encoding hardware and controller within a single chromosome. We call the proposed architecture "SHACT" because it stands for simultaneous hardware and controller tuning. Much like a traditional evolutionary algorithm, it is an unsupervised, generic population-based meta-heuristic learning technique [20]. Evolutionary algorithms are loosely based off biological evolution, and borrows techniques such as reproduction, mutation, recombination, and selection to create candidate solutions to the optimization problem. Every generation, a new population is created using the biological operators, $G_{1,2,...,n}(x)$, and the goal is to maximize the fitness function, which is a measure of the effectiveness of the solution. A generic genetic algorithm is shown in Algorithm 1, and the Island GA is shown in Algorithm 2.

Algorithm 1 Simple Genetic Algorithm

Input: Stopping condition, S
Output: Chromosome with the highest fitness, C

procedure RUNGA(S)
  $L_C \leftarrow \text{InitPopulation}()$
  $F_C \leftarrow F(x)$ for x in L_C ▷ Calculate fitness
  while $S(F_C) \neq \text{False}$ **do**
    while $\text{len}(L_C) < \text{PopulationSize}$ **do**
      $P_1,\ P_2 \leftarrow \text{SelectParents}(F_C,\ L_C)$
      $C_1,\ C_2 \leftarrow \text{Crossover}(P_1,\ P_2)$
      $C_1 \leftarrow \text{Mutation}(C_1)$
      $C_2 \leftarrow \text{Mutation}(C_2)$
      $L_C \leftarrow L_C + C_1 + C_2$
    $F_C \leftarrow F(x)$ for x in L_C
  return $\text{sorted}(L_C)[0]$

The main idea behind SHACT is to encode both the controller parameters, $Q = [q_1, ..., q_n]$, and the hardware parameters, $H = [h_1, ..., h_m]$, in a single chromosome, $C = [Q, H] = [q_1, ..., q_n, h_1, ..., h_m]$. This is possible with a traditional genetic algorithm [21], but the novel technique is being able to differentiate the fitness contribution from Q, $F(Q)$, and the fitness contribution from H, $F(H)$, individually and intelligently breeding chromosomes based on $F(Q)$ and $F(H)$.

Every set number of generations, the SHACT algorithm will run. It starts by identifying the $F(Q)$ and $F(H)$ contributions to identify the most generalized chromosomes. The best hardware and controller combinations are rank-selected and copied into the next generation to preserve the optimal solutions. Finally, several of the optimal controllers are used with randomized hardware to fill the remaining chromosomes in the population and encourage hardware variation.

Algorithm 2 Island Genetic Algorithm

Input: Stopping condition, S
Output: Chromosome with the highest fitness, C

procedure RUNIGA(S)
  $L_C \leftarrow$ InitIslands()
  $F_C \leftarrow F(x)$ for x in L_C ▷ Calculate fitness
  while $S(F_C) \neq$ False **do**
    for i in NumIslands **do**
      while len$(L_C) <$ PopulationSize **do**
        $P_1, P_2 \leftarrow$ SelectParents(F_C, L_C)
        $C_1, C_2 \leftarrow$ Crossover(P_1, P_2)
        $C_1 \leftarrow$ Mutation(C_1)
        $C_2 \leftarrow$ Mutation(C_2)
        $L_C \leftarrow L_C + C_1 + C_2$
      Migrate(i) ▷ Migrate i to all other islands
    $F_C \leftarrow F(x)$ for x in L_C
  return sorted$(L_C)[0]$

A. Identifying $F(Q)$ and $F(H)$ Contributions

Identifying $F(Q)$ and $F(H)$ is crucial to tuning hardware and controllers. Evolutionary algorithms tend to get stuck in local minima [22], and identifying the hardware and controller fitness individually can tailor evolution for hardware and controllers separately. For example, suppose a series of random mutations are applied to each C. There is a possibility that a fitness-degrading genetic operator occurred on H such that $F(H) > F(G_i(H))$, while an independent genetic operator on the same C increased $F(Q)$ such that $F(Q) < F(G_j(Q))$. In other words, it improves quality of the hardware while degrading the controller. Typically GAs select chromosomes based on overall fitness, and are not capable of identifying which part of a chromosome has become more fit—in this instance, the quality of the hardware improved and the quality of the controller degraded.

The idea is to breed controllers that are more general—work with more hardware variations, and breed hardware that is more general—work with more controllers. To identify the individual fitness contributions from the hardware in the chromosomes, replace their controllers with the top N controllers and accumulate the total fitness for each hardware variation. Hardware variations with higher fitness after using different controllers are more generalized and should have a higher probability of being selected for the next generation. To identify the individual fitness contributions from the controllers in the chromosomes, replace their hardware variations with the top N hardware variations and accumulate the total fitness for each controller. Controllers with higher fitness after using different hardware variations are more generalized and should have a higher probability of being copied into the next generation. After the hardware and controller fitness has been accumulated, perform rank-weighted selection from the more generalized hardware and controllers, combine them into chromosomes, and copy them into the next pool of chromosomes. This way, higher generalization increases the chance of being copied into the next population, but does not ensure that the optimal solution will persist reducing the likelihood of premature convergence. This is shown in Algorithm 3.

To set an upper bound on N, only test the top $N = \lfloor\sqrt{P_n}\rfloor$ controllers and hardware variations, where P_n is the population size. If N is set appropriately, then the number of combinations is equal to P_n and the identification step can be run as another generation in the simulation. Instead of a true measure of generalizability, this approximates the generalizability using the top N chromosomes. Further studies might be important to determine if a stochastic selection of chromosomes would produce more generalized hardware variations and controllers.

Algorithm 3 Identifying F(Q) and F(H)

Input: Population sorted by fitness, L_C
Output: Chromosomes for the next population, L_{Q+H}

procedure INTELLIGENTBREED(L_C)
  $L_N, F_H, F_Q \leftarrow [], [], []$
  for i in $\lfloor\sqrt{P_n}\rfloor$ **do**
    for j in $\lfloor\sqrt{P_n}\rfloor$ **do**
      $L_N \leftarrow L_N +$ Chromo$(L_C[i].H, L_C[j].Q)$
  $F_N \leftarrow F(x)$ for x in L_N ▷ Calculate fitness
  for i in $\lfloor\sqrt{P_n}\rfloor$ **do**
    for j in $\lfloor\sqrt{P_n}\rfloor$ **do**
      $F_H[i] \leftarrow F_H[i] + F_N[i \times \lfloor\sqrt{P_n}\rfloor + j]$
      $F_Q[i] \leftarrow F_Q[i] + F_N[j \times \lfloor\sqrt{P_n}\rfloor + i]$
  return RankSelection(F_H, F_C)

B. Preventing Early Hardware Convergence

Several experiments using the SHACT algorithm showed that the hardware tends to converge before the controller, even if there are more optimal hardware configurations. This is likely because the controllers have more parameters and are more complex. To help mitigate this, the last M chromosomes in each population are created by generating entirely new hardware for M controllers selected using rank-weighted controllers. In other words, the optimal controllers are used with randomized hardware alleles to encourage additional mutation in hardware.

IV. Experiments and Results

A. Rocket Environment

The simulation of the rocket was done in a custom 2D environment created in Python3 with the use of PyBox2D physics engine as is shown in Fig. 1. This environment spawns rockets with a certain specification on a platform. All the rockets share a common objective: reach the stationary or moving target while remaining upright. The fitness function optimizes three criteria: minimize the distance to the target, minimize the angular velocity of the rocket, and minimize the angle of the rocket so that it stays upright as shown in (1), (2), and (3). P_R is the rocket position, P_T is the target position,

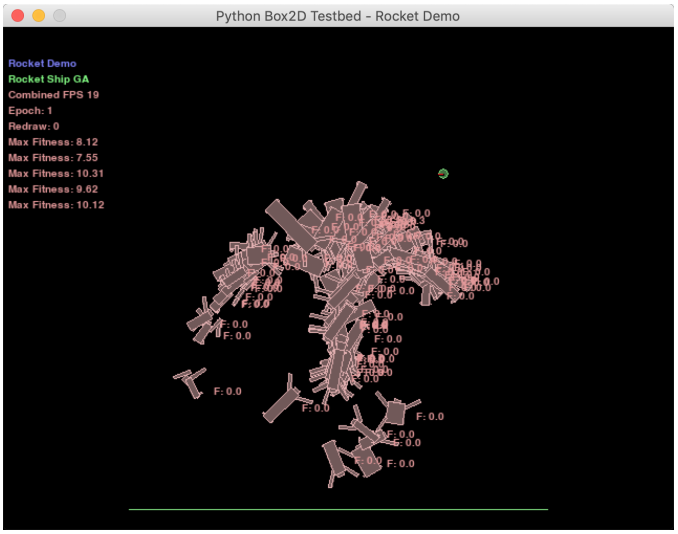

Fig. 1. Rocket environment written using Python3 and PyBox2D.

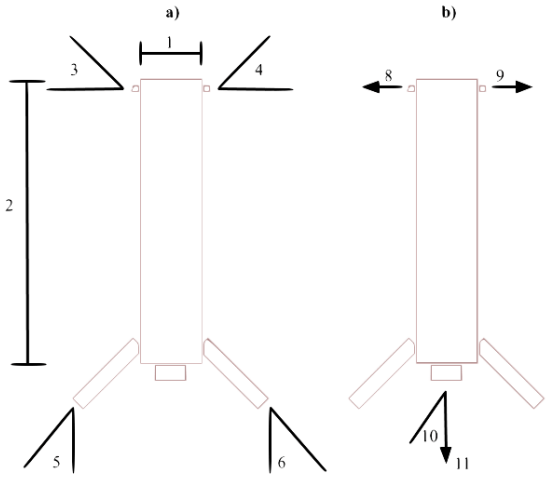

Fig. 2. Tunable characteristics of both Rocket and Controller.

ω_R is the angular velocity of the rocket, and θ_R is the angle of the rocket. The rocket fitness is updated at every time step, t, as shown in (4).

$$F_\delta = \frac{1}{1 + \exp\left(-3.5 + 0.6 \left\| P_R - P_T \right\|\right)} \tag{1}$$

$$F_\omega = -\left|0.1\omega_R\right| \tag{2}$$

$$F_\theta = -\left|0.5\theta_R\right| \tag{3}$$

$$F_{t+1}\left(C\right) = F_t\left(C\right) + \max\left(0,\, F_\delta + F_\omega + F_\theta\right) \tag{4}$$

In order to validate the SHACT algorithm, both the hardware as well as the controller parameters must be tunable to allow them to evolve. The hardware parameters include rocket height, rocket width, left leg angle, and right leg angle to adjust center of mass of the rocket, as well as the angles of the bottom, left, and right thrusters at the top of the rocket for thrust control as shown in Fig. 2.

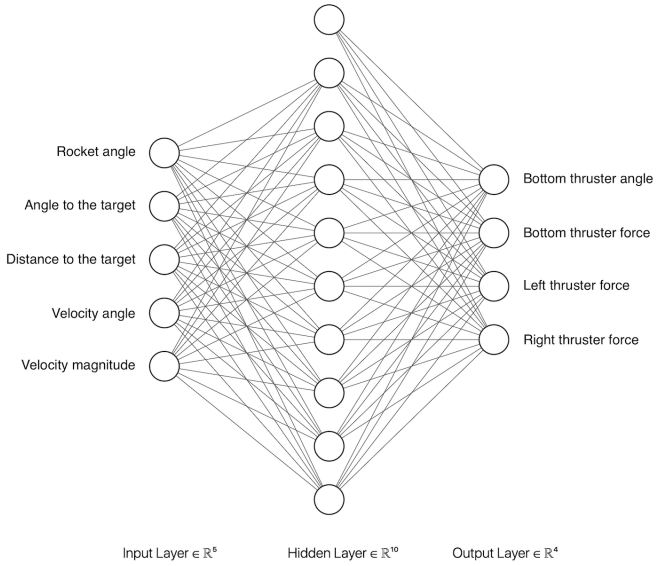

Fig. 3. ANN controller structure for evolutionary approaches.

All of the controllers have the following inputs: angle of the rocket, angle to the target in the inertial frame, distance to the target, angle of the velocity in the inertial frame, and the magnitude of the velocity. The controllers have four possible output values: bottom thrust, bottom thruster angle, left thrust, and right thrust. In the case of the evolutionary algorithms, the controller is an Artificial Neural Network (ANN) with the structure shown in Fig. 3.

The simulation framework allows the simulation to be played back in either real-time or sped up to run the simulation faster. Using a laptop processor, about 10 generations can be run in 1 minute using a population size of 100 and a simulation duration of 15 seconds in the physics engine.

The various controller design algorithms will be tested against two scenarios in the simulation environment. Scenario A is when the target is placed at point $(x,\, y) = (10,\, 30)$ and the rocket has to reach the point and hover in place. This demonstrates the controller's ability to hold the rocket in a constant and upright position. Scenario B demonstrates the controller's ability to follow a moving reference position. The target follows the parametric equations in (5), given the running time of the current generation, t. In both scenarios, the rocket has to stay as close to the target as possible while remaining upright.

$$(x,\, y) = (t \sin 0.5t,\, 20.0) \tag{5}$$

B. Approaches Tested

To validate the performance of SHACT, rocket performance was tested against traditional controller methods used in research and in the industry. The same fitness function was applied to each method which analyzes the distance to target, to minimize the tracking error; the angular velocity of the

rocket, which minimizes the jerk or "comfort parameter" [23]; and the angle of the rocket, which optimally positions the rocket to counter gravity.

The controllers P, PD, and PID were implemented and tuned using an evolutionary algorithm and its results compared with other approaches including SHACT. The P controller only utilizes the cross track error alongside a gain value for control. This method is prone to severe overshoot and oscillation around the desired path which, in this case, is directly upwards. The PD controller includes the Derivative term which "dampens" the amount of correction the rockets actually needs. Thus preventing the behaviour observed with only a P controller. The addition of an I term for a PID controller prevents the build up of steady state error that can accumulate throughout the duration of a generation. However, the I term is not effective when the thrust actuator only works in one direction [24]. The cross track error is not directly related to how the fitness function is calculated but is correlated. Each controller has an independent controller gain value which usually is calculated by modeling the system and determined theoretically then tested and tuned. Instead, the gain values were tuned with the use of an EA because this is a relatively simple model with few parameters.

C. Parameter Setting

The evolution parameters for the BB and P controllers are as follows: population size of 100 chromosomes, 5 mutations per 100 chromosomes on the first generation and decreases to 1 mutation per 1000 chromosomes on the 100^{th} generation, crossover rate of 80 crossovers out of 100 chromosomes, and the 2 best chromosomes are kept per generation. The parameters for the PD and PID controllers are the same except they had a starting mutation rate of 3 mutations per 100 chromosomes because there are more parameters involved.

As mentioned above, the I-GA approach is a variant of the standard GA, with an additional mutation policy that allows sub-populations to migrate between one another. The main parameters, in addition to the standard GA, include a number of sub-populations, the number of chromosomes per sub-population, a migration frequency that dictates the number of generations between migration from one sub-population to each other sub-population, and a migrate rate which dictates how many chromosomes of the sub-population migrate. The evolution parameters for the I-GA are as follows: population size of 1000 chromosomes—10 islands with 100 chromosomes each, 1 mutation per 100 chromosomes, and a crossover rate of 80 crossovers out of 100 chromosomes. Finally, each sub-population migrates every 50 generations (migration frequency), and the top 2 chromosomes are migrated (migration rate of 2 percent).

D. Results

1) Performance Metric: The results from the two scenarios—stationary rocket and moving rocket—are shown in Table I. The SHACT algorithm received slightly higher final fitness value than the PD and PID controllers. The way the rocket managed to outperform the PD and PID controllers was by rotating its thrusters upwards to counter the initial bottom thrust to prevent overshoot. The rockets were able to reach the target position faster at the beginning of each generation. A comparison of the standard rocket as used in BB, P, PD, and PID compared with the GA, I-GA, and SHACT evolved rockets is shown in Fig. 4. Another hardware modification the SHACT algorithm made was reducing the width of the rocket and increasing its height. It is difficult to speculate why, but this makes the rocket lighter and increases the torque moment from the side thrusters. The various approaches are compared using a performance metric, P. The performance metric is average maximum fitness for 10 runs of that approach as shown in (6), where F_n^{Max} is the maximum fitness of the n^{th} run. The approach with the maximum performance metric was SHACT, closely followed by the PD controller.

$$P = \frac{\sum_{n=1}^{10} F_n^{\text{Max}}}{10} \tag{6}$$

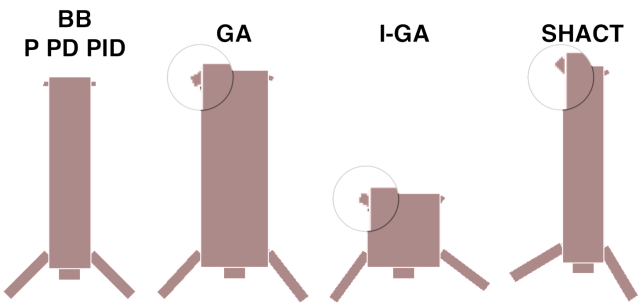

Fig. 4. Final rockets evolved with each technique. The I-GA and SHACT algorithms evolved rockets that have a negative-y thrust direction. Magnifiers help show the angle of the left and right thrusters.

TABLE I
RESULTS OF TESTING THE SHACT ALGORITHM AGAINST THE OTHER APPROACH ON TWO DIFFERENT SCENARIOS WITH 10 RUNS PER APPROACH.

Scenario A — Target Stationary Tests a rocket's ability to hover in place				
Category	Method	Max Fitness	Generations	P Score
Controller-based	BB	42.04	4	31
	P	188.21	12	137
	PD	692.56	9	682
	PID	691.26	26	683
Evolutionary	GA	572.96	224	547
	IGA	436.34	260	404
	SHACT	697.37	217	692
Scenario B — Target Moving Tests a rocket's ability to follow a trajectory				
Category	Method	Max Fitness	Generations	P Score
Controller-based	BB	211.98	2	117
	P	469.45	8	465
	PD	716.51	17	692
	PID	716.45	33	696
Evolutionary	GA	547.47	901	418
	IGA	418.13	742	343
	SHACT	718.54	572	704

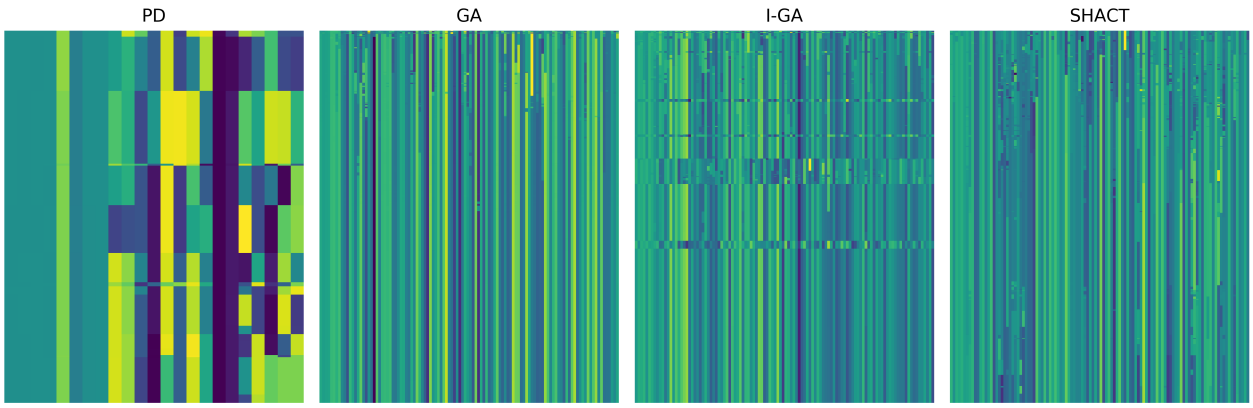

Fig. 5. The best chromosome from each generation for each algorithm type. Along the x-axis of each image is the allele, and down the y-axis of each image is the generation number. The first 8 columns represent the hardware alleles and the remaining columns represent the controller alleles.

2) Evolution Graph: To visualize how the best chromosome evolves over time, Fig. 5 shows every individual allele in the best chromosome for each generation. The leftmost image represents the alleles for the PD controller. The hardware alleles are fixed and cannot mutate because the goal of this method is to evolve the controller only. Moving down the rows of the image corresponds to an increase in generation number. Moving across the columns of the image corresponds to the different alleles in the chromosome. Throughout the generations for the PD controller, the gain parameters underwent significant changes but finally converged and reached steady-state.

The GA evolution graph shows that once the algorithm converged, it remained in steady-state without changing any of the alleles. It reached a local maxima in the solution space because it did not have a fitness as high as the SHACT, PD, or PID algorithms. Another interesting thing to note in the evolution graph is that the I-GA algorithm had the best chromosome come from different islands throughout the simulation. It was too computationally intensive to run the simulation to convergence, so the fitness of the I-GA approach is lower than the other approaches. The horizontal bands in the I-GA evolution graph represent when a new chromosome, from a different island, had the highest fitness.

The SHACT algorithm reaches steady-state, but because the algorithm is being injected with random hardware, the hardware does not converge as quickly as with the GA or I-GA algorithms and the solution did not reach a local maxima. Both the controller and hardware parameters continue to mutate until the simulation was stopped which shows the generalization abilities of the algorithm.

3) Trajectory Visualization: The trajectory of the best chromosome after the evolution converged is shown in Fig. 6. The rocket evolved using the SHACT algorithm reaches the reference trajectory earlier than the other approaches because the left and right thrusters evolved to point upwards, allowing the initial thrust to be countered and prevent overshoot. The x-axis and y-axis in the 3d plot are broken out into individual x-axis and y-axis versus generation running time to visualize

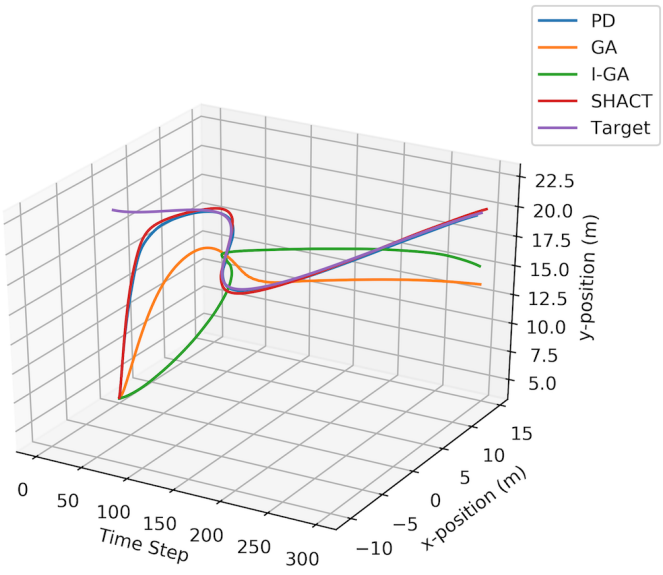

Fig. 6. 3D trajectory of the rockets and the reference position.

overshoot properties of the tracking and is shown in Fig. 7.

V. Summary and Conclusions

This paper set out to investigate the benefits of integrating hardware and controller parameters in a simultaneous optimization problem using genetic algorithms. A unique application of the design of a rocket (hardware and controller) is used to test this approach. The hardware parameters of the rocket involve the dimensions of the rocket as well as the booster angles, while the controller parameters are compromised of the weights and biases of an artificial neural network (5 input neurons, 10 hidden neurons, and 4 output neurons). The method of tuning the ANN using a GA is known as neuroevolution. Various approaches were tested. The first 3 approaches were use to determine a baseline maximum fitness. These approaches involve fixing the hardware and simply tuning the controller as P, PD, and PID controllers using a

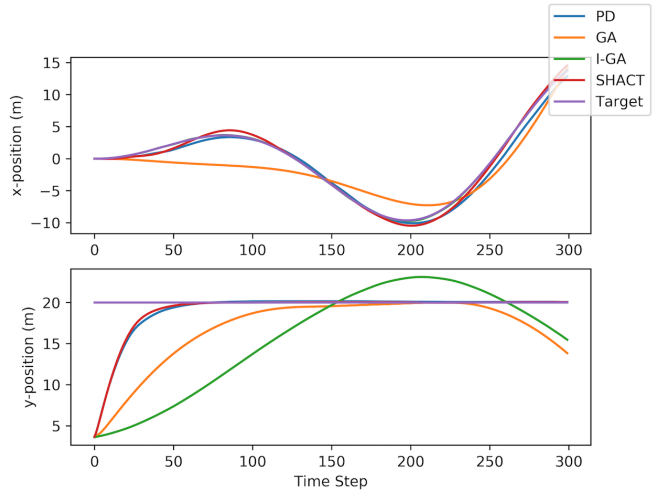

Fig. 7. 2D trajectory of the rockets split up into x-position and y-position.

GA which drastically simplifies the search space. The optimal results of these approaches were in using a PD controller, which converged in 9 generations for a stationary target, and 17 generations for a moving target. The next two approaches involve using a GA to tune the combined hardware and control chromosome; the first type is using the standard GA and the second is using an island variant of the GA, known as I-GA. Of these, the standard GA approach was optimal, yielding a higher fitness for a stationary target and moving target, converging at 224 and 901 generations respectively. Finally, the novel approach, a custom variant of a GA, named SHACT was tested. This approach was the optimal of all sets of approaches, yielding the highest fitness levels for both stationary and moving targets, converging at 217 and 572 generations respectively. In conclusion, while the SHACT algorithm outperformed all other approaches, it did so at the compromise of time (many more generations to complete versus PD control). In some applications this may or may not be an issue.

Moving forward, a few improvements can be made. Specifically, arbitrary mutation, crossover rates were applied. As learned from class and mentioned in literature, choose these parameters explicitly for the application may yield more optimal results. Secondly, various types of GAs are known to perform better than the standard GA algorithm, specifically due to the style of breeding new populations and mutation. The integration and hybridization of these two non-traditional GAs into the SHACT algorithm might allow for a more optimal solution, time permitting.

References

[1] Linden, Derek. (2002). Antenna Design Using Genetic Algorithm. 1133-1140.

[2] Averill, Ron and Goodman, Erik and Lin, Shyh-Chang and Ding, Ying. (1995). Design Using Genetic Algorithms - Some Results for Laminated Composite Structures.

[3] Swarnalatha, A and Shanthi, A.P. (2014). Complete hardware evolution based SoPC for evolvable hardware. Applied Soft Computing. 18. 10.1016/j.asoc.2013.12.014.

[4] F. M. Amaral, Jose and Tanscheit, Ricardo and Pacheco, Marco. (2001). Tuning PID Controllers through Genetic Algorithms.

[5] AlHamaydeh, Mohammad and Jaradat, Mohammad and Serry, Mohamed and Sawaqed, Laith and Hatamleh, Khaled. (2017). Structural control of MR-dampers with genetic algorithm-optimized Quasi-Bang-Bang controller. 10.1109/ICMSAO.2017.7934887.

[6] Mohammadi, Arash and Asadi, Houshyar and Mohamed, Shady and Nelson, Kyle and Nahavandi, Saeid. (2017). Optimizing Model Predictive Control Horizons using Genetic Algorithm for Motion Cueing Algorithm. Expert Systems with Applications. 92. 10.1016/j.eswa.2017.09.004.

[7] Drr, Peter and Mattiussi, Claudio and Floreano, Dario. (2006). Neuroevolution with Analog Genetic Encoding. 4193. 10.1007/11844297_68.

[8] Gomez, Faustino and Miikkulainen, Risto. (2003). Active Guidance for a Finless Rocket Using Neuroevolution. Lecture Notes in Computer Science. 2724. 10.1007/3-540-45110-2_105.

[9] Kirkpatrick, Scott and D. Jr. Gelatt, C and P. Jr. Vecchi, M. (1983). Optimization by Simulated Annealing. Science (New York, N.Y.). 220. 671-80. 10.1126/science.220.4598.671.

[10] Moriarty, David and Miikkulainen, Risto. (1995). Efficient Reinforcement Learning Through Symbiotic Evolution. Machine Learning. 22. 10.1023/A:1018004120707.

[11] Stanley, Kenneth and Miikkulainen, Risto. (2002). Evolving Neural Networks through Augmenting Topologies. Evolutionary computation. 10. 99-127. 10.1162/106365602320169811.

[12] J. Montana, David and Davis, Lawrence. (1989). Training Feedforward Neural Networks Using Genetic Algorithms. 762-767.

[13] Whitley, Darrell. (1998). A Genetic Algorithm Tutorial. Statistics and Computing. 4. 10.1007/BF00175354.

[14] Gong, Yiyuan and Fukunaga, Alex. (2011). Distributed island-model genetic algorithms using heterogeneous parameter settings. 2011 IEEE Congress of Evolutionary Computation, CEC 2011. 820 - 827. 10.1109/CEC.2011.5949703.

[15] Meng, Qinxue and Wu, Jia and Ellis, John and Kennedy, Paul. (2017). Dynamic Island Model based on Spectral Clustering in Genetic Algorithm. 1724-1731. 10.1109/IJCNN.2017.7966059.

[16] Scott Gordon, V and Whitley, Darrell. (1998). Serial and Parallel Genetic Algorithms as Function Optimizers. Serial and Parallel Genetic Algorithms As Function Optimizers.

[17] Whitley, Darrell and Rana, Soraya and Heckendorn, Robert. (2006). Island model genetic algorithms and linearly separable problems. 10.1007/BFb0027170.

[18] Maruyama, T and Kita, E. (2006). Improvement of generation change on SSE algorithm. 451-458. 10.2495/DATA060451.

[19] Rathee, Seema and Ratnoo, Saroj and Ahuja, Jyoti. (2019). Instance Selection Using Multi-objective CHC Evolutionary Algorithm: Proceedings of Third International Conference on ICTCS 2017. 10.1007/978-981-13-0586-3_48.

[20] Beheshti, Zahra and Shamsuddin, Siti Mariyam. (2013). A review of population-based meta-heuristic algorithm. International Journal of Advances in Soft Computing and Its Applications. 5. 1-35.

[21] Sims, Karl. (1994). Evolving virtual creatures. Proceedings of the 21st annual conference on Computer graphics and interactive techniques. 8. 15-22. 10.1145/192161.192167.

[22] Etschberger, Stefan and Hilbert, Andreas. (2002). Multidimensional Scaling and Genetic Algorithms: A Solution Approach to Avoid Local Minima.

[23] Wang, Yuyang and Chardonnet, Jean-Remy and Merienne, Frederic. (2018). Speed Profile Optimization for Enhanced Passenger Comfort: An Optimal Control Approach. 723-728. 10.1109/ITSC.2018.8569420.

[24] Bacci di Capaci, Riccardo and Scali, Claudio. (2018). An augmented PID control structure to compensate for valve stiction. IFAC-PapersOnLine. 51. 799-804. 10.1016/j.ifacol.2018.06.181.

```
from data_persist import DataPersist

from numpy import random
from random import uniform


class Genetic():

    def __init__(self, newEpochCallback, chromosomeClass):
        super(Genetic, self).__init__()

        self.epoch = 0
        self.chromosomes = []
        self.oldChromosomes = []
        self.chromosomeClass = chromosomeClass
        self.intraEpisodeTime = 0.0
        self.newEpochCallback = newEpochCallback
        self.dataPersist = DataPersist()

        if chromosomeClass.isEvolutionary():
            self.numChromosomes = 100
            self.numKeepChromosomes = 2
        else:
            self.numChromosomes = 1
            self.numKeepChromosomes = 1

        self.firstEpoch()

    def updateGenetic(self):
        self.intraEpisodeTime += 1 / 20.0
        if self.intraEpisodeTime > 15.0:
            self.intraEpisodeTime = 0.0
            self.newEpoch()

    def crossover(self, parent1, parent2):
        child1 = self.chromosomeClass(parent1.values)
        child2 = self.chromosomeClass(parent2.values)
        if uniform(0, 1) < self.chromosomeClass.crossoverRate():
            values1 = parent1.values
            values2 = parent2.values
            for i in range(len(values1)):
                if uniform(0, 1) < self.chromosomeClass.indexCrossoverRate(i):
                    child1.values[i] = values2[i]
                    child2.values[i] = values1[i]
        return child1, child2

    def mutate(self, chromosome):
        mutated = self.chromosomeClass(chromosome.values)
        for i in range(len(chromosome.values)):
            if uniform(0, 1) < self.chromosomeClass.mutationRate(i):
                mutated.values[i] = self.chromosomeClass.randomAllele(i)
        return mutated

    def computeDistribution(self, chromosomes):
        totalFitness = 0.0
        for chromosome in chromosomes:
```

```
            totalFitness += chromosome.fitness
        if totalFitness == 0.0:
            totalFitness = 0.01

        distribution = []
        sumOfProbabilities = 0.0
        for chromosome in chromosomes:
            probability = chromosome.fitness / totalFitness
            distribution.append(sumOfProbabilities + probability)
            sumOfProbabilities += probability

        self.dataPersist.append(chromosomes, totalFitness)

        print('Epoch {:3d} Max {:.2f} Total {:.2f}'.format(self.epoch,
            chromosomes[0].fitness, totalFitness))
        return distribution

    def selectParentFromDistribution(self, distribution, chromosomes):
        selection = uniform(0, 1)
        for i, prob in enumerate(chromosomes[1:]):
            if selection < distribution[i]:
                return chromosomes[i - 1]
        return chromosomes[0]

    def firstEpoch(self):
        for i in range(self.numChromosomes):
            self.chromosomes.append(self.chromosomeClass())
        self.newEpochCallback(self.chromosomes)

    def newEpoch(self):
        self.epoch += 1

        self.oldChromosomes = sorted(self.chromosomes, key=lambda x: x.fitness,
            reverse=True)
        distribution = self.computeDistribution(self.oldChromosomes)

        self.chromosomes = []
        for i in range(self.numKeepChromosomes):
            self.chromosomes.append(self.chromosomeClass(
                self.oldChromosomes[i].values))

        while (len(self.chromosomes) < self.numChromosomes):
            parent1 = self.selectParentFromDistribution(distribution,
                self.oldChromosomes)
            parent2 = self.selectParentFromDistribution(distribution,
                self.oldChromosomes)
            child1, child2 = self.crossover(parent1, parent2)
            self.chromosomes.append(self.mutate(child1))
            self.chromosomes.append(self.mutate(child2))

        self.newEpochCallback(self.chromosomes)

    def savePlotAndData(self):
        self.dataPersist.savePlotAndData(str(self.chromosomeClass.__name__),
            self.oldChromosomes)
```