Department of Mathematics, College of Engineering, Design and Physical Sciences, Brunel University

Mathematics (MMath)

Academic Year 2021 - 2022

Applications of Genetic Algorithms

By David Blair

Supervisor: Dr Lashi Bandara

Contents

1 Abstract

In this paper, we will discuss a number of different applications of genetic algorithms as well as their theoretical and historical background. We have built a package for the programming language R which makes the construction and subsequent parameter optimisation easier and quicker to accomplish. While this package is ideal in terms of its design, it is not without its limitations, most notably its long execution time compared to low-level implementations. With this package, a variety of examples are explored. We have looked at how one may minimise distances between two points in Euclidean and Non-Euclidean space. For the Non-Euclidean case, we have used a matrix function to represent the relative transformation to the space at any point. As well as this, we have provided a way of deriving this matrix from a multivariant function which represents the relative transformation. We then looked at a number of alternative ways to measure distance such as the Manhattan and Minkowski metrics to further test the usability of the package. Lastly, we explored how someone may integrate a physics simulation as an alternative way of calculating the fitness score for each genotype. This is very much a "proof of concept" but does demonstrate our initial intent and provides a starting point for further research into this area.

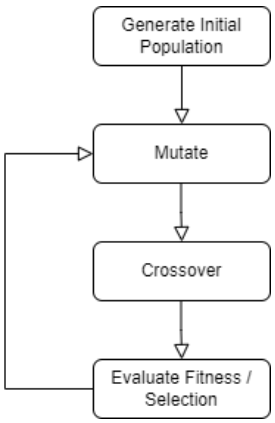

Figure 1: Steps in a Genetic Algorithm

2 Genetic Algorithms: An Overview

2.1 Introduction

A genetic algorithm is a type of evolutionary algorithm intended to mimic the function of evolution towards maximising or minimising a function. The algorithm has wide reaching applications in a variety of different domains. While a basic genetic algorithm has a simple construction (see figure 1), its capabilities can be significantly affected by the individual choices of the algorithms architect. In a lecture given by the late MIT Professor Patrick Winston [18], the importance of these choices is emphasised as he attempts to maximise a function with many local maxima, increasing the sensitivity of the algorithm to these choices. We will discuss the history of genetic algorithms, their various components and some of the tools we've developed to enhance and accelerate their production.

2.2 History

In Alan Turings 1950 paper [17], he describes the idea of a "learning machine" that could mimic the function of evolution towards simulated optimisation at a far greater speed than nature itself. His algorithm includes several familiar features that we find in genetic algorithms today including the need for a numeric representation of our species known as the genotype, mutations in this genotype and a fitness function to evaluate the physical representation known as the phenotype. Throughout the 1950s, further work was done to bring evolutionary algorithms into the modern age. The first use

of a computer to develop these simulations can be attributed to Nils Aall Barricelli in his 1954 paper "Esempi Numerici di processi di evoluzione" [7]. However, genetic algorithms (and more generally evolutionary algorithms) gained significant prominence through John Hollands 1975 book "Adaptation in natural and artificial systems" [9]. It is possible that the first use of crossover within a genetic algorithm could be attributed to John Holland [19] however it is not certain. In any case, his book did contribute greatly to defining the basic framework of a genetic algorithm as well as developing several types of crossover algorithm. He also developed a formalised system for predicting the quality of subsequent generations known as Holland's Schema Theorem. From the late 1970s onward, the growth in processing power [5] meant that commercial applications for genetic algorithms became more viable. From the late 1980s onwards, genetic algorithms began to be used to develop commercial products and in 1989, Axcelis, Inc. released Evolver, the worlds first desktop application designed to perform genetic algorithms and remained the first and only software for this purpose until 1995. Today, many open source and proprietary software packages exist for this purpose such as MATLAB which contains many optimisation algorithms including genetic algorithms.

2.3 Example: Euclidean Distance

We will go through an example to demonstrate how someone might construct a genetic algorithm from scratch. Given two points $A, B \in \mathbb{R}^2$, what is the minimal distance between these points (figure 2)? We will use the case where $A = (10, 5)$ and $B = (40, 20)$. The code for this example can be found on my Github here: euclidean_example_scratch.R

2.4 Problem Suitability

In order for a problem to be suitable for a genetic algorithm, we need to ensure that each of our candidate solutions can be represented as an array of real numbers called the genotype. Interestingly, there are a number of different ways this can be done for our example, all of which have the capability to reach the optimum solution (see figure 2). We may want to use pairs of (x, y) coordinates meaning each of our genotypes will be a 2D array. Below is this genotype representation for the curve in figure 2:

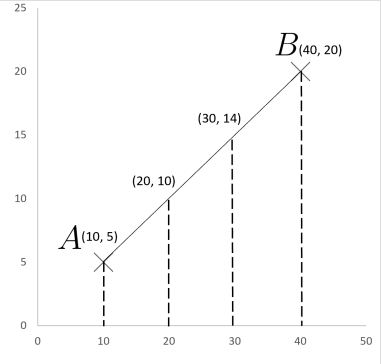

Figure 2: Representation of a Curve

$$\boldsymbol{G} = \begin{pmatrix} 10 & 5 \\ 20 & 10 \\ 30 & 14 \\ 40 & 20 \end{pmatrix} \tag{1}$$

Alternatively, We could also fix the x coordinates at equally spaced intervals between the x coordinate of one point and the x coordinate of the other point, taking only the height of the corresponding curve. Below is this genotype representation for the curve in figure 2:

$$\boldsymbol{G} = \begin{pmatrix} 5 \\ 10 \\ 14 \\ 20 \end{pmatrix} \tag{2}$$

This second option has the added advantage of having fewer numbers to optimise and will most likely converge on an optimum solution faster. In our implementation, we will be using the 2D representation of the curve. This is because its easier to see how you may go from this implementation to the 1D case rather than the other way around.

Secondly, we need to be able to determine how well each of our genotypes performs using a predefined fitness function. For our example, this function is the total length

of the curve represented by the genotype. In other cases however, the presence of confounding factors mean that the derivation of a formula may not be the most suitable fitness calculation method. In these cases, we can use a simulation environment. We will touch more on this later.

2.5 Initial Population

Now that we can represent each of our potential solutions, we can generate an initial population. This population will add a dimension to our genotype array, the length of which will be the total number of genotypes in our population. The size of the population is largely a matter of choice but I've found that using a larger population increases diversity and generally leads to convergence on an optimum solution much quicker.

There is much debate as to which initial population algorithm is more suitable but generally this choice is problem specific. The algorithms usually fall into two categories, random and heuristic. In a 2007 paper [15] they describe a method by which a randomly generated solution and its "anti-solution" are both added to the population. This allows (in many cases) for the algorithm to converge on the optimum solution quicker. Another interesting algorithm developed in 2015 [4] uses k-means clustering to improve diversity in local clusters of a randomly selected initial population. While a heuristic approach is often used to converge on our optimum solution quicker, this also must be balanced with a random approach to maintain diversity and avoid premature convergence on a less than optimum solution.

For our example, we will be constructing an array $\boldsymbol{P}$ of size $(2, 2, 1000)$ such that each $\boldsymbol{P}_{i,j,k} \sim U(\min(\lambda), \max(\lambda))$ where $\lambda = \{A_x, A_y, B_x, B_y\}$. The bounds on this distribution were chosen such that the array $\boldsymbol{P}$ will contain numbers spanning the finite space in which we expect our optimal solution to be. We will discuss in a later section the effects of these initial parameters, specifically in cases where there exists many optimal solutions or local minima.

```
# generate an initial population sampled from a uniform distribution
initial_population_uniform <- function(min, max, genotype_length, number_of_genotypes){
  # Change 'runif' if a different distribution is wanted
  return(
    array(
      runif(
        genotype_length * number_of_genotypes * 2,
        min = min,
        max = max
      ),
      dim = c(genotype_length, 2, number_of_genotypes)
    )
)}
# genotype length is 2 since we will be adding the start and end coordinates
# at a later stage.
current_population <- initial_population_uniform(min = min(c(A, B)), max = max(c(A, B))
                                ,genotype_length = 2, number_of_genotypes = 1000)
```

2.6 Mutation

At this stage, we now have an array which represents our population. if our population can't change over time, it won't be able to converge on anything other than the initial configuration. In nature, many factors cause our DNA to change slightly from generation to generation. In the scientific journal The American Naturalist, geneticist H. J. Muller describes how cosmic rays can actually mutate the DNA of humans [13] and so it is thought that this was a contributor to our own evolution. He later went on to win a Nobel Prize for this work. Coincidentally, cosmic rays are also known to cause a phenomenon known as a bit-flip. This is where radiation causes a change in the bits that form computer memory. Because of this, it's possible (but highly improbable) that the mutations in our genetic algorithm could be induced in a similar way to how our own DNA is mutated. There's even a widely used mutation operator called a string bit mutation which mimics this behaviour.

The genotypes that we will be working with are composed of real numbers. Our mutation operator will add an amount to each of the elements of our array taken from either a Gaussian Distribution or a Cauchy Distribution. For our purposes, we will be using the Cauchy Distribution. Let $\boldsymbol{P}_n$ be the population array in generation n. $\boldsymbol{P}_{n+1} = \boldsymbol{P}_n + \boldsymbol{D}$ where each $\boldsymbol{D}_{i,j,k} \sim \text{Cauchy}(\alpha, \beta)$. α is the location parameter and indicates the location of the mean (in our case this $\alpha = 0$) and β is the measure of spread. The dimensions of $\boldsymbol{D}$ are the same as $\boldsymbol{P}$.

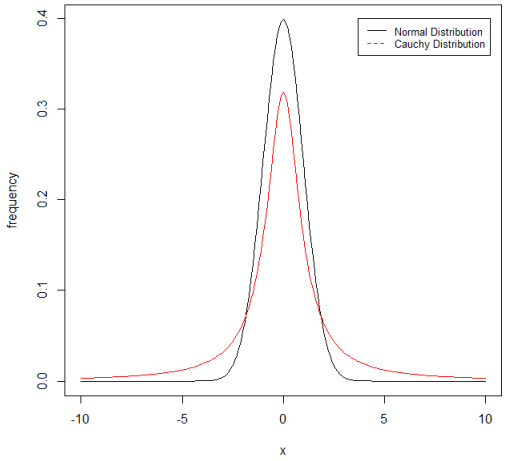

Figure 3: Comparison of Normal and Cauchy Distribution

The shape of these distributions will mean that smaller mutations are more common but there's still a chance for larger mutations. The Cauchy Distribution is thinner in the middle but has much thicker tails meaning the probability for larger mutations is greater (figure 3). This type of mutation operator will hopefully reduce the change of convergence on a local minima or maxima and maintain high population diversity. Below is the code to apply this operator:

```
 # add on a random number to each element from a cauchy distribution
mutation_cauchy <- function(population, location, spread){
  change <- array(
    rcauchy(prod(dim(population)), location = location, scale = spread),
    dim = dim(population)
  );
  return(
    array(as.numeric(population) + as.numeric(change), dim = dim(population))
  )
}

# I've found that a small spread is better for minimisation problems
population <- population %>% mutation_cauchy(location = 0, spread = 0.00000007)
```

2.7 Crossover

In 1909, F.A Janssens was conducting research on the heredity of Drosophila (a genus of fly) [10] and described a phenomena by which genetic material is exchanged between two members of a species during sexual reproduction. Later, building on this work, T.H Morgan constructed the theoretical basis for this process after which a

physical basis was devised by Harriet B. Creighton and Barbara McClintock in 1931 [3].

Our genetic algorithm will perform a similar type of process to that found in genetics. We will split parts of our genotypes to then recombine them, forming new members of our population. This will have a significant effect on the population diversity and it has been shown to greatly improve the convergence speed [12].

Of the different operators that feature in a genetic algorithm, crossover most likely has the most variations. The variations that we will be using are called k-point crossovers (figure 4).

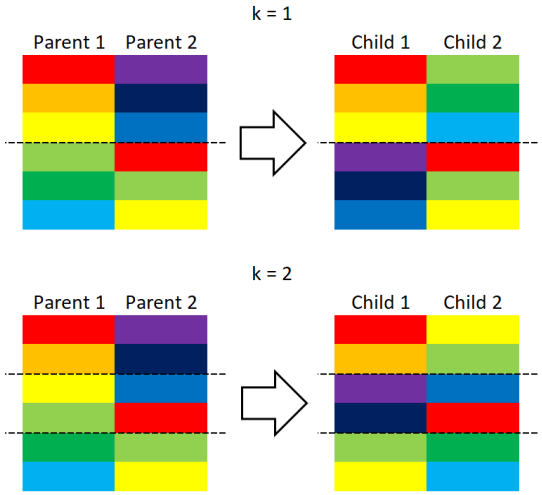

Figure 4: K-Point Crossover

These involve splitting two genotypes along a specific dimension such that we have $k+1$ segments, not necessarily of the same size. These segments are then combined in such a way that two new genotypes are formed. This will greatly improve population diversity and increase the population size. This is important as we will be removing members of the population at a later stage.

The code for this operator uses $k = 1$. We first define a function which can take two parent genotypes and then returns two children. It begins by choosing a location to slice each genotype. Then, it recombines these components before returning them to the user.

```
# breed two genotypes
crossover_breed <- function(gen_1, gen_2){
  pivot_point <- sample(2:(dim(gen_1)[1] - 1), size = 1);
  return(rbind(gen_1[1:pivot_point, ], gen_2[(pivot_point + 1):dim(gen_2)[1], ]))
}
```

Now that we can breed two genotypes, we can apply this process to the population. Firstly, we select a sample of genotypes from our population. Secondly, for each child genotype that we need, we select the parents and breed them using the previous function. We then bind all of these genotypes together with the population and return it to the algorithm.

```
# apply the crossover algorithm
crossover_basic <- function(population, number_of_offspring){
  # Get the people chosen for crossover
  chosen_people <- population[ , , sample(1:dim(population)[3],
                                    size = number_of_offspring * 2, replace = F)];

  offspring <- array(dim = c(dim(population)[1], 2, 0));
  # for each child needed:
  for (genotype in 1:(dim(chosen_people)[3] / 2)){
    # figure out who their parents will be (already randomly sorted)
    gen1 <- chosen_people[, , genotype];
    gen2 <- chosen_people[, , dim(chosen_people)[3] + 1 - genotype];

    # create the child
    current_offspring <- crossover_breed(gen1, gen2);

    offspring <- abind(offspring, current_offspring, along = 3);
  }
  return(abind(population, offspring, along = 3));
}

# apply the crossover operator on the population
population <- population
    %>% crossover_basic(number_of_offspring = as.integer(0.5 * 1000))
```

2.8 Selection

When learning about evolution in school, the presence of certain traits in a species are often characterised in terms of the advantage they have towards their survival, subsequently justifying the longevity of the species. While this is true, its not the entire story. What has allowed this species to evolve is that members of the species who don't have advantageous adaptations are more likely to be killed by predators or their environment.

This brings us on nicely to the selection operator in a genetic algorithm. We evaluate every member of our population using a predefined fitness function, producing an array of "fitness scores" for each member. For our example, this fitness function would be the length of the candidate curve. It's also important to note that since this example is a

minimisation problem, we would take the negative of the fitness scores and attempt to maximise this value. The highest scoring / worst performing members will be removed and the highest scoring members will be allowed to move on to the next generation. The percentage of members that will be removed is largely up to the algorithm designer but we will be setting this percentage such that it is equal to the number of members that we created during the crossover stage. This will ensure the population size stays consistent over the course of the simulation.

Our code for the selection operator involves four functions. The first function will add the start and end coordinates onto a genotype and return it to the user.

```
# add the start and end coordinates
add_start_end <- function(genotype){
  return(
    rbind(
      array(A, dim = c(1, 2)), genotype,
      array(B, dim = c(1, 2)))
    )
}
```

The second function will produce a fitness score for a single genotype which in this example is the euclidean length of the curve. We begin by adding the start ($A = (10, 5)$) and end ($B = (40, 20)$) coordinates. Then, we calculate the length of each segment of the curve, the total of which is returned to the user as the complete length of the curve.

```
# function to calculate euclidean distance
fitness_function_euclidean <- function(genotype){
    genotype_complete <- add_start_end(genotype);
    distance <- 0;
    for (segment_index in 1:(dim(genotype_complete)[1] - 1)){
        diff <- abs(genotype_complete[segment_index, ]
                - genotype_complete[segment_index + 1, ])
        distance <- distance + sqrt(sum(diff^2));
    }
    return(distance)
}
```

The third function will take a population array and run `fitness_function_euclidean` on each genotype. The fitness scores (the length of each candidate curve) will then be returned back to the user.

```
# calculate fitness euclidean
fitness_function <- function(population){
  fitnesses <- apply(population, MARGIN = 3, FUN = function(genotype){
    return(fitness_function_euclidean(genotype))
  })
  return(fitnesses)
}
```

The last function will perform the selection. Firstly, it calculates the fitness of each genotype. we then use the built-in function `order()` to convert this vector into ranks. These ranks can then be used to sort the population and remove the worst performing genotypes. The number of genotypes that we remove is equal to the amount added during the crossover phase.

```
# apply the selection operator
selection <- function(population, number_to_remove){
  fitnesses <- fitness_function(population);
  fitness_order <- order(fitnesses, decreasing = descending);
  new_population <- population[, ,fitness_order];
  new_population <- new_population[ , ,
        (number_to_remove + 1):(dim(new_population)[3])];
  return(new_population)
}

# perform selection
population <- population %>%
    selection(population, number_to_remove = as.integer(0.5 * 1000));
```

2.9 Results

The process that we have described can be repeated for a large number of generations and over time, the average fitness will decrease as intended. You can see the results of this in figure 6 and how the average fitness changed over time in figure 5. The code to execute this process can be found below although I would recommend taking the code from the Github page if you'd like to run this example. It will also print some information about the algorithm while it is executing along with the execution time.

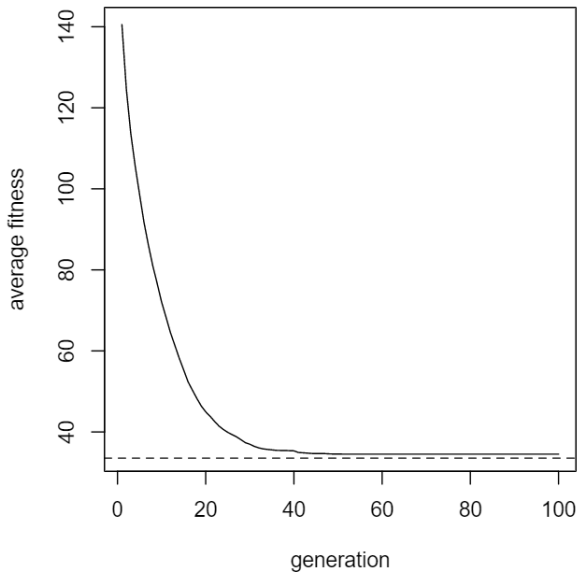

Figure 5: Euclidean Distance: Average Fitness Over Time

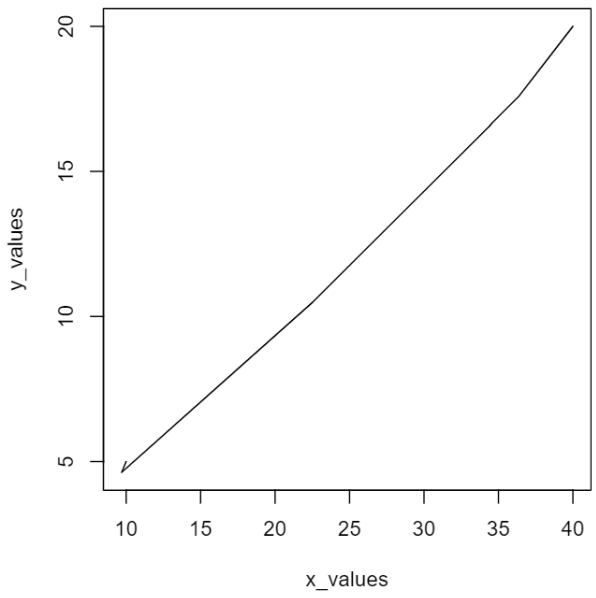

Figure 6: Euclidean Distance: Best Curve From The Final Generation

```
# Some global parameters for the problem
A <- c(10, 5);
B <- c(40, 20);

initial_min <- 0;
initial_max <- 40;

R                     <- 5;
population_size       <- 1000;
remove_add_percentage <- 0.5;
mutation_size         <- 0.00000007;

generations <- 100;
descending <- TRUE;

# create the initial population
initial_population <- initial_population_euclidean(
    min = initial_min,
    max = initial_max,
    genotype_length = R,
    number_of_genotypes = population_size
);
current_population <- initial_population;

fitnesses <- c();

start.time <- Sys.time();

# run the simulation 100 times
for (i in 1:generations){
  print(paste0("Current Generation: ", i, "/", generations))
  # apply each of the operators
  current_population <- current_population %>%
    mutation_cauchy(location = 0, spread = mutation_size) %>%
    crossover_basic(
    number_of_offspring = as.integer(remove_add_percentage * population_size)) %>%
    selection(number_to_remove = as.integer(remove_add_percentage * population_size));

  # get the best perfoming genotype and its corresponding fitness
  fitest <- current_population[, ,1];
  fitest_fitness <- fitness_function_euclidean(fitest);
  fitnesses <-  append(fitnesses, fitest_fitness);

  # if this is the first generation, save the genotype and its fitness
  if (i == 1){
    initial_fitness <- as.numeric(fitest_fitness);
    initial_member <- add_start_end(fitest);
  }

  # print the fitness of each generation
  print(paste0("Current Fitness: ", fitest_fitness));
}

fitest <- add_start_end(fitest)

# plot the initial best member, the fitnesses and the best member of the
# last generation
```

```
plot(initial_member[,1], initial_member[,2], type = "l")
plot(fitest[,1], fitest[,2], type = "l", xlab = "x_values", ylab = "y_values")
plot(1:length(fitnesses), fitnesses, type = "l", xlab = "generation",
ylab = "average␣fitness")
abline(a = 33.54, b = 0, lty = 2)
print("====================================")
print(paste0("Elapsed␣Time␣(seconds):␣", Sys.time() - start.time))
print("====================================")
print(paste0("Initial␣Best␣Fitness:␣", initial_fitness))
print(paste0("Final␣Best␣Fitness␣:␣", fitest_fitness))
print("====================================")
```

The theoretical optimal length for this example is $\sqrt{(40-10)^2+(20-5)^2} = 15\sqrt{(5)} \approx 33.54$, shown by the dashed horizontal line in figure 5. The genetic algorithm has been able to converge on the optimum solution.

3 GA-Package

During this project, we have developed an R package which allows the programmer to quickly implement a genetic algorithm towards solving a variety of problems. It was designed to make it easier to construct, modify and tune a genetic algorithm with minimal amendments to the code. We will use this package to solve a very famous problem in mathematics, the brachistochrone problem. This will help us to demonstrate the capabilities of the package. You can find the package at the link that follows but I would wait until we get to the installation section before trying to do anything with this code. https://github.com/DavidBlairs/gapackage_submit.

3.1 Example: Brachistochrone Problem

3.1.1 Background

The brachistochrone problem is a widely known and very old problem in mathematics. Let A and B be two point in R^2 such that $B_x > A_y$ and $A_x, B_y = 0$ (B and A exist on the x and y axis respectively). If we were to drop a particle P at the point A what is the optimum curve from point A to point B such that the time taken for the particle P to travel between these points is minimised given that the only force acting on the particle (other than reactive forces from the curve) is gravity?

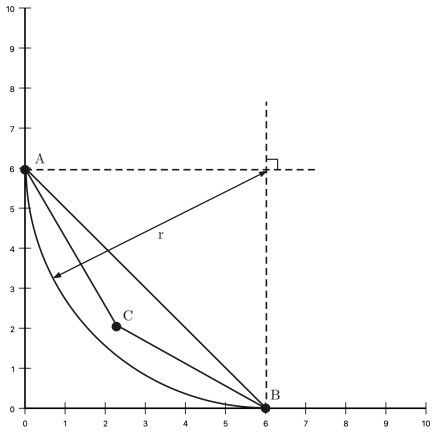

Figure 7: Galileo's Investigation

This problem can be traced as far back as Galileo's 1638 publication "Dialogues Concerning Two New Sciences" [8]. However his version of the problem considered the case when $A_y = B_x$ (see figure 7). Galileo considered the line $\vec{AB}$ (the hypotenuse of a right angled triangle) and noted that if the line were split into two line segments of equal length, $\vec{AC}$ and $\vec{CB}$ such that the point C was on the arc of a circle centred at (B_x, A_y) and passing through both A and B then $T(\vec{AC}) + T(\vec{CB}) < T(\vec{AB})$ where $T(Z)$ is the time taken to traverse along a specific curve Z. This highlights the importance of the gravitational effect on the particle. In 1696, Johann Bernoulli proposed the generalised version of the brachistochrone problem in the scientific journal *Acta Eruditorum* [2], cementing the brachistochrone problem in mathematical history. We will go through his approach to this problem.

3.1.2 Johann Bernoulli's Approach

In order to determine the shortest path between A and B, we will need to borrow Fermat's principle from physics. It says that for a ray of light travelling between two points A and B, the path taken will be the path of least time. From this principle, we can derive Snell's Law. Given two mediums which meet at a horizontal boundary and a ray of light travelling at speed v_1 in the first medium towards the boundary and travelling at speed v_2 once it has crossed the boundary, the following equation holds true (where θ_1 is the angle of incidence and θ_2 is the angle of refraction):

$$\frac{\sin\theta_1}{v_1} = \frac{\sin\theta_2}{v_2} \tag{3}$$

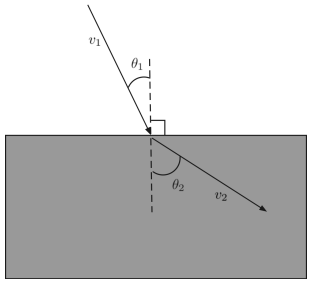

Figure 8: Snells Law

Now lets consider the Brachistochrone problem. When the particle P is at any point along a hypothetical curve Z, its kinetic energy is given by $\frac{1}{2}mv^2$ where m is the mass of the particle and v is its velocity. This kinetic energy is equivalent to the loss in gravitational energy from height A_y to the corresponding height of the particle P_y. Let $A_y - P_y = y$. The loss in gravitational potential energy is ymg where m is the mass of the particle and g is the gravitational constant. This means that $\frac{1}{2}mv^2 = ymg$. This gives the velocity as:

$$v = \sqrt{2gy} \tag{4}$$

If we imagine that the finite plane which our curve could possibly occupy is divided up into N horizontal layers where n_i corresponds to the ith layer. Within each layer n_i, the velocity of light in this material is given by v_i and the corresponding drop in height from A_y to the lower boundary of the layer is given by y_i. For example, n_1 is the first layer beginning at height A_y with light velocity v_1 and lower boundary height y_1. From equation 4, we know that in any layer n_i, $v_i \propto \sqrt{y_i}$. It follows that equation 3 can be written as:

$$\frac{\sin\theta_1}{\sqrt{y_1}} = \frac{\sin\theta_2}{\sqrt{y_2}} \tag{5}$$

We can now imagine that $N \to \infty$ where each layer is infinitely small. In the limiting

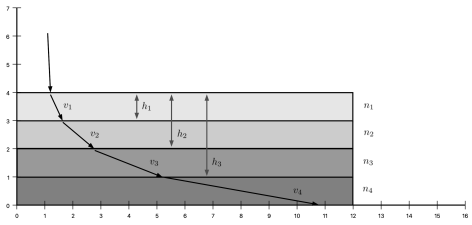

Figure 9: Snells Law for the Brachistochrone Problem

case, we can see that from equation 5:

$$\frac{\sin\theta_1}{\sqrt{y_1}} = \frac{\sin\theta_2}{\sqrt{y_2}} = \frac{\sin\theta_3}{\sqrt{y_3}} = \cdots = \frac{\sin\theta_i}{\sqrt{y_i}} = C \tag{6}$$

Where C is a constant. In other words, the sin of the angle θ_i between any tangent line along our curve and the vertical, divided by the square root of the loss in height from A_y to our curve at that point is always constant. Equation 6 can also be used to describe the path of a point on a circle moving along a horizontal. This is known as a cycloid. This means that we can use the path of a cycloid to solve the brachistochrone problem.

3.1.3 Installation

Before we can use the package, we need to install it. You will need to use a Windows operating system, ideally version 10 or greater. Do not attempt to install anything on a University computer as you will most likely have issues with permissions. You will also need to ensure that your default browser places downloaded files into the downloads folder. If you haven't changed this setting then it is set correctly by default. Your downloads folder should also be named "Downloads". This is the default for computers designed for English speakers but this will not be the case for other languages. You will need to install the following pieces of software in the order presented:

1. R Version 4.0.0 or greater [14].
2. R Studio Version 1.2.5042 or greater [6].
3. RTools4 [1].

After you have successfully installed all of these, open R Studio and create a new project. Once this has loaded, go the R terminal at the bottom of RStudio. You'll now need to install a package called devtools. You can do this with the following command:

```
# (1) install devtools
install.packages("devtools")
```

Lastly, you can now install the ga-package with the following command:

```
devtools::install_github("DavidBlairs/gapackage_submit", force = TRUE)
```

Unfortunately, I cannot predict all system configurations that may be used to run this package. While every effort has been made to ensure it will work on as many

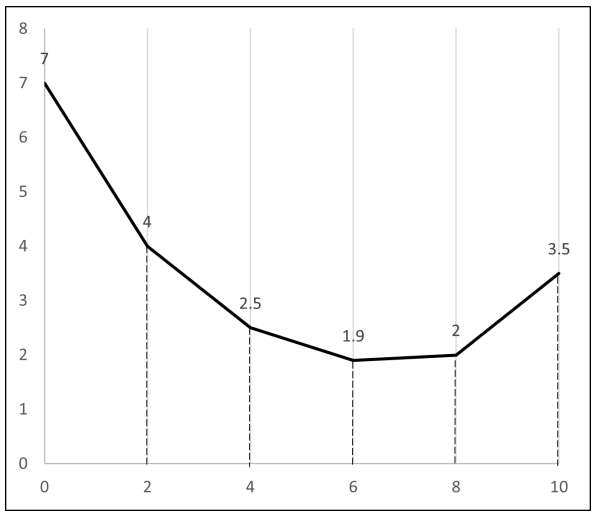

Figure 10: Brachistochrone Problem Curve Representation

systems as possible, this can't be garanteed. If you run into any issues, feel free to contact myself at 1807730@brunel.ac.uk.

In the remainder of this section, reference is made to the package documentation. You can find the full documentation for the ga-package here: https://davidblairs.github.io/

3.1.4 Genotype Representation

For this example, we will be representing each candidate curve as a 1 array, where each value represents the height of the curve at a set of fixed x coordinates. For example, the curve in figure 10 will be represented like so:

$$\boldsymbol{G} = \begin{pmatrix} 7 \\ 4 \\ 2.5 \\ 1.9 \\ 2 \\ 3.5 \end{pmatrix} \tag{7}$$

We will add the x coordinates to each curve when they are evaluated with the fitness function.

3.1.5 Implementation

The example used in this section can be run using the following command from the gapackage:

```
# run the brachistochrone example
gapackage::ga_example(name = "brachistochrone");
```

And if you would like to run the example with the UI, you can use the following command:

```
# run the brachistochrone example with UI
gapackage::ga_example(name = "ui");
```

Now that the package is installed and we can represent each of our genotypes, it can be imported using the `library()` function after which we can begin to construct our algorithm.

```
# (2) import the library
library(gapackage)
```

We first have to setup a new instance of a `gapackage::ga` class using the `new()` method along with a number of function parameters:

1. The `dim` argument is the dimensions of the entire population. In this example, we will be using functions that come built-in to the GA-Package which work in a 2D context.What this means is that each of our genotypes is represented by a 1D array and the entire population is a 2D array of these genotypes. `dim = c(10, 1000)` means that each genotype is of length 10 and the total population size is 1000.

2. The `store_data` parameter tells the algorithm whether it should store data in a data frame or a named list. Setting this to `FALSE` will improve performance when the UI is being used.

3. The `initial` parameter indicates the function that will be used to generate the initial population. This can be a custom function or one of the built-in functions. You can see that we have decided to use `initial_rand_uni_2d`.

```
# setup an instance of the ga class object
brach <- gapackage::ga$new(
  dim = c(10, 1000),
  parameters = list(
    initial_min     = 0,
    B_x             = 180,
    initial_max     = 10,
    A_y             = 40,
    geno_length     = 20,
    g               = 9.81,
    remove          = 0.5,
    add_proportion  = 0.5,
    mutation_size   = 0.0000007,
    generations     = 100,
    population_size = 1000,
    maximise        = FALSE,
    location        = 0
  ),
  store_data = FALSE,
  initial = initial_rand_uni_2d
);
```

Looking at the source code for `initial_rand_uni_2d`, we can see what this function does.

```
# generate an array, sampling from a uniform distribution
initial_rand_uni_2d = function(self, par){
  return(
    array(
      runif(
        par$geno_length * par$population_size,
        min = par$initial_min,
        max = par$initial_max
      ),
      dim = c(par$geno_length, par$population_size)
    )
  )
}
```

This function will create a `par$geno_length` by `par$population_size` sized array such that each element is sampled from a uniform distribution. The minimum and maximum values are `par$initial_min` and `par$initial_max` respectively. Note that these parameters are not passed explicitly as arguments to the function. Rather, they must be passed to the named parameters list when the class instance is initialised. When you use a built-in function, you should check the documentation to ensure that you know the parameters it uses and subsequently pass these to to the class.

Next we have to add the mutation and crossover operators using the `add_operators()` method. Operators are, in general, functions which take a population array and make some changes to it. They then return the amended array meaning they can be chained together. You can see that the `mutation_cauchy` operator doesn't have a suffix, indicating it works on a population of any dimensions. If we look at the source code (or the documentation) we can see what these operators do.

```
# mutate a population according to a cauchy distrubution
mutation_cauchy <- function(self, population, par){
  change <- array(
    rcauchy(prod(dim(population)), location = par$location, scale = par$mutation_size),
    dim = dim(population)
  );
  return(
    array(as.numeric(population) + as.numeric(change), dim = dim(population))
  )
}
```

This mutation operator will add an amount to each member of the population taken from a cauchy distribution with a location of `par$location` and a scale of `par$mutation_size`.

```
# apply a basic crossover algorithm to the population
crossover_basic_2d <- function(self, population, par){
  number_of_offspring <- as.integer(par$add_proportion * par$population_size);
  chosen_people <- population[,sample(1:ncol(population),
                        size = number_of_offspring * 2, replace = F)];

  offspring <- matrix(ncol = 0, nrow = nrow(population));
  for (genotype in 1:(ncol(chosen_people) / 2)){

    gen1 <- chosen_people[, genotype];
    gen2 <- chosen_people[, ncol(chosen_people) + 1 - genotype];

    current_offspring <- utility_crossover_breed_2d(gen1, gen2)
                            %>% matrix(ncol = 1, nrow = nrow(population));
    offspring <- cbind(offspring, current_offspring);
  }
  return(cbind(population, offspring))
}
```

The crossover algorithm we have chosen will first select (`par$add_proportion`)% of the genotypes in the population at random. It will then pair each of these genotypes together and use the `utility_crossover_breed_2d` to split and recombine segments of these two genotypes to produce two new genotypes to be added back to the population.

We can check the documentation for both of these operators and find that we need to pass two more parameters `par$location` and `par$add_proportion` which was done when the class was initialised.

```
# add mutation and crossover operators
brach$add_operators(
  operators = list(
    mutation_cauchy,
    crossover_basic_2d
  )
)
```

The next operator we need to add is the selection operator. The documentation says that this operator takes two parameters in addition to what we have already added, `par$remove` and `par$maximise`. As well as this, it also requires two dependency functions. To explain dependency functions, we can use the selection operator as our example. This operator will first run the named function `fitness_function_single` on each genotype of the population. This will then be passed to `fitness_function` which will create an array of these fitnesses. The selection operator will then use this information to decide which members of the population to carry to the next generation. The reason we want to use a separate function `add_dependents()` to add these functions is because we can use the other features of the genetic algorithm (such as the UI) to monitor what goes on inside of them. This structure is also useful when we want to alter the fitness function of a single genotype to be applied to a different problem with minimal amendments to our code.

```
# add dependancies for the selection operator
brach$add_dependents(
  dependents = list(
    fitness_function_single = utility_fitness_brachistochrone_2d,
    fitness_function = utility_fitness_population_2d
  )
)

# add the selection operator
brach$add_operators(
    selection_basic_2d
)
```

Lets take a look at how we have derived our fitness function. `utility_fitness_brachistochrone_2d` is a function designed to calculate the traversal time of a particle traveling along a curve in $\mathbb{R}^2$.

```
# calculate traversal time of a curve in a 2D context
utility_fitness_brachistochrone_2d <- function(self, genotype, par){
  # add the first and last height to the genotype
  complete_genotype <- c(par$A_y, genotype, 0);

  # calculate the drop in height from A_y to each height
  height_drop <- abs(par$A_y - complete_genotype);

  # determine the velocities (loss in GPE is gain in KE)
  velocities <- sqrt(2 * height_drop * par$g);
  total_time <- 0;

  # this is the length of the gap between heights
  x_diff <- abs(par$B_x / (length(genotype) + 1));

  # for each edge in our curve
  for (segment_index in 1:(length(genotype) + 1)){
    # calculate the difference in height between the two heights of the segment
    y_diff <- abs(complete_genotype[segment_index]
    - complete_genotype[segment_index + 1]);

    # get the coordinates of the segment
    first_coordinate  <- c(x_diff * (segment_index - 1),
    complete_genotype[segment_index]);
    second_coordinate <- c(x_diff * (segment_index),
    complete_genotype[segment_index + 1]);

    # calculate the distance between the coordinates
    distance <- sqrt(sum(abs(first_coordinate - second_coordinate)**2));

    # calculate acceleration in the direction of the segment
    acceleration <- par$g * (y_diff / x_diff);

    # determine the time according to the equations of motion
    total_time <- total_time +
    ((2 * distance) / (velocities[segment_index] + velocities[segment_index + 1]));

  }
  return(total_time)
}
```

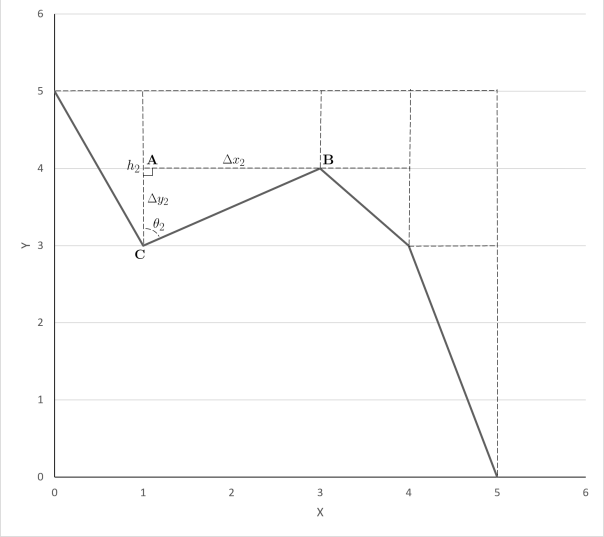

Figure 11: Calculation of the Traversal Time

We will use the example in figure 11 to explain the function, assuming that there is no air resistance and the particle isn't elastic. We begin with a 5x2 matrix representing the coordinates of our curve:

$$\mathbf{G} = \begin{pmatrix} 0 & 5 \\ 1 & 3 \\ 3 & 4 \\ 4 & 3 \\ 5 & 0 \end{pmatrix} \tag{8}$$

We will need to calculate the change in height from the starting position $(0, 5)$ to each coordinate:

$$\mathbf{H} = \begin{pmatrix} 5 \\ 5 \\ 5 \\ 5 \\ 5 \end{pmatrix} - \mathbf{G}_{1-5;2} = \begin{pmatrix} 5 \\ 5 \\ 5 \\ 5 \\ 5 \end{pmatrix} - \begin{pmatrix} 5 \\ 3 \\ 4 \\ 3 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \\ 1 \\ 2 \\ 5 \end{pmatrix} \tag{9}$$

The law on the conservation of energy says that the total energy of an isolated system will remain constant over time. Since the particle does not gain or loose any energy, the kinetic energy at each coordinate is equal to the loss in gravitational potential energy from the initial height:

$$KE = GPE \tag{10}$$

$$\frac{1}{2}m\mathbf{V}^{\circ 2} = gm\mathbf{H} \tag{11}$$

$$\mathbf{V} = (2g\mathbf{H})^{\circ \frac{1}{2}} \tag{12}$$

$$\mathbf{V} = \begin{pmatrix} 0 \\ 4g \\ 2g \\ 4g \\ 10g \end{pmatrix}^{\circ \frac{1}{2}} \tag{13}$$

Where g is the gravitational constant $g = 9.81m^2$ and m is the mass of the particle. Lets consider one segment of our curve, the line $\overline{CB}$ in figure 11. We first need to calculate the change in x and y. $\Delta x_2 = |3 - 1| = 2$ and $\Delta y_2 = |4 - 3| = 1$. From this, we can calculate the length of the segment $\overline{CB} = \sqrt{5}$. We will also need the acceleration of the particle at the point C in the direction of travel, towards the point B. The only force acting on the particle is gravity. Using this we can derive the acceleration a_2:

$$a_2 = g\ tan(\theta_2) \tag{14}$$

$$= g\frac{\Delta y_2}{\Delta x_2} \tag{15}$$

$$= \frac{1}{2}g \tag{16}$$

We now have what we need to calculate the traversal time along this segment. The initial velocity at the point C is $\mathbf{V}_2 = 4g$ and the final velocity at the point B $\mathbf{V}_3 = 2g$. Since acceleration is constant, we can use the kinematic equations to derive the traversal time t_2:

$$t_2 = \frac{2\,\overline{CB}}{\mathbf{V}_2 + \mathbf{V}_3} \tag{17}$$

$$= \frac{\sqrt{5}}{3g} \tag{18}$$

This process is repeated for each segment, giving us the total traversal time.

Lets move back to our script. We need to tell the algorithm what it is that we are interested in seeing. For example, in the brachistochone problem, we want to actually plot this curve for each generation and see if it is working correctly. To do this, we can use output graphs. We first need to define a function which takes three arguments. `self` will be your instance of the `gapackage::ga` class, `population` will be the population array and `par` will be a named list containing your parameters passed when you initialised the class. Your function should then return a named list containing x and y coordinates which will be used to create and update an output graph every generation.

```
# setup an output plot function
brachistochrone_plot <- function(self, population, par){
  genotype_vector <- as.vector(population[,ncol(population)]);
  x_values <- 1:length(genotype_vector)
  y_values <- genotype_vector;

  return(list(
    x = x_values,
    y = y_values
  ))
}
```

Now that the function has been created, we can use the `add_output_graph()` method to monitor what this output graph function looks over many generations. We can also pass some other parameters affecting the style of the graph.

```
# add the output graph
brach$add_output_graph(
  list(
    xlab = "horizontal␣distance",
    ylab = "vertical␣distance",
    FUN = brachistochrone_plot
  )
)
```

Before we move onto how to use the UI, it is important to note that we don't actually

need to use the UI to interact with the algorithm. If performance is important, you could use the following code to run the simulation on the R terminal and plot best the curve of the final generation (see figure **??**).

```
# run the simulation for 200 generations
for (i in 1:200){
  print(paste0("Current␣Generation:␣", i))

  brach$next_iteration()

  fitnesses <- brach$metrics$average_fitness;
}

# plot the best member
best <- brach$population[, dim(brach$population)[2]];
plot(1:length(best), best, type = "l");
```

This code will produce the curve seen in figure 12. You can see how the average fitness changed over time in figure 13.

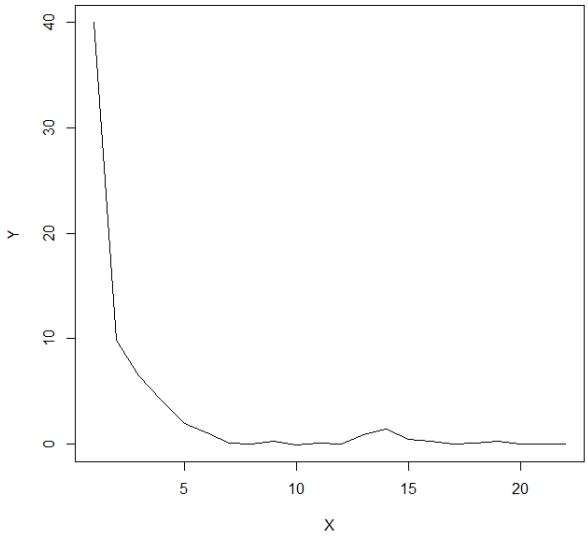

Figure 12: The Brachistochrone, more or less

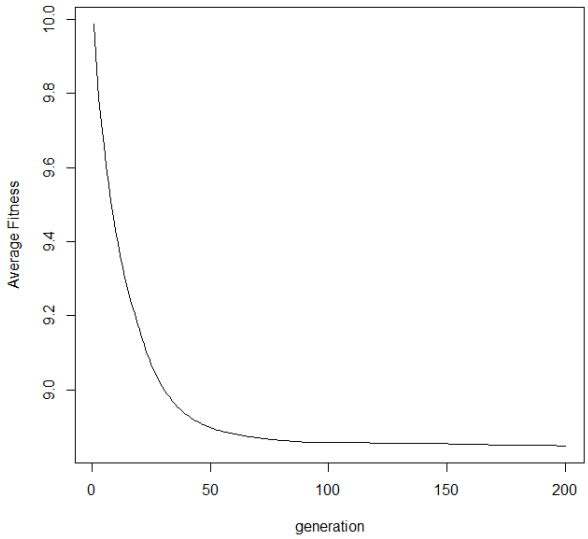

Figure 13: Change in Average Fitness Over Time

The `brach` instance contains a lot of attributes and methods to see whats going on and make changes to the simulation during execution. These can be found in the documentation. If instead we want to use the UI, we can run the following code:

```
# run the ui
print(brach$run_ui())
```

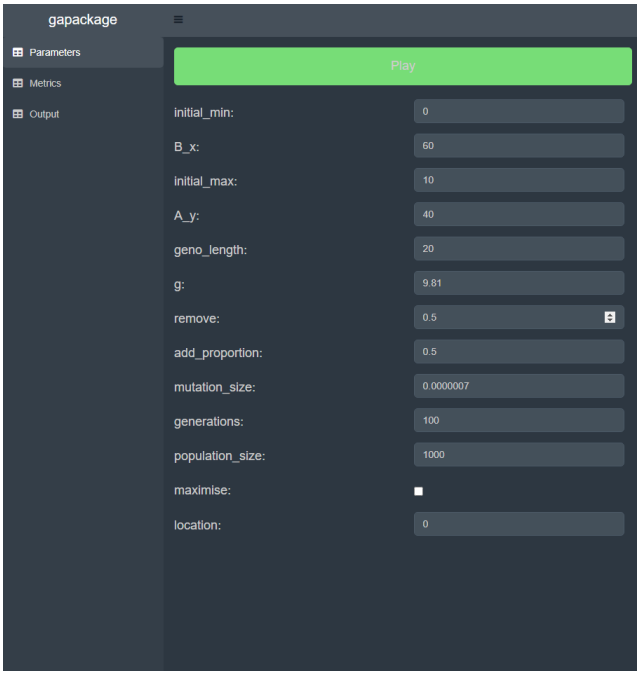

Figure 14: GA-Package Parameters Page

If you are running this in R Studio, the UI will appear in the viewer window by default and will look something like figure 14. Click on the Play button to begin the simulation and the Pause button to stop it.

You'll notice that all the parameters that we passed to the class object when it was initialised are now present on the parameters page. You can change any of these parameters, even during the execution of the simulation. Play the simulation and click on the metrics page 15.

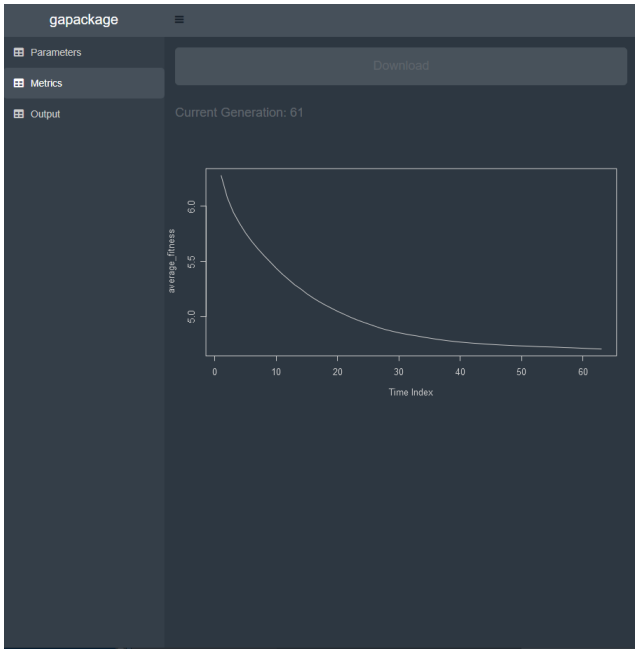

Figure 15: GA-Package Metrics Page

The metrics page allows you to view how certain values change during the course of the simulation. You'll see that the only metric on the page is `average_fitness`. This is the average fitness of our population at each generation. This graph will continue to update every 2 seconds while the simulation is running. If we look at one of our dependency functions, we can see why this metric graph appears and see how we can add any sort of graph we'd like. Lets look at the code for the dependency function `utility_fitness_population_2d`.

```
utility_fitness_population_2d <- function(self, population, par){
  # apply the dependency function on each genotype
  fitnesses <- apply(population, MARGIN = 2, FUN = function(genotype){
    return(self$dependents$fitness_function_single(self, genotype, par))
  })

  # add a metric to track
  self$add_metric(
    name = "average_fitness",
    value = sum(fitnesses) / length(fitnesses)
  );
  return(fitnesses)
}
```

This function is designed to apply a fitness function on each genotype of a population. It then collects all these fitnesses and returns them to the selection function. As was mentioned previously, by adding this function as a dependency, it is provided with a parameter `self` through which we can access functions in the ga-package object. If you want to add a metric somewhere in the genetic algorithm to appear in the UI, you can override the default function where you'd like to calculate the metric and use the `self$add_metric()` function to add the information to the algorithm. You can see in the function above how we have added the average fitness inside of the function. You can have as many metrics and output graphs as you'd like. You can also access the metrics data from the named list `self$metrics`. For this project, we will not be using the UI to help improve the performance.

3.2 Example: Non-Euclidean Distance

The following example can be run using the example function from the ga-package:

```
# run the non-euclidean-const example
gapackage::ga_example(name = "non-euclidean-const")
```

In the following example, we will be attempting the minimise the distance between two point in $\mathbb{R}^2$ under some relative transformation to the space. We will be using the same operators from the previous example however they will be working with a 3D population. The genotype representation of the curve will be pairs of coordinates. With the ga-package, this conversion is made easier. First, we change each suffix from `2d` to `3d` where applicable. Then we change `fitness_function_single` to the new fitness function that we want to use. We also need to make sure we add all the necessary param-

eters for this new function. In this case, we need to assign `par$non_euclidean_start`, `par$non_euclidean_end` and `par$non_euclidean_A`. We also need to modify the end of the code so we can plot the final curve for this type of population. Below is the final code:

```
# import gapackage
library(gapackage)

# setup an instance of the ga class object
  brach <- ga$new(
    dim = c(10, 1000),
    parameters = list(
      initial_min          = 0,
      initial_max          = 10,
      non_euclidean_start  = c(0, 0),
      non_euclidean_end    = c(10, 10),
      non_euclidean_A      = array(c(1, 0, 0, 1), dim = c(2, 2)),
      non_euclidean_bounds = c(0, 1),
      geno_length          = 3,
      remove_proportion    = 0.5,
      add_proportion       = 0.5,
      mutation_size        = 0.0000000007,
      generations          = as.numeric(settings$generations),
      population_size      = 1000,
      maximise             = FALSE,
      location             = 0
    ),
    store_data = FALSE,
    initial = initial_rand_uni_3d
  );

  # add mutation and crossover operators
  brach$add_operators(
    operators = list(
      mutation_cauchy,
      crossover_basic_3d
    )
  )

  # add dependancies for the selection operator
  brach$add_dependents(
    dependents = list(
      fitness_function_single = utility_fitness_non_euclidean_3d,
      fitness_function = utility_fitness_population_3d
    )
  )

  # add the selection operator
  brach$add_operators(
    selection_basic_3d
  )

  # run the simulation for 100 generations
  for (i in 1:brach$par$generations){
    cat(noquote((paste0("Current␣Simulation:␣", i))), "\n")
```

```
  # run next iteration
  brach$next_iteration()

  # print the average fitness for the current generation
  all_average_fitnesses <- brach$metrics$average_fitness;
  cat(noquote(paste0("Current␣Average␣Fitness:␣",
  all_average_fitnesses[length(all_average_fitnesses)])), "\n")
}

# plot the fitest member
fitest_member <- brach$population[,,dim(brach$population)[3]];

fitest_member <-
rbind(brach$par$non_euclidean_start, fitest_member, brach$par$non_euclidean_end)
plot(fitest_member[,1], fitest_member[,2], type = "l",
ylab = "Y", xlab = "X", main = "Non-Euclidean␣Distance")
```

Lets take a look at how we have derived the fitness function for this case.

Definition 1 *Let* $f : \mathbb{R}^2 \to \mathbb{R}$ *and* $\Phi : \mathbb{R}^2 \to graph(f) \subseteq \mathbb{R}^3$ *where* $\Phi(x, y) = (x, y, f(x, y)))$ *and both* f *is a smooth functions.* Φ *is invertible where* $\Phi^{-1} : graph(f) \to \mathbb{R}^2$ *and* $\Phi^{-1}(x, y, f(x, y)) = (x, y)$.

Proposition 1 *Because* f *is smooth,* Φ *is also smooth. Consequently, they are both infinitely differentiable.*

Definition 2 *Let* $\sigma : [0, 1] \to graph(f) \subseteq \mathbb{R}^3$ *with* $\sigma(t) = (\sigma_1(t), \sigma_2(t), \sigma_3(t))$. *The length of this curve* $l(\sigma)$ *in Euclidean space is given by:*

$$l(\sigma) = \int_0^1 \sqrt{\sigma_1^2(t) + \sigma_2^2(t) + \sigma_3^2(t)}\ dt \tag{19}$$

$$= \int_0^1 \sqrt{\sigma'(t) \cdot \sigma'(t)}\ dt \tag{20}$$

Proposition 2 *Let* $\gamma : [0, 1] \to \mathbb{R}^2$ *and set* $\sigma = \Phi \circ \gamma$. *From definition 2, the length of* $\sigma(t)$ *under the transformation is:*

$$l_\Phi(\sigma) = \int_0^1 \sqrt{(\Phi \circ \gamma)'(t) \cdot (\Phi \circ \gamma)'(t)}\ dt \tag{21}$$

Definition 3 *For two vectors in* $\mathbb{R}^2$, $\boldsymbol{u} = (u_1, u_2)$ *and* $\boldsymbol{v} = (v_1, v_2)$, *the dot product is* $\boldsymbol{u} \cdot \boldsymbol{v} = u_1v_1 + u_2v_2$. *However, we will need to use a more general form, the inner product to account for the transformation to the space. The inner product* $\langle \boldsymbol{u}, \boldsymbol{v} \rangle$ *must satisfy the following conditions:*

i $\langle \ . \ , \ . \ \rangle : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$

ii $\langle \boldsymbol{u}, \boldsymbol{u} \rangle \geq 0$ *and* $\langle \boldsymbol{u}, \boldsymbol{u} \rangle = 0 \iff \boldsymbol{u} = 0$

iii $\langle \alpha(\boldsymbol{u} + \boldsymbol{w}), \boldsymbol{w} \rangle = \alpha\langle \boldsymbol{u}, \boldsymbol{w} \rangle + \alpha\langle \boldsymbol{u}, \boldsymbol{w} \rangle$

iv $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \langle \boldsymbol{v}, \boldsymbol{u} \rangle$

Let $\boldsymbol{e_1} = (1, 0)$ *and* $\boldsymbol{e_2} = (0, 1)$ *be our base vectors in* $\mathbb{R}^2$ *and* $\langle \boldsymbol{u}, \boldsymbol{v} \rangle$ *is as follows:*

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle = u_1v_1(\boldsymbol{e_1} \cdot \boldsymbol{e_1}) + u_1v_2(\boldsymbol{e_1} \cdot \boldsymbol{e_2}) + u_2v_1(\boldsymbol{e_2} \cdot \boldsymbol{e_1}) + u_2v_2(\boldsymbol{e_2} \cdot \boldsymbol{e_2}) \tag{22}$$

$$= (u_1, u_2) \begin{pmatrix} \boldsymbol{e_1} \cdot \boldsymbol{e_1} & \boldsymbol{e_1} \cdot \boldsymbol{e_2} \\ \boldsymbol{e_2} \cdot \boldsymbol{e_1} & \boldsymbol{e_2} \cdot \boldsymbol{e_2} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \tag{23}$$

$$= (u_1, u_2) \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \tag{24}$$

$$= \boldsymbol{u}^\intercal \mathbb{I} \boldsymbol{v} \tag{25}$$

Note that in euclidean space, $\boldsymbol{e_1} \cdot \boldsymbol{e_2} = \boldsymbol{e_2} \cdot \boldsymbol{e_1} = 0$ and so $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = u_1v_1 + u_2v_2$ which is the dot product. Our inner product is now formed such that we can derive a matrix $\boldsymbol{A}$ from base vectors of our transformed space.

Definition 4 *Let* $\boldsymbol{A}$ *be a 2x2 real, symmetric and positive definite matrix such that* $\boldsymbol{A} : \mathbb{R}^2 \to Matrices(\mathbb{R}^2)$.

Definition 5 *Our inner product in a non-euclidean space transformed at any point* (x, y) *by* f *is given by* $\langle \boldsymbol{u}, \boldsymbol{v} \rangle_f(x, y) = \boldsymbol{u}^\intercal \boldsymbol{A}_f(x, y) \boldsymbol{v}$.

Our claim is that the expression used to calculate the length of $\gamma(t)$ in non-euclidean space (equation 21) can be expressed as an inner product (definition 5) containing the matrix $\boldsymbol{A}$. More formally:

$$(\Phi \circ \gamma)'(t) \cdot (\Phi \circ \gamma)'(t) = \langle \dot{\gamma}(t), \dot{\gamma}(t) \rangle_f (\dot{\gamma}(t)) = \dot{\gamma}(t) \boldsymbol{A}_f(\dot{\gamma}(t)) \dot{\gamma}(t) \tag{26}$$

Proposition 3 *The matrix $\boldsymbol{A}$ that satisfies equation 26 is the following:*

$$\boldsymbol{A}_f(x,y) = \begin{pmatrix} 1 + \frac{\partial f}{\partial x}(x,y)^2 & \frac{\partial f}{\partial x}(x,y)^2 \frac{\partial f}{\partial y}(x,y) \\ \frac{\partial f}{\partial x}(x,y) \frac{\partial f}{\partial y}(x,y) & 1 + \frac{\partial f}{\partial y}(x,y)^2 \end{pmatrix} \tag{27}$$

Proof: Let $v_1^{(x,y)}(t) = t\boldsymbol{e_1} + (x,y)$ and $v_2^{(x,y)}(t) = t\boldsymbol{e_2} + (x,y)$ be our base vector functions. These functions are chosen such that $\dot{v}_1^{(x,y)}(t) = \boldsymbol{e_1}$ and $\dot{v}_2^{(x,y)}(t) = \boldsymbol{e_2}$. Our formulation of the inner product in equation 23 gives some indication on how our $\boldsymbol{A}$ matrix will be formed. If we compose Φ with our base vector functions then we can extract a function which provides the base vectors of the transformed space at any point (x,y). This composition for $v_1^{(x,y)}(t)$ and $v_2^{(x,y)}(t)$ is $(\Phi \circ v_1^{(x,y)})'(0)$ and $(\Phi \circ v_2^{(x,y)})'(0)$ respectively. And, from this we can define our $\boldsymbol{A}$ matrix:

$$\boldsymbol{A}(x,y) = \begin{pmatrix} (\Phi \circ v_1^{(x,y)})'(0) \cdot (\Phi \circ v_1^{(x,y)})'(0) & (\Phi \circ v_1^{(x,y)})'(0) \cdot (\Phi \circ v_2^{(x,y)})'(0) \\ (\Phi \circ v_2^{(x,y)})'(0) \cdot (\Phi \circ v_1^{(x,y)})'(0) & (\Phi \circ v_2^{(x,y)})'(0) \cdot (\Phi \circ v_2^{(x,y)})'(0) \end{pmatrix} \tag{28}$$

Now we need to evaluate each of these expressions. Firstly, we can simplify v_1 and v_2 as $v_1^{(x,y)}(t) = (t + x, y)$ and $v_2^{(x,y)}(t) = (x, y + t)$ respectively. We will look at v_1 since the derivation is similar to that of v_2.

$$(\Phi \circ v_1^{(x,y)})'(0) = (t + x, y, f(t + x, y))'(0) \tag{29}$$

$$= (1, 0, \frac{d}{dt} f(t + x.y))(0) \tag{30}$$

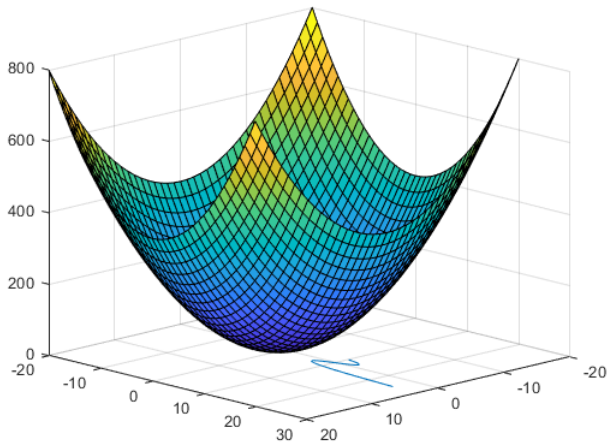

Figure 16: A random curve in blue and the mapping funtion $f(x, y) = x^2 + y^2$.

And given that by the chain rule:

$$\frac{d}{dt}f(x(t), y(t)) = f_x(x(t), y(t))\dot{x}(t) + f_y(x(t), y(t))\dot{y}(t) \tag{31}$$

The expression becomes:

$$(\Phi \circ v_1^{(x,y)})'(0) = (1, 0, \frac{\partial f}{\partial x}(x, y)) \tag{32}$$

And when we evaluate the dot product, our entry for row 1 column 1 of the $\boldsymbol{A}$ *matrix becomes:*

$$(\Phi \circ v_1^{(x,y)})'(0) \cdot (\Phi \circ v_1^{(x,y)})'(0) = (1, 0, \frac{\partial f}{\partial x}(x, y)) \cdot (1, 0, \frac{\partial f}{\partial x}(x, y)) \tag{33}$$

$$= 1 + \frac{\partial f}{\partial x}(x, y)^2 \tag{34}$$

Similar calculations will yield the other entries in the table, verifying proposition 3.

Lemma 3.1 *The following equation is true:*

$$(\Phi \circ \gamma)'(t) \cdot (\Phi \circ \gamma)'(t) = \dot{\sigma}_1^2(t) + \dot{\sigma}_2^2(t) + (\frac{\partial}{\partial x}f(\sigma(t))\dot{\sigma}_1(t) + \frac{\partial}{\partial y}f(\sigma(t))\dot{\sigma}_2(t))^2 \tag{35}$$

Proof:

$$(\Phi \circ \gamma)(t) = (\sigma_1(t), \sigma_2(t), f(\sigma(t))) \tag{36}$$

$$(\Phi \circ \gamma)'(t) = (\dot{\sigma}_1(t), \dot{\sigma}_2(t), \frac{d}{dt} f(\sigma(t))) \tag{37}$$

$$= (\dot{\sigma}_1(t), \dot{\sigma}_2(t), \frac{\partial}{\partial x} f(\sigma(t))\dot{\sigma}_1(t) + \frac{\partial}{\partial y} f(\sigma(t))\dot{\sigma}_2(t)) \tag{38}$$

$$(\Phi \circ \gamma)'(t) \cdot (\Phi \circ \gamma)'(t) = \dot{\sigma}_1^2(t) + \dot{\sigma}_2^2(t) + (\frac{\partial}{\partial x} f(\sigma(t))\dot{\sigma}_1(t) + \frac{\partial}{\partial y} f(\sigma(t))\dot{\sigma}_2(t))^2 \tag{39}$$

Proposition 4 *We can now prove our claim that* $(\Phi \circ \gamma)'(t) \cdot (\Phi \circ \gamma)'(t) = \dot{\gamma}(t)\boldsymbol{A}_f(\gamma(t))\dot{\gamma}(t)$.

RHS:

$$\dot{\gamma}(t)\boldsymbol{A}(\gamma(t))\dot{\gamma}(t) = (\dot{\gamma}_1(t), \dot{\gamma}_2(t)) \begin{pmatrix} 1 + \frac{\partial f}{\partial x}(x,y)^2 & \frac{\partial f}{\partial x}(x,y)^2 \frac{\partial f}{\partial y}(x,y) \\ \frac{\partial f}{\partial x}(x,y)\frac{\partial f}{\partial y}(x,y) & 1 + \frac{\partial f}{\partial y}(x,y)^2 \end{pmatrix} \begin{pmatrix} \dot{\gamma}_1(t) \\ \dot{\gamma}_2(t) \end{pmatrix} \tag{40}$$

$$= (\dot{\gamma}_1(t) + \dot{\gamma}_1(t)\frac{\partial}{\partial x} f(\gamma)^2 + \dot{\gamma}_2(t)\frac{\partial}{\partial x} f(\gamma(t))\frac{\partial}{\partial y} f(\gamma(t)), \tag{41}$$

$$\dot{\gamma}_2(t) + \dot{\gamma}_2(t)\frac{\partial}{\partial x} f(\gamma(t))^2 + \dot{\gamma}_1(t)\frac{\partial}{\partial x} f(\gamma(t))\frac{\partial}{\partial y} f(\gamma(t))) \begin{pmatrix} \dot{\gamma}_1(t) \\ \dot{\gamma}_2(t) \end{pmatrix} \tag{42}$$

$$= \dot{\gamma}_1(t)^2 + \dot{\gamma}_1(t)^2 \frac{\partial}{\partial x} f(\gamma(t))^2 + 2\dot{\gamma}_1(t)\dot{\gamma}_2(t)\frac{\partial}{\partial x} f(\gamma(t))\frac{\partial}{\partial y} f(\gamma(t)) \tag{43}$$

$$+ \dot{\gamma}_1(t)\dot{\gamma}_2(t) + \dot{\gamma}_2(t)^2 + \dot{\gamma}_2(t)^2 \frac{\partial}{\partial y} f(\gamma(t))^2 \tag{44}$$

$$= \dot{\gamma}_1^2(t) + \dot{\gamma}_2^2(t) + (\frac{\partial}{\partial x} f(\gamma_1(t), \gamma_2(t))\dot{\gamma}_1(t) + \frac{\partial}{\partial y} f(\gamma_1(t), \gamma_2(t))\dot{\gamma}_2(t))^2 \tag{45}$$

Which is equivalent to the result derived in lemma 3.1. Therefore, our claim that $(\Phi \circ \gamma)'(t) \cdot (\Phi \circ \gamma)'(t) = \dot{\gamma}(t)\boldsymbol{A}_f(\gamma(t))\dot{\gamma}(t)$ is true. From this, we can redefine the non-euclidean length:

$$l(\sigma) = \int_0^1 \sqrt{\langle \sigma'(t), \sigma'(t) \rangle_f(x,y)}\, dt \tag{46}$$

$$= \int_0^1 \sqrt{\sigma'(t)\boldsymbol{A}_f(\sigma(t))\sigma'(t)^\intercal}\, dt \tag{47}$$

Lets look at the programmatic implementation of this in the function `utility_fitness_non_euclidean_3d`. At the begining, we used the identity matrix for our `par$non_euclidean_A` parameter. If the curve between two points is a straight line, then this can verify that the implementation is correct. The code below is an interpretation of equation 47 however we are treating time as discrete rather than continuous.

```
# function to calculate non euclidean distance
utility_fitness_non_euclidean_3d <- function(self, genotype, par){
  complete_genotype <- rbind(par$non_euclidean_start, genotype, par$non_euclidean_end);

  if (is.null(par$non_euclidean_A)){
    par$non_euclidean_A <- array(c(1, 0, 0, 1), dim = c(2, 2));
  };

  bounds <- par$non_euclidean_bounds;
  curve <- complete_genotype;

  differential <- differentiate(genotype);

  time_interval <- (bounds[2] - bounds[1]) / (dim(curve)[1] - 1);
  time_stamps   <- ((1:dim(curve)[1]) - 1) * time_interval;

  differential <- array(append(time_stamps, differential), dim = dim(curve));

  total_area <- 0;
  for (time_index in 1:(dim(curve)[1] - 1)){
    current_gradient <- differential[time_index, ];
    A_component <- current_gradient %*% par$non_euclidean_A;
    magnitude   <- sqrt(sum(A_component * current_gradient));

    local_area <- magnitude * time_interval;
    total_area <- total_area + local_area;
  }
  return(total_area)
}
```

Using the identity matrix as A will result in the curve shown in figure 17. Because this graph is a straight line, we can now replace our matrix A with a matrix function that changes depending on the location in the space. We will use $f(x, y) = x^2 + y^2$ to define the matrix A:

$$A_f(x,y) = \begin{pmatrix} 1 + \frac{\partial f}{\partial x}(x,y)^2 & \frac{\partial f}{\partial x}(x,y)^2 \frac{\partial f}{\partial y}(x,y) \\ \frac{\partial f}{\partial x}(x,y) \frac{\partial f}{\partial y}(x,y) & 1 + \frac{\partial f}{\partial y}(x,y)^2 \end{pmatrix} = \begin{pmatrix} 1 + 4x^2 & 4xy \\ 4xy & 1 + 4y^2 \end{pmatrix} \tag{48}$$

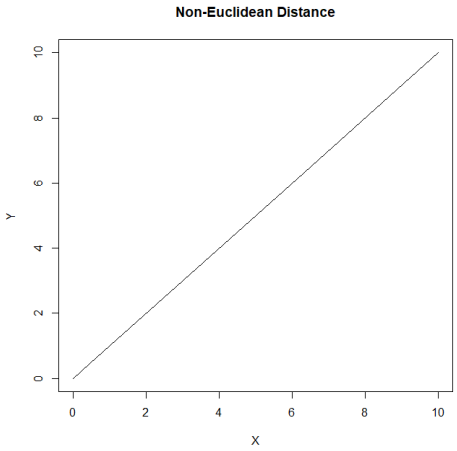

Figure 17: Non-Euclidean Distance: Using an Identity Matrix

```
# calculate the relative A matrix for the function f(x, y) = x^2 + y^2
get_A <- function(coordinate){
x <- coordinate[1];
y <- coordinate[2];

return(array(c(
  1 + (4*(x^2)), 4*x*y, 4*x*y, 1 + (4*(y^2))
), dim = c(2, 2)))
}
```

We've replaced the parameter `par$non_euclidean_A` with the function `get_A()` inside of the fitness function, which will return the corresponding matrix A for a given coordinate. The coordinate we are using is the average of the coordinates on each side of the segments which make up the curve. The `par$non_euclidean_start` and `par$non_euclidean_end` will be $(0, 10)$ and $(10, 0)$ respectively. You can run this simulation using the `ga_example()` function from the ga-package:

```
# run the non-euclidean-var example
gapackage::ga_example(name = "non-euclidean-var")
```

The simulation will result in the curve shown in figure 18. You can see that it curves around the parabolic shape of the function. I've taken the data for this curve and used MATLAB to plot the curve on the function to make it more clear what is going on. You can see this in figure 19. The blue line represents the shortest Euclidean distance between the two coordinates. The red line shows the shortest distance between the two coordinates where the space is transformed according the function $f(x, y) = x^2 + y^2$.

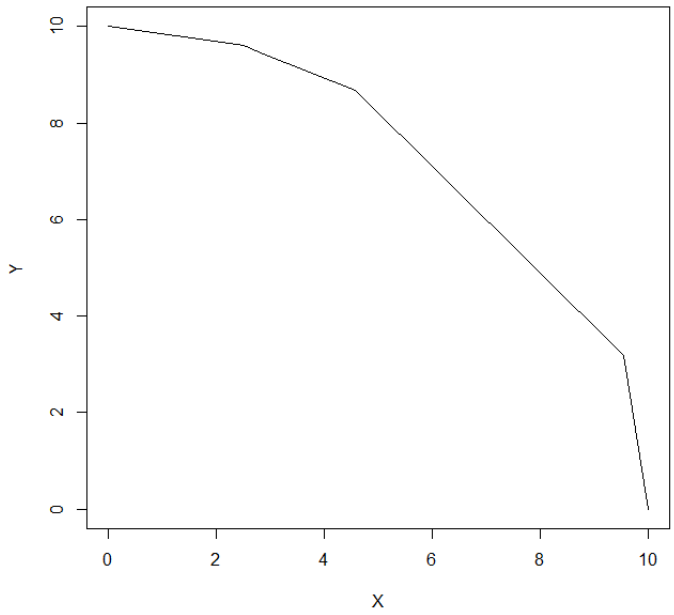

Figure 18: Non-Euclidean Distance: Relative A Matrix Results

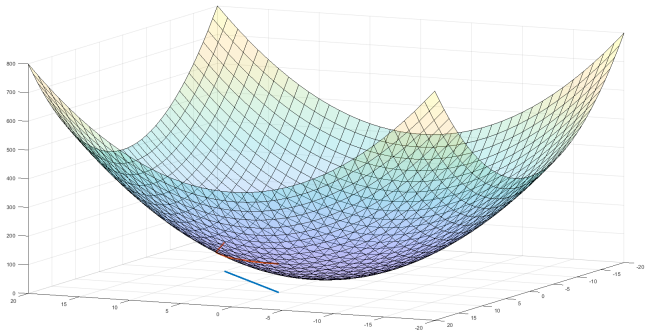

Figure 19: Euclidean Distance (Blue) / Non-Euclidean Distance (Red) on Function $f(x, y) = x^2 + y^2$

3.3 Example: Distance Metrics

During this project, we explored a number of different distance metrics to test the capabilities of the ga-package. A distance metric is an alternative way of calculating the distance between points. We will explore the implementation of these examples and take a look at the results.

For each of these examples, we have used the standard genetic algorithm construction described previously however we will be modifying the `fitness_function_single` with our alternative distance functions and using functions with a 3D population. Each example can be run using the `ga_example()` function in the ga-package. Details on this will be found at the beginning of each example.

3.3.1 Manhattan Distance

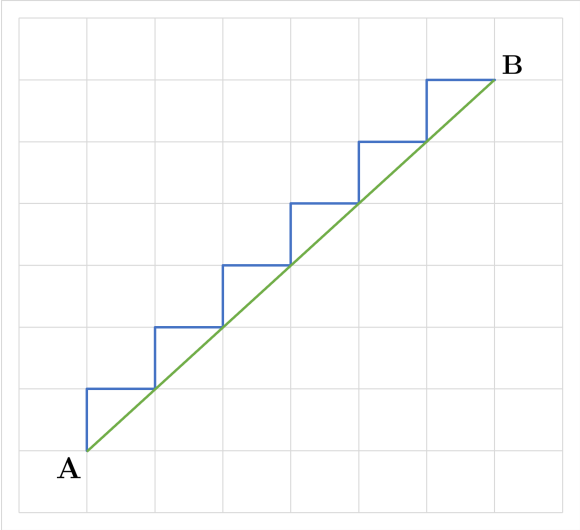

Figure 20: Euclidean Distance (in green) and the Manhattan Distance (in blue)

You can run the following example using the `ga_example()` function from the gapackage:

```
# run the manhattan example
gapackage::ga_example(name = "manhattan")
```

The Manhattan Distance or Taxi-Cab Distance is a distance metric in which the value is the sum of the absolute differences of their Cartesian coordinates. If we wanted

to calculate the shortest euclidean distance d_e between the point $\mathbf{A} = (A_x, A_y)$ and $\mathbf{B} = (B_x, B_y)$, we would use the Pythagorean theorem:

$$d_e = \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2} \tag{49}$$

However, if we want to calculate the Manhattan Distance d_m we need to modify the formula slightly:

$$d_m = |A_x - B_x| + |A_y - B_y| \tag{50}$$

In our implementation, we represent each of our genotypes as a 2D matrix where each row is a pair of coordinates. Firstly we have to add the start and end coordinates. This is to ensure that the coordinates of our curve don't converge onto a single point with a Manhattan Distance of 0. We then use equation 50 to calculate the Manhattan distance for each segment, the total of which is the overall fitness of our genotype. The following function will be passed as the `fitness_function_single` dependency.

```
# calculate the manhattan distance
utility_function_manhattan_3d <- function(self, genotype, par){
  distance <- 0;
  genotype_complete <- rbind(c(0, 40), genotype, c(40, 0));

  for (segment_index in 1:(dim(genotype_complete)[1] - 1)){
    diff <-abs(genotype_complete[segment_index, ] - genotype_complete[segment_index + 1, ]);
    distance <- distance + sum(diff);
  }
  return(distance)
}
```

From figure 20, you can see the coordinates which result in the shortest Euclidean Distance can be the same as the coordinates that result in the shortest Manhattan Distance. When the genetic algorithm is attempting to minimise euclidean distance, each adjustment to the curve decreases its length and improves the fitness. However, the Manhattan distance function encounters many more local minimums on its way towards the optimal solution. If we change the minimum and maximum values used to sample the initial population to 0 and 10 respectively, we can see this behaviour (see

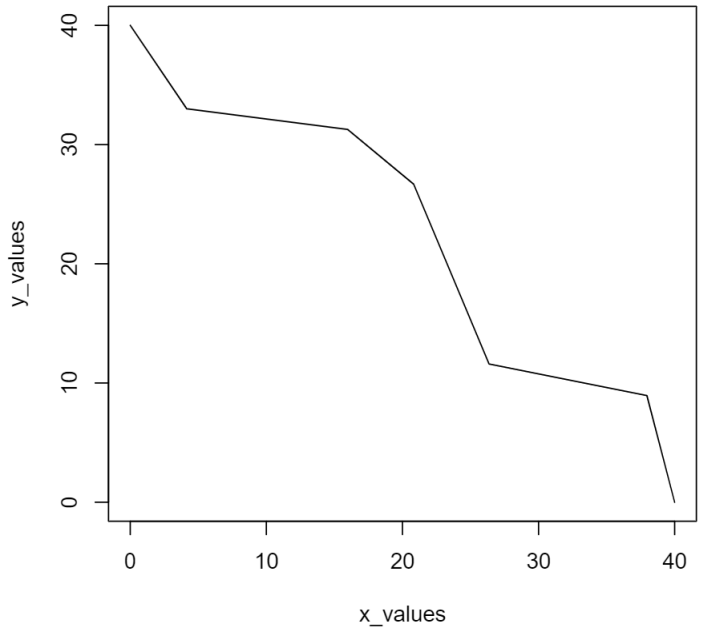

Figure 21: Manhattan Distance: Fittest Member of the Final Generation

figure 22). This is a common problem in tasks where the optimum solution is not be known.

3.3.2 Minkowski Distance

The following example can be run using the `ga_example()` function from the gapackage:

```
# run the minkowski example
gapackage::ga_example(name = "minkowski")
```

If we compare the Euclidean Distance with the Manhattan Distance, a pattern starts to emerge. We can generalise the pattern with the following formula, known as the Minkowski Distance:

$$d_k = (\sum_{i=1}^{n} |x_i - y_i|^p)^{\frac{1}{p}} \tag{51}$$

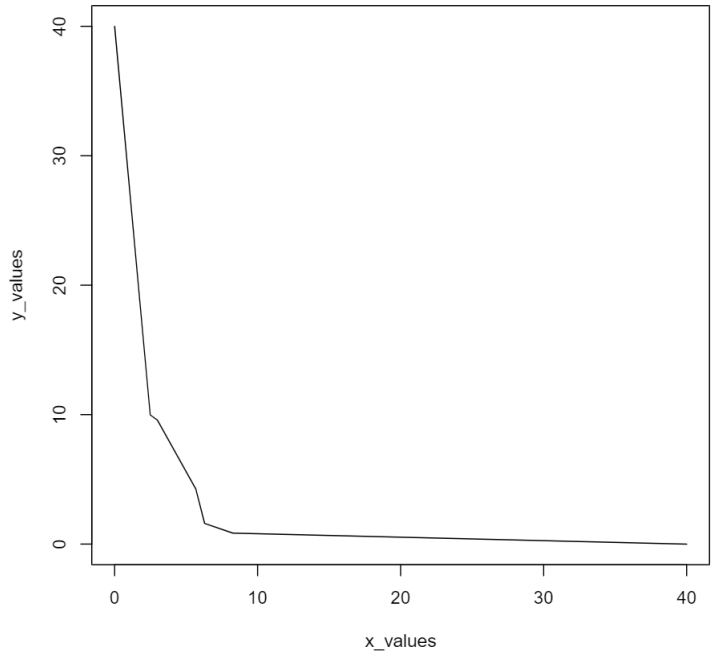

Figure 22: Manhattan Simulation at a local minimum

We have implemented this in a similar way to the manhattan distance however, we have changed the fitness function to the code below. When you run the example, you can choose a parameter p for the equation.

```
# function to calculate the minkowski distance
utility_fitness_minkowski_3d <- function(self, genotype, par){
  complete_genotype <- rbind(par$minkowski_start, genotype, par$minkowski_end);

  total_distance <- 0;
  for (segment_index in 1:(nrow(complete_genotype) - 1)){
    first_coordinate <- complete_genotype[segment_index, ];
    second_coordinate <- complete_genotype[segment_index + 1, ];

    diff <- abs(second_coordinate - first_coordinate);

    total_distance <- total_distance + sum(diff^par$minkowski_p)^(1/par$minkowski_p);
  }
  return(total_distance)
}
```

4 Physics Simulation

4.1 Justification and Issues

Previously, we have been using fitness functions that were derived according the specific problem that they were being designed for. For example, in the brachistochrone problem, we derived a function which could calculate the time taken to traverse a curve in $\mathbb{R}^2$. The question is, can we introduce a programmatic way to evaluate the fitness of each genotype such that the same method could be used on many other problems? This was our inspiration for using a physics engine. Lets consider the brachistochrone problem. We can setup the simulation environment such that we have a curve and a ball. Then, we can run the simulation and see how long it takes for the ball to traverse the curve. This process can then be repeated for each genotype for each generations.

In my project plan, I described how I had conducted some performance tests on our chosen physics engine, Matter.js [11]. While these tests demonstrated that you could place a very high number of bodies within the simulation, I didn't test how the physics engine would perform when the simulation speed was increased. The problem here is that when the simulation speed is high, objects in the simulation with a high velocity are moving very fast across the screen. When they encounter another stationary object, there is a chance they will overlap as the position of the moving object is calculated in discrete time intervals rather than continuously. Once the simulation has caught up, it doesn't know whether the moving object is on one side of the stationary object or the other and so more often than not, the moving object will teleport through the stationary object. I tried a number of different solutions. Firstly, I tried to manually update the screen at smaller time intervals. I also tried to change some parameters in Matter.js, notably the `engine.positionIterations` and `engine.velocityIterations`. The last solution I tried was using another javascript library called polydecomp. The idea was to create a terrain representing our curve rather than rectangular segments. The thought here was that even if the moving object overlapped the stationary terrain, the large size of the terrain would force it back on to the correct size. Sadly, none of these solutions worked. This problem seems to be common to the Matter.js library. The better solution would be to use a different, low-level physics engine but time constraints sadly meant this wasn't possible.

Another problem we encountered was trying to unify R Shiny and Matter.js. R

Shiny is based around the idea of reactivity. This means that it triggers pieces of code when certain events occur. Matter.js on the otherhand works in a game loop, updating the screen at 60 frames per second. As well as this, since these two systems are treated as two seperate processes, we need a way to hault R Shiny while Matter.js is running. However, this leads to further issues. If we hault R Shiny temporarily, no events will be triggered and we won't have a simple way of communicating the duration from javascript back to R Shiny. While these were some of the major problems, there are also a number of small subtle issues that arose while writing this script.

We will now talk through how this was implemented. It is important to note that this is very much a "proof of concept" and could never be used for anything practical. We will talk later about the many, many ways in which this could be improved.

4.2 Implementation

You can run this example using the `ga_example()` function from the gapackage:

```
# run the physics simulation
gapackage::ga_example(name = "matter")
```

Make sure to check the following:

1. The downloads folder on your default browser is named "Downloads". This is case sensitive.
2. You are using a windows PC.
3. You haven't changed where your browser downloads files.
4. When you run the example, your browser will ask you if you'd like to allow multiple downloads. Make sure you click "yes".

To implement this simulation, we have used the example of the Brachistochrone problem described previously. Each curve will be represented as a 1D array of real numbers corresponding to the height of the curve at predetermined x coordinates.

Within the R script, we will be using the ga-package to handle the genetic algorithm itself. The first thing we need to do is import a number of libraries. We will be using

R Shiny to present the physics engine and shinyjs to communicate to javascript from R. The stringr library will help with communicating from javascript back to R Shiny.

```
# load required packages
library(shiny)
library(shinyjs)
library(htmltools)
library(gapackage)
library(stringr)
```

We will be setting up a standard genetic algorithm which will be working in a 2D context. We will use the `initial_rand_uni_2d` to generate our initial population. The mutation and crossover operators will be `mutation_cauchy` and `crossover_basic_2d` respectively. We have already discussed what these functions do and so we won't go into this here. You can see that we have forced the first member of the population to be a specific set of heights. This will setup the simulation such that our ball will reach the end of the curve, allowing us to clearly see the simulation in action.

```
# setup an instance of the ga class object
brach <- gapackage::ga$new(
  dim = c(10, 1000),
  parameters = list(
    initial_min     = 0,
    B_x             = 180,
    initial_max     = 300,
    A_y             = 500,
    geno_length     = 5,
    g               = 9.81,
    remove          = 0.5,
    add_proportion  = 0.5,
    mutation_size   = 0.0000007,
    generations     = 100,
    population_size = 1000,
    maximise        = FALSE,
    location        = 0
  ),
  store_data = FALSE,
  initial = initial_rand_uni_2d
);

brach$population[,1] <- c(400, 300, 200, 100, 0)

# add mutation and crossover operators
brach$add_operators(
  operators = list(
    mutation_cauchy,
    crossover_basic_2d
  )
)
```

Next will will be defining a number of functions. What these functions will do is

allow us to read, write and delete a text file located in the same directory as the R script. We will demonstrate later why these are important but for now, the best way to think of these functions is that they allow us to reliably communicate between different pieces of code without using the global namespace. We can also trigger events when these files are created.

```
# remove local file
remove_file <- function(name){
    if (file.exists(name)){
        file.remove(name)
    }
}

# write text to a file
write_file <- function(name, text){
    remove_file(name = name);

    file_conn <- file(name)
    writeLines(c(text), file_conn)
    close(file_conn)
}

# get the contents of a file
get_file_contents <- function(name){
    file_conn <- file(name);
    return(readLines(file_conn))
}
```

The following function is created for a similar reason to those described above. The difference is that this function is designed to allow for easy communication between javascript and R in cases where the R script has been halted. It will search through the downloads folder for files which could indicate that the simulation has ended and then returns these files to the user. The reason for this will become much clearer later on.

```
get_sim_end_recent <- function(){
  path_root <- "C:/Users/david.blair/Downloads/simulation_end*.txt";
  matches <- Sys.glob(path_root);
  return(matches[length(matches)])
}
```

Below is a piece of code that may be familiar. What we have done is add the necessary dependencies to our genetic algorithm to allow it to determine the fitness of each member of the population. Each time the genetic algorithm does this, it will run `fitness_function_single` after which it will provide a fitness score which in this case will be the time it took for the ball to traverse the curve. The fitness function we have created is called `utility_fitness_simulation`. We will take a look at how this function runs the simulation but first we need to look at the javascript we have written

to aid in this process. But before we do that, lets take a look at the UI code.

```
# add dependancies for the selection operator
brach$add_dependents(
  dependents = list(
    fitness_function_single = utility_fitness_simulation,
    fitness_function = utility_fitness_population_2d
  )
)
```

Within the UI, we need to import all of the javascript code that we will need and setup the elements on the page. The first thing we need to do is import the Matter.js library and our own javascript code in that order. We also need to create a HTML canvas element to display the simulation, making sure to give it an ID that we can reference later. R Shiny doesn't support the canvas element (yet) and so we need to explicitly pass HTML code using the `HTML()` function.

```
# generate the UI
ui <- fluidPage(
    shiny::includeScript("www/matter.js"),
    shiny::includeScript("www/matter_example.js"),
    HTML("<canvas␣id␣=␣'matterjs-canvas'></canvas>")
)
```

On the javascript side, we want to make sure that we don't try and do anything until R Shiny has fully loaded. The following code is wrapped around our entire script and ensures that nothing will run until a successful connection has been made to our local R Server.

```
// only run once connected
$(document).on("shiny:connected", function() {
    // rest of the code is here
}
```

We next need to locate the canvas element on which our simulation will be displayed. You can see that we are using the standard javascript function `document.getElementById()` to locate the element with the ID 'matterjs-canvas' which we defined in our UI earlier. Next we need to set the dimensions of this canvas as 800x800 pixels.

```
// get the canvas element
var canvas = document.getElementById('matterjs-canvas');

// set the width and height
var width = 800,
  height = 800;

canvas.width = width;
canvas.height = height;
```

Now we have to setup the engine that will perform the actual calculations for the simulation. It requires one parameter, the canvas element that we will be using. This is so it can understand the environment that it is working with. As well as this, we define a number of shorthand aliases to make things easier later on.

```
// module aliases
var Engine = Matter.Engine,
  Render = Matter.Render,
  Runner = Matter.Runner,
  Body = Matter.Body,
  Events = Matter.Events,
  Composites = Matter.Composites,
  Common = Matter.Common,
  Constraint = Matter.Constraint,
  MouseConstraint = Matter.MouseConstraint,
  Mouse = Matter.Mouse,
  Composite = Matter.Composite,
  Bodies = Matter.Bodies,
  World = Matter.World;

// create an engine
var engine = Engine.create(canvas);
```

Once we have created the engine, we need a way to be able to convert these physics based calculations into objects we can see on the screen. This job is given to the renderer which is created below.

```
// create a renderer
var render = Render.create({
  canvas: canvas,
  engine: engine,
  width: 800,
  height: 800
});

// run the renderer
Render.run(render);
```

The following two lines are parameters within Matter.js that will hopefully improve

performance at the expense of CPU time. They increase the number of position and velocity calculations performed every second.

```
engine.positionIterations = 100000;
engine.velocityIterations = 100000;
```

The next piece of code will setup the game loop. I haven't evaluated the expressions describing the delay between game ticks because it makes it easier to add a number and overclock the simulation. However, we haven't done this in our implementation due to the issues I described earlier.

```
// run the engine
  setInterval(function() { Engine.update(engine, 1000 * 5 / 60); }, (1000 / 60));
```

In the event that we want to debug our simulation, we will need to be able to interact with the objects in the environment using the mouse. For example, we can move objects such that they trigger certain events we can look out for. To do this, we can add a `MouseContraint` to the engine with the following code.

```
// add mouse control
var mouse = Mouse.create(render.canvas),
  mouseConstraint = MouseConstraint.create(engine, {
      mouse: mouse,
      constraint: {
          stiffness: 0.1,
          render: {
              visible: false
          }
      }
  });

Composite.add(engine.world, mouseConstraint);
```

Now that the simulation is setup, we can begin to add a number of bodies to the environment. The first body is a tall, thin rectangle which will be static and sit on the right side of the canvas. This body will be used to detect when the ball in our simulation as finished travelling along the curve.

After we have created this body, we can add it to the world. To do this, we use the `Composite.add()` function. Each Matter.js world is called a composite. A composite itself is a collection of other composites. The idea behind this is to provide a hierarchical structure to all the objects in the simulation, making navigation much easier for Matter.js. For those familiar with web development, this structure is analagous to the DOM structure present in HTML files.

```
right_segment = Bodies.rectangle(755, 0, 1, 1600, {
  collisionFilter: { group: group },
  isStatic: true,
  label: "right_boundary"
}
);

Composite.add(engine.world, right_segment);
```

For the remainder of the code, we are using a Shiny feature known as message handlers. When we want to send information from Shiny to Javascript, we first need to define a custom message handler. We also need to provide a name and a function which will be executed on the javascript side. Take the code below as an example. This will define a new message handler named `set_speed` which will change the speed of the simulation to the parameter `speed`.

```
// set simulation speed
Shiny.addCustomMessageHandler("set_speed", function(speed){
engine.timing.timeScale = speed[0];
})
```

And once this is defined on the javascript side, we can then refer to it on the R side using the code below. Note that, there is not a direct relationship between R data types and JavaScript data types meaning that the data communication is standardised using a JSON file structure.

```
# change the speed of the simulation
session$sendCustomMessage("set_speed", shiny:::toJSON(c(0.2)))
```

Lets take a look at one of the message handlers that we have defined. We may not be decribing them in the order they were defined in the JavaScript file but this isn't particularly important for the execution of the code. We will be defining a new message handler that allows us to turn a 1D array of heights into bodies which can represent the curve in the simulation. We will go through this piece by piece. We start by setting up the message handler.

```
 // add curve
  Shiny.addCustomMessageHandler("add_curve", function(points){
    // rest of the code goes here
  })
```

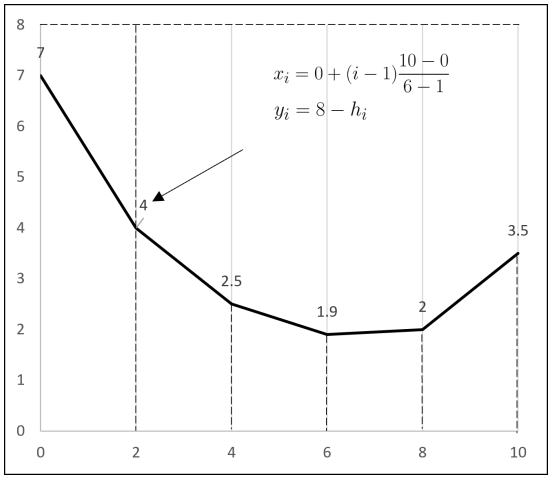

Figure 23: Calculation of the Curve Coordinates

We next need to setup a number of local variables. `x_values` and `y_values` will be used to store the location of the centre of mass of each of the segments that make up our curve. `length_values` will store the length of these segments after rotation and `theta_values` will store the angle of rotation of each segment in radians.

```
// variables to save all calculated values
let x_values      = [],
    y_values      = [],
    length_values = [],
    theta_values  = [];
```

An important thing to note about Matter.js is that the coordinates do not start in the bottom left corner. Rather, they start in the top left. This is because graphical engines render each pixel value in your computer starting from the top left corner. It seems that they didn't have mathematics in mind when they developed this. In order to compensate for this, we use the following piece of code to calculate the x and y coordinates. The x coordinates are calculated at even intervals between two predefined points. You can see a diagram explaining these calculation in figure 23.

```
let n = points.length;

// calculate x and y coordinates of all the points
for (let i = 1; i <= n; i++){
  // calculate x / y
  let x = 50 + ((i - 1) * (700 / (n - 1))),
      y = 525 - points[i - 1];

  // populate arrays
  x_values[i - 1] = x,
  y_values[i - 1] = y;
}
```

Once we have the coordinates of the start and end of our segments, we need to determine their rotational component and corresponding length after this rotation. For each of our segment, we first gather the coordinates at the start and the end of the segment. We can use this information to then calculate length of the segment using the Pythagorean Theorem. We can also use some trigonometry to get the angle that our segment must be rotated in order to join the two coordinates together.

```
// calculate segment length and rotational component
for (let i = 1; i <= n - 1; i++){
  // get points on either side of the segment
  let first_point  = [x_values[i - 1], y_values[i - 1]],
      second_point = [x_values[i]    , y_values[i]    ];

  // calculate x and y difference between points
  let x_diff = second_point[0] - first_point[0],
      y_diff = second_point[1] - first_point[1];

  // calculate segment length
  length_values[i - 1] = Math.sqrt(Math.pow(x_diff, 2) + Math.pow(y_diff, 2));

  // calculate rotational component
  if (y_diff === 0){
    theta_values[i - 1] = 0;
  } else {
    theta_values[i - 1] = Math.atan(y_diff / x_diff);
  }
}
```

And now that we have calculated all the information we need, we can add each segment to the simulation environment with these given parameters. What we also need to do is ensure that the curve is static and that none of the segments which make up the curve will collide with one another. For this, we can use collision filters. Bodies which have the same collision filter will be able to overlap one another. In our case, we define a number as the variable `group` and then pass this as the collision group to get the intended behaviour.

```
var group = Body.nextGroup(true);

// construct body
for (let i = 0; i < n - 1; i++){
  current_segment = Bodies.rectangle(
    (x_values[i] + x_values[i + 1]) / 2,
    (y_values[i] + y_values[i + 1]) / 2,
    length_values[i] + 10, 50, {
      collisionFilter: { group: group },
      isStatic: true,
      angle: theta_values[i]
    }
  );
  Composite.add(engine.world, current_segment);
}
```

We now have a way of adding a curve to the simulation environment from within R Shiny using a vector of heights. We next need a way to add a ball. This is what the following code does. It will place a ball at the top left of the canvas and allow it to fall under gravity. We also make sure that it has its own collision filter group so it will not overlap with anything.

```
// add a ball
Shiny.addCustomMessageHandler("add_ball", function(arg){
var group = Body.nextGroup(true);

ball = Bodies.circle(75, -100, 25, {
  collisionFilter: { group: group }
});

Composite.add(engine.world, [ball])
});
```

Before we proceed, I want to outline a custom message handler we defined named `curve_solid`. This function works in a similar way to our `add_curve` function however it adds the curve as a solid, single component using the polydecomp package for javascript rather than individual segments, similar to the terrain in a 2D video game. We didn't end up using this in the final version but it does work and I will mention later how you can use this instead of `curve_solid`

```
  Shiny.addCustomMessageHandler("curve_solid", function(points){
    // variables to save all calculated values
    let n = points.length;

    coordinates = [];

    // calculate x and y coordinates of all the points
    for (let i = 1; i <= n; i++){
      // calculate x / y
      let x = 50 + ((i - 1) * (700 / (n - 1))),
          y = 525 - points[i - 1];

      coordinates[i - 1] = {x: x, y: y};
    }
    coordinates[n] = {x: coordinates[n - 1]["x"] + 300, y: 800};
    coordinates[n + 1] = {x: coordinates[0]["x"] - 300, y: 800};

    console.log(coordinates)

    terrain = Bodies.fromVertices(400, 400, coordinates, {
      isStatic: true
    });

    Composite.add(engine.world, terrain);
  })
```

The final message handler we need to look at is called `clear_screen`. As the name may suggest, this will completely clear the screen and then add a body on the right of the screen to detect collision with the ball in the next execution of the simulation.

```
// clear screen
Shiny.addCustomMessageHandler("clear_screen", function(arg){
Composite.clear(engine.world);


right_segment = Bodies.rectangle(755, 0, 1, 1600, {
    collisionFilter: { group: group },
    isStatic: true,
    label: "right_boundary"
  }
);

Composite.add(engine.world, right_segment);
})
```

Last but not least, we need to setup an event listener that will check to see if the ball has collided with the right segment. We do this by looking at all the bodies on the screen that are currently in contact. If the ball collides with the right segment, this will then execute a function named `download()`. Lets look at this function and explain why it is necessary.

```
// an example of using collisionStart event on an engine
Events.on(engine, 'collisionStart', (event) => {
event.pairs.forEach((collision) => {
  if (collision.bodyA.label === "right_boundary"
  || collision.bodyB.label === "right_boundary"){
    console.log("Run")
    download("simulation_end.txt", "complete;")
  }
});
});
```

Sending messages from JavaScript to R Shiny is usually a very easy process. You can use `Shiny.setInputValue` to trigger an event in R Shiny and send information along with it. The problem in our case is that when we run the simulation, we pause the execution of the R Script and so when we attempt to send a message back, R Shiny isn't able to trigger the event call. Our solution to this is somewhat odd but it does the job. When the ball collides with the right segment, it triggers an event on the JavaScript side. We then add an invisible link element to our HTML page. The attributes of this element are changed to ensure that once clicked, it will download a text file containing the duration of the simulation into our downloads folder. On the R Side, we look out for this text file and once it appears, we know that the simulation has finished and we can move on to the next genotype.

```
function download(filename, text) {
var pom = document.createElement('a');
pom.setAttribute('href', 'data:text/plain;charset=utf-8,' + encodeURIComponent(text));
pom.setAttribute('download', filename);

if (document.createEvent) {
    var event = document.createEvent('MouseEvents');
    event.initEvent('click', true, true);
    pom.dispatchEvent(event);
}
else {
    pom.click();
}
}
```

Lets quickly summarise what we now have. We can now add a curve to the simulation environment, trigger an event when the ball traverses the curve, communicate this duration back to R Shiny and clear the screen ready for the next simulation.

We can now move back to R. Lets take a look at the server code for our R Shiny app. This code is designed to run an iteration of the genetic algorithm every 0.1 seconds. It uses a concept known as reactive timers which alters the value of `reactive_iteration_timer` in order to trigger the code inside the `observe` section regularly. You'll also notice that we add a parameter to our genetic algorithm which will be the `session` variable passed to the server function. This is so we can access it later when we run the simulation.

```
server <- function(input, output, session) {
    reactive_iteration_timer <- reactiveTimer(100)

    observe({
        reactive_iteration_timer();
        brach$par <- append(brach$par, list(current_session = session));

        brach$next_iteration();
    })
}
```

Before we describe how we can actually run the simulation, I want to mention a couple of functions inside of the R script. Each of these functions allows us to communicate with the message handlers in javascript. They act as proxy functions to make it easier to execute these functions on the R side.

```
add_curve <- function(session, heights){
    session$sendCustomMessage("add_curve", shiny:::toJSON(heights))
}

add_ball <- function(session){
    session$sendCustomMessage("add_ball", shiny:::toJSON(c(1)))
}

curve_solid <- function(session, heights){
    session$sendCustomMessage("curve_solid", shiny:::toJSON(heights))
}

clear_screen <- function(session){
    session$sendCustomMessage("clear_screen", shiny:::toJSON(c(1)))
}

change_speed <- function(session, speed){
    session$sendCustomMessage("set_speed", shiny:::toJSON(c(speed)))
}
```

We now have everything setup to run our simulation. Lets take a look at the function we passed as a dependancy, `utility_fitness_simulation`. Firstly, we setup our genotype with the start and end heights and then run the simulation using the `run_simulation` function. the last parameter is the the max amount of time the simulation will run in seconds before it stops.

```
# add the first and last height to the genotype
complete_genotype <- c(par$A_y, genotype, 0);

# execute the simulation
run_simulation(par$current_session, complete_genotype, 5);
```

Lets look at the `run_simulation` function. We start by cleaning up and removing any files from the previous iteration. Then we save the current time as a unix timestamp to a raw text file named "simulation_results". This is so we can determine the duration at a later stage.

```
# remove existing indicator files
remove_file(name = "simulation_results")
start_time <- as.numeric(Sys.time());
write_file(name = "simulation_start_time", text = as.character(start_time))
```

Next we add the specified curve to our simulation environment and then the ball. You can see that we need to pass the session parameter, hence why we saved this in the genetic algorithm.

```
# begin the simulation
add_curve(session, heights);
add_ball(session);
```

Next we have to wait until the simulation ends. There are two reasons that the simulation may end. Firstly, the duration time may exceed the timeout length specified. Secondly, the ball may reach the end of the curve. In the first case, we calculate the difference between the start time and the current time and if this is greater than the timeout length, we create a file called `simulation_results` containing the text `complete;timeout`. If the while loop locates this file, we break the loop because the simulation has ended due to a timeout.

```
# wait until the simulation ends
while (TRUE){
    current_duration <- Sys.time() - start_time;
    if (file.exists("simulation_results")){
        break
    }

    if (current_duration > timeout){
        # simulation ends due to a timeout
        write_file("simulation_results", "complete;timeout")
        break
    }
```

The other event that causes our simulation to end is that the ball has reached the other side of the curve. You'll remember that from the JavaScript side we have an event which will trigger when the ball reaches the other side of the curve and then downloads a file to to the downloads folder. The following code looks for this file and when it finds it, the duration is calculated and the file is removed from the downloads folder.

We had a few issues where windows would automatically name the file located in the downloads file with an unwanted suffix. Because of this, we look for the most recent file with a particular string at the start rather than an exact match.

```
# check if the timer has ended (long and stupid)
downloads_dir <- "C:/Users/david.blair/Downloads/";

if (length(Sys.glob(paste0(downloads_dir, "simulation_end*.txt"))) > 0){
  file_name <- get_sim_end_recent();
  start_time <- as.numeric(get_file_contents(name = "simulation_start_time"))
  duration_time <- as.numeric(Sys.time()) - as.numeric(start_time);
  print(duration_time)
  write_file(name = "simulation_results", text = paste0("complete;", duration_time))
  remove_file(file_name)
  break
}
```

Before we continue, I've added a delay of 1 second to let everything load and allow windows to catch up with respect to the deletion of files.

```
Sys.sleep(1)
```

After we have run the simulation, we need to wait until the simulation results have been produced by using a while loop which will break when it finds the file named "simulation_results".

```
# wait for the results
does_file_exist <- FALSE;
while (!does_file_exist){
  print(does_file_exist)
  does_file_exist <- file.exists("simulation_results");
}
```

Once the simulation has fully ended, we get the total duration from simulation_results. If the duration is "timeout" then we set the total time to a very large number. Otherwise, we set it to the duration that is given.

```
simulation_results <-  get_file_contents("simulation_results")
  duration <- stringr::str_split(simulation_results, stringr::coll(";"))

  if (duration[[1]][2] == "timeout"){
      total_time <- 100;
  } else {
      total_time <- as.numeric(duration[[1]][2]);
  }
```

lastly, we clean up the environment and then clear the screen. The total time is then returned to the genetic algorithm.

```
remove_file(name = "simulation_results")
remove_file(name = "simulation_start_time")
clear_screen(par$current_session)

return(total_time)
```

We will discuss in the conclusion the numerous ways in which this implementation can be improved and several ideas of how to extend it further so its more useful.

5 Conclusion and Recommendations

In the project plan, we outlined a number of goals that we wanted to achieve:

1. Design a package in R that allows the user to construct a genetic algorithm with minimal effort.
2. Create a UI to make interacting with the genetic algorithm easier.
3. Use a genetic algorithm to solve the Brachistochrone problem.
4. Use a simulation as a fitness function to solve the Brachistochrone problem.

During this project, we constructed an R package to satisfy item 1 and 2 from the list above. The original aim was that a genetic algorithm class object would hold a number of operator class objects designed to amend another object representing the population. It became clear very early on that representing these different objects as classes would reduce the simplicity offered by the package. Because of this, we decided to maintain an object orientated approach for the genetic algorithm itself, embedding a more procedural approach with each operator being a standardised R function and our population being a multidimensional array. The implications of this deviation meant that the population dimensions have to be maintained and carefully considered when using the majority of the operators. While the structure we have chosen has its limitations, its generality as a class object containing methods which allow it to change over time mean that it can be used to model any type of stochastic process. In the future, I would love to recreate this package in a low-level language, most likely C++. I still believe that an object orientated approach is better for this type of package but its overall success (in industry or academia) will be determined by the speed at which it can perform these simulations, leading to the natural conclusion that a low-level object orientated language would of been better.

When the UI was originally being designed, we intended for it to be the location from which a user could view all the information about the genetic algorithm and change the operators during the execution of the simulation. We wanted to have a location in the UI where users could type R code that would then get processed by the package into an operator. The problem with this is that we were designing things in this way to appeal to a wider audience. If we require that the user has prerequisite knowledge

of R to use the UI in the first place then having a UI would only make things more complicated for the actual people who would be using the package. This is why we decided to use a UI to view what was going on and make changes to parameters but the actual construction of the genetic algorithm itself would be done completely in R. This meant that the package would appeal to the type of user it was designed for. I don't think that the UI is actually that helpful for seasoned programmers and so we won't be expanding on this in the future.

The original goal of this project was to construct a genetic algorithm to solve the brachistochrone problem. This has certainly been achieved and can be accomplished using the built in functions from the ga-package. As well as this, a novel suggestion by Dr Bandara to attempt to minimise distances in non-Euclidean space transformed by a 2x2 matrix provided some interesting ideas for further research. We have restricted ourselves to working in $\mathbb{R}^2$ but there isn't any reason that this can't be generalised for higher dimensions. If I had more time, I would of loved to explore this further. I would also like to look at different types of functions which transform the space, especially those which would provide many local minima and maxima to see how they would affect the algorithms choice of path.

In my original plan, I was hoping to include a fully integrated physics engine into the gapackage that could be used as a fitness function very easily. It became clear very early on that this was not going to be doable in the amount of time we had. Firstly, my choice of physics engine was very poor. I had run a number of performance tests on this engine prior to implementation but I failed to investigate how matter.js handles object clipping. Secondly, since this javascript engine was never designed to work with R, the design of both were incredibly incompatible, with one being a tied to game loop and the other being reactive. This meant communication between the two was incredibly difficult. However, even though our original goals were scaled back significantly, we have produced a proof of concept to at least demonstrate the intention. There is a lot of ways in which this could be improved. Firstly, we never actually used the physics engine to solve any real world problems. This is because it is dreadfully slow, taking a maximum of 1 hour and 20 minutes per generation. For it to be useful, we will need to run many physics simulations at any given time. As well as this, we will need to use a better physics engine and choose a more low-level programming language. I would really like to explore evolutionary algorithms at a later stage. A video by Computer Researcher Karl Sims [16] demonstrates how he was able to use simulated evolution to teach "fish" how to swim. What I would like to do is explore neural networks and see if we can use them to create genotypes where their structure and size is actually

changeable during the simulation. As well as this, I would like to introduce multi-objective fitness function, other species and predators into Karl's simulation to see how things evolve over time given those constraints.

References

[1] 2022. URL: https://cran.r-project.org/bin/windows/Rtools/rtools40.html.

[2] *Acta eruditorum*. Acta eruditorum. Christoph Günther, 1696. URL: https://books.google.co.uk/books?id=4q1RAAAAcAAJ.

[3] H. B. Creighton and B. McClintock. "A Correlation of Cytological and Genetical Crossing-Over in Zea Mays". eng. In: *Proceedings of the National Academy of Sciences of the United States of America* 17.8 (Aug. 1931). PMC1076098[pmcid], pp. 492–497. ISSN: 0027-8424. DOI: 10.1073/pnas.17.8.492. URL: https://doi.org/10.1073/pnas.17.8.492.

[4] Yong Deng, Yang Liu, and Deyun Zhou. "An Improved Genetic Algorithm with Initial Population Strategy for Symmetric TSP". In: *Mathematical Problems in Engineering* 2015 (Oct. 2015), p. 212794. ISSN: 1024-123X. DOI: 10.1155/2015/212794. URL: https://doi.org/10.1155/2015/212794.

[5] Peter J. Denning and Ted G. Lewis. "Exponential Laws of Computing Growth". In: *Commun. ACM* 60.1 (Dec. 2016), pp. 54–65. ISSN: 0001-0782. DOI: 10.1145/2976758. URL: https://doi.org/10.1145/2976758.

[6] *Download the RStudio Idea*. URL: https://www.rstudio.com/products/rstudio/download/#download.

[7] D.B. Fogel. "Nils Barricelli - artificial life, coevolution, self-adaptation". In: *IEEE Computational Intelligence Magazine* 1.1 (2006), pp. 41–45. DOI: 10.1109/MCI.2006.1597062.

[8] G. Galilei and H. Crew. *Dialogues Concerning Two New Sciences*. Martino Fine Books, 2015. ISBN: 9781614277941. URL: https://books.google.co.uk/books?id=ffLrrQEACAAJ.

[9] John H. Holland. "Adaptation in natural and artificial systems". In: 1975.

[10] F. A. Janssens, Romain Koszul, and Denise Zickler. "The chiasmatype theory. A new interpretation of the maturation divisions. 1909". eng. In: *Genetics* 191.2 (June 2012). 191/2/319[PII], pp. 319–346. ISSN: 1943-2631. DOI: 10.1534/genetics.112.139725. URL: https://doi.org/10.1534/genetics.112.139725.

[11] *Matter.js*. URL: https://brm.io/matter-js/.

[12] Melanie Mitchell. "Genetic algorithms: An overview." In: *Complex.* Vol. 1. 1. Citeseer. 1995, pp. 31–39.

[13] H. J. Muller and T. S. Painter. "The Cytological Expression of Changes in Gene Alignment Produced by X-Rays in Drosophila". In: *The American Naturalist* 63.686 (1929), pp. 193–200. DOI: 10.1086/280253. eprint: https://doi.org/10.1086/280253. URL: https://doi.org/10.1086/280253.

[14] *R-4.1.3 for windows (32/64 bit)*. URL: https://cran.r-project.org/bin/windows/base/.

[15] Shahryar Rahnamayan, Hamid R. Tizhoosh, and Magdy M.A. Salama. "A novel population initialization method for accelerating evolutionary algorithms". In: *Computers Mathematics with Applications* 53.10 (2007), pp. 1605–1614. ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2006.07.013. URL: https://www.sciencedirect.com/science/article/pii/S0898122107001344.

[16] Karl Sims. 2022. URL: https://www.youtube.com/watch?v=bBt0imn77Zg&t=29s.

[17] A. M. TURING. "I.—COMPUTING MACHINERY AND INTELLIGENCE". In: *Mind* LIX.236 (Oct. 1950), pp. 433–460. ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. eprint: https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf. URL: https://doi.org/10.1093/mind/LIX.236.433.

[18] Patrick Winston. *MIT Lecture on Genetic Algorithms*. MIT. 2014. URL: https://www.youtube.com/watch?v=kHyNqSnzP8Y.

[19] Xin-She Yang. "Chapter 1 - Introduction to Algorithms". In: *Nature-Inspired Optimization Algorithms*. Ed. by Xin-She Yang. Oxford: Elsevier, 2014, pp. 1–21. ISBN: 978-0-12-416743-8. DOI: https://doi.org/10.1016/B978-0-12-416743-8.00001-4. URL: https://www.sciencedirect.com/science/article/pii/B9780124167438000014.