Politecnico di Torino

FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria informatica

Tesi di laurea magistrale

A genetic algorithm for optimizing velocity in the Bridgestone World Solar Challenge

Candidato:
Pietro Chiavassa

Relatore:
Filippo Gandino

Correlatori:
Dirk Roose
Dries Ketelslegers

Anno Accademico 2019–2020

Contents

Abstract

In the Bridgestone World Solar Challenge (WSC) student teams from various universities use solar technology and advanced engineering to create sustainable innovative solar powered electric vehicles able to travel 3,020 km from Darwin to Adelaide. During the race, the strategy unit of the teams must be able to decide at which speed the car should drive at any given moment. The objective is to finish the race in the least amount of time while avoiding the risk of running out of battery.

The Belgian Team (Agoria Solar Team) uses a Genetic Algorithm to solve this problem. A population of possible solutions is evolved, in which individuals are represented as vectors of velocities that the car should drive at in each segment of the race. A fitness function evaluates each solution by taking into account weather conditions, spatial position, topography information, car speed, car specifications and race specific constraints. The output of the function is the time to finish the race, penalized in the case of discharged batteries. Individuals with better fitness are selected for reproduction, creating new children for the next generation. Mutation is also applied on the newly generated children.

The objective of this work is to improve the performance of the algorithm in terms of solution quality and computation time. The first goal is achieved by improving the genetic operators and performing more extensive tuning of the algorithm's parameters. The second objective is reached by porting the code, originally developed in Matlab, to a compiled language (C++) and introducing support for parallelization. The software was developed focusing on portability, extensibility and ease of use. All the fundamental functionalities were made accessible from the Python environment by wrapping the library into a Python module.

The new implementation of the algorithm improved the computation speed of about 60 times when run with comparable operators and parameter settings. Thread parallelization further increased the performance by achieving almost perfect scaling in optimal conditions. The extensive testing unveiled important insights on how different parameters and operators affect the evolutionary process and quality of solutions. The proposed modifications also led to a refinement of the model, which is now able to produce more realistic speed profiles.

Chapter 1

Introduction

This work is focused on developing a software library that is able to support the strategic decisions of the Agoria Solar Team during the World Solar Challenge competition and it is intended as a continuation of [10].

The document is structured in eight chapters, including this introduction. The second chapter is meant to provide some background information on the World Solar Challenge and to defines the problem that has to be solved. Chapter 3 gives some background information on genetic algorithms, which can be useful to better understand this work. Chapter four focuses on problem modelling, while chapter five discusses the genetic operators that are used. Chapter six is about the specifics of the C++ software implementation and the introduction of parallelization support. Chapter seven evaluates the performance of the algorithm and explains the tuning and testing procedure that were followed. The eighth and final chapter is a conclusion that summarizes the results achieved by this work and proposes some new ideas for further development.

Chapter 2

Problem description

The objective of this chapter is to present in great detail the problem that this work is focused on. A general introduction of the World Solar Challenge [2] is given in the beginning, followed by a more in depth analysis of the relevant aspects concerning the problem to be solved. All the information regarding the race are referred to the 2019 edition.

2.1 The World Solar Challenge

The World Solar Challenge is a competition open to student and company teams from all the world to build and run a solar-powered electric vehicle across Australia. The idea for the race originated from the 1982 quest of Hans Tholstrup and Larry Perkin, who drove a home-built solar car from East to West Australia. From the first race, held in 1987, the 2019 edition reached a participation of 53 teams from 24 countries. The objectives of the event are "to showcase the development of advanced automotive technology and promote alternatives to conventional vehicle engines" [1].

The competition is divided in three different classes of vehicles: The Challenger Class, The Cruiser Class and The Adventure Class. The class considered in this work is the first one, which is focused on fast single-seat vehicles.

The race starts in Darwin, in the North, and ends in Adelaide, located in the South. It is approximately 3020 km long and has to be completed is less than seven days. The image of the route, traversing the desertic region of the Australian outback, is showed in Figure 2.1.

2.1.1 The car

The car developed by the Agoria Solar Team (BluePoint) is shown in Figure 2.2. The battery capacity is 5300 Wh and is recharged via the solar panels present on the top. The solar panels can be lifted and oriented toward the sun while the car is still, to increase the energy production. The car shape is designed to improve aerodynamics and was tested in a wind tunnel.

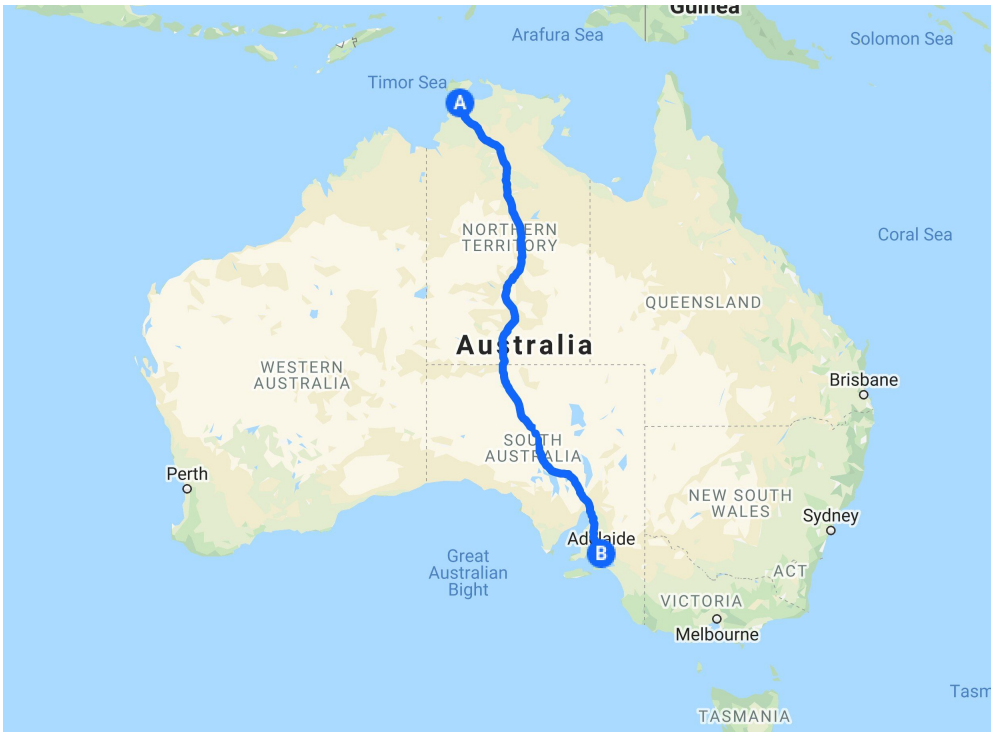

Figure 2.1: Route of The World Solar Challenge 2019 [7]

The most important information that is used in this work are the battery size, the energy production given the panel orientation and solar radiation, the energy consumption of the engine at a given speed, street slope, wind speed and air density. The variation of efficiency of the solar panels due to ambient temperature is also taken into account.

All the data described above was provided by the solar team by the means of multi dimensional tensors which are looked up by the algorithm during its execution. An on-line computation of these value would have been too much time consuming. Some more detailed information about the car can be found in Table 2.1.

2.1.2 Rules

The start time of the race is 8:30 AM for the first day and 8:00 AM for the following ones. Each day ends at 5:00 PM and the car is allowed to run for a maximum of 10 minutes past this time. The start time on the following day is delayed according to the additional time that the car runs for after 5:00 PM.

It is still possible to recharge the car batteries until sunset and from sunrise before the start of the race the new day, by using its solar panels. During the night the energy storage packs must be removed from the car and placed in a secured box under the supervision of an observer. On the first day, the race is started with fully charged batteries.

Along the route there are several control stops where the car has to stop for 30 minutes

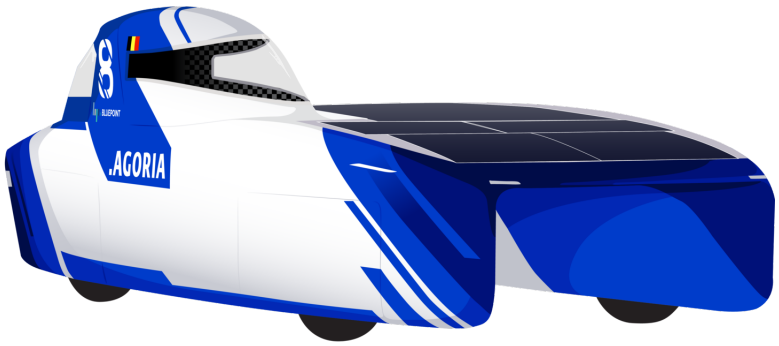

Figure 2.2: The "Bluepoint" solar car [16]

before it is allowed to continue the race. During this time it is still possible to use the solar panels to recharge the batteries.

During the race the car is also subjected to the same speed limits and australian general traffic regulations which apply to normal vehicles.

2.2 Problem

During the race, the strategy unit of the Agoria Solar Team has to take decisions about the speed the car should drive at, in each remaining section of the race. The value to minimize is the time required to complete the race, with additional constraints: the energy stored in the batteries should not go below a threshold before reaching the finish line, and not too much unused energy must remain in the batteries when reaching the destination.

The difficulty of the task comes from the multitude of factors that concur in defining the solution of the problem. The ones that have a direct impact on the finish time are the incoming and outgoing energy of the car. However, these have some additional dependencies on other variables such as the car speed, wind speed, solar radiation, ambient temperature, air density and slope of the road.

To make the problem more difficult, most of these variables have strong reciprocal dependencies. As an example, by changing the speed, the position of the car at a given time is also changed and so is the weather, the solar radiation and ultimately the incoming energy. This would increase or decrease the optimal speed of the car. An additional complexity is also given by the uncertainty of the weather predictions, which reliability decreases with time.

All of these variables define a problem (a search problem) which complexity is out of reach for a "by hand" solution. This is why there is a necessity of creating a software tool that support the decision making of the team.

As a last note, it is important to highlight that the model used in this tool is an extreme simplification of the real world problem. As a consequence, the solutions that it provides will be treated as a reference for the team to take its decisions, which will be based on expertise and experience, taking into account additional factors which the model is unaware

of. Example are the presence of traffic on the road, eventual problems to the car, margin of error on energy expense, etc..

FEATURE	DESCRIPTION
Width	160 cm
Length	330 cm
Height	100 cm
Mass	135 kg
Top speed	130 km/h
Battery autonomy	600 km without sun
Numbers of wheels	4
Solar cells	2.64 m^2 of Azurspace Multijunction cells
Motor	Mitsuba in-wheel direct drive motor
Battery	20 kg li-ion cells with a capacity of 5300 Wh
Body	TEI carbon fiber reinforced with Dupont Kevlar fibers
Suspension	Double wishbone
Steering	Four-wheel steering with crabbing system
Braking	Regenerative & mechanical braking

Table 2.1: Car specifications [15]

Chapter 3

Genetic algorithms

This chapter gives some background information on genetic algorithms and defines some of the terms that will be used in the next chapters. The discussion is based on the information present in [14].

3.1 Introduction

Genetic algorithms are a part of the area of computer science called evolutionary computing. Techniques developed in this area get their inspiration from the biological process of evolution. Before this definition was introduced lots of separate techniques were developed under different names. To list some, as discussed in [14], it is possible to mention evolutionary programming, genetic algorithms, evolution strategies, and genetic programming.

The inspiration from biology comes from two main aspects of the evolutionary theory: the Darwinian Evolution and genetics. From the former, the main concept that is taken into account is natural selection. This states that, since the resources available in the environment are limited, only better individuals are able to survive and reproduce. In this way, traits of these individuals are transmitted to the next generations. It is important to specify that the term "better" is not absolute but is relative to the environment the individual lives in and interacts with. From this consideration, the term "better" is substituted by "fittest", to highlight this relation.

From genetics, evolutionary computing takes the concepts of phenotype, genotype and the processes of mutation and crossover that concur in defining the genetic heritage of new individuals. Genotype refers to the information that is encoded in the DNA of an organism, while phenotype is the expression of this information into its visible traits.

Evolutionary computing is successfully used to solve optimization, modelling and simulation problems. Only optimization problems will be considered in the following.

3.2 Structure

Representation The first step in the design of a genetic algorithm is to find a way to represent the problem at hand. This is done by defining a way to encode solutions of the original problem into a representation that can be handled by the algorithm. The structure of a solution in the original problem space is identified as the phenotype, while the encoding as the genotype. Typical encodings are binary, integer, real-valued, permutations or tree representations [14]. For these reasons, in the context of genetic algorithms, solutions are also called individuals.

Fitness function The role of the fitness function is to evaluate the quality of a solution in the context of the problem. The evaluation process can be seen as a translation of the genotype into the phenotype and the evaluation of the latter. From a biological point of view, this corresponds to the fact that the genetic material does not directly defines the survival of an individual: it is only its phenotypic expression that plays a role in the natural selection process. In the case of optimization problems, the fitness function correspond the function that needs to be maximised or minimized.

Working principle A population of individuals is evolved for a number of generations. In each generation the best individuals are selected to generate the new population. The final population should contain the optimal of solution to the problem or, when this not reached, high quality alternatives.

Initialization The initial population is usually randomly generated in search space of the problem for each new run. A fixed configuration of the initial population may harm the quality of the solutions by making it hard to reach the optimum.

Parent selection In this phase the best individuals are selected to transmit their genetic material to the next generations. The selected solutions are also called parents. The selection process is based on the fitness evaluation of the solutions performed with the fitness function. Lots of techniques have been developed to perform this task but, depending of the problem characteristics, some of them can harm be harmful to the evolutionary process (e.g premature convergence, as defined in the next section).

Crossover Parents are taken in couples and their genetic information is used to create one or more children (usually not more than two). Each couple of parents is selected for crossover according to a probability called crossover rate. For this reason, some of the parents may remain unaltered in this phase. Depending on the representation, lots of different techniques can be used for the recombination. The ones relevant to this work will be presented in the following chapters.

Mutation Mutation is applied to the product of crossover, i.e. to both the new children and unaltered parents. This is usually a small and random variation of the genotype of the individuals. Also in this case, a mutation rate defines how many solution will be subjected to this process. The mutation techniques used in this work will be explained later.

Reinsertion The newly created individuals must be inserted in the population. Since its size is usually kept constant among generations, these have to replace old individuals. Multiple techniques can be used for this process, that is also referred to as "survival selection". Sometimes the n best solutions in the population are kept the next generation. These solutions are called elite individual and n is referred to as elitism. n is usually chosen greater than zero to avoid loosing the best solution found so far, while higher values can be used to influence the evolutionary process. Care must be taken when increasing n in order to avoid premature convergence.

3.3 Considerations

Some important considerations have to be made on how genetic algorithms operate.

Forces The evolving population can be seen as a group of points that, at each generation, moves through the search space. Two main forces, called exploration and exploitation [14], drive the process. Exploration pushes the population towards unexplored areas of the search space, while exploitation focuses the search in the proximity of already known good solutions. Crossover and mutation are responsible for the former while parent selection and reinsertion for the latter. If these two behaviors are not well balanced problems may arise. When exploitation is excessive, the population may immediately converge to a local optimum instead of looking for a global one (premature convergence). When the opposite happens, the search process becomes inefficient and leads to long computational times.

Performance According to [14], genetic algorithm are known be effective when their performance is averaged on a large range of problems. This aspect is in contrast with problem-tailored techniques that, while outperforming evolutionary algorithms in their target task, usually fall short when applied in different contexts. However, their versatility is limited by the free lunch theorem, which states that no algorithm can outperform random search when applied to the set of all possible problems.

Theory Even if evolutionary algorithms are clearly a heuristic approach to optimization, some attempt have been made to describe them in a theoretical form. An example of this is the schema theorem, as explained in [14].

Chapter 4

Modelling the problem

In this chapter it is discussed how the problem is modelled, that is, how the objective function and constraints are implemented and enforced. This will define the landscape in which the genetic algorithm performs the search.

4.1 Data structures

The fitness function, in order to evaluate a solution, uses some pre-computed data. This is done to speed up the algorithm, offloading whichever calculation does not have to be done in real time to an off-line setting. The data is stored in five different tensors, that will be from now on called matrices, to keep the notation coherent with the previous work [10]:

- Power out matrix (41 x 401 x 81 x 26)
- Stage matrix (605 x 10)
- Solar matrix (720 x 5 x 3021 x 6)
- Density matrix (720 x 3021 x 6)
- Wind matrix (720 x 2 x 3021 x 6)

4.1.1 Power out matrix

This tensor contains the power consumption of the car, expressed in W, depending on the car speed, slope of the road, wind speed and air density. The relationship between the energy consumption and both speed and terrain slope can be clearly understood: maintaining a higher speed on a steeper hill requires the engine to consume more power. The same goes for the wind speed, that increases the air resistance opposed to the car motion. For what concerns the air density (ρ), this is determines the dynamic pressure (P_d) acting on the car while it moves through air:

$$P_d = \frac{1}{2}\rho(v_{car} + v_{wind})^2 \tag{4.1}$$

The term v_{wind} refers to the car's head wind, that will be introduced later. The dynamic pressure is directly proportional to aerodynamic forces such as drag and lift [12], thus its importance.

The dimensions represent the dependencies in the following order: car speed, slope, wind speed and air density. Given the velocity of the car, the speed index is computed according to 4.2. Therefore the power consumption presents a speed granularity of 1 km/h and the range of speed for which an evaluation is possible is [70, 110] km/h. The other values used to correctly index the power out matrix are retrieved from the other matrices, as it will be explained later.

$$Speed\_index = Speed - 70 \tag{4.2}$$

Figure 4.1 shows how the terrain slope, wind speed and air density influence the power consumption of the car at different speeds. The calculation were done by the Agoria Solar Team by taking into account the weight of the car, weight of the driver, aerodynamics and the engine specifications.

4.1.2 Stage Matrix

The stage matrix contains information about the route of the race. The whole 3020 km are divided in 604 stages that are 5 km long (apart from the last one measuring 4.4 km), each represented by one row in the matrix. It can be noticed that the first dimension has size 605, instead of 604, but this is just due to representation purposes. For each stage the following information is reported, according to increasing column number (1-10):

- Start of the stage (km)
- Latitude (Degree)
- Longitude (Degree)
- Altitude (m)
- Presence of control stop (bool)
- Speed limit (km/h)
- Length of stage (km)
- Position index
- Slope (%)
- Heading (°)

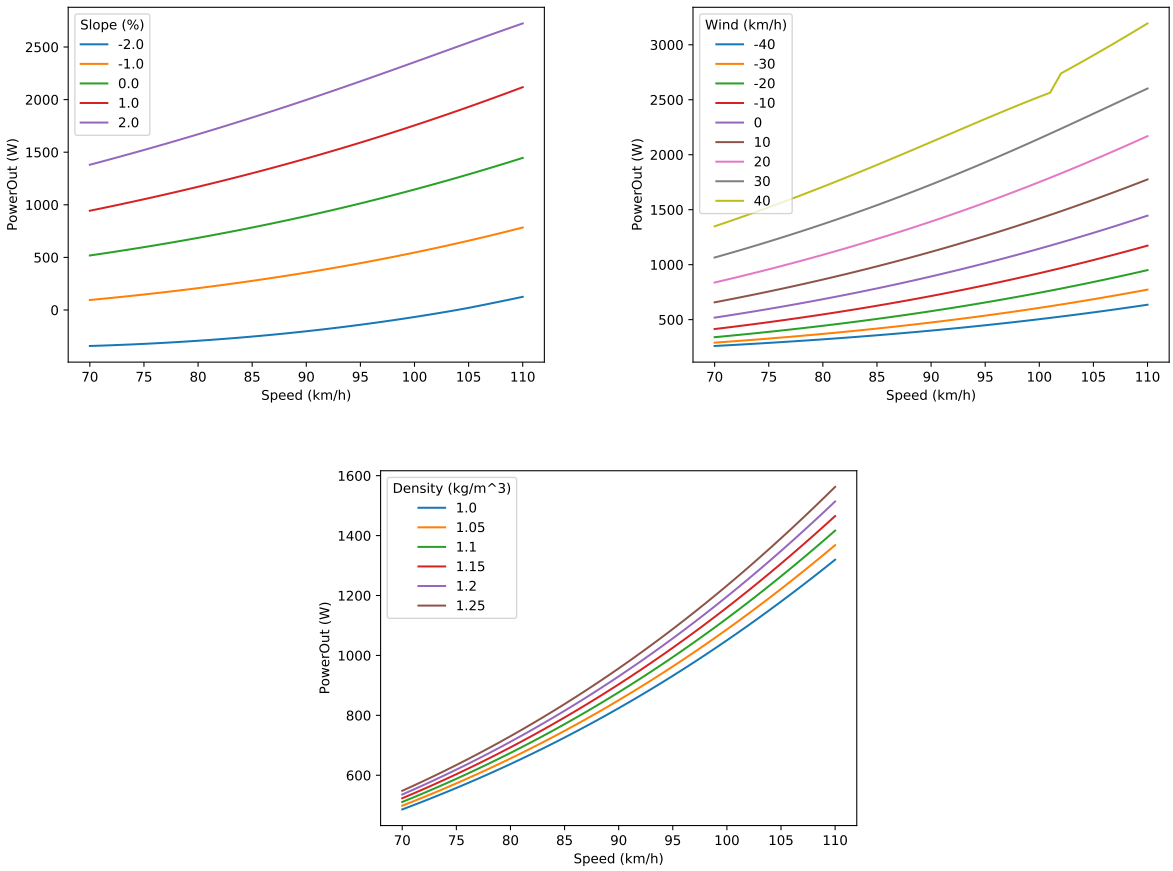

Figure 4.1: Power consumption

The terrain slope is averaged along the stage length according to 4.3. In order to use it to index the power out matrix, the value is converted according to 4.4. By looking at cardinality of the second dimension of the power out matrix it is possible to notice that values for the slope must be in the [-2, 2]% range.

$$Slope = \frac{Altitude(stage\_end) - Altitude(stage\_start)}{length(stage) \times 1000} \times 100 \tag{4.3}$$

$$Slope\_index = (Slope + 2) \times 100 \tag{4.4}$$

The heading is the azimuth angle (clockwise angle with respect to the north). The position index, instead, is used to index the entries of the weather matrices (wind, solar and density) that correspond to the stage. This allows for a non uniform length of the stages and avoids a real time computation of the indexes.

Altitude and slope variations along the route are depicted in Figure 4.2.

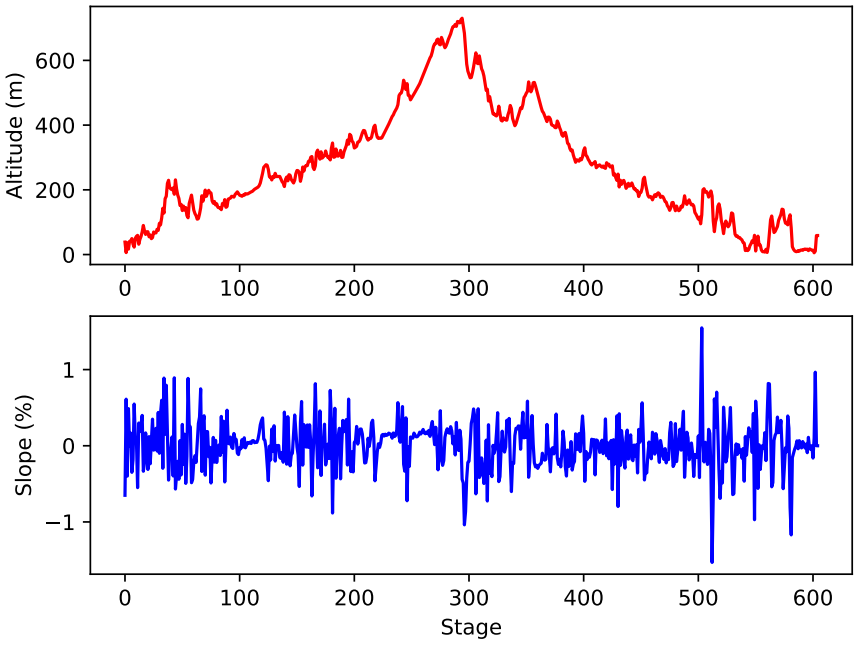

Figure 4.2: Altitude and slope

4.1.3 Solar matrix

The solar matrix contains information on the solar radiation for each day of the race. Weather predictions are initially available with a granularity of one hour, which is then increased to three hours after 72 hours, due to the uncertainty of the forecasts. However, the granularity is evenly reduced to two minutes by cubic interpolation while generating the matrix. The first dimension, as a consequence, refers to each two minute interval in a day, while the fourth dimension refers to the day number. For what concerns the space granularity, this is also reduced from 20 to 1 km, giving the size of the third dimension. This is indexed by using the position index from the stage matrix.

The second dimension refers to different measurements of the solar radiation, that are needed in different contexts. It is important to mention that all the primitive measures, which are needed to construct the tensor, are initially given in a cumulative representation. Therefore, at the beginning, an additional conversion is required. Here a listing, followed by an explanation, is provided:

- Global Horizontal Irradiance (GHI) (W/m^2)
- Direct Normal Irradiance (DNI) (W/m^2)
- Diffuse Horizontal Irradiance (DHI) (W/m^2)

- PowerIn driving (Wh)
- PowerIn loading (Wh)

The DNI is measured as a direct solar radiation on an unit of surface perpendicular to the sun. The diffused radiation is not taken into consideration in this measure. This is evaluated by the DHI, which is measured on a horizontal surface with radiation coming from all directions, except directly from the solar disk. The two measures are combined into the GHI, which expresses the total radiation coming on a horizontal surface of unit size [18] (z is the zenith angle):

$$GHI = DHI \times DNI \times cos(z) \tag{4.5}$$

"PowerIn driving" an "PowerIn loading" express the power that can actually be accumulated in the batteries while the car is driving or standing still (at a control stop or during the phases of morning and evening charge). The values differ because in the second case the solar panels can be oriented towards the sun to increase the efficiency of the charging process. These values are derived from the GDI, DNI and DHI by also considering losses and other effects in the energy conversion system of the car.

In this work different solar matrices will be used. One is artificially generated to simulate a clear sky scenario while the others gather real weather data from September 2019. An example of one containing real data is presented in Figure 4.3.

4.1.4 Density matrix

The air density matrix shares a similar structure with the solar matrix. The space and time subdivision is the same as the previous one and this is reflected by its dimensions. Also in this case the time granularity is reduced to two minutes via cubic interpolation. The same goes for space granularity which is reduced from 20 km to 1 km. The unit of measure for the air density is kg/m^3.

In order to index the power out matrix, once the value of the density (ρ) is read, it is capped to the [1, 1.25] kg/m^3 range. The index is then evaluated as follows:

$$Pressure\_index = 100 \times (\rho - 1) \tag{4.6}$$

The result lays in the range [0,25], coherently with the fourth dimension of the power out matrix.

Also in this case different matrices will be used. The artificially created one is set to the standard atmospheric air density of the altitude at which each section is located. The others record the actual data registered in September 2019. An example is shown in Figure 4.4.

4.1.5 Wind matrix

The wind speed matrix also follows the same structure as the air density. The time and space dimensions are the same, and the same procedure is followed to obtain a finer granularity. In

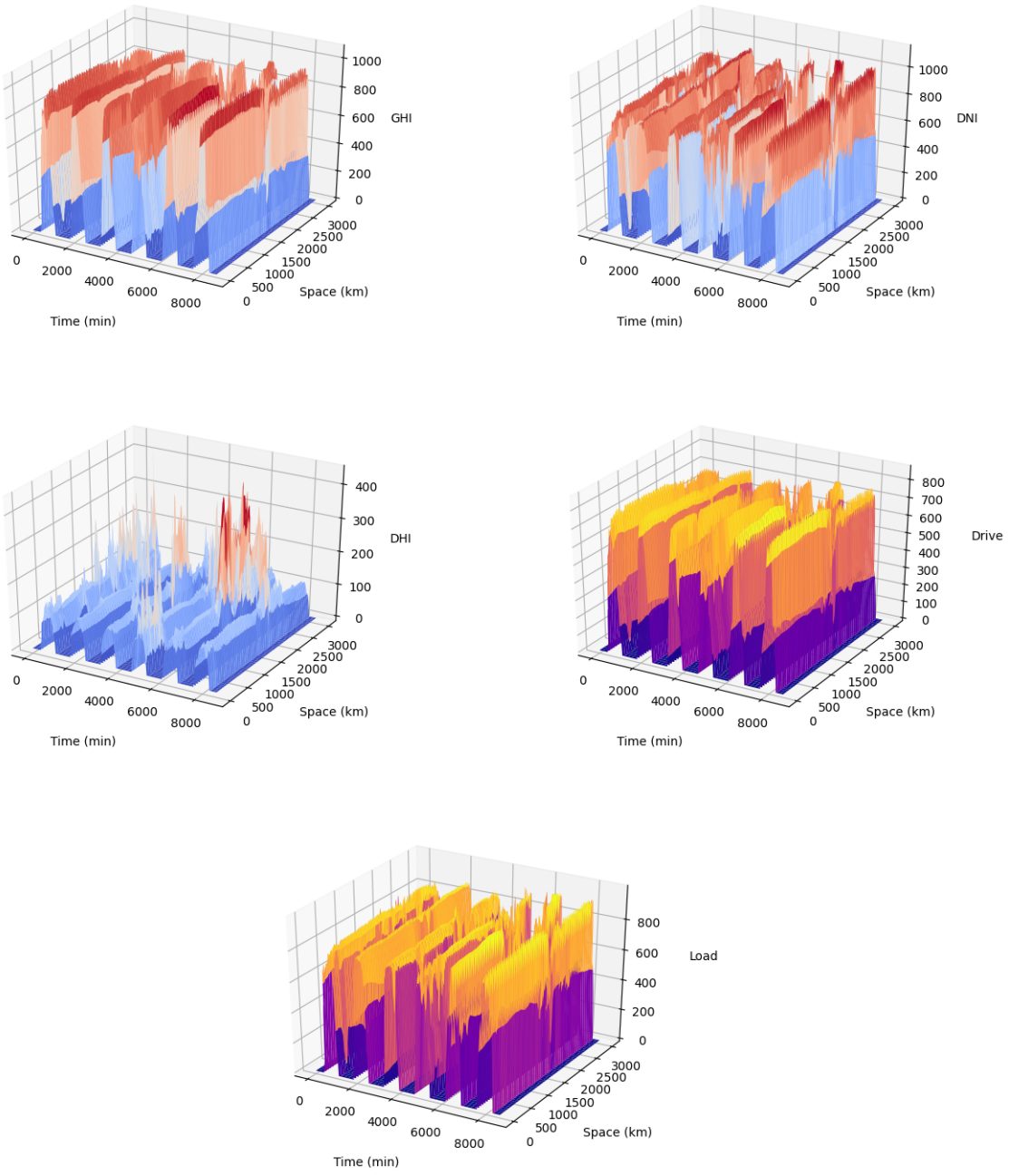

Figure 4.3: Solar matrix with real data

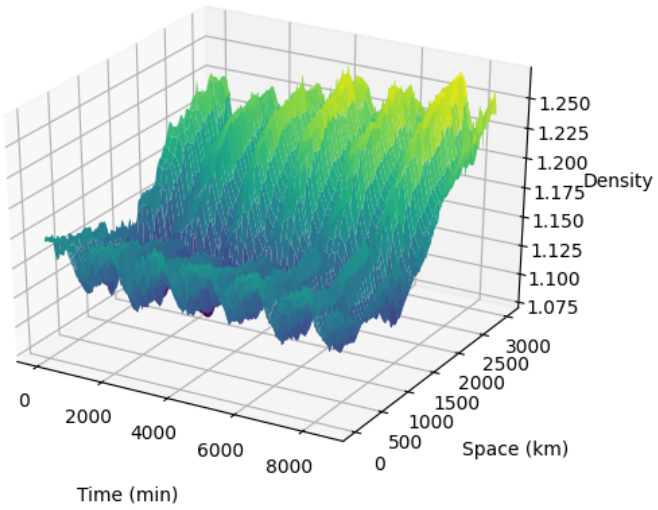

Figure 4.4: Density matrix with real data

this case two values are associated with the second dimension. These are the two components of the wind speed vector: u, v (expressed in m/s). The first is the Zonal Velocity (toward east) while the second one is the Meridional Velocity (toward south). The front and side wind hitting the car can be derived from these two components by using the following formulas (where h is the heading, that is obtained from the stage matrix):

$$front\_wind = v \times cos(h) + u \times sin(h) \tag{4.7}$$

$$side\_wind = v \times sin(h) + u \times cos(h) \tag{4.8}$$

The side wind speed is not taken into account into the evaluation of the consumed power. This is a change that will be made in the future by the solar team, when the side wind effect will be introduced in the aerodynamic model of the car. The power out matrix will be consequently modified with the addition of this parameter.

In order to index the power out matrix, the value of the $front\_wind$ is converted to km/h and capped to the [-40, 40] km/h range. The value of the index is then evaluated as follows:

$$Wind\_index = front\_wind + 40 \tag{4.9}$$

The result lays in the range [0,80], coherently with the third dimension of the power out matrix.

This work is focused on two different wind matrices. As before, the first one is artificial, where the wind is absent. The others, again, refer to real data of September 2019. An example is shown in Figure 4.5.

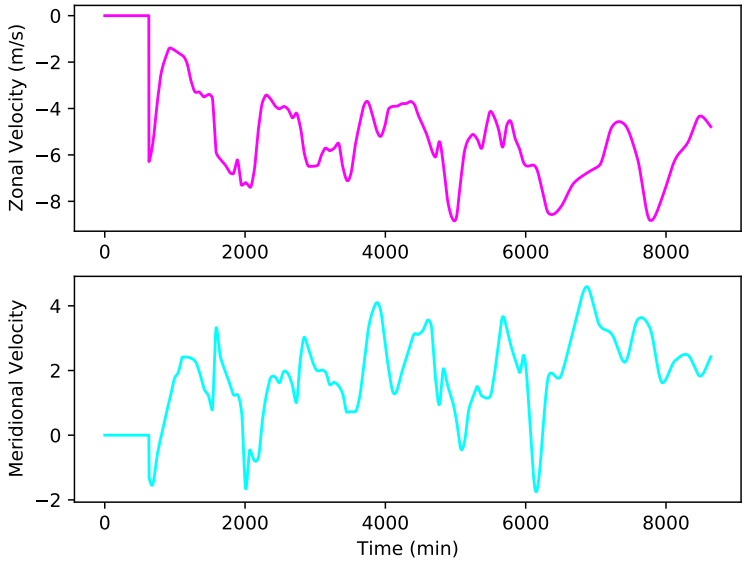

Figure 4.5: Real wind measurements at fixed position

4.2 Fitness function

The fitness function is used to evaluate each solution in the population and therefore it defines the search space in which the genetic algorithm operates.

4.2.1 Representation and constraints

The genotype of an individual consists of a vector of speeds that the car should maintain in each of the stages. In the parts of the race where speed limits are present, if the limit is lower than 75 km/h, the car is always set to drive a the max allowed speed. This decision was made because the speed is low enough so that it never impacts in a negative way the energy consumption of the car. This also has the advantage of reducing the dimension of the genotype space to be explored. When speed limits are not present, the velocity is limited to the 75 to 95 km/h range, for the same reasons. Above 95 km/h the outgoing power quickly increases due to aerodynamic forces, so it is known a priori that good solutions won't be discarded.

The speed values are real numbers, stored in double precision. However it is important to notice that such a precision in the speed values of the solution is not useful in the real world scenario. The car will be always driven with a discretization of speed around 1 km/h.

4.2.2 Fitness evaluation

The fitness function, given the vector of speeds for the whole race, computes the time required to finish the competition. The evaluation of the fitness must be performed in an incremental manner, one stage at a time. This is needed because the speed in the first sectors will determine the time at which the next sector will be reached and, consequentially, the weather condition that will be encountered.

The car starts with a full battery (5300 Wh) and in each step a charge variation is calculated as a balance between the incoming and outgoing energy. These values are retrieved by performing lookup operations over the solar and power out matrices, respectively. Each stage is analysed in the same way by taking into account two important aspects: the presence of a control stop and whether the end of the day is reached.

Standard case

In the standard case in which neither of these are present, the total time is increased by the length of the stage divided by the car speed. The position index (from the stage matrix) together with the time spent in the stage is used to index the solar matrix. Since the car is running, the second dimension index is set to PowerIn driving. When multiple time indexes (2 min zones) are traversed, the incoming power is averaged along this dimension and then multiplied by total time spent in the stage. If the battery is full the additional power is discarded.

The power out is calculated by retrieving the wind speed and air density relative to the beginning of the stage and to the time at which this is reached. From the stage matrix the index for the slope is also obtained. The outgoing power is then multiplied by the time spend in the stage to obtain the energy consumption in the section.

Control stops

When control stops are reached the car must stop and wait for 30 minutes before continuing the race. During this time the car can charge its batteries and, since the vechicle is still, the 2nd dimension of the solar matrix is indexed with 5 (Power in loading). As done before, the incoming power is averaged over the number of 2 minutes time zones and multiplied by the time at the stop. The total time considered for the stop is set to 33 minutes, to take into account the car and driver preparation to resume the race. This time is added to the total running time.

Night and morning charges

In the stages in which the car stops for the night the incoming power accumulated while driving is summed to the one of the evening and morning charges. Also in this case the panels can be oriented for maximum efficiency and the index for the solar matrix is chosen accordingly. The incoming power is averaged and multiplied by the duration of the break. The calculation takes into account the eventuality in which the car uses the 10 minute margin

in the evening and delays the start the following morning. The end-of-the-day stop does not contribute to the total time of the race. This feature was added by the previous work [10], since it removed big discontinuities in the fitness function. An example of this is when a small speed reduction makes the car reach the end of the race the following day (15 hours later).

4.2.3 Penalty and final time

Energy constraints are taken into account by adding a time penalty to solutions that reach negative energy levels during the race. Two different norms to evaluate the penalty are used. The first one, used in the previous work[10], is evaluated as follows:

$$Penalty = \sum_{stage} |battery(stage)| \quad \text{s.t.} \quad battery(stage) < 0 \tag{4.10}$$

Where $battery(stage)$ refers to the remaining battery charge at the end of stage $stage$. In this work the infinite norm will also be tested:

$$Penalty = \|battery(stage)\|_{\infty} \quad \text{s.t.} \quad battery(stage) < 0 \tag{4.11}$$

The conversion between the penalty (Wh) and time (s) is done via the factor K (4.12). The resulting $Penalty\_time$ is added to the total time needed to complete the race, therefore reducing the fitness of the individual.

$$Penalty\_Time = K \times Penalty \tag{4.12}$$

The solar team provided some indications on the possible value of K. According to their experience, a reduction of 100Wh corresponds, on average, to a 30 seconds penalty. From this consideration, K should be close to 0.3 s/Wh. In the previous work [10], this was not taken into account and the tuning of this parameter, based only on solution quality, led to a value of 16 (53 times greater).

4.2.4 Smoothing solutions

The previous implementation [10] led to solutions with high speed difference between consecutive stages. This is a problem because sudden accelerations and decelerations require an energy expense that is not considered in the model. When the vehicle accelerates, the energy stored in the batteries is converted in kinetic energy bound to the car motion. The opposite happens during deceleration thanks to regenerative breaking. A fraction of the converted energy is wasted in the process, and this can be modeled as shown in 4.13, where v_i and v_f are the initial and final speed of the car, respectively.

$$Wasted\_Energy = K_{conv} \times \frac{1}{2}(mass\_car + mass\_driver) \times (v_f - v_i)^2 \tag{4.13}$$

According to the solar team, K_{conv} (from 4.13) should be in the [0.05, 0.1] range and can be considered equal for both the acceleration and deceleration process. The resulting energy consumption is added to the energy expense of the following stage. During the testing phase it will be evaluated if this change has the wanted effect of producing smoother solutions.

4.2.5 Summary

Table 4.1 shows what has been introduced or tested in the new version of the fitness function.

Feature	Addition
Penalty	Reduction of penalty coefficient
Norms	Infinite norm
Refinements	Smoothing of solutions

Table 4.1: Improvements in fitness function

Chapter 5

The algorithm

In this chapter the genetic operators that have been used are introduced. When useful to the discussion, the differences with the previous implementation [10] are also mentioned.

5.1 Population initialization

The population is initialized by drawing from a normal distribution centered around 85 km/h (average between the maximum and minimum allowed values, 75km/h and 95 km/h respectively). The standard deviation of the distribution is a parameter of this operator. In the case in which the car is subjected to a speed limit that is lower that 95 km/h, the speed is set to be equal to the limit, for the reasons already expressed in the previous chapter. Samples that end up outside the speed boundaries are saturated to the edge values. The size of the population, that is kept constant across generations, is also defined in this phase. In the implementation of the previous work [10], computationally feasible population sizes (in the context of the race) are in the hundreds of individuals.

5.2 Parent selection

Parents are selected using a tournament selection process. A group of K individuals is extracted from the population and the best among them (according to the fitness evaluation) is selected as a parent. The procedure is repeated with replacement until the number of wanted parents is reached. The size of the tournament is the only parameter of this operator. The number of parents to be extracted depends on the requirements of the crossover operator and on the value set for elitism.

5.3 Crossover operator

Three different crossover operators are implemented. The first one derives from the previous work [10], while the new ones try to address some of its problems that are discussed in the

following.

5.3.1 Quasi blend crossover

This operator, coming from the previous work [10], is renamed quasi blend crossover since it is a variation over a traditional blend crossover. The difference of corresponding speeds in the parents' genotype vectors is multiplied by a zero-mean normal distribution ($N(0, \sigma)$) and then added to the average velocity. Equation 5.1 shows how the jth element of a child is created. This process is repeated for each couple of speed values using different samples drawn from the distribution.

The standard deviation of the gaussian distribution is a parameters of the operator. However, since it is multiplied by the speed difference, the actual variance of the added noise is not fixed but depends on the speed vectors of the parents.

$$Child(j) = (Parent_1(j) - Parent_2(j)) \times N(0, \sigma) + (Parent_1(j) + Parent_2(j))/2 \quad (5.1)$$

The principle that shares with the normal blend crossover is the possibility that the velocities of the children lay outside the interval defined by the parents' speeds. The probability of this happening is not uniform, but become smaller as the distance from the center of the interval increases, following the trend of the normal distribution.

One possible drawback of this approach, that will be evaluated later, is the fact that the operator may lead to premature convergence of the evolutionary process. This could be caused by the averaging operation and the bias toward the center of speeds given by the normal distribution.

The behaviour of the operator is controlled by the following parameters: the crossover rate and the standard deviation of the Gaussian distribution (σ). Since two parents produce a single child, the number of selected parents must be two times the number of desired children.

5.3.2 Point Crossover

Point crossover is one of the most commonly used crossover operators. It does not introduce new genetic material, but the one already present in the parents is combined to create the offspring. N random crossover points are randomly drawn (with replacement) along the length of their speed vectors. The chromosomes of the first parent are copied into the new individual from the beginning of the speed vector, until a split point is reached. When this happens, the copy operation continues from the other parent. This happens each time a split point is encountered. The second child can be easily created by exchanging the role of the parents. For this reason the number of parents to be selected equal to the number of wanted children.

This operator was introduced to reduce the effects of premature convergence, since no averaging operations, that may harm diversity, are used. Another reason that led to its

testing was the fact that some sample runs showed that the role of the quasi blend crossover were marginal with respect to mutation. In fact, tuning of the mutation probability led to much better improvements that the tuning of crossover rate.

This operator seemed a good compromise between the quasi blend crossover and no crossover at all, which will be also tested. In fact, the operator should help with the exploration of the search space without solely relying on stochastic contributions, as in the case of only mutation.

The parameters of this operator are the crossover rate and the number of split points.

5.3.3 Blend Crossover

Blend crossover is also introduced to see how it compares to the quasi blend variation. In this case the jth chromosome of the child is created in the following way:

$$Child_1(j) = Uniform(lw - \alpha \times diff, up + \alpha \times diff) \tag{5.2}$$

$$diff = Parent_1(j) - Parent_2(j) \tag{5.3}$$

$$up = max(Parent_1(j), Parent_2(j)) \tag{5.4}$$

$$lw = min(Parent_1(j), Parent_2(j)) \tag{5.5}$$

Blend crossover allows the resulting speeds to fall out of the interval defined by parents. The magnitude of this effect can be controlled by changing the parameter α. Since the distribution used to generate the children is uniform there in no bias towards the average speed as in the quasi blend operator.

This also allows for the second child to be created using the same formula, without the problem of creating two nearly identical children. Therefore the number of selected parents equals the number of desired children.

Together with α, as in the previous operators, the other parameter is the crossover probability.

5.3.4 Crossover rate implementation

To decide whether a couple of parents will generate children, a number between 0 and 1 is randomly drawn, and if it is lower or equal than the crossover rate the parents are crossed. Otherwise the parents are transmitted unaltered as children.

In the case of the quasi blend crossover, only one child is generated from each couple of parents. When the crossover is not performed, only one of the parents must be selected. The procedure is the following: if the Nth children must be generated and N is even the first parent is selected, otherwise the second is chosen.

5.4 Mutation operator

The mutation is performed adding to each speed value of the individual a random number drawn from a normal distribution (Equation 5.6). Mutation is applied, according to the mutation rate, to all the individuals created by crossover, even if they are just copies of the parents.

$$Child(j) = Child(j) + N(0, \sigma) \tag{5.6}$$

The parameters that control the operator are therefore the standard deviation of the gaussian noise and the mutation rate. The way the mutation rate is implemented is the same as what it is done for the crossover rate.

5.4.1 Constraints

After crossover and mutation are performed, the new individuals may contain speeds that lay outside the allowed interval. Non conforming velocities are modified to be equal to the closest speed boundary. In the previous implementation [10] this operation was performed after each of the two operators, but since this did not provide any benefit in term of solution quality, the double enforcement was removed in the new version.

5.5 Survival Selection

When the new generation is created, a percentage of the old generation is kept as elite individuals. The others are replaced by the products of crossover and mutation. This is one of the easiest and more effective ways of doing survival selection and was kept from the previous work [10].

Chapter 6

Software implementation

The previous version [10] of the genetic algorithm was implemented in Matlab, using the the features available in the Global Optimization Toolbox. Despite the ease of use provided by the Matlab language, this approach showed two major drawbacks. The first was a slow speed of execution, due to the fact that Matlab is an interpreted language, while the second was the lack of parallelization. This led to the decision to re-write the algorithm using C++, which provides the speed of a compiled language and offers the ability to exploit parallelisation. This chapter presents all the details regarding the software implementation and it is structured to serve as a reference for further development.

6.1 Design objectives

The choice of C++ was made to obtain the speed of a compiled language while maintaining good readability and expressiveness of the code. Members of the solar team must be able to use it and performs modifications without extended knowledge in software engineering. The object oriented paradigm offered by the language provides huge helps in reaching these objectives. In addition to that, the C++ standard library contains all the most important data structures and algorithms, so that a manual implementation isn't required. This drastically reduces the complexity of the code and the number of possible mistakes that may be introduced. In fact, one of the most important features of the standard library is that memory management is already implemented via the RAII [4] paradigm (Resource Acquisition Is Initialization). By following this paradigm each object is in charge of freeing its dynamically allocated resources upon destruction, thus helping in the prevention of memory leaks.

The code will be structured to create a library which exports an easy to use interface. The library can be linked to other C++ executables to provide the wanted functionalities. However, at the time of writing, the Agoria Solar Team is in the process of porting its entire code base from Matlab to Python. This introduces a new requirement for the code, which is the ability of using the library from the python interpreter, without compromises on its performance.

The last aspect that was taken into account during the design and development is the

one of portability. Even if the Agoria Solar Team, as of now, deploys Windows machines to run its software, the possibility that another platform will be chosen in the future is not to be excluded. The code was made OS independent by always using standard features of the language and choosing libraries with cross platform availability. This aspects also extends to the build systems, so that it doesn't require a lot of effort and knowledge to compile and build the code on different platforms. CMake [11] provided huge help in achieving this objective, since it can be used to automatically generate build files for the target platform.

6.2 Data structures

The tensors that were introduced in previous chapters need to be stored in long term memory in a compact and easy-to-use representation. Matlab uses its own file format which is identified by the ".mat" extension. All the data contained in the weather, stage and power out matrices was stored on disk using this format.

6.2.1 Retaining the format

A first attempt to retain the ".mat" file format was made by looking for some C or C++ libraries that allow the accessing of these data files. Two different options were found. The first was to use the official MAT-API provided by MathWorks. However, the main drawback of this approach was the fact that the code had to be compiled by using a special tool that added a configuration layer over the compiler. The documentation regarding this aspect wasn't very clear and would have probably limited the possible compiler choices. To avoid compatibility and portability problems this option was discarded.

The next step was to use the open source Matio library. The documentation for this library was hard to find and not very extensive regarding the functionalities. The open-source aspect of the project, however, made it easier to look at the code and figure out problems. The installation procedure is well explained for the Linux environment, but for Windows the process is more complex and only documented in the context of the Visual Studio environment. Since the solar team uses Windows on all its machines this was big argument against its adoption. In addition, the C interface exposed by the library was not well suited for a clean wrapping into a C++ object. This led to discarding also this option and the ".mat" format altogether.

6.2.2 Hdf5 file format

The next step was to find a different file format to store the tensors. Some research led to Hdf5 (The Hierarchical Data Format version 5). This is an open source file format, developed by "The HDF Group", designed to store and organize large amounts of data. The HDF Group also provides a cross platform implementation of the standard for many programming languages, among of which C++. According to the information found online this library has become widely adopted, given its benefits of great support and reliability.

The other big advantage of this choice is the fact that Matlab natively supports the exporting functionality to the Hdf5 file format. In fact, the latest versions of the ".mat" file format are themselves based on Hdf5. This avoided problems in using the already developed Matlab scripts to produce data useful for the development and testing of the C++ library.

From the C++ side, CMake has a built in functionality for finding the library and automatically importing the information used to compile and link. This makes the process of creating CMake build scripts that work on all platform very trivial. The only drawback is found on the Windows side, where the support for the Mingw development environment is not present. Since the source code of the library is available it is still possible to manually compile the code for this environment but it is known that this leads to several issues. This limited the possible choices to the Microsoft C++ Build Tools. In the end this is the approach that was selected.

6.2.3 Tensors in memory

The C++ language already provides data structures that can be used to store tensors, but none of them fit our needs well. The first ones to be considered are C multi dimensional arrays:

Listing 6.1: C multidimensional array

```
1 //define a simple matrix
2 double mat[2][3] = {{1, 2, 3},
3                     {4, 5, 6}};
```

The data in this case is organized in physical memory as a linear array, saving data along the rightmost dimension first and proceeding with the others, as are listed from right to left (row major ordering). Therefore, in memory, the matrix will be saved as 1, 2, 3, 4, 5, 6. In the case of an higher number of dimensions the same process is applied.

Given the indexes for each dimension and having the size of each dimension the offset in the linear vector can be easily computed:

$$Pos = Index(N) + \sum_{i=i}^{N-1} Index(i) \prod_{k=i+1}^{N} Card(k) \tag{6.1}$$

In 6.1 $Index(i)$ indicates the index referring to the ith dimension in the matrix. $Card(k)$, instead, refers to the cardinality of dimension k.

The storage of the tensor does not have any overhead in terms of space. In addition, since the number and cardinality of dimensions is known at compile time, the results of the product operator (in 6.1) are pre-computed. These values don't even have to be fetched from memory since they can be embedded as constants into the assembly instructions. This makes the indexing process very fast. The major drawback, however, due to the fixed the number of dimensions and cardialities, every change in the data structures requires a change in the code.

The second approach would have been to use dynamic C arrays:

Listing 6.2: C dynamical array

```
1 //define a simple matrix
2 double** mat = new int*[N];
3 for(int i = 0; i < N; i++)
4 {
5     mat[i] = new double[M];
6 }
```

This solves the problem of the fixed cardinalities, even if the number of dimensions still cannot be changed, but introduces big drawbacks in terms of space and access times. The matrix is not stored as a linear array anymore, but there is an array of pointer that points to the different rows of the matrix, that are stored as a linear arrays. By adding dimensions the level of indirection increases, since an array of pointer will point another array of pointers and so on and so forth... The access time will becomes linear in the number of dimensions, since each pointer is the chain must be individually fetched. The need of storing of the arrays of pointers also increases memory occupation.

Another element to take into account is that each row (in the case of a two dimensional matrix, but this can be extended) is allocated separately. This makes the data structure to be separated in memory, with each row possibly residing in a different cache line. This adds the problem of frequent cache misses while accessing the data structure.

The option of using the vector class of the C++ Standard Template Library (STL) is similar to the one just discussed.

Listing 6.3: C++ STL multidimensional vectors

```
1 //define a simple matrix
2 std::vector<std::vector<double>> mat;
```

The last option is to manually store the whole tensor as a linear array, which requires of keeping track of the information regarding number of dimension and cardinality. This has the benefits of both approaches and is used as a starting point in our implementation. In order to benefit from automatic memory management and boundary checking (only for development phase) STL vectors are used instead of C arrays:

Listing 6.4: C++ STL vectors

```
1 //define a simple matrix
2 std::vector<double> mat(size);
```

The size parameter can be computed when information on number of dimensions and cardinalities is obtained from the file that stores the tensor.

6.2.4 The MatVar class

The design objective is to have a class representing a generic tensor, similar to a matrix variable in Matlab. Objects of this class must be able to load tensors from storage, save them

into memory, provide information about the number of dimensions (rank) with respective cardinalities and, most important, allow for fast indexing. The UML diagram of the class is presented in Figure 6.1.

The data of the matrix is stored as an unidimensional C++ std::vector. The information on the rank and cardinality of dimensions are also present as private members and are needed to allow indexing. The interface is quite simple and gives access to these attributes via normal getters.

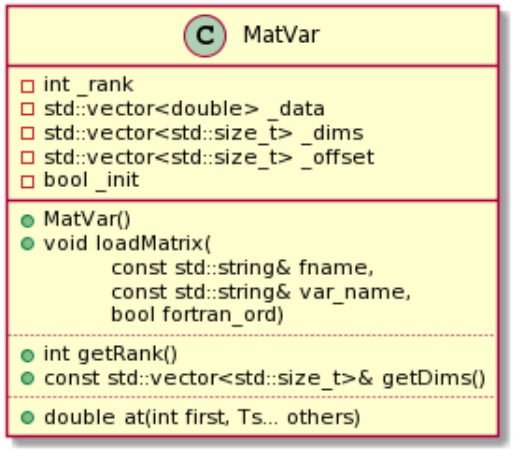

Figure 6.1: MatVar class

However, the important parts to discuss are the loading and indexing of the matrix. The loading is done via functionalities provided by the Hdf5 library. Two different interfaces for the library are used: the standard C interface and the Hdf5 lite one, which provides an higher abstraction level. The procedure is composed of two main step: reading the dataset information (rank and cardinality of dimensions) and the copy in memory of the tensor.

From the first step the total size of the data set can be computed and the standard vector can be resized to be able to contain it. The pointer to the beginning of the vector is passed to a library function that loads the matrix starting from the provided address. By using a unidimensional vector there is the added benefit that the shape of the data does not have to be changed to be saved in RAM, thus avoiding waste of computation power and longer waiting times. The offset vector is also filled with the results of the precomputed products, as presented in equation 6.1.

The indexing is done using the public method "at"(Listing 6.5). Since the shape of the matrix is not pre-defined the function must be able to accept a variable number of parameters. In C++ this can be done by exploiting functionality introduced in the 2017 standard for the language: parameter pack expansion [3]. This works in a similar way as templates do. Each time a new template specialization for the signature is used in the code, a new function is automatically generated at compile time. The structure of the function can be shaped according to the list of the given parameters.

Listing 6.5: C++ STL vectors

```
1  double MatVar::at(int first, Ts... others)
2  {
3      int i = 0;
4      std::size_t index = static_cast<std::size_t>(first)*_offset[0];
5      (void)std::initializer_list<int>
6      {
7          (index += _offset[++i]*static_cast<std::size_t>(others), 0)...
8      };
9      return _data[index];
10 }
```

As shown in the code at line 7, the expressions calculating the value for the actual index are derived by expanding the one parametrised with "others". The "..." symbol says that for each parameter that is assigned to "others" a new expression is computed that updates the value of index. The variable "i", that indexes the vector of pre-computed coefficients, is incremented in each of the newly generated expressions. The initializer list is used to impose an order on the evaluation of the generated expressions, so that the increments of "i" and update of "index" are performed in the correct order.

In this way the class presents only one method that can index matrices with any number of dimensions while removing the need of multiple functions. Since all the expressions are singularly specified when the parameter pack "others" is expanded, the indexing the "_offset" vector does not require a loop cycle . The usage such of a construct to go through "_offset" would require jumps in the code, slowing down the computation.

The speed of the indexing operation is a key factor for a fast algorithm, since the number of calls of this function is linearly dependent of the number of generations, number of individuals and the length of the speed vectors.

6.2.5 Matrix representations

This section makes some brief considerations on the way Hdf5 tensors are stored that are essential to avoid mistakes and to optimize the speed of the computation.

At first we have to differentiate row major and column major orderings. The former is the one already presented in the code at 6.1, where data is stored by linearizing the tensors along the rightmost dimension first. This is the approach taken by C style languages. The latter does the opposite, by storing the tensors starting from the leftmost dimension. This is the approach is taken by languages such as Fortran. This is important because Matlab, at its origins, was written in Fortran and then converted to C and C++. However, the column major ordering was preserved in the new implementation.

The Hdf5 file standard, when saving a matrix, does not perform a conversion of the representation for obvious performance reasons. However, the standard says that the matrix dimension must be reported using the C notation, by indicating the slowest changing dimension first. For this reason, when a matrix is saved using the Fortran or Matlab Hdf5

API, the vector of dimensions is reversed. The opposite process is performed upon loading. Therefore, when the matrix is generated in a column major representation, the dimensions of the matrix, upon loading, will appear reversed in a C++ application.

The problem was addressed by using the Boolean flag "fortran_ord" in the loading method. By setting this to true the list of dimensions is reversed before the creation of the "_offset" vector used for indexing. This allows to maintain the same indexing method (column or row major) of the generating application. Since the matrices used for this work are generated with Matlab, the flag was set to true to have the same index order. When the codebase will be moved to python, by setting the flag to false, it will be possible to use the same indexing order of the python environment (row major). According modifications will be obviously required in the code.

It is important to mention that the matrix representation, i.e. which is the fast and the slow changing dimension, plays an important role in the performance of the algorithm. In our case, accesses to the matrix are usually performed along the time dimension to compute time averages. In order to reduce the number of cache misses during the fitness evaluation, this dimension is chosen to be the fast changing one in all matrices.

6.3 Software structure

In this section presents an in-dept description of the software implementation. Figure 6.2 shows an high level UML class diagram of the software. For the sake of simplicity not all the classes are reported, but each one of them will be eventually introduced and explained.

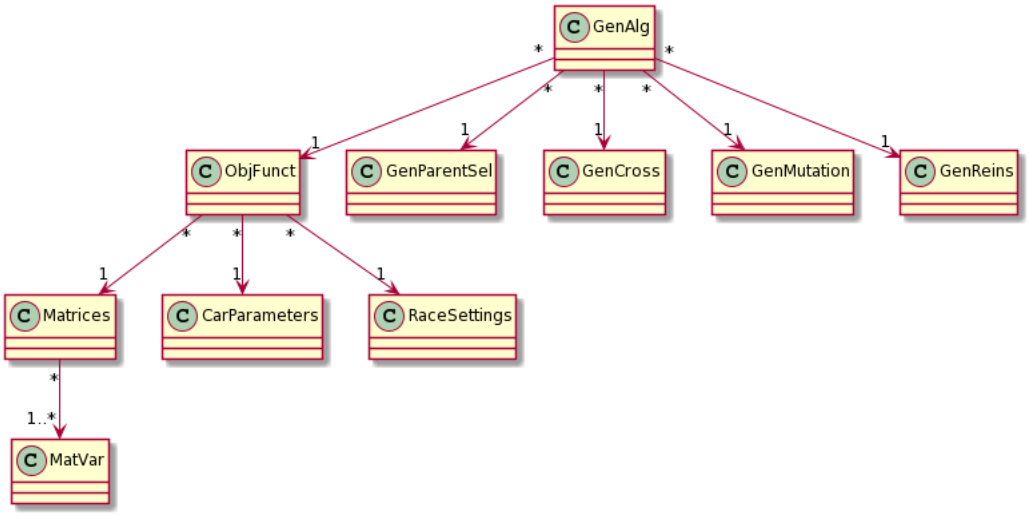

Figure 6.2: High level class diagram

6.3.1 Objective function

The ObjFunct class represents the search problem and encapsulates all the problem specific information. It is designed this way so that it is easier to perform modifications without having to propagate the changes to the rest of the code. This is meant to help avoiding errors during future development. To achieve the decoupling, ObjFunc is the only class that has access to the problem data. This aspect is also useful from a redundancy perspective, because it removes the need of spreading references to the same data structures all around the code.

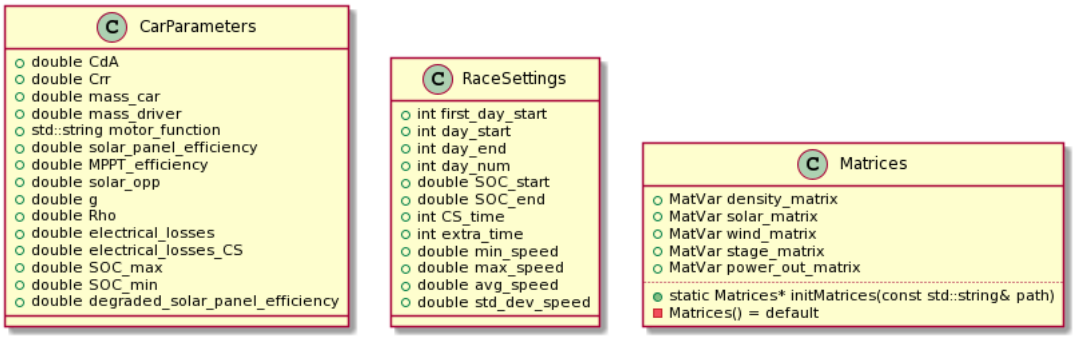

Figure 6.3: Data classed

The data is contained in three different classes: Matrices, CarParameters and RaceSettings. The full definition is reported in Figure 6.3. Matrices just contains the various tensors, which are stored in MatVar objects. CarParameters has all the information regarding the solar vehicle, as it is shown in the image. The same goes for RaceSetting, which contains information about the race. By changing their content, the algorithm can adapt to different race rules and car designs. ObjFunct refers to these classes via raw pointers. The use of shared pointers was avoided in order to reduce the access time to the data. Memory management is not handled by ObjFunct and needs to be performed externally.

The UML class diagram of ObjFunct is shown in Figure 6.4. The design of the class allows for standalone usage, outside the context of the genetic algorithm. In fact, the method operator() can be used to evaluate a single individual. By setting flag save_state to true, the class records all the information of the solution that it evaluates. This is the energy profile, the penalty profile and a boolean value indicating whether the solution reached a negative energy level during the run. This information can be later retrieved via specific getters. If save_state is set to false, only the final time is returned. evalPoupulation uses the latter option when calling operator() to evaluate the fitness of the entire population. evalPopulation is also the method that GenAlg uses during the algorithm run.

The ObjFunct class is also in charge of the diversity calculation, the generation of random individuals for the initial population and the enforcement of constraints to the new generated solutions. These tasks are assigned to ObjFunct in order to decouple problem specific information from the rest of the genetic algorithm, as explained before.

Figure 6.4: ObjFunct class diagram

More detailed aspects are left out for the sake of brevity. The parallelization functionalities will be discussed separately in the following section.

6.3.2 Operators

The genetic operators all follow a similar design patter. The mutation operator (Figure 6.5) will be used as a reference to explain the design choices. What will be discussed can be easily extended to the other operators.

All the operators are designed according to the inheritance pattern. A base abstract class defines the principal features of the operators that are specialized in the children. In particular, it declares operator() as an abstract method. Derived classed will implement the specific behaviour of this function. In addition to the abstract property, the method is also virtual, thus enabling polymorphic behavior. This allows the GenAlg class to store a references to derived classes as pointers to the base one. In this way, new operators can be defined and fed to the algorithm without requiring any modification. Their behaviours become completely transparent to the caller, which only sees the interface provided by the base class. All these features make it really easy to introduce new operators and to modify

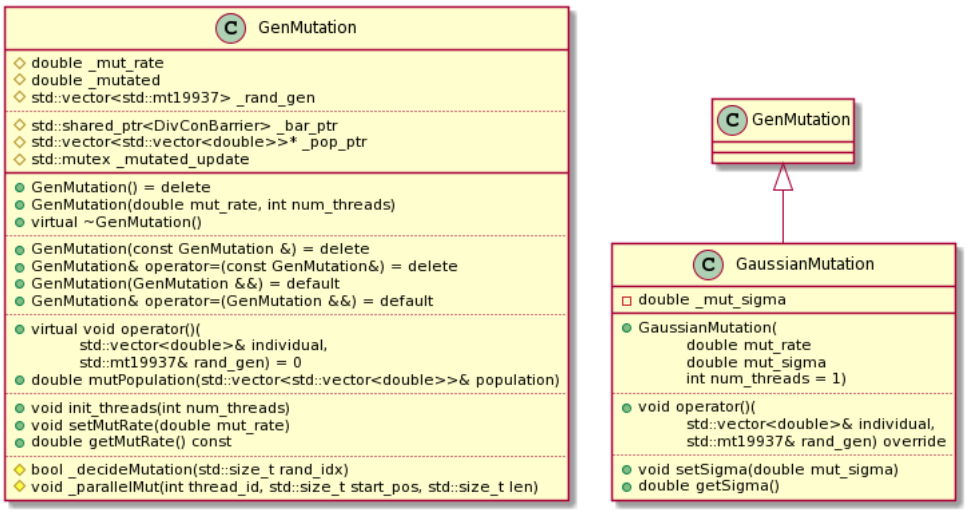

Figure 6.5: Mutation base class and gaussian derivative

the way the algorithm operates.

The mutation base class and the gaussian derivative are shown in Figure 6.5, (left and right, respectively). As it is illustrated, the base class also contains parameters that are general to all the possible implementations, such as the mutation rate. The children will add more of them, as is its shown for the standard deviation of the noise in the gaussian mutation.

It is important to mention that the base classes of mutation and crossover, as it also happens for the fitness function, completely encapsulate parallelization. In this case, however, this also is completely transparent to the children classes, which are totally unaware of that. The only restriction imposed by parallelization is that derived classes cannot change their state via the operator() without explicitly looking the resources they modify. More information on parallelization will be provided in a specific section.

The other base operators and their derive classed which are used in this work are shown in Figures 6.6, 6.7, 6.8 and 6.9. While parents selection and reinsertion are simple, also due to the lack of parellelization support, the children of GenCross can overrive an additional method of the parent: parentsMutliplier. This is can return either 2 or 1, indicating if, given the number of wanted children, the number of parents has to be doubled or not. If it is not overridden the default return value is 1, but, for example, QuasiBlendCross changes that to 2.

6.3.3 GenAlg

The GenAlg class is shown in Figure 6.10. As explained before, it stores pointers to the fitness function and to the base class of the operators. These need to be set upon construction and can be subsequently changed via the appropriate setters. Memory management of these objects is not performed by the class and has to be done externally.

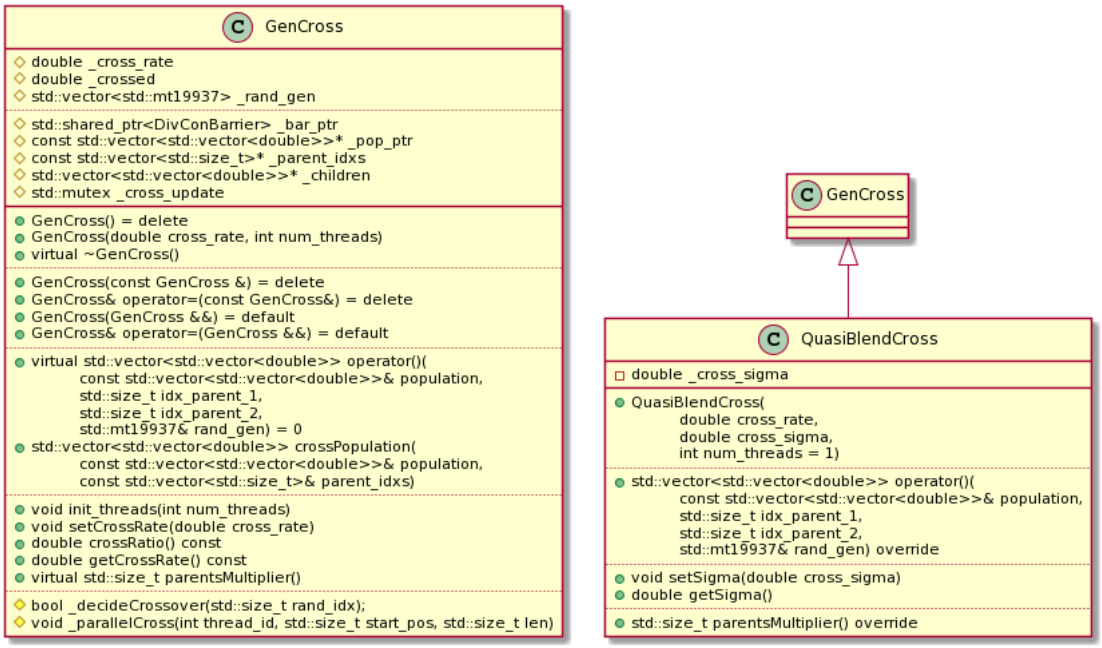

Figure 6.6: Crossover base class and quasi blend derivative

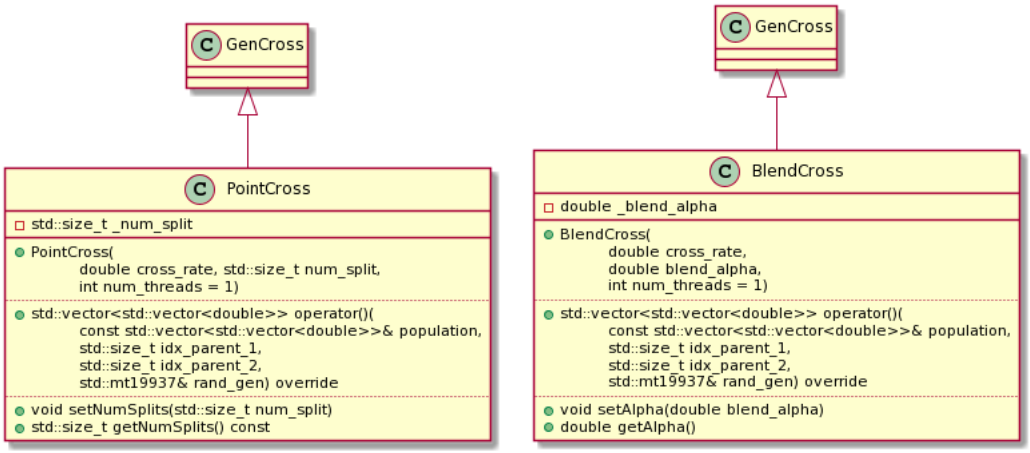

Figure 6.7: Point crossover and blend crossover

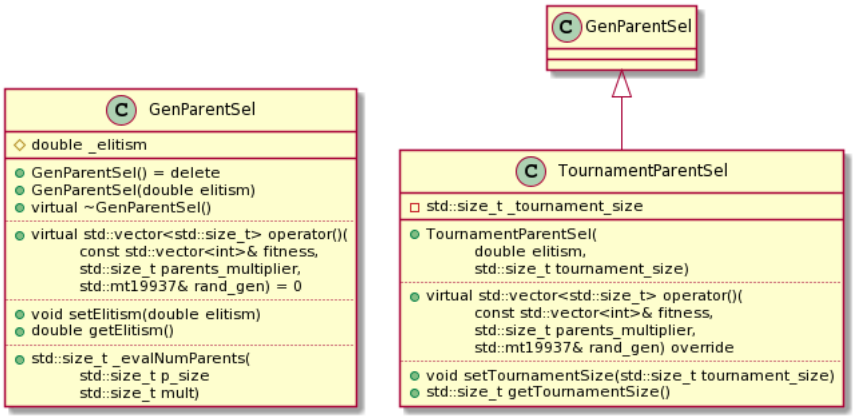

Figure 6.8: Parent Selection base class and tournament derivative

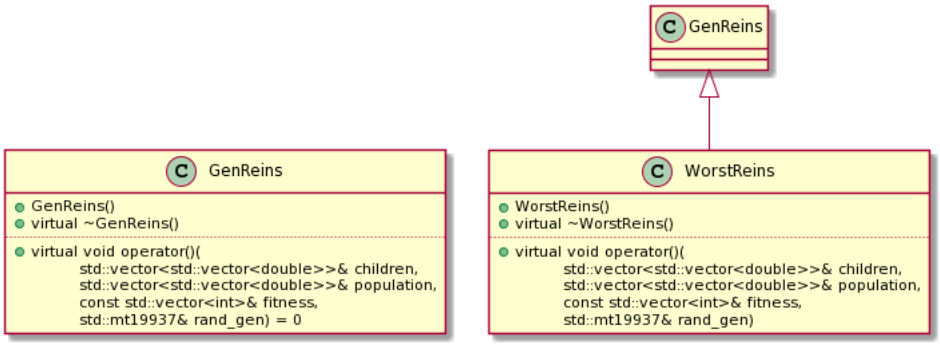

Figure 6.9: Reinsertion base class and derivative

Since the parameters for the operators are contained in their respective objects, the only two remaining, number of generations and number of individuals, are provided as arguments to the run method, which starts the computation.

During the run, the worst, best and average fitness of each generations are recorded. The same is also done for diversity. This information can be obtained through the appropriate getters while additional ones provide access to the final population and its fitness.

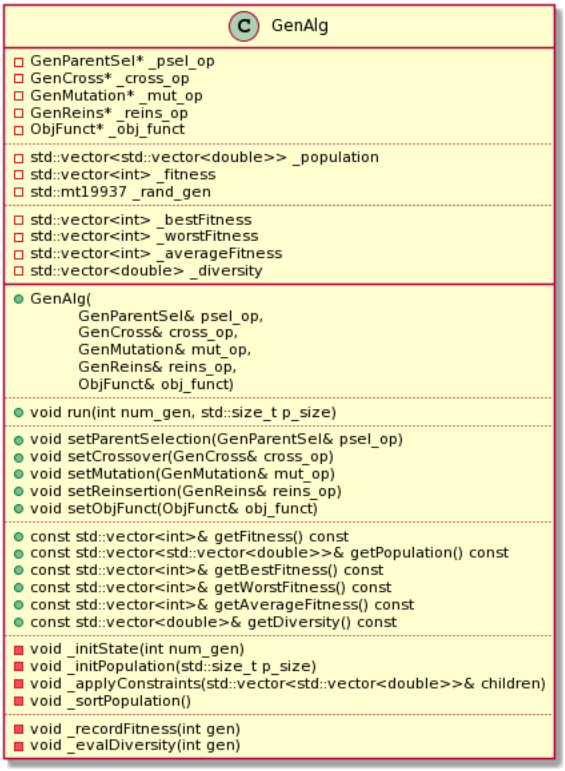

Figure 6.10: GenAlg class

6.4 Parallelization

Multicore CPUs are the norm nowadays and powerful workstation are widely available, providing easy access to parallel computation. However, the implementation of genetic algorithms offered by Matlab does not provide any support for parallelization as of now.

On the contrary, C++ offers great support for multi-threading, both in the standard itself and external libraries. One of the most used one is OpenMp, which was the first option to be considered for this work. As it is explained in [9], OpenMp gives great support for data parallelization which can be achieved with great simplicity. Is is sufficient to add simple compiler directives to execute entire blocks of code in parallel. The easiest example is when an array is iterated with a simple for loop and a computation is performed on each one of the

elements. Multiple threads are spawned before starting the loop, each operating on different portions of the array. Obviously, this is only possible when computations on one element depends on the results of computations on the preceding ones.

Another important aspect to take into account is the granularity at which parallelization is introduced. For what concerns this work, there are two different options to choose from. The first is to have it at the individual level, with multiple threads operating on different parts of the same solution. The second one is to work at the population level, with different threads operating on disjointed subgroups of the population, where individuals are considered in their entirety. The first option can be immediately discarded, because, by taking a look at the fitness evaluation, the incoming and outgoing energy of a stage depends on the position at which the car is found, which can only be obtained by first evaluating the preceding stages. Instead, the second option is far easier to implement, because the fitness function, and most of the operators, usually operates on single individuals (two in the case of crossover) and independently of the other members of the population. By considering the population as an array of solutions, it is possible to see the problem as a specialization of the simple for loop example previously introduced.

6.4.1 OpenMp

Once the software design was established, according to what was discussed in the previous chapter, it was clear that each operator should have taken care of the parallelization of its own operations. To offer the wanted decoupling between the operators and the algorithm, it was not possible to introduce parallelization on the algorithm as a whole. This would have introduced strict relationships between different operators, leading to the disruption of one of the main design objectives.

A simple model of the parallelization problem is presented in the following pseudocode:

Listing 6.6: Modelling of the parallelization problem

```
1  for (int gen = 0; gen < num_gen; ++gen)
2  {
3      //fitness evaluation
4      for(int ind = 0; ind < num_individuals; ++ind)
5      {
6          eval(population[ind]);
7      }
8      //parent selection
9      auto parents = parent_sel(population);
10
11     //crossover
12     for(int par = 0; par < num_parents; par+=2)
13     {
14         children[par] = ...
15         children[par+1] = ...
```

```
16      }
17
18      //mutation
19      for(int chld = 0; chld < num_children; ++chld)
20      {
21          children[chld] = ...
22      }
23
24      //constraints
25      applyConstraints(children);
26
27      //reinsertion
28      reinsert(population, children);
29
30      //sorting
31      sort(population, fitness);
32  }
```

The algorithm is modeled by the outermost for loop, which iterates over different generations. Fitness evaluation, crossover and mutation are also represented as for loops over the population, selected parents and children of crossover, respectively. As explained before, it is perfectly possible to parallelize the computation by splitting the for loops over multiple threads, since no dependencies are present. However, the important factor to consider is that these operators can be distant both in space and time. In space because they reside in the compilation units of the respective classes, which are separate from each other. In time because serial computations can be interleaved between them. This happens because, as it will be shown, not all the operations, such as parent selection, reinsertion and sorting, provide considerable advantages when parallelized. In addition to this, since each operator is independently defined, it is possible for some of them to enable parallelization while others don't.

By considering a naive implementation of OpenMp, it is possible to assume that threads are created before the loop of each operator and destroyed after it finishes, via a join call from the main thread. The problem is that, at each new each generation, threads have to be created again and then destroyed. The creation of a thread, even if not as expensive as the generation of a new process, still requires lots of work from the operating system.

In fact, many real OpenMp implementation try to amortize this cost by keeping the threads alive when the same parallel loop is encountered multiple times. However, it is safe to assume that this is harder to achieve when parallel sections are distant from each other and from their next repetition.

To avoid possible slow downs due to this problem, and given the fact that the use of OpenMp restricts the choice to fewer compiler options, another route was chosen.

6.4.2 DivConBarrier

The great parallelization support offered by the C++ standard, and the relatively limited complexity of the problem led to the design of a custom parallelization class: DivConBarrier (Divergence Convergence Barrier). The objective of the class is to provide the same functionalities offered by OpenMp, while assuring that threads executing each parallel section are not destroyed and re-created at every generation.

The general concept, as expressed by the name, is that the main thread, which executes the serial operation, is able to diverge the flow of execution over multiple threads when a parallel section is encountered. It will then wait for all the threads to finish before proceeding to the following serial operations (convergence). The "Barrier" part of the name was chosen to highlight the fact that the main thread is stopped at an ideal barrier while waiting for the working threads to finish.

The UML class diagram is shown in Figure 6.11.

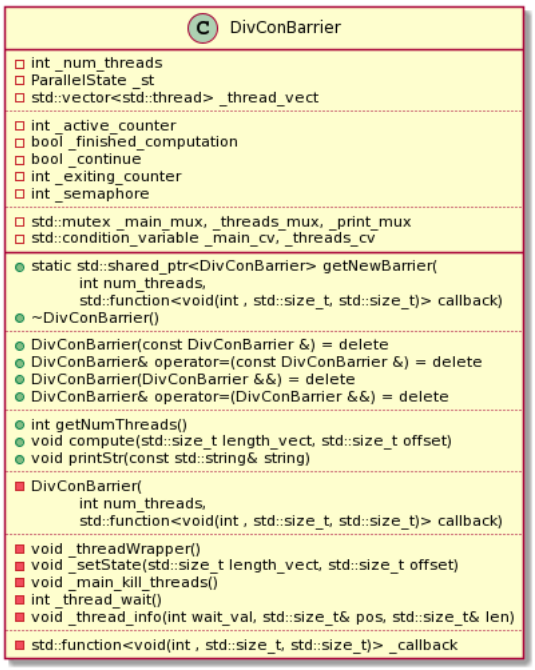

Figure 6.11: GenAlg class

The class is responsible to create, maintain and destroy (via RAII [4]) a pool of threads of a fixed size, which is defined upon construction. In this way, it is not necessary to destroy and re-create threads at every generation. The class uses a factory design pattern [17] that only returns shared pointers to newly created instances. This is done to prevent the spread of raw references to instances of the class which may be already destroyed. Shared pointers assure that the class is deleted only when nobody is still referencing it. The implementation is based on reference counting, as explained in the documentation [6].

A reference to the barrier is initialized and maintained in each operator that wants to exploit its functionalities. The constructor takes as parameters the number of wanted threads and a callback. The latter is called on each worker thread when the parallel computation starts. The callback is stored in a std::function object that is able to represent both simple function pointers and functors (object of classes defining functions, that may also contain a state). The ObjFunct class and the generic operators that are parallelized, all define methods that are sent to DivConBarrier as callbacks (eg. _parallelEval, _parallelCross, etc..). Since the methods of a class have access to its internal state, it is possible to save some information into it, so that it can be accessed by the callback when it is called. Examples of this are usually a pointer to the population, to fitness measures or to vectors in which to store the created data, such in the case of children for crossover. In order to achieve this, however, the method takes as a first hidden parameter the pointer to its class (called "this"). This would make the signature of the callback different from the one required by DivConBarrier. For this reason, std::bind [5] is used to dynamically create a functor with the right signature. "this" is removed from the parameters and fixed in the functor's state.

Once the callback is set up, the main thread just needs to call the compute method. The provided parameters are the total size of the vector on which the computation has to be performed and an optional offset in the case not all the elements have to be considered. As an example, the latter can be different from zero to avoid re-evaluation of elite individuals. DivConBarrier will evaluate the best work assignment for the threads and set the callback parameters accordingly. The signature of the callbak provides an identifier for the thread (from zero to the number of threads minus one), the starting position of the elaboration and the number of elements to be considered. The thread number is useful when indexing a vector of resources, each to be used by a different thread, such as in the case of random number generators in GenCross and GenMutation.

The basis observation on which the work subdivision among threads is based on is the fact that the computation will be as slow as the slowest thread. Three situation may be encountered: the number of elements is smaller than the number of threads, the number of elements is divisible by the number of threads or the number of individuals is higher but not divisible by the number of threads. In the first case the surplus of workers is not used, while in the second case perfect work subdivision is reached. The third and more general case is solved as follows:

Listing 6.7: Work subdivision in third case

```
1 //assume third case and integer division
2 ind_assigned = num_individuals/num_threads;
3 ind_assigned++;
4 used_threads = num_individuals/(ind_assigned);
5 assigned_last = num_individuals % ind_assigned
6 assigned last > 0 ? used_threads++ : assigned_last = ind_assigned;
```

Assuming the third case, performing the integer division between number of elements and number of threads gives an assignment per thread that is not sufficient to satisfy all the

needed work. Therefore, the assignment is incremented by one and the number of needed threads is re-evaluated by the division on line 4. The result will be less than the number of total threads. The additional work, if present, is then handed to an additional thread. Otherwise a new worker won't be added and the last thread will perform the same work as the others.

This solution is optimal in the sense that the work will be performed with the smallest increase in computation time and with the lowest number of threads. The first is because the increment of the work for each thread is just one (lowest possible) the last thread will have equal or less work than the predecessors, thus avoiding bottlenecks. The lowest amount of used threads is also desirable, since the synchronization effort comes with a cost that increases with their number.

The specifics of the implementation are not reported, because the most of the discussion would be to explain concepts that can be better understood in specialized books about thread synchronization and locking. It is enough to mention that it heavily relies on mutexes and condition variables to achieve synchronization in the critical phases (thread initialization, computation and thread destruction). Great attention was also devoted to prevent busy waiting and other inefficiencies.

6.4.3 Performance evaluation

The use of paralllellization where it isn't needed can introduce more drawbacks that benefits, since it increases the complexity of the software and therefore the exposure to programming errors. With this consideration in mind, it was evaluated which parts of the code were more expensive in terms of computation time. The algorithm was run for 100 generation with a population ranging from 100 to 1000 individuals (with steps of 100). Crossover and mutation probabilities were increased to 100% to simulate the worst case scenario. The results, averaged over the number of generations, are shown in Figure 6.12. Quasi blend crossover was used for this test, but similar results were also obtained with other crossover operators.

By looking at the picture, it is clear that, apart from crossover, mutation and fitness evaluation, the effects of the others operations are negligible. For this reason, it was chosen to parallelize only the expensive part of the computation. When the same tests are run on 4 threads on a quad-core machine (6.1, the same used for the previous test), the results shown in Figure 6.13 are obtained. The algorithm speed, as expected, is roughly quadruplicated. This trend will continue until the effects of the other parts of the computation won't be negligible anymore.

The last consideration to be made is about the non perfect linear behaviour of the crossover and fitness evaluation in Figure 6.13. This may be caused by the scheduler of the operating systems, that reallocates CPU resources to threads of other processes. In fact, these tests were conducted while other programs were running on the same computer. By using a dedicated machine, where it is easier to limit the number of interfering tasks, this effect can be reduced. This could also lead to meaningful improvements in running times.

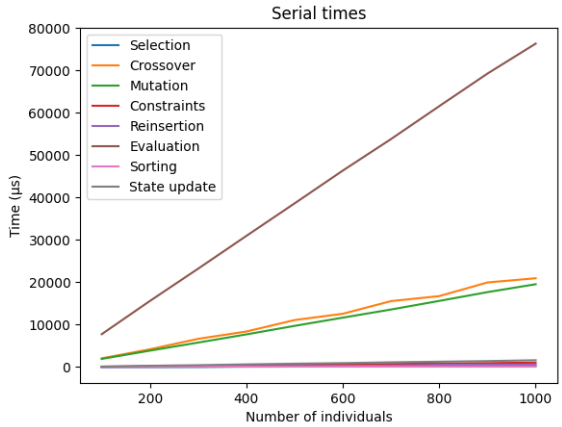

Figure 6.12: Time measurements for serial computation

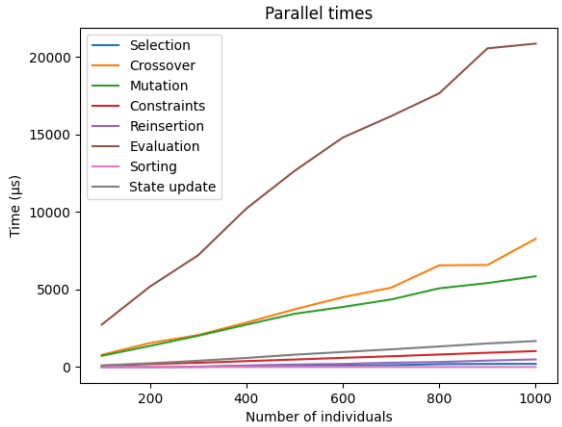

Figure 6.13: Time measurements for parallel computation (4 cores)

6.5 Python bindings

One of the main requirements is the integration of the library in Python. To achieve this goal the pybind11 library [8] was used. It automatically generates python wrapping for C++ classed, enabling the access to all the public methods and attributes from the python side. The library achieves this by exploiting the functionalities of template meta-programming [19], a very important and powerful feature of the C++ language.

The use of the library is very straightforward, due to its clear documentation and friendly design. Few lines of code have to be written to export the wanted feature. In fact, the interface of the classes accessible from Python can be defined as any subset of the C++ one. However, there are few important factors that have to be carefully considered. The main one concerns memory management of class instances. Since the objects are used from both the python and C++ side, it is fundamental to establish who is their actual owner, which

CPU model	Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
Architecture	X86_64
Cores	4
Threads per core	2
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	8192K
Ram model	SK Hynix DDR4 HMA851S6JJR6N-VK
Speed	2667 MT/s
Capacity	4GB
CAS Latency	19
Num dims	2

Table 6.1: Computer specifications

is also responsible for their destruction. A common mistake, for example, is to have the python garbage collector automatically destroying an object that is currently in use on the C++ side.

The second important aspect is that frequent interactions between the two environments are quite expensive and have to be minimized as much as possible, especially during the run of the algorithm.

6.6 Differences from previous work

The new implementation of the algorithm presents some noteworthy differences from the previous work [10].

6.6.1 Crossover and mutation

The first and main one is the fact that the Matlab Optimization Toolbox uses a less common approach for what concerns crossover and mutation. In their implementation, crossover and mutation are exclusive. Given the selected individuals, a fixed percentage of them (crossover ratio) is crossed while the remaining ones are subjected to mutation. In the C++ implementation, as it is more common in literature, the percentage of individuals that at each generation is subjected to crossover an mutation is not fixed. Each individual and couple of parents is randomly selected for mutation and crossover following the probability defined by the respective rates. In addition to this, the mutation applies to both the children of crossover and to the individuals that were unaltered during the crossover phase.

6.6.2 Fitness evaluation

A second difference was introduced in the fitness evaluation. In the Matlab version, the time was expressed in different units of measure that were stored in floating point variables. The C++ version, instead, unifies the unit of measure of time to seconds, and stores it using integers. This introduces a one-second time discretization for all the time measures, which is completely acceptable given the problem. This was also meant to prevent the algorithm from focusing the optimization of race time on fractions of seconds, which is completely irrelevant and counterproductive. However, as the tests showed, the change did not introduced a noticeable difference in terms of solution quality.

6.6.3 Constraints enforcement

The third difference concerns the application of constraints to the newly generated solutions. Since in the old version the sets of individuals that undergo mutation and crossover are disjointed, when new solutions are generated, constraints are applied to all the individuals that are produced. In the new implementation, instead, constraints are only applied after the mutation phase. In fact it is possible, for a children of crossover, to undergo to mutation while containing illegal speed values. This is only fixed after mutation, when constraints are enforced on all the newly generated solutions.

6.6.4 Fixed errors

The new version also fixes some errors and imprecisions in the old code. Due to a conversion error, the value of the evening extra time led to a wrong evaluation of the individuals. In addition to this, when a later arrival was considered, the incoming energy during the extra time was added twice: one time as incoming energy while driving, and the second as part of the night stop. This is basically negligible in terms of the influence on the evaluation and was already been noticed, so it counts as a refinement.

Other small changes involved the indexing of the solar matrix in some specific situation, such as the morning and evening charges, and a different use of the efficiency factor of the solar panels, that takes into account the overheating influence when the car is still.

Chapter 7

Testing and parameter tuning

The first objective of this chapter is to describe the testing and tuning procedure used to evaluate the performance of the algorithm while trying to maximise it. The second goal is to provide some insights on the algorithm's optimization process and on the solutions that it produces. All the tests and results that will be presented have been obtained using the machine described in Table 6.1.

7.1 Performance metrics

In order to test and tune the algorithm it is important to establish a way to measure its performance given a parameter configuration.

7.1.1 Main metrics

The main performance metric is the fitness of the best solution at the end of the run. This is chosen because, in the context of the race, it is important to find a really good solution one time rather than having not so good ones more consistently. This is also supported by the fact that a run of the algorithm, in a serial implementation, takes less than one second in a standard configuration (100 generation, 100 individuals). This allows for multiple runs, thus reducing the need of more consistent results. Together with the best fitness, the final population's average fitness is also recorded, to provide some additional information on the other parts of the population.

Another useful metric is the computation time required to find the best solution. The problem with this metric is the strict hardware dependency that affects it. This is misleading because of the discrepancy in hardware performance between the machine used for testing (laptop) and the ones that are available during the race (server workstations equipped with high core count CPUs). However, this is still useful in the context of relative measurements when different parameter combinations are compared.

The last metric that will be considered is the generation at which the best solution is found. This is very useful in establishing the actual speed of the algorithm. In fact, if the

solution is found early in the run, the actual computational effort can be reduced by decreasing the number of generations. However, this shows a problem when slight increments of the solution time (few seconds) happen late in the run. This can be prevented by establishing a minimum improvement according to which the solution can be considered to be better.

7.1.2 Diversity

An important factor for a genetic algorithm to effectively explore the search space is to maintain enough genetic diversity in the population. If the diversity decreases too quickly in few generation, the algorithm can be affected by premature convergence. This is when the population immediately converges to a local optimum instead of exploring the solution space in search for a better one (and possibly a global one). This issue badly affects multi-modal search problems, which present lots of local optima that may lead to sub-optimal solutions.

$$Diversity = \frac{1}{N_{st}} \sum_{stage} \sqrt{\sum_{ind} ((speed(ind, stage) - \frac{1}{N_{ind}} \sum_{ind} speed(ind, stage))^2} \tag{7.1}$$

The population diversity is evaluated according to 7.1, where N_{ind} and N_{st} represent the number of individuals in the population and the length their speed vectors (number of stages), respectively. $speed(ind, stage)$ refers to the speed of solution ind at stage $stage$.

At the end of each run the diversity of the last generation is recorded to see if the algorithm has converged. If this is the case, it is improbable that increasing the number of generation will provide significantly better solutions. The speed of convergence can be determined by recording the number of the generation after which the diversity is close, in terms of a percentage, to the final one.

7.1.3 Other metrics

A summary of the aforementioned performance metrics is reported in Table 7.1. These will be the ones used to compared different parameters combinations. To take into account the stochastic nature of the algorithm, multiple runs will be performed for each parameter set, as it will be explained later in the document.

However, before proceeding to the tuning phase, it is important to get an idea on how the various operator parameters and penalties influence the evolutionary process and therefore the quality of solutions. This is also useful when a good tuning is achieved, to see how the algorithm generates good quality individuals. For this reason, in the sample runs in section 7.2, the best, worst and average fitness of each generation are recorded, together with the population diversity. It would be too complex to use all this information to compare parameter sets but it is essential to understand the behaviour of the algorithm and the effects of changes in the parameters.

Performance metrics
Best fitness in final population
Average fitness in final population
Diversity in final population
Generation number of low diversity
Generation number of best individual
Computation time

Table 7.1: Performance metrics

7.2 Testing the old configuration

Before starting the testing and tuning phase, some observations have to be made on the way the algorithm operates. Some of the issues present the previous work [10] will be discussed, and the proposed solutions will be explained. For this reason, some sample runs have been performed with the old algorithm structure and parameter combination. All the details are reported in Table 7.2.

It is important to note that these experiments were conducted using the new software implementation, which presents some differences with respect the old one, as explained in previous chapters. To face the fact that the old version strictly separates new individuals in mutation and crossover ones, while the new one mutates both the children created from crossover and the uncrossed parents, the mutation rate was set as follows:

$$Mutation\_rate = 1 - Crossover\_rate \tag{7.2}$$

Role	Operator
Parent Selection	Tournament
Crossover	Quasi blend
Mutation	Gaussian
Reinsertion	Worst replaced
Penalty	Integral (Eq. 4.10)
Penalty Acceleration	Not present

Parameter	Value
Num generations	100
Population size	100
Elitism	0.01
Tournament size	6
Crossover rate	0.65
Crossover σ	0.4
Mutation rate	0.35
Mutation σ	1
Penalty factor (K)	16

Table 7.2: Operators and parameters

Figure 7.1 shows a sample run where diversity, best, average and worst fitness are represented for each generation. The weather data consisted of actual measurements from September 2019, which were introduced in previous chapters. Figure 7.2 presents the best

solution at the end of the run and its energy profile. The whole individual is split along three consecutive sections of the route for ease of representation. Figure 7.3, instead, shows the fitness distribution in the final population. Lots of considerations can be made on these images.

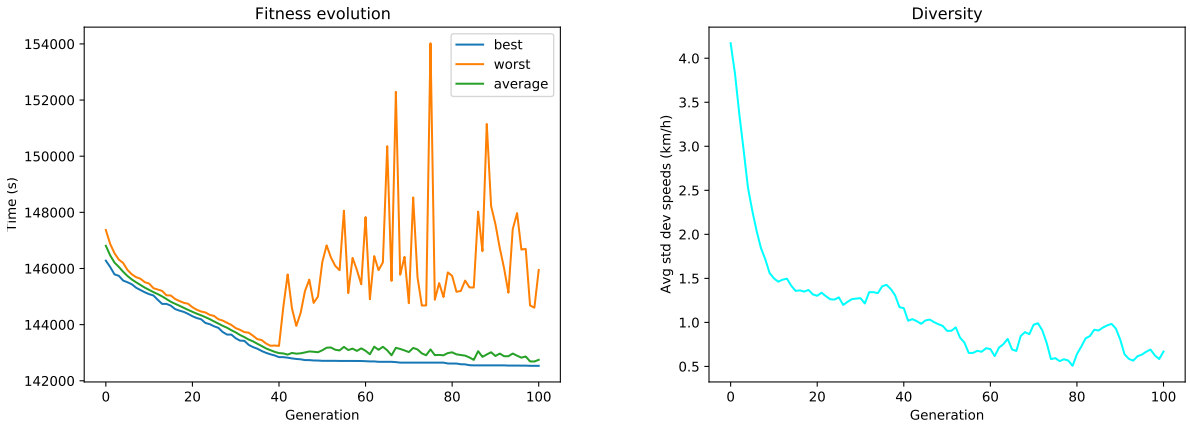

Figure 7.1: Evolution of fitness (left) and diversity (right) in sample run

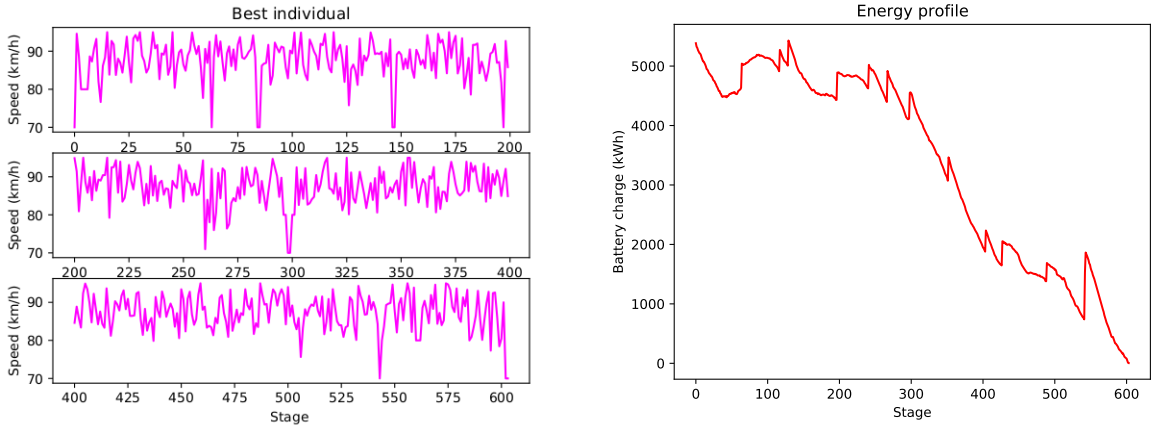

Figure 7.2: Best individual (left) and its energy profile (right)

7.2.1 Worst fitness

By looking at Figure 7.1, the first thing that can be noticed is the strange behavior of the worst fitness around generation 40. Here, the final time of the worst solution spikes up and starts to oscillate abruptly. At first it seemed a problem of the new implementation, but it was then observed that the old Matlab algorithm showed the same behaviour. The

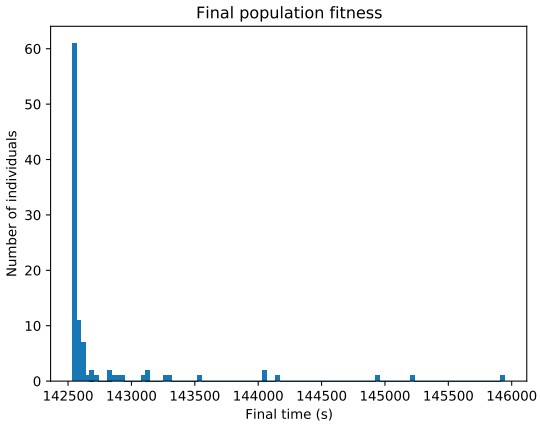

Figure 7.3: Distribution of fitness in final population

real explanation is quite simple considering what is expected to be a good solution. The ideal solution is the one in which the car drives at the maximum possible speed and uses all the available energy by the end of the race. The latter condition is well visualized in the energy profile of final best solution, in Figure 7.2. Around generation forty the population has converged to an optimum, and most of the solutions already take advantage of all the available energy. Random mutation of these individuals, that slightly increase the energy consumption, are heavily penalized for leading to negative energy values. This theory can appear to be wrong by taking into account that the mutation can both increase and reduce the energy consumption, and the effects should balance out. In reality, the use of Eq. 4.10 propagates the penalty of a negative energy level to the following stages. Even if the next stage has a positive energy balance the total battery level can remain below zero and the penalty still increases.

This was the main reason that led to the testing of the infinite norm (Eq. 4.11), which penalizes the solution only based on the lowest energy level that was reached. However, this could still lead to some drawbacks, because the search algorithm could exploit this fact to remain on a below zero energy level, which, even with the additional penalty, could lead to better total race times. This will be tested in the following sections, where additional evidence will be provided to support the hypothesis.

It is important to notice that this peculiarity of the problem is harmful to the search algorithm: once a local optimum is reached it is hard for the population to escape from it, thus leading to premature convergence. In fact, in correspondence of the spike in worst fitness, the best fitness becomes stable, meaning that there are no further improvements. This is also evident by looking at Figure 7.3. Here the fitness distribution of the final population is shown in a histogram composed of 100 bins. It is clear that the population converged to an optimum, and the few solutions that are not close to it are way worse (with up to more than an hour in delay). In addition, the percentage of solutions that don't share the same fitness with the best one is really close to the mutation probability. This means

that most (if not all) the mutated individuals end up being penalized.

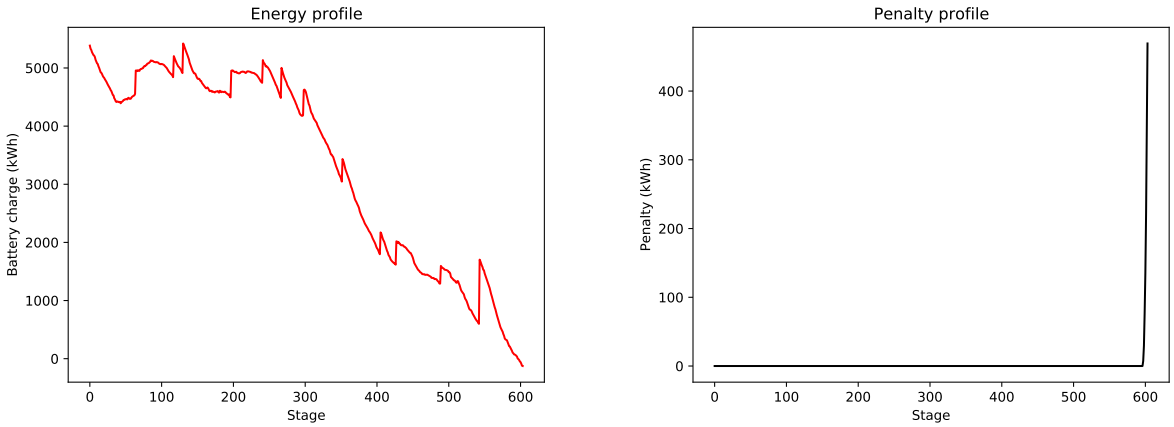

Figure 7.4: Energy (left) and penalty (right) profiles of the worst solution

This can also be seen in Figure 7.4, where the energy and penalty profiles of the worst solution at the end of another sample run are shown. As it can be observed, the only contribute to the penalty comes from the final stages. In fact, the speed increase with respect to the optimum only prevented the car from driving the final kilometers. It is also possible to notice that the energy profile is very similar to the one of the best solution of the previous run (Figure 7.2). It is also worth mentioning that the peaks shown in in the graphs are due to control stops and night charges.

7.2.2 Final solution

In Figure 7.2 the best solution obtained from the sample run is shown. It can be seen that the solution presents harsh speed differences between consecutive stages. As already mentioned before, in a real world scenario this is strictly avoided, due to the high inefficiency of frequent accelerations and decelerations. The proposed approach to solve this problem has already been introduced and the results will be presented if the following sections.

As a note, it is important to clarify that the big speed drops are due to the imposed speed limits, and don't reflect problems in the solutions.

7.3 Influence of the new modifications

This section tries to graphically evaluate how the unwanted behaviour, discussed in the previous section, can be influenced by parameters and penalty choices.

7.3.1 Worst fitness

To address the problem of the worst fitness abruptly increasing, it is possible to lower the mutation strength (σ) and reduce the energy-to-time penalty conversion factor (K). Two runs were performed: one by setting σ to 0.4 and another by reducing K to 1, while maintaining all the other setting as indicated in Table 7.2. Results can be seen in Figures 7.5 and 7.6.

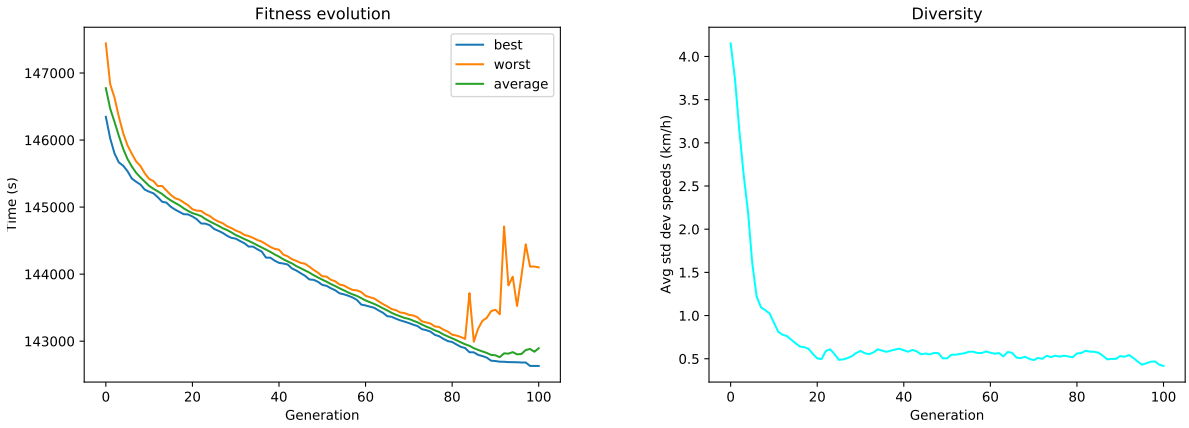

Figure 7.5: Run with reduced mutation

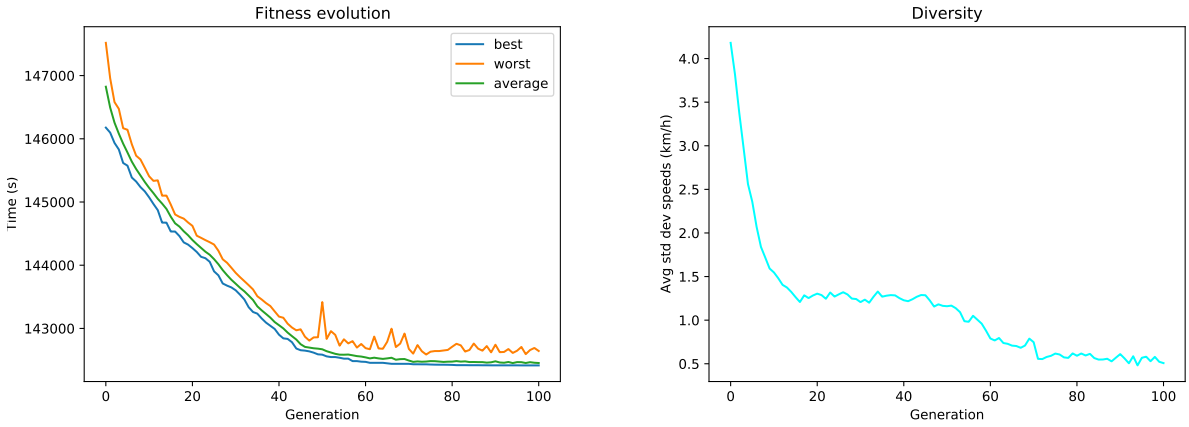

Figure 7.6: Run with reduced penalty

In the case of lower mutation, the effect is both reduced in magnitude and postponed to the very end of the run. This is because the population takes more time to reach the optimum value. However, this also increased the speed of convergence for diversity, which is not a desired effect. In the second case, where the applied penalty is reduced, it is possible to see that the oscillating behaviour is highly mitigated.

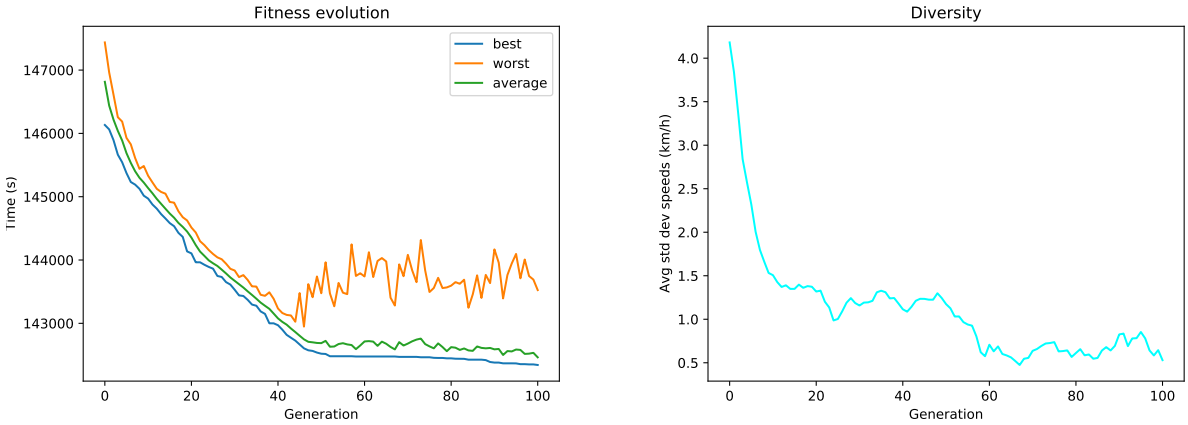

Figure 7.7: Infinite norm with high penalty

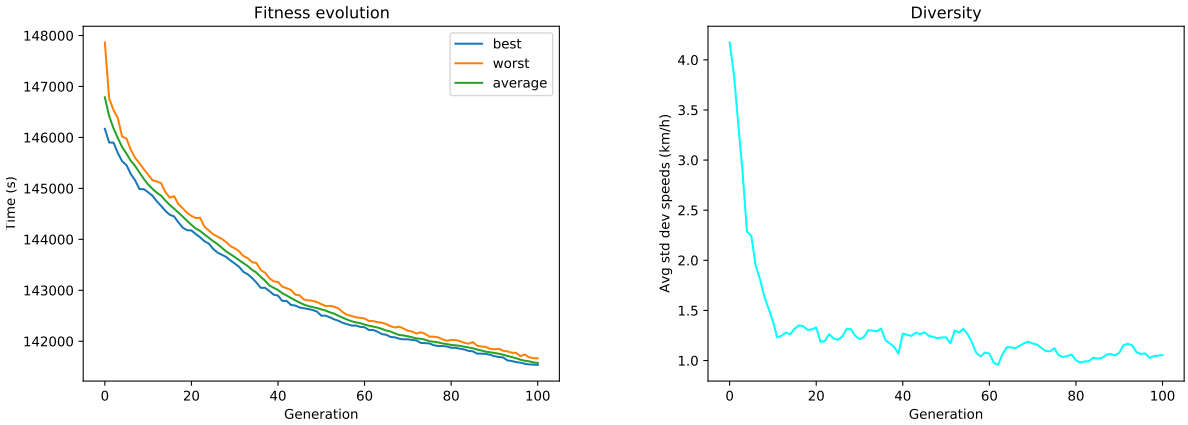

Figure 7.8: Infinite norm with low penatly

In Figures 7.7 and 7.8 two runs are shown where the infinite norm is used. The values for K are 16 and 1, as in the previous tests, while the other setting of the algorithm are the ones in tab 7.2. When using the same high penalty factor of the original norm (Figure 7.1, the infinite norm reduces the magnitude of the distortion effect. This also happens when comparing the low penalty factor, but in this case the best solution found by the algorithm also seems to improve. In addition to that, more diversity is found in the population at the end of the run.

However, this apparent improvement hides a significant problem, that is revealed by looking at the penalty profile of the best solution. In Figure 7.9 this is compared with one of the best solution obtained with the other norm. As already feared, the algorithm exploits the fact that only the highest penalty is considered to remain below zero energy for a lot of stages.

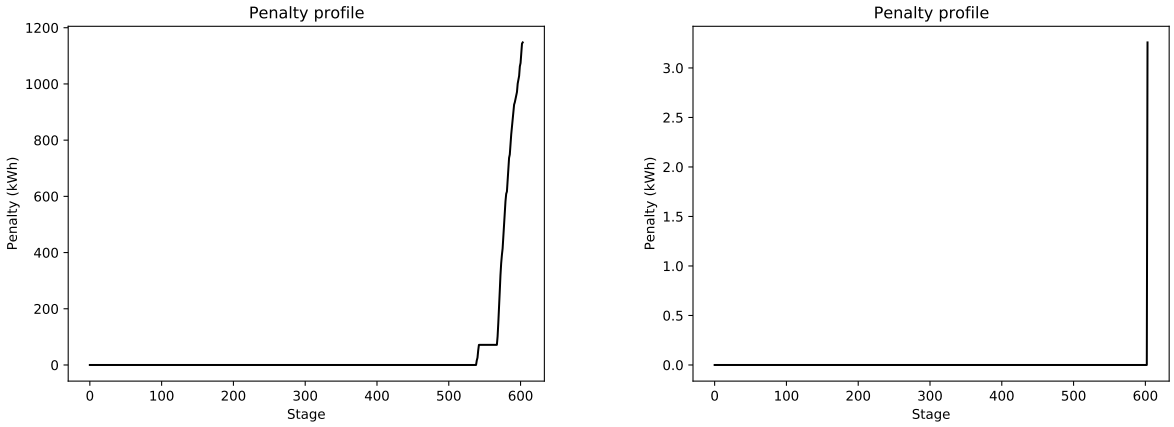

Figure 7.9: Best solution of infinite (left) and original norm (right)

7.3.2 Speed jumps

The speed jumps problem is addressed by introducing a energy expense for each time the car accelerates or decelerates. The effect of this approach is tested by visualizing the speed vectors of the best solutions produced by different values of K_{conv} (0.2, 0.5 and 1). Also in this case, the other settings of the algorithm are the ones in Table 7.2. Results can be seen in Figure 7.10. For the three solutions shown in the picture, the average speed differences among consecutive stages, for increasing values of K_{conv}, are 3.81, 2.44 and 2.02 km/h.

Both the pictures and the measurements show that the approach achieves its objective by limiting the unwanted behaviour. However, even if the magnitude of the jumps can be reduced, the solutions still present an oscillating behaviour. It would be better if speed changes would be achieved by smooth acceleration and deceleration among multiple stages. In addition to this, the values of K_{conv} that are required to achieve the effects shown in the images are way higher than the ones found in reality. A value of K_{conv} of 0.2 corresponds to an energy efficiency of the acceleration and deceleration process of about 80%, which is worse that what actually expected from the car.

7.4 Testing and tuning

In this section the testing and tuning of the algorithm is discussed. In the first subsections, the testing procedure is introduced, while the following ones the achieved results are shown.

7.4.1 Limiting the parameter space

The preliminary testing executed in the previous section showed that the algorithm is affected by premature convergence. Given this observation it is possible to focus the tuning phase in a more specific direction. Both of the size of the tournament selection and the percentage

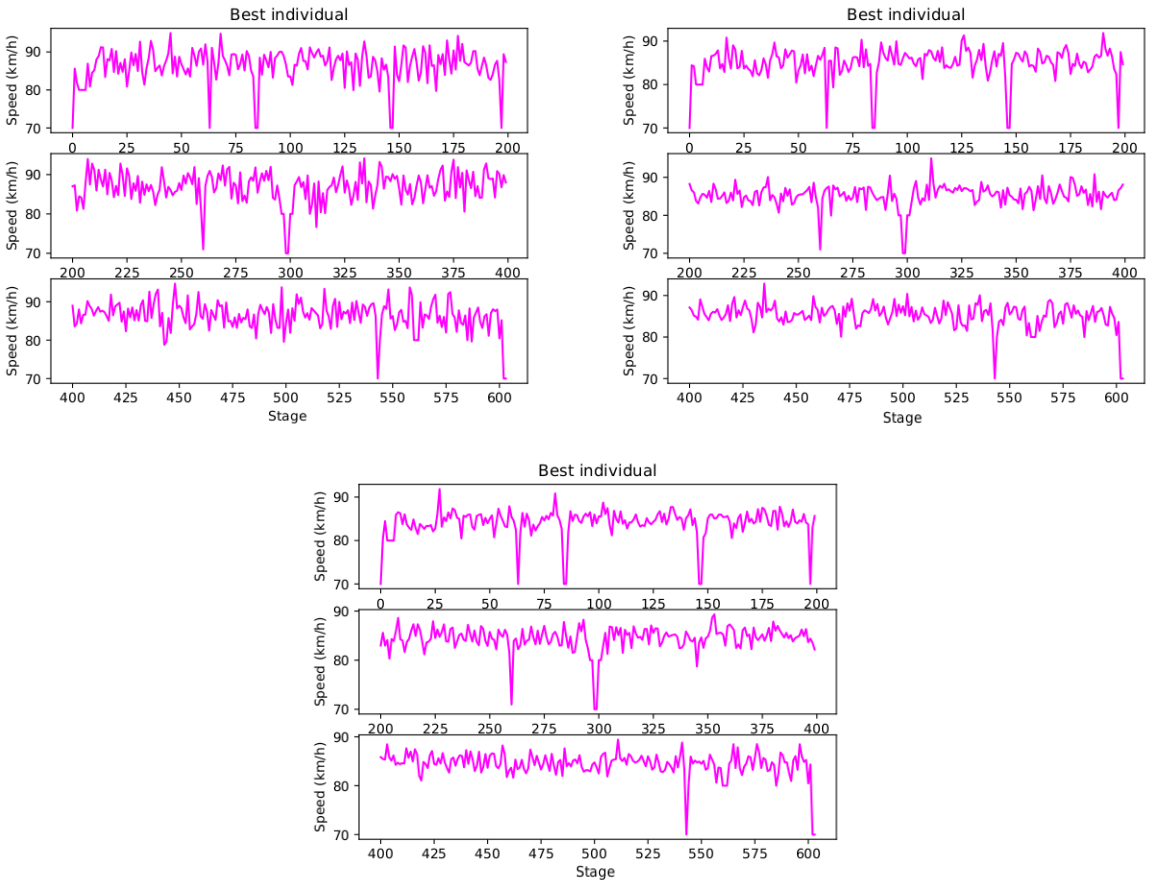

Figure 7.10: Speed jumps reduction for increasing values of K_{conv}

of elite individuals are already quite low. An increase of these value would only lead to more selection pressure which is clearly not wanted. For these reason the value of these two parameters are fixed to 6 and 0.01 across all tests.

In addition to that, the infinite norm is clearly not effective, because it lets the algorithm exploit the non-idealities of the model, by producing solutions where the car drives tens of stages with a negative battery charge. Therefore the infinite norm won't be considered in the tuning phase.

The last observation is about the speed jumps reduction in the solutions. Even if an higher value for K_{conv} produces smoothest solution, this makes the model diverge from the real problem. As a compromise, the value used across all tests is 0.2.

7.4.2 Testing and tuning procedure

The testing and tuning procedure is carried out according to the following stages:

- Energy penalty factor (K)
- Mutation rate and mutation σ
- Crossover rate and σ parameter in quasi blend crossover
- Crossover rate and number of split points in point crossover
- Crossover rate and α parameter in blend crossover
- Effect of increased population size and number of generations

Role	Operator
Parent Selection	Tournament
Reinsertion	Worst replaced
Penalty	Integral (Eq. 4.10)
Penalty Acceleration	Yes

Parameter	Value
Num generations	100
Population size	100
Elitism	0.01
Tournament size	6
Penalty factor (K)	16
Penalty Acceleration (K_{conv})	0.2

Table 7.3: Fixed parameters

In Table 7.3 are shown the parameters that are fixed for all the tests. The number of generations and the population size is varied only in the last phase. The best value found for a parameter in a previous stage will be used in the subsequent ones. The first two tests use quasi blend crossover with a probability of 0.65 and mutation strength of 0.4, which are the values found to be optimal in the previous work [10]. The same goes for the mutation probability and strength in the first test, where they are set to 0.35 and 1, respectively. All

combinations of parameters are tested 10 times to take into account the stochastic nature of the algorithm. The generation at which the best solution is found is the last generation that provided an improvement of at least one minute in the best solution. The generation of low diversity is the last generation in which the diversity was not greater than 10% of the final value. More information on each phase is provided in its dedicated section.

The data used for the test are the actual weather measurements of September 2019, since it is more relevant in defining the performance of the algorithm.

7.4.3 Penalty factor

In order to keep the model close to reality, the penalty factor (K) is limited in the (0, 1] range. The test varies it with step 0.1 across the interval. Results are shown in Table 7.4.

Penalty	Best (s)	Avg (s)	Div (km/h)	Best gen	Div gen	Time (s)
0.1	143750	143821	0.5595	47	81	0.3993
0.2	143851	143938	0.5673	54	77	0.3930
0.3	143916	144001	0.5531	46	82	0.3981
0.4	144029	144112	0.5562	59	92	0.4020
0.5	144082	144168	0.5364	47	85	0.4070
0.6	144122	144206	0.5327	56	86	0.4072
0.7	144123	144217	0.5688	38	71	0.3979
0.8	144176	144259	0.5538	44	83	0.4066
0.9	144243	144354	0.5667	43	75	0.4005
1	144256	144349	0.5507	43	75	0.3985

Table 7.4: Test penalty

By looking at the table, it does not appear to be significant differences in the diversity or convergence time. Both the best and the average get worse for increasing values of the penalty. However, this is due to the fact that the solutions are able to finish the last stages with negative energy levels, without being penalized enough to avoid this behavior. For this reason, the value chosen to be used in the following tests is 1. This is done in order to limit as much as possible the influence of this behavior, without introducing huge penalty coefficients that are completely unrealistic.

7.4.4 Mutation

The values of σ are initially varied in the [1, 3] km/h range, with a step of one. Mutation rate, instead, is tested in the [0.2, 1] range with step 0.2. These values will be refined with a second test where the areas of major interest are explored.

As it is show in Table 7.5, the best and average fitness seem to get worse when σ is higher than one. From what concerns the mutation rate, the value that gives both the best average and best fitness is 1.0.

It is also possible to notice that the diversity largely increases when higher values of both mutation rate and strength are used. This, however does not translates in a better algorithm, because, as it was shown in the previous section, the only individuals that are different in the final generation are the ones that are mutated. Having an higher percentage of them with an heavier mutation applied is not a symptom of a better population.

The computation time also increases with the mutation rate, since more operations have to be performed at each generation.

Another test is performed around the best values that were discovered, in order to refine them even more. The mutation rate is varied in [0.85, 1] with steps of 0.05. σ, instead, is evaluated in the interval [0.8, 1.2] with step 0.1. Results are shown in Table 7.6.

It can be seen that the improvement are very small. However, the fact that are consistent among each other in this smaller interval signifies that the combination found in the previous step wasn't a result of a random effect. The best values to be selected for mutation rate and strength, given the best solution found, are 1.0 and 0.8.

7.4.5 Crossover

The three crossover operators are tested with the best values for the mutation parameters that were found in the previous stage. For all the operators the crossover rate is varied in the range [0.2, 1] with step 0.2.

For quasi blend crossover the σ value is in the range [0.2, 0.8], with step 0.2. This range is chosen because it includes the optimum value found by the previous work (0.4) while not being to high to behave as an additional mutation. For point crossover, the number of split points is varied from 10 up to 50 % of the length of the individual (about 600 stages). This makes the value of this parameter to vary in the range [60, 300] with step 60. For what concern blend crossover, values of alpha are tested in the range [0.2, 1], with step 0.2. Values higher from 1 are not considered, otherwise the probability of ending up outside range defined by the speed values of the parents becomes too high. In fact, with alpha equal to 1, this probability is already 2/3. The results are shown in Tables 7.7, 7.8 and 7.9.

In the case of quasi blend crossover, the best solutions are found for a mutation rate of 0.2 and a crossover rate of 0.8 or 1. However, the problem is that these are two isolated cases. In general the performance seems to get worse for increasing values σ, except in a few cases. The opposite is found for the crossover ratio which does not provide any substantial improvements when changed. On the other hand, an higher crossover rate with an higher σ increases the diversity. However, this is similar to the case already discussed for mutation, in which the diversity is mostly given by all the bad solutions that don't satisfy the energy constraints.

Point crossover shows worse but more consistent solution across the range of values that the parameters assume. The only good aspect is the fact that final diversity is quite high, without the drawback of the previous operator. This is because point crossover does not introduce new speed values that may be too high to satisfy the energy constraints, since it limits itself at recombining the ones which already present in the parents.

Mut rate	σ (km/h)	Best (s)	Avg (s)	Div (km/h)	Best gen	Div gen	Time (s)
0.2	1	144205	144271	0.4075	46	91	0.3827
0.2	2	144560	144829	0.8178	29	78	0.3913
0.2	3	145148	145762	1.2872	31	74	0.3778
0.4	1	144135	144255	0.6006	48	76	0.3975
0.4	2	144453	145403	1.2116	43	57	0.4001
0.4	3	144951	147464	1.7708	53	62	0.4271
0.6	1	144050	144280	0.7744	53	56	0.4181
0.6	2	144351	145857	1.4957	56	69	0.4094
0.6	3	144744	150665	2.188	62	61	0.4196
0.8	1	143742	144087	1.1714	84	67	0.4406
0.8	2	144548	147612	2.2102	72	48	0.4336
0.8	3	145344	155435	3.0356	62	69	0.4425
1	1	143572	143954	1.8323	90	16	0.4472
1	2	145780	147498	3.3607	44	6	0.4609
1	3	146582	151781	4.5687	0	0	0.461

Table 7.5: Test mutation

Mut rate	σ (km/h)	Best (s)	Avg (s)	Div (km/h)	Best gen	Div gen	Time (s)
0.85	0.8	143478	143708	1.1227	90	50	0.4299
0.85	0.9	143569	143905	1.2049	89	62	0.4274
0.85	1	143696	144079	1.2931	83	61	0.4354
0.85	1.1	143778	144335	1.4333	84	44	0.4275
0.85	1.2	143927	144522	1.5370	78	60	0.4288
0.9	0.8	143457	143703	1.2534	88	36	0.4205
0.9	0.9	143614	143942	1.3468	76	37	0.4230
0.9	1	143699	144094	1.4892	78	39	0.4217
0.9	1.1	143860	144431	1.6076	81	52	0.4359
0.9	1.2	144056	144664	1.7462	74	40	0.4347
1	0.8	143347	143591	1.5251	90	14	0.4415
1	0.9	143431	143753	1.6906	91	15	0.4480
1	1	143644	144051	1.8088	80	22	0.4197
1	1.1	143789	144230	2.0249	85	12	0.4486
1	1.2	144106	144645	2.1674	76	19	0.4456

Table 7.6: Fine test mutation

Cross rate	σ (km/h)	Best (s)	Avg (s)	Div (km/h)	Best gen	Div gen	Time (s)
0.2	0.2	143834	144101	1.3484	94	28	0.4027
0.2	0.4	144364	144724	1.5013	79	57	0.4046
0.2	0.6	145718	146071	1.5329	86	87	0.4003
0.2	0.8	145931	146501	1.6637	78	66	0.4020
0.4	0.2	143418	143652	1.2674	90	22	0.4101
0.4	0.4	143764	144087	1.5221	84	35	0.4168
0.4	0.6	145574	146036	1.6776	84	82	0.4170
0.4	0.8	146057	146972	1.7473	70	65	0.4130
0.6	0.2	143240	143430	1.1913	91	21	0.4291
0.6	0.4	143417	143678	1.5088	88	15	0.4300
0.6	0.6	145545	146130	1.8449	84	64	0.4290
0.6	0.8	146284	147448	1.7912	63	72	0.4228
0.8	0.2	143100	143268	1.1464	92	10	0.4468
0.8	0.4	143192	143421	1.4881	88	14	0.4460
0.8	0.6	145308	145910	1.8763	79	84	0.4360
0.8	0.8	146457	148732	2.1100	24	60	0.4512
1	0.2	142989	143122	1.1038	91	6	0.4507
1	0.4	143124	143323	1.4511	90	12	0.4502
1	0.6	145139	145904	2.2396	82	65	0.4618
1	0.8	146527	151975	2.5857	15	55	0.4555

Table 7.7: Test quasi blend crossover

Cross rate	Num split	Best (s)	Avg (s)	Div (km/h)	Best gen	Div gen	Time (s)
0.2	60	145210	145594	1.6788	88	54	0.3958
0.2	120	145198	145540	1.5888	91	72	0.4045
0.2	180	145279	145636	1.6466	85	70	0.3948
0.2	240	145309	145680	1.6408	92	70	0.3928
0.2	300	145233	145584	1.6424	87	67	0.3956
0.4	60	144859	145195	1.6191	85	84	0.3977
0.4	120	144876	145247	1.6678	87	69	0.4004
0.4	180	144889	145227	1.6592	85	76	0.4269
0.4	240	144905	145234	1.6637	85	75	0.4026
0.4	300	144897	145234	1.7720	89	55	0.4089
0.6	60	144568	144958	1.7921	88	60	0.4040
0.6	120	144531	144892	1.7454	90	81	0.3887
0.6	180	144583	144948	1.8853	92	56	0.3952
0.6	240	144639	144998	1.7348	87	75	0.4139
0.6	300	144631	144978	1.8161	87	63	0.4158
0.8	60	144316	144697	1.9195	83	65	0.4051
0.8	120	144306	144686	1.8460	89	73	0.4188
0.8	180	144363	144723	1.7957	82	72	0.4282
0.8	240	144408	144797	1.8570	81	75	0.4377
0.8	300	144435	144828	1.8522	85	75	0.4577
1	60	144110	144525	1.9637	84	59	0.4068
1	120	144100	144566	1.9925	85	60	0.4218
1	180	144209	144639	2.0047	89	52	0.4372
1	240	144118	144552	1.9659	85	67	0.4545
1	300	144113	144544	1.9790	88	66	0.4544

Table 7.8: Test point crossover

Cross rate	Alpha	Best (s)	Avg (s)	Div (km/h)	Best gen	Div gen	Time (s)
0.2	0.2	144459	144801	1.5911	84	42	0.4153
0.2	0.4	145288	145687	1.5929	87	67	0.4171
0.2	0.6	145904	146337	1.6028	82	72	0.3876
0.2	0.8	145930	146411	1.4815	85	86	0.3903
0.2	1	146080	146770	1.6875	79	54	0.3869
0.4	0.2	143809	144152	1.5324	89	37	0.3937
0.4	0.4	144891	145287	1.6675	85	70	0.4060
0.4	0.6	146002	146537	1.6801	87	84	0.3997
0.4	0.8	146087	146851	1.7466	71	71	0.4062
0.4	1	146140	147652	1.8727	79	54	0.3989
0.6	0.2	143522	143807	1.4922	86	29	0.4125
0.6	0.4	144693	145094	1.7468	88	68	0.4042
0.6	0.6	145980	146626	1.7482	86	85	0.4122
0.6	0.8	146202	147425	1.8819	80	60	0.3958
0.6	1	146317	147930	1.9168	63	61	0.4204
0.8	0.2	143311	143535	1.4989	89	14	0.4080
0.8	0.4	144421	144847	1.7832	79	72	0.4113
0.8	0.6	146198	147085	1.9751	82	56	0.4186
0.8	0.8	146412	148374	2.0334	59	75	0.4138
0.8	1	146442	149595	2.0638	45	92	0.4098
1	0.2	143170	143385	1.4630	87	14	0.4253
1	0.4	144244	144740	1.8706	78	72	0.4269
1	0.6	146426	147803	2.2851	46	69	0.4296
1	0.8	146521	152570	2.6939	9	56	0.4237
1	1	146571	160105	3.0538	7	66	0.4295

Table 7.9: Test blend crossover

Cross rate	Alpha	Best (s)	Avg (s)	Div (km/h)	Best gen	Div gen	Time (s)
0.2	0.05	144087	144427	1.4819	91	23	0.4004
0.2	0.1	144219	144515	1.5034	89	31	0.3827
0.2	0.15	144266	144587	1.4698	82	68	0.3970
0.2	0.2	144421	144739	1.5408	81	56	0.3946
0.4	0.05	143601	143859	1.3967	90	22	0.3876
0.4	0.1	143676	143926	1.4249	88	27	0.4070
0.4	0.15	143728	143992	1.4904	90	27	0.4136
0.4	0.2	143823	144168	1.5292	87	41	0.3950
0.6	0.05	143320	143550	1.3474	89	13	0.4121
0.6	0.1	143343	143579	1.3934	90	14	0.4039
0.6	0.15	143422	143668	1.4590	89	14	0.4006
0.6	0.2	143502	143793	1.5076	85	33	0.4108
0.8	0.05	143168	143351	1.2835	88	11	0.4123
0.8	0.1	143176	143374	1.3379	87	12	0.4171
0.8	0.15	143236	143440	1.4039	90	13	0.4069
0.8	0.2	143268	143481	1.5059	91	14	0.4166
1	0.05	143028	143193	1.2324	91	9	0.4116
1	0.1	143026	143189	1.2948	89	10	0.4243
1	0.15	143055	143243	1.3626	91	13	0.4140
1	0.2	143171	143374	1.4644	89	14	0.4242

Table 7.10: Second test blend crossover

Normal blend crossover achieves worst results across the board. The only good solutions are found when the alpha parameter is the lowest possible (0.2). For this reason another test was performed, in which the alpha value is varied from 0.05 to 0.2 with step 0.05. Results are shown in Table 7.10. This makes that results better and more consistent.

As a conclusion, it is possible to see that the influence of crossover on the problem, whatever operator is used, is not very relevant. When bad values of specific crossover parameters don't lead to bad solutions, the influence of the crossover ratio is almost irrelevant, provided that it isn't too low (less or equal to 0.2). A minimum value is necessary, otherwise the algorithm becomes close to a random search in the solution space.

7.4.6 Number of generations and population size

In this section the performance provided by the algorithm are compared in term of number of individuals and population size. In order to conduct a fair experiment, the product of these two factor is kept constant. This allows to compare results that are obtained with similar computation effort. The best parameters found in the previous tests are used. For what concerns the crossover operator, blend crossover is used with a crossover ration of 1.0 and alpha equal to 0.1 (optimal values in previous tests).

Results are shown in Table 7.11. As it can be seen, the middle ground is the one that produces better results. If not many individuals are present the population loses the diversity necessary to find better solutions. This can be seen by dividing the generation number at which the best solution is found for the number of generations for which the algorithm is run. By increasing the number of generations and decreasing the number of individuals their ratio is reduced to roughly 50%. This means that half of the generations for which the algorithm is run are wasted, because they don't provide a better result. On the other end, if too many individual are present, with respect to the number of generations, the added diversity cannot be exploited to find better solutions. This can be seen in the first row: here the worst of all solutions is found, while still having the greatest diversity in the final generation.

Num gen	Num ind	Best (s)	Avg (s)	Div (km/h)	Best gen	Div gen	Time (s)
100	800	142666	142820	1.4011	83	13	3.053
200	400	142525	142701	1.3705	154	13	3.362
400	200	142539	142740	1.3116	240	14	3.474
800	100	142639	142859	1.2331	438	26	3.556

Table 7.11: Test number of generations and individuals

Chapter 8

Conclusions

The document shows that the main objectives have been accomplished. The software, rewritten in C++, has reduced running times from more that 1 minute to abut a second in the standard case of 100 individuals and 100 generations. Support for parallelization was also introduced, providing almost a perfect scaling in performance with the number of CPU cores. In addition to that, the code is now organized with the object oriented design, being more clear and easy to work on. All the most important functionalities are also accessible from Python, as if a native module was used.

From the algorithm perspective, more operators and penalty evaluations have been introduced and tested. The most significant improvement that is achieved is the reduction of the speed bumps in the final solutions. In addition to this, more insights on the effects that the various settings have on the behaviour of the algorithm have come to light.

A last note can be made on the improvements that can be done to this work. The first and more important aspect is to further increase the effort on the testing and tuning process. As a second objective, new approaches to the search process can be explored, such as the introduction of biogeography-based optimization [13].

Bibliography

[1] World Solar Challenge. *History*. URL: https://www.worldsolarchallenge.org/about_wsc/history.

[2] World Solar Challenge. *Website*. URL: https://www.worldsolarchallenge.org/.

[3] cppreference.com. *Parameter pack*. URL: https://en.cppreference.com/w/cpp/language/parameter_pack.

[4] cppreference.com. *RAII*. URL: https://en.cppreference.com/w/cpp/language/raii.

[5] cppreference.com. *std::bind*. URL: https://en.cppreference.com/w/cpp/utility/functional/bind.

[6] cppreference.com. *std::shared_ptr*. URL: https://en.cppreference.com/w/cpp/memory/shared_ptr.

[7] Google. *Map data*. 2020. URL: https://www.google.com/maps/d/.

[8] Wenzel Jakob. *pybind11 — Seamless operability between C++11 and Python*. URL: https://pybind11.readthedocs.io/en/stable/.

[9] Ananth Grama Vipin Kumar Anshul Gupta George Karypis. *Introdution to Parallel Computing*. Pearson Education, 2003.

[10] Robbe Keppens. "A genetic algorithm for optimizing velocity in the Bridgestone World Solar Challenge 2019". In: (2019).

[11] Kitware. *CMake*. URL: https://cmake.org/.

[12] NASA. *Dynamic Pressure*. URL: https://www.grc.nasa.gov/WWW/K-12/airplane/dynpress.html.

[13] Dan Simon. "Biogeography-Based Optimization". In: (2008).

[14] A.E. Eiben J.E. Smith. *Introduction to Evolutionary Computing*. Springer, 2015.

[15] Agoria Solar Team. *Bluepoint*. URL: https://www.solarteam.be/en/cars/bluepoint.

[16] Agoria Solar Team. *Bluepoint, team 8*. URL: https://www.solarteam.be/en/history/bluepointbrsmallsmall-team-8smallsmal.

[17] Wikpedia. *Factory method pattern*. URL: https://en.wikipedia.org/wiki/Factory_method_pattern.

[18] Wikpedia. *Solar Irradiance*. URL: https://en.wikipedia.org/wiki/Solar_irradiance.

[19] Wikpedia. *Template metaprogramming*. URL: https://en.wikipedia.org/wiki/Template_metaprogramming.