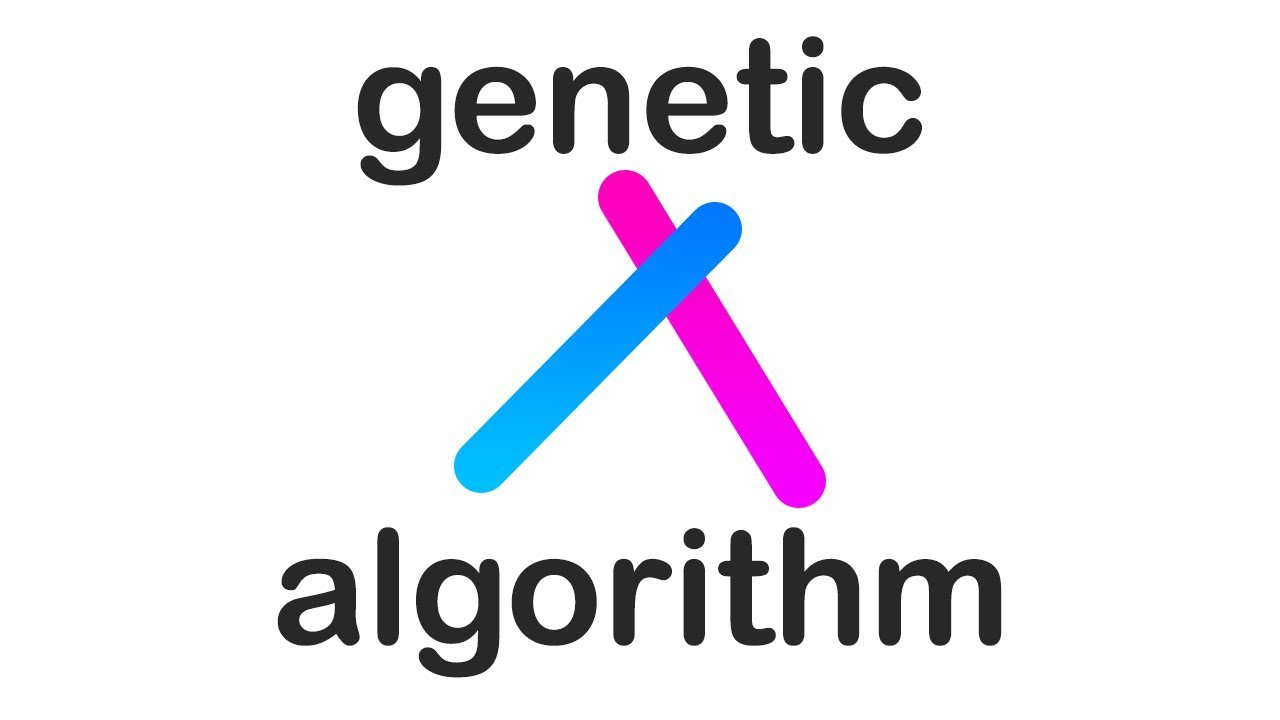
**Investigation into Genetic Algorithms.**



What is a Genetic algorithm?

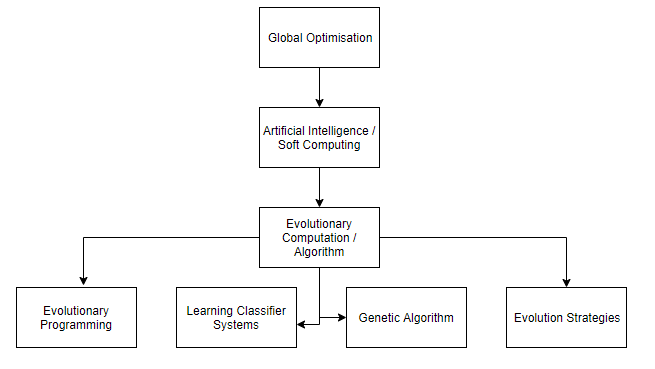
The term “genetic algorithm” refers to a specific algorithm implemented in a specific way to solve specific sorts of problems.

Genetic Programming (GP) is a type of Evolutionary Algorithm, a subset of machine learning. Evolutionary Algorithms are used to discover solutions to problems humans do not know how to solve, directly. Free of human preconceptions or biases, the adaptive nature of EAs can generate solutions that are comparable to, and often better than the best human efforts.

Inspired by biological evolution and its fundamental mechanisms primarily based on Darwin’s theory of evolution, Genetic Programming software systems implement an algorithm that uses random mutation, crossover, a fitness function, and multiple generations of evolution to resolve a user-defined task.

Disciplines of Computer Science connected to Genetic Algorithm:

The field of Evolutionary Computation encompasses several types of evolutionary algorithm. These include *Genetic Algorithms* (GAs), *Evolution Strategies*, *Evolutionary Programming* and *Learning Classifier Systems*.



History of Genetic Algorithms:

The first record of the proposal to evolve programs is that of Alan Turing in the 1950s. However, there was a gap of some thirty years before Richard Forsyth demonstrated the successful evolution of small programs, represented as trees, to perform classification of crime scene evidence for the UK Home Office.

In 1962, John Holland's work on adaptive systems laid the foundation for later developments. In 1975, Holland and his students published the book Adaptation in Natural and Artificial Systems based on the concept of Darwin’s theory of evolution; afterwards, his student David E. Goldberg extended GA in 1989. In 1992 John Koza has used genetic algorithm to evolve programs to perform certain tasks. He called his method "genetic programming" (GP).

Genetic Algorithm terminology:

**The fitness function** is the function you want to optimize. For standard optimization algorithms, this is known as the objective function.

Write the fitness function as a file or anonymous function and pass it as a function handle input argument to the main genetic algorithm function.

**Individuals**- An individual is any point to which you can apply the fitness function. The value of the fitness function for an individual is its score. For example, if the fitness function is

f(x1,x2,x3)=(2x1+1)2+(3x2+4)2+(x3−2)2,

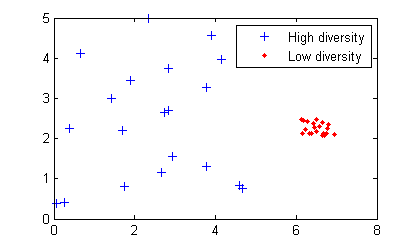
the vector (2, -3, 1), whose length is the number of variables in the problem, is an individual. The score of the individual (2, –3, 1) is f(2, –3, 1) = 51.

An individual is sometimes referred to as a **genome** and the vector entries of an individual as **genes / DNA**.

**Populations** and **Generations** - A population is an array of individuals. For example, if the size of the population is 100 and the number of variables in the fitness function is 3, you represent the population by a 100-by-3 matrix. The same individual can appear more than once in the population. For example, the individual (2, -3, 1) can appear in more than one row of the array.

At each iteration, the genetic algorithm performs a series of computations on the current population to produce a new population. Each successive population is called a new generation.

**Mutation**– mutation refers to the genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. In mutation, the solution may change entirely from the previous solution. Hence GA can come to a better solution by using mutation. Mutation occurs during evolution according to a user-definable mutation probability.



Mutation is essential to the genetic algorithm because it enables the algorithm to search a larger region of the space.

**Fitness Values and Best Fitness Values** - The fitness value of an individual is the value of the fitness function for that individual.

**Parents and Children** - To create the next generation, the genetic algorithm selects certain individuals in the current population, called parents, and uses them to create individuals in the next generation, called children. Typically, the algorithm is more likely to select parents that have better fitness values.

There are three main steps in a genetic algorithm:

1. The algorithm begins by creating a random initial population.
2. The algorithm then creates a sequence of new populations. At each step, the algorithm uses the individuals in the current generation to create the next population. To create the new population, the algorithm performs the following steps:
   1. Scores each member of the current population by computing its fitness value. These values are called the raw fitness scores.
   2. Scales the raw fitness scores to convert them into a more usable range of values. These scaled values are called expectation values.
   3. Selects members, called parents, based on their expectation.
   4. Some of the individuals in the current population that have lower fitness are chosen as elite. These elite individuals are passed to the next population.
   5. Produces children from the parents. Children are produced either by making random changes to a single parent—mutation—or by combining the vector entries of a pair of parents—crossover.
   6. Replaces the current population with the children to form the next generation.
3. Repeat until the user-defined termination criteria are met, such as number of generations, time limit and fitness limit.

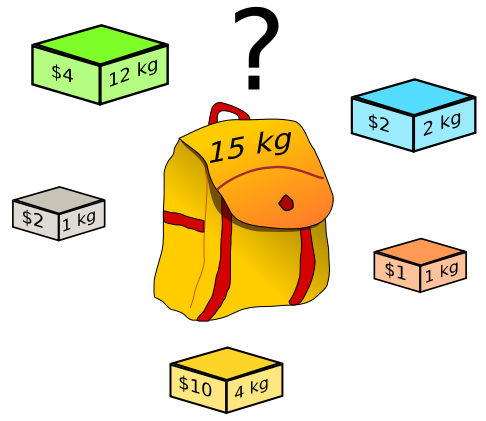
In GP, each generation is composed of a population of individual programs. Each program is a mathematical function that when executed against the given data, produces a value.

The initial population (generation 0) is composed of randomly constructed programs built upon user-defined arithmetic, Boolean operators, and user-defined operands, variables which represent specific data parameters. The quantity of programs in the initial and all subsequent populations is defined by the user and remains constant throughout the entire run.

The entire initial population is evaluated, meaning each individual program is executed against the given dataset, producing a single outcome per program. This outcome is measured against the known solution and recorded as the fitness score. From this initial population, individuals’ programs are randomly selected for a tournament in which the program with the best fitness score is copied into the subsequent generation, following application of a genetic operator:

1. With elite children, the program is copied without modification.
2. With mutation, a portion of the program is randomly modified.
3. With crossover, two programs are selected to contribute a portion of their own code to a new program.

Each mutation and crossover might result in a reduction or improvement in the fitness score for each individual program. As with the initial population, all programs in each new generation are evaluated, randomly selected for a tournament, and then evolved for the subsequent generation. Because those programs selected exhibit the highest fitness score, their contribution to the next generation enables an overall improvement. This process continues until the user-defined termination criteria are met.

Why do we use of genetic algorithms over the traditional deterministic algorithms?

In the world, Genetic Algorithms are used when traditional algorithms are not cost or time efficient in solving a problem. One of their main application is to find an optimum solution where the deterministic algorithms are too costly.

Genetic Algorithm are also used to provide good approximate solutions to problems that cannot be solved easily using other techniques. Many optimisation problems fall into this category.

It may be too computationally intensive to find an exact solution but sometimes a near-optimal solution is enough. In these situations, evolutionary techniques can be effective. Due to their random nature, evolutionary algorithms are never guaranteed to find an optimal solution for any problem, but they will often find a good solution if one exists. Such as the travelling salesman problem and the knapsack problem.

|  |  |
| --- | --- |
| Classical Algorithm | Genetic Algorithm |
| Generates a single point at each iteration. The sequence of points approaches an optimal solution. | Generates a population of points at each iteration. The best point in the population approaches an optimal solution. |
| Selects the next point in the sequence by a deterministic computation. | Selects the next population by computation which uses random number generators. |

The uses of Genetic Algorithm:

There are many applications of genetic algorithms and genetic programming. They include automotive design, computer gaming, robotics, investments, traffic/shipment routing and school/university scheduling. The table below shows different application of Genetic algorithms in different sectors:

|  |  |  |  |
| --- | --- | --- | --- |
| **Natural sciences, mathematics and computer sciences** | **Earth sciences** | **Finance, and social sciences** | **Industry, Management and Engineering.** |
| Data Centre/Server Farm. | Design of water resource systems. | Financial maths. | Container loading optimization |
| Electronic circuit design, known as evolvable hardware. | Estimation of heat flux between the atmosphere and sea ice. | Automated design of sophisticated trading systems in the financial sector. | Timetabling problems, such as designing a non-conflicting class timetable for a large university |
| Learning robot behaviour using genetic algorithms. | Modelling global temperature changes. | Design of anti-terrorism systems. | Mobile communications infrastructure optimization |

**ST5 X-band antenna:**

A real-life application of genetic programming is the ST5 X-band antenna. In 2006, a team of engineers at NASA used genetic algorithms and evolutionary programmes to mimic Darwinian natural selection process to find the perfect shape of the antenna for a certain radiation pattern and is much more efficient than a standard antenna (for example helix antennas), partly because of its asymmetrical shape. This antenna was later used in the 2006 NASA’s Space Technology 5 (ST5) mission.

The antenna had an unusual structure and was evolved to meet a challenging set of mission requirements, notably the combination of wide beam width for a circularly polarized wave to cover the up and down link frequencies at X-band.

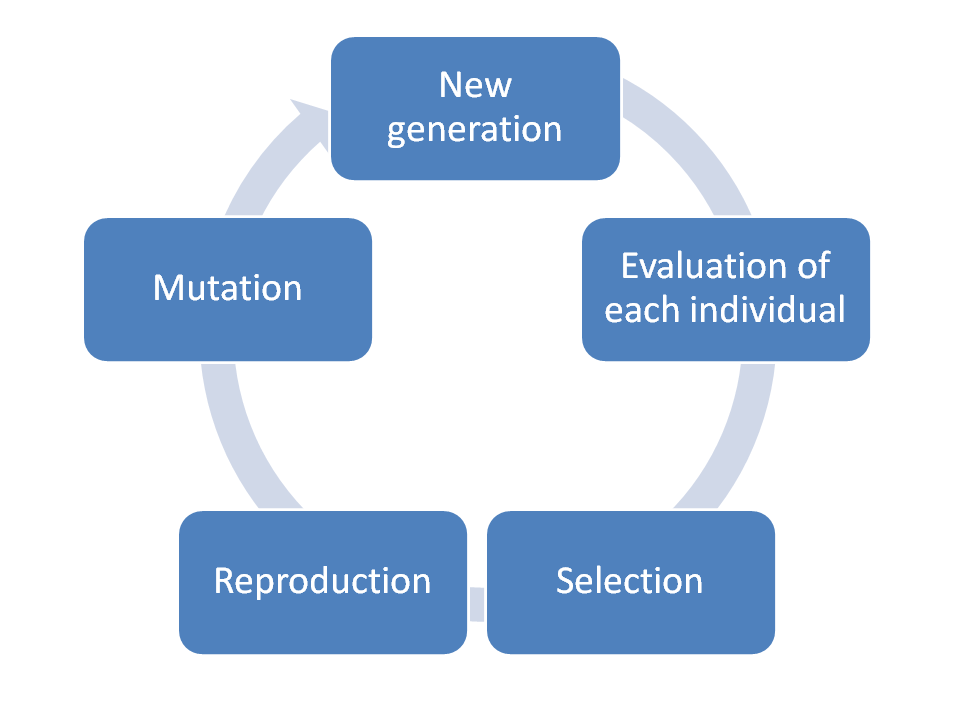
There are also other evolved antennae currently used in different industries as well. NASA also does research on Genetic Algorithms that design chips that fix themselves, circuits, coevolutionary algorithms and schedules for satellite fleets.

The process of making the ST5 X-band antenna:

The computer programme starts with simple antenna shape. It then adds or modifies elements in a semirandom manner to create a number of new candidate antenna shape.

These are then evaluated to determine how well they fulfil the design requirements, and a numerical score (fitness score) is computed for each element.

After a set number of generations, the population of antennas is evaluated, and the design with highest fitness score is chosen.

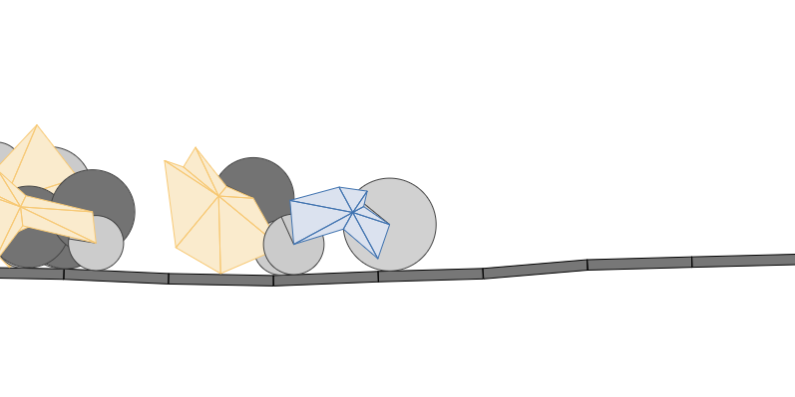
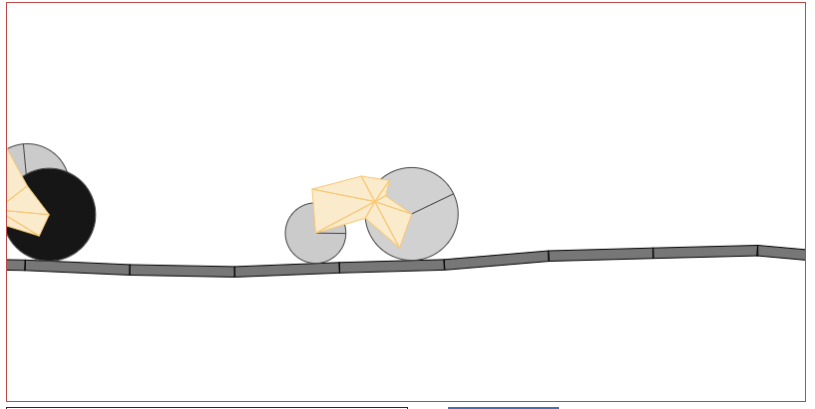


Using these antennas, the programme repeats the procedure using methods like mutation crossover and Elite children, generating a population of with higher fitness scoring designs.

Then, in a step like natural selection, a portion of the candidate antennas with the worst fitness scores are discarded, leaving a small population of the elements with the highest fitness scores.

The resulting antenna often outperforms the best manual designs, because it has a complicated asymmetric shape that could not have been found with traditional manual design methods.

**Genetic Cars 2D:**

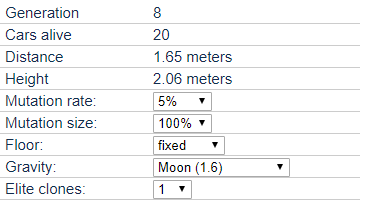
Another example of genetic algorithm being used is the Genetic Cars 2D. Genetic Cars 2D is a programme that uses genetic algorithm to learn how to build a car.  It starts with a population of 20 randomly generated shapes with wheels and runs each one to see how far it goes. The cars that go the furthest reproduce to produce offspring for the next generation. The offspring combine the traits of the parents to produce better cars.

The programme starts with 20 randomly generated cars within the parameters, which include maximum 3 wheels, and triangular shapes. The cars are then allowed to run on the track.

The cars are killed off when they stop moving or their velocity is negative. The cars are then evaluated and calculated their fitness scores based on their distance travelled for each element.

Similar to natural selection, the cars with the highest fitness function are crossed over and reproduce to form a new population in the next generations. Some cars with high fitness function will also be crossed over to the next generation. Some of the parameters such as the mutation rate and the mutation size can be modified by the user (mentioned below).

The step will be repeated until the number of generations the programme allowed to run is reached or the stopping criteria is met.



Generation: Tells us the current generation of the genome (car)

Cars alive: Give us the number of cars alive now. Cars are killed off when they stop moving or their velocity is negative.

Distance: Gives the distance travelled by the car from the starting point.

Height: Height of the car from the starting point.

Mutation rate: The chance that each gene in everyone will mutate to a random value when a new generation is born.

Mutation size: the range each gene can mutate into. Lower number mean the gene will have closer values to the original.

Elite clones: the n number of cars with the best fitness value will be copied be copied over to the next generation.

Floor: fixed or mutable for every next generation

Gravity: Earth’s acceleration is the default for the car on the road., but there also other planets gravity.

Different ideas for representing genetic algorithms and how they work:

* Using genetic algorithms to build a car.
* Using genetic algorithms to teach a simulated human to walk.
* Using genetic algorithms to teach a computer to programme.
* Using genetic algorithms to create a schedule for a school or an institute.

Genetic Algorithm has been a special focus in the field of computer science and global optimisation algorithms because of its capability to be implemented as a universal optimiser that could be used to optimise any type of problem.

**Genetic Rockets**

My project:

In order to demonstrate how genetic algorithms, work and how they can be used to optimise any kind of problems. I will be creating a programme called Genetic Rockets. Genetic rocket is a simulation of rockets trying to reach its target quickly and efficiently as possible.

Genetic rockets provide an elegant example of genetic algorithms. NASA uses evolutionary computing techniques to solve all kinds of problems, ranging from radio telescope scheduling to satellite antenna design to construction or rocket firing patterns. In a future of un-manned space probes, the long-term goal will be to have machines that can evolve to adapt to unexpected conditions. Such a machine might use genetic algorithms to evaluate a huge number of possible strategies, and to choose the one that is most likely to be a success. My simulation provides a realistic use of genetic algorithm.

What are Genetic Rockets:

Genetic Rockets is a programme that uses Genetic Algorithm to create the best or the optimum rocket firing pattern to reach its target. My project is going to be a simulation of how the rockets evolve to find the optimum rocket firing patterns that uses smallest amount of fuel to fly to its target (explained further below).

End user:

My end user for the simulation of the genetic rockets will be Mark Szolkowski. Mark is a mathematics teacher with programming expertise at my sixth form. The simulation of genetic rockets will help demonstrate to his students, how genetic algorithms work and how individuals evolve in every generation to get better at reaching the target.

Questionnaire:

1. Why do you need a simulation to demonstrate how genetic algorithms work?

I want to demonstrate to students how cool GA’s are! The fact that a computer can ‘evolve’ an acceptable solution to a complicated problem is something that still makes me say “Wow!” even though I encountered GA’s in 1993.

1. How much of the simulation do you want to be visualised?

I want to see how the rocket with the best fitness score is reaching the target. Also, how the rockets handle if there was an obstacle is introduced and how they converge to a solution. In this case to the target by altering their path. Even a graph to represent the evolution of the rocket would be good!

1. What parameters would you like to have the ability to change in the programme?

Population size; burn time; velocity of rocket; mutation rate would be ideal.

1. What is the initial population of the rockets?

Perhaps 100 would be a good start, but we might need to be able to adjust this if the algorithm does not converge to a solution quickly.

1. Would you like to have more children born from mutation or crossover?

Crossover, as there might be a risk with mutation of destroying optimal solutions.  Mutation is still useful though in order to help avoid settling on a plateau.

1. How would you want to input obstacles in the simulation?

I would like to draw different obstacle in the simulation dragging my mouse. May be even able to save a landscape obstacle and then loading them back on to the simulation

1. Would you like to have any information about the previous generation?

The data from the fittest population member, along with the best ever population member. Also, the worst fitness score and the type of rocket. As well as the mean score of the population.

1. Would you like to have adaptive mutation (explained below) as one of the functions you could turn on?

Yes

1. Would you like to change the target’s location?

Yes, this would help to visualise how the rocket would change and adapt to new constraints.

1. Is there any other functions you would like to have on the simulation?

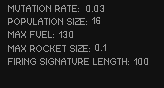
How about a graph displaying the best and low fitness score for each generation as well as a button to display fitness score for each rocket. As I said before, the option to save an load obstacles and also able to save and load a generation of rocket.

After the interview with my end user, we have decided what functionalities should be included, what parts of the genetic algorithm needs to be visualised and what is the main type of reproduction for the rockets.

Research:

There are some existing simulations of genetic rockets. One such example is the smart rockets simulation. Website: <http://www.blprnt.com/smartrockets/>

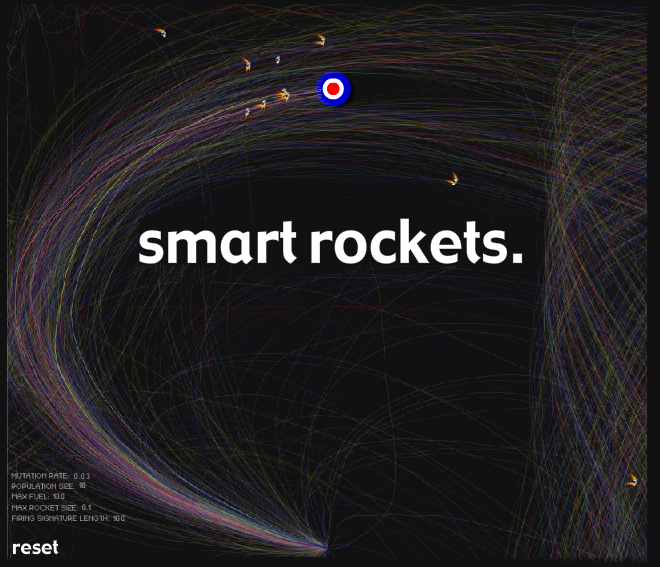
Each spaceship has five rockets. Those rockets can be placed at any angle and are variable in strength. Also, each rocket follows a firing pattern-allowing the spaceships to execute delicate manoeuvres.



One of the features that I really liked was the ability to change some of the parameters of the genetic algorithm such as the mutation rate, population size, max fuel and rocket size. In my simulation I hope to use a similar feature. However, there were no scale on the parameter editor. Which made it difficult to know what the limit is for some of the parameters. Therefore, in my programme I am going to use a slider with discrete values to change the parameters.

Another feature I like about the smart rocket simulation is the colour coded lines that depicts the path the spaceships flew. Every generation is given a specific colour and the spaceship that belongs to that generation will depict its flight path in that colour. This feature is useful because the users can see the mutation to the current rockets from the previous generation rockets. however, it displays the trail of every rocket and its hard to get any other details clearly. So in my simulation, the trail of the best rocket from each population will be displayed.

Another feature I like about the smart rocket simulation is the reset function where you can restart the simulation to see how a new set of random rockets would be able to travel to the target and compare them with the evolved rockets.



Origin point for the rockets

Target

obstacle

A problem that I encountered on this simulation was that the simulation didn’t display information regarding the fitness score of the rockets. The user has no information how the fitness score was calculated or how the fitness score changes as the rocket gets closer to the target. Therefore, in my simulation, the user will have the option to display the fitness score alongside the rocket to show how the score improves as the rocket gets closer to the target and decreases as it gets further form the target.

Another feature that I felt that this programme was missing was the ability to draw obstacles. Whilst this simulation does have the ability to switch between different obstacle layouts. It doesn’t have the ability to draw obstacle as per the user’s requirements. I believe this feature is essential to introduce a new obstacle in the middle of a simulation and then to visualise ho the rockets learn and evolve to avoid that obstacles and reach the target. I also believe that having the ability to change the location of the target would also be useful to visualise how the rockets mutate and evolve to reach the target. By moving the targets location, the fitness functions parameters would be changed and suddenly the worst performing rocket might be the best option to reach the target and become the strongest member of the population and produce more offspring’s. Just like in nature, where the species with a certain mutation would be able to survive and reproduce in certain conditions

Adaptive mutation:

|  |  |
| --- | --- |
| Number of Successful rockets: | Mutation Rate: |
| Population/4 or more | 9% |
| 0 | 70% |

Another problem I encountered with the smart rockets simulation and with other genetic algorithm simulation is that if the mutation rate is too low, the rockets struggle to mutate enough to reach its target. In order to solve this issue, I am going to introduce adaptive mutation where the mutation rate is raised to 70%. This allows the future generation to find the target not bounded by precedents. On the other hand, if the winning rockets are at least 25% of the population the mutation rate is then decreased to a value less than 0.9%.

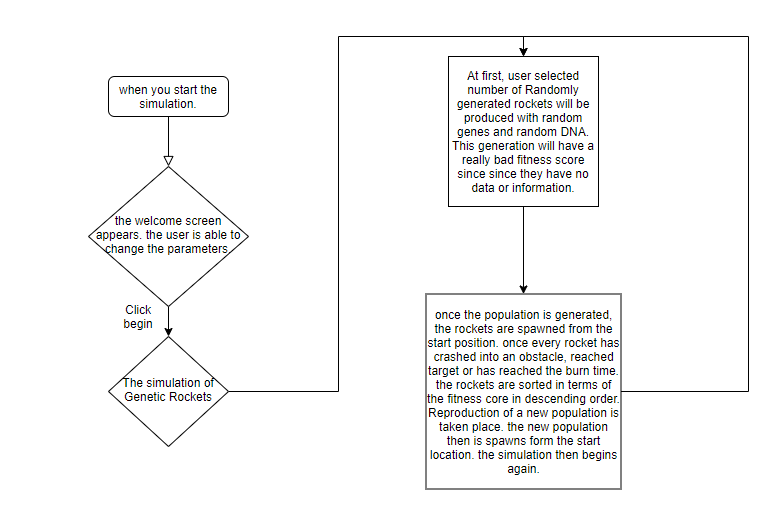
This function changes the parameter for the reproduction method. This would inevitably change how the rockets would be reproduced in the next generation. this would allow for a failing generation to randomly produce a successful offspring.

Genetic rockets:

The simulation begins with a welcome screen, where there’s a brief description how a GA works and how the genetic rocket simulation work and how they evolve and how they reproduce to give an idea of a GA functions.

The user will also be able to change some of the parameters of the simulation and then when the user clicks begin the simulation begins.

The simulation begins with Pre – selected number of randomly generated rockets starting from the bottom centre of the canvas. Their objective is to reach their target which might be blocked by obstacles. The rockets have limited amount of time to reach the target and when the time run out, the generation ends. Once the burn time is reached, crashed into an obstacle or has reached the target. A new generation begins. This process runs with no end.



The rockets:

The individual/genome of the simulation is the rockets. In the simulation there are different types of rockets with different number of thrusters and different placement.

At the start, there are 5 species of rockets:

Rocket with 1 thruster:

This type of rocket will have a single thruster attached to the rocket. The thruster will have a binary firing pattern, which would dictate if the rocket would fire in that frame. This species will fill 20% of the initial population.

Rocket with 2 thrusters:

This type of rocket will have 2 thrusters attached to the rocket at different angles. The thrusters will also have a binary firing pattern, which would dictate if the rocket would fire in that frame. This species will fill 20% of the initial population.

Rocket with 3 thrusters:

This type of rocket will have 3 thrusters attached to the rocket at different angles. The thrusters will also have a binary firing pattern, which would dictate if the rocket would fire in that frame. This species will fill 20% of the initial population.

Rocket with 4 thrusters:

This type of rocket will have 2 thrusters attached to the rocket at different angles. The thrusters will also have a binary firing pattern, which would dictate if the rocket would fire in that frame. This species will fill 20% of the initial population.

Rocket with 5 thrusters:

This rocket contains 5 thrusters which would be placed at different angles. Each thruster has a binary firing pattern which initially is random. Where the rocket would fire according to the sequential pattern. The firing pattern which will be part of the DNA, will controls the rockets movement and interaction in the simulation world. This species will fill 20% of the initial population.

The DNA of the rockets are the binary firing patterns of the 5 thrusters and the position of where the thrusters are placed around the rocket.

when the rocket reproduces after each generation, a species of rocket might randomly evolve into a new species. This is similar to in nature, where a single ember of species randomly evolves into another new species. For example: the rocket with 4 thrusters might randomly mutate to a Rocket with 5 thrusters or a Rocket with 5 thrusters might mutate into rocket with 6 thrusters.

So, there’s no limit for how many types of rockets are there in a generation.

The kill condition for the rockets:

* If the rockets crash into an obstacle.
* If the rocket flies outside of the simulation window.

The winning condition for the simulation is when the rocket reaches the target.

Once every rocket has either been killed off, or has reached the burn time or has reached the target the simulation starts again with a new population.

The fitness function:

The fitness score of the rocket is calculated for every genome i.e. rocket, the fitness score is calculated by taking into factors like the distance from the target, the time taken for the rockets to reach the target and if the rocket crashed into an obstacle. The fitness function is calculated for every genome in the population at every frame.

Fitness function = (1 000/ (distance from the target))^0.8.

If the rockets reach the target: Fitness function = fitness + 10/frame count

Else if the rocket crashed into obstacle: fitness/2

If the rockets reach the target, the fitness function takes into account for the amount of how many frames have passed, so the rocket that reached the target quicker will have a higher fitness score then the rockets that reached the target later. I have used this equation to calculate the fitness function for the rockets since they provide a convergence.

This graph represents the equation how the fitness function varies as distance from the target increase.

Distance from the target.



If the rocket is killed off by crashing into an obstacle, the fitness score is divided by 2 so that it has a lower chance of being chosen as the parent.

Reproduction:

Reproduction option controls how the programme produces the population for the next generation. The options are:

* **Elite count – the** number of individuals with the best fitness values in the current generation that are guaranteed to survive to the next generation. The default elite count is 10% of the population (any species).

The genetic algorithm uses the individuals in the current generation to create the children that make up the next generation. Besides elite children, the algorithm creates:

* Crossover children by selecting vector entries, or genes, from a pair of individuals in the current generation and combines them to form a child. The crossover fraction will be set to 50% of the population. The rockets can crossover with other species. During the crossover procedure, for each child rocket, 2 parent rockets are selected from the previous generation. The chance of a rocket being selected as a parent is directly proportional to the fitness score of the rocket. When 2 rockets are selected, both of their DNA are crossover to make a rocket that has the properties of both of the rocket.

For EX:

A rocket with 3 thruster and a rocket with 2 thrusters are chosen to cross over:

The child rocket:

The child rocket produced will be the species that has the greatest number of thrusters of the parents.

* The rest of the population will be made up of mutated children. When, the rocket with the best fitness score from the previous generation is chosen and then applied random mutation to produce offspring for the next generation. The magnitude of the difference between the parent rocket and the offspring will depend on the mutation rate.

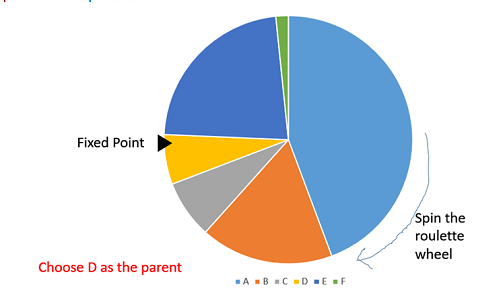
Selection:

The basic part of the selection process is to stochastically select from one generation to create the basis of the next generation. The requirement is that the fittest individuals have a greater chance of survival than weaker ones. This replicates nature in that fitter individuals will tend to have a better probability of survival and will go forward to form the mating pool for the next generation. Weaker individuals are not without a chance.

The selection process chooses parents for the next generation based on their fitness score.

One of the selection processes is called **fitness proportional fitness** value also called the roulette wheel selection. It is a genetic operator that is used to select an individual from a population. The fitness function associates a fitness value to each genome in the population.

Consider a circular wheel. The wheel is divided into n pies, where n is the number of individuals in the population. Each individual gets a portion of the circle which is proportional to its fitness value.  A fixed point is chosen on the wheel circumference as shown and the wheel is rotated. The region of the wheel which comes in front of the fixed point is chosen as the parent. If a second parent is needed, the same process is repeated.



It is clear that a fitter individual has a greater pie on the wheel and therefore a greater chance of landing in front of the fixed point when the wheel is rotated. Therefore, the probability of choosing an individual depends directly on its fitness.

+ The genome with the highest fitness score will have the greater chance to be selected as a parent. Since they occupy more space/ range.

* Often, it’s not fair because the rocket with the lowest fitness score will have very small probability of population.

Another form of selection is **Linear Rank Selection**. It is based on the rank of individuals rather than on their fitness. The rank n is accorded to the best individual whilst the worst individual gets the rank 1. Thus, based on its rank, each individual I has the probability of being selected given by the expression:



+ The diversity is preserved, since all the genome has a chance to be chosen.

- This method can lead to slower convergence to an efficient solution since the chances do not vary much from the best genome

There are also many other different types of selection process, which could be used. However, I have decided to use the fitness proportionate selection as the selection process for genetic rockets because this process prevents too quick convergence.

Parent for mutated children:

Once the fitness score has been calculated, the rocket with the highest fitness score is chosen to be mutated.

Parents for the crossover children:

The parents for crossover children varies, the rocket will use one of the selection function to choose both of the parents form the population. Each couple will produce a child for the next generation.

For example: for the population of 100 rockets, 50 of the children will be reproduced by the crossed over method. So, 100 randomly chosen rockets are crossed over to produce 50 crossover rockets.

The elite children:

The 10% of the population with the highest fitness score will also be carried over to the next generation.

Features of the simulation:

For the simulation to fully depict the fundamental parts of the Genetic Algorithm, the user will have the ability to change some of the parameters. The functionalities that the user can expect from the programme are:

* Ability to change the mutation rate. The amount of mutation happening to the DNA of the rocket. To change the mutation rate, I’ll be adding a slider with discrete values from 0 to 1. This rate should be set low. If it is set too high, the search will turn into a primitive random search.
* Ability to change the population size. To change the population size, I’ll also be adding a slider which ranges from 50 to 500. The reason I have 500 as the maximum population size because of the limited computational power and if there were to have more rockets the programme will start to slow down and crash.
* Ability to change the maximum burn time available for the rockets to fire. To change the maximum burn time, I will also be using a slider ranging from 5 to 100 seconds.
* Ability, to change the velocity of the rockets, which ranges from 0 to 10. Which means when the velocity is equal to 0, the rockets will be frozen.
* Ability to toggle the adaptive mutation on or off. I am going to include the function to disable or enable the adaptive mutation function because adaptive mutation is not one of the options available on a traditional Genetic Algorithm.
* Ability to change the location of the target by simply left clicking on the location where the user wishes the target to be moved.
* Ability to see the fitness score of each rocket on the simulation window and able to turn them off.
* To draw different obstacle types.
* To be able to distinguish the best rocket in a population at any given time.
* To be able to save and load an obstacle layout.
* To be able to save and load a generation of rockets with the obstacles as well as data for the graphs (explained below).

The simulation also will have an information tab where it’ll display the following information about the previous generation:

* No. of successful rockets that reached the target
* Highest fitness score
* Lowest fitness score
* The mean fitness score
* The current generation of rockets
* Which species had the highest fitness score
* Which species had the lowest fitness score

The simulation will also be able to display the best, mean and the low fitness score in a graph for each generation.

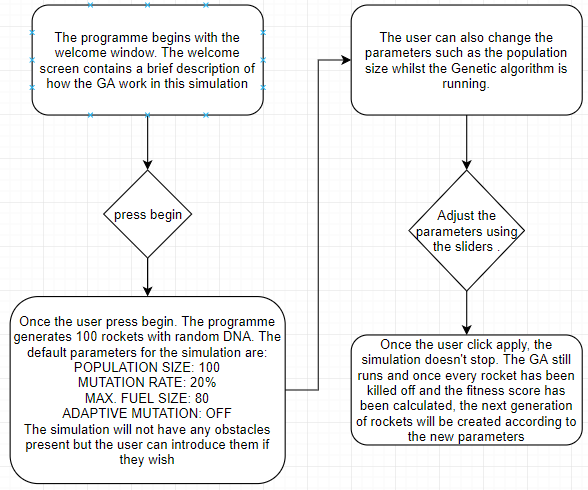
The burn time:

The MAX. Burn time is one of the parameters that the user can change in the simulation. The burn time controls how long the thrusters can fire for, once the lifetime of the rocket exceeds the MAX. Burn time the rocket dies. The MAX. Burn time is equal to the life spam of the rocket.

Saving a generation:

when the save function is called, the entire information of the generation will be saved on to a text file. The saved file will contain the velocity of the rocket, the best, low and the mean fitness to produce a graph that is identical to the graph that is identical to the one that is being saved, as well as the list of obstacles in that generation so that the condition of the loaded generation is identical to the one that was saved. Finally, the DNA and each gene of the rocket for every rocket is also saved. So, when the simulation is loaded, its identical to when it was saved. This is done by creating a new population with rockets using the DNA and the gens from each rockets in the file.

Flow chart to represent how the simulation functions:



Once the user press begin, the programme generates 100 rockets with random DNA. The default parameters for the simulation are:

POPULATION SIZE 100

MUTATION RATE: 20%

MAX FUEL SIZE: 100

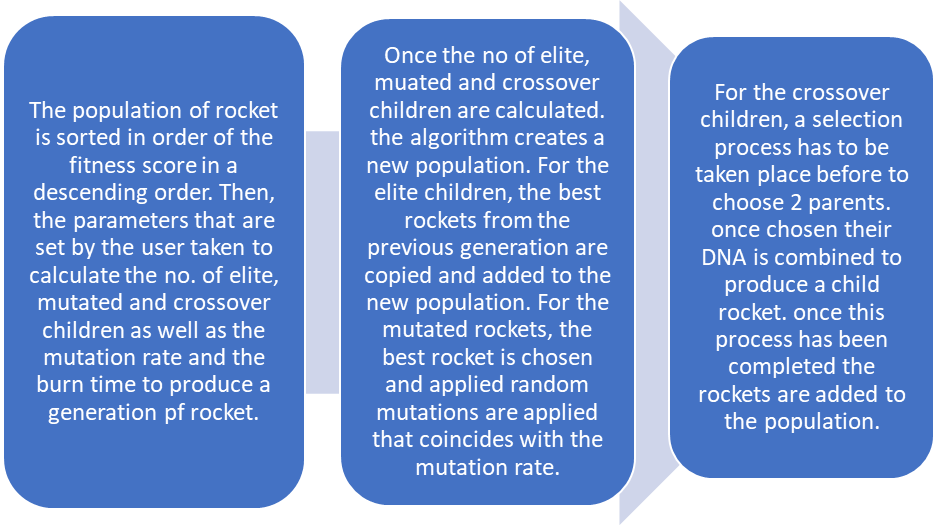
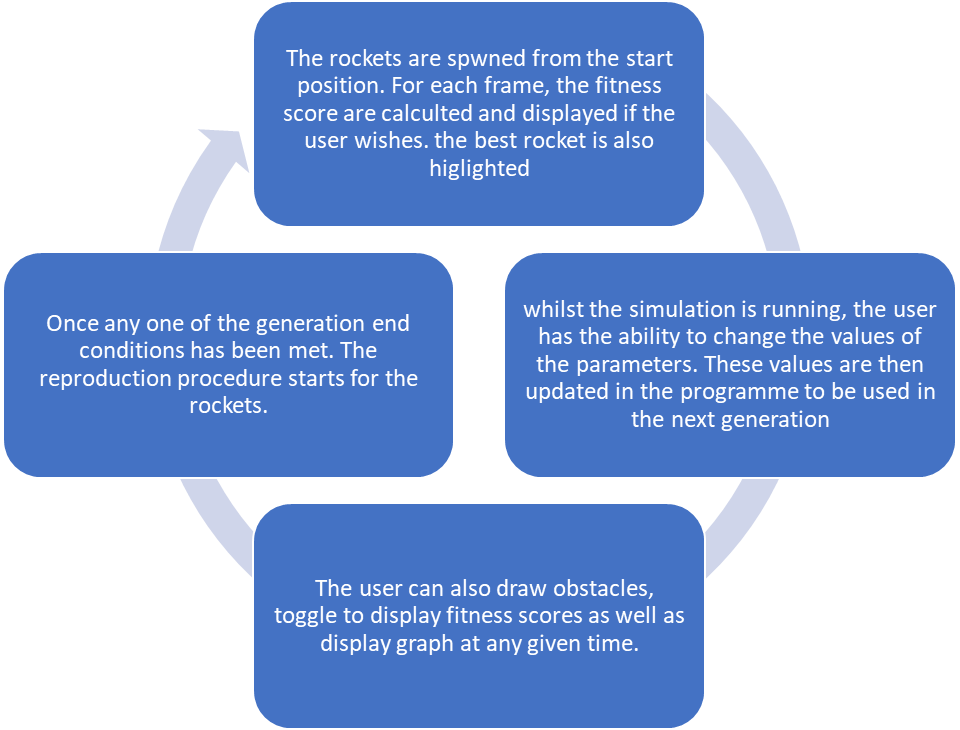
ADAPTIVE MUTATION: OFF

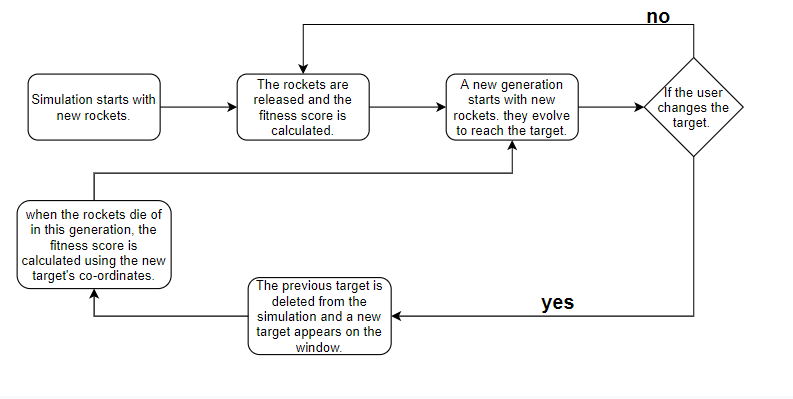
The simulation will not have any obstacles

present but the user can introduce them if

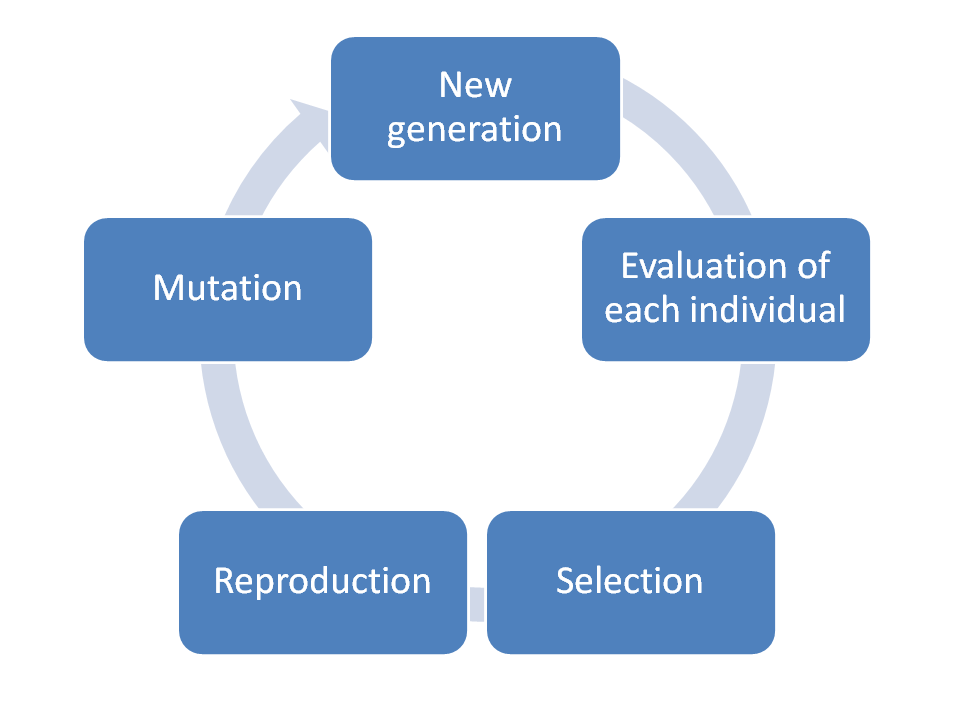
they wish

How the reproduction procedure takes place:



How the target moving effects the simulation:

The process of making the perfect rocket to get to the target:



The rockets are then evaluated to determine how close they get to the target and how much fuel they use up. Using these data, a fitness score is computed for each individual.

Then, a step like natural selection, the rockets with the worst fitness scores are discarded, leaving a small population of the elements with the highest fitness scores. Default rocket remaining will be 5.

Using these rockets, the programme repeats the procedure using methods like mutation crossover and Elite children, generating a population of with higher fitness scoring designs.

Once the children for the next generation is produced, they are released into the simulation and the GA continue.

The programme randomly generates 100 rockets with random DNA. The rockets are then released into the simulation with the default parameters

Objectives:

My project is focused on investigating and demonstrating how GA’s work. In order to demonstrates how GA’s work and how they stochastically reach a solution, I’ll be making the genetic rocket simulation. This simulation will portray all the necessary properties of a GA.

Summary of Properties of a GA includes:

1. Completely random spawning of a population.
2. Assessing the fitness of each genome in the population as well as to be able to adapt its fitness function to certain circumstances.
3. Then ordering the population of genomes in order of the best fitness score.
4. When, a generation end criterion has been met.
   * To be able to reproduce a new population using reproduction methods: elite, mutation, and crossover.
   * In order to be able to reproduce using the above-mentioned methods, a selection process has to be taken place.
   * Once the selection process has been completed, the reproduction of a new population takes place.
5. Once a new population is created, the genomes are spawned again in the simulation and the process starts again.
6. The data collected from each generation should be evaluated and compiled to make sense for the user.

Objectives in terms of Genetic Rockets:

1. Create a home page for the simulation, which includes:

* The title and a brief description of how the GA works.
* The ability for changing the parameters and enable some functionalities of the simulation in the home page.

1. To design the simulation page which displays the:

* Simulation of genetic rockets
* Details about the previous generation. Which calculates and displays the as the number of successful rockets that reached the target, mean fit, best, low fitness core and which species of rocket had the highest and the lowest fitness score.
* Ability to adjust the parameters of the simulation.

1. To successfully implement features of the simulation (mentioned above).

* To have an active timer that counts down to 0 from the burn time and when the burn time reaches 0. The generation starts again.
* A graph to display the trend of how the algorithm has evolved to produce a better generation of rockets. The graph will include:
* Best fitness score for each generation
* Low fitness score for each generation
* Mean fitness score for each generation
* As well as a key.

1. Creating a completely random spawning of rockets:
   1. Taking the population size and producing a equal number of rockets from each species.
   2. To create different species of rockets with different number of thrusters ranging from 1 to 5. However, there should be no limit to how they evolve to gain more thrusters.
   3. To Design a rocket which contain different number of thrusters depending on the species of the rocket, With the movements being completely randomised for the 1st generation.
   4. To be able to make the rockets move, according the DNA of the rockets. Each rocket must be represented accurately on the simulation window. The representation of the rocket must display the critical details of the rocket. Such as the thruster and which angles they are placed around the rocket. They should also be able to display which of the thrusters are firing at a given time and which are idle.
   5. As well as to make/ draw obstacles as well as to detect any collision between the rockets and the obstacle and then destroys them.
   6. Create a target that’s movable.
   7. To be able to kill of the rockets when the killing conditions are met and highlight them from the simulation screen.
2. Assessing the fitness of each Rocket in the population.
   1. For each rocket, a fitness rocket should be calculated depending on factors including:
      1. distance from the target
      2. the time taken for the rockets to reach the target
      3. if the rocket crashed into an obstacle, these variables should affect the fitness score for each rocket.
   2. The fitness function should be calculated at each frame, and the best rocket will also have to be highlighted to the user. The fitness function of each rocket should also be displayed if the user wishes.
3. For the best rocket to be highlighted and for the reproduction process to begin, the rockets has to be sorted in descending order of the fitness score. To sort the rockets using a sorting algorithm that has the optimum/ minimum time complexity for a large list. The trail for the best rocket from each generation should also be displayed, this would be helpful for the user to see how the best rockets movement differ from other rockets.
4. To be able to determine when one of the criteria for ending a generation is reached and then proceed to the next stage:
   1. To reproduce a new generation of rocket using elite carry, mutation and cross over.
   2. For elite reproduction: once the population of rockets are sorted in order, 10% of the rockets in the population should be chosen to be carried over into the next generation.
   3. For mutation: choosing the best rocket in the population, and then applying random mutations to its DNA, to create a new set of rockets to make up 40% of the new population. The random mutations applied to the parent rocket to create an offspring will be limited by the mutation rate which can be changed by the user. There is also a 2% chance that the rocket might evolve into a new species.
   4. For crossover:
      1. To be able to successfully implement fitness proportionate selection to select the parents for reproducing through crossover.
      2. Once 2 parents are chosen to be crossed over, their DNA should be retrieved and combined to make a new child with the combined properties of both the 1st and the 2nd parents. The parents chosen doesn’t necessarily have to be the same species either.
      3. To be able to inherit properties from the parent with the highest fitness score if there is an opposing value for the DNA with the parents.
5. Once the population has been built, the simulation starts again with the new population.
6. When the generation has ended, the data from each rocket has to be used to calculate mean fitness, best and low fitness. It is also used in reproduction for crossover, during fitness proportionate selection.

* To be able to save and load an obstacle layout.
* To develop adaptive mutation and give the user the option to use it. To show how the mutation rate would change when a condition of adaptive mutation is met.
  + To change the mutation rate to 9% if the number of rockets reached the target is greater then 25% of the population and the mutation rate to be changed to 70% if no rockets reached the target.
* To be able to save and load a population of rocket. When they are saved they should be able to save the information about the graphs and about the obstacles (if there were any) and when loading to be able to load the entire population of rockets as well as the graph of the save simulation and the obstacles.
* To change the parameters of the simulation using a mouse instead of keyboard as my end-user can stand near to the interactive board and use the touch screen to change the arguments.

**Documented design.**

Overall system:

As mentioned in the analysis, I’ll be designing a simulation to investigate how a genetic algorithm works using a simulation of genetic rockets. Where they evolve to get better at reaching its target.

The simulation requires a Homepage and a simulation page:

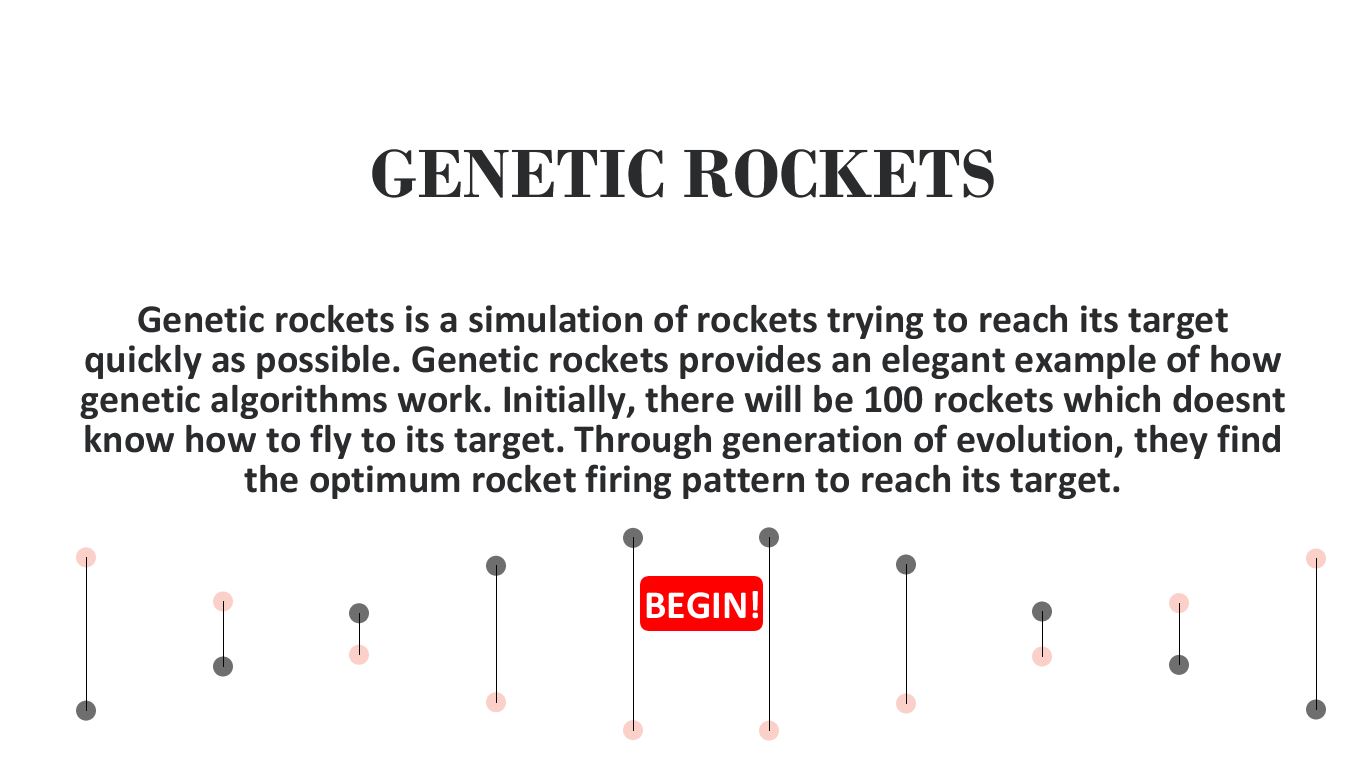
Homepage:

This page will act as a way of entry towards the simulation, meaning that before the simulation begins the user is greeted with the page, which contains the title “Genetic Rockets”, a brief explanation of how the Genetic Algorithm functions in terms of the rockets and a begin button.

The homepage must contain:

* The title and a brief description of how the GA works.
* The ability to change the parameters by the user:
  + To vary the population size.
  + To vary the mutation rate.
  + To vary the velocity of the rockets.
  + To vary the burn time of the simulation.
  + To enable adaptive mutation.
  + To display the fitness values of the rocket.
  + To display the graph.
  + To change between obstacle types
  + To save and load obstacle layouts.
  + To save and load a generation of rockets.
* A begin button to start the simulation.
* A reset button that returns the simulation back to its homepage.
* The user will also have the ability to hide the sliders and the buttons when he/she wishes.

Prototype 1:



The title of the simulation

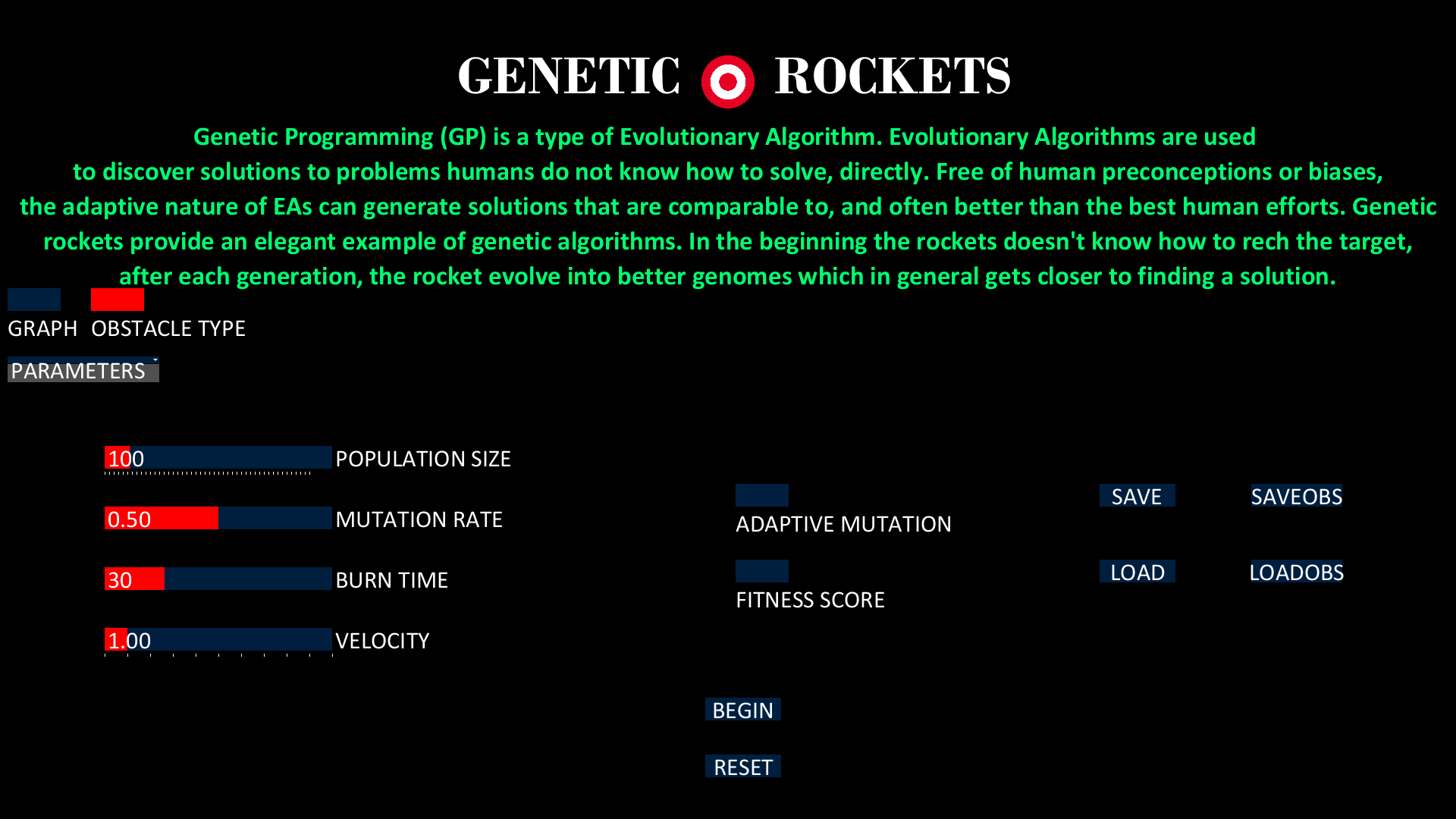
A brief explanation of how genetic rockets will work.

When the user clicks the begin button the user will be transported into the simulation page and the simulation of genetic rockets will begin.

An animation of a double helix DNA.

Whilst this design of the homepage contains a description of GR’s and has a begin button as well as the title, it doesn’t fulfil all the requirements of the homepage, also the description must be changed since it doesn’t fully give a brief description.

Prototype 2:



The title of the simulation

A brief explanation of how genetic rockets will work.

An image of the target.

Parameters that can be changed by the user.

A save and load button to load and save function for a generation of rockets and obstacle layouts.

Toggle button to display graph as well as change between obstacle types

2 buttons: begin and the reset button.

This tab allows the user to hide all the functionalities, except the graph and obstacle type toggle button.

The 2nd prototype design satisfies all the condition and the features needed on a homepage. This design has the feature for the user to change, a brief description of GR’s and a tab to hide all the parameters if the user wishes.

A close up of text on a whiteboard

Description automatically generated

Simulation page:

This page will be where the simulation takes place. The simulation window has 4 parts:

* Simulation
* Previous generation details window
* User functionality window (same as the homepage)
* Graph

Simulation Window: The simulation window will occupy the entire screen. The simulation window is where the users see how the rockets evolve to reach the target. At the beginning of the simulation, user defined number of random rockets will spawn out from the centre bottom of the window. The rockets will try to reach the target, which is initially at the centre top of the window. The rest of the section such as graph user functionality window, etc will be on top of the window.

Previous Generation Details window: The information box will contain information such as mean fit, lowest fitness and best fitness score from the previous generation. It will also contain which rocket had the best and the lowest fitness rocket form the previous generation. It will also have a timer which will display how long is left before the generation as well as the generation count. The information box will be layered on top of the simulation window. The box will be placed on the top left corner. The box will be transparent, which allow the user to see the simulation with minimal visual obstruction and the text will be written in bold to highlight it for the user.

User functionality window: The user functionality window will contain the functionality to allow the user to adjust the arguments for parameters like population size as mentioned above. Just like the homepage, the simulation page houses the ability to change the parameters of the simulation as well as enable some of the features of the simulation. I will be using a processing library “CONTROLP5” to display the sliders to change parameters, toggle buttons to enable the functionalities for the simulation and etc (explained below).

Graph: As mentioned before, the user can see a graph containing information regarding the previous generation of rockets. In the graph, fitness score will be plotted against the generation number. The plot will have multiple layer for the best, low and the mean fitness score. For this feature I’ll also be using a processing graphing library to plot the points on a graph.

Libraries:

CONTROLP5:

ControlP5 is a GUI library made for processing. I’ll be using this library to programme:

* Slider for population size ranging from 50 to 500. That can only be incremented by 10.
* Slider for mutation rate ranging 0 to 1.
* Slider for burn time ranging from 5 to 100.
* Slider for velocity 0 to 10. Which will increment in 1.
* Toggle button to turn on/off adaptive mutation rate.
* Toggle button to display or hide fitness score of the rockets.
* 2 buttons to save and load a generation of population.
* 2 buttons to save and load a layout of obstacles.
* A button to begin the simulation.
* A button to reset the simulation.
* A text field to display a brief description of the toggle buttons and what they are supposed to do.
* The above-mentioned controllers must be also be able to hide from the homepage and the simulation page. I will be using a group controller to group all the controllers together under one controller. Which can be used to hide the controllers form the page.

There are also 2 more controllers that is exempt from the group they are:

* Toggle button to display the graph.
* Toggle button to switch between obstacle types.

These features will be the same for the homepage as well as the simulation page.

The parameter values will then be updated with the values in the sliders.

GRAFICA:

I’ll be using a processing library called “grafica “to plot the graph with the co-ordinates calculated.

GRAFICA is a configurable plotting library for processing. Once a generation has been ended and the population has been sorted. The best fitness score is retrieved from the rocket that’s 1st on the list and the lowest fitness score is retrieved from the rocket that’s lowest on the list. The mean fitness value is also calculated. This is done by adding the fitness score together and then dividing by the population size. Once the values are calculated, they are added to 3 lists. One for lowest fitness, another for mean fitness, and another for highest fitness. This will be done for each generation.

Then, each value is the plotted against the generation number it was recorded in.

Since there are 3 different values for each generation, I’ll be using 3 different layers in a single plot to display different values. Each layer will be designated for each type of fitness.

Then the plot will be drawn on the simulation page. The plot will also display the key for the user to understand what the values mean, axis title as well as a title.

A close up of text on a whiteboard

Description automatically generated

Overall design:

In order to make my Project more organizable and to make the code work properly, I have split the code into 10 classes. This allows me to work on one part of the simulation at a time and then once I have completed one part of the code, I can move on to the next part. By separating into 10 classes, I can easily identify any errors in the code and fix them easily.

**Simulate:**

Whilst all the other parts are a class. Simulate is the main section where I’ll be programming the GUI, setup the programme and draw the programme.

This section will be the critical main section of my simulation. Here, I will import all of my libraries as well as create an instance of the library, then create the necessary variable for the libraries.

In this section, the simulation is also initially setup, so some of the variables as well as data structures will be instantiated. As well as any images that’s needed to be loaded into the simulation will be loaded.

The screen size as well as the background and the framerate will also be declared in the setup sub-routine.

This section also contains the draw subroutine as well as many sub-routines needed by the controllers for the user interface.

Draw is a subroutine which is Called directly after setup(), the draw() function continuously executes the lines of code contained inside its block until the program is stopped.

The number of times draw() executes in each second may be controlled with the frameRate() function. In this section, I will be able to draw and display the homepage as well as the simulation window.

A close up of text on a whiteboard

Description automatically generated

Saving a population:

The saving subroutine is also located in the simulate section.

When the save button is pressed:

A similar thing will be done when saving the obstacles:

* Obstacles in the simulation will be stored as strings with an initial character describing the type and four values .

Load:

This subroutine will be used to load the population saved in the text file.

A close up of text on a white background

Description automatically generated

**Global:**

Global is a static class that can be acceded from any parts of the code, without creating an object in each section of the code, where the variables are used.

|  |
| --- |
| Class -Global |
| IDCounter - static int |
| Start - static Vector |
| Target - static Vector |
| Burn\_time - static int |
| Velocity - static float |
| population\_size - static int |
| mutationRate - static float |
| Generation - static int |
| Adaptive - static Boolean |
| best\_Fitness - static FloatList |
| low\_Fitness - static FloatList |
| mean\_Fitness - static FloatList |
| rocImage - static PImage |
| currentTime - static int |
| T – static ArrayList<ArrayList<PVector>> |

the IDCounter is used during adding a rocket to a population and assigning a value to the id of the rocket. Which are accessed from the reproduction and the population class.

The start and the target vectors are used in the simulate section of the code as well as in the rocket class. So the need for the vectors to be global will be essential.

The burn time will also be used in many classes such as: Gene, DNA, Reproduction as well as the simulate section.

Velocity, mutationRate, population\_size and burn\_time are variables which are used in different sections of code, they are also variables that can be changed by the user in the simulation.

Generation is a count of the current generation.

Adaptive is a Boolean value that stores if the adaptive mutation is enabled.

best\_Fitness , low\_Fitness¸ mean\_Fitness are list used for adding the best mean and the lowest fitness values from each generation that will be used to plot a graph.

rocImage = holds an image of the rocket, which will be used in the rocket class

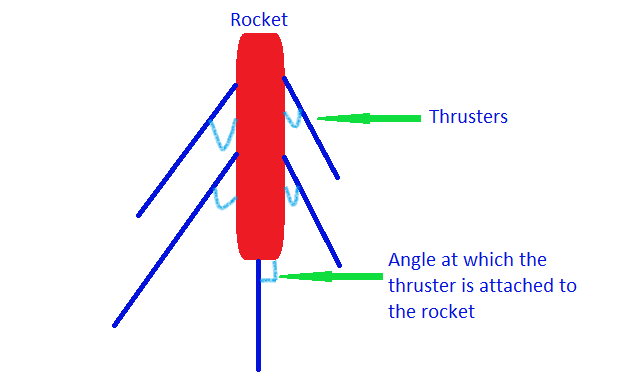
currentTime is the time since the generation started. Which is calculated in the simulate section and used in the movement and the display section of the rocket class.

T – is a list of list of Vectors. This list of list of vectors is used on the simulate section of the code to add the vector list of the trail of the best rocket, after each generation ends. This is then used in the draw subroutine to draw a trail for the best rockets int previous generation. The list adds a new entry after the end of each generation.

**Gene:**

The first stage of my programming is to programme the Gene of the thruster. The Gene is the most important part of the simulation and the Genetic Algorithm. This class will be used to create the Gene for a thruster. Each species have a different number of Genes. For example, a rocket with a single thruster will have a single a Gene as a part of its DNA and a rocket with 5 thrusters will have 5 Genes as a part of its DNA.

|  |
| --- |
| Class - Gene |
| “thruster” – string |
| “angle” – int |
| + getThruster() : string |
| + getAngle() : int |
| + showdetails() |
| +Gene() |
| +Gene(String, int) |
| +savedetails() |

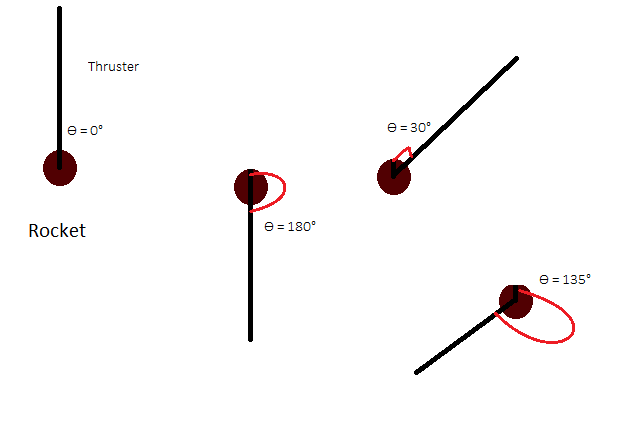
The string “thruster” is used to generate random binary firing pattern for the thruster. The string will be made up of randomly generated binary sequential patterns. The binary firing pattern, which is made up of 1’s and 0’s. The string length is equivalent to the burn time. As this dictate how long the rocket will be alive. As mentioned in the analysis, the unit for burn time is seconds, so the firing pattern dictates whether to fire the thrusters and apply a force to make the rocket move in the specific direction every second.

The diagram portrays a rocket with 5 thrusters

For example: if string “thruster” = “010011001101010101001111010001” has this firing pattern. The string length is equal to the Burn time. For each second of the simulation, a character represents if the thruster would fire or not for that second. For the simulation, the frame rate will be set 30Hz. Which means for each seconds, there are 30 frames run each second. So a single character, will be used to judge if the thruster will fire or not.

The firing pattern for the first generation will be created using a definite “for” loop. The “for” loop will run from 1 to and including burn time. Within the for loop, there will be a random generator that will generate a number either a 0 or a 1 in each iteration which then will be converted in to a character and then added to the string.

The int “angle” is used to generate the angle the thruster will be placed around the rocket.



The angle at which the thruster is placed around the thruster is placed around the rocket will be chosen using a random number generator. The angle generated will always be:

Ѳ = integer angle value.

The range for Ѳ: 0° <= Ѳ <= 259°.

The constructors:

There will be 2 constructors in the Gene class:

Gene(): this constructor is used on the 1st generation. To create a truly a random gene with a random firing pattern as well as a random angle for the thruster.

Gene(string thrusterChild, int angleChild): this constructor will be used on the other generation where a gene is reproduced.

The methods:

getAngle() - will be used to retrieve the angle at which the specific thruster will be placed around the rocket.

getThruster() – will be used to retrieve the binary firing pattern generated for the specific thruster.

Showdetails() – will be used to print the firing pattern and the angle of the thruster on to the console.

Savedetails() – will be used to save details about the gene.

The Gene class will be called by the DNA class to produce a specific number of gene for each species.

**DNA:**

The DNA class will be used to create DNA for the rocket. The DNA will contain different number of Genes (depends on the species of the rocket). The DNA for all specie of rocket will be very similar and will be generated using the same constructer.

The DNA class will have a list of Genes, which is a dynamic data structure to hold the Genes for each structure. The size of the list depends on the rockets type. If the rocket have 4 thrusters, the list will have a size of 4, one for a Gene which corresponds to one thruster.

|  |
| --- |
| Class – DNA |
| “genes” – List<Genes> |
| + DNA(int rocket\_type) |
| +DNA(string[] thruster, int[] angle) |
| +DNA(string[] thruster, int[] angle, int a) |
| +showdetails() |
| +savedetails() |
| +getAngles() : int[] |
| +getThrusters() : string[] |
| +getRocketType() : int |
| +bubblesort(Gene – list) : gene – list |
| +setPatter(String[] p ) |

Each instance of a DNA, which contains varied number of genes. Which will be held in a dynamic list of genes.

The purpose of this class is to manage the genes of the rockets which are essential for the rockets and different for each rocket.

The constructors:

There are 3 constructors in this class:

* DNA (int Rocket Type): this constructor is used to generate DNA for the 1st generation of rockets. This use a for loop to instantiate a number of random genes that is required for the rocket and then will be added to the list.
* DNA (string[] thrusters, int[] angles): This constructor is used when a population is reproduced. When the string array for the firing pattern and a int array for the genes are passed in from the reproduction class. The array will be used to create genes and then will be added to the list of genes
* DNA (string[] thrusters, int[] angles, int a): This constructor is also used when a population is reproduced and has the same function as the constructor I mentioned before. However, it also has a another property. This class is used in reproduction specially where the species mutates into a new species. Where the rocket gains a new thruster. So a new gene is added on top of the reproduced genes, using the random gene constructor.

The methods:

getAngles() - will be used to retrieve the angles at which all the thrusters will be placed around the rocket.

getThrusters() – will be used to retrieve the binary firing pattern generated for all the thrusters of the rocket. The getThruster and the getAngles functions will be used to reproduce the rockets.

Showdetails() – will be used to print the firing patterns and the angles of the thrusters on to the console.

Savedeatils() – will be used to print the firing patterns and the angles of the thrusters on to a text file for each rocket of the population.

getRocketType() – which uses the size of the gene list to calculate how many thrusters it’ll have. The integer returned is the rocket type.

bubblesort(Gene – list) – once DNA objected is instantiated. The genes in the list are sorted in order of the angle size using quick sort in ascending order. The reason for the genes to be sorted is, so that it is easier to reproduce rockets using crossover. Since we need to combine the genes of the rockets that has the angle similar to the other rocket’s gene. I’ll be using bubble sort to sort the list of genes in order because the list are relatively small so with the time complexity of bubble sort being n2, the sorting process would be relatively quick.

This sorting algorithm will sort the genes in the list in ascending order of the angle property of the gene.

Bubble Sort:

The bubble sort algorithm has a time complexity of n^2 due to the nested for loop inside a for loop. Which is ideal for small lists.

Pseudocode:

for i from 1 to size of list

for j from 0 to size of list-1

if ( a[j]>a[j+1] )

swap(a[j], a[j+1])

the first for loop iterates from 1 to the size of the list. Inside this for loop, the nested loop will run from 1 to size of the list – 1. Within this for loop, the if statement compares the value of the angle at the position j to the adjacent gene value and if the j angle value is higher. The 2 genes position in the list will be swapped. After the 1st iteration of the main for loop, the biggest value of the list will be at the last position of the list.

For example:

If the DNA had these 4 genes in the list:

|  |  |  |  |
| --- | --- | --- | --- |
| Gene 1:  Angle : 50°  Firing pattern: “010101” | Gene 2:  Angle : 2°  Firing pattern: “101001” | Gene 3:  Angle : 200°  Firing pattern: “010001” | Gene 4:  Angle : 100°  Firing pattern: “100101” |
| Gene 2:  Angle : 2°  Firing pattern: “101001” | Gene 1:  Angle : 50°  Firing pattern: “010101” | Gene 4:  Angle : 100°  Firing pattern: “100101” | Gene 3:  Angle : 200°  Firing pattern: “010001” |
| Gene 2:  Angle : 2°  Firing pattern: “101001” | Gene 1:  Angle : 50°  Firing pattern: “010101” | Gene 4:  Angle : 100°  Firing pattern: “100101” | Gene 3:  Angle : 200°  Firing pattern: “010001” |
| Gene 2:  Angle : 2°  Firing pattern: “101001” | Gene 1:  Angle : 50°  Firing pattern: “010101” | Gene 4:  Angle : 100°  Firing pattern: “100101” | Gene 3:  Angle : 200°  Firing pattern: “010001” |
| Gene 2:  Angle : 2°  Firing pattern: “101001” | Gene 1:  Angle : 50°  Firing pattern: “010101” | Gene 4:  Angle : 100°  Firing pattern: “100101” | Gene 3:  Angle : 200°  Firing pattern: “010001” |

+ setPattern(String[] p) : will be used to set the firing pattern of the string passed in as the firing pattern of the genes in the DNA.

**Rocket:**

Rocket Design:

Each rocket must be represented accurately on the simulation window. The representation of the rocket must display the critical details of the rocket. Such as the thruster and which angles they are placed around the rocket. They should also be able to display which of the thrusters are firing at a given time and which are idle.

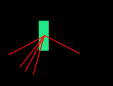
The thrusters will be placed around the main body which is the main rocket which can represent a rocket. The thrusters will also be displayed in different colours at different time of the simulation to portray which thrusters are firing at that moment. The thruster colours will be red if that specific thruster is firing at the frame i.e. the firing pattern is ‘1’ at that frame/time. Else the colour of the thruster will be blue.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 1 |

If the frame count is 62 and the frame rate is 30. We could calculate the time by diving 62 by 30 to get 2.0667. so the 2nd character is chosen to display.

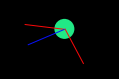
So that thruster would be coloured in red at that time.

Prototype design of rocket 1:

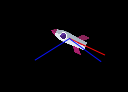


This is a preliminary design of a rocket design. This design portrays the rocket as rectangular green block, and the thrusters around them as thin lines which are either red/ blue depending on the thruster firing pattern at that time. Whilst the thrusters around the relays accurate and informative information about the rockets and how the rockets move due to a specific thruster firing whilst the other is idle, when the rockets are moving, the rocket is moving the rocket will only be pointed towards one direction.

Prototype design of rocket 2:

This design has almost the same properties as the 1st design, except the rocket is represented as circle. This design is an improvement because the circle doesn’t have to be rotated to the direction of motion. However, this does not give the user the impression of a rocket.

Prototype design of rocket 3:

This design will use an image of a rocket and then will also have thrusters around them. The rocket will also be rotated so the rocket is pointed towards the direction of motion to make the simulation more realistic.

|  |
| --- |
| Class – Rocket |
| “ID” – int |
| “dna” – DNA |
| “pos” – Pvector |
| “vel” – Pvector |
| “prevel” – Pvector |
| “hitTarget” – Boolean |
| “dead” – Boolean |
| “fitness” – float |
| “roc” – Pimage |
| “trail” – list<Pvector> |
| Constructors & Methods |
| +Rocket(int ID, int Rocket\_type) |
| +Rocket(DNA dna) |
| + showdetails() |
| +getDetails() : string |
| + getRocketType() : int |
| +getAngles() : int[] |
| +getThrusters() : string[] |
| + setID(int a) |
| + setThrusterPattern(String[] pattern) |
| +display() |
| +move() |
| +checkHIt() |
| +checkDead(ArrayList<Obstacle> obstacles) |
| +calcFitness() |
| +getFitness() |
| +displayFitness() |
| +displayTrail() |
| +displayline() |
| +returnTrail(): List<Vectors> |

ID: the integer id property is assigned to every single rocket in a generation. The id of a rocket is useful to identify a rocket in the screen as well as to check their fitness score and the position of the rocket in the population in terms of the rocket.

DNA – dna is the data that prescribes the movement of the rocket. As mentioned above, the dna contains the genes of the rocket. The dna and the genes within the dna will be used to move the rocket in a specific path.

Pos – the vector pos contains the current position of the rocket.

Vel – the vector vel holds the velocity of the rocket. The velocity is worked out using the genes of the rockets for each frame.

Prevel – this vector holds the velocity of the rocket when the velocity is not equal to zero, this is useful for making the rocket point towards direction of motion.

hitTarget – is a Boolean that says true when the rocket reaches the target.

Dead - is a Boolean that says true when the rocket is killed by collision or going outside the simulation screen.

Fitness – is a float that holds the fitness value of the rocket, that is calculated every frame.

Roc – is the image of the rocket that represents the rockets.

Trail – this list of Pvector holds the position of the rocket after it has been moved, this will be used to draw a trail for the best rocket.

The Constructors:

Rocket(int ID, int Rocket\_type): this constructor is used to produce the 1st generation of rockets. this constructor takes in arguments for the id and the type of rocket. This is then used to create a new random DNA. the id of the rocket is also stored in the property ID.

Rocket(DNA dna): this constructor is used to produce rockets when a generation is reproduced. A reproduced DNA is passed into the constructor and the dna is assigned to the DNA property of the rocket.

Methods:

Showdetails() – will be used to print the firing patterns and the angles of the thrusters on to the console. This is done by calling the showdetails() procedure from the DNA object in the class.

Getdeatils() – will be used to print the firing patterns and the angles of the thrusters on to a text file for each rocket of the population. This is done by calling the savedetails() procedure from the DNA class.

getRocketType() – is a function to return the type of the rocket, which is done by calling the function of the same in the DNA class.

getAngles() - will be used to retrieve the angles at which all the thrusters will be placed around the rocket.

getThrusters() – will be used to retrieve the binary firing pattern generated for all the thrusters of the rocket. The getThruster and the getAngles functions will be used to reproduce the rockets. Both, getThrusters() and getAngles() will operate by calling the functions in the DNA class.

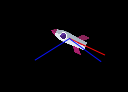
setID(int a) – is a procedure that sets the id value of the rocket, with a value that passed in. this method is used for rockets that are produced after the 1st generation.

setThrusterPattern(String[] pattern) : will call the dna.setPatternFunction(pattern) subroutine. This subroutine will be used in the reproduction class to amend the firing pattern.

Display() – the display procedure is used to display the rocket on the simulation screen. For the simulation I will be representing each rocket in the simulation by the design 3. The main body is an image of a rocket which is loaded when a rocket is instantiated, into the roc image property.

currentTime – is the time it has been since the simulation has begun. Calculated by framecount/frametime. This is done in the draw subroutine of the main page.

At the start the rockets will be pointed upwards, then using the get angles function and get thrusters function are used to create to arrays with the firing pattern and the angles. Then a for loop will be used to display the thrusters around the main body at the right angle, and then determining the right colour for that specific thruster using the current time and the value of the string at the index (current time).

the main image of the rocket is also rotated to the direction of motion, using the vector velocity.

Since this function is called every single frame, the colours of the thrusters will vary between red and blue depending on the time.

Also, if the rocket is dead or reached the target, all the thrusters will be white.

Pseudocode:

Represent rocket using the image, set image of the rocket at the vector pos, then rotate the image of the rocket facing the direct using the vector vel.

Then display the thrusters around the rockets, by retirving the angles they are placed around the target and then rotating the thruster, which will be represented by a thin rectangle. The colour of the thruster is determined by the state of the thruster and the rocket.

Image (roc) at pos.x and pos.y

Roatate ()

For each thruster, draw rectangle at pos.x , pos.y, then rotate at the angle it’s going to be placed: rotate(gene.angle).

If(thruster is firing) fill {red}

Else if (dead || hitTarget): fill {white}

Else: fill{blue}

Move() – this subroutine is used to move the rocket, according to the DNA of the rocket. Similar to the display subroutine, the angle and the thrusters are retrieved form the DNA to make 2 arrays. Velocity vector is set to 0 and the rocket is checked if it has reached the target. Then a for loop is iterated from 0 to the type of the rocket – 1. The angle at that index is used to create a vector and then if the rocket is firing at the time i.e. the character of the firing pattern for that gene is ‘1’ at the index of the current time, the vector will be added to the velocity vector.

When the loop is ended, the velocity is added to the position of the rocket, which moves the rocket to that location. This is only done when the rocket are not dead and not reached the target.

Pseudocode:

Checkhit()

If not dead and not hitTarget

{

Foreach gene:

Vector angle = Convert angle to a normailsed vector,

If the thruster is firing at the current time

{

Vel add velocity \* vector angle

}

Then the vel vector is added to the pos vector.

Fitness is calculated.

}

checkHit() – this subroutine is called inside the move subroutine. This functions check if the rocket has reached the target, by calculating the distance between the rocket and the target. If the rocket did reach the target, the Boolean hitTarget is set true.

checkDead (ArrayList<Obstacle> obstacles) - this subroutine is used to detect any collision between the rocket and the obstacle or if the rocket leaves the simulation window, the rocket is classed as dead. This function is called form the population class by passing the list of obstacles in the simulation as argument and checking if the rockets collided with them.

Pseudocode:

Check if the rocket position is outside of the screen bounds, if it is set dead = true.

Then foreach obstacle

{

If shape =rectangle

If( pos.x >= rectx1 && pos.x <= rectx2 && pos.y >= recty1 && pos.y <= recty2) then dead = true

Else

If(dist between pos and centre <= radius) then dead = true

}

calcFitness() – is the procedure I will be using to calculate the fitness of the rocket. This procedure is called in the move subroutine after the rocket has been moved.

Pseudocode:

Float X = Distance between the rocket and the target is calculated, using Pythagoras: ( (target.x – pos.x)2 + (target.y – pos.y)2)1/2.

Fitness =

if (dead) fitness/2

if (hitTarget) fitness = fitness + 1000/framecount

return fitness.

getFitness() : returns fitness.

displayFitness() : this function is used to display the fitness score of the rocket next to the rocket if the user wishes to see it. This is one of the features that the user can enable in the simulation page or the homepage.

Displayline() – this subroutine is only used by one rocket in each generation. This subroutine draws line from the rockets current position to the target. This only done to the rocket with the best fitness score in the generation. This is done by sorting the population and taking the best rocket and calling this subroutine.

+displayTrail() – this subroutine is only used by one rocket in each generation. This subroutine draws a trail for the best rocket.

+ returnTrail(): List<Vectors> - this function returns the list of pvectors, which will be used to draw the trail.

**Population:**

The population class is responsible for containing all the rockets in a generation. Every generation, a new population is created. The population holds all the rockets in that generation in a dynamic list of rockets.

The population class is responsible for managing the rockets, as well as sorting the rockets in descending order of the fitness score, displaying a generation, moving the rockets according to their DNA, collision detection between the rockets and the obstacles for the rockets in the population.

|  |
| --- |
| Class – Population |
| “rocket” – List<Rocket> |
| Constructors and Methods |
| +Population(int populationSize) |
| +Population() |
| +populationrocket() : List<Rocket> |
| + rocketAt(int i) : Rocket |
| +addRocket(Rocket roc) |
| +showdetails() |
| +getDetails() |
| +displayPopulation() |
| +movePopulation() |
| +displayPopulationFintess() |
| +checkCollPop( list<Obstacle> obstacles) |
| +order() |
| +quickSort(list<Rocket> roc) : list<Rocket> |
| +printPopFitness() |
| + population\_info() : string |
| +getnoOfSuccRockets() : int |
| + getnoOfDeadRockets() : int |

The only property that population class has is the list of rockets. This list will hold all the rockets in the population. There are 2 ways a population can be initialised.

Constructors:

Population( int populationSize) – this constructor used to create an instance of population class for the 1st generation of rockets. the population size is divided by 5. Population of rocket is created with rockets from species 1 through to 5. The number of rockets form each species is 20% of the population size. The rocket that are created using this constructor is truly random. Therefore, the rocket constructor that is called will be Rocket(int ID, int Rocket\_type).

Population() – this constructor is used to create an instance of the population class. This constructor will be used for population that is produced using reproduction or loading from a saved file. This constructor doesn’t do anything other an create an instance of a class.

Methods:

populationrocket() : List<Rocket> - returns a list of rockets in that population. This will be useful for calculating information about a population as well as reproduction.

rocketAt(int i) : Rocket – this function returns the rocket at the index of the int that’s being passed in.

addRocket(Rocket roc) – this procedure is used to add a rocket to the list of rockets in the population. This is primary way a rocket is added into the solution if an instance of a population is created using the second constructor.

Showdetails() – will be used to print the firing patterns and the angles of the thrusters of every rocket in the list on to the console.

getDetails() – will be used to print the firing patterns and the angles of the thrusters on to a text file for each rocket of the population.

displayPopulation() - will be used to display each rocket in the population by calling the subroutine display() for the rockets using a for loop.

movePopulation() - will be used to move each rocket in the population by calling the subroutine move() for the rockets using a for loop.

displayPopulationFintess() – this subroutine will be used to display the fitness score of the rocket on the simulation pages if the user enables the feature. This will be done by calling the displayFitness() subroutine from the rocket class.

checkCollPop( list<Obstacle> obstacles) – this procedure is called everyframe before a rocket is moved or displayed to check if there has been any collision. This will be done by calling the checkDead (ArrayList<Obstacle> obstacles) procedure in the Rocket class for each rocket in the list.

printPopFitness() – prints the fitness score of each rocket with the Id on the console for each rocket in the population.

population\_info() : string – this calculates information about the population such as the best fitness score, what type of rocket has that score, the low fitness score, what type of rocket has that, as well as the number successful rockets reached the target. This function also calculates the mean fitness score of the population by adding all the fitness scored and then dividing by the size of the population i.e. the size of the list of rockets. which is then returned as a string.

getnoOfSuccRockets() : int – returns the number of successful rockets to reach the target.

getnoOfDeadRockets() : int – returns the number of dead rockets in that generation at that moment.

Order() : this procedure is called to sort the rocket in the list in descending order of the fitness score. In this procedure the quickSort(list<Rocket> roc) is called to sort the rockets and then the list of rockets will swapped with the sorted list.

quickSort(list<Rocket> roc) : list<Rocket> - I’ll be using quick sort to sort the rockets based on the fitness socre. I have decided to use quick sort because, the list of rockets is often quiet large, so it would be inefficient in using bubble sort. Since, quick sort has time complexity of N\*log(N), this sorting algorithm will be suitable to sort the list of rockets.

how quick sort works:

* Select the middle number (round up if necessary, so 10 numbers the middle number would be the 6th) and choose this as the first **PIVOT.** Indicate this choice by underlining.
* Re-write the list so that everything smaller than the pivot is written to its left and everything larger or equal to its right. Do not change the order of any of the non-pivot numbers as you do this. In the new list indicate the pivot with a box.
* The pivot will have divided the list into one or two **SUB-LISTS.** For each sub-list, choose the middle number as a new pivot, indicating it by underlining.
* Re-write the list, repeating the procedure in for each sub-list. The new pivots (as well as the previous one) should be indicated with a box.
* Repeat the procedure, choosing a new pivot for each sub-list and then sorting them.
* Continue until all items have become pivots. Then the list is sorted.

For example: sorting the list : 4,3,8,1 in descending order

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 3 | 8 | 1 |
| 8 | 4 | 3 | 1 |
| 8 | 4 | 3 | 1 |
| 8 | 4 | 3 | 1 |

* Represents a pivot point
* represents a point that’s already sorted.

Pseudocode:

function quicksort(list)

less, greater := two empty list

if length(array) > 1

pivot := select middle element of array

for each x in array

if x < pivot then add x to less

else add x to greater

quicksort(less)

quicksort(greater)

list := concatenate(less, pivot, greater)

return list.

 The steps for in-place Quicksort are:

* Pick an element, called a pivot, from the array.
* Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
* Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which are in order, so they never need to be sorted.

This sorted list is then replaced by the rocket list of the class.

Graph:

For the graph to be displayed in the simulation window, the grafica library will have to be imported into the project, and then create an instance of a graph. Theses will be done in the main simulate section of the code. In this section, a plot is created, the dimensions of the graph is set and the x – axis as well as y – axis will be set a label. The x – axis will be the fitness score where the y axis will be the generation. On the single graph, there will 3 plots drawn, for: Best fitness, Low fitness, Mean fitness Will be plotted for each generation.

In order to store the best, low, mean fitness for each generation, there will be 3 Global lists that can be accessed anywhere form the project. The values that are stored in the 3 global lists will be calculated and added in the POP\_info class.

**POP\_info:**

This class is used to calculate information such as best fitness, mean fitness and low fitness for each generation of rockets. the values are the added to the 3 Global lists. Which are then used to plot 3 type of information in a graph.

|  |
| --- |
| Class – POP\_info |
| bfpoints – GpointsArray |
| lfpoints – GpointsArray |
| mfpoints – GpointsArray |
| Constructor and methods |
| POP\_info(Population population) |
| POP\_info() |
| setSize() |
| drawG() |

The 3 GpointsArray are types of arrays used by the library to store co-ordinates used by the graph to plot the points, its similar to a 2d array. This data structure is takes in the x co-ordinate and the y co-ordinate for each input. There are 3 of the GpointsArray to store the best fitness, mean fitness and low fitness for each generation. These are GpointsArray have not yet been instantiated yet. This will be done in the constructor and the setSize() method.

Constructor:

POP\_info(Population population) : this constructor is used in the main section of the code to create an object of the POP\_info class. This constructor will be called once a generation has been ended and just before reproduction takes place. In this constructor, the ended population is passed in to calculate the best, lowest mean fitness of the population. Which is then, added to the Global list of mean, low and best fitness score.

Then the setsize() method is called to instantiate the GpointsArray with the size of the arrays being the length of the Global list.

Then the drawG() subroutine is called to add the values of the in the Global list to the GpointsArrays. Since the GpointsArrays require, 2 inputs, x co-ordinate and the y co-ordinate. So when the list is being added to the GpointsArrays, the x co – ordinate will always be the position of the value in the list that is being added. This is because values are added linearly to the Global list.

POP\_info() : this constructor is used when a generation is being loaded in from a text file, in this constructor there is no arguments passed in. when the saved generation is loaded, the values in the Global list is replaced by the data loaded from the text file. In this constructor, the set size method is called then the drawG() method.

**Obstacles**:

As mentioned in the analysis, the user will have the ability to draw new obstacles into the simulation. This will be done by clicking the mouse in one location, and then dragging it across the screen and releasing the mouse to draw the obstacle.

There will be 2 obstacle types that the user can switch between and draw them on the screen.

One of them is a rectangle shaped obstacle.

The other obstacle shape is a circle.

The user can switch between these 2 type using the toggle button (mentioned above) to draw either of these shapes on to the simulation screen.

When the user selects the obstacle type and then clicks the mouse and drags across the screen and release the mouse. An instance of an obstacle is created using the information about the mouse position when the mouse is pressed and when its released and the shape of the obstacle. This object of the obstacle is then added into a list of obstacles. This list is then used to display the objects on the simulation screen.

Obstacle Class:

|  |
| --- |
| Class – Obstacle |
| X1 – float |
| y1 – float |
| X2 – float |
| y2 – float |
| Radius – float |
| Shape – char |
| Constructor & methods |
| Obstacle (PVector obsStrt, PVector obsEnd, char shape) |
| +dis() |

Properties:

The floats x1, y1, x2, y2 are the positions of the obstacles. For the rectangle it’ll be the x and y co-ordinates of opposite corners. For the circle it’ll be the x and y co – ordinate of the centre as well as the co-ordinate where the mouse was released. Which will be used to calculate the Radius of the circle.

The Radius is the variable where the radius for the circle is calculated using Pythagoras and stored.

The shape is a string which holds if the string is either a rectangle or a circle.

Constructor:

Obstacle (PVector obsStrt, PVector obsEnd, String shape) – this constructor has parameters of 2 vectors and a string. One of the vectors is the position of the mouse when obstacle start and the other for when the obstacle ends. The string is for the type of obstacle.

+dis() – this subroutine is used to display the obstacles onto the simulation screen.

The list of rocket will be in the simulate section of the code and the obstacles will be added to the list there as well.

Selection:

Once one of the generations ending criteria has been met, the population is reproduced. One of the ways reproduction is crossover. In order for crossover to take place, 2 parents must be chosen from the previous population to produce a single child. The selection process I have decided to use is fitness proportionate selection also called roulette selection.

In fitness proportionate selection, the fittest individuals have a greater chance of survival than weaker ones. This replicates nature in that fitter individuals will tend to have a better probability of survival and will go forward to form the mating pool for the next generation. Weaker individuals are not without a chance. In nature such individuals may have genetic coding that may prove useful to future generations.

In my project, the selection process will be called from the reproduction stage before every crossover child is produced.

This will be implemented by:

* Summing the fitness value of all the rockets in the population.
* Then for each rocket, ( the fitness score is then divided by the sum of the fitness value ) and multiplied by 360 and then added to the previous value of the fitness proportionate. Which are then stored in an array.
* Then a random number is generated between 0 and 360. Then, the random number is compared with the values in the array and when a value in the array exceeds the random number. the position of the value in the array is taken and the rocket in that position in the list is taken and returned as one of the parents.

I’ll be creating a class for the fitness proportionate selection.

**Selection**:

|  |
| --- |
| Class – RhouletteSelection |
| totalFitness – float |
| Range – float |
| Pop – population |
| rocketSelection - List<Rocket> |
| Size – float[] |
| Constructors & methods |
| RhouletteSelection(Population population) |
| + calcRange (float fitness) : float |
| +selectParent(float Rndselect) : Rocket |

Total fitness : stores the total fitness value of all rockets.

Range : which is used in calculating the fitness proportionate for each rocket.

Pop – holds the population that’s being passed in. when the roulette selection is instantiated.

rocketSelection – holds the list of rockets in the population.

Size – is an array that holds the fitness proportionate for each rocket.

Constructor:

RhouletteSelection(Population population) – the class is instantiated by passing in the population of the current generation. Using information form the rockets in the population, the fitness proportion is calculated for each rocket and stored in the size array.

Methods:

calcRange (float fitness) : float – calculates the fitness proportionate for each rockets and returns them. This will be used in the constructor.

selectParent(float Rndselect) : Rocket – by passing the random number as the argument a rocket is selected using the fitness proportionate selection as a parent and returned.

Pseudocode:

For every rocket in the previous population

{

If(fitness proportionate>=randomselect)

Position of rocket is the number of iteration passed.

Return rocket at that position.

}

This class will be used in the reproduction class to choose 2 parents for each cross over child.

Reproduction:

The Reproduction class will also be used to select the parents for the next generation and to produce the DNA for the children. The class will use crossover, mutation and elite method to reproduce.

Once a generation has finished, the fitness value of every rocket is calculated and rockets are arranged in order with the rocket with the highest fitness score being first. The reproduction class uses the DNA to create new DNA for rockets using the DNA from the previous generation.

In reproduction, an empty population is created with no rockets in them. Then using the above-mentioned reproduction methods new rockets are produced and then added to the list.

Percentage of rockets born form each type of reproduction in each population:

Elite – 10%

Mutate – 40%

Crossover – 50%.

These 3 types of reproduction methods will produce the rockets needed for the population. Once the rockets are added to the population, the simulation start again with the new population.

|  |
| --- |
| Class – Reproduction |
| eliteChildren - int |
| mutateChildren – int |
| crossChildren – int |
| Children – Population |
| population – Population |
| Constructor & Methods |
| Reproduction(Population population) |
| + elite() |
| +mutate() |
| +crossover() |
| +returnPop : Population |

The integers eliteChildren, mutateChildren, crossoverChildre are the number of rockets that would be produced from each type of reproduction.

The population children is an empty population with no rocket in the list. Where the off spring rockets will be added to.

The population stores the previous population that being passed in.

Constructor:

Reproduction(Population population) : in this constructor the previous population is passed in as the argument. Using the population size determined by the user, the eliteChildren, mutateChildren, crossoverChildre are calculated. In this constructor, mutation rate also will be changed, if adaptive mutation is on and the condition to change adaptive mutation is met.

This will be done by getting the number of successful rockets reaching the target and if the conditions are met the mutation rate will be set to either 0.09 or 0.7.

Once the mutation rate is set the subroutines elite, mutate, and crossover are called, these subroutines will produce offspring's from elite mutate and crossover types of reproduction (as mentioned above). These rockets are added to the population.

One of the feature this project have is to change the burn time of the rocket in the simulation. If the user did change the burn time of the simulation, the length of the firing pattern has to be either increased or decreased depending on the change. So, after reproduction process has been finished, each rocket in the population will be retrieved and the string array of patterns will be retrieved. If the burn time is less than the length of the firing pattern, the firing pattern will be trimmed to the size of the burn time. However, if the burn time is greater than the length of the firing pattern, then more characters of either one or 0 will be added to the firing pattern, so that The length of the firing pattern matches with the burn time. The characters being added is completely random.

once this process has been completed, in the simulate section of the code, the previous population will be overwritten by the reproduced population.

Elite() – this method is called in the constructor to reproduce 10% of the population using elite carry over.

mutate() - this method is called in the constructor to reproduce 40% of the population using mutation. In this method, an instance of mutate is created and then rockets produced using this method is added to the population.

crossover() - this method is called in the constructor to reproduce 50% of the population using cross over. In this method, an instance of cross over is created and then rockets produced using this method is added to the population.

returnPop() : Population – this function returns the new population.

Elite:

10% of the new population will be made up of elite rockets. elite rockets are rocket with the best fitness score from the last generation carried over to the new generation.

This will be done in the reproduction class.

Using a for loop which iterates from 0 to elite children -1. The rockets in the previous population are retrieved and there DNA & Gene is used to make a new instance of a rocket. These new rockets are then added to the new population using “addRocket(rocket R)” method in the population class.

Crossover:

This type of reproduction will allow 2 rockets from the same species to reproduce a child. After the rockets are arranged in order. 50% of the new population will be from crossover children. For each child, 2 parents are chosen using fitness proportionate selection.

The crossover method works by combining the DNA from the parents to produce to produce a DNA for the rocket that’ll often get a higher fitness score.

This method works by creating DNA and gene – which is the firing pattern and the angle placement similar to both parents.

Crossover steps:

The 2 parents are chosen and called into the crossover function. The rocket’s selected as parents might be different species to each other. For example, a rocket with 4 thrusters might be chosen to crossover with a rocket that has 2 thrusters. So when they cross over, the 2 thrusters in the rocket with the 2 thrusters will be crossed over with 2 thrusters from the 4 thruster rockets. the choice of which thrusters are being crossed over will depend on how close the angles are to each other. Once the 2 thrusters crossed over to produce 2 new genes, the remaining 2 thrusters from the 4 thrusters rocket is also retrieved to make an identical 2 genes. With the 4 new genes a DNA is created and in turn a new rocket. So, when 2 rockets crossover the rocket offspring produced will be the species of rocket with the higher number of thrusters from the parents.

When 2 rockets are chosen and their thruster pattern as well as angles are retrieved and the type of the child is determined, the most similar Genes between 2 rocket in relation to the angle of the rocket will be chosen. Then, for the angle for the child gene will be the average of the 2 angles from both parents. As for the firing pattern, a for loop is used to determine each character of the string.

If the characters from both of the parents are the sae at the position, there is a 70% chance that the child rocket firing pattern for that gene at that index will be the same character. A random number generator will be used to calculate a number between 0 and 1. If the number is greater 0.7 the character will be the opposite of the character in that index. However, if the characters at that index don’t match for both parents, the parents fitness values will be compared and the character with the higher fitness score will be stored in to the child rockets firing pattern.

As mentioned above, the rockets with higher number thrusters will also have thruster that have not been still crossed over, those thrusters will automatically be added to the new DNA.

Then using the new thruster firing pattern and the angles a new DNA is created, then using the DNA a new instance of a rocket is created.

This process is iterated using a for loop to generate required number of children’s using cross over reproduction.

The rockets produced will be added to the new population i.e. the list of rockets inside the population.

The crossover process will take place in the Crossover class and then the crossed over list of rocket will be retrieved in the reproduction class and then added to the new population.

**Crossover:**

In this class, the population in which the selection and crossover takes place as well as the number of children’s needed to be produced will be passed in from the reproduction class. These are then used to generate rockets and will be stored into the list of rockets.

|  |
| --- |
| Class – Crossover |
| population – Population |
| Offsprings – list<Rocket> |
| Constructors & Methods |
| Crossover(Population p, int crossChildren) |
| Selection() : list<Rocket> |
| opp(char t) : char |
| Rockets() : list<Rocket> |

Population – this variable will be used to store the population that’s being passed in to the constructor.

Offsprings – Is a list for storing rockets, that are produced using crossover.

Constructor:

In this class there is one constructor:

Crossover(Population p, int crossChildren) - this constructor will take in the old population as well as the number of off-springs need to be produced. In this constructor, a for loop will iterate from 1 to crossChildren, within the for loop, 2 parents will be chosen using the fitness proportionate selection and then the crossover procedure will take place.

+ Selection() : list<Rocket> - this method is called form the constructor to choose parents for the crossover. In this method, a new instance of fitness proportionate selection will be created and then 2 parents will be chosen and the returned in a list to the constructor for them to be cross over.

opp(char t) : char – as mentioned above, when a new firing pattern is created there is a chance that the character in that position will be swapped for the opposite character. This function return returns the opposite value of the character that’s being passed in. for example, returns ‘1’ when ‘0’ is being is passed in as the argument.

Rockets() - list<Rocket> - returns the list of rocket.

Mutation:

As mentioned in the analysis, the best rocket is chosen and then applied random mutation to the rocket to produce new offspring. 40% of the new population will be made up of rockets produced by mutation.

In the reproduction class, the best rocket is chosen. A new instance of the mutation class is created by passing in the parent rocket, and the number of rockets that’s going to be produced (The class for mutation will be designed below).

For each offspring produced, there is a possibility that the new off-spring produced will be able to mutate into a new species with an extra thruster i.e. a new gene is added to the DNA of the offspring. The chances of the child being mutated into another species is 5%. The program will use a random generator to decide if it the child will be born into a new species. This will be done in the mutation class.

Once the rockets have been mutated, the rockets are added to the new population in the reproduction class.

**Mutation:**

In this class, the rocket and the number of children’s needed to be produced will be passed in from the reproduction class. These are then used to generate rockets and will be stored into the list of rockets which will be then added to the new population in the reproduction class.

|  |
| --- |
| Class – Mutate |
| chanceOfhavingNewThruster – float |
| childRockets – list<Rocket> |
| Methods & Constructors |
| Mutate(Rocket rocket, int numberRocket) |
| + mutThruster(String thrusterPattern) – String |
| + mutAngle(int angle) – int |
| + Rockets() - list<Rocket> |

The float chanceOfhavingNewThruster stores the value for being mutated, which is 5%.

The list of rockets “childRockets” will be used to store the mutated rockets into this list.

Constructor:

This class will have one constructor.

Mutate(Rocket rocket, int numberRocket) – this construct will take a rocket and the number of rockets as the arguments. Then the thruster patterns and the angles for the parent rocket will be retrieved. Then a for loop iterates from 1 to “numberRocket” to create a rocket and then added to the list of rockets.

Mutation for each rocket works by creating an empty array of string and int with the size of the rocket type of the parent. Then for each gene in the DNA of the parent rocket a mutation function takes place and then stored into the new arrays.

For each Gene, the thruster firing pattern is mutated and the angle is mutated. Explained below in the methods.

Once a new set of firing patterns and angles have been created, random number generator is used to calculate a number between 0 & 1. If the random number is less then 0.05, the DNA created by the reproduction will also have an extra Random Gene, which means it’ll be mutated into a new species. If the random number is greater then “chanceOfhavingNewThruster”, then the species of the child will be the same as the parent.

Using the DNA, an instance of a rocket is created, which is then added to the list of rocket. This process is iterated using a for loop.

mutThruster(String thrusterPattern) – String : this function is called from the constructor to mutate the firing pattern that’s being passed in as the argument and then return a mutated firing pattern. This will be done by going through each character in the string as well as generating a random number between 0 and 1 for each charcharchter, and if the number is less than the mutation rate set by the user, the character for the child thruster at that position will be the opposite to the character of the parent thruster firing pattern

for example:

mutation rate: 0.5

parent firing pattern:

“10011”

For the 1st character lets say the random number generated was 0.2.

Since 0.2<mutation rate, the child thruster will have ‘0’ at the position.

Child firing pattern:

“0”

Then the for the next character:

“10011”

say the random number generated was 0.3.

Since 0.3<mutation rate, the child thruster will have ‘1’ at the position.

Child firing patter:

“01”

Then the for the next character:

“10011”

say the random number generated was 0.6.

Since 0.3>mutation rate, the child thruster will have ‘0’ at the position.

Child firing patter:

“010”

And so on, this iterates from 0 to parent firing patter length -1. Once the loop has been ended the new mutated patter in returned to the constructor.

mutAngle(int angle) – int : this function is called from the constructor to mutate the angle of the thruster that’s being passed in as the argument and then return a mutated angle. This is done by multiplying 360 by the mutation rate. And then using a random number generator to select number between ‘-‘ve of the calculated value and the posit calculated value and then added to the angle that passed in and then return that angle.

Pseudocode:

Int temp = 360\*parent angle

Float rand = random (- temp, temp)

Int new angle = parent angle + int(rand)

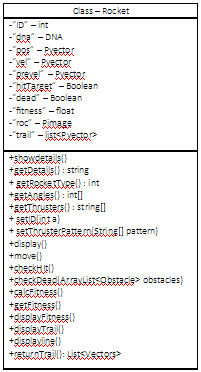
Return new angle

This function uses the mutation rate to calculate a range of how much the new angle vary and a random number generator is used to choose a value that would be added to the angle.

Rockets() - list<Rocket> - returns the list of rocket.

A close up of text on a white background

Description automatically generated

 A close up of a map

Description automatically generated

T - static ArrayList<ArrayList<PVector>>

A screenshot of a cell phone

Description automatically generated

**Technical solution.**

As I mentioned in the design, the code is split into many parts:

“//” the double slashes represent the coments made by me

**Simulate**:

import controlP5.\*; “will be used to import the GUI library “controlP5””

import grafica.\*; ” will be used to import the graphing library “grafica” “

ControlP5 cp5; ” object of the library class.”

GPlot plot;”object of the library class which is used to draw plots.”

PImage targetimage, ro; ” will be used to store the image of the target as well as the image of the main body of the rocket.”

Group g1; ”will be used to group controllers together which can be hidden if the user wishes.”

Button begin, reset, Save, SaveObs, Load, LoadObs; “ buttons to begin and reset the simulation as well save/load and saveobs/loadobs.”

Slider popSize, mutRate, burnTime, velocity; ”sliders for changing the population size, mutation rate, velocity of the rockets as well as burn time.”

Toggle adaptive, fitScore, typeObs, displGraph; “toggle buttons for enabling or disabling adaptive mutation, display fitness score, type of obstacle as well as whether to display graph. For the type of obstacle, the user can draw rectangle obstacle when the toggle is set to true and draw circle if the toggle is set to false.”

Textarea explain; ”text area is used to give brief description of the features available for the user”

boolean boot; ”This Boolean will be used to move to simulation page from the homepage. This will be done by drawing the homepage when the boot Boolean is set to false and will draw the simulation page and the simulation when the boot Boolean is true.”

Population population; ”Is a Population object, which will hold the population for each generation”

Rocket best, low; “are 2 rocket object which will hold the rockets with best and lowest fitness score from the previous generation”

Reproduction reproduce; “an object of the reproduction class which will be instantiated when a generation ending criterion has been met”

POP\_info graph; “an object of the POP\_info class which will be instantiated when a generation ending criterion has been met. This is class will be used to update the graph”

ArrayList<Obstacle> obstacles; “is a list of obstacles which will hold all the obstacle the user has added into the simulation.”

PVector obsStrt; “of is a PVector which will be used to store the position of the position of the mouse when the mouse was pressed. This is used to create a new instance the class then add it to the list”

PVector obsEnd; “is a PVector which will be used to store the position of the position of the mouse when the mouse was released. This is used to create a new instance the class then add it to the list”

String details; “holds the details of the previous population, which will be calculated and stored in this variable when a generation ends”

int tempBurn; “this variable will temporarily hold the burn time when the user changes the slider for the burn time. This is used because one of the ending criteria is calculated using the burn time and if the burn time changes mid-generation, this would change the end criteria and result in an error.”

boolean loadedGen = false; “ this variable determines if the generation is loaded or not.”

void setup() “this subroutine is run once in the simulation, at the beginning of the simulation as well as when the reset button is clicked.”

{

“This procedure sets up all the variables as well as creates the user interface”

fullScreen(1); //this is used to setup the size of the canvas of the screen. this sets the size of the canvas to the size of the screen.

background(0); //sets the background to black

frameRate(30); //frame rate is set to 30Hz.

Global.start = new PVector(width/2, (3\*height)/4); // Assigns the Global variable start, a Pvector. this is the start location for the rocket

targetimage = loadImage("target.png"); //load the image of the target in to the PImage varibale.

Global.target = new PVector(width/2, height/10); // Assigns the Global variable target, a Pvector. this is the target's original location. this is variable will be used to display the target as well for the rockets essential calculation.

ro = loadImage("rocket.png"); //load the image of the rocket in to the PImage variable. which will be used in the rocket class. To represent each rocket using this image.

obstacles = new ArrayList<Obstacle>(); //instantiates a new empty list of obstacles.

details = "NO INFORMATION AVAILABLE”; //set the string regarding previous generation. since on the 1st gen, there is no info for the previous generation.

boot = false; //boot is set to false. since at the start the programe should draw the homepage of the simulation.

Global.t = new ArrayList<ArrayList<PVector>>(); // instantiates an emepty list of list of Pvectors.

cp5 = new ControlP5(this); //instantiates the class need for the GUI.

cp5.setFont(createFont("Calibri", 32));

cp5.setColorActive(0xffff0000);

cp5.setColorForeground(0xffff0000);

cp5.setColorBackground(#001f3f); // sets the fonts as well as the colours for the GUI controllers.

g1 = cp5.addGroup("g1")

.setLabel("PARAMETERS")

.setWidth(200)

.setPosition(10,width/4)

.setBackgroundColor(color(255,80))

.setBackgroundHeight(25)

.activateEvent(true); //create a group by the name g1 and adds it to the GUI. It also sets the poition where its displayed as well as the colour.

begin = cp5.addButton("begin")

.setPosition(width/2-40,height/2 - 100)

.setSize(100,30)

.activateBy(ControlP5.RELEASE)

.setGroup(g1); // creates a new button for the begin function. then adds it to the the group g1. this is done so that the user can hide all the feature if he/she wish.

reset = cp5.addButton("reset")

.setPosition(width/2-40,height/2 -25)

.setSize(100,30)

.activateBy(ControlP5.RELEASE)

.setGroup(g1);// creates a new button for the reset function. then adds it to the the group g1. this is done so that the user can hide all the feature if he/she wish.

Save = cp5.addButton("Save")

.setPosition(width\*3/4,height/10 + 50)

.setSize(100,30)

.activateBy(ControlP5.RELEASE)

.setGroup(g1); // creates a new button for the save function. then adds it to the the group g1. this is done so that the user can hide all the feature if he/she wish.

Load = cp5.addButton("Load")

.setPosition(width\*3/4,height/10 + 150)

.setSize(100,30)

.activateBy(ControlP5.RELEASE)

.setGroup(g1); // creates a new button for the load function. then adds it to the the group g1. this is done so that the user can hide all the feature if he/she wish.

SaveObs = cp5.addButton("SaveObs")

.setPosition(width\*3/4+200,height/10 + 50)

.setSize(120,30)

.activateBy(ControlP5.RELEASE)

.setGroup(g1); // creates a new button for the save obstacle function. then adds it to the the group g1. this is done so that the user can hide all the feature if he/she wish.

LoadObs = cp5.addButton("LoadObs")

.setPosition(width\*3/4+200,height/10 + 150)

.setSize(120,30)

.activateBy(ControlP5.RELEASE)

.setGroup(g1); // creates a new button for the load obstacle function. then adds it to the the group g1. this is done so that the user can hide all the feature if he/she wish.

popSize = cp5.addSlider("popSize")

.setLabel("POPULATION SIZE")

.setColorCaptionLabel(0)

.setPosition(width/15,height/10)

.setSize(450,30)

.setRange(50,500)

.setNumberOfTickMarks(46)

.setSliderMode(Slider.FIX)

.setValue(100)

.setColorLabel(255)

.showTickMarks(true)

.setGroup(g1);// create a slider for changing the population size, set the location, range as well as the staring value and how much they increment by. also added to the group

mutRate = cp5.addSlider("mutRate")

.setLabel("MUTATION RATE")

.setColorCaptionLabel(0)

.setPosition(width/15,height/10+80)

.setSize(300,30)

.setRange(0,1)

.setValue(0.5)

.setColorLabel(255)

.setGroup(g1);// create a slider for changing the mutation size, set the location, range as well as the staring value and how much they increment by. also added to the group

burnTime = cp5.addSlider("burnTime")

.setLabel("BURN TIME")

.setColorCaptionLabel(0)

.setPosition(width/15,height/10+160)

.setSize(300,30)

.setRange(5,100)

.setValue(10)

.setColorLabel(255)

.setGroup(g1);// create a slider for changing the burn time, set the location, range as well as the staring value and how much they increment by. also added to the group

velocity = cp5.addSlider("velocity")

.setLabel("VELOCITY")

.setColorCaptionLabel(0)

.setPosition(width/15,height/10 + 240)

.setSize(300,30)

.setRange(0,10)

.setNumberOfTickMarks(11)

.setSliderMode(Slider.FIX)

.setValue(1)

.setColorLabel(255)

.setGroup(g1);// create a slider for changing the velocity, set the location, range as well as the starting value and how much they increment by. Also added to the group

adaptive = cp5.addToggle("adaptive")

.setLabel("ADAPTIVE MUTATION")

.setColorCaptionLabel(0)

.setPosition(width/2,height/10 + 50)

.setValue(false)

.setColorLabel(255)

.setSize(70,30)

.setGroup(g1);//create a toggle for the adaptive mutation feature. set the initial value to false then added to the group.

fitScore = cp5.addToggle("fitScore")

.setCaptionLabel("FITNESS SCORE")

.setColorCaptionLabel(0)

.setPosition(width/2,height/10 +150)

.setSize(70,30)

.setValue(false)

.setColorLabel(255)

.setGroup(g1); //create a toggle for displaying fitness score.set the initial value to false then added to the group.

typeObs = cp5.addToggle("typeObs")

.setCaptionLabel("OBSTACLE TYPE")

.setColorCaptionLabel(0)

.setPosition(120,width/4 - 100)

.setSize(70,30)

.setValue(true)

.setColorLabel(255); //create a toggle for switching between the different types of obstacles. however this is not added to the group since this should be always visible.

explain = cp5.addTextarea("explain")

.setText("")

.setPosition(width\*3/4 -50,8\*height/10-80)

.setSize(380,270); // create the textarea which will be used to descirbe the features.

grp(); // subroutiine which will be used to create a new graph.

displGraph = cp5.addToggle("displGraph")

.setLabel("GRAPH")

.setColorCaptionLabel(0)

.setPosition(10,width/4 - 100)

.setSize(70,30)

.setValue(false)

.setColorLabel(255); //create a toggle for displaying the obstacle. however this is not added to the group since this should be always visible.

}

void draw() // draw subroutine which will be used to draw the homepage as well as the simulation.

/\*

Called directly after setup(), the draw() function continuously executes the lines of code contained inside its block until the program is stopped.

The number of times draw() executes in each second may be controlled with the frameRate() function.

\*/

{

background(0); //to clear the contents of the window, after each iteraion has been completed.

imageMode(CENTER);

image(targetimage, Global.target.x, Global.target.y, 70, 70); // draws the image of the loaded target at the specified loaction.

for(int i = 0; i< obstacles.size(); i++){obstacles.get(i).dis();} //uses a for loop to iterate from 0 to no.of obstcales -1 to draw the obstacles in the list of obastcles onto the window.

if(!boot) //if boot is flase draw the homepage.

{

fill(255);

textFont(createFont("Bodoni MT Bold", 72));

textAlign(CENTER, CENTER);

text(" GENETIC ROCKETS", width/2, height/12); //draws the title of the simulation

fill(#01FF70);

textFont(createFont("Calibri Bold", 35));

textAlign(CENTER, TOP);

text("Genetic Programming (GP) is a type of Evolutionary Algorithm. Evolutionary Algorithms are used \n"

+"to discover solutions to problems humans do not know how to solve, directly. Free of human preconceptions or biases,\n"

+"the adaptive nature of EAs can generate solutions that are comparable to, and often better than the best human efforts. Genetic\n"

+"rockets provide an elegant example of genetic algorithms. In the beginning the rockets doesn't know how to rech the target,\n"

+"after each generation, the rocket evolve into better genomes which in general gets closer to finding a solution.",width/2,height/6 -20);// a brief description about the program

if(cp5.isMouseOver(adaptive))

{

explain.setText("If active, the mutation rate is raised when all the rockets arent able to find the path. However,if atleast 1/5 of the population reaches the target mutation rate is reduced to less then 0.2.");

}

if(cp5.isMouseOver(fitScore))

{

explain.setText("Display the fitness score of each rocket.");

}

if(cp5.isMouseOver(typeObs))

{

explain.setText("Draw a rectangle obstacle when true and a circle when false");

}

else if(!cp5.isMouseOver())

{

explain.setText("");

}//when the user hovers over the feature buttons and toggles the textarea "explain" will write a small description

}

else//if boot is set to true, draws the simulation page.

{

fill(#03FF69);

textSize(18);

textAlign(TOP, TOP);

text("Generation: "+ Global.generation, 0, 0); // prints the number of the current Generation

text("Current Population Size: "+ population.populationrocket().size() , 0, 22); // prints the size of the current population.

text(details,0,44); //writes details about the previous generation, on the top left of the window.

if(Global.generation > 1 && !loadedGen){best.display();low.display();}

//displays the best and worst rocket from the previous population next to the details about the previous population. but will only dsplay them if the generation is not loaded or the Generation > 1. since its not possible to get the best & worst rocket if the gnerartion is loaded or if its the 1st gen

if(frameCount == (round(frameRate)\*Global.burn\_time) -1 || population.getnoOfDeadRockets() == Global.population\_size || population.getnoOfSuccRockets() == Global.population\_size || population.getnoOfDeadRockets() + population.getnoOfSuccRockets() == Global.population\_size )

/\*

generation ending criteria:

exceeding burn time: so if the framecount = to the frame rate \* burn time

or if all the rockets are dead

or if all the rockets reached the target

or if the all the rockets either died or reached the target

\*/

{// start the generation ending procedure.

Global.burn\_time = tempBurn; // set the tempburn as the Global burntime

population.order();// order the population of rocket interms of their fitness score. biggest to smallest in the list.

population.printPopFitnesss(); //print the fitness score of the population in the console.

details = population.population\_info(); //calculate tthe deatils regarding the generation that ended. best, low, mean fitness scores which species of rocket was the best and the lowest. and then store in to the details

best = population.rocketAt(0); //assigns the best rocket which is in the 1st position of the list in the population to the "best" rocket variable

Global.t.add(new ArrayList<PVector>(best.returnTrail())); // adds the trail of the best rocket to the Global list of list of PVectors.

low = population.rocketAt(population.rocket.size()- 1); //assigns the worst rocket which is in the last position of the list in the population to the "low" rocket variable

best.pos= new PVector(100, 120);

low.pos= new PVector(100, 255);//sets the position of the rocket at the top left corner of the window, so they can be displayed next to the details of the previous generation window

graph = new POP\_info(population);// creates a new instance of the POP\_info which calculates the mean fitness as well get the best and the lowest fitness score. Adds them to the global lists of best,mean and low fitness. Then generates GPointsArray for the plots.

plot.setPointColor(0);

plot.setPoints(graph.bf());

plot.addLayer("low fitness", graph.lf());

plot.getLayer("low fitness").setPoints(graph.lf());

plot.getLayer("low fitness").setLineColor(#F70F0F);

plot.addLayer("mean fitness", graph.mf());

plot.getLayer("mean fitness").setPointColor(#0F1CF7);

plot.getLayer("mean fitness").setLineColor(#0F1CF7);

plot.getLayer("mean fitness").setPoints(graph.mf());// since i ll be plotting 3 diferent points for every x value, i am making 2 more layers for the mean and low fitness, which plots the point so the graph

Global.adaptive = adaptive.getBooleanValue();// gets the value of the adaptive toggle button and assigns the value to the Global adaptive bool which is needed in the next step.

reproduce = new Reproduction(population);//creates a new instance of reprodcuction class by passing the currnet population, to reproduce and build a new population

population = reproduce.returnPop();//get the new population fromm the reproduction class and assing them as the main population.

population.showdetails(); //print the deatils about them to the console.

frameCount = 0; // set the frame count to 0, in order to reset the current time as well reneder generation critira null i.e.exceeding burn time.

loadedGen = false; // set it to false since the population has be reproduced and new generation will start.

Global.generation++; //increments the Generation

redraw(); //call the draw() subroutine again

}

Else //this is drawn every other time unless the generation ending criteria has been met.

{

Global.currentTime = int(frameCount/frameRate); //calculate the current time by dividing the frame count by frame rate

int time = Global.burn\_time - Global.currentTime;

text("TIME REMAINING: " + time, 0, 350); //caluclate and display time remaining before the generation ends.

population.checkCollPop(obstacles); //pass in the obstacle list to check for any rockets in the population that collided with obstacle or flew out of the window and classify those rocket dead

population.movePopulation();// move the rockets in the population according to their DNA.

population.displayPopulation(); //display those rockets in the population.

population.order(); // order the population of rocket interms of their fitness score. biggest to smallest in the list.

population.rocketAt(0).displayline(); //get the best rocket and higlight the rocket by drawing a line from the target to the rocket

population.rocketAt(0).displayTrail(); //get the best rocket and draw a trail fo r the path it has travelled

for(int x = Global.t.size() -1 ; x >= 0; x--)

{

stroke(#FE12FF);

ArrayList<PVector> trail = Global.t.get(x);

for(int i = 0; i < trail.size()-1;i++)

{

line(trail.get(i).x, trail.get(i).y, trail.get(i+1).x, trail.get(i+1).y);

}

}

// use a for loop to draw the trails of the bestrocket from the previous gen. which are stried in the global list of list of rockets

if(fitScore.getBooleanValue()){ population.displayPopulationFitness();}

//get the toggle value of the fitscore toggle if its true display the fitness score of the rocket on the window

if(displGraph.getBooleanValue()){drwgrp();}

//get get the toggle value of the dsiplGraph toggle if its true display the Graph on the window. drawgrp() is a subroutine which is uesd to draw the graph

}

}

}

public void begin()

// this subroutine is called everytime the begin button is pressed, this is used to start the simulation of rockets

{

Global.burn\_time = tempBurn;

// set the tempburn as the Global burntime, which is essential in making DNA and GENE

population = new Population(Global.population\_size); // create a new instnace of the population class. this constructor creates completely random population of rockets. it would be equally made up of rockets from species ranging from 1 to 5.

population.showdetails(); //prints the details of the poulation into the console

Global.generation = 1; //set the Gloabl generation count to 1

boot = true; //set boot = true, which means the draw function will start drawing the simulation of rockets instead of the homepage.

frameCount = 0; //set the frame count to 0, in order to reset the current time.

begin.hide(); //hides the begin button since, after starting the simulation button, there is no point of the begin button.

}

public void reset() // this subroutine is called everytime the reset button is pressed, this is used to return the user to the homepage.

{

Global.best\_Fitness = new FloatList();

Global.low\_Fitness = new FloatList();

Global.mean\_Fitness = new FloatList();//resets the Global list of best, mean and low fitness which is used to draw the graph.

frameCount = 0; //set the frame count to 0, in order to reset the current time.

setup(); // calls the setup() subroutine which is called on the beginning of the simulation. to reset all variables. this includes boot which is set to false so draw will draw the homepage

begin.show(); // show the begin button, which would be hiddden if the user clicked begin before.

}

public void Save() // this subroutine is called when the user clicks the save button.

/\*

this save method saves the:

the users prefernces, features eneabled,

the list of best mean and low fit scores

the obstacles

the rockets information i.e. gene and DNA.

\*/

{

PrintWriter output = createWriter("save.txt"); //create a text file callled "save.txt"

String bfs = "";

String lfs = "";

String mfs = ""; //before the Gloabl lists can be stored they have to be converted into strings.

for(int i = 0; i<Global.best\_Fitness.size();i++)

{

bfs += str(Global.best\_Fitness.get(i)) +",";

lfs += str(Global.low\_Fitness.get(i)) +",";

mfs += str(Global.mean\_Fitness.get(i)) +",";

} //add the values in the list to the string. each entry seperated by a ','

bfs = bfs.substring(0, bfs.length() - 1);

lfs = lfs.substring(0, lfs.length() - 1);

mfs = mfs.substring(0, mfs.length() - 1); //remove the last ',' since when loading the list theyll need to be split. so removing the last ',' is critical.

output.println(Global.mutationRate+","+Global.velocity+","+Global.adaptive+","+displGraph.getBooleanValue());//prints the changes made by the user and the features enabled to the text file

output.println(bfs);//prints the string which holds the best fit scores.

output.println(lfs);//prints the string which holds the lowest fit scores.

output.println(mfs);//prints the string which holds the mean fit scores.

output.println(obstacles.size()); //prints the no.of obstacles in the simulation. this is essintiall to load the obstacles when the user loads the generation. This will be used to make a for loop to add the obstacles

for(int i = 0; i < obstacles.size(); i++)

{

output.println(obstacles.get(i).shape+","+obstacles.get(i).x1+","+obstacles.get(i).y1+","+obstacles.get(i).x2+","+obstacles.get(i).y2);

//for every iteration, a single obstacle is stored on a single line. for each obstacle : its shape, the x,y coordinate when the user started drawing as well as x,y coordinate when the user finished drawing will be stored.

}

output.println(population.getDetails());//population.getDetails() retrieved information about every rocket in the population in the list. Refer to Population Class

output.flush(); // Writes the remaining data to the file

output.close(); // Finishes the file

}

public void SaveObs()// this subroutine is called when the user clicks the saveobs button. this stores a layout of obstacle

{

PrintWriter output = createWriter("saveObs.txt"); //create a text file callled "saveObs.txt"

output.println(obstacles.size()); //prints the no.of obstacles in the simulation. this is essintiall to load the obstacles when the user loads the generation. this will be used to make a for loop to add the obstacles

for(int i = 0; i < obstacles.size(); i++)

{

output.println(obstacles.get(i).shape+","+obstacles.get(i).x1+","+obstacles.get(i).y1+","+obstacles.get(i).x2+","+obstacles.get(i).y2);

//for every iteration, a single obstacle is stored on a single line. for each obstacle : its shape, the x,y coordinate when the user strted drawing as well as x,y coordinate when the user finished drawing will be stored.

}

output.flush(); // Writes the remaining data to the file

output.close(); // Finishes the file

}

public void LoadObs() //this subroutine is called when the loadObs button is clicked. this subroutine will be used to load layout of obstacles saved.

{

obstacles = new ArrayList<Obstacle>(); //create a new empty list of obstacles

String[] lines = loadStrings("saveObs.txt"); //load the save file into a string array, where each entry represent each line

char s;

float x1, y1, x2, y2; //5 variables for properties which descirbe the obstacle.

for(int x = 1; x<lines.length ;x++)

{ //Starting from one the full loop iterates to the link of the length of the array -1. at each iteration :

String[] f = split(lines[x], ","); //each line is split into a string array by a ‘,’.

s = f[0].charAt(0); //The first value in the array is the character wht type it is.

x1 = float(f[1]);

y1 = float(f[2]);

x2 = float(f[3]);

y2 = float(f[4]);//the x,y coordinate when the user strted drawing as well as x,y coordinate when the user finished drawing

obstacles.add(new Obstacle(new PVector(x1,y1),new PVector(x2,y2), s));

//Using this information a new instance of an obstacle class is created. this obstacle is then added to the obstacle list

}

}

public void Load() // this subroutine is used to load an entire generation that being save.

{

grp();// clear the graph by creating a new instance of the graph.

obstacles = new ArrayList<Obstacle>();//create a new empty list of obstacles

Global.best\_Fitness = new FloatList();

Global.low\_Fitness = new FloatList();

Global.mean\_Fitness = new FloatList();//resets the Global list of best, mean and low fitness which is used to draw the graph.

population = new Population();// create an empty population.

String[] thrusters;

int[] angles;// these will be used to store the retrieved Genes of the rockets. Angle and the firirng pattern seperately.

int popSizeof = 0;//holds the size of the pop.

String[] lines = loadStrings("save.txt");//load the save file into a string array

for (int i = 0 ; i < lines.length; i++)

{//in the string file each line represent different vlaues and difeerne variables.

if(i == 0) // the 1st line holds the information about the parameters changed by the user

{

String[] f = split(lines[i], ",");//each line is split into a string array by a ‘,’.

Global.mutationRate = int(f[0]); // the 1st entry is the mutation rate

popSize.setValue(Global.mutationRate);

Global.velocity = int(f[1]);// the next is the velocity.

velocity.setValue(Global.velocity);

Global.adaptive = Boolean.parseBoolean(f[2]); //the next is a boolean value for the adaptive mutation

adaptive.setValue(Global.adaptive);

displGraph.setValue(Boolean.parseBoolean(f[3])); //the final one is a boolean value for displaying the graph

}

else if(i==1) //the next line holds the best fitness list

{

String[] f = split(lines[i], ","); //each line is split into a string array by a ‘,’.

for(int x = 0; x<f.length ;x++)

{

Global.best\_Fitness.append(float(f[x])); //each entry is converted into a float then appended to the Global best fit score list

}

Global.generation = f.length;// the global generation is set as the length of the list. since the no.of entries in the list is directly proportinal to the generation count.

}

else if(i==2) //the next line holds the lowest fitness list

{

String[] f = split(lines[i], ","); //each line is split into a string array by a ‘,’.

for(int x = 0; x<f.length ;x++)

{

Global.low\_Fitness.append(float(f[x])); //each entry is converted into a float then appended to the Global lowest fit score list

}

}

else if(i==3)//the next line holds the mean fitness list

{

String[] f = split(lines[i], ",");//each line is split into a string array by a ‘,’.

for(int x = 0; x<f.length ;x++)

{

Global.mean\_Fitness.append(float(f[x]));//each entry is converted into a float then appended to the Global mean fit score list

}

}

else if(i==4)// this line will hold the no. of obstacles in the simulation

{

int nu = int(lines[i]); //converts to int

char s;

float x1, y1, x2, y2;

for(int x = 0; x<nu ;x++)//a for loop is iterated from 0 to the no. of obstacels saved which is stored in the variable "nu".

{

i++;

String[] f = split(lines[i], ",");//each line is split into a string array by a ‘,’.

// i is incremented and the next line is chosen and split into a string array. since the original line only contained the no. of obstcales,

//where the following lines hold the inforamtion regaridng the obstacles.

s = f[0].charAt(0);//The first value in the array is the character wht type it is.

x1 = float(f[1]);

y1 = float(f[2]);

x2 = float(f[3]);

y2 = float(f[4]);//the x,y coordinate when the user strted drawing as well as x,y coordinate when the user finished drawing

obstacles.add(new Obstacle(new PVector(x1,y1),new PVector(x2,y2), s));

//Using this information a new instance of an obstacle class is created. this obstacle is then added to the obstacle list

}

}

else//all the other lines holds genetic infromation regarding the rocket.

{

popSizeof++;//the pop size increase.

String[] pieces = split(lines[i], ",");//each line is split into a string array by a ‘,’.

int type = int(pieces[0]);//for each rocket, the 1st value is the number of thruster the rockets have, followed by the genes, which are the firing pattern and the angle.

Global.burn\_time = pieces[1].length();

tempBurn = pieces[1].length();//set the burntime since burn time and temp burn time is the same as the length of the firing time.

thrusters = new String[type];

angles = new int[type];//instantiate the arrays with size of the rocket type.

int x= 0 , y = 0;

for(int a = 1; a < pieces.length ; a++)

/\*

after the rocket type, the firing pattern and the angles are placed in alternating order.

so the 1st entry in the array will be stored as the array, where as the 2nd entry as the angle.

so in order to takle this issue, for all the odd value of a, they will by stored in x and for the even values of a will be stored in the angle array.

\*/

{

if(a%2 == 1)

{

thrusters[x] = pieces[a];

x++;

}

else

{

angles[y] = int(pieces[a]);

y++;

}

}

DNA dna= new DNA(thrusters, angles);//once the genes are added, a new DNA is created using the 2 arrays.

population.addRocket(new Rocket(dna));// the new DNA is then used to make a new instance of the rocket. which is added to the population

// this process iterated untill the for loop ends.

}

}

population.showdetails();//prints the details of the population

Global.population\_size = popSizeof;

popSize.setValue(popSizeof);

burnTime.setValue(Global.burn\_time);

Global.population\_size = popSizeof;//updates the population size as well as update the sliders

graph = new POP\_info();//create a new instance of the POP\_info class which will update the GPonitsArray with the new updated lists.

plot.setPointColor(0);

plot.setPoints(graph.bf());

if(Global.generation == 1)

{

plot.addLayer("low fitness", graph.lf());

plot.addLayer("mean fitness", graph.mf());

}

plot.getLayer("low fitness").setPoints(graph.lf());

plot.getLayer("low fitness").setLineColor(#F70F0F);

plot.getLayer("mean fitness").setPointColor(#0F1CF7);

plot.getLayer("mean fitness").setLineColor(#0F1CF7);

plot.getLayer("mean fitness").setPoints(graph.mf());//plotting the 3 different plot in different layers

loadedGen = true;// set true

details = "NO INFORMATIOIN AVAILABLE";//no infor available since this is a loaded gen

boot = true;//if the draw function is till on the homepage, move to the simulation page.

Global.t = new ArrayList<ArrayList<PVector>>();// empty the list of list of PVectors.

frameCount = 0;//set the frame count to 0, in order to reset the current time.

redraw();//call the draw() funtion

}

public void popSize(int size)// this is called when a popSize slider event is triggered i.e. the user has chnaged the slider.

{

Global.population\_size = size;//the argument that was passed in will be set as the Global population size.

}

public void mutRate(float val)// this is called when a mutRate slider event is triggered i.e. the user has chnaged the slider.

{

Global.mutationRate = val;//the argument that was passed in will be set as the Global mutation Rate.

}

public void burnTime(int time)// this is called when a burnTime slider event is triggered i.e. the user has chnaged the slider.

{

tempBurn = time;//the argument that was passed in will be set as the temp burn time and will be set as the global burn time just before the reproduction takes place.

}

public void velocity(float vel)// this is called when a velocity slider event is triggered i.e. the user has chnaged the slider.

{

Global.velocity = vel;//the argument that was passed in will be set as the Global velocity.

}

void mouseClicked()

{

if (mouseButton == RIGHT && !g1.isOpen()) //the user has the ability to change the loaction of the target by right clicking where they want to move it.

// however, it can only work if the group is closed, this reduces any accidental changes the user can make

{

Global.target = new PVector(mouseX, mouseY);//set the target to the mouses location

}

}

void mousePressed()

{

if (!g1.isOpen()) //in order to draw an obstacle into the simulattion the user has to press the mouse and drag it to the required position.

{

obsStrt = new PVector(mouseX, mouseY);// whne the user clicks to draw the obstacle, the obsStrt stores the PVector of the mouse location.

}

}

void mouseReleased()

{

if (!g1.isOpen())

{

obsEnd = new PVector(mouseX, mouseY);// and when the mouse is released, the obsEnd PVector stores the location of the mouse again.

if(typeObs.getBooleanValue())//get the value of the typeObs toggle, if true draw a rect obstacle.

{

obstacles.add(new Obstacle(obsStrt,obsEnd, 'r'));//the x,y coordinate when the user strted drawing as well as x,y coordinate when the user finished drawing as well as the char 'r'

//is used to make new obstacle which is added to the list of obstacle.

}

else//if the value of the typeObs toggle is false draw rectangle obstacle

{

obstacles.add(new Obstacle(obsStrt,obsEnd, 'c'));//the x,y coordinate when the user strted drawing as well as x,y coordinate when the user finished drawing as well as the char 'c'

//is used to make new obstacle which is added to the list of obstacle.

}

}

}

void keyPressed()//if any keyboard is pressed

{

obstacles = new ArrayList<Obstacle>();//delete all the obstacle and create a new empty list.

}

void grp()//this subroutine is used to create a new Graph.

{

plot = new GPlot(this);//create a new instance

plot.setPos(width - 600, 0);

plot.setDim(500, 500);//set dimension as well the eposition

plot.setTitleText("FITNESS");

plot.getXAxis().setAxisLabelText("GENERATION");

plot.getYAxis().setAxisLabelText("FITNESS SCORE");

plot.activatePointLabels();

plot.activateZooming();//set the title as well, x, y axis titles. also, enable zooming

}

void drwgrp()//called in the Draw() funtion to draw the actual graph on to the window.

{

plot.beginDraw();

plot.drawBackground();

plot.drawBox();

plot.drawXAxis();

plot.drawYAxis();

plot.drawTopAxis();

plot.drawRightAxis();

plot.drawTitle();

plot.drawPoints();

plot.drawLines();

plot.drawLabels();

plot.drawGridLines(GPlot.BOTH);

plot.endDraw();

fill(0);

ellipse(width - 550, 20, 20,20);

text(" - Best Fitness score", width - 535, 10);

fill(#F70F0F);

ellipse(width - 550, 575, 20,20);

text(" - Low Fitness score", width - 535, 565);

fill(#0F1CF7);

ellipse(width - 180, 20, 20,20);

text(" - Mean Fitness score", width - 165, 10);//drawing legends for the graph.

}

**Global:**

static class Global

{

static int IDCounter = 0;//id counter which is used to hold the id of the rockets in the population

static PVector start = new PVector();

static PVector target = new PVector();//Pvector of the start and target location

static int burn\_time;// the burn time of the gen

static float velocity = 1;// velocity of the rockets

static int population\_size = 100;//the population size

static float mutationRate = 0.5; //mutation rate

static int generation = 1;//generation count

static boolean adaptive = false;//bool to hold if adapt. mutation is on/ off

static FloatList best\_Fitness = new FloatList();

static FloatList low\_Fitness = new FloatList();

static FloatList mean\_Fitness = new FloatList();//hold the best, low, mean fit scores of the populations

static PImage rocImage;//image of the rocket

static int currentTime;// current time of the generation

static ArrayList<ArrayList<PVector>> t;// list of list of PVectors used to hold the trails of best rockets

}

**Gene:**

class Gene//user defined data type, represents a thruster,

{

String thruster = "";//holds a firing pattern

private int angle;//angle

Gene()// contructor to generate a random Gene i.e. thruster

{

for(int i = 1 ; i <= Global.burn\_time; i++) //a for loop that iterates from 1 to the Global burn time to create a random firing pattern

{

char temp = char(int(random(0,2))+48);//random genereator to choose between o and 1 then convert it ninto ascii by adding 48 then turing it into a character

thruster += temp;//this character is then added to the the string thruster

}

angle = int(random(0,360));// choose a random angle sung a random number generator between o and 360.

}

Gene(String thrusterChild, int angleChild)//constructor where the angle and the firing pattern is passsed during instantiation.

// this is usually done during the reprdoduction/ loading a population processs.

{

thruster = thrusterChild;

angle = angleChild;//set thearguments passed in as the properties.

}

public String getThruster() { return thruster; }

public int getAngle() { return angle; }

public void showdetails()// used to print the genes info into the console.

{

println(getThruster());

println(getAngle());

}

public String savedetails()// used during the saving process.

{

String info = "";

info += getThruster() +",";

info += getAngle()+",";//thruster is added to the string then the angle, which is seperated by the ','.

return info;//which is then returned

}

}

**DNA:**

class DNA

{

private ArrayList<Gene> genes = new ArrayList<Gene>();// a list of Genes, which varies depending on the species of rockets.

DNA(int rocket\_type)//this constructor is used to generate DNA for the 1st generation of rockets

{

for(int i = 0; i < rocket\_type; i++)//a for loop is used to iterate from 0 to the rocket type -1

{

genes.add(new Gene());// a new random gene is instantiated, which is then added to the list of Genes.

}

genes = bubblesort(genes);// the Genes in the list are also sorted in ascedning order in terms of there angles.

}

DNA(String[] thrusters, int[] angles)//This constructor is used when a population is reproduced.

//When the string array for the firing pattern and a int array for the genes are passed in from the reproduction class.

{

for(int i = 0; i < angles.length ; i++)//a for loop is used to iterate from 0 to the size of the array -1

{

genes.add(new Gene(thrusters[i], angles[i]));//create a new Gene by passing a firing pattern as well as the angle. which is then added to the list

}

genes = bubblesort(genes);// the Genes in the list are also sorted in ascending order in terms of there angles.

}

DNA(String[] thrusters, int[] angles, int a)//This constructor is used when a population is reproduced.

//When the string array for the firing pattern and a int array for the genes are passed in from the reproduction class.

//however, this constructor is only used in unique circumstances, where the rocket mutates into a new species rocket.

{

for(int i = 0; i < angles.length ; i++)//a for loop is used to iterate from 0 to the size of the array -1

{

genes.add(new Gene(thrusters[i], angles[i]));//create a new Gene by passing a firing pattern as well as the angle. which is then added to the list

}

genes.add(new Gene());//in addition a new random gene is also created and then added to the list of genes.

genes = bubblesort(genes);// the Genes in the list are also sorted in ascedning order in terms of there angles.

}

public void showdetails()//print the genes information on to the console.

{

for(int i = 0; i < genes.size(); i++)//this is doneby usge a for loop and calling the showdetails() procedure from that class.

{

genes.get(i).showdetails();

}

}

public String savedetails()//returns a string conating all of the genes information.

{

String dets = "";

for(int i = 0; i < genes.size(); i++)

{

dets += genes.get(i).savedetails();

}

dets = dets.substring(0,dets.length() - 1);//removes the last ',' form the string since thats the last thing thats added to the in the savedeatils() function in the gene class.

dets += "\n";//makes a new line for the next rocket

return dets;

}

public int[] getAngles()

{

int[] angles = new int[genes.size()];

for(int i = 0; i < angles.length; i++)

{

angles[i] = genes.get(i).getAngle();

}

return angles;

}

public String[] getThrusters()//returns the thrusters fring patterns in the DNA on an array.

{

String[] thruster = new String[genes.size()];

for(int i = 0; i < thruster.length; i++)

{

thruster[i] = genes.get(i).getThruster();

}

return thruster;

}

public void setPattern(String[] p)//set the firing pattern array thats being passed as the Genes firing pattern for all the genes

{

for(int i = 0; i < p.length; i++)

{

genes.get(i).thruster = p[i];

}

}// this is used during the reprdoduction process.

public int getRocketType()

{ return genes.size();}

public ArrayList<Gene> bubblesort(ArrayList<Gene> vs)

// as mentioned in the design, bubble sort will be used to sort the genes in order to make reproduction more successful

{

for (int x = 0; x < vs.size(); x++)//the main loop is iterated from 0 to the size of the list -1.

{//within this for loop, a nested for loop is present

for (int i = 0; i < vs.size()-1; i++)//which iterates from 0 to the size of the list -2.

{

if (vs.get(i).getAngle() >= vs.get(i+1).getAngle())//if the gene in the 1st position has a greater angle then the gene that adjacent in the list.

{

Gene temp = vs.get(i);

vs.set(i,vs.get(i + 1));

vs.set(i + 1 , temp);// swap the Genes position in the list.

}

} // this loop is iterated for every single element, length -1 no of times.

}

return vs;//returns the sorted list.

}

}

**Rocket:**

class Rocket

{

private int ID;

private DNA dna;

private PVector pos = new PVector(Global.start.x,Global.start.y);

private PVector vel = new PVector();

private PVector prevel = new PVector();

private boolean hitTarget = false;

private boolean dead = false;

private float fitness = 0f;

private PImage roc = ro;// the pimage roc is assigned the image ro which was loaded an image of the rocket in the simulate section

private ArrayList<PVector> trail = new ArrayList<PVector>();

Rocket(int ID, int rocket\_type)//is used to produce the 1st generation of rockets.

//This constructor takes in arguments for the id and the type of rocket. This is then used to create a new random DNA.

{

this.ID = ID;//sets the id equal

dna = new DNA(rocket\_type);//creates a new Random DNA

trail.add(new PVector(pos.x, pos.y));//add the current position of the rocket which is the strt location to the list.

}

Rocket(DNA dna)

{

this.dna = dna;// assign the reprodued dna as the DNA of the rocket

trail.add(new PVector(pos.x, pos.y));//add the current position of the rocket which is the strt location to the list.

}

public void showdetails(){ dna.showdetails();}// prints the detail of the rocket by calling the showdeatils() method of the DNA

public String getDetails(){ return dna.savedetails();}//returns the details of the rocket obe saved

public int getRocketType(){ return dna.getRocketType();}

public String[] thrusters(){return dna.getThrusters();}

public int[] angles(){return dna.getAngles();}

public DNA getDNA(){ return dna;}

public void setID(int a){ID = a;}

public void setThrusterPattern(String[] pattern){dna.setPattern(pattern);}//set the firingg patterns for the genes within the DNA ny the strings thats being passed in

public void display()//this method is used to draw the rocket in the simulation window.

{

// 1st the main body of the rocket will be displayed.

/\*

every time the rocket is dispalyed, the main body of the rocket will be pointed towards the direction of the velocity i.e. the way the rocket is moving.

in order for that to be succesfull the image roc will have to be applied transformation.

\*/

imageMode(CENTER);// dteremines how the paremters are interpreted, the 2nd and 3rd arguments are the position of the centre of the rocket.

pushMatrix();

//Pushes the current transformation matrix onto the matrix stack, pushMatrix() function saves the current coordinate system to the stack. so the transformation can be applied freely without affecting other objects.

translate(pos.x,pos.y);// inorder for the rockets body to be rotated to the direction of motion, the grid has to be moved to the current position of the rocket since,

rotate(atan2(prevel.y, prevel.x)+PI/2);//coordinates are always rotated around their relative position to the origin. refer information regarding prevel at the move(). once the grid has been roated to the required angle using artan(opp/adj).

image(roc, 0, 0, 20, 40);// the image is drawn, the x,y coordinates are 0,0 since the grid has be tranlsated to the current position of the rocket.

popMatrix();//Pops the current transformation matrix off the matrix stack, popMatrix() restores the prior coordinate system.

noStroke();

String[] thrusters = dna.getThrusters();//gets the firing patterns for the thrusters i.e. Gene

int[] angles = dna.getAngles();//gets the angles for the thrusters i.e. Gene

for (int i = 0; i < angles.length; i++)//for every thruster in the rocket

{

String thrusterTemp = thrusters[i];//get thefring pattern for that specfic thruster

if(dead == true || hitTarget == true){fill(#FFFFFF);}// if the thruster is ded or hitTarget colour the thruster white.

else if(thrusterTemp.length() > Global.currentTime && thrusterTemp.charAt(Global.currentTime) == '1'){fill(#FA0808);}//if the thruster is firing i.e. charchetr at index(currenttime) ==1

//colour the thruster red.

else{fill(#0811FA);}// if not firing, colour blue will be the thruster's colour.

//each thruster is represented by a thin rectangle, which is rotated to the appropriate angle according to the Gene.

pushMatrix();

translate(pos.x,pos.y);

rotate(radians(angles[i]+90));// get the angle of the thruster, convert ot raidans, rotate to the appropriate direction

rect( 0, 0, 1, 40);//draw the rectangle

popMatrix();//pop matrix

}

}

public void move()

{

checkHit();// chcke if the rocket reched the target,

if(!dead && !hitTarget)//if the rocket didnt crash or reach the target

{

vel.set(0,0);//set the vel = 0, since the velocity is calculated every farme using the DNA, fring patterns of the rockte and added to the pos.

int[] angles = dna.getAngles();

String[] thrusters = dna.getThrusters();//gets the firing patterns and the angle for the thrusters i.e. Gene

for (int i = 0; i < angles.length; i++)//for every thruster

{

PVector thrusterVector = PVector.fromAngle(radians(angles[i])); // get the angle of the thruster. convert to radians and then form a PVector using the angle.

thrusterVector.normalize();//noramlise so the magnitude is 1.

String thrusterTemp = thrusters[i];// get the firing pattern of the thruster

if(thrusterTemp.length() > Global.currentTime && thrusterTemp.charAt(Global.currentTime) == '1')// if the thruster is firing i.e. charchetr at index(currenttime) == 1

{

vel.x += Global.velocity\*thrusterVector.x;//multiply the x comp of the angle vector by the Global velocity and add it to the velocity x comp

vel.y += Global.velocity\*thrusterVector.y;//multiply the y comp of the angle vector by the Global velocity and add it to the velocity y comp

}

}

pos.add(vel);//then add the velocity to the position of the rocket

if(vel.mag() != 0)

{

prevel.set(vel);//one of the problems with rotating the rockets body to the direction of motion is, when the rockets vel = 0, the rocket dirction is

//returns to the original angle which is horizontal. this is however, unrealistic, since rockets dont just turn back to their previous direction when they stop firing.

// therefore, if the vel = o, the prevel stays unchanged i.e. previous value of the vel. the prevel vector will be used to rotate, so it doent return to the original position.

trail.add(new PVector(pos.x, pos.y));//adds the position of the rocket to the trail list.

}

fitness = calcFitness();//fitness of the rocket is calculated and assigned

}

}

public void checkHit()//check collsion between the rocket and the target

{

float d = dist(pos.x , pos.y , Global.target.x , Global.target.y);//using pythagoras to calculate the distance between the rocket and the target location.

if(d < 35)//if less the 35

{

hitTarget = true ;

}

else

{hitTarget = false;}

}

public void checkDead(ArrayList<Obstacle> obstacles)//check collision between obstacles as well as the screen bounds

{

if (pos.y < 0) //if the rocket travels outside of the screen, they are declared dead.

{

dead = true;

}

else if (pos.y > height)

{

dead = true;

}

else if (pos.x > width)

{

dead = true;

}

else if (pos.x < 0)

{

dead = true;

}

for(int i = 0; i < obstacles.size(); i++)//for every obstacle in the list that was passed in as an argument

{

if( obstacles.get(i).shape == 'r')//obstacle type is a rectangle

{

if(pos.x >= obstacles.get(i).x1 && pos.x <= obstacles.get(i).x2 && pos.y >= obstacles.get(i).y1 && pos.y <= obstacles.get(i).y2)

//if the rockets x pos is greter then the obstacle left corner and less then obstacles roght corner

//and the rockets y po is greater then the top corner and less then the bottom corner

{

dead = true;

}

}

else// if the obstacle is a circle

{

float d = dist(pos.x , pos.y , obstacles.get(i).x1, obstacles.get(i).y1);//using pythagoras to calculate the distance between the rocket and the obstcale centre

if(d <= obstacles.get(i).radius)//it its less then or equal to the radius

{

dead = true ;

}

}

}

}

public float calcFitness() //calculating fitness of the rocket.

{

float distance = dist(pos.x, pos.y, Global.target.x, Global.target.y);//using pythagoras to calculate the distance between the rocket and the target location.

if(hitTarget){distance = 34;}//if the rocket reached the target, set the distance = 1; this is ude to some rocket might get closer to the rocket at later time but

//due to their DNA might get closer to the target. so might get a higher fitness score then the rocket that reached the traget first.

float x = 1000 /distance;

float tempfitness = pow(x,0.8);

if (dead) {tempfitness /= 2;}

if (hitTarget) { tempfitness += 1000/frameCount;}

return tempfitness;// reurn the fitness score

}

public float getFitness()

{

return fitness;

}

public void displayFitness()// displays the fitness score of the rocket next to the rocket on the simulation, if the user enables the feature.

{

textSize(11);

fill(#DB5DFF);

if(width/2 < pos.x)

{

text(ID + ":"+ fitness, pos.x - 20, pos.y);

}

else

{

text(ID + ":"+ fitness, pos.x + 20, pos.y);

}

}

public void displayline()//display a straight line from the rocket to the target. only used by the best rocket. used to highlight the best rocket.

{

stroke(255);

line(pos.x,pos.y,Global.target.x, Global.target.y);

}

public void displayTrail()//displays the trail of the rocket, which it path it followed to reach that position. only used by the best rocket

{

stroke(#2BED13);

for(int i = 0; i < trail.size()-1;i++)

{

line(trail.get(i).x, trail.get(i).y, trail.get(i+1).x, trail.get(i+1).y);

}

}

public ArrayList<PVector> returnTrail()

{

return trail;

}

}

**Population:**

class Population

{

private ArrayList<Rocket> rocket = new ArrayList<Rocket>();//list of rockets, which holds the all the rockets in the population

Population()//constructor will be used for population that is produced using reproduction or loading from a saved file.

{

}

Population(int population)//to create an instance of population class for the 1st generation of rockets.

{

int rocket\_type = 5;// fr the 1st gen, there ll be 20% of rockets from each species ranging 1 to 5

for(int i = 0; i < 5; i++)//iterates 5 time to create 5 diiferent species.

{

//starts with the rockets with 5 thruster and decremnts

for(int a = 0; a < population/5; a++)//for 20% of the population

{

rocket.add(new Rocket(Global.IDCounter, rocket\_type));//create new instance of rockets of that species then add it to the list of rockets

Global.IDCounter++;

}

rocket\_type--;//when one iteration is completed the rocket type decrements, to make 20% of rockets with 4 thrusters.

}

}

public ArrayList<Rocket> populationrocket()

{

return rocket;

}

public Rocket rocketAt(int i)//get the rocket at the index of i

{

return rocket.get(i);

}

public void addRocket(Rocket roc)//adds a rocket to the list of rocket, this is the main way a rocket is added to the population during reproduction as well as loading.

{

roc.setID(Global.IDCounter);//set the ID of the rocket using the global varibale IDcounter

rocket.add(roc);//add the rocket to the list

Global.IDCounter++;//increment the global variable IDcounter

}

public void showdetails()//prints the details of all the rockets in the population.

{

for(int i = 0; i < rocket.size(); i++)

{

println("the rocket has " + rocket.get(i).getRocketType() + " thrusters");

rocket.get(i).showdetails();

println();

}

}

public String getDetails()//returns all the details about all the rockets in the population

{

String details ="";

for(int i = 0; i < rocket.size(); i++)// for each rocket

{

details += rocket.get(i).getRocketType() +",";

details += rocket.get(i).getDetails();//there will be:the rocket type, the DNA i.e. the Genes, printed in the order of the firing pattern then the angle.

}

details.trim();

details = trim(details);//removes blank lines at the end.

return details;

}

public void displayPopulation()// used to display the rockets in the population on the simulation window

{

for(int i = 0; i < rocket.size(); i++)

{

rocket.get(i).display();

}

}

public void movePopulation()// used to move the rockets in the population on the simulation window

{

for(int i = 0; i < rocket.size(); i++)

{

rocket.get(i).move();

}

}

public void displayPopulationFitness()// used to display the fit score of rockets in the population on the simulation window

{

for(int i = 0; i < rocket.size(); i++)

{

rocket.get(i).displayFitness();

}

}

public void checkCollPop(ArrayList<Obstacle> obstacles)// used to check if any rocket collided with the obstacle or went out of bounds in the population on the simulation window

{

for(int i = 0; i < rocket.size(); i++)

{

rocket.get(i).checkDead(obstacles);

}

}

public void order()//will be used to call the sorting algorithem and then assign the sorted list of rocket as the main list.

{

ArrayList<Rocket> sortfitness = quickSort(rocket);

rocket = sortfitness;

}

public ArrayList<Rocket> quickSort(ArrayList<Rocket> roc)//passes in a list of rockets

{

int size = roc.size();

float pivot = 0;

int ind = size/2;

int i;

if (size<=1) //base case when the size of the list is 1 or less, return the list

{

return roc;

}

else

{

ArrayList<Rocket> L = new ArrayList<Rocket>(); //create a new empty list for the left side of the list i.e. before pivot

ArrayList<Rocket> R = new ArrayList<Rocket>(); //create a new empty list for the right side of the list i.e. after pivot

ArrayList<Rocket> sorted = new ArrayList<Rocket>();// create a new empty lift for the sorted list.

pivot = roc.get(ind).fitness;// get the fitness score of the pivot rocket, i.e. the rocket thats in the middle of the list.

for (i=0;i<size;i++) //a for loop that iterates from 0 the length of the list -1 .

{

if (i!=ind)//skips the pivot position

{

if (roc.get(i).fitness>pivot)//if the rocket at i has higher fit score then the pivot rocket add it to the left list

{

L.add(roc.get(i));

}

else// if it has a smaller fitness score add it to the right list

{

R.add(roc.get(i));

}

}

}

L = quickSort(L);

R = quickSort(R);//by recursion call the same function to sort the left and right lists,

//once the lists has been sorted

sorted = L;

sorted.add(roc.get(ind));

sorted.addAll(R);//join the left side of the list then, the pivot then the right side.

return sorted;//return the sorted list.

}

}

public void printPopFitnesss()// used to prin the fit score of rockets in the population on the console window

{

for(int i = 0; i < rocket.size(); i++)

{

println(rocket.get(i).ID + ": " + rocket.get(i).fitness);

}

}

public String population\_info()// this function is called at the end of the generation.

{//this caclulates some essential information regarding the genereation. which are: best fit score, best rocket type, worst fit score, worst rocket type, mean fit score, no of successful rocket

float best\_Fitness = rocket.get(0).getFitness();//fitness score of best rock

int rocketTypeOfBest = rocket.get(0).getRocketType();//rocket type of best rock

float low\_Fitness = rocket.get(rocket.size()-1).getFitness();//fitness score of worst rock

int rocketTypeOfWorst = rocket.get(rocket.size()- 1).getRocketType();//rocket type of worst rock

float temp = 0;

for(int i = 0; i < rocket.size(); i++)

{

temp += rocket.get(i).getFitness();

}

float mean\_Fitness = temp/Global.population\_size;//calculates the mean fit score of pop

int succesRockets = getnoOfSuccRockets();// no of rockets that reached the target

String det= "";

det= "BEST FITNESS SCORE IS: " + best\_Fitness+".\nTHE BEST ROCKET TYPE HAS: " + rocketTypeOfBest+ " THRUSTERS.\n\n\n\nTHE WORST FITNESS SCORE IS: "+low\_Fitness+ ".\nTHE WORST ROCKET TYPE HAS " +rocketTypeOfWorst + " THRUSTERS.\n\n\n\nTHE MEAN FITNESS SCORE IS: "+ mean\_Fitness+".\nNUMBER OF ROCKETS THAT REACHED THE TARGET: " + succesRockets;

return det;

}

public int getnoOfSuccRockets()

{

int count = 0;

for(int i= 0; i < rocket.size(); i++)

{

if(rocket.get(i).hitTarget)

{

count++;

}

}

return count;

}

public int getnoOfDeadRockets()

{

int count = 0;

for(int i= 0; i < rocket.size(); i++)

{

if(rocket.get(i).dead)

{

count++;

}

}

return count;

}

}

**Obstacle:**

class Obstacle

{

private float x1, y1, x2, y2, radius;

private char shape;

Obstacle(PVector obsStrt, PVector obsEnd, char s)//to create obstacles you need 2 vectors as well as a char.

{

shape = s;

x1 = obsStrt.x;

y1 = obsStrt.y;

x2 = obsEnd.x;

y2 = obsEnd.y;

radius = dist(x1, y1, x2, y2);//using pythagoras to calculate the distance between the location where the user click the mouse and where the user released the mouse. only used by the circle

}

public void dis()// used to draw the obstacle on the screen

{

fill(#A09C9D);

stroke(255);//colour of the obstacle and the stoke colour

if(shape == 'r')// if obstacle is a rectangle

{

rectMode(CORNERS);//rectMode(CORNERS) interprets the first two parameters of rect() as the location of one corner, and the third and fourth parameters as the location of the opposite corner.

rect(x1, y1, x2,y2);

}

else

{

ellipseMode(RADIUS); // the first two parameters of ellipse() as the shape's center point, uses the third and fourth parameters to specify half of the shapes's width and height.

ellipse(x1, y1, radius, radius);

}

}

}

**POP\_info:**

class POP\_info//primarily used to update the points in the graph

{

private GPointsArray bfpoints;

private GPointsArray lfpoints;

private GPointsArray mfpoints;

//3 GpointsArray are types of arrays used by the library to store co-ordinates used by the graph to plot the points, its similar to a 2d array. This data structure is takes in the x co-ordinate and the y co-ordinate for each input. There are 3 of the GpointsArray to store the best fitness, mean fitness and low fitness for each generation.

POP\_info(Population population)// this constructor is used to update the GPointArrays, in turn the graph.

{

setSize();// new 3 empty GPointArrays are creted with the size of no. of generation

Global.best\_Fitness.append(population.rocketAt(0).getFitness());

Global.low\_Fitness.append(population.rocketAt(population.populationrocket().size()-1).getFitness());

float temp = 0;

for(int i = 0; i < population.populationrocket().size(); i++)

{

temp += population.populationrocket().get(i).getFitness();

}

Global.mean\_Fitness.append(temp/population.populationrocket().size());

//calculate the best, low, mean fit score and add to the respective Global lists

drawG();//then copy the list into the GPointsArray

}

POP\_info()//this constructor is used during the loading process.

{

setSize();// new 3 empty GPointArrays are creted with the size of no. of generation

drawG();//then copy the list into the GPointsArray

}

public void drawG()

{

for (int i = 0; i < Global.best\_Fitness.size(); i++)

{

bfpoints.add(i+1, Global.best\_Fitness.get(i));

lfpoints.add(i+1, Global.low\_Fitness.get(i));

mfpoints.add(i+1, Global.mean\_Fitness.get(i));

}//uses a for loop to iterate from 0 to no. of entries in the list

//within the loop , 2 values are added to the GPointsArray. one is the pos of the value in the list+1 and the other is the vlaue in the list. since the posiotion in the array can help us give info. on what gen the val was added.

}

public void setSize()

{

int a = Global.generation;

bfpoints = new GPointsArray(a);

lfpoints = new GPointsArray(a);

mfpoints = new GPointsArray(a);

}

public GPointsArray bf()

{

return bfpoints;

}

public GPointsArray lf()

{

return lfpoints;

}

public GPointsArray mf()

{

return mfpoints;

}

}

**Reproduction:**

class Reproduction//this class is reponsible for creating a new popultion usng multiple reprdouction methods

{

private int eliteChildren;

private int mutateChildren;

private int crossChildren;

private Population children = new Population();//creates an empty new population

private Population population;

Reproduction(Population population)//this constructor takes in population that has already ended.

{

Global.IDCounter = 0;// the id counter is reset

this.population = population;//the population that was passed in is assigned to the varibale "population".

int popsize = Global.population\_size;

eliteChildren = int(popsize\*0.1);

mutateChildren = int(popsize\*0.4);

crossChildren = popsize - eliteChildren - mutateChildren;//calculate the the number of rockets produced from each reproduction.

if(Global.adaptive)//if adaptive mutation is on and if the no of rockets reached the target is 0 set mutation rate to 0.7 and if the no of rockets reached the target is > 0.25 population,

//mutation rate is set to 0.09

{

if(population.getnoOfSuccRockets() == 0)

{

Global.mutationRate = 0.7;

}

else if(population.getnoOfSuccRockets() >= Global.population\_size/4)

{

Global.mutationRate = 0.09;

}

}

mutRate.setValue(Global.mutationRate);

elite();

mutate();

crossover();

//calls the elite, crossover and mutate methods

ArrayList<Rocket> childrenRocket = children.populationrocket();// once every rocket has been added to the new population.

//for every rocket, their firirng pattern's length will be same as the last population.

//however, the user could change the burn time, so if the user did change the burn time we have to trim the firing pattern or increase the firng pattern

for(int i = 0; i < childrenRocket.size(); i++)// for every rocket in the population

{

Rocket r = childrenRocket.get(i);

String[] thrust = r.thrusters();//get their firing pattern

for(int x= 0; x < r.getRocketType(); x++)//for every thruster in the rocket

{

//if the firing patten is longer then the burn time, trim the strings.using substrings.

if(thrust[x].length() >= Global.burn\_time)thrust[x] = thrust[x] = thrust[x].substring(thrust[x].length()-Global.burn\_time);

//if the burn time is greater then the length of the firirng pattern. use a random number generator similar to the one that was used by the gene constructor to increase the length of the string

else

{

for(int y = thrust[x].length(); y<Global.burn\_time; y++)

{

char temp = char(int(random(0,2))+48);

thrust[x] += temp;

}

}

}

r.setThrusterPattern(thrust);//the modified string array is then set as the new string for the genes for the rocket.

}

}

public void elite()//this method will be used to create 10% of the population

{

for(int i = 0; i < eliteChildren; i++)// a loop which iterates from 0 to elitechildren - 1.

{

Rocket eliteRocket = population.rocketAt(i);//gets the rocket at position i

String[] thrusters = eliteRocket.thrusters();

int[] angles = eliteRocket.angles();//retrieves thruster patterns and angles.

DNA dnaEliteChild = new DNA(thrusters, angles);// create a new DNA using the firing pattern arrays as well as the angle array.

children.addRocket(new Rocket(dnaEliteChild));//create a new Rocket, which is then added the poulation

}

}

public void mutate()//this method will be used to create 40% of the population

{

Rocket mutating = population.rocketAt(0);//choose the best rocket

Mutate mut = new Mutate(mutating, mutateChildren);//create an instance of the mutate class, which takes the best rocket and the number of rockets needed to produce.

ArrayList<Rocket> Mutated = mut.Rockets();//get the reproduced rockets.

for(int i = 0; i< mutateChildren; i++)

{

children.addRocket(Mutated.get(i));

}//then add the rockets to the population

}

public void crossover()//this method will be used to create 50% of the population

{

Crossover c = new Crossover(population,crossChildren);//create an instance of the mutate class, which takes the population and the number of rockets needed to produce.

ArrayList<Rocket> rs = c.rockets();//get the reproduced rockets.

for(int i = 0; i< rs.size(); i++)

{

children.addRocket(rs.get(i));

}//then add the rockets to the population

}

public Population returnPop(){return children;}//returns the new population

}

**Mutate:**

class Mutate// class which is responsible for mutation

{

private float chanceOfhavingNewThruster = 0.05;// this number is the chance that the new mutated rocket will mutate into a new species or not

private ArrayList<Rocket> childRockets = new ArrayList<Rocket>();

//a list of rockets which will hold the offsprings

Mutate(Rocket rocket, int numberRocket)//the constructor takes in the best rocket and the number of offsprings needed

{

String[] thrusters = rocket.thrusters();

int[] angles = rocket.angles();

//the firing pattern array and the angle array are retieved.

for(int i = 0; i < numberRocket; i++)//for every rocket this loop is iterated

{

String[] newThrusters = new String[thrusters.length];

int[] newAngles = new int[angles.length];

//create a new thruster firing pattern array of the same size as well

//as an angle array of the same size as the parent.

for(int a = 0; a < rocket.getRocketType(); a++)//a for loop which iterates for species of the parent.

{

newThrusters[a] = mutThruster(thrusters[a]);// a muatted version of thruster pattern is assigned to the new array at the same position as the parent.

newAngles[a] = mutAngle(angles[a]);// a muatted version of angle is assigned to the new angle array at the same position as the parent.

}

DNA dnaChild;// an object of DNA

if(random(0,1) <= chanceOfhavingNewThruster){dnaChild = new DNA(newThrusters, newAngles,1);}

//If the random generator produces a number that is less than the chance of having a new thruster a new DNA will be created with an extra gene which means it will be mutated into a new rocket

else{dnaChild = new DNA(newThrusters, newAngles);}//if eles it would cretea new rocket of the same type

childRockets.add(new Rocket(dnaChild)); //this DNA is then used ot create a rocket wg=hich is then added to the list.

}

}

public String mutThruster(String thrusterPattern)

//this function is called from the constructor to mutate the firing pattern that’s being passed in as the argument and then return a mutated firing pattern

{

String newThruster = "";//an empty string

for(int i = 0; i < thrusterPattern.length(); i++)//iterates from 0, length of the thruster - 1.

{

char temp = thrusterPattern.charAt(i);//the char at i

float rand = random(0,1);

if(rand <= Global.mutationRate && temp == '0')

{

newThruster += '1';

}

else if(rand <= Global.mutationRate && temp == '1')

{

newThruster+='0';

}

else

{

newThruster += temp;

}

/\*

if the random no. is less the mutation rate set by the user, the character for the child thruster at that

position will be the opposite to the character

\*/

}

return newThruster;

}

public int mutAngle(int angle)

{//function is called from the constructor to mutate the angle of the thruster

float temp = 360\*Global.mutationRate;//caclulate the range the angle can vary

float rnd = random(-temp,temp);//chosse a random number between neagitve of the number and the positive

return int(angle+rnd);//retunr the number thats athe random number to the angle.

}

public ArrayList<Rocket> Rockets()

{

return childRockets;

}

}

**RhouletteSelection:**

class RhouletteSelection//this class is used in the crossover class. this is usesd to slec tthe parents for crossover.

//therefore, without this class its impossible for the crossover process to begin.

{

private float totalFitness = 0;

private float range = 0;

private Population pop;// population that will be passed in assigned to this property

private ArrayList<Rocket> rocketSelection;// the rockets from the population will be stored here

private float[] size;//an array which will store the fitness proportionate of the population

RhouletteSelection(Population population)//constructor takes in the previous population as an argument.

{

pop = population;

rocketSelection = population.populationrocket();

for(int i = 0 ; i < rocketSelection.size() ; i++)

{

totalFitness += rocketSelection.get(i).getFitness();

}//calculate the total fitness

size = new float[rocketSelection.size()]; //create an empty float array of the size of the population

for(int i = 0 ; i < rocketSelection.size() ; i++)

{

size[i] = calcRange(rocketSelection.get(i).getFitness());

// for every rocket a fitness proportinate range is calculate using the calcRange method, and then stored in the array designated for each rcoket.

}

}

public float calcRange (float fitness)//this method caclulates a fitness proportinate range by passing in the fitnes of the rocket.

{

range += 360\*(fitness/totalFitness); //range is caclulated by adding the previous range with proportiante fitenss of the argument.

return range;

}

public Rocket selectParent(float Rndselect)// this method is used to select a parent for crossover using fitness proportinate selection.

//a random number is passed in as the argument.

{

int position = 0;

for(int i = 0 ; i < rocketSelection.size() ; i++)

{// for every range in the fitness proprotionate array

if(size[i] >= Rndselect)// if the value at position i is greater then the random float

{

position = i;//set position = i

return pop.rocketAt(position);//return the rocket at the psotion

}

}

return pop.rocketAt(position);

}

}

**Crossover:**

class Crossover//this class would be responsible for reproducing offspring using crossover

{

Population population;//assigns the population thats being passedin

ArrayList<Rocket> offsprings = new ArrayList<Rocket>();//holds the rockets thats produced using the crossover methods.

Crossover(Population p, int crossChildren)//the constructor takes in the population from where the parents are chosen and the number of offsprings needed

{

population = p;

for(int i = 0; i < crossChildren; i++)//iterates for the number of offsprings needed.

{

ArrayList<Rocket> Parents = Selecting();//call the selecting method to return 2 parents

String[] thrustersp1 = Parents.get(0).thrusters();

int[] anglesp1 = Parents.get(0).angles();//get the 1st parents angles and the firing patterns

String[] thrustersp2 = Parents.get(1).thrusters();

int[] anglesp2 = Parents.get(1).angles();//get the 2nd parents angles and the firing patterns

int typeOfrocket;

boolean secondRocketisbigger;

boolean sametype;

int prevburn = thrustersp1[0].length();

if(Parents.get(0).getRocketType() < Parents.get(1).getRocketType())

{

typeOfrocket = Parents.get(1).getRocketType();

secondRocketisbigger = true;

sametype =false;

}

else if(Parents.get(0).getRocketType() == Parents.get(1).getRocketType())

{

typeOfrocket = Parents.get(1).getRocketType();

sametype = true;

secondRocketisbigger =false;

}

else

{

typeOfrocket = Parents.get(0).getRocketType();

secondRocketisbigger = false;

sametype = false;

}//the comparison methods determine, which rockets have more number of thrusters or if the rockets are of the same species

//as well as, the rocket type of the offspring is determined

String[] thrustersChild = new String[typeOfrocket];// set it as an empty string

int[] anglesChild = new int[typeOfrocket];//new instances of empty array with the size of rocket type

//since, the offspring will be the same species as the rocket with the most no. of rockets

if(secondRocketisbigger)

{

for(int z = 0; z < Parents.get(0).getRocketType();z++)//for every thruster the smaller rockets has

{

int temp = anglesp1[z];//get the 1st rockets angle for the thruster.

int closestdiffer = 360;

int thrusterPos = 0;

for(int x = 0; x < Parents.get(1).getRocketType(); x++)

{//for very thruster in the 2nd rocket

int diff = abs(temp - anglesp2[x]);//get the difference between that angle and the angle of the thruster of the 1st rocket

if (diff<closestdiffer)//if tht difference is smaller then the previous closest difference

{

closestdiffer = diff;//the diff is set as the closest difference

thrusterPos = x; // set the position of the thruster as the Gene positon which is chosen to cross

}

}// the above algorithem will be used to find the thruster in the second rocket with the similar thruster placement around the rocket.

//this is done to make sure that the genes that combined is similar

int temp2 = anglesp2[thrusterPos];//get the angle at the chosen pos

anglesChild[z] = (temp + temp2)/2;//get the average as the combined angle

String tempS1 = thrustersp1[z];

String tempS2 = thrustersp2[thrusterPos];//thruster pattern for the gene for the 1st and 2nd rocket

float f1 = Parents.get(0).getFitness();

float f2 = Parents.get(1).getFitness();// the fitness of the both rockets

thrustersChild[z] = "";//set the firing pattern empty for the offspring

for(int b = 0; b < prevburn; b++)//for the burntime of the prevoius gen

{

float rnd = random(0,1);//get a random no.

if(tempS1.charAt(b) == tempS2.charAt(b))//if both char are the same at the index

{

if(rnd <= 0.7)//if the rnd no. <= 0.7

{

thrustersChild[z] += tempS1.charAt(b);//add the char to the string

}

else

{

thrustersChild[z] += opp(tempS1.charAt(b));//add the opposit char to the string

}

}

else if(f1 >= f2)

{

thrustersChild[z] += tempS1.charAt(b);//if the char are not the same, add the character, where the rocket of the char has bigger fitness score

}

else

{

thrustersChild[z] += tempS2.charAt(b);

}

}

}

for(int z = Parents.get(0).getRocketType(); z < typeOfrocket;z++)// for the rest of the thrusetrs that havent been produced using crossover

{

anglesChild[z] = anglesp2[z];

thrustersChild[z] = thrustersp2[z];

//get the thruster pattern as well as the angle of the thrusters that are left form the bigger parent and assign them to the offspring's array

}

}

else if(sametype)// if both parents aare of the same type

{

for(int z = 0; z < typeOfrocket;z++)//for every thruster in any of the 2 parent rockets

{

anglesChild[z] = (anglesp1[z] + anglesp2[z])/2;//get the avg.

String tempS1 = thrustersp1[z];

String tempS2 = thrustersp2[z];//thruster pattern for the gene for the 1st and 2nd rocket

float f1 = Parents.get(0).getFitness();

float f2 = Parents.get(1).getFitness();// the fitness of the both rockets

thrustersChild[z] = "";//set the firing pattern empty for the offspring

for(int b = 0; b < prevburn; b++)//for the burntime of the prevoius gen

{

float rnd = random(0,1);

if(tempS1.charAt(b) == tempS2.charAt(b))//if both char are the same at the index

{

if(rnd <= 0.7)//if the rnd no. <= 0.7

{

thrustersChild[z] += tempS1.charAt(b);//add the char to the string

}

else

{

thrustersChild[z] += opp(tempS1.charAt(b));//add the opposit char to the string

}

}

else if(f1 >= f2)

{

thrustersChild[z] += tempS1.charAt(b);//if the char are not the same, add the character, where the rocket of the char has bigger fitness score

}

else

{

thrustersChild[z] += tempS2.charAt(b);

}

}

}

}

else//if the 1st rocket is bigger

{

for(int z = 0; z < Parents.get(1).getRocketType();z++)//for every thruster the smaller rockets has

{

int temp = anglesp2[z];//get the 2nd rockets angle for the thruster.

int closestdiffer = 360;

int thrusterPos = 0;

for(int x = 0; x < Parents.get(0).getRocketType(); x++)

{//for very thruster in the 1st rocket

int diff = abs(temp - anglesp1[x]);//get the difference between that angle and the angle of the thruster of the 2nd rocket

if (diff<closestdiffer)//if tht difference is smaller then the previous closest difference

{

closestdiffer = diff;//the diff is set as the closest difference

thrusterPos = x; // set the position of the thruster as the Gene positon which is chosen to cross

}

}// the above algorithem will be used to find the thruster in the first rocket with the similar thruster placement around the rocket.

//this is done to make sure that the genes that combined is similar

int temp2 = anglesp1[thrusterPos];

anglesChild[z] = (temp + temp2)/2;//get the average as the combined angle

String tempS1 = thrustersp1[thrusterPos];

String tempS2 = thrustersp2[z];//thruster pattern for the gene for the 1st and 2nd rocket

float f1 = Parents.get(0).getFitness();

float f2 = Parents.get(1).getFitness();// the fitness of the both rockets

thrustersChild[z] = "";//set the firing pattern empty for the offspring

for(int b = 0; b < prevburn; b++)//for the burntime of the prevoius gen

{

float rnd = random(0,1);

if(tempS1.charAt(b) == tempS2.charAt(b))//if both char are the same at the index

{

if(rnd <= 0.7)//if the rnd no. <= 0.7

{

thrustersChild[z] += tempS1.charAt(b);//add the char to the string

}

else

{

thrustersChild[z] += opp(tempS1.charAt(b));//add the opposit char to the string

}

}

else if(f1 >= f2)

{

thrustersChild[z] += tempS1.charAt(b);//if the char are not the same, add the character, where the rocket of the char has bigger fitness score

}

else

{

thrustersChild[z] += tempS2.charAt(b);

}

}

}

for(int z = Parents.get(1).getRocketType(); z < typeOfrocket;z++)// for the rest of the thrusetrs that havent been produced using crossover

{ anglesChild[z] = anglesp1[z];

thrustersChild[z] = thrustersp1[z];//get the thruster pattern as well as the angle of the thrusters that are left form the bigger parent and assign them to the offspring's array

}

}

DNA dnaCrossChild = new DNA(thrustersChild, anglesChild);// create a new DNA using the firing pattern arrays as well as the angle array.

offsprings.add(new Rocket(dnaCrossChild));//create a new Rocket, which is then added the list off rcokets

}

}

public ArrayList<Rocket> Selecting()//this method is used, used to select 2 parents from the population for a crossover

{

float selectionNum1 = random(0, 360);

float selectionNum2 = random(0, 360);// 2 random numbers gnerated between o and 360, which will be used to find the parents

ArrayList<Rocket> ParentsForCrossing = new ArrayList<Rocket>();//create an empty list

RhouletteSelection select = new RhouletteSelection(population); //instantiate the rhoulette selection, which would calculate all the the fitness proportionate selection

Rocket Parent1 = select.selectParent(selectionNum1);

Rocket Parent2 = select.selectParent(selectionNum2);// pass the random number generated to the select parent method to choose 2 parenst and then add it to the list.

ParentsForCrossing.add(Parent1);

ParentsForCrossing.add(Parent2);

return ParentsForCrossing;//return the list

}

public char opp(char t)// returns the opposite char of the one that was passed in.

//if 1 is passed in return 0, vice versa

{

if(t == '1'){return '0';}

return'1';

}

public ArrayList<Rocket> rockets(){return offsprings;}

}

**Testing**

**Evaluation:**

|  |  |  |
| --- | --- | --- |
| OBJECTIVES | Has the objectives met or not? | How I have met or not met ? |
| 1. Create a home page for the simulation, which includes:  * The title and a brief description of how the GA works. * The ability for changing the parameters and enable some functionalities of the simulation in the home page. | yes | In order to achieve this objective, I have successfully created a GUI for the user to access all the features available on the simulation. The user can also read the brief description of the GR’s. which gives them a brief idea of how the simulation works. The home page also displays the target image as well as the title of the rocket. When the user clicks begin or load, they are transported to the simulation page. |
| 1. To design the simulation page which displays the:  * Simulation of genetic rockets * Details about the previous generation. Which calculates and displays the as the number of successful rockets that reached the target, mean fit, best, low fitness core and which species of rocket had the highest and the lowest fitness score. * Ability to adjust the parameters of the simulation.  1. To successfully implement features of the simulation (mentioned above).  * To have an active timer that counts down to 0 from the burn time and when the burn time reaches 0. The generation starts again. * A graph to display the trend of how the algorithm has evolved to produce a better generation of rockets. The graph will include: * Best fitness score for each generation * Low fitness score for each generation * Mean fitness score for each generation * As well as a key. | yes | I have also successfully completed this objective.   * The rockets at the start of each generation, they spawn from the start location, and move around the window according to their DNA. where they eventually reach the target. * After each generation is ended, some information about that generation is calculated, displayed on the window. This is visible on the top left side of the scree, where the best rockets fitness, type as well image is displayed, as well as worst rockets fitness, species as well as image is displayed. The mean fitness as well the no. of successful rockets from each population is displayed from the previous population. * Even in the simulation page, the user can change the parameters as well access the features of the simulation. Most changes made by the user take place on the next generation, except for the velocity. The parameters do affect how the simulation behave, for example: increasing population size give more chance of rockets to find a path for the target. * The timer, which is displayed below the previous generation information page. Which counts down from the burn time. * As shown during testing, a graph is displayed on the right side of the window, on the graph the best, mean and the low fitness of the rockets from each generation is displayed, a legend is also displayed to inform the user which line represent which plot. |
| 1. Creating a completely random spawning of rockets:    1. Taking the population size and producing a equal number of rockets from each species.    2. To create different species of rockets with different number of thrusters ranging from 1 to 5. However, there should be no limit to how they evolve to gain more thrusters.    3. To Design a rocket which contain different number of thrusters depending on the species of the rocket, With the movements being completely randomised for the 1st generation.    4. To be able to make the rockets move, according the DNA of the rockets. Each rocket must be represented accurately on the simulation window. The representation of the rocket must display the critical details of the rocket. Such as the thruster and which angles they are placed around the rocket. They should also be able to display which of the thrusters are firing at a given time and which are idle.    5. As well as to make/ draw obstacles as well as to detect any collision between the rockets and the obstacle and then destroys them.    6. Create a target that’s movable.    7. To be able to kill of the rockets when the killing conditions are met and highlight them from the simulation screen. | yes | * In the beginning, when the user clicks begin, a random new generation of rockets is created without the need for any previous genetic material. * By taking the population size, equal number of rockets from each species is produced. * The rockets produced have different number of thruster with thrusters placed around the rocket at different angles. The firing pattern within the rockets will also be different. * Each rocket, whether it be produced by reproduction or a random new generation. They will move around the window according to their DNA. for every thruster in the rocket, if the firing pattern is 1, the rockets will fire in that direction. This makes the movement of the rockets completely dependent on the DNA of the rocket as well as unique. * The user can also draw different types of obstacle which can obstruct the rockets path to the target, this evident on the testing where the user drew different types of obstacles on the screen, where the rockets are deemed dead when the collide them. * The user can also move the targets location, this is done by right clicking the mouse. This feature will allow the user to re-evaluate the fitness of each rocket and find another path to the target. |
| 1. Assessing the fitness of each Rocket in the population.    1. For each rocket, a fitness rocket should be calculated depending on factors including:       1. distance from the target       2. the time taken for the rockets to reach the target       3. if the rocket crashed into an obstacle, these variables should affect the fitness score for each rocket.    2. The fitness function should be calculated at each frame, and the best rocket will also have to be highlighted to the user. The fitness function of each rocket should also be displayed if the user wishes. | yes | * Each rocket has a fitness value which is calculated every frame, their values depend on many factors which are mentioned on the left. * The user can also cho0se to display the rocket fitness next to the rocket on the window. Where they could see if the fit. Score increases or decreases due to its movement. * At each frame, the population sorted using quicksort and the best rocket is highlighted by drawing a line from the rocket to the target. * This shows I have successfully achieved my 4th objectives. |
| 1. For the best rocket to be highlighted and for the reproduction process to begin, the rockets has to be sorted in descending order of the fitness score. To sort the rockets using a sorting algorithm that has the optimum/ minimum time complexity for a large list. The trail for the best rocket from each generation should also be displayed, this would be helpful for the user to see how the best rockets movement differ from other rockets. |  | For the best rocket, in the simulation, which is chosen by sorting the population, a trail is displayed for that rocket. This trail is displayed by a green line. Once a generation is ended, the trail of the best rocket is displayed but displayed in a pink line to represent it’s a trail from a previous generation. This objective is also successfully completed. |
| 1. To be able to determine when one of the criteria for ending a generation is reached and then proceed to the next stage: 2. To reproduce a new generation of rocket using elite carry, mutation and cross over. 3. For elite reproduction: once the population of rockets are sorted in order, 10% of the rockets in the population should be chosen to be carried over into the next generation. 4. For mutation: choosing the best rocket in the population, and then applying random mutations to its DNA, to create a new set of rockets to make up 40% of the new population. The random mutations applied to the parent rocket to create an offspring will be limited by the mutation rate which can be changed by the user. There is also a 2% chance that the rocket might evolve into a new species. 5. For crossover:    * 1. To be able to successfully implement fitness proportionate selection to select the parents for reproducing through crossover.      2. Once 2 parents are chosen to be crossed over, their DNA should be retrieved and combined to make a new child with the combined properties of both the 1st and the 2nd parents. The parents chosen doesn’t necessarily have to be the same species either.      3. To be able to inherit properties from the parent with the highest fitness score if there is an opposing value for the DNA with the parents. | yes | When the rockets satisfy the ending criteria the reproduction process is called.   1. As shown in the testing phase, the rockets in the next generation will be produced from the elite, mutate, crossover reproduction. 2. All the reproduction methods, comply with how a generic GA is supposed to produce off springs. 3. For the elite carry production, the best rockets are chosen and then added to the population. 4. For the mutate reproduction, the best rocket is chosen and mutated number of times to make up 40% of the population. 5. For the crossover reproduction, parents are successfully chosen using fitness proportionate selection and their DNA is successfully combined to make new rockets. as well as inherit more from the rocket with a better fitnesss score. |
| 1. Once the population has been built, the simulation starts again with the new population. | Yes | Using the new population the gen. starts again |
| 1. When the generation has ended, the data from each rocket has to be used to calculate mean fitness, best and low fitness. It is also used in reproduction for crossover, during fitness proportionate selection. | Yes | Using the information calculated a plot of graph is drawn as well info about prev gen is displayed. |
| * To be able to save and load an obstacle layout. | Yes | When the user saves the obstacle layouts, they are saved and can click load obs to load them into the population. |
| * To develop adaptive mutation and give the user the option to use it. To show how the mutation rate would change when a condition of adaptive mutation is met.   + To change the mutation rate to 9% if the number of rockets reached the target is greater then 25% of the population and the mutation rate to be changed to 70% if no rockets reached the target. | Yes | The mutation rate automatically changes when the conditions are met. |
| * To be able to save and load a population of rocket. When they are saved they should be able to save the information about the graphs and about the obstacles (if there were any) and when loading to be able to load the entire population of rockets as well as the graph of the save simulation and the obstacles. | Yes | The user can save and load a population of rocket. When they are saved they save the information about the graphs and about the obstacles (if there were any) and when loading to be able to load the entire population of rockets as well as the graph of the save simulation and the obstacles. I have completed this objective successfully however, I could improve by saving and loading the rockets trails from the simulation |
| * To change the parameters of the simulation using a mouse instead of keyboard as my end-user can stand near to the interactive board and use the touch screen to change the arguments. | yes | As demonstrated on the testing phase, the user can access al the features of the simulation using just the interactive board. |

How I could improve:

One of the problems In the simulation is that, if the Rockets velocity is too high and the obstacles are really small The Rockets just fly past them or through them because the patient is keeping is much greater than the position of the obstacle that the obstacle occupy. this can be solved by using recasting which I would improve the next time I do this project.

another problem I faced was that the elite rockets the are produced doesn't always follow the exact path they followed in the previous generation this is due to a significant reduce the processors performance after certain number generation this might be due to a memory leak in the program.

I could also make my list of trails an array with a certain size. During when the simulation runs there are way too many trails after 100 generations. in order to tackle this problem hi could make the list into an array and remove the first position of the array after the area became full or shift them down so the first version of the array is removed i.e. the 1st trail is remote.

Also during the saving process the trails of the best rockets are not saved so when the population is loaded from a stream file that populations trail is disappeared so in my next item and I will also save and load the trails of the Rockets.

As you know The Rockets have the ability to mutate into a new species after a few generations even though mute the chances of mutating into a new species it's really small there is a chance how about whether I run simulation for more than 3000 generations The Rockets have mutated To species of rockets with more than 400 thrusters. this is however not realistic so in my next change, I should change my fitness scores so do they consider the number of transactions they used to calculate the fitness score as well.

Feedback from the target user:

Pros:

I really like how the user interface is laid out.

I also like how the graph displays the mean the best and the low fitness scores

I also find it informative to see which trusts are firing in which thrusters are not firing and how they affect the motion after rocket.

I also like visualization of the genetic algorithm. As this helps me see how The Rockets evolve to find the target.

I also like adaptive mutation this avoid the rock is reaching a plateau

drawing up obstacles is a good feature to have to show my students how The Rockets will change it's path.

I also like the fact that you can fully interact with the simulation using the touch screen or an interactive board

Cons:

Even though I enjoyed the fitness course being displayed on the screen the number of decimal places of the fitness scores made it difficult to grasp the values.

Listen number of rockets from each population should have the chance to mutate into a new species therefore reduce the chance of me being mutated into a new species.