

AIAB REPORT

By 184510



3 czerwca 2020

University of Sussex

Acquired Intelligence & Adaptive Behaviour

# Hillclimber, Genetic Algorithm and Vehicles:

## Hillclimber:

### Introduction:

Hillclimber is a heuristic search algorithm used to perform mathematical optimization problems in the field of Artificial Intelligence [1]. Hill climber is given a large set of inputs and a good heuristic function. With this, it then tries to find a sufficiently good solution to the problem. The two biggest characteristics of a hill climbing algorithm is that uses the greedy approach as well as it is a variant of generate and test algorithm.

Greedy algorithms are algorithms that make an optimal choice at each step, in order to find the most overall optimal way to solve the entire problem. The biggest drawback of a greedy algorithm is that it does not consider all the data and therefore it might not find the globally optimal solution [Figure 1]. As shown on the diagram [Figure 1] the double circles show how a greedy algorithm would come up with the most optimal way of solving the problem and the green circles show the actual most optimal solution [2]. A “generate and test” kind of algorithm is an algorithm that will create multiple possible solution and check if each solution is the most optimal one [Figure 2] [1]. Therefore, hill climber is called a “variant” of a generate and test algorithm as it only takes the feedback from the procedure and based on it, it decides the net move in search space.

Obraz zawierający urządzenie, zegar

Opis wygenerowany automatycznie

Figure - Example of a greedy approach that would not give correct result.

Obraz zawierający kwiat

Opis wygenerowany automatycznie

Figure - Flowchart showing how "generate and test" algorithm works

There are three different types of hill climbers: “simple hill climber”, “steepest-ascent hill climber” and “Stochastic hill climber”. The “simple hill climber” as the name suggest is the easiest way to implement a hill climbing algorithm. This type of hill climber performs evaluation taking only one state of a neighbour node at a time. It does it by investigating the current cost/size and declares its current state. Then it tries to check the status of the next neighbour state and if it finds the rate of success to be better then the previous state it moves to the better state otherwise it stays in the same position. This version of a hill climber is good as it consumes very little time however it does not guarantee the best optimal solution as it gets affected by the local optima [3]. The “Steepest-Ascent hill climber” is another variation of a simple hill climbing algorithm. The difference is that now it examines all the neighbour’s nodes of the current state and selects the one which is closes to the goal. This version is more accurate, but it takes more time as it searches for multiple neighbours [1]. The “Stochastic hill climber” is a much different type of a hill climbing algorithm. This is because, rather than examining for all its neighbours before moving, it selects a neighbour node at random and decides whether to choose it as a current state or examine another state [1].

The hill climber I will create will solve the “Knapsack” problem; this will be an example of the “stochastic” hill climbing algorithm. This means it will try to fit in items with different benefits and sizes into a backpack with a specific capacity. The algorithm will try to maximize the value of the benefit while making sure that the volume does not exceed the back packs capacity. In this report I will also try to answer the question of how the cross mutation rate affects the performance of the hill climber.

### Method:

#### Initializer function:

The initializer function initializes all the different variables. It sets the mutation rate to the value entered by the user when calling the class. It will be between 0 and 10. The higher the number the higher the chance of mutation on the gene. It also sets the max capacity of the backpack. Then it initializes an array that stores different items that can be put inside of the backpack. For each item it stores its name, benefit and size. Then the genes variable is initialised and sets at random which items start in the backpack and which ones do not.

#### Mutate function:

The mutate function upon running will perform the mutation on the genes. First, a random number between 0 and 9 is saved as a variable. Then a copy of the genes variable is saved within a new variable called “temporaryGenes”. Then an if statement will check if the random number is lower or higher then the mutation rate. If the random number will be lower then mutation rate, then it will check if the temporary gene at that position is within the backpack, if not it will place it in the backpack and if it is it will remove it from the backpack. Then another check happens that will check if the temporary genes after mutating have better fitness value then the original ones with the use of the “fitness” function. If they are better, then the original genes are set to the temporary genes.

#### Fitness function:

The main functionality of the fitness function is to calculate the total benefit and volume of one set of genes. First, the benefit and volume variables are initialized and set to zero. Then a for loop will perform 10 times, and on each execution, it will check with the use of an if statement, if the gene at the current position is within the backpack. If yes it would add the benefit and volume of that gene to the total value respectively. Once that is done for every gene, outside of the for loop an if statement checks if the total volume of all the items that have been placed within the backpack is not exceeding the total capacity of the backpack. If it does the function returns 0 and if not, it returns the total benefit.

#### Run function:

The run function manages the running of each mutation. First the method initializes a new empty array. Then a for loop runs the amount of times that the user will specify when calling this method. Then within that for loop the run function will call the mutate function and run it. Once that is done it will add the next fitness value into the array initialised within this method at each iteration of the loop. Once the for loop executes it will return the array with all the different possible fitness values.

#### Pseudo Code:

1. Fitness track 🡨 new empty array
2. For (amount of times specified by user)
   1. Run the mutate function
   2. Fitness track 🡨 fitness of the gene (runs fitness function)
3. Return fitness track

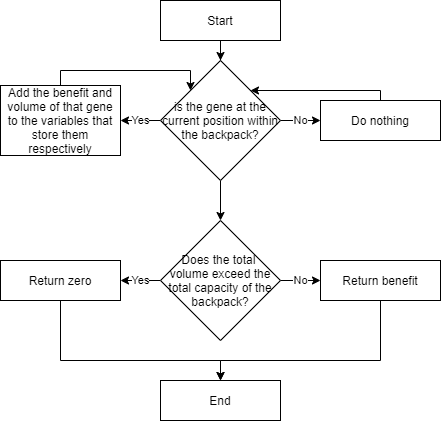
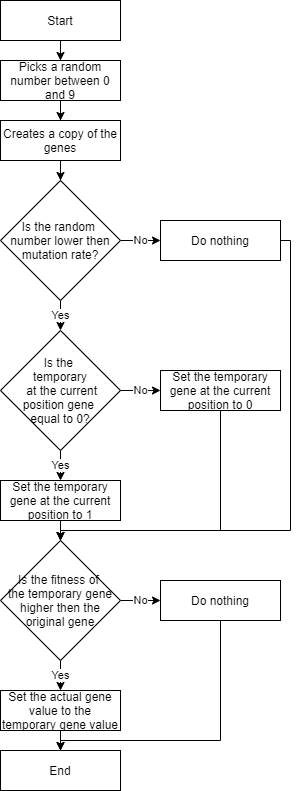


Figure - Flowchart for mutate method Figure - Flowchart for the fitness method

### Results:

(Disclaimer: All graphs generated have been saved with plt.savefig() in order to insert them within this word document)

#### Different mutation rates:

Obraz zawierający zrzut ekranu

Opis wygenerowany automatycznieObraz zawierający zrzut ekranu

Opis wygenerowany automatycznie

Figure - hillclimber low mutation rate Figure - hillclimber high mutation rate

The two diagrams show how the hillclimber performs for over 100 generations [Figure 5 and 6]. As shown on the first diagram [Figure 5] when the low mutation rate is entered (20% in that case) the fitness of the individual has not improved at all over the 100 generations. This is because the lower the mutation rate, the less chance there is for the individual to mutate and improve its fitness. In the second diagram [Figure 6] the mutation rate is set to 100%. This means that the mutation will happen every new generation. As shown on the diagram [Figure 6] the mutation either have positive or neutral impact on the individual’s fitness.

#### Different amount of generations:

Obraz zawierający zrzut ekranu

Opis wygenerowany automatycznieObraz zawierający zrzut ekranu

Opis wygenerowany automatycznie

Figure - 10 different hill climbers for 100 generations Figure - 10 different hill climbers for 1000 generation

The two diagrams show ten different hill climbers each, one of them show the hill climber working over 100 generations [Figure 7] and the other one represents the hill climber working over 1000 generations [Figure 8]. The first diagram shows very well how hill climber can perform differently with the same mutation rate (100% in this case) [Figure 7]. As shown on the diagram one individual represented with the light blue line has not improved fitness at all even that the mutation rate was 100% and it went for 100 generations. The hill climber creates the genes at random at the beginning and it might not be fixable by changing just one gene at the time [Figure 7]. Then the dark blue line in the diagram shows another interesting case where the randomly created genes at the beginning have very good fitness and over 100 generations changing one gene at a time does not improve the fitness at all. Another interesting case is the grey line. The randomly created genes are very good and then over few generations the fitness is improved and reaches the best possible fitness. The comparison between 100 generations and 1000 generation [Figure 7 and 8] shows really well how after not that many generations the hill climber reaches the highest limit and that limit is caused by the genes that are randomly generated at the beginning.

### Discussion and Conclusion:

During the development of my hill climber I have decided to create a stochastic hill climber. The results that I have achieved presented by the graphs in the “Results” section showed that the hill climber met all the requirements. We have also observed that the results provided by the hill climber are strongly affected by the mutation rate which has been explored in the results section.

In future the thing that could be changed is to approach the problem and create a simple hill climber and steepest-ascent hill climber and use them in order to solve the “Knapsack” problem. By doing this I would be able to discuss in my report how different types of hill climbers solved the problem as well as show graphically which hillclimber provided the most accurate and promising results.

## Genetic Algorithm:

### Introduction:

Genetic algorithm just like any other algorithm is trying to find a solution to a problem by using step-by-step procedure. However, what is special about genetic algorithm is that they are based on genetic models. The genetic algorithms are inspired by the Darwin’s theory of Natural Selection [4]. The Darwin’s theory says that there are three main principles that are needed for an evolution to happen: heredity, variation and selection [4]. The heredity principle means that there needs to be a process that will allow the children to receive the property of their parents [5]. The variation principle means that the population needs variety of traits [5]. Then the selection principle means that there must be a process that will select strongest and best adapted and make them parents that will have children that they will pass their genes to [5]. The genetic algorithm works similar way. This is because it creates a population, then picks the fittest items within the population and make them parents of a child and this process will carry on until perfect population.

There are many different types of genetic algorithms. One of them is the simple genetic algorithm. The SGA works by first, selecting parents with the use of the fitness function. Then the reproduction can be performed in multiple different ways. For example, it can be performed with the use of the roulette wheel [Figure 9]. Once that is done the crossover is performed on parent chromosomes. Then the offspring chromosomes are mutated, and then add the offspring back to the pool. Then this process repeats for the specified amount of generations [6]. Another type of genetic algorithm is the parallel genetic algorithm [8]. This type of a genetic algorithm is few different genetic algorithms that are performing the same task. Then once each one finishes and picks their best individual the best one is picked once again from the ones that different types of GA picked [8][Figure 10].

The genetic algorithm I have created is the microbial genetic algorithm. This means that this type of GA has a mutation and crossover rate. This GA takes two individuals from the population at random and compare their fitness based on their genes. The genes of the winner are copied over to the loser which is affected by the crossover rate. Then the loser with the new genes is mutated which is affected by the mutation rate. Once that is done the mutated loser and the original winner is put back into the population. This happens as many times as the specified generation number [9]. In this report I will also try to discuss how the mutation rate affects the performance of the genetic algorithm I have created.

Obraz zawierający zegar

Opis wygenerowany automatycznie

Figure - Roulette wheel selection inspired by [7]

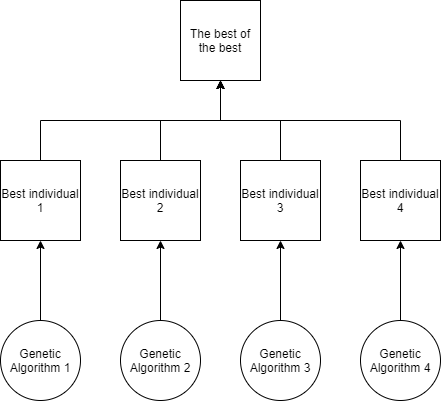


Figure - Diagram representing parallel genetic algorithm inspired by [8]

### Method:

#### Initializer function:

The initializer function initializes all the different variables. It sets the mutation rate, crossover rate, quantity and k to the values entered by the user when calling the class. The quantity will be used to set how many individuals will be within the population. The k value will be used to set the boundaries of how many neighbours will be taken under consideration away from the individual picked at random. Then the initializer will set the capacity of the backpack. Then it initializes an array that stores different items that can be put inside of the backpack. For each item it stores its name, benefit and size. Then a new empty array will be initialized and with a use of a for loop it will store a population of individuals.

#### Fitness function:

The fitness function is the same within the genetic algorithm class as it was in the hill climber class.

#### Mutate function:

The mutate function is nearly identical to the one used within hill climber class; within the genetic algorithm class it has an extra parameter called location. This time the mutate function takes under consideration the location which allows to access different individuals within population.

#### Survival of the fittest function:

The survival of the fittest function will pick two individuals, one fully randomly and one at a position within the range, set with the variable k. Then with the use of the fitness method the fitness of the two individuals will be compared. Depending which individual has a higher fitness value that individual’s genes will be copied over to the other individual. Once this happens the mutate function will be performed on the individual with the copied over genes.

#### Run function:

The run function manages the running of each mutation. First the method initializes a new empty array. Then a for loop runs the amount of times that the user will specify when calling this method. Then within that for loop the run function will call the survival of the fittest function and run it. Once that is done it will initialize a new variable that will be used to store the total fitness of the population and for now it will set it to 0. Then a for loop will be used to go through the whole population. For each item within population it will add the fitness to the total fitness variable. Once that is done and the for loop executes the average of the total fitness will be calculated and stored within the total fitness variable. Then the method will add next fitness value into the array. Once that is done this will repeat for the amount of times set by user and the array with all the values will be returned.

#### Pseudo Code:

1. Fitness track 🡨 new empty array
2. For (amount of times specified by user)
   1. Run the survive of the fittest function
   2. Total amount of fitness 🡨 0
   3. Fitness track 🡨 fitness of the gene (runs fitness function)
   4. For (number of individuals within population)
      1. Total amount of fitness 🡨 fitness of the gene at a specified position (uses fitness function)
   5. Total amount of fitness 🡨 average of total amount of fitness
   6. Fitness track 🡨 Total amount of fitness
3. Return fitness track

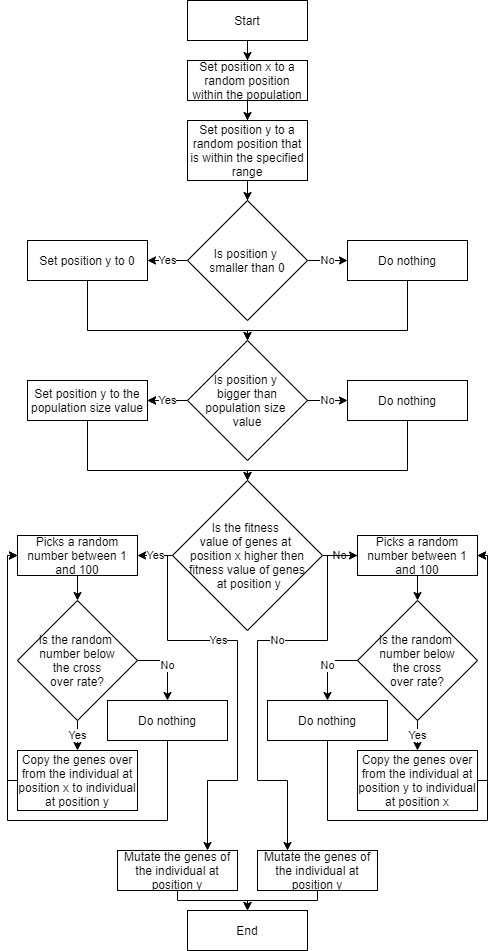


Figure - Flowchart for survival of the fittest method

### Results:

(Disclaimer: All graphs generated have been saved with plt.savefig() in order to insert them within this word document)

#### Different number of individuals within population:

Obraz zawierający tekst, mapa

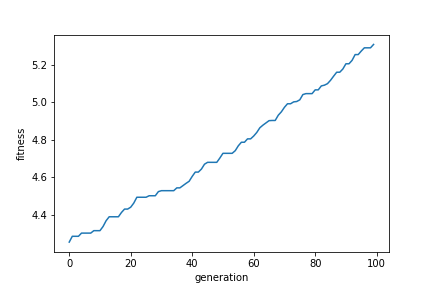
Opis wygenerowany automatycznie

Figure - Genetic Algorithm small population Figure - Genetic Algorithm large population

The diagrams above show the difference when the number of individuals within population is changed [Figure 12 and 13]. The first diagram has only 10 individuals within it’s population and after 100 generations the fitness goes from 0 to the maximum possible fitness [Figure 12]. Whereas the other diagram shows 1000 individuals within it’s population and after 100 generations the fitness goes from 0 to only 5.2 which is a very small difference [Figure 13]. This is because the population displayed is the average of all the individual’s fitness, and because there is so many there is a lot of individuals that are not getting mutated at all.

#### Different amount of generations (large population):

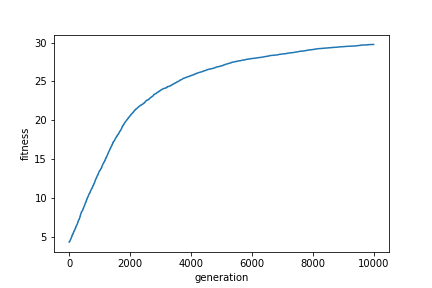


Figure - Genetic Algorithm large population, 10 thousand generations

The diagram above shows how a population with 1000 individuals can reach the maximum fitness for every individual within the population [Figure 14]. Comparing the two diagrams [Figure 13 and 14] I conclude that the higher the number of individuals within population the more generations are needed for them to reach the highest possible fitness.

#### Different mutation rate:

Obraz zawierający tekst, mapa

Opis wygenerowany automatycznieObraz zawierający tekst, mapa

Opis wygenerowany automatycznie

Figure - Genetic Algorithm high mutation rate Figure - Genetic Algorithm low mutation rate

The diagrams above show how the different mutation rate affect the performance of the genetic algorithm [Figure 15 and 16]. As shown on the diagram the genetic algorithm with high mutation rate (100% in this case) is very consistent at improving the fitness [Figure 15]. Whereas the genetic algorithm with low or even non existing mutation rate (10% in this case) is less consistent at improving the population average fitness. This is because when there is no mutation, or the mutation rate is low the chance of improving the genes is way lower as the improvement of fitness is done mostly by copying the genes from the winner to a loser and the loser with copied genes does mutate or mutates very slightly which makes the improvements less consistent as shown on the diagram [Figure 16].

#### Different number of neighbours being considered when picking the second individual:

Obraz zawierający tekst, mapa

Opis wygenerowany automatycznieObraz zawierający tekst, mapa

Opis wygenerowany automatycznie

Figure - Genetic Algorithm a lot of neighbours Figure - Genetic Algorithm not many neighbours

The diagrams above show how changing the number of neighbours being taken under consideration changes the results of the genetic algorithm [Figure 17 and 18]. The results are similar to the ones from the previous comparison. Genetic algorithm fitness improves more consistently with higher number of neighbours being taken under consideration [Figure 17]. Whereas when only two neighbours being taken under consideration the fitness improvement is way less consistent [Figure 18]. This is because when more individuals are taken under consideration when being picked to be the pair the chance of picking someone with better much better genes is higher and therefore the consistency improves [Figure 17]

#### Different crossover rates:

Obraz zawierający mapa

Opis wygenerowany automatycznieObraz zawierający tekst, mapa

Opis wygenerowany automatycznie

Figure - Genetic Algorithm high crossover rate Figure - Genetic Algorithm low crossover rate

The diagrams above show how the change in crossover rate affects the fitness of the genetic algorithm [Figure 19 and 20]. As shown on the diagram when the crossover rate is high (100% in this case) the fitness with each generation either improves or stays the same and it reaches the max fitness fairly quickly [Figure 19]. Whereas when the crossover rate is low (10% in this case) the fitness with each generation can either improve, stay the same or it can get worse [Figure 20]. Because of this it takes way longer for the algorithm to improve the average fitness of the population [Figure 20]. This happens because when the cross over rate is not 100% then the chance that each gene of the winner being copied over to the loser is lower.

### Discussion and Conclusion:

During the development of my genetic algorithm I have decided to create a microbial genetic algorithm. The results that I have achieved presented by the graphs in the “Results” section showed that my genetic algorithm met all the requirements. I have also managed to discuss and show how different crossover and mutation rates can affect the results provided by the genetic algorithm. By creating the genetic algorithm, I also understood more how the Darwin’s natural selection theorem applies when it comes to designing and creating the genetic algorithm.

In future the thing I could do to improve the performance of my genetic algorithm is to approach the “Knapsack” problem with the use of “parallel genetic algorithm” approach. I believe that by using different types of genetic algorithms and then using the parallel genetic algorithm the end result, the population created by it could be even fitter than the one created by my microbial genetic algorithm.

## What are the differences between hillclimber and GA?

Obraz zawierający tekst, mapa

Opis wygenerowany automatycznieObraz zawierający zrzut ekranu

Opis wygenerowany automatycznie

Figure - Genetic Algorithm high mutation rate Figure – Hillclimber high mutation rate

Obraz zawierający tekst, mapa

Opis wygenerowany automatycznieObraz zawierający zrzut ekranu

Opis wygenerowany automatycznie

Figure - Genetic Algorithm low mutation rate Figure - Hillclimber low mutation rate

Obraz zawierający tekst, mapa

Opis wygenerowany automatycznieObraz zawierający zrzut ekranu

Opis wygenerowany automatycznie

Figure - Genetic Algorithm with mutation rate set to 0 Figure - Hillclimber with mutation rate set to 0

The biggest difference between a hill climber and a genetic algorithm is that a genetic algorithm deals with whole population of individuals whereas the hill climber only deals with on individual. The genetic algorithm is better at improving the fitness of individual compare to the hill climber [Figure 21 and 22]. Another difference between genetic algorithm and hillclimber is that the hillclimber is much more affected with the mutation rate compare to the genetic algorithm [Figure 21, 22, 23 and 24]. This is because if cross over rate is 0% the hill climber won’t be able to improve the fitness of the individual at all [Figure 26], whereas the genetic algorithm uses different methods to improve the average fitness and therefore it can still improve fitness even if the mutation rate is set to 0% [Figure 25].

## Vehicles:

### Introduction:

The idea behind the Braitenberg Vehicles is to create simple vehicles with a simple internal structure that will behave in unexpectedly complex ways [10]. The control mechanisms that generate behaviour are very simple yet the behaviours that they can produce can be called: aggression, love, optimism and even foresight [10]. The Braitenberg vehicles follow the law of uphill analysis and downhill invention [10]. This law means that it is much more difficult to try and guess the internal structure of the vehicles from just observing their behaviour than it is to create the structure that gives that behaviour [10].

The Braitenberg vehicles are mechanisms with cognitive functionality (artificial intelligent) that are quite simple [11]. They are simple mobile machines that use basic “sensory-motor” to produce something that will be considered as cognitive behaviours [11]. “Sensory-motor” means that the model will have a sensor receptor that will affect how the model work [12].

In this report I will try and create four different vehicles. First one will represent aggression. In this vehicle the sensor-motor connections can be crossed and if that happens the behaviour changes [11]. If the simulation is directly ahead of the vehicle then the vehicle will move directly towards it, however if the simulation is to one side then the vehicle will tend to veer towards it and as that happens the speed of the vehicle will increase [11]. The second vehicle will represent a coward type of vehicle. This kind of vehicle will spend more time in places where there is less simulation and it will speed up when it will get exposed to the simulation [11]. If the simulation would be directly ahead of it then the vehicle could hit the source otherwise it will tend to turn away from the stimulation [11]. The next kind of vehicle will represent the lover. This vehicle will be created by making a connection between the lover and the aggressor inhibitory [11]. The lover will approach the stimulation and it will orient towards it and come to rest facing it [11]. Then the last the type of vehicle I will create is the explorer type of vehicle. This type of vehicle will show a completely different behaviour to the lover. The explorer will come to rest facing away from the stimulation and it will go around the source of light until it finds a stronger source of light [11].

### Method:

All the different vehicles will be using the run function that has been provided to us. It will be used to create the different types of vehicles as well as it will be used to display them in a graph format.

#### Aggressor:

As the aggressor is contra-lateral which means that is relating to or denoting the side of the body opposite to that on which a structure or condition occurs. Aggressor also has positive connections which means that the motors will turn with stimulation. In order to do this, we will need to create genotype. As the aggressor is contra-lateral we will only use two wires that connect left sensor to the right motor and second one that connects right sensor to the left motor. As the aggressor has positive connections we will set the bias numbers to zeros so the turn will be performed without simulation.

#### Coward:

As the coward is ipsilateral which means that it is beginning to or occurring on the same side of the body. Coward also has positive connections which means that the motors will turn without stimulation. In order to do this, we will need to create genotype. As the coward is ipsilateral we will only use two wires that connect left sensor to the left motor and second one that connects right sensor to the right motor. As the coward has positive connections, we will set the bias numbers to zeros so the turn will be performed without simulation.

#### Lover:

As the lover is ipsilateral which means that it is beginning to or occurring on the same side of the body. Lover also has negative connections which means that the motors will turn with stimulation. In order to do this, we will need to create genotype. As the lover is ipsilateral we will only use two wires that connect left sensor to the left motor and second one that connects right sensor to the right motor. As the lover has negative connections, we will set the bias numbers to ones so the turn will be performed with simulation.

#### Explorer:

As the explorer is contra-lateral which means that is relating to or denoting the side of the body opposite to that on which a structure or condition occurs. Explorer also has negative connections which means that the motors will turn with stimulation. In order to do this, we will need to create genotype. As the explorer is contra-lateral we will only use two wires that connect left sensor to the right motor and second one that connects right sensor to the left motor. As the explorer has negative connections, we will set the bias numbers to ones so the turn will be performed with simulation.

#### One eyed phototaxis:

The one eyed phototaxis will use a modified version of the run function that have the connection between one eye (one senor) and the wheels (two motors) hard coded within it. This time when entering the genotype, we only pass in the biases. One will be bigger than the other so the vehicle will spin.

#### Pseudo code:

1. Set the time
2. Set the initial position
3. Set the initial bearing
4. Set the geno type
5. Set if you want the graph to be shown
6. Run the run method and pass to it all the variables set above

This pseudo code will be used by each different type of vehicle and by setting the variables to different values the code will be able to create different vehicles with the use of the run function provided to us.

### Results:

(Disclaimer: All graphs generated have been saved with plt.savefig() in order to insert them within this word document)

#### Aggressor:

Obraz zawierający mapa

Opis wygenerowany automatycznie

Figure - Graph representing aggressor vehicle behaviour

As seen on the diagram above [Figure 27] the vehicle starts at a position (-5, -5). The time has been set to 200 and initial bearing to 30 so that the vehicle shows really well the behaviour of the vehicle. Then the expected behaviour has been visualised on the diagram. The vehicle went around the light at increasing speed and then it carried on moving forward [Figure 27].

#### Coward:

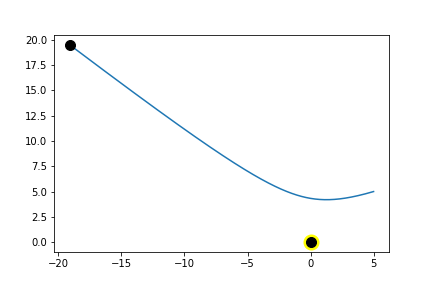


Figure - Graph representing coward vehicle behaviour

As see on the diagram above [Figure 28] the vehicle starts at position (5, 5). The time has been set to 200 and the initial bearing has been set to 200 so that the vehicle shows really well the behaviour of the vehicle. Then the expected behaviour has been visualised on the diagram. The vehicle was moving towards the light and then it turn away from the light and go opposite direction [Figure 28].

#### Lover:

Obraz zawierający mężczyzna

Opis wygenerowany automatycznie

Figure - Graph representing lover vehicle behaviour

As see on the diagram above [Figure 29] the vehicle starts at position (-5, 0). The time has been set to 200 and the initial bearing has been set to 30 so that the vehicle shows really well the behaviour of the vehicle. Then the expected behaviour has been visualised on the diagram. The vehicle was approaching the source of light as well as it oriented towards it and it came at rest facing towards it [Figure 29].

#### Explorer:

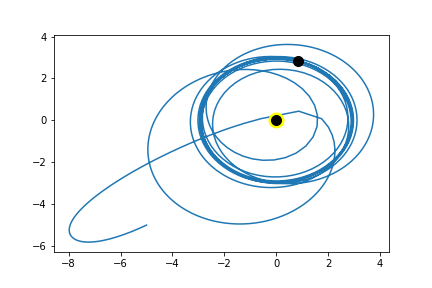


Figure - Graph representing explorer vehicle behaviour

As see on the diagram above [Figure 30] the vehicle starts at position (-5, -5). The time has been set to 200 and the initial bearing has been set to 30 so that the vehicle shows really well the behaviour of the vehicle. Then the expected behaviour has been visualised on the diagram. The vehicle approaches the light and when it was close it went around it and started to spin around the light of source. This is because explorer type of vehicle wants to be close to the source of light but as soon as stronger source will be found it will move towards the stronger source [Figure 30].

#### One eyed phototaxis:

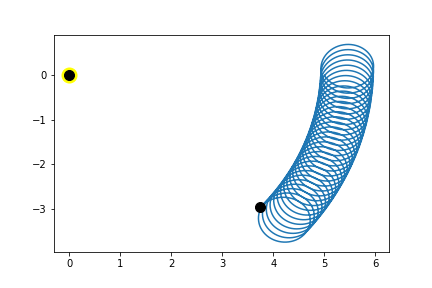


Figure - Graph representing one eyed phototaxis behaviour (short period of time)

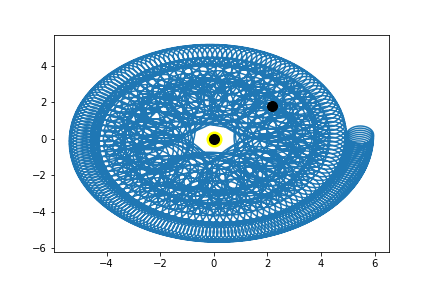


Figure - Graph representing one eyed phototaxis behaviour (long period of time)

The diagrams above [Figure 31 and 32] show behaviour of one eyed phototaxis. This vehicle has only one sensor that is being attracted by the light. Then the expected behaviour has been seen as one bias is bigger than the other the vehicle spins and as it spins it get’s closer to the source of light [Figure 31 and 32] . The two diagrams also show that when there is only one sensor the vehicle will get closer and closer to the source of light the more time is given [Figure 31 and 32].

### Discussion and Conclusion:

#### Difference between the behavioural strategies of one-eyed and two eyed phototaxis:

The biggest difference between one eyed and two eyed phototaxis that I have observed is that one eyed phototaxis has one sensor and the other one has two sensors. This shows the results of having one eye (in the middle) instead of having two eyes (left and right). This would limit our vision and how wide we can see. The vehicles showed that well as when there have been two sensors, we could have manipulated how the vehicle reacts to the source of light whereas with only one sensor the control we had over the vehicle behaviour was much lower. It also showed how the eye is not the only thing that is needed to control the vehicle as other things (in this case the wheels/motors) were equally important in order to make the vehicle move.

#### How rich behaviour emerges from a simple brain:

Different versions of the vehicle showed really well how very basic models that represent a very simple brain can emerge into very reach behaviour. With a very simple model we have been able to see how the vehicle would react to light. We have been able to recreate two sensors that would react depending on how the sensors have been coded and the reaction was performed accordingly. For example, the aggression vehicle represented a aggression behaviour towards the source of light. As shown on the diagram [Figure 27] the vehicle confronted and attacked in some way the source of light. Then the coward type of vehicle showed really well a rich behaviour of when the vehicle would get closer to the source of light it would get scared by it and run the other way. Then the lover vehicle showed the behaviour of the vehicle getting closer and closer to the light and at the end it was facing towards the light and nothing else. Then the explorer showed the opposite of the lover as it was attracted by the light and it was going around it but as soon as a stronger source of light would appear the vehicle would move to that source of light. This over all shows really well how very simple vehicles with simple brain where very good at representing rich behaviours.

#### Conclusion:

During this project I have managed to create correctly all different vehicles. For each vehicle I managed to get the expected behaviour that I have documented. For future I would like to change the numbers used within genotypes to check how that will reflect on the behaviour of the vehicles. Another thing that I would like to explore further in the future would be how one eyed phototaxis act after very long period of time. In order to do this, I would need to have a better machine to perform this as when I tried doing it on my computer it caused the laptop to start overheating and was taking a lot of time to compile.

# References:

1. Mahyavanshi B. (2019) *Introduction to Hill Climbing | Artificial Intelligence.* Available at: <https://medium.com/@bhavek.mahyavanshi50/introduction-to-hill-climbing-artificial-intelligence-a3714ed2d8d8> (Accessed: 10/05/2020)
2. Moore K. & Khlm J. & Ross E. (2020) *Greedy Algorithms.* Available at: <https://brilliant.org/wiki/greedy-algorithm/> (Accessed: 10/05/2020)
3. Bahirat T. (2020) *An Introduction to Hill Climbing Algorithm in AI (Artificial Intelligence).* Available at: <https://www.mygreatlearning.com/blog/an-introduction-to-hill-climbing-algorithm/> (Accessed: 10/05/2020)
4. Kumar M. (2018) *Understanding Genetic Algorithms in the Artificial Intelligence Spectrum.* Available at: <https://medium.com/analytics-vidhya/understanding-genetic-algorithms-in-the-artificial-intelligence-spectrum-7021b7cc25e7> (Accessed: 11/05/2020)
5. Gildenhuys P. *Natural Selection.* Available at: <https://plato.stanford.edu/entries/natural-selection/> (Accessed: 11/05/2020)
6. Sivanandam S.N. & Deepa S.N. *Classification of Genetic Algorithm.* Available at: <https://link.springer.com/chapter/10.1007%2F978-3-540-73190-0_5> (Accessed at: 11/05/2020)
7. Students of Newcastle University (unknown) *Roulette wheel selection.* Available at: <http://www.edc.ncl.ac.uk/highlight/rhjanuary2007g02.php> (Accessed: 11/05/2020)
8. Ivan (2018) *Parallel and distributed genetic algorithms.* Available at: <https://towardsdatascience.com/parallel-and-distributed-genetic-algorithms-1ed2e76866e3#:~:text=Parallel%20genetic%20algorithm,a%20single%20task%20%5B1%5D.&text=The%20difference%20between%20these%20two,selected%20for%20mutation%20and%20crossover.> (Accessed: 11/05/2020)
9. Harvey I. (Unknown) *The Microbial Genetic Algorithm.* Available at: <http://users.sussex.ac.uk/~inmanh/MicrobialGA_ECAL2009.pdf> (Accessed: 11/05/2020)
10. Wiseman J. (1999) *Braitenberg Vehicles.* Available at: <http://people.cs.uchicago.edu/~wiseman/vehicles/> (Accessed: 20/05/2020)
11. Thornton C. (Unknown) *AI Lecture: Braitenberg Vehicles.* Available at: <http://users.sussex.ac.uk/~christ/crs/kr-ist/lecx1a.html> (Accessed: 20/05/2020)
12. Escobar-Juárez E. & Schillaci G. & Hermosillo-Valadez J. & Lara-Guzmán B. *A Self-Organized Internal Models Architecture for Coding Sensory-Motor Schemes.* Available at: <https://www.frontiersin.org/articles/10.3389/frobt.2016.00022/full> (Accessed: 20/05/2020)

# Appendix:

## Hillclimber:

*#all the required imports*

**import** **random**

**import** **matplotlib.pyplot** **as** **plt**

*#the hillClimber class that will be used to solve the knapsack problem.*

**class** **hillClimber**:

*#initializer of the hillClimber class*

**def** \_\_init\_\_(self, mutationRate):

*#sets the mutation rate variable to the parameter passed into the hillClimber class.*

self.mutationRate = mutationRate

*#sets the capacity to 20, which represents the volume of the backpack in the knapsack problem.*

self.capacity = 20

*#initialises items that can be put inside of the backpack. It sets for each item its name, benefit and size.*

self.items = [["a", 5, 3], ["b", 6, 2], ["c", 1, 4], ["d", 9, 5], ["e", 2, 8], ["f", 8, 9], ["g", 4, 10], ["h", 3, 1], ["i", 7, 6], ["j", 10, 7]]

*#initialises at random which items start in a backpack and which ones do not*

self.genes = [random.randrange(2) **for** x **in** range(10)]

*#performs the mutation*

**def** mutate(self):

*#sets the variable to a random number between 0-9.*

i = random.randrange(10)

*#creates a variable that will store temporary genes.*

temporaryGenes = self.genes.copy()

*#checks if the randomly generated number is lower than the mutation rate and if it is performing the code within it.*

**if**(i < self.mutationRate):

*#checks if the gene at position i is equal to zero and if yes it sets the gene at position i to one*

**if**(temporaryGenes[i] == 0):

temporaryGenes[i] = 1

*#and if it is not equal to zero then it sets it to zero*

**else**:

temporaryGenes[i] = 0

*#checks if the fitness value of the temporary genes is higher than the fitness value of the actual genes*

**if**(self.fitness(temporaryGenes) > self.fitness(self.genes)):

*#if yes it sets the actual genes to the temporary genes*

self.genes = temporaryGenes

*#calculates the total benefit of all the genes within the set*

**def** fitness(self, genes):

*#intitalises and sets the benefit variable to 0*

benefit = 0

*#initialises and sets the volume variable to 0*

volume = 0

*#for loop that executes 10 times*

**for** x **in** range(10):

*#checks if the gene at position x is within the "backpack"*

**if**(genes[x] == 1):

*#if yes it adds the benefit and the volume*

benefit += self.items[x][1]

volume += self.items[x][2]

*#checks if the total volume did not exceed the total capacity of the "backpack"*

**if**(volume <= self.capacity):

*#if not it returns the total benefit*

**return** benefit

**else**:

*#otherwise it returns zero*

**return** 0

*#manages the running of each mutation*

**def** runFunction(self, loops):

*#new variable that will store and array*

fitnessTrack = []

*#for loop that goes runs the passed in loops amount of times.*

**for** x **in** range(loops):

*#runs the mutate function*

self.mutate()

*#adds the next fitness value into the array*

fitnessTrack.append(self.fitness(self.genes))

*#returns the array with the fitness of different sets of genes.*

**return** fitnessTrack

*#initializes a new hillclimber and sets the mutation rate*

a = hillClimber(10)

*#initializes the run function and sets the loops which represents the generations.*

b = a.runFunction(100)

*#sets the y lable to be fitness*

plt.ylabel("fitness")

*#sets the x lable to be generations*

plt.xlabel("generation")

*#plots the graph*

plt.plot(b)

*#displayes the graph*

plt.show()

*#this will run the hill climber class 10 times for 10 different results*

**for** x **in** range(10):

a = hillClimber(10)

b = a.runFunction(100)

plt.plot(b)

plt.ylabel("fitness")

plt.xlabel("generation")

plt.show()

## Genetic Algorithm:

*#the GeneticAlgorithm class that will be used to solve the knapsack problem.*

**class** **GeneticAlgorithm**:

*#initializer of the GeneticAlgorithm class*

**def** \_\_init\_\_(self, quantity, mutationRate, k, crossoverRate):

*#sets the crossover rate variable to the parameter passed into the GeneticAlgorithm class.*

self.crossoverRate = crossoverRate

*#sets the k variable to the parameter passed into the GeneticAlgorithm class.*

*#This variable will be used to set how many neighbours can be taken under consideration from the original individual.*

self.k = k

*#sets the quantity variable to the parameter passed into the GeneticAlgorithm class.*

*#sets how many individuals are within the population*

self.quantity = quantity

*#sets the mutation rate variable to the parameter passed the hillClimber class.*

self.mutationRate = mutationRate

*#sets the capacity to 20, which represents the volume of the backpack in the knapsack problem.*

self.capacity = 20

*#initialises items that can be put inside of the backpack. It sets for each item it's name, benefit and size.*

self.items = [["a", 5, 3], ["b", 6, 2], ["c", 1, 4], ["d", 9, 5], ["e", 2, 8], ["f", 8, 9], ["g", 4, 10], ["h", 3, 1], ["i", 7, 6], ["j", 10, 7]]

*#initializes a new empty array*

self.population = []

*#this for loop will set up a population based on how big the quantity variable is.*

**for** i **in** range(quantity):

self.population += [[random.randrange(2) **for** i **in** range(10)]]

*#calculates the total benefit of all the genes within the set*

**def** fitness(self, genes):

*#intitalises and sets the benefit variable to 0*

benefit = 0

*#initialises and sets the volume variable to 0*

volume = 0

*#for loop that executes 10 times*

**for** x **in** range(10):

*#checks if the gene at position x is within the "backpack"*

**if**(genes[x] == 1):

*#if yes it adds the benefit and the volume*

benefit += self.items[x][1]

volume += self.items[x][2]

*#checks if the total volume did not exceed the total capacity of the "backpack"*

**if**(volume <= self.capacity):

*#if not it returns the total benefit*

**return** benefit

**else**:

*#otherwise it returns zero*

**return** 0

*#performs the mutation on the specified location within the population array.*

**def** mutate(self, location):

*#sets the variable to a random number between 0-9.*

i = random.randrange(10)

*#creates a variable that will store temporary genes from the specified location in array list.*

temporaryGenes = self.population[location].copy()

*#checks if the randomly generated number is lower than the mutation rate and if it is performing the code within it.*

**if**(i < self.mutationRate):

*#checks if the gene at position i is equal to zero and if yes it sets the gene at position i to one*

**if**(temporaryGenes[i] == 0):

temporaryGenes[i] = 1

*#and if it is not equal to zero then it sets it to zero*

**else**:

temporaryGenes[i] = 0

*#checks if the fitness value of the temporary genes at a specified location is higher than the fitness value of the actual genes at the same location.*

**if**(self.fitness(temporaryGenes) > self.fitness(self.population[location])):

*#if yes it sets the actual genes at the specified location to the temporary genes*

self.population[location] = temporaryGenes

*#compares two individuals from the population and copies the genes from the winner to the loser*

**def** survivalOfTheFittest(self):

*#initializes a new variable and sets it to a random number between 0 and the max quantity - 1.*

positionX = random.randrange(self.quantity)

*#initializes a new variable and sets it to a value that has been set before.*

allowedValues = list(range(-self.k, self.k))

*#removes 0 from the variable*

allowedValues.remove(0)

*#sets the y position to a random position within the allowed boundaries.*

positionY = positionX + random.choice(allowedValues)

*#makes sure the position is at least the position 0*

**if**(positionY < 0):

positionY = 0

*#makes sure that the position is not outside of the upper boundary.*

**if**(positionY > (self.quantity - 1)):

positionY = self.quantity - 1

*#checks which indvidual has higher fitness value that is checked using the fitness method.*

**if**(self.fitness(self.population[positionX]) > self.fitness(self.population[positionY])):

*#if the position x fitness value is higher a for loop will run 10 times.*

**for** i **in** range(10):

*#initializes a new variable and sets it to a random value between 1 and 100.*

chance = random.randrange(1, 100)

*#checks if the randomly generated number is lower than the crossover rate*

*#this sets the chance of the genes being or not being copied over.*

**if**(chance < (self.crossoverRate \* 100)):

*#copies over the genes from the population at position x to the population at position y.*

self.population[positionY][i] = self.population[positionX][i]

*#mutates the genes of the population at position y.*

self.mutate(positionY)

**else**:

*#if the position y fitness value is higher a for loop will run 10 times.*

**for** i **in** range(10):

*#initializes a new variable and sets it to a random value between 1 and 100.*

chance = random.randrange(1, 100)

*#checks if the randomly generated number is lower than the crossover rate*

*#this sets the chance of the genes being or not being copied over.*

**if**(chance < (self.crossoverRate \* 100)):

*#copies over the genes from the population at position y to the population at position x.*

self.population[positionX][i] = self.population[positionY][i]

*#mutates the genes of the population at position x.*

self.mutate(positionX)

*#manages the running of each mutation.*

**def** runFunction(self, loops):

*#new variable that will store and array.*

fitnessTrack = []

*#for loop that goes runs the passed in loops amount of times.*

**for** x **in** range(loops):

*#runs the survivalOfTheFittest.*

self.survivalOfTheFittest()

*#new variable that is set to 0.*

totalFitness = 0

*#for loop that will go through every individual within population.*

**for** x **in** range(self.quantity):

*#the fitness is being added to the totalFitness of the population at the position x.*

totalFitness += self.fitness(self.population[x])

*#total fitness is set to the average of the total fitness.*

totalFitness = totalFitness / self.quantity

*#adds the next fintess value into the aray.*

fitnessTrack.append(totalFitness)

*#returns the array with the total fitness.*

**return** fitnessTrack

*#initializes a new GeneticAlgorithm and sets the quantity, mutation rate, k and crossover rate.*

a = GeneticAlgorithm(10, 10, 10, 1)

*#initializes the run function and sets the loops which represents the generations.*

b = a.runFunction(100)

*#sets the y lable to be fitness*

plt.ylabel("fitness")

*#sets the x lable to be generations*

plt.xlabel("generation")

*#plots the graph*

plt.plot(b)

*#displayes the graph*

plt.show()

*#this will run the genetic algorithm class 10 times for 10 different results.*

**for** x **in** range(10):

a = GeneticAlgorithm(10, 10, 10, 1)

b = a.runFunction(100)

plt.plot(b)

plt.ylabel("fitness")

plt.xlabel("generation")

plt.show()

## Vehicles:

**import** **numpy** **as** **np**

**import** **matplotlib.pyplot** **as** **plt**

**def** run(T,initial\_pos,initial\_bearing,geno,plot\_flag):

*#runs a simple agent described by a geno and returns the trajectory*

*#the light is at [0 0]*

*#input*

*#T is the time you want run it for*

*#pos initial position*

*#bearing initial bearing (degrees) e.g [90; 90]*

*#geno genotype e.g [w\_ll w\_lr w\_rl w\_rr bl br]*

*#plot\_flag set to tru to output the figure*

*#output a 2-d vector with the x and y coordinates*

*#run a simple aget*

*#geno = np.array([-1, 0, 0, -.4, 1, 0.5])*

*#initial\_pos = [1,1]*

*#T=100*

*#out = simple\_agent.run(T,initial\_pos,90,geno,1)*

*# Initial setup*

dt=0.05;

R =0.05; *#radius*

b = 45; *#(degrees) sensor anglr*

print(geno[5])

*#conver geno params*

w\_ll = geno[0]; *#left motor to left sensor*

w\_lr = geno[1]; *#left motor to right sensor*

w\_rl = geno[2]; *#right motor to left sensor*

w\_rr = geno[3]; *#right motor to right sensor*

bl = geno[4];

br = geno[5];

sl\_pos = np.zeros((2,1));

sr\_pos = np.zeros((2,1));

rho=np.zeros((2, 1));

sensor\_gain = 1;

motor\_gain =1;

vl=0;

vr=0;

*#convert to radians*

initial\_bearing = initial\_bearing/360\*2\*np.pi;

b=b/360\*2\*np.pi;

pos = np.zeros((2,int(T/dt)));

bearing = np.zeros((1,int(T/dt)));

pos[:,0] = initial\_pos;

bearing[:,0] = initial\_bearing;

**for** i **in** range(1, int(T/dt)):

vc = (vl+vr)/2;

va = (vr-vl)/(2\*R);

pos[0,i] = pos[0,i-1]+ dt\*vc\*np.cos(bearing[0,i-1]);

pos[1,i] = pos[1,i-1]+ dt\*vc\*np.sin(bearing[0,i-1]);

bearing[0,i] = np.mod(bearing[0,i-1] + dt\*va,2\*np.pi);

*# Calculate left sensor position*

sl\_pos[0] = pos[0,i] + R\*np.cos(bearing[0,i]+b);

sl\_pos[1] = pos[1,i] + R\*np.sin(bearing[0,i]+b);

*# Calculate right sensor position*

sr\_pos[0] = pos[0,i] + R\*np.cos(bearing[0,i]-b);

sr\_pos[1] = pos[1,i] + R\*np.sin(bearing[0,i]-b);

*# Calculate (square) distance to element*

dl = np.sqrt((sl\_pos[0])\*\*2+(sl\_pos[1])\*\*2);

dr = np.sqrt((sr\_pos[0])\*\*2+(sr\_pos[1])\*\*2);

*# Calculate local intensity*

il = sensor\_gain/dl;

ir = sensor\_gain/dr;

lm = il\*w\_ll + ir\*w\_rl + bl;

rm = il\*w\_lr + ir\*w\_rr + br;

*# Scale by motor gains*

vl =motor\_gain\*lm;

vr =motor\_gain\*rm;

**if** plot\_flag==1:

plt.plot(pos[0,:],pos[1,:])

*#final postion*

x=pos[0,int(T/dt)-1];

y= pos[1,int(T/dt)-1];

f\_bearing = bearing[0,int(T/dt)-1];

*# Calculate left sensor position*

sl\_pos[0] = x + R\*np.cos(f\_bearing+b);

sl\_pos[1] = y + R\*np.sin(f\_bearing+b);

*# Calculate left sensor position*

sr\_pos[0] = x + R\*np.cos(f\_bearing-b);

sr\_pos[1] = y + R\*np.sin(f\_bearing-b);

plt.plot(0,0,marker='.',markersize=30,color='yellow');

plt.plot(0,0,marker='o',markersize=10,color='black');

*# Plot sensors*

*#*

plt.plot(sl\_pos[0],sl\_pos[1],marker='.',markersize=10,color='red');

plt.plot(sr\_pos[0],sr\_pos[1],marker='.',markersize=10,color='red');

*# Plot body*

plt.plot( x, y,marker='.',markersize=10,color='blue');

plt.plot(x,y,marker='o',markersize=10,color='black');

*#Plot trajkectory*

plt.show()

**return** pos

*#The code below will create a representation of a aggresor vehicle*

time = 200

initialPosition = [-5, -5]

initialBearing = 30

genotype = np.array([0, 1, 1, 0, 0, 0])

plotFlag = 1

out = run(time, initialPosition, initialBearing, genotype, plotFlag)

*#The code below will create a representation of a coward vehicle*

time = 200

initialPosition = [5, 5]

initialBearing = 200

genotype = np.array([2, 0, 0, 2, 0, 0])

plotFlag = 1

out = run(time, initialPosition, initialBearing, genotype, plotFlag)

*#The code below will create a representation of a lover vehicle*

time = 200

initialPosition = [-5, 0]

initialBearing = 30

genotype = np.array([-1, 0, 0, -1, 1, 1])

plotFlag = 1

out = run(time, initialPosition, initialBearing, genotype, plotFlag)

*#The code below will create a representation of a explorer vehicle*

time = 200

initialPosition = [-5, -5]

initialBearing = 30

genotype = np.array([0, -20, -20, 0, 2, 2])

plotFlag = 1

out = run(time, initialPosition, initialBearing, genotype, plotFlag)

*#The function below has been modified so the vehicle has one sensor (one eye) connected to the two wheels.*

**def** run2(T,initial\_pos,initial\_bearing,geno,plot\_flag):

*#runs a simple agent described by a geno and returns the trajectory*

*#the light is at [0 0]*

*#input*

*#T is the time you want run it for*

*#pos initial position*

*#bearing initial bearing (degrees) e.g [90; 90]*

*#geno genotype e.g [w\_ll w\_lr w\_rl w\_rr bl br]*

*#plot\_flag set to tru to output the figure*

*#output a 2-d vector with the x and y coordinates*

*#run a simple aget*

*#geno = np.array([-1, 0, 0, -.4, 1, 0.5])*

*#initial\_pos = [1,1]*

*#T=100*

*#out = simple\_agent.run(T,initial\_pos,90,geno,1)*

*# Initial setup*

dt=0.05;

R =0.05; *#radius*

b = 45; *#(degrees) sensor anglr*

*#conver geno params*

bl = geno[0];

br = geno[1];

sr\_pos = np.zeros((2,1));

rho=np.zeros((2, 1));

sensor\_gain = 1;

motor\_gain = 1;

vl=0;

vr=0;

*#convert to radians*

initial\_bearing = initial\_bearing/360\*2\*np.pi;

b=b/360\*2\*np.pi;

pos = np.zeros((2,int(T/dt)));

bearing = np.zeros((1,int(T/dt)));

pos[:,0] = initial\_pos;

bearing[:,0] = initial\_bearing;

**for** i **in** range(1, int(T/dt)):

vc = (vl+vr)/2;

va = (vr-vl)/(2\*R);

pos[0,i] = pos[0,i-1]+ dt\*vc\*np.cos(bearing[0,i-1]);

pos[1,i] = pos[1,i-1]+ dt\*vc\*np.sin(bearing[0,i-1]);

bearing[0,i] = np.mod(bearing[0,i-1] + dt\*va,2\*np.pi);

*# Calculate right sensor position*

sr\_pos[0] = pos[0,i] + R\*np.cos(bearing[0,i]-b);

sr\_pos[1] = pos[1,i] + R\*np.sin(bearing[0,i]-b);

*# Calculate (square) distance to element*

dl = np.sqrt((sr\_pos[0])\*\*2+(sr\_pos[1])\*\*2);

dr = np.sqrt((sr\_pos[0])\*\*2+(sr\_pos[1])\*\*2);

*# Calculate local intensity*

ir = sensor\_gain/dr;

lm = ir + ir + bl;

rm = ir + ir + br;

*# Scale by motor gains*

vl =motor\_gain\*lm;

vr =motor\_gain\*rm;

**if** plot\_flag==1:

plt.plot(pos[0,:],pos[1,:])

*#final postion*

x=pos[0,int(T/dt)-1];

y= pos[1,int(T/dt)-1];

f\_bearing = bearing[0,int(T/dt)-1];

*# Calculate left sensor position*

sr\_pos[0] = x + R\*np.cos(f\_bearing-b);

sr\_pos[1] = y + R\*np.sin(f\_bearing-b);

plt.plot(0,0,marker='.',markersize=30,color='yellow');

plt.plot(0,0,marker='o',markersize=10,color='black');

*# Plot sensors*

*#*

plt.plot(sr\_pos[0],sr\_pos[1],marker='.',markersize=10,color='red');

*# Plot body*

plt.plot( x, y,marker='.',markersize=10,color='blue');

plt.plot(x,y,marker='o',markersize=10,color='black');

plt.savefig("vehicles6.png")

*#Plot trajkectory*

plt.show()

**return** pos

*#The code below will create a representation of a one eyed phototaxis over short period of time*

time = 200

initialPosition = [5, 0]

initialBearing = 300

genotype = np.array([0.1, 0.2])

plotFlag = 1

out = run2(time, initialPosition, initialBearing, genotype, plotFlag)

*#The code below will create a representation of a one eyed phototaxis over long period of time*

time = 3000

initialPosition = [5, 0]

initialBearing = 300

genotype = np.array([0.1, 0.2])

plotFlag = 1

out = run2(time, initialPosition, initialBearing, genotype, plotFlag)