2012

A Hardware Compact Genetic Algorithm for Hover Improvement in an Insect-Scale Flapping-Wing Micro Air Vehicle

Kathleen M. Timmerman
Wright State University

A Hardware Compact Genetic Algorithm for Hover Improvement in an Insect-Scale Flapping-Wing Micro Air Vehicle

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science

by

Kathleen M. Timmerman
B.S. Computer Science, Wright State University, 2010

2012
Wright State University

Wright State University
SCHOOL OF GRADUATE STUDIES

September 10, 2012

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Kathleen M. Timmerman ENTITLED A Hardware Compact Genetic Algorithm for Hover Improvement in an Insect-Scale Flapping-Wing Micro Air Vehicle BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Computer Science.

John Gallagher
Thesis Director

Mateen Rizki, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on
Final Examination

John Gallagher, Ph.D.

Mateen Rizki, Ph.D.

Michael Raymer, Ph.D.

Andrew Hsu, Ph.D.
Dean, School of Graduate Studies

ABSTRACT

Timmerman, Kathleen. M.S. Comp Sci., Department of Computer Science and Engineering, Wright State University, 2012. *A Hardware Compact Genetic Algorithm for Hover Improvement in an Insect-Scale Flapping-Wing Micro Air Vehicle.*

Wing and airframe damage to insect scale micro air vehicles potentially cause significant losses in pose and position control precision. Although one can imagine many possible means of adapting the flight controllers to restore precise pose and position control, severe limits on computational resources available on-board an insect sized vehicle render many of them impractical. Additionally, limits on sensory capability degrade any such vehicle's ability to critique its own performance. Any adaptive solutions one would propose to recover flight trajectory precision, therefore, would require a resource light implementation, preferably without need for relatively expensive floating-point operations, along with the capability to assess control quality via relatively infrequent and possibly imprecise point estimates of vehicle pose and position.

This thesis will expand on previous work that employed an adaptive oscillator as a component of an altitude controller inside a simulated insect-scale flapping-wing micro air vehicle. In that work, it was demonstrated that an adaptive oscillator could learn novel wingbeat patterns unique to the capability of specific, possibly damaged, wings. These wingbeat patterns would restore the relationships between control outputs and wing motion; and thus; restore correct whole-vehicle action. In that earlier work, the core of the adaptive oscillator was an evolutionary algorithm (EA) that operated as a mutation driven stochastic hill climber. In this work, we explore the use of and the potential benefits of an EA variant that operates as a crossover driven schema/hyperplane sampler. For this work we selected the Compact Genetic Algorithm (cGA), as it possess an efficient hardware-level implementation and is clearly a crossover-driven hyperplane sampler. The thesis will present experimental results from which one may assess the relative performance of this style of genetic search as well as speculate upon its potential utility for more complex

flight control problems.

Contents

List of Figures

List of Tables

Acknowledgment

First, I would like to thank my advisor, Dr. John Gallagher, who has provided great support and guidance throughout my graduate career. I would also like to thank my friends and family for their patientence and understanding over the years.

Introduction

Micro Air Vehicles (MAV) possess a multitude of potential uses including surveillance in crowded areas, navigation of wreckage of a natural disaster, and traversing terrain not passable by ground vehicles. The recent first flight of an insect-scale Flapping-Wing Micro Air Vehicle (FW-MAV) in 2008 [17], has increased interest in the design and development of similarly scaled flapping-wing vehicles. However, the small size of these vehicles introduces a number of unique challenges.

Even slight errors in manufacturing will affect vehicle performance. Along with manufacturing faults, the vehicle may sustain damage while in flight. Even small amounts of damage to the wings and airframe can produce up to 1000% error in pose and position control precision [13]. Although there are many possible solutions to adapting the altitude controllers to restore pose and position control, the vehicle size severely limits the computational resources available on-board. Additionally, limits to sensory capability diminish the vehicle's ability to critique its own performance. These limitations render many traditional solutions impractical. Any proposed adaptive solution would need to possess a resource-light implementation, entail few or no floating-point operations, while determining control quality through infrequent and possibly imprecise point estimates of vehicle pose and position.

This thesis will expand upon previous work that employed an adaptive oscillator as a component of an altitude controller inside a simulated insect-scale FW-MAV [7] [14] [8]. In the previous work it was demonstrated that an adaptive oscillator could learn distinct wing

beat patterns unique to the capability of specific, possibly damaged wings. These wing beat patterns can restore the relationships between control outputs and the production of desired net forces and torques by the wings for hover control of the vehicle, correcting whole vehicle action. In previous work, the core of the adaptive oscillator was an Evolutionary Algorithm (EA) that operated as a mutation driven stochastic hill climber. In this thesis, the potential benefits of an EA variant that operates as a hyperplane sampler is examined.

The Compact Genetic Algorithm (cGA) [12] is an exclusively crossover-driven hyperplane sampler EA that possess an efficient hardware-level implementation [15]. For these reasons, this work considers cGA as a candidate for the learning core inside the existing adaptive oscillator component. The cGA represents the population as percentages rather than storing the exact population at any given point in time. The result is a significant reduction of memory space needed even in a fairly small genome [12]. In the case of a controller for a FW-MAV, wherespace efficiency is an issue, the cGA offers an interesting alternative to other hyperplane sampling algorithms.

This thesis will present experimental results from which one may assess the relative performance of a hyperplane sampling genetic search as well as speculate upon its potential utility for more complex flight control problems. This thesis is divided into five chapters. Chapter 2 provides the background material and brief overview of topics that should clarify terminology used throughout the remainder of the thesis. It also reviews current literature in the field. Chapter 3 discusess the methodology along with analysis techniques and adaptations to the cGA. Chapter 4 contains the experiment data, and chapter 5 contains the conclusion as well as comments on future work.

Background

This chapter summarizes previous work as well as defines the terminology used throughout the remainder of this thesis. It begins with background information on the type of algorithm that will be applied to the vehicle and then continues on to give a description of the vehicle on which the experiments were run. It concludes with a discussion on a previously tested algorithm and the issues with that algorithm.

2.1 Evolutionary Computing

Evolutionary Computing (EC) is a field of computational study inspired by natural selection [2]. EC methods maintain a population of candidate solutions and apply successive rounds of reproduction, variation, and selection of fittest candidates to move the average quality of solutions upward. Once many members of a population attain sufficiently high quality, it is presumed that one member of that population would suffice as a solution. EC encompasses many different algorithms, with the three main variants being genetic algorithms [10], evolution strategies [6], and evolutionary algorithms [3]. These algorithms can be envisioned as consisting of the following component parts: representation, evaluation function, parent selection, variation operations, survivor selection, initialization, and termination [2].

EC algorithms start with an initialization process to create an initial population of possible solutions. Then the algorithm will loop until the termination criteria is met. As shown

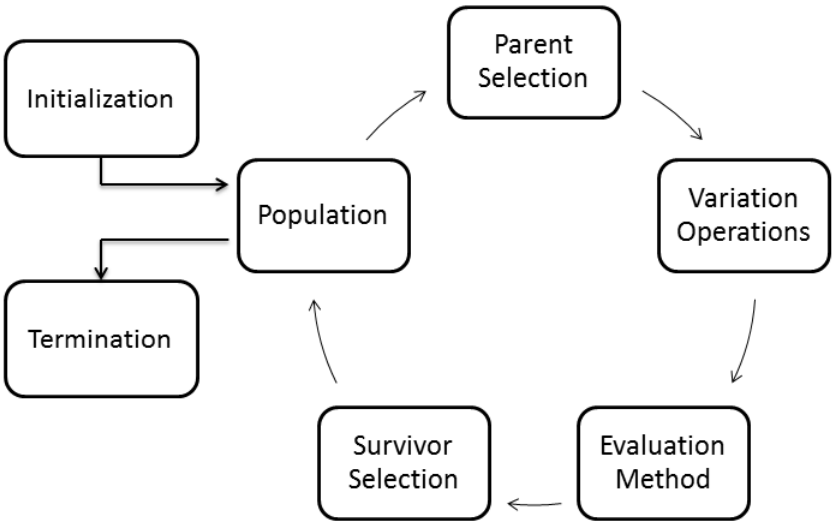

Figure 2.1: Basic Evolutionary Computing Algorithm [2]

in Figure 2.1, the loop begins with a population of possible solutions. Next parent selection is applied in order to determine which solutions may be used to create new solutions. This step is usually followed by the variation operations which create new possible solutions. The solutions are then evaluated and survivor selection is typically applied to maintain a constant population size. The termination criteria is then tested to see if the loop should terminate. Once the loop terminates, the best solution in the end population is used.

2.1.1 Representation

Each possible solution is called a candidate solution or individual. In its encoded form, which is the representation of the solution operated upon by the EA, it is called a genotype. Genotypes can be decoded into actual problem solutions, or phenotypes, via a mapping

function.

Each genotype is made up of loci. Generally, a loci represents one phenotypical component (or characteristic) of the candidate solution. Each locus is made up of different possibilities called alleles. All of the alleles concatenated together would represent the whole genotype. The alleles are encoded in various ways depending on the type of EA being used. Genetic Algorithms generally employ bit-strings to represent a genotype with identified sub-strings in each candidate encompassing alleles. Evolution Strategies and Evolutionary Programming generally employ arrays of real values to represent a genotype where specific array elements represent allele values directly. Note that in some cases, especially in Evolution Strategies, some alleles actually represent search algorithm parameters. This thesis will not be taking advantage of coding of search parameters on a genotype, but it is mentioned for completeness, and possible future consideration. Because representation has an impact on evolutionary search and correct choices for other components of an EA, proper solution coding is critical and is often determined on a case-by-case basis.

2.1.2 Evaluation Function

The evaluation function, also called the fitness function, is a method for testing how successful a solution is at solving the current problem. Each generation, cadidate individuals are evaluated via the fitness function, and a quality score, or fitness, is assigned to each. That score is the value being optimized by the entire algorithm. In this work, every genotype to be evaluated will be decoded into a phenotype. Control parameters for the adaptive oscillator will be extracted from the decoded phenotype and provided to the adaptive oscillator. Flight will be simulated for a set period of time using the desired settings and a quality score will be returned to the EA. This process is not dissimilar to processes used to evaluate electro-mechanical device performance in any number of automated optimization pursuits.

2.1.3 Parent Selection

Parent selection is a process that determines which individuals in the population can be included in the pool of possible candidates to undergo variation operations. This process differs from algorithm to algorithm. One common method, fitness-proportuonate selection, is to select individuals based on some probabilistic measure of an individual's fitness level compared to the fitness level of other individuals in the population. This method gives higher selection probabilities to individuals with better fitness scores. By the selection of more fit individuals, the overall population should increase in fitness as time progresses.

Another common method for parent selection is tournament selection. In tournament selection multiple individuals (usually two) are randomly selected from the population. The individuals then enter a tournament against each other where the winner is the individual with the best fitness score. That individual is then entered into the pool of candidates for variation operations. As in the first method, the overall population should increase in fitness as time progresses, since the more fit individuals are being added to the pool.

An algorithm could apply a universal parent selection process that selects every individual in the population to be in the pool of possible candidates to undergo variation operations. While this is a selection process in the sense that individuals are selected to enter the pool, it applies no selection pressure (preferential selection of fitter individuals). Therefore, this selection process would not tend to increase the overall population's fitness. When this method is used, the selection pressure needs to be applied elsewhere in the algorithm, typically by a survivor selection method, as will be defined later in the document.

When an individual within the pool is actually selected to undergo a variation operation, that individual gains the title of parent. The solution that is derived from the parent is referred to as the child or offspring. The term child could also refer to an individual that did not undergo a variation operation but was still selected to survive into the next population.

2.1.4 Variation Operations

Variation operations are used to create new individuals by either combining features of two or more candidate parents (recombination) or by creating randomly modified copies of candidate parents (mutation). In practice, there exists a huge variety of recombination and mutation operations. This chapter will focus only on canonical and representative operators in each category.

Recombination

Recombination is a variation operation that takes multiple parents, typically 2, and uses a combination of their genomic material to make a child. The child will presumably then exhibit a combination of the parents' features.

Discrete recombination techniques choose one or more points in a genome and, in the case of two-parent crossover, swap the information on one side of that break point (one-point crossover) or between the break points (multi-point crossover). Intermediary crossover will generally complete some arithmetic averaging of all the alleles in two or more parent genomes. In this work, we will be discussing discrete recombination methods exclusively.

As already implied, there are various ways of implementing discrete recombination. What is common to all of them is the idea of choosing division points and swapping genetic material from genome to genome in and among regions defined by these points. A key distinction in practice between EAs in the style of a genetic algorithm (GA) and EAs in the style of an Evolutionary Program (EP) or an Evolution Strategy (ES) is that GAs will divide genomes at any point of the string, even within loci, and EP/ES will choose division points at locus boundaries. Proponents of GA approaches often claim the ability to swap partial alleles as an advantage and have developed a "schema theory" to add some credibility to their claims. Although some aspects of schema theory are still controversial, it is still often worth considering if EAs that operate using only abilities entailed in schema theory

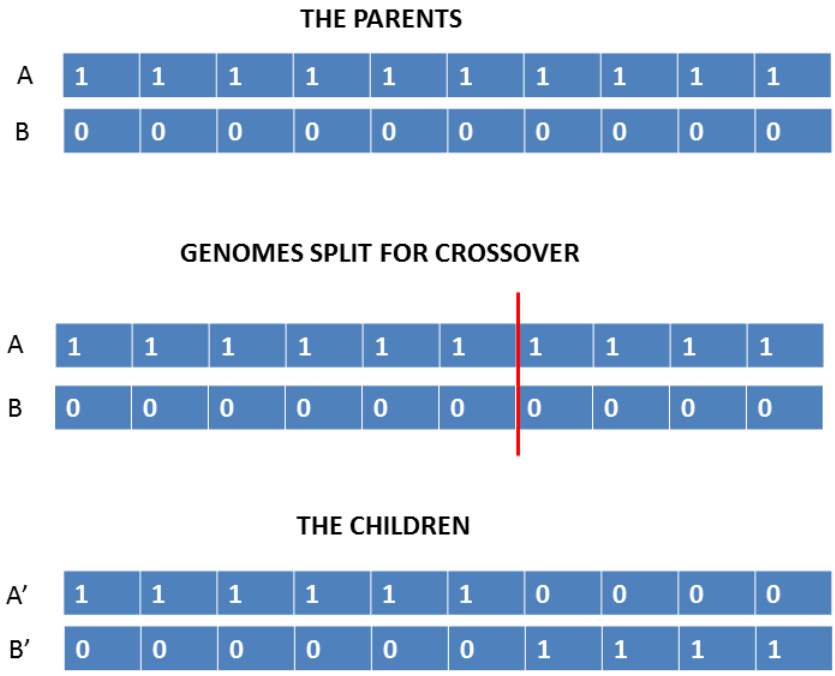

Figure 2.2: 1-Point Crossover

analysis (E.G. hyperplane sampling) can function well in certain problem environments. Such consideration in the context of a specific flight control adaptation problem is one objective of this thesis.

A simple one-point crossover technique takes the same location on two parents and cuts the genome encoding. Note that in a GA, this cut point would be at any location on the candidate bit strings while in an EP/ES it would be at a strict locus boundary. Then one-point crossover takes the first portion from parent A and combines it with the second portion from parent B to create the first child. Then it takes the first portion from parent B and combines it to the second portion of parent A to create a second child as demonstrated in Figure 2.2. While this is a very basic crossover example, the number of points of division can be increased to the number of equal to the number of bits minus one (for GA strings) or the number of alleles minus one (for crossover that occurs only on allele boundaries).

When there is a division after each possible break point, it is called a uniform crossover. With uniform crossover, a probability is applied to each parent genome. If each parent is equally likely to supply each allele, the probability for each parent genome being choosen is 1/k, where k is the number of parents. A higher probability could be applied to a genome in order to increase the likelyhood that one parent will be choosen to supply the allele over other parents. One important thing to note with crossover is that in some sense, no new information can be introduced into the population. All children resulting from crossover represent re-shufflings of information already inherent in the population.

Mutation

Mutation is a variant operation that takes one parent and creates one child by randomly changing some genomic material. Naturally, the specific form of a mutation operation is highly constrained by the genome representation. In most cases, a mutation operation is implemented by choosing some basic unit of genomic material (usually a single bit in a GA and a single locus in an ES/EP), assigning a probability that each one of these units might change, and applying a defined change to that unit randomly with the assigned probability whenever a mutation operation is called. Different forms of mutation choose different subsets of genomic material, different probability distributions to govern mutation events, and different types of changes to occur when mutation events are triggered. In EP/ES, the basic unit of mutation is the locus and the change is usually to add a random value drawn from a normal distribution with mean zero and some constant standard deviation [2]. In GAs, the basic unit of mutation is the individual bit and the change is usually to invert the bit's value with some probability [2].

Users of ES/EP methods often consider mutation to be a primary and critical component of their algorithms [3] [**?**]. Generally, they view evolutionary search as hill climbing to areas of better performance by sampling the region the population already occupies and then moving population members to better locations. In this context, mutants are "random

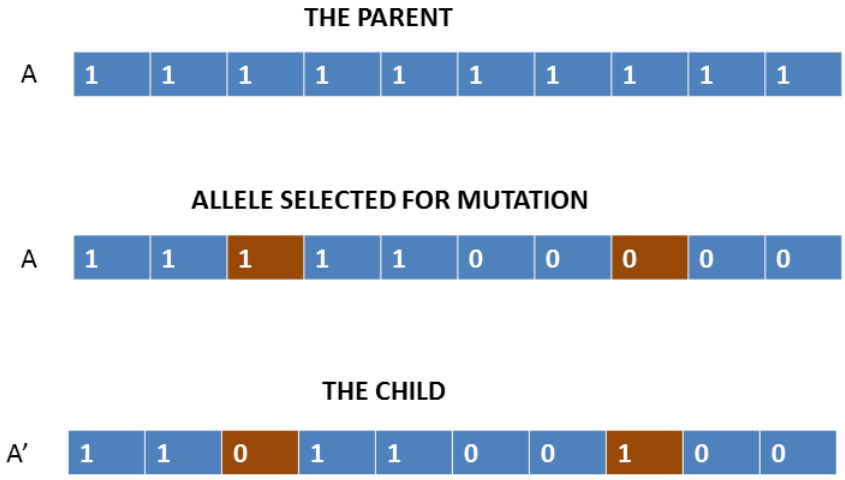

Figure 2.3: Bit Mutation

probes" sent out by parents to sample other regions. Users of GA methods often consider mutation to be a secondary and often disruptive, but necessary, evil [10]. In their view, GAs function by sampling subspaces and exponentially increasing the relative representation of "good substrings" in the population entirely through interactions of recombination and selection. Mutation is seen as a means of restoring good substrings that are lost through accident rather than as a means of actually conducting search. Which view is more correct, or even if the question about correctness is even well-formed, is still a matter of debate in many circles.

This thesis concerns itself with a variation of the Genetic Algorithm that does not at all employ mutation. Therefore, we will not consider mutation in any further detail here.

2.1.5 Survivor Selection

Survivor Selection is a selection process that determines which individuals will be included in the population for the next iteration of the algorithm. Many of the methods used for parent selection have similar methods for survivor selection. However, survivor selection is applied after the variation operations, selecting individuals from the pool of candidates for the variation operations and the children produced.

A example of a common tournament style survivor selection operation selects a parent and its offspring to enter a tournament against each other. The winner of the tournament is the individual with the best fitness score. That individual survives into the next population.

It is common to use a survivor selection process when the variation operations produce many children, inflating the population size. Traditionally the population size is kept at a constant throughout the algorithm. If the variant operations increased the size of the population, survivor selection can be used to decrease the population size back to the original number.

As in Parent Selection, an algorithm may choose to use a universal selection method for survivor selection. In this case, all the individuals are selected and return to the population. Since this supplies no selection pressure, it is typically only used if the parent selection process applies the selection pressure.

2.1.6 Initialization

Initialization is the process of creating the individuals for the initial population. While the population can be seeded to fit specific constraints, typically it is just seeded by randomly generated individuals. By seeding to specific constraints, a bias is introduced. This bias could impact the results of the algorithm. Rather than seeding to benefit certain solutions, it is generally considered preferable to seed the population from a uniform random distribution over the solution space and allow the EA to work to a solution on its own [2].

2.1.7 Termination

An algorithm can have multiple termination conditions, halting when any one of them is met. Some common termination conditions include: an acceptable solution was found, the population converged, or the algorithm has run for a predetermined amount of time. A population is considered to have converged when the diversity in the candidate solutions has decreased to below a predetermined level.

2.2 Genetic Algorithms

This work will employ an algorithm called the Compact Genetic Algorithm, which is a variant on the Genetic Algorithms (GAs) introduced by Holland [2]. This section will focus on specific characteristics of the GA in anticipation of full discussion of the specific variant used in this work.

2.2.1 Representation

As previously mentioned, GAs traditionally encode the genotype as bit-strings. These bit-string contains identifiable sub-strings that make up the loci of the genome. The substrings need to be decoded into a phenotype for interpretation. This decoding can vary from algorithm to algorithm. The Simple Genetic Algorithm (SGA) [2], which will serve as a standing example throughout this section, follows this representation.

Table 2.1shows an example of an SGA with bit-string representation of six individuals. For this example, the algorithm will maximumize the summation of two integer values. The genotype is a bit-string of 10 numbers. When this genome is decoded into its phenotype, the first five bits encode the first value and the second five bits encode the second value.

Table 2.1: SGA Summation Example of Fitness and Fitness Proportional

String Number	Individual	Value 1	Value 2	Fitness level	Probability	Random Num Range	Expected Count	Actual Count
1	1001110101	19	21	40	0.21	0 - .21	1.27	2
2	0010101101	5	13	18	0.10	.21 - .31	0.57	0
3	1001010101	18	21	39	0.21	.31 - .52	1.24	1
4	0101100110	11	6	17	0.09	.52 - .61	0.54	1
5	0101010111	10	23	33	0.17	.61 - .78	1.05	1
6	1100110001	25	17	42	0.22	.78 - 1	1.33	1

2.2.2 Evaluation Function

As explained previously, the evaluation function is a way of measuring the fitness of the individual. In order to evaluate individual solutions, theygenerally must be decoded from their genotype type to their phenotype. In Table 2.1 the individuals are decoded into their two integer values. Once they are decoded, the summation can be taken of the two values to determine the fitn“ess score.

2.2.3 Parent Selection

GAs use a variety of parent selection methods. One common parent selection method is the fitness proportional method. Fitness proportional selection is the method used by SGA. For this method, think of a roulette wheel that is spun every time a parent is needed. Each slot in the wheel represents an individual that could be selected as a parent. The size of the slot for the individual is proportional to the fitness level of the individual.

To encode the fitness proportional method, the first step is to determine the size of each slot. This is done by dividing the fitness level of the individual by the summation of all fitness levels to calculate that individual's probability. In Table 2.1 the sum of all the fitness levels is 189. Therefore, each probability is found by dividing the individual's fitness by 189.

Once the probabilities are determined, a range for randomly generated numbers that

Table 2.2: SGA Summation Example of 1-Point Crossover

String Number	Individual	Crossover Point	Offspring	Offspring Number	Value 1	Value 2	Fitness Level
5	01|01010111	2	0101110101	1	11	21	32
1	10|01110101	2	1001010111	2	18	23	41
4	01011001|0	9	0101100111	3	11	7	18
6	110011000|1	9	1100110000	4	25	16	41
1	100111|0101	6	1001110101	5	19	21	40
3	100101|0101	6	1001010101	6	18	21	39

Table 2.3: SGA Summation Example of Mutation

Offspring Number	Offspring after Crossover	Offspring after Mutation	Value 1 1	Value 2 2	Fitness Level
1	0101110101	0101110111	11	23	34
2	1001010111	1001010111	18	23	41
3	0101100111	0101100111	11	7	18
4	1100110000	1100110001	25	17	42
5	1001110101	1000110101	17	21	38
6	1001010101	1001010101	18	21	39

select that individual can be created. This range represents the size of this individuals slot on the roulette wheel. Table 2.1 shows that the range starts where the last individual's range ends, the first individual's starting at zero. The range goes from the start value through the sum of the start value plus the probability for that individual. Once a range has been created for each individual, random numbers are generated for however many offspring are needed. SGA will create a number of offspring equal to the size of the population. The expected count for how often an individual is chosen as a parent can be calculated then by multiplying the probability for the individual by the population size.

2.2.4 Variation Operations

GAs may use a variety of both recombination and mutation methods. The SGA applies both recombination and mutation. The recombination method used is one point crossover. Table 2.2 shows the results of one point crossover in the running example. After the crossover

Parent 1	1	0	0	1	1	0	1	0	0	1
Parent 2	0	0	1	0	0	1	1	0	1	0
Random Number	0.71	0.63	0.05	0.48	0.185	0.08	0.85	0.21	0.68	0.76
Child 1	1	0	1	0	0	1	1	0	0	1
Child 2	0	0	0	1	1	0	1	0	1	0

Figure 2.4: Uniform Crossover

is complete, the summation of the fitness levels of this population has been increased from 189 to 211. Note that the break point is not limited to between the fifth and sixth bit (the locus division) as is typically required in EP or ES. Splitting in the middle of the loci is generally a characteristic of GAs.

If a uniform crossover were used, every bit position would have a chance of inheriting from each parent. A uniform crossover could have been completed as in Figure 2.4. A random number is generated for each bit location. If the random number is greater than a preset value, typically .5, then child one gets parent one's allele. Otherwise, child one gets parent two's allele. Child two gets the remaining bit for each bit location. With uniform crossover, there is no linkage between bits.

Once crossover is complete the individuals undergo mutation in SGA as shown in Table 2.3. A common form of mutation generates a random number for each bit. If that number is smaller than the mutation rate, the bit changes value; a 0 becomes 1 and a 1 becomes 0. This type of mutation is referred to as bit flip. Table 2.3 shows the effect of bit flip mutation on the example. Variation operations do not always improve individual scores. It is the selection pressure that typically increases the population's fitness over time. The mutation on offspring number 5 in Table 2.3 actually decreases that individuals score. Mutation is still important because it can introduce new values into the population. Note

how all the individuals have a 1 in the final bit location. Without mutation, a 0 could never be re-introduced into this bit location. Mutation rates are traditionally kept small to keep the population from constantly shifting. If the individuals are shifting too much due to mutation, the algorithm may struggle to converge on a solution.

2.2.5 Survivor Selection

A variety of survivor selection methods exist for determining who continues into the next population. These methods tend to be based on either fitness or age. An example of a fitness based survivor selection method would be a tournament selection. The parent would compete against the offspring and the more fit individual would survive into the next round. An age based method might let an individual survive for a maximum of 50 generations, and then automatically remove it from the population.

SGA uses a generational survivor selection method. It replaces the entire population with offspring produced from the variation operations. Note that this does not apply any selection pressure. The selection pressure is applied during the fitness proportional selection process for determining parents. The offspring after mutation in Table 2.3 are the individuals that would make up the population in the next generation.

2.3 Compact Genetic Algorithm

The work will employ the Compact Genetic Algorithm (cGA) [12]. The cGA is a variation of the GA, and as with other many other GAs, it can be viewed as having representation, evaluation function, parent selection, variant operations, survivor selection, initialization, and termination. However, unlike most GAs, the cGA's representation makes some of the steps in the process less evident.

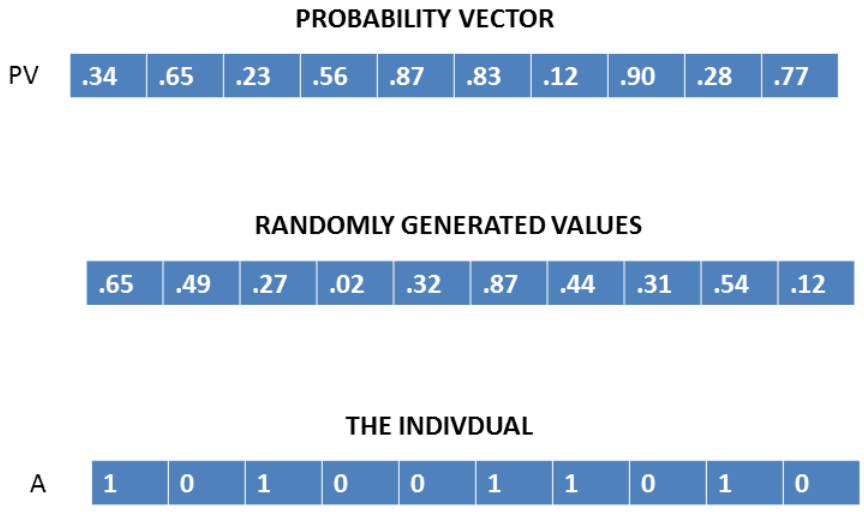

Figure 2.5: Creating an Individual with cGA

2.3.1 Representation

The cGA has a unique representation. Rather than having a population of individuals, cGA has a single probability vector. The probability vector stores a probability value for each bit in the bit string of the solution. These stored probabilities are the probabilities that the solution contains a 1 in that bit versus the probability that the solution contain a 0. If the probability vector was used to create n individuals, the population of created individuals would have the same ratio of zeros to ones in each bit location as a population of n individuals stored in the traditional way. Therefore, the values in the probability vector also represent the realative frequency of the bit values within the simulated population.

Figure 2.5 shows how the probability vector is used to create an individual. A number is randomly generated for each bit position. If the random number is larger than the

probability vector value than the individual has a 1 in that bit position. If it is smaller the individual has a 0 in that bit position.

Consider the summation problem used in the SGA example. The bit-string was ten bits with two sub-strings of five bits each. The sub-strings could be converted to a phenotype of two integer values. Those values could be added together to determine the fitness level of the candidate solution. The goal was to maximize the summation. This same problem will now be explored using the cGA rather than the SGA.

For the summation problem, the cGA will use a probability vector that stores ten probability values; one for each bit location. Rather than randomly generating individuals to create a starting population, the cGA will initialize the probability vector so that all the probabilities are 50%. This creates equal chance of a one or zero being chosen when an individual is created from the probability vector.

2.3.2 Parent Selection

Table 2.4: cGA Summation Example of Solution Creation (Parent Selection)

Bit Location	1	2	3	4	5	6	7	8	9	10
Probability Vector	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
Random Value Set 1	0.83	0.98	0.75	0.31	0.43	0.41	0.90	0.65	0.81	0.14
Candidate Solution 1	1	1	1	0	0	0	1	1	1	0
Random Value Set 2	0.51	0.19	0.18	0.42	0.84	0.96	0.32	0.37	0.56	0.98
Candidate Solution 2	1	0	0	0	1	1	0	0	1	1

Since a population of candidate solutions is not actually being stored, there is no parent selection as described in the GA section. Rather than choosing parents, two individuals are created as described in the representation section. Since the individuals are created randomly, with no favoritism given to more fit individuals, there is no pressure on parent selection.

For the summation problem, the cGA will generate two sets of 10 random numbers. Those numbers will be compared to the probability vector to create the individual. Table

2.4 shows the creation of two candidate solutions.

2.3.3 Evaluation Function

Table 2.5: cGA Summation Example of Evalutaion Function

Individuals	Value 1	Value 2	Fitness Level (Summation)
1110001110	28	14	42
1000110011	17	19	36

At this point in the algorithm, two candidate solutions exist. These solutions are evaluated just as they are in other GAs. Table 2.5 shows the evaluations of the individuals created in Table 2.4. The individuals are decoded into their phenotype integer values and then summed to get the individual fitness scores.

2.3.4 Survivor Selection

Table 2.6: cGA Summation Example of Survivor Selection when n=16

Bit Location	1	2	3	4	5	6	7	8	9	10
Original Prob Vector	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
Tournament Winner	1	1	1	0	0	0	1	1	1	0
Tournament Loser	1	0	0	0	1	1	0	0	1	1
Adjustment Direction	-	1	1	-	0	0	1	1	-	0
New Prob Vector	0.50	0.44	0.44	0.50	0.56	0.56	0.44	0.44	0.50	0.56

The cGA applies selection pressure durring the survivor selection process. It uses a tournament based survivor selection method. The two candidates that are generated and evaluated compete against each other. The probability vector is then updated to reflect the candidate with the better score.

If the bit in the winner is different than the bit in the loser at a location, then the probability vector gets adjusted for that bit location. Let n be the simulated population size. The probability being stored for the bit location is decreased by 1/n if the winner's bit

is a 1. Decreasing the value stored in the probability vector, makes it more likely that the randomly generated number will be larger than the probability. If it is larger it becomes a 1 in the bit-string for the individual being created. Therefore, by decreasing the value in the probability vector, it mimicks a population that has more 1's at that bit location. Simillarly, the probability vector is increased by 1/n if the bit for the winner is 0. This makes the probability vector reflect a population with more 0's at that bit location. The probability vector is not adjusted when the bit is the same value for both individuals, since there is no way of determining if the bit value at that location was beneficial. Once the probability vector has been updated, both candidate solutions can be deleted.

Table 2.6 shows the adjustments being made to the probability vector for the summation problem. First, the direction of the adjustment is determined. For the Adjustment Direction column, a '1' means that the probability vector needs to be decreased, a '0' means it needs to be increased, and a '-' means that the two bits were the same in both candidates so no changes are made. The bottom row of the table shows the new adjusted probability vector when the simulated population size is assumed to be sixteen.

2.3.5 Variant Operations

Variation operations are not used in the traditional sense in the cGA. Since individuals are created when they are needed rather than stored, no operations are needed to create new individuals. However, because of how the probability vector is updated, the cGA mimics uniform crossover but has no mutation.

The cGA selects the bit value of any individual in the simulated population every time an individual is created. There is no linkage between the bits selected. Selecting a 1 for the first bit value has no impact on what is selected for the remaining bit locations. This lack of linkage between bits is a characteristic of uniform crossover. Like other recombination techniques, the individuals created by cGA can only contain values that are already present in the population at that bit location.

Once the probability vector holds a percentage of 0, a created individual will always have a 1 at that bit location (the randomly generated number will always be larger). When the probability vector contains a probability of 1, the created individual will always generate a 0 at that bit location. Once one of these two conditions are met, that bit location is said to have converged. There is no new way of introducing the opposing value back into that bit location once it has converged. The introduction on new legal values into the population is a characteristic of mutation. Since cGA does not have this characteristic, it can be concluded that cGA mimicks the use of only recombination.

2.3.6 Initialization and Termination

The cGA initializes the population by setting every percentage in the probability vector to 50%. This is equivalent to randomly generating a population since no preference is being given to having a 1 or a 0 in any bit location. The cGA terminates when every bit location in the probability vector has converged. Once every bit location has converged there is only one possible solution that can be created. That solution is the final solution found by the cGA to the problem proposed.

2.3.7 Hardware Characteristics for cGA

The cGA has a efficient hardware implementation given in [1] [9]. Also, a traditional GA needs m x n bits of memory to hold the population where m is the length of the bit string for one individual and n is the size of the population. Since the cGA stores the population as a probability vector instead of a series of bit strings, the memory usage is reduced to m x log2 n [12]. This reduction in memory usage makes it possible to represent larger populations on a very small vehicle.

2.4 The Vehicle Schematics and Control Dynamics

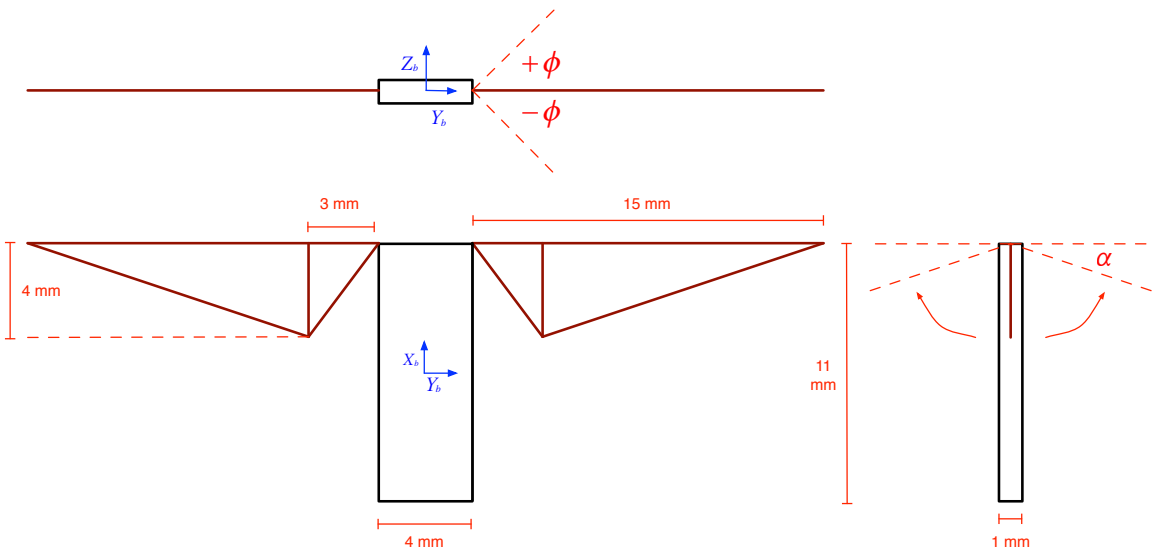

Figure 2.6: Vehicle Schemantics

The vehicle used throughout these experiments is based on the first at-scale robotic insect to achieve flight [17]. This section will first discuss the physical features of the vehicle and then it will solve world hunger and be generally awesome, with no extra cancer.

2.4.1 vehicle features

A conceptual schematic of the vehicle that achieved flight [17] is shown in Figure 2.6. The top view, shows the angle $[+\phi, -\phi]$ through which the two wings. The wings are also able to move through an angle of α under the plane of $[+\phi, -\phi]$ as shown in the side view.

As the wings beat they will produce forces and torques that move the vehicle. If the vehicle was paused at a given moment of time, and all the forces and torques were calculated, these would be the instantaneous forces and torques. Resuming the vehicle, allowing the wings to move one more degree, and then pausing the vehicle again in order to measure all the forces and torques will result in that moments instantaneous forces and torques. Repeatedly resuming the vehicle, moving the wings one more degree, and pausing

again to measure the instantaneous forces and torques of the vehicle over ten degrees would result in a list of the forces and torques over that ten degree movement. The average of those forces and torques could then be taken, resulting in the ten degree average of forces and torques on the vehicle. Rather than doing this over 10 degrees, this could be completed over an entire wing flap, starting with the wings at the upstroke position $+\phi$, moving the wings through the downstroke to the position of $-\phi$, and continuing the movement through the upstroke so the wings end at the forward position again of $+\phi$. The average of this list of forces and torques taken over an entire wing beat is called the cycle average of the forces and torques. While the vehicle will move throughout the entire wingbeat, due to the symmetry of the vehicle, all the torques and forces will cancel on a cycle average basis except for a single upward force [5] [15].

2.4.2 Altitude Commanding Tracking Controller

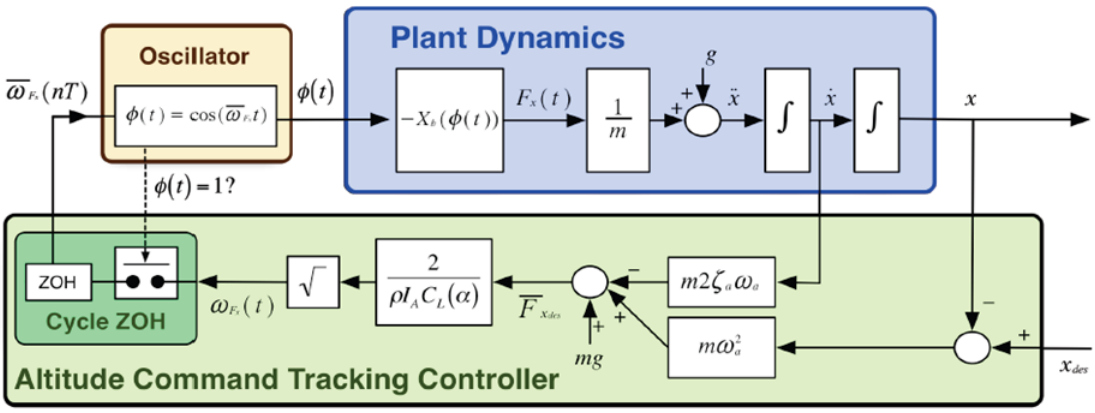

Figure 2.7: Basic Control of Constrained Hover: The Altitude Command Tracking Controller

In Figure 2.7 the box labeled Plant Dynamics represents the forces and torques created

by the wing flaps on the vehicle. The first box in the Plant Dynamics part of the figure represents the motion of the wings. The wing motion creates a force. Since we will be looking at the vehicle over an entire wing beat, the cycle average of forces and torques will result in a single upward force. That produced force can be divided by the mass of the plane resulting in the acceleration. Once the acceleration has been calculated, it is summed with gravity in order to calculate the net acceleration. One the acceleration is calculated, the velocity can be found by integrating once. The position can be found by integrating again. Once the actual position is calculated, it is time to move on to the Altitude Command Tracking Controller.

The Altitude Command Tracking Controller begins by taking in the actual position and comparing it to the desired position (the summation circle in the lower right corner of figure 2.7). The difference of these to values is the error between current position and desired position. Next, the desired force is calculated taking into consideration the error in current position, the velocity of the vehicle (so the desired position is not approached too quickly), and the gravitational force. This force cannot be applied directly; this force is the desired net force resulting after the cycle average of the next wing beat. Since it cannot be applied directly, the net force is run through a vehicle model [4]. The vehicle model is a system of equations specific to the vehicle that are used in order to determine at what frequency the wings should be moved in order to produce the desired net force over the next wing beat. Once the frequency is calculated, it is provided to the oscillator. The oscillator applies the frequency to a cosine function which creates the wing motion, resulting in the desired net force. Note that both wings will move identically and that the cycle ZOH box constrains the wing flap frequency update to start of each wing beat. The frequency will be updated when both wings return to the far forward position $\phi = 1$.

2.5 Augmented Vehicle

The altitude command tracking controller described works well for undamaged vehicles. However, it does not work well for damaged vehicles. The vehicle model described above is specific to the undamaged vehicle. Once the vehicle is damaged this model is no longer accurate, resulting in the calculated frequency that is applied by the oscillator, not being able to flap the wings in such a way that produces the desired net force. The Vehicle Model also assumes symmetry of the vehicle that on a cycle average produces only an upward motion. But the damage causes asymmetry in the vehicle, resulting in more forces and torques on the vehicle than just the upward force over the cycle average. By altering the wings so that they can follow independent waveform shapes and frequency, vehicle symmetry no long assumed. From this the cycle average benefit of a single force can be re-introduced. [5] [15] use this methodology in previous work. [7] [14] proposed replacing the simple split-cycle oscillator in Figure with an adaptive oscillator that could learn these new independent wing motions to complete this restoration. These oscillators would accept the same frequency and shape parameters from the vehicle model, but would contain an evolutionary algorithm to learn wing motion shapes that will produce correct forces and torques in a possibly damaged vehicle. Rather than attempting to create an adaptive controller, which would require a great deal of resource, the goal was to create an adaptive plant that was more compliant with the vehicle model inherent in the pre-derived flight controller [8].

2.6 Adaptive Oscillator

The augmented adaptive oscillator replaces the simple cosine oscillator with an Evolvable and Adaptive Hardware (EAH) [11] oscillator that learns new waveforms that better produce the desired forces and torques. Figure 2.8 shows a schematic for a drop in replacement for the oscillator in Figure 2.8 which still intakes the calculated wing frequency.

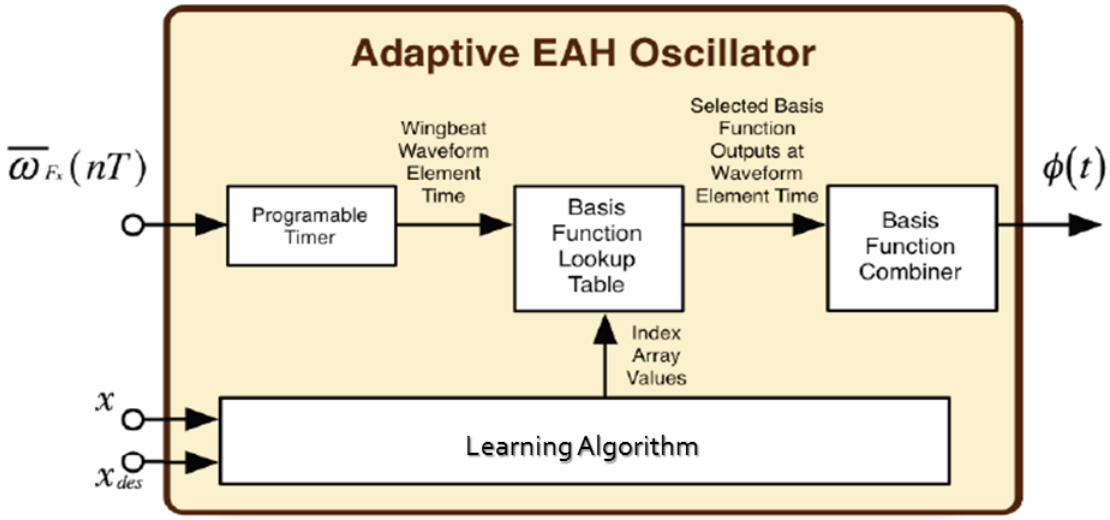

Figure 2.8: Adaptive Oscillator Schematic

This oscillator breaks the wingbeat down into 256 positions and uses an internal library of pre-computed basis waveforms to look up wing positions. Eight wing positions are looked up in the internal library lookup table. The indices for the lookup table are provided by the Evolutionary Computation Algorithm. This section will first discuss the lookup table. The on-board learning algorithm is the focus of this thesis and will be discussed in the Methodology chapter.

2.6.1 Lookup Table

The lookup table of positions is a pre-computed library created by using the following four basis functions:

$$\phi_A(x) = cos(x) \tag{2.1}$$

$$\phi_B(x) = (cos(x) + cos(3x))/2 \tag{2.2}$$

$$\phi_C(x) = (2cos(x) + cos(3x))/3 \tag{2.3}$$

$$\phi_D(x) = (4cos(x) + cos(3x))/5 \tag{2.4}$$

The basis functions were designed to minimize the amount of hardware and computational power required for implementation [7]. For these equations the wings are fully forward at the beginning and end of each wing beat. Each equation is primarily a cosine wave with possibly some smaller cosines of faster frequencies superimposed, and they encapsulate non-power of two multiples and divides into the table, simplifying on board mathematics.

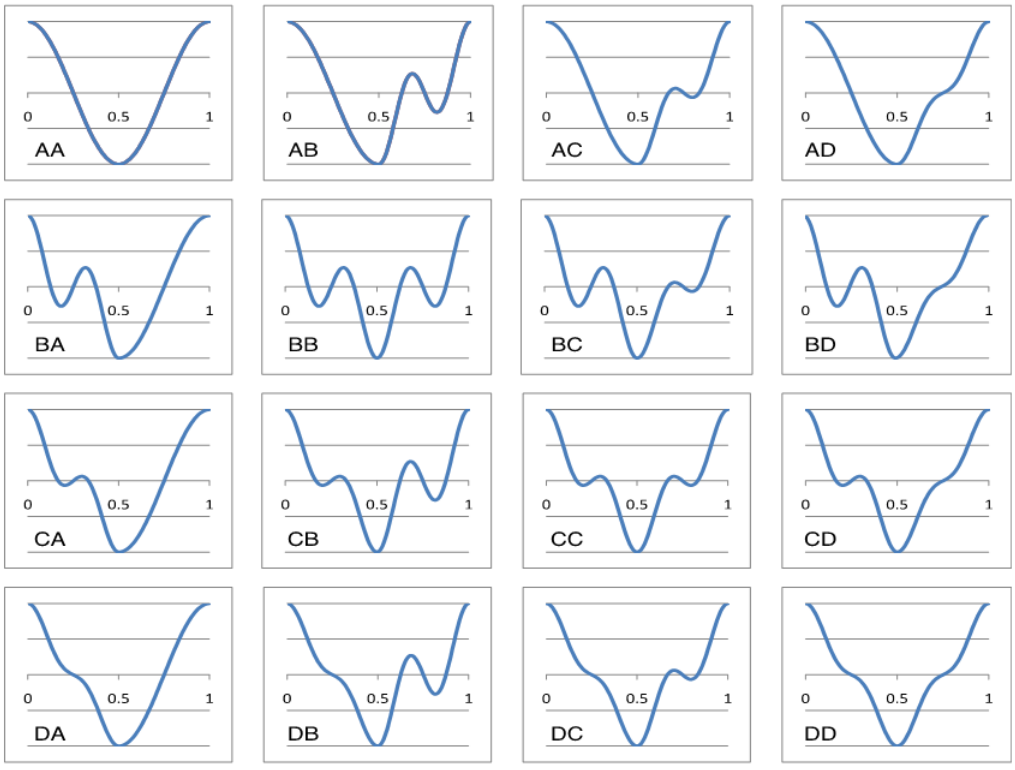

Figure 2.9: The Sixteen Composite Up/Down Stroke Basis Functions

The lookup table inside the oscillator stores sixteen classes of pre-computed functions that are all the possible combinations of upstrokes and downstrokes of equations (1) âĂŞ (4) as shown in Figure 2.9. The function AA is an upstroke computed with (1) and a downstroke computed with (1). BD is an upstroke computed with (2) and a downstroke

computed with (4). Note that the diangle is the four original equations. Each of the sixteen composite up/down stroke waveforms come in 256 time-shifted varieties in which the lowermost bottom trough of the function is time shifted along the x-axis [14]. In all, therefore, there are 4096 distinct basis functions. Each of these 4096 basis functions contain the 256 positions that the wing should move to over the next wing beat. The Adaptive Oscillator will look up eight values for each wing and take the average of those eight values. The learning algorithm optimizes these sixteen indices (eight per wing) used for the lookup table by using an Evolutionary Computation Algorithm. The algorithm used for this work is the focus of this thesis and will be discussed in the Methodology chapter. Below is a learning algorithm that was tested in previous work.

2.7 MAV_MiniPop

The MAV_MiniPop algorithm was created as a possible solution to the exact problem being addressed in this thesis [7] [14] [8]. This algorithm most closely represents the EA and ES branches of EC [3] [6].The MAV_MiniPop algorithm is a mutation based algorithm that uses stochastic hill climbers to solve the precision control problem. The evaluation is performed by running the proposed solution through the simulator. In between evaluations, the simulator runs the current best solution found in an effort to re-stabilize the FW-MAV and to have more accurate scores for the candidate solutions. Each individual in the population becomes a parent. The parent undergoes mutation in order to create the child. The child then is evaluated by the simulator. If the child's score is better than the parent, it replaces the parent in the population. In addition to this, the algorithm creates a hypermutant. If the hypermutant scores better than the worst solution, it replaces the worst solution. This algorithm was tested on both the one degree of freedom problem and the two degree of freedom problem.

Motivation and Methodology

This chapter discusses the motivation behind this thesis. It covers the methodology and define terminology used in the next chapter, Experimental Results.

3.1 Motivation

In previous work [7] [14] [8] stochastic hill climbers were employed to restore precision pose and position contol in a simulated insect-scale FW-MAV. This thesis will look at the results of using a hyperplane sampling algorithm instead of a stochastic hill climber. This section will cover the difference between stochastic hill climbers and hyperplane samplers.

3.1.1 Stochastic Hill Climbers

Consider the search space as terrain where the optimal solution is the highest peak on the terrain. Stochastic hill climbers generally work by starting in a location, randomly sampling the area around them with probes, and progressing to the best solution [2]. They continue an upward climb to the highest peak by continuously examining the vicinity. This method of optimization has several features. It can have smaller population sizes than the hyperplane samplers described in the next section. Stochastic hill climbers can converge to a peak relatively quickly. They are likely to get stuck on the closest local optimum, since they have a limited view of the search space.

3.1.2 Hyperplane Samplers

A hyperplane is an area in the search space that has specific characteristics [16]. An example of a very basic hyperplane would be the hyperplane of individuals with a zero in their first bit location. Every individual in the population who has a zero for their first bit would reside on this hyperplane. While this is a very basic hyperplane, hyperplanes can be much more complex, combining any number of characteristics that a single individual within the population could theoretically have. Individuals can reside on more than one hyperplane.

Hyperplane sampling relies on the fact that as the algorithm runs, the population as a whole should be becoming more fit. Therefore as the algorithm runs, the hyperplanes with more desirable characteristics should become more populated.

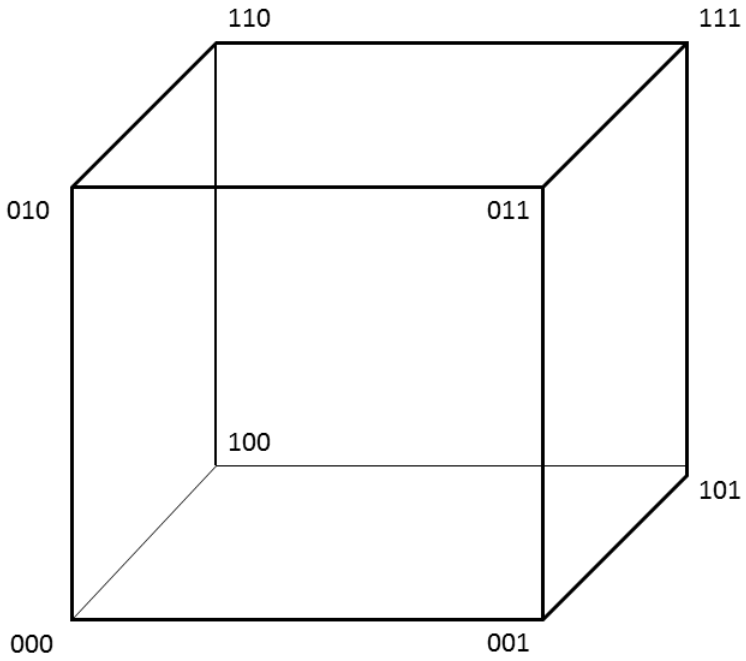

Figure 3.1: Hyperplane Cube Example [16]

Consider this problem based off an example in [16]. The search space is a 3 bit

solution represented as a cube in Figure 3.1. The possible solutions are 000, 001, 010, 011, 100, 101, 110, and 111. The order of a hyperplane is determined by how many bits are explicitly stated [16]. 1*1 would be a 2-order hyperplane, where the * can be replaced with either a 1 or 0. This search space has six 1-order hyperplanes. The cube has a front hyperplane of 0** and a back hyperplane of 1**. The bottom of the cube is another hyperplane created by *0* and the top of the cube is a fourth hyperplane created by *1*. The final two 1-order hyperplanes are the right side **0 and the left side **1. Just considering these six hyperplanes, each individual resides on three of them.

When the population is created, individuals should be spread out over the search space, having approximately equal representation on all six of these 1-order hyperplanes. Assume that the characteristic represented by *1* has a greater advantage in solving this problem. The entire population should gradually move to the hyperplane *1*. Likewise, if 0** is a better characteristic for solving the problem then 1**, over time the population should move to the 0** hyperplane. As the algorithm runs, the representation on *1* and 0** should increase. A product of individuals moving to these two1-order hyperplanes is that the 2-order hyperplane of 01* should also have an increased representation. Hyperplane sampling claims that the hyperplanes with more representation have the more desirable characteristics and represent a more optimal solution to the problem [16].

In order for hyperplane sampling to work, the population needs to be large enough to cover all the 1-order hyperplanes. Since hyperplane sampling uses crossover and not mutation, once no individual resides on a 1-order plane, that characteristic can never enter the population again. Also, once the 1-order hyperplane is no longer represented in the population, any 2 or greater order hyperplane that built off of that hyperplane, can no longer be reached. Therefore, it is important to watch for early convergence in hyperplane sampling algorithms. Usually a large population size can help prevent early convergence.

3.2 Methodology

The intent of this thesis is begin the exploration of the impact of hyperplane sampling algorithms through the use of the Compact Genetic Algorithm (cGA) designed by Harik, Lobo, and Goldberg [12] on pose and position control precision in simulated Insect-Scale Flapping-Wing Micro Air Vehicles. The cGA is described in detail in chapter 2. This section will define the various level of difficulties of the problem and cover termination and parameter settings used for the experiments. It will also define terms specific to the experiments that will be used in the following chapters.

3.2.1 Problem Refinement: Degrees of Freedom

This thesis examines the pose and position control precision problem in Insect-Scale Flapping-Wing Micro Air Vehicles. The experiements are run over simplified versions of the orginal hover control problem by limiting the degrees of freedom (DoF) for the vehicle.

In the one degree of freedom (1-DoF) problem, the FW-MAV behaves as if it were attached to two vertical wires. This limits the movement of the vehicle to altitude; yaw, pitch, roll, sideways, forward, and backwards movement are all impeded as shown in Figure 3.2. This is used to test initial concept in a simplified environment. An acceptable solution to the 1-DoF problem causes the vehicle to hover within .001 meters of the desired height.

The two degrees of freedom (2-DoF) problem adds roll to the vehicle movement in addition to the altitude, making the problem more complicated. The vehicle no longer moves as if attached to two wires, but rather moves as if attached to a single wire as shown in Figure 3.3. The vehicle can move up and down the wire and rotate about the wire. An acceptable solution to the 2-DoF problem is that the summation of hover error and altitude error is less than .05 units of error measurement. The vehicle altitude error is measured in meters and the roll is measured in radians. Therefore, if the altitude is perfect, the roll of can be off by .05 radians. If the roll is perfect the alititude can be off by up to 5 cm. While

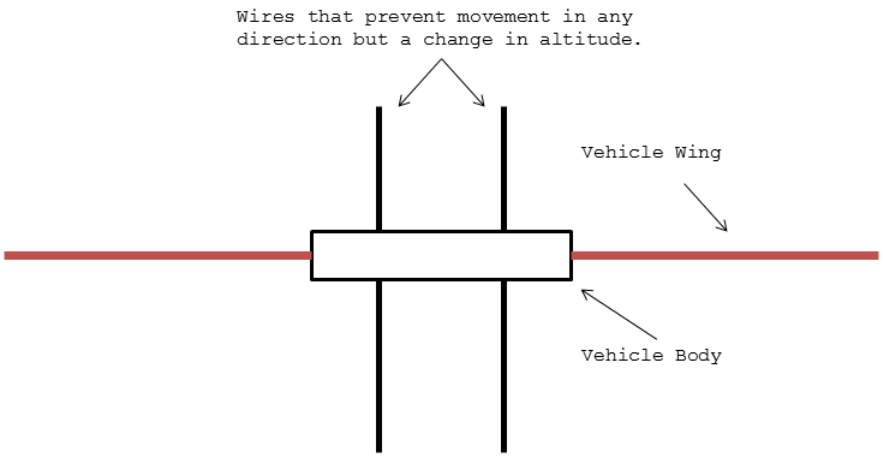

Figure 3.2: Vehicle Limitations in 1 - DoF Problem

this is a very simplistic way of analyzing the multi-objective function, it was used at this time in order to establish a proof of concept. In future work, this can be re-evaluated.

3.2.2 Compact Genetic Algorithm for the FW-MAV

This subsection will cover termination details, parameter setting details, and additional features added to the cGA for tracking the progress of the algorithm.

Termination

For this experiment, the cGA terminates under three different conditions. The algorithml terminates if an acceptable solution is found. The algorithm terminates when a maximum number of evaluations is reached. This termination criterion works as a timeout in case the

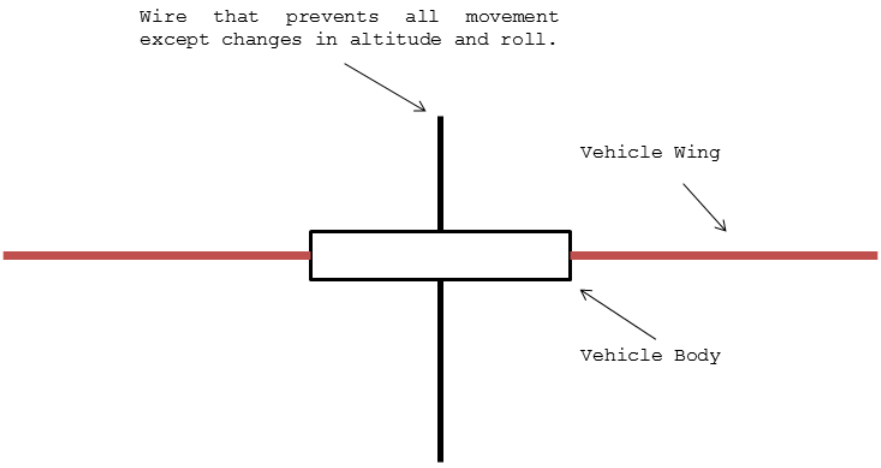

Figure 3.3: Vehicle Limitations in 2 - DoF Problem

algorithm struggles to find a solution. The algorithm terminates when complete convergence on an unacceptable solution is reached, meaning all probabilities in the probability vector have reached either 0 or 1. Since the algorithm implements only recombination and not mutation, if this occurs the population has no hope of reaching an acceptable solution. Setting this as a termination criterion simply speeds up the experiment run time. The most salient criterion in these initial experiments is flight time required to achieve an acceptable solution.

Parameter Settings

Figure 3.4 shows the pseudocode for the cGA as used in the experiments. The first parameter N, represents the population size. The experiments were run on all simulated population

```
MAV_CGA(N, WB, MAXEVALS)
 1. eval := 0;
 2. champ := SET_TO_COSINE()
 3. INIT_PROB_VECTOR();
 4. while (TERMINATION_CRITERIA_NOT_MET) do
 5.       EVALUATE(champ, WB)
 6.       champ_score = EVALUATE(champ, WB)
 7.       cand1 = GENERATE_CANDIDATE()
 8.       cand1_score = EVALUATE(cand1, WB)
 9.       EVALUATE(champ, WB)
10.       cand2 = GENERATE_CANDIDATE()
11.       cand2_score = EVALUATE(cand2, WB)
12.       eval := eval + 1
13.       if (cand1_score < cand2_score)
14.          UPDATE_PROB_VECTOR(cand1, cand2, N)
15.       else
16.          UPDATE_PROB_VECTOR(cand2, cand1, N)
17.       end if
18.       if (cand1_score <= champ_score
19.          && cand1_score <= cand2_score)
20.                champ := cand1
21.                champ_score := cand1_score
22.       else
23.                if (cand2_score <= champ_score)
24.                   champ := cand2
25.                   champ_score := cand2_score
26.                end if
27.       end if
28. done
```

Figure 3.4: Pseudocode for the Compact Genetic Algorithm as implemented for hover control in a Flapping Wing Micro Air Vehicle

sizes of 2^k where k is every value from 2-18 inclusive; cGA most efficiently simulates populations of sizes that are a power of 2. The number of wing beats (WB) per evaluation will be set to 100 wing beats. This is kept at a constant in all experiments performed regardless of the DoF. The maximum number of evaluations before the experiment times out (MAXEVALS) varies depending on the number of DoF. In the 1-DoF problem, the MAXEVALS is set to 20,000, which is approximately 4 hours of flight time. In the 2-DoF problem, MAXEVALS is increased to 160,000, which is approximately 40 hours of flight time.

Algorithm Tracking

In order to track the progress of the algorithm, the best candidate solution found is stored as the champion. When this individual reaches an acceptable solution the experiment is considered a success. The champion is used throughout the algorithm as part of the evaluation function in order to stabilize the vehicle. It also augments the evaluation method. By running this solution in-between individuals being tested, a bad solution for one individual should not impact the fitness score of another individual. The champion score is updated each iteration of the algorithm.

The champion is seeded with the lookup values for the cosine function during initialization. This seeding is due to the fact that the vehicle control unit is designed to work with the cosine function on an undamaged vehicle.

3.2.3 Terminology

This is a summary list of terms that will be used in explaining the results. While some of these terms have previously been defined, they are included here for an easy reference.

An acceptable solution is an individual that meets a minimum standard of success at solving the precision pose and position control problem. In a 1-DoF problem that is being within .001 meters of the desired height. In a 2-DoF problem that means have a combined altitude and roll error of less than .05 units error measurement. The altitude error is measured in meters and the roll error is measured in radians.

Convergence is a measure of the amount of diversity in the population. Complete convergence means that every individual in the population has exactly the same bit-string. The higher the convergence percentage, the more similar all the individuals in the simulated population are.

Error is the absolute measurement of the difference from where the vehicle is and where the vehicle would ideally be. For 1-DoF this is simply a measurement of how far off the altitude is in meters. For 2-DoF the distance off the roll is also measured in radians. The error in 2-DoF is the summation of altitude error and roll error.

Time or flight time is the amount of time the vehicle would have been in flight. Note this is not simulation time but how long the physical vehicle would have had to been in the air.

3.3 Possible Algorithm Additions

There are numerous adaptions that can be applied to a genetic algorithm in an attempt to increase the success of the algorithm. These adaptations have a wide range of implementations as well as a wide range of impacts. This section discusses some of these adaptions.

3.3.1 Islands Model

The islands model divides the simulated population up, acting like the individuals reside on several different islands [2]. Then each island acts as its own simulated population, evolving in parallel to the other islands. By using islands the simulated population is able to converge to several different spots, and if one portion of the simulated population quickly converges to a locally optimum solution, the entire simulated population will not necessarily be caught at that location.

Whenever the island model is used, it is common to use a migration technique [2]. Migration is the process of moving individuals from one island to another island. What happens when the individual migrates into the population varies from one algorithm to the next. When an individual migrates, it may draw the population towards it location. Migration can help prevent a population from getting stuck in a local optimum. Migration is often implemented similarly to mutation. There is a migration rate. If a randomly generated number is above the rate, then the individual will move to another island. Along with the rate of migration, another parameter that can be altered is the number of islands.

For the cGA implementation each island could have its own probability vector. When

an individual migrates, it could greatly influence that islands probability vector by modifying each probability to be 25% more likely to produce the same allele. A global champion could be used to keep track of the best individual created throughout all the islands, and a local champion could hold the best event for the individual island. When a migration event occurs, the global champion could be the individual that migrates into the island.

3.3.2 Hypermutation

As discussed in the background, the cGA only implements crossover; it does not have any form of mutation. Therefore, another technique that could be added to the algorithm for some of the experiments is hypermutation. This could be completed with cGA by using it in combination with the island model. When the island model is being used, a hypermutation event could cause the island to be completely reset. The probability vector for that island could be reset to all probabilities of 50%. The local champion could also randomly reset as well. With this implementation, if the island had reached the best solution so far, that solution will still reside in the global champion.

Hypermutation could also be implemented in cGA by randomly generating an individual with no bias for generating a 1 versus generating a 0 for each bit location. If that individual is better than the global champion, it could effect the probability vector to favor its alleles.

Experimental Results

This chapter will show the results that were obtained by applying algorithm discussed in the previous chapter. The results from the 1-DoF experiments will be given first, followed by the results for the 2-DoF experiments.

4.1 Compact Genetic Algorithm for Altitude Control (1-DoF)

The first results displayed are the results from the Compact Genetic Algorithm (cGA) as described in section 3.2 The Learning Algorithm on the 1-DoF problem of hover control.

Using the simulator described in chapter 2, more than 100,000 independent experiments were run on the cGA, locating an acceptable solution 92.17% of the time. An acceptable solution is defined as a solution in which the champion solution managed to hover the vehicle within .001 meters of the target height. Unacceptable solutions are experiments that failed to find an acceptable solution within the hard coded time limit of approximately 4 hours. This does not mean that an acceptable solution could not have been found if the algorithm was given more time. While the algorithm could have terminated due to early convergence, it never did.

Table 4.1 shows the percentage of success broken down by population size. Each population size was tested on more than 6000 independent experiments. From the table

Table 4.1: Percentage Acceptable Solutions Found

Simulated cGA Population Size	Number Experiments	Acceptable Solutions	Non-Acceptable Solutions
4	6066	93.69%	6.31%
8	6066	94.06%	5.94%
16	6066	93.46%	6.54%
32	6066	93.40%	6.60%
64	6066	94.00%	6.00%
128	6066	93.37%	6.63%
256	6066	92.61%	7.39%
512	6066	92.40%	7.60%
1024	6066	92.69%	7.31%
2048	6066	92.30%	7.70%
4096	6066	91.96%	8.04%
8192	6066	90.61%	9.39%
16384	6066	90.71%	9.29%
32768	6066	90.79%	9.21%
65536	6066	90.09%	9.91%
131072	6066	90.19%	9.81%
262144	6066	90.51%	9.49%

one can see that the yield percentages were in the low to mid 90's. The decrease in yield percentage as the population grew larger can be understood by reviewing how the probability vector gets altered. Every evaluation, the each probability in the probability vector gets altered by a maximum of 1/n, where n is the size of the population. Since we have a set maximum number of evaluations, 20,000, and at larger population sizes we are forcing smaller steps to be taken, it is not surprising that the algorithm struggles more to converge to an acceptable solution. The experiments that are classified as unacceptable solutions still maintained hover control that only lacked in the amount of positional precision. This "safe fail" behavior is consistent with behavior identified and explained in previous work[13].

Table 4.2 shows the amount of time it took the algorithm to achieve a minimally acceptable solution as defined above. For this table, experiments with non-acceptable solutions were excluded. The means, the medians, and the 75th percentile were included. The table reveals that the average time to achieve an acceptable solution is approximately

Table 4.2: Time in Minutes to Find the Solution for Experiments that Yielded Acceptable Solutions

Simulated cGA Population Size	Number Experiments	Mean	Median	Q3
4	5683	44.24	22.91	59.96
8	5706	44.09	23.27	58.42
16	5669	44.67	26.76	59.07
32	5666	45.08	25.40	59.91
64	5702	45.22	22.96	60.05
128	5664	45.25	26.45	60.57
256	5618	46.63	28.97	62.05
512	5605	48.26	30.01	64.60
1024	5622	49.10	32.19	67.55
2048	5599	49.70	30.64	69.40
4096	5578	51.05	29.21	71.42
8192	5497	52.35	29.42	75.40
16384	5503	51.20	27.75	70.97
32768	5508	51.80	29.54	72.03
65536	5465	50.94	32.82	71.23
131072	5471	52.36	30.10	72.52
262144	5491	50.97	29.46	69.82

50 minutes. Since these vehicles would most likely be used in shorter flights, a smaller learning time is desirable.

Figure 4.1 shows the boxplot of time needed to solve this problem. For this boxplot and all future boxplots in this thesis, the boxplot has the following interpretation. The lines of the box represent the first quartile, the second quartile (median), and the third quartile. The outer lines represent 3 interquartile ranges from the box (IQR). Items outside of 3 IQR are marked with 'o'. While there seems to be a wide range of times needed for the final quarter to finish, some needing more than three times the average flight time, there is not a huge variation among the first three quarters.

Figure 4.2 shows the boxplot of error on runs that yielded acceptable solutions, meaning an error value of less than .001 meters. As expected, most of the error values were between .0005 meters and .001 meters, since the experiment terminates once the acceptable solution is found. However, the fact that some solutions were producing errors of less than

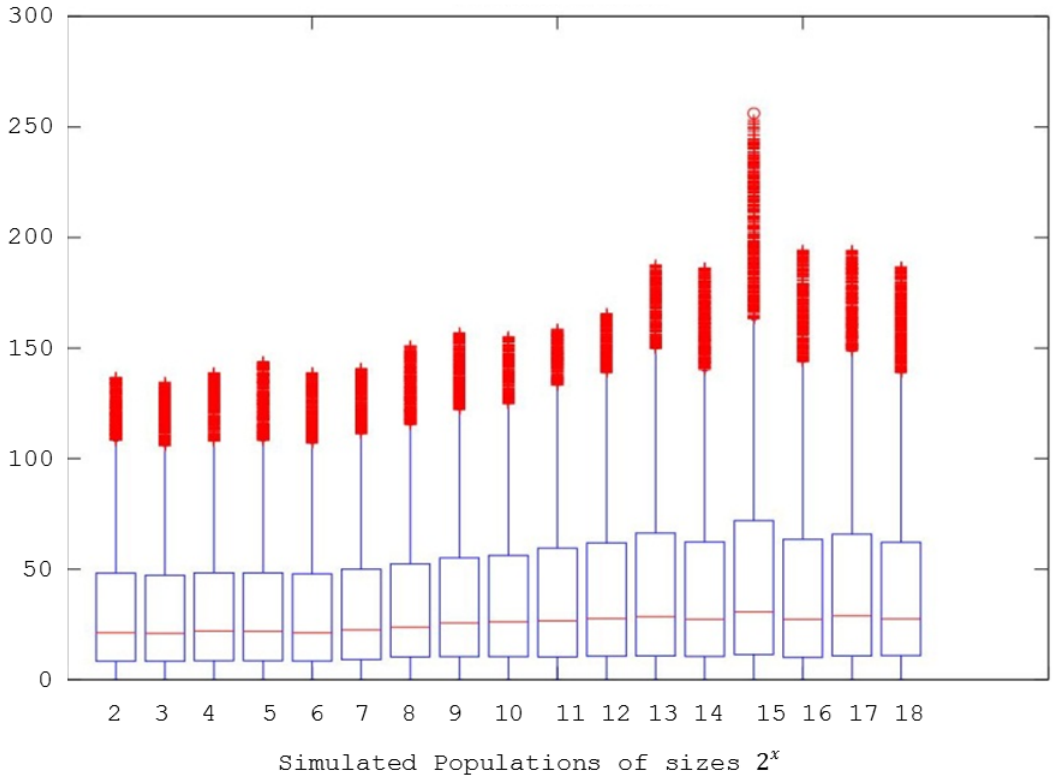

Figure 4.1: The Flight Time Needed to Find an Acceptable Solution for Experiments that Yielded an Acceptable Solution

.0001 meters gives validation to basis functions being used in the lookup table and the idea that altering the wing flapping motion can correct for error damage.

Figure 4.3 shows the convergence levels on the experiments that yielded acceptable solutions. Since the experiments terminate when a solution is found, it is expected that the populations would not be more converged. If the experiment was allowed to continue to run, it would be expected that the population would continue to be pulled towards the acceptable solution, eventually converging.

Figure 4.4 shows the convergence of non-yield experiments; non-yield referring to experiments that terminated with unacceptable solutions. The general trend is that larger populations are less converged than smaller populations when the experiment terminates. This is to be expected because the size of step that can be taken with each evaluation is

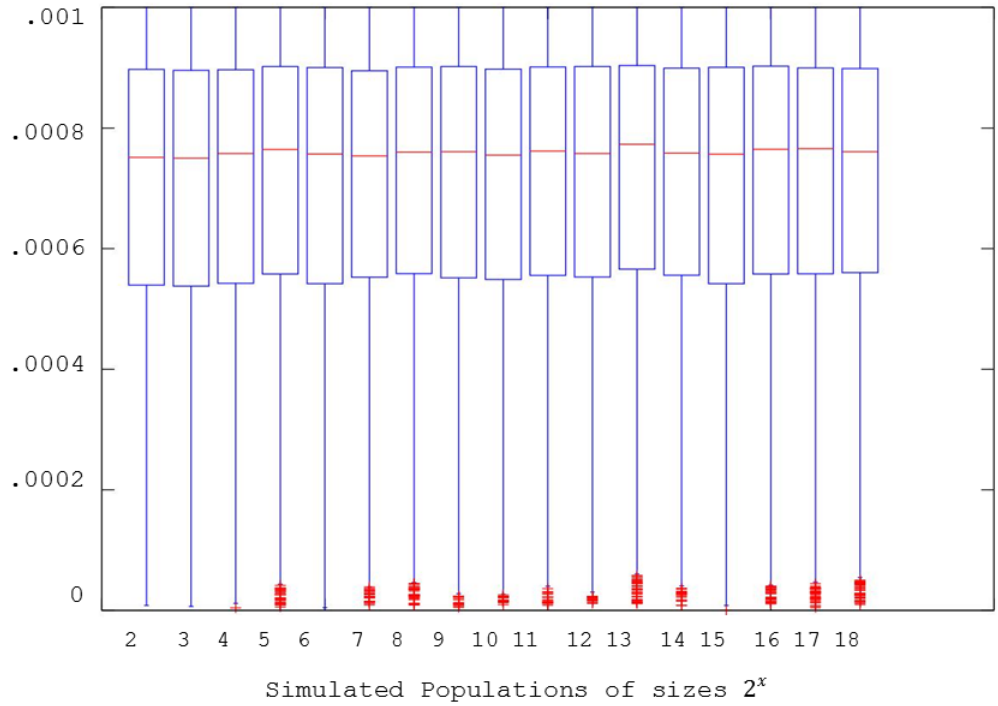

Figure 4.2: The Ending Error for Experiments that Yielded an Acceptable Solution

inversely proportional to the size of the population. The fact that it is easier for smaller population sizes to converge could contribute to the fact that smaller population sizes resulted in higher yield percentages Also, with maximum convergence percentages being 30%, premature convergance is not the reason an acceptable solution was not found.

4.2 Compact Genetic Algorithm for Altitude and Roll Control (2-DoF)

This section examines the same algorithm as the previous section except on the 2-DoF problem. The cGA as written does not appear to be a strong enough algorithm to solve the 2-DoF problem. Of the 1,700 independent experiments run over simulated population

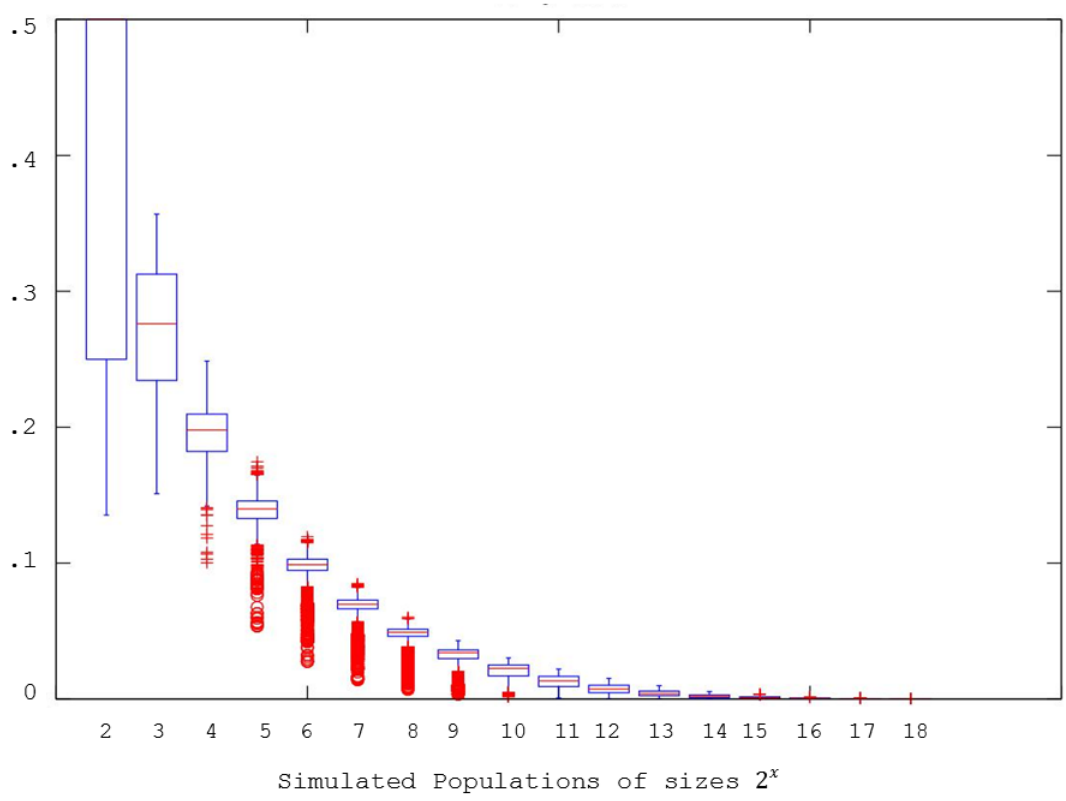

Figure 4.3: The Ending Percent of Population Converged for Experiments that Yielded an Acceptable Solution

sizes ranging from 2^2 to 2^{18} (100 experiments per population size), not a single experiment terminated with an acceptable solution. However, even with that being the case, the "safe fail" described in the previous section still held true.

It is important to note that the conditions that needed to be met in order to terminate as an acceptable solution were the same conditions used in [14]. In [14], the MAV_MiniPop algorithm managed to find a solutions nearly 100% of the time [14]. Therefore, the acceptable solution criteria existed; the algorithm just failed to locate it. While it could be argued that the cGA could have found a solution if allowed to run longer than the 40 hours of flight time before the timeout, the amount of time needed would have been impracticle for the user.

While the algorithm as proposed in [12] did not work for the 2-DoF problem, by

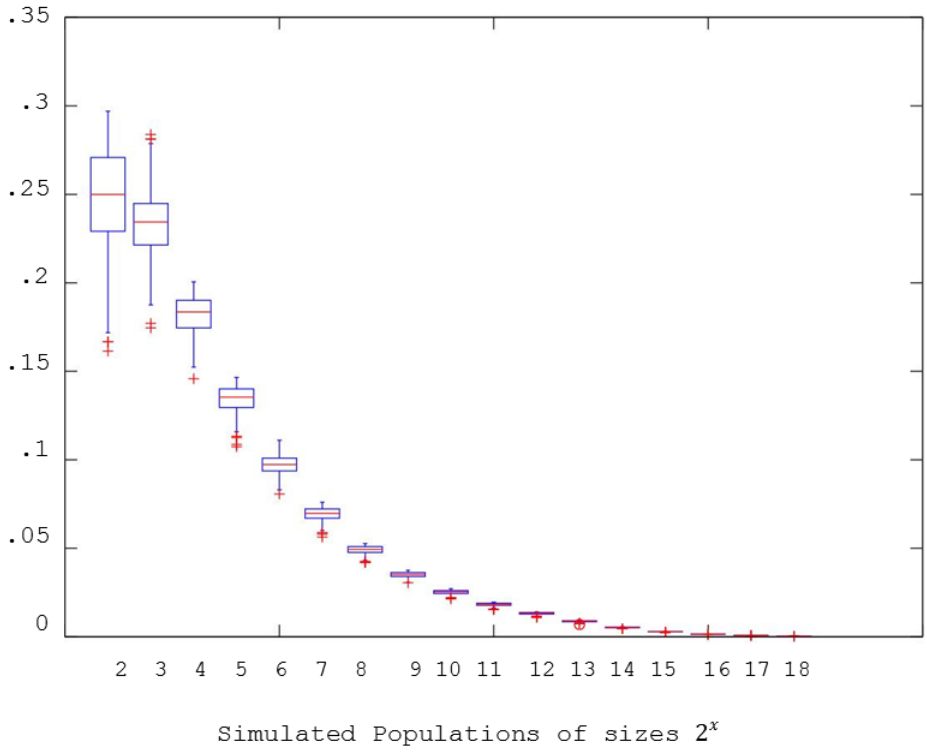

Figure 4.4: The Ending Percent of Population Converged for Experiments that did not Yield an Acceptable Solution

making modifications to the algorithm, the algorithm might be better able to solve this problem. These proposed modifications can be added without losing some of the benefits of the original algorithm. These modifications will be explored in the next chapter.

Conclusion and Future Work

This chapter begins with a review of the results in the previous chapter and attempts to extract insight into the benefits of hyperplane sampling. The second section of this chapterl contains an introduction into algorithm modifications that are intended to be looked into with future work, and the reasoning behind them.

5.1 Conclusion

Table 5.1: Comparison of cGA and MAV_MiniPop

	1-DoF		2-DoF	
	cGA	MAV_MiniPop	cGA	MAV_MiniPop
Yields Percent	90-95	near 100	0	near 100
Mean Times on Yield (minutes)	near 60	less than 10	N/A	near 60

This thesis explored the possibility of using the Compact Genetic Algorithm (cGA) for two primary reasons: the algorithm had already been shown to be a hardware efficient evolutionary algorithm implementation and it allowed for explicit recombination to be examined in the hover control problem for an insect-scale flapping-wing micro air vehicle (FW-MAV).

While the cGA algorithm managed to solve the one degree of freedom hover control problem, it was less efficient than the MAV_MiniPop in percentage of acceptable solutions found and time needed to find a solution. Table 5.1 shows a comparison of percentage

yield and time needed to find an acceptable solution for the MAV_MiniPop compared to the cGA. On the one degree of freedom problem, the MAV_MiniPop solution had near 100 percent yields [7] where the cGA algorithm only produced in the low ninty percent yields. Similarly, the amount of time needed by the MAV_MiniPop was on average less than 10 minutes [7] where the time needed for the cGA ranged from 44 to 53 minutes. When the two degrees of freedom problem is considered, the difference between the two algorithms becomes even more apparent. While the MAV_MiniPop yields percentages near 100% again and finds solutions in an average of 60 minutes [14], the cGA algorithm failed to ever produce an acceptable solution. The MAV_MiniPop clearly performs better than the cGA algorithm.

The cGA still has a benefit in storing the simulated population sizes by a bounding of log2 (n) rather than n. Also, stochastic hill climber algorithms that have small population sizes can have a problem with prematurely converging. The genetic algorithms implicit parallelism might help reduce the reliance on mutation events as a preventitive measure for early convergence.

From previous work [7] [14] [8], it has been observed that there are probably many acceptable solutions to the problem within the search space. At a minimum, the search space is reflected upon itself several times, since the average of eight lookup indices is used. The insignificance of the order of the lookup indices causes the search space to be reflected. If it is the case that there are many acceptable solutions, it could help explain why a stochastic hill climber algorithm outperformed the hyperplane sampler algorithm. Stocastic hill climbers in general converge more quickly than hyperplane samplers. In a situation with many quality locations, converging quickly on an acceptable location would not be unlikely. However, in the case of hyperplane sampling with slower convergence, especially in larger population sizes, the solution might continuously be pulled around the search space. In future work, modifications to the cGA algorithm will attempt to converge the simulated population more quickly to see if that improves yields and flight time to find

an acceptable solution.

Future work will attempt to better balance the observed exploration and exploitation strengths of cGA and MAV_MiniPOP respectively. Fortunately, the hardware cost of such a hybrid approach is still bounded by log2 n and is completely feasible in the time and space restricted realities we face with insect-scale flapping-wing micro air vehicles.

5.2 Future Work

5.2.1 Elitism Modification

The Elitism Adaptation is very similar to the cGA algorithm, with one change. Rather than generating two candidates each evaluation and making the two candidates compete, the algorithm only produces one candidate each evaluation, and then forces that candidate to compete against the champion. This adaptation will cause convergence to happen more quickly. As described in the conclusion, the reason this quicker convergence is desired is because it is believed that the search space contains many acceptable solutions. The Elitism Adaptation makes the cGA algorithm more closely mimic the MAV_MiniPop Algorithm's ability to converge much more quickly.

5.2.2 Island Modification

After seeing the results of a mutation only solution in MAV_MiniPop and the results of a recombination only solution in the cGA code, future work should test some balance between the two variation operations. The Island Adaptation that is proposed will use the cGA probability vector for storage as well as a similar update step. Rather than updating by 1/n where n is the population size, the Island Adaptation will update its individual island probability vector by 1/k, where k is the number of individuals on that island. The value of k can be calculated by dividing n by the number of islands, causing each island to have the

same number of individuals. By using this storage system, the algorithm keeps the hyper plane sampling by recombination. Also by keeping the representation of cGA, the memory reduction of m x log2 n is kept. However, since a probability vector is needed for each island the resulting memory needed is j x m x log2 k where j is the number of islands, m is the size of the bit string, and k is the number of individuals on a given island. The algorithm could also try the Elitism Adaptation described above, only generating one candidate and testing it against the local champion (the best solution found by that individual island).

Since islands are bring used in this variation, immigration from one island to another would be allowed. The immigration rate would be a parameter that would need to be tested at various settings. The proposed way of handling an immigration event is to have the global champion (the best solution found on any island) immigrate into the island, shift each probability in the islands probability vector to be twenty five percent more likely to produce the global champion. The large impact of the migration of the global champion can be important in pulling the population towards a good solution, especially if the island produced the global champion, but then underwent a hypermutation event that now has the island preforming poorly.

By using islands in this adaptation, a smaller population can be simulated on each island to make convergence easier without the large worry of early convergence. When one island prematurely converges, other islands are still able to hunt for a solution, so the entire algorithm is not stuck waiting on a rare hypermutation event to start the algorithm moving again.

In addition to adding islands, the Island Adaptation can also add a hypermutation event. The hypermutation event is another event that has a parameter set rate that would need to be tested at various settings. When a hypermutation event occurs, the island is essentially reset. The island's individual probability vector is reset to all fifty percent, and the local champion is reset to a random solution. If the local champion was also the global champion, that solution would still remain within the global champion; if an immigration

event occurred, that solution would be used to greatly alter the islands probability vector again.

In order to gain the benefits of both cGA as well as the MAV_MiniPop algorithms, the proposed Island Adaptation hybrid algorithm would use both mutation and recombination, as well as the probability vectors of the cGA. When run, these experiments would need to cover a range of mutation rates, immigration rates, and number of islands.

5.2.3 Consistent Manufacturing Error Modification

The Consistent Manufacturing Error Adaptation attempts to exploit the faults created during manufacturing. Whenever there is manufactoring fault due to manufacturing error, it is reasonable to assume that the impact to the vehicles will be consistent across the entire batch of FW-MAV. For example, an error in the mold used, would produce an identical type of damage in each vehicle produced by the faulty mold. This adaptation attempts to use this consistency to speed up the solution optimization.

Rather than storing the damages as a single type of error that is randomly generated to represent a combination of manufacturing faults and sustained damage, the Consistent Manufacturing Error Adaptation will store two types of error. The first will be a randomly generated error for sustained damage, and the second will be an error that is applied to all vehicles in the current experiment. Each experiment will try to modify multiple vehicles at once, in order to find acceptable solutions for each vehicle. However, while each vehicle will be running an algorithm independently, occasionally the vehicles will use wireless communication to pass the individual vehicle's best solution to the other vehicles. The passed solution would then compete with the vehicle's champion on the next evaluation, and if the passed solution is better, it will then replace the vehicle champion. After that the vehicle will continue with its independent algorithm until another communication event occurs. When an algorithm passes its champion out to the other vehicles, its independent algorithm is in no way affected, other than the slight pause taken to send the information

out.

Since part of the damage on one vehicle is identical to part of the damage on another vehicle, it is not unreasonable to assume that a solution that compensates for all the damage on one vehicle is likely to compensate for at least some of the damage on another vehicle, making the Consistent Manufacturing Error Adaptation a reasonable adaptation to test in the future. This adaptation does not rely on a specific algorithm being used on the vehicle. If this adaptation works well, it can be updated with the best independent algorithm known to date. Parameters that would need to be tested at various values throughout the experiments are the rate of communication events and the number of vehicles in each experiment. Preliminary testing of the Consistent Manufacturing Error Adaptation using the Elitism Adaptation modification has shown promising results.

Bibliography

[1] *A hardware implementation of the compact genetic algorithm*, In Proceedings of the 2001 Congress of Evolutionary Computation, 2001.

[2] J. E. Smith A. E. Eiben. *Introduction to Evolutionary Computing*. Springer, 2007.

[3] T. Back, U. Hammel, and H.-P. Schwefel. Evolutionary computation: comments on the history and current state. *Evolutionary Computation, IEEE Transactions on*, 1(1):3 –17, apr 1997.

[4] Bolender M.A. D.B. Doman, M.W. Oppenheimer and D.O. Sigthorson. Altitude control of a single degree of freedom flapping wing micro air vehicle. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, August 2009.

[5] M.W. Oppenheimer D.B. Doman and D.O. Sigthorson. Dynamics and control of a minimally actuated biomimetic vehicle: part i - aerodynamic model. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, August 2009.

[6] D. Fogel. *System Identification Through Simulated Evolution: A Machine Learning Approach to Modeling*. Ginn Press, 1991.

[7] J. Gallagher, D. Doman, and M. Oppenheimer. The technology of the gaps: An evolvable hardware synthesized oscillator for the control of a flapping-wing micro air vehicle. *Evolutionary Computation, IEEE Transactions on*, PP(99):1, 2012.

[8] J.C. Gallagher and M.W. Oppenheimer. Cross-layer learning in an evolvable oscillator for in-flight controller adaptation in a flapping-wing micro air vehicle. In *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, pages 1547 –1551, nov. 2011.

[9] J.C. Gallagher, S. Vigraham, and G. Kramer. A family of compact genetic algorithms for intrinsic evolvable hardware. *Evolutionary Computation, IEEE Transactions on*, 8(2):111 – 126, april 2004.

[10] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[11] G. Greenwood and A. Tyrrell. *Introduction to Evolvable Hardware: A Practical Guide for Designing Self-Adaptive Systems*. IEEE Press, 2005.

[12] G.R. Harik, F.G. Lobo, and D.E. Goldberg. The compact genetic algorithm. *Evolutionary Computation, IEEE Transactions on*, 3(4):287 –297, nov 1999.

[13] D.B. Doman J.C. Gallagher and M.W. Oppenheimer. Practical in-flight altitude controller learning in a flapping-wing micro air vehicle. submitted.

[14] M.E. Kijowski, J.C. Gallagher, and L.D. Merkle. Improved learning in an evolvable hardware hover controller for an insect scale flapping-wing micro air vehicle. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 426 –431, june 2011.

[15] D.B. Doman M.W. Oppenheimer and D.O Sigthorson. Dynamics and control of a minimally actuated biomimetic vehicle: part ii - control. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, August 2009.

[16] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994. 10.1007/BF00175354.

[17] R.J. Wood. The first takeoff of a biologically-inspired at-scale robotic insect. *IEEE Trans. on Robotics*, 24(11):341–347, 2008.