Self-learning locomotive motion applied on a four-legged robot using genetic algorithms

KIM FOGELESTRÖM
VIKTOR LAHTI
Self-learning locomotive motion applied on a four-legged robot using genetic algorithms

KIM FOGELSTRÖM
VIKTOR LAHTI

Bachelor’s Thesis at ITM
Supervisor: Damir Nesic
Examiner: Nihad Subasic

MMK 2017:28 MDAB 646
Abstract

The goal of this project was to examine and explain how genetic algorithms could be implemented and optimised on a cheap electronic hardware. The goal was that a 4-legged demonstrator to learn to walk by itself using only knowledge of the basic movements of its limbs. The report explains in detail how a basic genetic algorithm is constructed and what alterations can be made to fit the demonstrator in question. It also describes how the demonstrator with the help of a ultrasonic sensor measuring the distance, evaluates its own performance for every iteration and uses this information to improve. For the experiments, fractional factorial tests are performed and the demonstrator learns the walking movement by trying a total of 4500 times in 36 different tests. The scope of this project limits the robot to walking in a straight line on even ground and with a flat surface in front of a vertical surface for ease of distance measurement.
Sammanfattning

Självlärande robot

Målet med detta projekt var att undersöka och förklara hur genetiska algoritmer kan implementeras och optimeras på billig hårdvara för att tillåta en 4-bent robot att lära sig själv att gå med endast vetskapen om basrörelsen av sina ben. Rapporten går in på detalj hur en genetisk algoritm är uppbyggd och vilka förbättringar som kan göras för roboten i fråga. Den beskriver också hur roboten med hjälp av en ultraljudssensor för att mäta avstånd evaluerar sin prestation efter varje iteration och använder denna data för att förbättra sig själv. Experimenten utfördes enligt fractional factorial metoden och roboten lär sig att gå genom att testa totalt 4500 gånger i 36 olika test. Ramen av detta projekt tillåter endast roboten att gå i en rak linje på platt mark och ha tillgång till en plan vägg framför sig för att lätt kunna mäta avstånd.
Acknowledgements

Firstly we would like to thank Damir Nesic for being supportive and giving us feedback, helping us to row this project ashore. Thanks to all the mechatronics supervisors for giving advice and being available all throughout the duration of the project. Thanks to Nihad Subasic for organising lectures and helping our project get started by giving advice on all aspects of robot building.
4 Results

5 Discussion
  5.1 Review of results ........................................ 23
  5.2 Genetic algorithm limitations ........................... 24
  5.3 Testing errors ............................................ 24
  5.3.1 Compact genetic algorithms .......................... 25
  5.4 Conclusions ............................................... 25
  5.5 Future Work ............................................... 26

Bibliography .................................................... 27

Appendices ....................................................... 29

A Experiment setup ............................................ 29

B Electronic schematic ........................................ 31

C Demonstrator Code .......................................... 33
  C.1 GA.py .................................................... 33
  C.2 Robot_main.py ............................................ 41
  C.3 Distance.py .............................................. 43
Nomenclature

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>The distance between the middle of the transceiver and the middle of the receiver on the ultra sonic sensor.</td>
</tr>
<tr>
<td>B</td>
<td>The distance registered by the ultrasonic sensor.</td>
</tr>
<tr>
<td>C</td>
<td>The distance straight to the wall from the robot.</td>
</tr>
<tr>
<td>$N_u$</td>
<td>The size of the tournament in tournament selection.</td>
</tr>
<tr>
<td>$N_{\text{result}}$</td>
<td>The result from each of the tests where the parameter is set to its low value.</td>
</tr>
<tr>
<td>$P_1$</td>
<td>The effect of 1 parameter on the test result.</td>
</tr>
<tr>
<td>$P_{\text{result}}$</td>
<td>The result from each of the tests where the parameter was set to its high value.</td>
</tr>
<tr>
<td>$c$</td>
<td>The speed of sound.</td>
</tr>
<tr>
<td>$n$</td>
<td>The population size used in Compact Genetic Algorithms.</td>
</tr>
<tr>
<td>$p_v$</td>
<td>The probability vector used in Compact Genetic Algorithms.</td>
</tr>
<tr>
<td>$t$</td>
<td>The time from when the signal is sent from the transceiver of the ultra sonic sensor to the moment when the signal is received.</td>
</tr>
</tbody>
</table>
# Abbreviations

<table>
<thead>
<tr>
<th>Acronyms</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cGA</td>
<td>Compact Genetic Algorithm.</td>
</tr>
<tr>
<td>FF</td>
<td>Fractional Factorial.</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm.</td>
</tr>
<tr>
<td>MT</td>
<td>Mersenne Twister.</td>
</tr>
<tr>
<td>PV</td>
<td>Probability Vector.</td>
</tr>
<tr>
<td>PWM</td>
<td>Pulse Width Modulation.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The first section in the introduction is the background and introduces the project motivation. The second section is about the research questions that this thesis set out to answer. The questions are answered through literature studies and Fractional Factorial (FF) experiments. The third section is presents the scope of the project, what restrictions are put on the project to make it more feasible for a bachelor’s thesis. The fourth and final section is the method section. This section explains how the project is performed and what experiments will be conducted.

1.1 Background

Implementation of smarter technology in our society has the possibility to affect and improve our daily life. Take for example the possibility of autonomous driving [1]. Because of the impossibility to program every possible scenario or for every scenario predict the environment an autonomous system may encounter, advanced systems need to be able to adapt by themselves - the robot needs to be able to learn without human supervision [2]. This is the primary goal of evolutionary robotics. The field of evolutionary robotics is still very young and so far much of the research being done is on easy tasks that can be easier achieved with simple programming. But with constantly improving technology it is easy to envision a future where robots can predict and adapt to changes in their environment that were impossible to pre-program.

Self learning in machines can be viewed as the future of robots and automation. It can be all from self driving cars that slowly learns how to best drive the road they frequent and share this knowledge with other cars [1], or designing an antenna that gets the best reception (see Figure 1.1). The operations these machines perform were not exactly specified in code. Rather a series of guidelines were programmed for the robots to follow and the robots then test these out and act according to the best available option in every situation. There are several self learning machine algorithms, one of them being the Genetic Algorithm (GA).
CHAPTER 1. INTRODUCTION

1.2 Purpose

The purpose of this project is to examine how a servo motor command that enables a 4-legged robot to walk can be learned using a genetic algorithm implemented on an inexpensive computing platform. The details which the project aims to evaluate can be formulated into the following research questions:

- What kind of fitness criteria and initial population must be constructed to make a feasible walking movement within 10 generations of GA.

- What implementation of the population size, crossover probability, elitism and mutation probability gives the furthest possible distance travelled by the demonstrator.

1.3 Scope

In real world scenarios a robot could face many different surfaces and weather conditions, but for the purpose of this project the demonstrator will be constricted to:
1.4. METHOD

- Having eight servo motors, two for each leg. for each new iteration the servo motors will be reset to a neutral staring position. This to somewhat limit the walking motions to ensure a better result with a smaller number of iterations.

- Walking in a straight line on a flat surface in a controlled environment.

- Having access to a flat wall straight ahead of it in range of the ultrasonic sensor for ease of distance measurement.

- Genetic algorithms out of the field that is evolutionary robotics, using the tournament selection with $N_u$, a set number of randomly chosen individuals from the population set to 2, two point crossover method and the worst member replacement method.

- The algorithm will run on a Raspberry Pi 3 model B with a 1.2 Ghz quadcore processor and 1Gb of RAM.

- Each individual in the GA consists of 3 movements per servo.

1.4 Method

Genetic algorithms have the ability to make very complex movement possible, such as the locomotive movement of a snake [4]. For the scope and length of this project, a bipedal construction was considered but also eventually dropped. While fully possible, a simple multi-legged demonstrator design was chosen instead, see chapter 3 for more information about our demonstrator. A simple demonstrator design would allow us more time to focus on the programming and testing of the robot.

The hardware should be cheap, widely available and have a large community. This increases the likelihood of finding good code examples and preexisting libraries. The hardware should also enable eight servo motors to be run. For distance measure a sensor is required. It must be able to measure the distance from the demonstrator to a flat surface directly in front of it.

1.4.1 Experimental Design

The goal of the experiment is with the help of the demonstrator presented in chapter 3, to answer the research questions from section 1.3. From the research in chapter 2 the GA presents different parameters that changes the behaviour and therefore the result.

According to the research the selection process depends on the choice of the population size, crossover probability, elitism and mutation probability. And therefore our hypothesis is:
CHAPTER 1. INTRODUCTION

– If the population size is increased the whole process will improve. An decrease in crossover probability and an increase elitism means a higher converging rate. The higher mutation probability, the bigger diversity but results in lower converging rate.

For a detailed description of how the experiment was performed, see appendix A. The control variables are the starting point of the experiment and the values are set as following. The population size is set to 10, the probability for crossover is set to 80 %, elitism is set to 10% and the chance for mutation is set to 30%. For a detailed explanation on how these values are implemented and what they represent the experiment see chapter 3

To evaluate if the demonstrator follows the stated hypotheses, FF experiment design with Resolution IV \(2^4-1\) is applied[5]. The FF works by starting out with four test parameters that impact the experimental result. To determine how each parameter changes the final result of the experiment, tests with these parameters at either a low or a high value is conducted. The testing is only complete once every parameter value in combination with each other has been preformed. This results in eight iterations of the experiment being performed. Because of the structure the four parameters can be reduced to three. This is done with each parameter getting a value of either minus one or one depending on if they in that iteration hold their high value, represented by a one or their low value, represented by a minus one. The three first numbers are then multiplied into the fourth value for example:

\[1 = -1 \times -1 \times 1\] (1.1)

The fourth parameter would in this case attain its high value(1).

When all the tests have been preformed, the contribution of each parameter is calculated by the formula:

\[P_1 = \frac{\sum_{i=1}^{4}(-Nresult_i) + \sum_{i=1}^{4}(Presult_i)}{4}\] (1.2)

Where \(Nresult\) is the result from each of the tests where the parameter is set to its low value, \(Presult\) is the result from each of the tests where the parameter was set to its high value and \(P_1\) is the effect of one parameter. Due to this eight iterations of the experiment is conducted in the test environment defined in appendix A. Each test iteration will be performed three times to get a more reliable result. The result of the experiment is defined as the best individual, the individual that managed to walk the furthest distance. After the initial testing the most optimal parameters found by the Resolution IV \(2^{4-1}\) is set to the lowest test parameters in a new round of testing. The testing is done with the same step size as for the initial test. This to see if improvements in the GA works on a linear scale. The new tests are only repeated twice. This leads to 24 tests conducted with different parameters for the first test and 16 for the last.
Chapter 2

Theory

In the theory section, details of each phase of the genetic algorithm is explained. The alternative version of GA called Compact Genetic Algorithm (cGA), its structure and how it differs from regular GA are looked into. Lastly the general theory of pseudo random generators and the random generator that the Python programming language uses Mersenne Twister (MT) is explained.

2.1 Genetic algorithms

The basic idea behind GA is to mimic the Darwinian view of evolution to solve optimisation problems [6]. The goal with an optimisation problem is to find the maximum or minimum for a specified criteria. For example a worker wants the shortest travelling time home from work. The shorter time it takes to get home, the better. Hence, his cost is time and he wants the lowest cost.

The GA uses genetics as a model to solve these kind of problems. For a problem with a finite number of possible solutions, GA is one way of finding the solution that gives the lowest cost [7]. These finite number of solutions is in GA’s terminology called Individuals or Chromosomes. Each of these Individuals consist of controllable parameters whose values influences the solution. These parameters are called Genes and the GA’s goal is to find the values of these genes that gives the optimal solution. To compare how good the solution of different individuals is to each other, a fitness is introduced. The fitness is a comparable measure of how good the solution is. The higher fitness, the better solution [7].

To start the GA an initial population needs to be created. A bigger size and gene diversity greatly improves the process. In most cases the initial population is chosen according to [7], were the controllable parameters are randomly generated. The initial population also needs to be evaluated to receive its fitness. Continuing on the previous example, the initial population is randomly created ways for our worker to get home, e.g. walk half the distance and then take a helicopter for the rest of the way. The repeating breeding process of the GA is described in the
following subsections.

2.1.1 Selection

Selection is the process where chromosomes are chosen as parents to create a new generation of individuals, with the goal to create individuals with an even higher fitness than their parents. According to Darwin’s theory the fittest individual has the biggest chance to survive and create a new offspring [7]. The GA’s selection should mimic the same principle by randomly choosing the individuals according to the probability created by the fitness. If the more fitter individuals are strongly favoured in the selection the higher convergence to a solution. This is called a higher selection pressure. However, too high fitness criteria can lead to a sub-optimal result [7].

Tournament Selection is one method for selecting the parents. It works by holding a tournament between a set number, \( N_u \) of randomly chosen individuals from the population and the winner of the tournament is the one with the highest fitness. The Tournament Selection is repeated the same amount of times as the number of individuals in the population [7]. For our example worker, this means that maybe the optimal time-saver would be to combine two methods of transportation. For instance maybe first walk a path through the woods and then ride a bicycle the rest of the way home.

2.1.2 Reproduction

Selection is followed by reproduction. Reproduction is usually divided into two steps, crossover or recombination and mutation.

**Crossover** The process where a new individual, called in GAs a child, is created from taking two parent chromosomes, with the goal of creating better offspring, is called the crossover. The process of crossover is described in three steps:

1. A random pair of individuals are selected for mating.
2. Two cross points are selected at random in the chromosome.
3. The part of the chromosome between the points is switched, illustrated in Figure 2.1 by the individuals, hence two new individuals are created.

This particular variant above is called a Two Point Crossover. The advantages with this variant compared to others, e.g. Single Point Crossover and Uniform Crossover), is that a local group of good genes in the chromosome has a high chance of staying together. Genes with dependant characteristics should therefore be placed close together. This variant also avoids the issue that the Single Point Crossover suffers from where the tail and head from on of the chromosomes will always split between the children [7]. The before mentioned example using a worker to get home this means, combining two methods to get home. If one way is to walk
2.1. GENETIC ALGORITHMS

![Two Point Crossover diagram]

**Figure 2.1.** Illustration of how the *Two Point Crossover* works, made in *Adobe Illustrator*.

the whole way home and another is to bicycle, the new individuals will be walk-bicycle-walk and bicycle-walk-bicycle.

**Mutation** The goal of mutation is to maintain genetic diversity and prevent the population to become stuck in a local maximum. This is achieved by randomly modifying some of the chromosomes genes. The *simple mutation* method works by having a predefined probability that each gene in the chromosome will experience mutation. A high mutation leads to a lower converging rate [7]. For the worker wanting the quickest way home, this would be randomly changing a small part of his selected way home. For example maybe the last 30 meters to his house he makes the choice to run, or crawl. The mutation is completely random.

2.1.3 Evaluation

After the process of reproduction a new population is obtained. In the new population, each *individual* have to be evaluated by how good the solution is. This evaluation should be compared to a specified criteria, which reflects the desired result of the GA. The criteria combined with the solution from the *individual* creates the *fitness*. The *fitness* needs to take in to consideration how good the solution is and also how close it is to the optimal solution, hence the combination of criteria and solution. The worker would for example have to at the end of the week pick which of the means of transportation took the least amount of time. Its important for him not to just pick the fastest route and be done with it. Because how can he be sure that after the next week of combination, another faster way has not emerged.

2.1.4 Replacement

The last step of GA’s breeding process is the replacement. There are now two children for each pair of parents. This stage chooses which of these individuals should be replaced by new solutions created in the 2.1.2 process. If the new individuals are not better then the ones who already exists, there is no need for replacement. Since the example worker only has five days of work in a week. The newly created
means of transport home would be kept if they took a shorter time then those he already has in his weekly routine. There are two main methods for replacement:

1. Generational update schemes, which usually replaces all the parents with their children. The new generation consists only of children. The drawback with this method is that if the parents had a high fitness and is optimised, the replacing children will have divided the good traits which leads to a population of less fit individuals.

2. Steady state update, it consist of combining the parents and children into a population. But the addition of a new individual means that another individual needs to be replaced. One of the ways to do this is with tournament replacement. Tournament replacement puts two individuals against each other and the one with the highest fitness rating gets to replace the other one.[7]

2.2 Compact Genetic Algorithm

The basic principles of GA operation implies that there is a risk that after a number of iterations it reach a storage limit which depends on the hardware that is used to run the algorithm. This can be solved with cGA [8]. The native cGA only handles bits instead of whole data sets, e.g. integers, in its individuals. The cGA starts by creating a Probability Vector (PV) for each individual’s genes and sets all of its values to 0.5. It then generates two individuals with either a 0 or 1 on each gene, the PV tells the probability of it being a 1. For example, for the second gene in the individuals, the probability of it being a 1 is defined by the second value of the PV. The one with the highest fitness value wins. The next step is to update the PV with the winning individual:

If the first gene is a 1, then the first value in the PV becomes,

\[ p_v(1) = p_v(1) + 1/n. \] (2.1)

Where \( n \) is the population size and \( p_v \) is the PV. If the gene instead is a 0, then the it subtracts the \( 1/n \). This process continues until all values in the vector has been evaluated, and is shown in Algorithm 1 which is taken from [9].
2.3. ELITISM IN GA

Algorithm 1: Compact genetic algorithm psuedo code, structure made by [9].

Data: cGA
initialization counter t = 0;
initialization;
for i = 1:n do do
    Initialize PV[i] = 0.5;
end

while budget condition do
    generate 2 individuals a b by means of PV;
    [winner,loser] = compete(a,b);
    ** PV Update**;
    for i = 1:n do
        if winner[i] = loser[i] then
            if winner[i] = 1 then
                PV[i] = PV[i] + 1/Np
            else
                PV[i] = PV[i] - 1/Np
            end
        end
    end
    Counter update;
end

To exploit the advantage of cGA when real values are used the Real-Valued Compact Genetic Algorithm is applied. This method works in a similar fashion as cGA but the PV is instead a nx2 matrix. The matrix contains the mean and standard deviation of a Gaussian probability distribution function.[8]

2.3 Elitism in GA

To make sure the GA is constantly converging to a better optimisation solution, help in the form of elitism can be applied. Elitism works by favouring the individuals with very high fitness [7]. If an individual turns out to be an almost ideal solution to the problem, it would only harm this individual to dilute it by mutating and mixing it with other individuals. Leading to a worsened population. Elitism solves this problem by keeping the individual intact and carries it forward to the next generation. A drawback of elitism is that the algorithm can converge to a local maximum. This means that the GA outputs that it has reached the optimal solution but a better solution lies elsewhere. This is in turn solved by a bigger difference in the population to get as a data spread as big as possible. There are established elitism for both GA and cGA.[10]
2.4 Random number generation

The purpose of a random number generator is to generate a random number. True randomness cannot be generated by a coded generator. The pseudo random generators can be seen as simulated randomness, because it creates a number that is close to random. Almost all generators in this category uses an initial numerical value to start the generation, a seed. This seed is then manipulated iteratively, to create the pseudo random number [11]. The Mersenne Twister (MT) used by Python as its core generator[12] is a pseudo random number generator, introduced by [13]. It uses a Mersenne prime for the seed. The MT is one of the most widely used random number generator, with a long period of $2^{19937} - 1$ [14]. A period is defined as the interval between a repetition of a number in the sequence. This period is the longest of any pseudo random number generator.[13]
Chapter 3

Demonstrator

In the demonstrator chapter the construction of the robot is described, what parts are used and how they are put together. The overall principles that underline the software and electronics are also presented.

Figure 3.1. A picture of the demonstrator used in the experiment.
CHAPTER 3. DEMONSTRATOR

3.1 Hardware

The main hardware of the demonstrator is the Raspberry Pi, which fetches data from an ultrasonic sensor and sends output to eight MG90S servos. A more detailed description of the components used is described in the subsections below.

In Appendix B, Figure B.1 shows the electronic schematic used on the demonstrator. The demonstrator is powered by a net adapter connected to the power grid.

The base model of the demonstrator is constructed in the 3D-Cad software Solid Edge ST8 as seen Figure 3.2). The legs and the body is laser cut out of acrylic plastic, a sturdy but light material chosen for its stability and ability to withstand the torque of the servo motors. The first servo motor is attached to the body with a screw. To keep it stable and to enable two joint movement on each leg the two servo motors are glued together with epoxy glue. The leg is then attached with a screw to the second servo. For better friction against the surfaces the end of the legs have spikes, as seen in Figure 3.2. The Raspberry Pi is fitted on top of the body aligned with the screw holes and fastened in place, the Adafruit top hat is installed on top of the Raspberry Pi and the ultrasonic sensor is soldered to the preexisting breadboard surface of the Adafruit top hat.

![Figure 3.2. The CAD sketch of the body and legs, all measures in mm. The sketch was made in Solid Edge ST8.](image)

3.1.1 Raspberry Pi 3

The Raspberry Pi 3 is a microcontroller with a powerful processing unit. With a 1.2GHz 64-bit quad-core ARMv8 processor it is capable of running very complicated software but still being small enough to be easily integrated into project design, perfect for robotic projects. The Raspberry Pi 3 is driven by a operative system
3.1. HARDWARE

called raspian, a basic system that enables easy access to the Raspberry Pi:s file library.[15]

3.1.2 Servo Motors

A servo motor is an easy to use and cheap solution for enabling movement in robotics projects. The servo motor uses Pulse Width Modulation (PWM) signals to turn from 90°to -90° and can be programmed to reset to the same location after use. The position of the servo motor can easily be altered and is perfect for using as a leg module because of its inability to fully rotate 360°. The servo motor Mg90s has a duty cycle of 1-2 ms and a PWM period of 20 ms, where 1 ms is -90° and 2 ms is fully 90°. The full cycle can be seen in Figure 3.3.

![Figure 3.3. The PWM period and duty cycle of our chosen servo MG90s created by [16].](image)

3.1.3 Ultrasonic Sensor

The Ultrasonic sensor is a instrument to measure the distance to a object. It works by having one transmitter module that outputs a high frequency signal and one receiver module that registers the signal. The time between sending the signal and receiving it is measured and the distance from the target can be calculated through the equation where \( c \) is the speed of sound and \( t \) is the signal time:

\[
distance = \frac{(t \times c)}{2}
\]  

The division by two comes from the time it takes for the sound to travel to the object and back to the receiver. The object must be aligned straight in from the sensor. If the object is uneven or turned, the signal may bounce of the object colliding with another object elsewhere in the room further away then back to the object and then to the receiver. This will give a false interpretation of the distance. Our chosen model HC-SR04 has a measure distance of 2-400 cm and sends out 8 40 kHz signals for each distance measure. The HC-SR04 has 4 different ports; 5V power supply, GND, Echo pulse output and Trigger pulse input. When the trigger
pulse input receives a short 10uS pulse it starts the ranging, sending out eight cycle burst pulses of 40 Khz sound.[17] The ultrasonic sensor faces a problem at closer distance measurement, the distance between the transmitter module and the receiver module causes the signal to bend slightly at an angle and this will make the distance measurement inaccurate. To combat this behaviour the Pythagorean theorem can be used.

### 3.1.4 Adafruit PWM-hat

The Adafruit 16-channel PWM top hat functions as a breadboard for extending ports from the Raspberry Pi 3. As stated the Raspberry Pi 3 has only one PWM port and if more is needed the Adafruit can be added. It also has a breadboard surface so that you can solder whatever component you need to use on top of it. The Adafruit top hat board goes on top of the Raspberry Pi with a cobbler connection.[18] The Adafruit top hat is mounted on the raspberry as seen in figure 3.4.

![Figure 3.4. Adafruit PWM top hat mounted on a Raspberry Pi.](image)

### 3.2 Software

The demonstrator starts its movement by resetting the position by firstly turning the legs up in the air. This makes the demonstrator hang by the strings, as described in appendix A and makes the demonstrator point straight towards the reference wall. Then the demonstrator moves it legs to standing up straight relative to the ground. The demonstrator measures the initial distance from the wall, starts the movement sequence with the input from the GA. The input consist of a vector, which contains the three positions each servo will rotate to during the walking movement. Hence, the total number of values in the vector is 24 as defined in appendix A. The vector and its corresponding servo is illustrated in Figure 3.5.
3.2. SOFTWARE

Figure 3.5. Illustration of which position in the vector corresponds to what servo for the demonstrator, made in Adobe Illustrator.

The last step in the demonstrator’s execution it resets itself again to standing and measures the moved distance. The difference between the two distances is then returned to the software for the GA. The flowchart that the demonstrator executes for each test is illustrated in Figure 3.6.

Figure 3.6. The flowchart of the demonstrator software, created in [19].
3.2.1 Software for the Genetic Algorithm

To see the whole GA’s software, see appendix C. The main shell of the software is written by [20], with the modifications:

- The one point selection is replaced by the Two Point Crossover method to fit the scope of the project.
- The function that updates fitness is changed to gather data from the demonstrators ultra sonic sensor and to send data to the servos.
- An algorithm to write each generation’s population to a .csv documents is added to save the experiment data.
- One extra state to the initial population stage is created, this state’s goal is to continue a previously cancelled test instead of having to restart from the beginning each time something goes wrong while testing.

The main stages of the GA is described in section 2.1. Before the software is executed, the input parameters; population size, crossover probability, elitism and mutation probability are chosen. The population size is a positive integer and states the number of individuals in each generation. The crossover probability and mutation probability is a probability of the occurrence of crossover and mutation respectively. The elitism decides what percentile of the fittest individual passes over instantly to the next generation.

Each individual consists of a vector with initially random generated values. The random numbers are generated with MT, which is described in subsection 2.4. This vector is in GA’s terminology the chromosome and the values in it are the genes. The vector contains the 24 servo positions described in section 3.2. With values in the vector which approximately corresponds to -35° and 35°. This range is defined by how big of a movement the legs can make without hitting the body.

The main loop of the GA software starts by a tournament selection between three individuals to choose which individuals should mate. This is repeated the same amount of times as the population size, with the probability of it happening set by the crossover probability. Then the Two Point Selection method crosses the two chosen vectors to form two children. If the mutation probability evaluates to true, a random value in the child’s vector is mutated between the minimum and maximum value presented in the paragraph above. The newly created vectors are sent to the demonstrator, it then returns the distance travelled. The fitness is set to the absolute value of the difference between the gathered data and a desired walking distance. The closer the gathered distance is to the desired, the better fitness achieved. This walking distance is set to the value of 20 centimetres. The particular distance was chosen because of the nonexistent risk of the demonstrator walking that distance within the scope of three movements per servo. Therefore the
3.2. SOFTWARE

difficulty with the lowering of fitness that arises if the demonstrator walks past the desired distance is avoided. The last step of the loop is to sort the vector with the highest fitness or lowest value, and then a *Steady state update* with *worst member replacement* takes place to keep the population size the same.
Chapter 4

Results

The result from the experiment setup described in the section 1.4.1 is presented below. The control variables was change in the initial test to the following. The population size to 8 or 12, the probability for crossover to 75% or 85%, the chance for elitism to occur to 5% to 15% and the chance for mutation to occur from 25% to 35%. Where the low values represents the (-1) and the high values represent the (1). The dependent variable is the distance moved. The total amount of individuals/walking sequences the demonstrator made was approximately 4500 and the total testing time was 21 hours. The first phase in the testing is presented in table 4.1.

Table 4.1. Table of the first test using FF.

<table>
<thead>
<tr>
<th>Population size</th>
<th>Crossover</th>
<th>Elitism</th>
<th>Mutation</th>
<th>Best individual</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>5.33</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>6.67</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>6.33</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>8</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>7.85</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6.67</td>
</tr>
</tbody>
</table>

1.20   -0.37   0.96   -0.46   Effects

Figure 4.1 shows the mean value of whole population for each generation, with the data from the first test. Instead of using the whole population, Figure 4.2 shows the best individual in each generation from the first test. Both mentioned figures shows an improvement in distance moved between the initial population and generation 10 for all of the eight tests, initial population is generation 0 in the figures. The best individual from both figures is when the parameters are the high value for both the population size(1) and the elitism(1), and the low value for the crossover(-1) and mutation(-1).
CHAPTER 4. RESULTS

Figure 4.1. Graph over the first FF test with the mean of the three repetition and the mean of every individual in generations population. Each line of the legend represent the parameters, population size, crossover, elitism and mutation. Where the 1 and -1 represent the high value and the low value, described in 1.4.1. Graph made in Matlab.

Figure 4.2. Graph over the first FF test with the mean of the three repetitions with the best values for each generation. Each line of the legend represent the parameters, population size, crossover, elitism and mutation. Where the 1 and -1 represent the high value and the low value, described in 1.4.1. Graph made in Matlab.
For second round of testing the probability for crossover was changed to 65% or 75%, the chance for elitism to occur to 15% or 25% and the chance for mutation to occur to 15% or 25%. Where the low values represents the (-1) and the high values represent the (1). Table 4.2 presents the data from the second test, using the FF experiment design.

Table 4.2. Table of the second test using FF.

<table>
<thead>
<tr>
<th>Crossover</th>
<th>Elitism</th>
<th>Mutation</th>
<th>Best individual</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>5.33</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>8</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>6.67</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>6.33</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>7.85</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6.67</td>
</tr>
</tbody>
</table>

0.63 -0.13 1.88 Effects

Figure 4.3 and Figure 4.4 shows the same data type as in the first test but for the second test. Both figures shows the same improvement as the figures from the first round of tests, where all eight test improves on the walked distance from the initial population to generation 10. The best individual from both figures is when the parameters are the high value for both the crossover(1) and mutation(1) and the low value for the elitism(-1).
CHAPTER 4. RESULTS

Figure 4.3. Graph over the second FF test with the mean of the three repetition and the mean of every individual in generations population. Each line of the legend represent the parameters, population size, crossover, elitism and mutation. Where the 1 and -1 represent the high value and the low value, described in 1.4.1. Graph made in Matlab.

Figure 4.4. Graph over the second FF test with the mean of the three repetitions with the best values for each generation. Each line of the legend represent the parameters, population size, crossover, elitism and mutation. Where the 1 and -1 represent the high value and the low value, described in 1.4.1. Graph made in Matlab.
Chapter 5

Discussion

From the results of the experiments it can be concluded that the demonstrator was able to learn and improve using GAs. However the distance travelled and the time it took could be improved. In this chapter it’s explained what factors could impede and change the values of our parameters during the experiments.

5.1 Review of results

From the first round of tests, a bigger population size had the biggest positive effect on the result. This follows both the hypothesis, the FF showed that a lower crossover probability, a higher elitism ratio and a lower mutation gave a better result. This follows the hypothesis by proving a faster converging result. The Figure 4.2 shows how two lines from an early generation differentiates from the other testings, these two both contains the bigger population size, a lower crossover probability, which further proves the point. Elitism has the next highest effect on the result. Crossover probability and mutation has almost the same effect.

The second round of tests keeps the population size constant, because it would have increased the duration of the test. It also already showed in the first round of tests it was clear that a bigger population gave a better result. however in the second round, a higher crossover probability and mutation gave the better result, and a lower elitism gave a relatively low but positive effect on the result.

The second round of testing goes entirely against the first round. Maybe because of the lowered diversity a better result couldn’t be found. The reason for this can also be from the smaller testing size, where two repeating test was conducted, for the first round three repetitions were made. Comparing 4.2 and 4.4 they look almost the same but comparing the parameter combinations it seems flipped. The parameters set with the longest walking distance of the best individuals performed the lowest walking distance in the second round.
CHAPTER 5. DISCUSSION

With the result from the first round of testing we can say that our hypothesis is accepted. However, after continual experimental testing, the legs of our demonstrator started to wear down. This was caused by the constant chafing against the rough floor of our test rig. The lowered amount of repetitions may also made the second round of testing less reliable.

5.2 Genetic algorithm limitations

The random generation of these genes can impact if the robot is fast to find a solution. If the combination of the genes are far from a feasible movement pattern, finding this pattern will take longer then if the genes started closer to the optimal solution. The GA is a slow moving algorithm. For each new generation a larger time to execution is needed, this hinders testing of the algorithm.

Another consideration is that at the start of the algorithm it is pre-programmed which servo is actuated. This limits the available movements and makes it easier for the algorithm to work. If the servo and the individual were both random at all times the GA be hard pressed to converge to a better solution. The best case would be a movement pattern that worked on all servos. Otherwise the algorithm would never know where a specific servo was and which movement pattern worked best with that servo.

5.3 Testing errors

Due to our testing setup the robot sometimes did not reset itself entirely straight, this leads to the next test starting slightly off kilter. When this phenomenon occurred, the robot was manually reset to save the test data.

The ultrasonic sensor only measures forward and in centimetres. Two individuals that differ in distance up to 1 centimetres can get the same fitness due to rounding of the values.

During testing the distance between the transceiver and the receiver of the ultrasonic sensor is not taken into account. This causes minor errors in distance measurement when the robot is close to the wall, however this measurement error will be the same for all tests. This could be solved with Pythagoras theorem as follows. The distance we test our robot is 30 cm to 10 cm. This makes the equation for determining the correct distance:

$$B = \sqrt{\frac{(A)^2}{4} + C^2}$$

(5.1)

Where $A$ is the distance between the transceiver and the receiver on the ultrasonic sensor in cm. And $B$ is the distance the ultrasonic sensor registers, and $C$ is the straight distance to the wall. When the robot is 10cm away from the wall the distance error is at its maximum. When $C = 10$ cm the error is 0.078 cm and when $C = 30$ cm the error is 0.026 cm. The accuracy of the ultrasonic sensor is somewhere
5.4. CONCLUSIONS

in the 3 mm range\cite{17}. This means that the error from faulty measurement is less then the accuracy of the ultrasonic sensor and the error will not affect the test result. Another factor to take in to account is that when testing of the robot started, the spiked surfaces of the legs were still new and sharp. During testing of the robot, the numerous walking movements made by the robot made the spiked surfaces dull. The loss of friction may have had a negative impact on the last few iterations of testing. To see if this was true, during the later testing stages, a comparison was made with an already measured individual from the early stages of testing. The previous delta distance that the robot moved was 10 cm, the later test gave a delta of 9 cm.

The robot is always connected to the power grid with a cable which inhibits movement somewhat. Because of the long testing periods, a battery does not suffice.

5.3.1 Compact genetic algorithms

Due to the scope stating that a simple computing unit is used for running the GA, a cGA solution was considered. However the standard GA used never exceeded the memory limit of the computing unit. As such there would have been little to no difference in experimental results if cGA were used instead of GA.

5.4 Conclusions

With extensive testing of the GA the first thing to be concluded is that the demonstrator learned with each generation and gradually improved. The research questions that were set out to be answered by this project are:

- In what way can servo motor commands that enable a 4-legged robot to walk be learned using a genetic algorithm implemented on an inexpensive computing platform?

To implement GA, a vector with positional data for the servos is used as the chromosomes for the GA. Each servo movement is mapped to a gene in this vector.

- What kind of fitness criteria and initial population must be constructed to make a feasible walking movement within 10 generations of GA.

Before and after the servo movements the distance to a wall is measured, the difference between the measurements is the walked distance of the robot. The fitness criteria is then set to the absolute difference between the walked distance and a set desired length.

- What implementation of the population size, crossover probability, elitism and mutation probability gives the furthest possible distance travelled by the demonstrator.
The population size has the biggest effect, the bigger it is, the better end result. If the crossover probability and mutation probability is decreased, a higher converging rate is achieved but creates a smaller diversity. The elitism has the same effect if it is increased.

5.5 Future Work

For future work the first update that could be made is the general design of the robot. Right now the robot is relatively simple in its construction and a much more complex robot would give more insight in how the genetic algorithm can be utilised. Another thing would be to take away one leg off from the existing robot and see how the genetic algorithm would adapt.

Improvements that could be made to the algorithm would be to increase the population size of the standard GA to as large as the hardware enables. This would allow tests in how a larger population changes the way GA learns and adapts to this new rule set.

Right now the focus is on the standard GA and not any code from its many alternative forms that each have improved some bit of the algorithm. One of this is the fore mentioned cGA. With the cGA tests could be run to see if the same population really does lead to the same results as the ordinary GA does. Another experiment that could be done is to populate the cGA with as much population as the memory of the hardware can compute. This would lead to many more movements per servo and would improve the distance the robot can travel in one iteration.
Bibliography


Appendix A

Experiment setup

For the experiment the following parts are needed:

- The demonstrator.
- A test area with rough flat even ground for the demonstrator to walk on.
- Two strings.
- A metal railing that runs above the robot.

To start off the experiment the robot is attached to the metal railing running above it with two strings, shown in Figure A.1. The program is then initiated and the demonstrator executes the first distance measurement to a flat surface between 10 to 40 centimetres in front of it. After the measurement is completed the robot initiates its movement pattern based on our variables for the GA. It moves one leg at the time and when it is done with its entire movement pool (three movements per servo) the servos reset to their initial position. The demonstrator takes its final measurement of the distance travelled. If it were to turn in its testing phase the demonstrator would register a longer distance, than it would if it was pointing straight at the flat surface. As a final act of balancing itself out and returning to its original position, the robot raises all of its legs and is only supported by the two strings attached to it. The robot then repeats the process with the following sets of movements.

The demonstrator is also attached to a guide line to ease the resetting and make each start of an iteration as unvarying as possible.

The fittest individual’s distance travelled of the last generation will work as the variable for the experimental design.
Figure A.1. The test rig for the demonstrator, used for all our experiments.
Appendix B

Electronic schematic

Figure B.1. The electronic schematic for the demonstrator, made in Fritzing computer software.
Appendix C

Demonstrator Code

C.1 GA.py

# The MIT License
#
# Copyright (c) 2011 John Svazic
#
# Permission is hereby granted, free of charge, to any
# person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without
# limitation the rights
# to use, copy, modify, merge, publish, distribute,
# sublicense, and/or sell
# copies of the Software, and to permit persons to whom the
# Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice
# shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
# KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
# NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
```python
from random import (choice, random, randint)
from Robot_main import (robot, Joints)

_all_ = [‘Chromosome’, ‘Population’]

class Chromosome:
    
    """This class is used to define a chromosome for the genetic algorithm simulation.

    This class is essentially nothing more than a container for the details of the chromosome, namely the gene (the string that represents our target string) and the fitness (how close the gene is to the target string).

    Note that this class is immutable. Calling mate() or mutate() will result in a new chromosome instance being created.
    """

    _target_dist = 20
    _target_gene = [0]*8*3
    _Minval = -70
    _Maxval = 70

def __init__(self, gene, fitness=None, delta=None):
    if fitness == None and delta == None:
        self.gene = gene
        self.fitness, self.delta = self._update_fitness(gene)
```

else:
    self.gene = gene
    self.fitness = fitness
    self.delta = delta

def mate(self, mate):
    """Method used to mate the chromosome with another chromosome, resulting in a new chromosome being returned. Two point crossover.""
    firstpivot = randint(0, len(self.gene) - 1)
    secondpivot = randint(0, len(self.gene) - 1)
    if firstpivot == secondpivot:
        gene1 = self.gene[:firstpivot] + mate.gene[:firstpivot:]
        gene2 = mate.gene[:firstpivot] + self.gene[:firstpivot:]
        return Chromosome(gene1), Chromosome(gene2)
    elif firstpivot > secondpivot:
        pivot = firstpivot
        firstpivot = secondpivot
        secondpivot = pivot
        return Chromosome(gene1), Chromosome(gene2)

def mutate(self):
    """Method used to generate a new chromosome based on a change in a random character in the gene of this chromosome. A new chromosome will be created, but this original will not be affected."""
APPENDIX C. DEMONSTRATOR CODE

```python
gene = self.gene
delta = randint(self._Minval, self._Maxval-1)
dx = randint(0, len(gene) - 1)
gene[idx] = (gene[idx] + delta) % self._Maxval

return Chromosome(gene)
```

```python
@staticmethod
def _update_fitness(gene):
    # global Outputfile, i
    
    Helper method used to return the fitness for the chromosome based on its gene.
    
    fitness = 0
delta = 0
delta = robot(gene)
fitness = abs(Chromosome._target_dist-delta)
for a, b in zip(gene, Chromosome._target_gene):
    # fitness += abs(a - b)
    # Outputfile.writevector([i, fitness, delta, gene])
    print("Gene: %r, Delta: %d, Fitness: %d" %(gene, delta, fitness))

return fitness, delta
```

```python
@staticmethod
def gen_random():
    
    A convenience method for generating a random chromosome with a random gene.
    
    gene = []
    for x in range(len(Chromosome._target_gene)):
        gene.append(randint(Chromosome._Minval, Chromosome._Maxval))
    return Chromosome(gene)
```

```python
class Population:
    
```
A class representing a population for a genetic algorithm simulation.

A population is simply a sorted collection of chromosomes (sorted by fitness) that has a convenience method for evolution. This implementation of a population uses a tournament selection algorithm for selecting parents for crossover during each generation’s evolution.

Note that this object is mutable, and calls to the evolve() method will generate a new collection of chromosome objects.

```python
_tournamentSize = 2

def __init__(self, size=1024, crossover=0.8, elitism=0.1, mutation=0.03, Continue=False):
    self.elitism = elitism
    self.mutation = mutation
    self.crossover = crossover

    buf = []
    if Continue:
        ''' If the previously run was cancelled, run this to continue '''
        i = -size
        while i < 0:
            gene = []
            with open('Output.csv') as file:
                line = file.readlines()[i]
                line = line.rstrip('
')
                linevector = line.split(';', ')
                for number in linevector[3:27]:
                    gene.append(int(number))
            buf.append(Chromosome(gene, int(linevector[1]), int(linevector[2])))
            i += 1
    self.population = buf
```
APPENDIX C. DEMONSTRATOR CODE

```python
self.gen = linevector[0]
else:
    for i in range(size):
        buf.append(Chromosome.
            gen=gen_random())
self.population = list(sorted(buf, key=lambda x:
            x.fitness))

def _tournament_selection(self):
    """A helper method used to select a random chromosome from the population using a tournament selection algorithm."
    best = choice(self.population)
    for i in range(Population.tournamentSize):
        cont = choice(self.population)
        if (cont.fitness < best.fitness): best = cont
    return best

def _selectParents(self):
    """A helper method used to select two parents from the population using a tournament selection algorithm."
    return (self._tournament_selection(), self.
        _tournament_selection())

def evolve(self):
    """Method to evolve the population of chromosomes."
    size = len(self.population)
    idx = int(round(size * self.elitism))
    buf = self.population[:idx]

    while (idx < size):
        if random() <= self.crossover:
            (p1, p2) = self._selectParents()
            children = p1.mate(p2)
            for c in children:
```

---

38
if random() <= self.mutation:
    buf.append(c.mutate())
else:
    buf.append(c)
idx += 2
else:
    if random() <= self.mutation:
        buf.append(self.population[idx].mutate())
    else:
        buf.append(self.population[idx])
idx += 1

self.population = list(sorted(buf[size:], key=lambda x: x.fitness))

class writetofile:
    
    # Object treated as the output file.
    #
    def __init__(self, filename, size=None, crossover=None,
                 elitism=None, mutation=None):
        self.file = open(filename, "a")
        if size != None:
            self.file.write("\nSize; Crossover; Elitism;
                 Mutation\n")
            self.file.write("%r;%r;%r;%r\n%(size, crossover,
                 elitism, mutation))
        self.file.write("Generation; Fitness; Delta; Gene;\n                 \n")
    def writevector(self, vector):
        # Writes the vector to the file as a row.
        #
        for value in vector:
            if value == vector[-1]:
                for gene in value:
                    self.file.write("%d;%(gene))
            else:
                self.file.write("%d;%(value))
        self.file.write("\n")
    def close(self):
        # Closes the file

39
APPENDIX C. DEMONSTRATOR CODE

```python
self.file.write("End of test")
self.file.close()

def main():
    Start = 0
    while Start == 0:
        Servos = Joints()
        Servos.reset(0)
        with open('Output.csv') as file:
            last_line = file.readlines()[-1]
        if last_line == "End of test":
            state = 0
            Start = input("Skriv ett om du vill starta: ")
        else:
            state = input("Senaste testet avbrots 1 om fortsatt 0 om börja om: ")
            Start = 1
    else:
        size=10
crossover=0.8
elitism=0.3
mutation=0.3
maxGenerations = 10
if state == 0:
    Outputfile = writetofile("Output.csv", size,
                            crossover, elitism, mutation)

    pop = Population(size, crossover, elitism, mutation)
    for individual in range(0, size):
        Outputfile.writevector([0, pop.population[
                                    individual].fitness ,pop.population[
                                    individual].delta ,pop.population[
                                    individual].gene])
        i = 1
else:
    Outputfile = writetofile("Output.csv")
    pop = Population(size, crossover, elitism, mutation,
                      True)
    i = int(pop.gen)+1
for i in range(i, maxGenerations + 1):
    print("Generation %d: %r, Fitness: %d" % (i, pop.
                                         population[0].gene ,pop.population[0]).
```
C.2. ROBOT_MAIN.PY

```python
if pop.population[0].fitness == 0:
    break
else:
    pop.evolve()

    for individual in range(0, size):
        Outputfile.writerow([i, pop.population[i].fitness, pop.population[i].delta, pop.population[i].gene])

else:

    print("End of test")
    Outputfile.close()

if __name__ == "__main__":
    main()
```

C.2 Robot_main.py

```python
from __future__ import division
from Distance import distance
import Adafruit_PWMServoDriver
import time

# The code is written by Kim Fogelstrom and Viktor Lahti
# For the degree project in Mechatronics at KTH.
# TRITA: TRITA MMK 2017:28 MDAB 646

class Joints(object):
    """
    This class is used to move the robot's servos, reset
    the servos to 90 degrees, move legs up to reset the angle of the robot and turn
    the servos off.
    It creates an instance of the Adafruit_PWMServoDriver taken
    from
    https://github.com/Adafruit/Adafruit_Python_PWMServoDriver/blob
    /master/Adafruit_PWMServoDriver/PWMservo.py
    """

    def __init__(self):
        self.pwm = Adafruit_PWMServoDriver.PWMServoDriver()
        self.movemax = 70
        self.movemin = -70
```
self.movemiddle = 0
self.pwm.set_pwm_freq(50)

def move_joint(self, channel, bit):
    if bit <= 100:
        if channel in [0, 1, 4, 5]:
            bit = -bit
            bit = bit + 313
            self.pwm.set_pwm(channel, 0, bit)
        else:
            print("Bit, %d is too big!", %bit()

def reset(self, sleeptime):
    for channel in range(0, 8):
        self.move_joint(channel, 0)
        time.sleep(sleeptime)

def resetpos(self, sleeptime):
    self.reset(0)
    time.sleep(0.3)
    for bit in range(0, 130):
        for channel in [1, 3, 5, 7]:
            self.move_joint(channel, -bit)
        time.sleep(sleeptime)
    time.sleep(0.005)
    time.sleep(3)
    self.reset(3)

def turn_off(self):
    for channel in range(0, 8):
        bit = 0
        bit = bit + 313
        self.pwm.set_pwm(channel, 4096, bit)

def move_robot(Servopos, Servos, timesleep):
    Function to move all of the robots servos as required to achieve the walkin movement.
    channel = 0
    for pos in Servopos:
        Servos.move_joint(channel, pos)
        channel += 1
        if channel == 8:
C.3. DISTANCE.PY

```python
code
channel = 0
time.sleep(0.3)
else:
time.sleep(timesleep)

def robot(Servopos):
    
    # Call function for the GA.py, with each step used in the test.
    # Starts with resetting the robot, then measures distance,
    # makes the movement, resets the robot again and then measures the final distance.
    # To measure the distance, Distance.py is used.
    
    # Pos min: −188
    # Pos max: 187
    Servos = Joints()
    Servos.resetpos(0)
    l_1 = distance()
    time.sleep(0.3)
    move_robot(Servopos,Servos,0.3)
    time.sleep(0.3)
    Servos.reset(0)
    l_2 = distance()

    delta = l_1 - l_2

    return delta
```

C.3 Distance.py

```
# Libraries
import RPi.GPIO as GPIO
import time

# The code is written by Kim Fogelstrom and Viktor Lahti
# For the degree project in Mechatronics at KTH.
# TRITA: TRITA MMK 2017:28 MDAB 646

# GPIO Mode (BOARD / BCM)
GPIO.setmode(GPIO.BCM)
```
APPENDIX C. DEMONSTRATOR CODE

```python
# set GPIO Pins
GPIO_TRIGGER = 23
GPIO_ECHO = 24

# set GPIO direction (IN / OUT)
GPIO.setup(GPIO_TRIGGER, GPIO.OUT)
GPIO.setup(GPIO_ECHO, GPIO.IN)

def distance():
    """measures the distance from the ultrasonic sensor to the closest wall."""
    # set Trigger to HIGH
    GPIO.output(GPIO_TRIGGER, True)

    # set Trigger after 0.01ms to LOW
    time.sleep(2)
    GPIO.output(GPIO_TRIGGER, False)
    StartTime = time.time()
    StopTime = time.time()

    # save StartTime
    while GPIO.input(GPIO_ECHO) == 0:
        StartTime = time.time()

    # save time of arrival
    while GPIO.input(GPIO_ECHO) == 1:
        StopTime = time.time()

    # time difference between start and arrival
    TimeElapsed = StopTime - StartTime

    # multiply with the sonic speed (34300 cm/s)
    # and divide by 2, because there and back
    distance = (TimeElapsed * 34300) / 2

    return distance
```

44