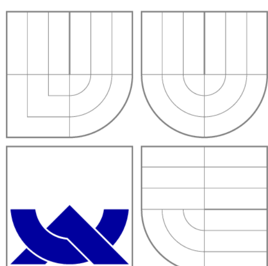
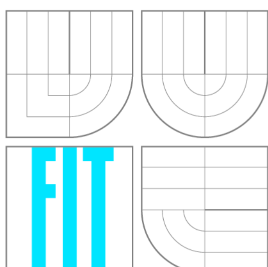


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AUTONOMNÍ ŘIDIČ ZÁVODNÍHO VOZU TORCS

AN AUTONOMOUS DRIVER OF A TORCS RACING CAR

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. LUKÁŠ BĚHAL
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ JAROŠ, Ph.D.
SUPERVISOR

BRNO 2012

Brno University of Technology - Faculty of Information Technology

Department of Computer Systems | Academic year 2011/2012

Master Thesis Specification

For: **Běhal Lukáš, Bc.**
Branch of study: Intelligent Systems
Title: **An Autonomous Driver of a TORCS Racing Car**
Category: Artificial Intelligence

Instructions for project work:

1. Study the issue of the autonomous racing car driver design. Examine features of the TORCS racing car simulator.
2. Study features and behaviour of common optimisation techniques. Focus deeply on bio-inspired optimisation techniques.
3. Propose a technique optimising the behaviour of the autonomous racing car driver.
4. Implement the proposed technique in the C++ or Java programming language.
5. Verify the behaviour of the proposed car driver on suitably chosen tracks. Compare and discuss its behaviour with a human driver.
6. Discuss the contribution of your work.

Basic references:

- According to the supervisor's instructions.

The Term Project discussion items:

- Items 1 - 3.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Master Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Jaroš Jiří, Ing., Ph.D.**, DCSY FIT BUT
Consultant: Pospíchal Petr, Ing., FIT VUT
Beginning of work: September 19, 2011
Date of delivery: May 23, 2012

L.S.

Zdeněk Kotásek
Associate Professor and Head of Department

Abstrakt

Tato práce popisuje simulátor TORCS a optimalizační algoritmy, jenž jsou využívány při tvorbě autonomních řidičů pro tento simulátor. Hlavním cílem je navržení nového autonomního řidiče, který se bude schopen s použitím přírodou inspirovaných optimalizačních technik vyrovnat již dříve navrženým řešením. Chování implementovaného řešení lze rozdělit do dvou hlavních částí, které jsou využívány v různých rozdílných etapách závodu. Zahřívací kolo je využito pro vytvoření modelu trati, ze kterého je posléze získána optimální trajektorie pomocí genetického algoritmu. Této trajektorie je potom využíváno v samotné kvalifikaci či závodě pro zajetí co nejrychlejšího kola. Z důvodu složitosti problému optimalizace celé trajektorie je nutno tuto trajektorii rozdělit na menší úseky nazývané segmenty, přičemž každý z nich je potom optimalizován odděleně. Jednotlivé optimalizované segmenty jsou následně spojeny dohromady, aby opět utvořily trajektorii pro celou trať. Protože některé přechody mezi segmenty mohou být nesouvislé, je zde znovu aplikován genetický algoritmus pro jejich vyhlazení. Během závodu je tato trajektorie následována, přičemž se z ní odvíjí i maximální možná rychlost v daném úseku. V práci jsme ukázali, že vzorkování trati s následnou optimalizací pomocí genetického algoritmu trvá pouze zlomek času vyhrazeného pro zahřívací kolo. Nejen díky tomuto se řešení jeví jako vhodné pro závody autonomních řidičů a může být dále rozšířeno.

Abstract

This work describes the TORCS simulator and optimization algorithms used in the field of autonomous driving competitions. The main purpose of this work is to design a new controller solution based on genetic algorithms. The controller's behavior can be divided into two main parts which are exploited during the distinct stages of the competition. The warm-up stage serves for the track model sampling and the race line optimization. The race stage logic then benefits from the data obtained in the warm-up stage. The track optimization is done by a Genetic algorithm while the track is divided into several segments optimized separately. A genetic algorithm is applied once again to the track trajectory to smooth out gaps caused by the segment composition. In this work was shown that the track sampling and race line optimization by a genetic algorithm can be done during the warm-up stage. This makes the controller suitable for an autonomous driver competitions.

Klíčová slova

TORCS, závody automobilů, autonomní řidič, přírodou inspirované optimalizační algoritmy, genetický algoritmus

Keywords

TORCS, car racing, autonomous driver, car controller, algorithms inspired by biology, genetic algorithm, racing line, the simulated car racing championship

Citace

Lukáš Běhal: An Autonomous Driver of a TORCS Racing Car, diplomová práce, Brno, FIT VUT v Brně, 2012

An Autonomous Driver of a TORCS Racing Car

Prohlášení

Prohlašuji, že jsem tuto semestrální práci vypracoval samostatně pod vedením pana Ing. Jiřího Jaroše Ph.D., přičemž jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Lukáš Běhal
May 21, 2012

Acknowledgment

I would like to thank to Ing. Jiří Jaroš Ph.D. for his lead, support and grammar correction provided while working on this thesis. The special thanks for grammar correction also deserves Audrey Wan. A big thanks belongs also to my parrents for their support. Finally, I would like to thank to all my friends for the amazing study years I had.

© Lukáš Běhal, 2012.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

1 Introduction **3**

1.1 Project outline . . . 3

2 TORCS **4**

2.1 The Simulated Car Racing Championship . . . 4

2.2 A new client development . . . 6

2.3 Sensors and Actuators . . . 6

2.4 Configuring TORCS Race . . . 7

3 Proposed techniques **10**

3.1 Fuzzy architecture . . . 11

3.2 Artificial Neural Networks . . . 12

3.3 Imitation . . . 12

3.4 Behaviour-Based Artificial Intelligence . . . 13

3.5 Genetic algorithms . . . 13

4 Genetic algorithms **15**

4.1 Representation of individuals . . . 15

4.2 Population . . . 16

4.3 Fitness evaluation . . . 17

4.4 Selection . . . 17

4.5 Crossover . . . 19

4.6 Mutation . . . 21

5 Racing line **24**

6 Controller design **25**

6.1 Track model representation . . . 25

6.2 Warm-up stage . . . 25

6.3 Race stage . . . 31

6.4 Interaction between warm-up stage and race stage . . . 32

7 Implementation **33**

7.1 Application structure . . . 33

7.2 Stage work-flow . . . 35

7.3 Warm-up stage . . . 36

7.4 Race stage . . . 38

7.5 Files specification . . . 38

7.6 Track visualization 39

8 Experiments **40**
8.1 Track sampling 40
8.2 Segment creation 41
8.3 Segment optimization 42
8.4 Track optimization 44
8.5 Comparison with other solutions 47

9 Conclusion **48**
9.1 Own contribution 48
9.2 Future work 49

A Fitness function **53**

B Matlab functions **55**
B.1 comparePaths function 55
B.2 getVectorComponentsFromFile function 55

C Optimized track figures **57**

Chapter 1

Introduction

Autonomous driving is a very interesting research topic supported by many vehicle manufacturers or the well-known Defense Advanced Research Projects Agency (DARPA) organization. Reducing fuel consumption/Efficient use of fuel, improving car safety and driver comfort are all aspects which could benefit from research on autonomous driving. The first long distance challenge, the DARPA Grand Challenge [1] has proved that real driver-less cars can race in a demanding desert environment. Car simulators became popular due to the fact that the cost of buying and modifying vehicles is unbearable for most researchers.

The Open Racing Car Simulator (TORCS) is a very realistic car racing simulator with a sophisticated physics engine used for many autonomous car racing competition challenges every year. This fact, combined with the large game community and possibility of controller comparison makes TORCS the most used simulator in the field of autonomous driving.

A countless number of optimization techniques were used for a driver behaviour adaptation however the genetic algorithms were the most used ones. Genetic algorithms (GAs) are able to solve many problems from various domains and this have been proven by their application to problems in business, engineering or science [2].

The performance of each driver relies on many miscellaneous factors but no driver will achieve the best results without following the optimal trajectory called racing line [3].

1.1 Project outline

This work is divided into nine main chapters and each of these chapters describe a distinct topic. Chapter 2 is dedicated to the TORCS simulator. Chapter 3 discusses currently proposed optimization techniques used in the field of autonomous driving. Genetic algorithms are summarized in chapter 4. Chapter 5 describes the racing line problem. The controller design is introduced in chapter 6 while chapter 7 discusses the implementation details of the proposed solution. Chapter 8 is dedicated to the experiments which were done on the designed controller. The last chapter, chapter 9 summarizes the contribution of this thesis and proposes future work.

Chapter 2

TORCS

The Open Racing Car Simulator [4] is an open source multi platform racing simulator which runs on various platforms. (e.g. GNU/Linux, FreeBSD, Mac OS X and Microsoft Windows) The simulator is written in C++ and licensed under the GNU GPL[1]. TORCS was created by Eric Espié and Christophe Guionneau and present code contributions are mostly made by Bernhard Wymann and Christos Dimitrakakis.

There are over 50 different cars and more than 20 cars can be driven by a keyboard, mouse, joystick or a steering wheel. The simulation includes a simple damage model, reliable physics system and many car properties which can change the car's behaviour such as springs, dampers, stiffness, ground effect, spoilers etc. A player can choose from various types of races from the practice session up to the championship. The game also offers multi-player by using a split screen mode. The online mode is the next development goal of interest. TORCS also has a big community of users and developers who keep software bug free and updated.

Due to these characteristics the simulator has became popular for several competitions held by congresses like IEEE Congress on Evolutionary computation (CEC), Computational Intelligence and Games Symposium (CIG), IEEE World Congress on Computation Intelligence (WCCI) and Genetic and Evolutionary Computation Conference (GECCO).

2.1 The Simulated Car Racing Championship

The Simulated Car Racing Championship is a joint event of three simulated car racing competitions held in 2011. A description of the championship, including the rules and regulations can be found at http://cig.dei.polimi.it/. [5]

The goal of the championship is to design a controller that would be able to finish a set of unknown tracks firstly alone in a certain time limit and then against other controllers.

Since The Open Racing Car Simulator comes as a stand-alone application with the build-in bots a new patch modifying TORCS to client-server architecture and grating simple development of controllers has been created.

The server module is a component of TORCS which provides communication to the remote controller. Each controller has to implement an API for the sensors and actuator models. Communication between the server and the client modules is performed through UDP connections. With every game tic the server sends current sensory inputs to each bot and waits for an answer with new actions from a client. If no information is received the

[1]GNU General Public License

last performed action is used. Two client modules have been provided. One is written in C++, the second one in Java.

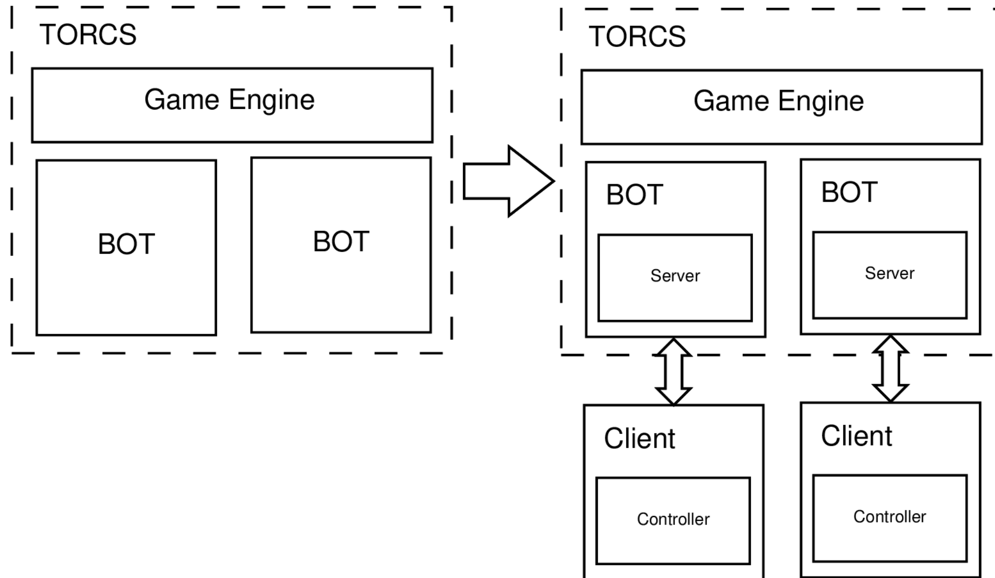


Figure 2.1: The architecture of the competition software. Inspired by [5]

2.1.1 Competition Rules

The championship consists of several races on different tracks divided into legs. Each race consists of three stages:

1. the warm up
2. the qualification
3. the race

During the warm-up stage, each controller races alone. This provides a controller with an opportunity to gather information about the track and to tune-up its behaviour.

The qualification is used for the selection of best controllers for the final race. Each driver can race for the same time period on each track and the eight controllers which reach the longest distance will take part in the final races.

During the final races, the eight most successful controllers race together. Races are done on three different tracks while eight runs are done on each of the tracks. Drivers are scored by the Formula1 system[2] and the driver performing the fastest lap in the race will get two extra points. First starting grid positions will be formed by qualification rankings and each subsequent race grid position will be shifted by one so that every driver will start from all grid positions.

More information can be found at [6].

[2] 1st controller gets 10 points, 2nd 8 points, 3rd 6 points, 4th 5 points, 5th 4 points, ...

2.2 A new client development

As I mentioned in section 2.1 two clients had been proposed to facilitate driver development. Let's have a look at the C++ one. Each driver should inherit from the `BaseDriver` class containing following virtual methods which need to be implemented:

- `void init(float *angles)` this method is called before the beginning of the race and it serves as an initial custom configuration of the track sensors. All of the 19 range finders need to be set in the parameter.
- `string drive(string sensors)` is a method used for controlling your driver during the race. This method receives all sensor values by the `sensor` parameter and returns a string of effectors representing the actions taken. For further details see tables 2.1, 2.2 and 2.3.
- `void onShutdown()` method called at the end of the race.
- `void onRestart()` method called when the race is restarted.

Both methods `onShutdown` and `onRestart()` should be used to close opened files, save data to disk and free allocated memory.

To identify the current stage of the race, the class attributes, `stage` and `trackName` are used. The current stage can be one of *warm-up*, *qualifying*, *race* or *unknown*. By these attributes, we can choose different car behaviour and adopt different strategies in different stages of the competition.

To compile your client you need to uncomment two commented lines at the beginning of `client.cpp` and modify them to use your driver class. Once the client is compiled without errors it can be run like a console application by:

```
$ ./client host:<ip> port:<p> id:<client-id> maxEpisodes:<me> \
  maxSteps:<ms> track:<trackname> stage:<s>
```

where `<ip>` is the IP address of the TORCS competition server (by default `localhost`), `<p>` is the port on which the server is listening(the default is 3001), `<client-id>` stands for your bot ID (by default *championship2011*), `<me>` is a maximum of learning episodes, `<ms>` is maximum number of control steps in each episode, `<trackname>` is the name of the track where the bot will race(by default unknown), `<s>` represents the stage of the competition (0 is *Warm-up*, 1 is *Qualifying*, 2 is *Race* and 3 is *Unknown*). All parameters are optional.

2.3 Sensors and Actuators

One of the reasons for the creation the competition software was to separate the game engine and the bots. Therefore no knowledge about the core engine is needed to develop a driver. For this reason the sensors and actuators layer was created.

Each controller perceives the racing environment through many sensors which contains information about the race, car status, position on the track and opponents. All of the sensors are listed in the tables 2.1 and 2.2. The distances between cars from opponent sensors are computed „as the crow flies" even if the path crosses the edges of the track.

The actuators which are used by a bot to control the car in the race are described in table 2.3. The typical set of effectors includes steering, the gas pedal, the brake pedal and the gearbox.

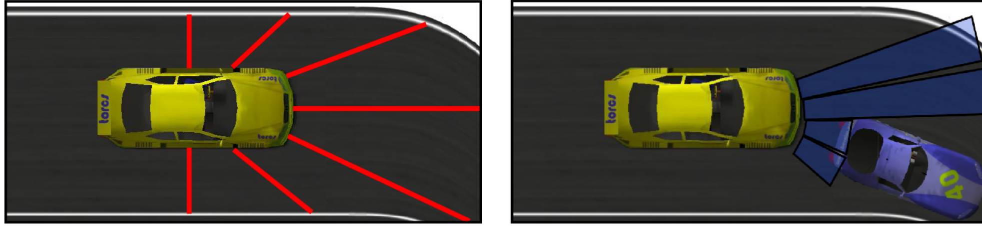


Figure 2.2: Edge and opponent sensors. [7]

2.4 Configuring TORCS Race

There are several options how to run the simulator with external drivers. The easiest way to configure the race is via the GUI by selecting `Race`→`Quick Race`→`Configure Race` where you can select the track and the bots as well. By selecting one of the `championship2011server` in the driver selection you can add one of the competition servers which provides connection to a programmed bot.

The race can also be configured through configuration files. The setting are stored in `practice.xml` and `quickrace.xml` files. These files are located in `config\raceman\` file structure of the installed simulator. Tracks can be selected in the *Tracks* section inside the XML file and drivers in the *Drivers* section.

TORCS can also be run in text mode which could be useful for running a selection of experiments where no GUI is needed. This can be done with a `-T` parameter in the command line:

```
$ torcs -T
```

The competition software can also be run with several other parameters to make it possible to conduct very long experiments. Fuel and damage should be disabled to decrease noise in the evaluation process because these two attributes change car behaviour and performance. The maximum lap time should be removed as well to let the car continue for as long as it needs. To disable these features TORCS needs to be run with following arguments:

```
$ torcs -nofuel -nodamage -nolaptime
```

Each bot has **10ms** by default to perform an action and reply to the competition server. If no response is received by the server, the last performed action will be chosen. This time constraint can be changed, which can help during debugging your driver. The desired timeout is measured in nanoseconds.

```
$ torcs -t <timeout>
```

By default the sensors return precise values but during the competition these sensors will be affected by noise to emulate the real world more precisely. The way in which the sensors will be affected is described for each sensor in tables 2.1 and 2.2. To enable noisy range finders, the following argument should be added:

```
$ torcs -noisy
```

Table 2.1: Description of the available sensors (part I). Ranges are reported with their unit of measure (where defined). [5]

Name	Range (unit)	Description
angle	$[-\pi, +\pi]$ (rad)	Angle between the car direction and the direction of the track axis.
curLapTime	$[0, +\infty)$ (s)	Time elapsed during current lap.
damage	$[0, +\infty)$ (point)	Current damage of the car (the higher the value is the higher the damage is).
distFromStart	$[0, +\infty)$ (m)	Distance of the car from the start line along the track line.
distRaced	$[0, +\infty)$ (m)	Distance covered by the car from the beginning of the race
focus	$[0, 200]$ (m)	Vector of 5 range finder sensors: each sensor returns the distance between the track edge and the car within a range of 200 meters. When noisy option is enabled, sensors are affected by normal noises with a standard deviation equal to the 1% of sensors range. The sensors sample, with a resolution of one degree, a five degree space along a specific direction provided by the client (the direction is defined with the focus command and must be in the range $[-\pi/2, +\pi/2]$ w.r.t. the car axis). Focus sensors are not always available: they can be used only once per second of the simulated time. When the car is outside of the track (i.e., pos is less than -1 or greater than 1), the focus direction is outside the allowed range ($[-\pi/2, +\pi/2]$) or the sensors has been already used once in the last second, the returned values are not reliable (typically -1 is returned).
fuel	$[0, +\infty)$ (l)	Current fuel level.
gear	$\{-1, 0, 1, \ldots, 7\}$	Current gear: -1 is reverse, 0 is neutral and the gear from 1 to 7.
lastLapTime	$[0, +\infty)$ (s)	Time to complete the last lap.
opponents	$[0, 200]$ (m)	Vector of 36 opponent sensors: each sensor covers a span of $\pi/18$ (10 degrees) within a range of 200 meters and returns the distance of the closest opponent in the covered area. When noisy option is enabled, sensors are affected by i.i.d. normal noises with a standard deviation equal to the 2% of sensors range. The 36 sensors cover all the space around the car, spanning clockwise from $+\pi$ up to $-\pi$ with respect to the car axis.
speedX	$(-\infty, +\infty)$ (km/h)	Speed of the car along the longitudinal axis of the car.
speedY	$(-\infty, +\infty)$ (km/h)	Speed of the car along the transverse axis of the car.
speedZ	$(-\infty, +\infty)$ (km/h)	Speed of the car along the Z axis of the car.

Table 2.2: Description of the available sensors (part II). Ranges are reported with their unit of measure (where defined). [5]

racePos	$1, 2 \ldots, N$	Position in the race with respect to other cars.
rpm	$[2000, 7000]$ (rpm)	Number of revolutions per minute of the car engine.
track	$[0, 200]$ (m)	Vector of 19 range finder sensors: each sensors returns the distance between the track edge and the car within a range of 200 meters. When noisy option is enabled, sensors are affected by normal noise with a standard deviation equal to the 10% of sensors range. By default, the sensors sample the space in front of the car every 10 degrees, spanning clockwise from $+\pi/2$ up to $-\pi/2$ with respect to the car axis. However, the configuration of the range finder sensors (i.e., the angle w.r.t. to the car axis) before the beginning of each race. When the car is outside of the track (i.e., pos is less than -1 or greater than 1), the returned values are not reliable.
trackPos	$(-\infty, +\infty)$	Distance between the car and the track axis. The value is normalized w.r.t to the track width: it is 0 when car is on the axis, -1 when the car is on the right edge of the track and +1 when it is on the left edge of the car. Values greater than 1 or smaller than -1 mean that the car is outside of the track.
wheelSpinVel	$[0, +\infty)$ (rad/s)	Vector of 4 sensors representing the rotation speed of wheels.
z	$(-\infty, +\infty)$ (m)	Distance of the car mass center from the surface of the track along the Z axis.

Table 2.3: Description of the available effectors. [5]

Name	**Range (unit)**	**Description**
accel	$[0, 1]$	Virtual gas pedal (0 means no gas, 1 full gas).
brake	$[0, 1]$	Virtual brake pedal (0 means no brake, 1 full brake).
clutch	$[0, 1]$	Virtual clutch pedal (0 means no clutch, 1 full clutch).
gear	$\{-1, 0, 1, \ldots, 7\}$	Gear value.
steering	$[-1, 1]$	Steering value: -1 and +1 means respectively full right and left, that corresponds to an angle of 0.785398 rad.
focus	$[-90, 90]$	Focus direction (see the focus sensors in Table 1) in degrees.
meta	$0, 1$	This is meta-control command: 0 do nothing, 1 ask competition server to restart the race.

Chapter 3

Proposed techniques

Many solutions for simulated car racing have been proposed in the past several years. The most important ones will be discussed in this chapter.

Listed below are three controllers which have taken part at *The Simulated Car Racing Championship* in the last two years. These specifications are adopted from the event presentations [8] and [7].

Autopia

- Fuzzy Architecture based on three basic modules for gear, steering and speed control, optimized with a genetic algorithm
- Learning in the warm-up stage
  - Maintain a vector with as many real values as track length in meters
  - Vector initialized to 1.0
  - If the vehicle goes out of the track or suffers damage then multiply vector positions from 250 meters before the current position by 0.95.
  - During the race the vector is multiplied by F to make the driver more cautious as a function of the damage, where $F = 1 - 0.02 \cdot round\left(\frac{damage}{1000}\right)$.

Jorge Muñoz

- Build a model of the track during the warm-up stage.
- Two neural networks predict the trajectory using the track model. Two neural networks predict the target speed given the model of the track and the current car position
- These four neural networks are trained with back propagation using data retrieved from a human player
- Learning during the warm-up
  - The car remembers where it has gone out of the track or drives far from the trajectory and in the next laps goes slower at those points
  - The car remembers where it has followed the trajectory perfectly and tries to go faster in the next laps.

Ready2Win

- Modular architecture
  - Driving module
  - Overtaking module
  - Recovery module
  - ABS module
- Track learning during the warm-up:
  - First lap is driven slow to identify turns (start, end, entry position, curvature) and learn the track model
  - Other laps have speed adaptation

3.1 Fuzzy architecture

Fuzzy logic has been used several times in the field of autonomous driving and fuzzy based controllers belong to the most successful ones. Both winners of the 2009 and 2010 CIG Car Racing Competition used fuzzy controllers for their drivers.

Fuzzy logic deals with the reasoning which does approximation of definite values for false or true by a value of truth in the range from 0 to 1. This can be easily applied for example for steering where range from 0 to 1 represents measure of steer while 0 means full steer to left, 1 full steer to right and 0.5 is to go straight.

A typical fuzzy controller consists of four main components: knowledge base, fuzzifier, inference engine and defuzzifier. *The knowledge base* contains the fuzzy sets which are sets of membership functions associated with each input/output of the system and the fuzzy rules which represents rules like the „IF condition THEN action“. *The fuzzifier* converts real input values into fuzzy values. These values are processed by *the interference engine* by interference with the fuzzy sets. *The defuzzifier* then turns the output fuzzy sets to real values. The whole process is displayed at 3.1.

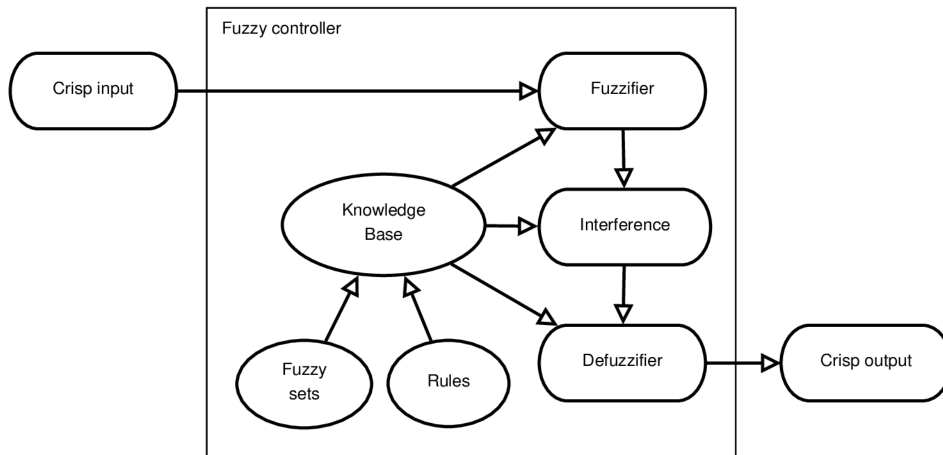


Figure 3.1: The architecture of a fuzzy controller [9]

Typically more fuzzy modules are used in the controller to obtain better results i.e. one fuzzy module for steering and another one for speed like in [9]. Fuzzy logic has been used

together with the Genetic optimization algorithm which has led to better results [10]. Also the opponent overtaking solution has been proposed [11].

3.2 Artificial Neural Networks

Artificial neural networks are commonly used in the field of racing games. A neural network is a mathematical model inspired by biological neural networks and it is created by interconnected artificial neurons. A typical neural network consists of several layers where the first one is the input layer, the last one is the output layer and the layers in between are called hidden layers. See figure 3.2.

The most important character of neural networks is the possibility of learning, which is obtained by changing the structure of the neural network during the learning phase. This is done by updating the weights of neurons.

Neural networks can be used to control one or more modules together which contributes to greater applicability. In [12], neural networks have been used as an effective solution of the controller where neural networks had been trained on human data. This data has been reduced by removing of the first lap from each race to reduce the noise. Two neural networks were used for trajectory and two other for velocity prediction.

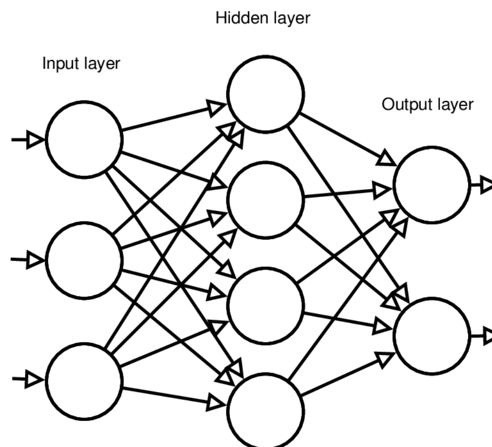


Figure 3.2: The architecture of an Artificial Neural Network

Generally, neural networks are able to outperform other controller architectures [13] and have a potential to learn and improve themselves. The biggest disadvantage of this approach is the initial duration of needed training.

3.3 Imitation

Imitating human behaviour is quite a new interesting research topic which has even been used several times in commercial games. One of them is Forza Motorsport for the Microsoft XBox.

The main idea of this approach is to adopt behaviour of a given controller which can also be a human player. In Forza Motorsport, this approach was used for creating bots called Drivatars[1] which were acting like players. They were able to react to the player's improve-

[1] http://research.microsoft.com/en-us/projects/drivatar/forza.aspx

ments and imitate his driving skills during the races which had led to more competitive bots. Due to this fact the game became more entertaining.

The disadvantage of this approach is that the controller created by the imitation will never be better than the original controller. However, there have been some quite successful experiments.

In [14], the imitation has been used to *Human player*, *NEAT*[2] *controller* and *Hand coded controller*. For learning purposes the Artificial Neural Network with a back propagation algorithm was used. From the results, the most complicated controller to learn is the human player because it does not act the same way in the same circumstances and it makes a lot of mistakes which had to solve. This leads to unexpected behaviour and the ANN is not able to learn anything useful. Non-player controllers are then easier to imitate. Another conclusion is that combining two types of controllers does not work because the controller learns mixed features from both of them but none of these features is learned properly.

3.4 Behaviour-Based Artificial Intelligence

Behaviour-Based Artificial Intelligence is a technique known for over two decades, however, only a few solutions for racing games have been implemented so far.

The system is divided into many modular components which are relatively simple and robust. Each of these components is able to react to conditions of the environment, therefore none of them has access to another's internal representation. All components are organized into layers in a hierarchy where a higher layer may subsume a lower layer by affecting its inputs and outputs, also called as subsumption architecture [15].

However, this approach needs more manual work caused by non-ability to learn new actions by itself, it has great potential mainly because of its simplicity and robustness. Each module can also be optimized for example by a genetic algorithm.

More details can be found at [16].

3.5 Genetic algorithms

Genetic algorithms [2] are very powerful techniques used at many different artificial intelligence fields while in the autonomous driving problem these algorithms are mainly used for controller optimization.

Generally genetic algorithms program the car setting to be compiled into strings of 0s and 1s while these strings can then be modified by mutation or crossover. Mutation maps to a random change of several 0s or 1s to the opposite value, crossover maps to a recombination of bits between two strings. Each solution is evaluated by the fitness function. At the beginning of the algorithm, the initial population is created and mutation and crossover are applied. Best solutions of this population evaluated by the fitness function are then used in the next iteration. The algorithm typically ends by a given number of populations or time expiration.

In [17], genetic algorithms have been used for evolving fuzzy sets in a fuzzy based controller. GAs have also been used to evolve a controller in [18].

[2]The NeuroEvolution of Augmenting Topologies - its a method for evolving artificial neural networks with a genetic algorithm. The idea is to start with a small, simple networks and let them increase to become more complex.

In comparison with the Artificial Neural Networks, GAs are more usable in the short term as they learn faster than ANNs but in the long term ANNs overcome GAs. The most common use of genetic algorithms is for them to be used in conjunction with another approach.

Chapter 4

Genetic algorithms

Genetic algorithms are a frequently used technique of the evolutionary algorithms. GAs have been applied to various problems and became popular mostly due to the complexity of the problem they are able to solve. This chapter provides a brief introduction to the Genetic algorithms which will be frequently discussed in the following chapters. Most of the information was taken from [2], [19] and [20].

The principle of Genetic algorithms is based on evolution of the population while searching for the best individual fit to the given conditions. The key part of the algorithm is the proper representation of individuals. At the beginning, the initial population with hight diversity needs to be created. The fitness of every individual is evaluated and the best solutions selected and modified by mutation or recombination to form a new population. The algorithm terminates either if a sufficient individual has been found or the given number of populations has been reached. The general work-flow chart is illustrated in figure 4.1.

4.1 Representation of individuals

Each candidate solution needs to be encoded as a string of values (genes) referred as a chromosome or genotype. Due to the problem of diversity, chromosomes can have various types of representation. This part covers the most common ones.

4.1.1 Binary representation

A chromosome is represented by a set of binary values. The special type of binary representation is the Gray encoding which can produce more superior results than the classic binary representation. Gray code is characterized by the Hamming distance of 1 between adjacent values. An example of a binary represented chromosome:

$$c_1 = 0|1|0|0|1$$

4.1.2 Integer representation

An individual is represented by a string of integer numbers. This type is used to represent problems which have natural integer variables like image processing parameters or categorical values from a fixed set. An example of an integer representation:

$$c_2 = 4|0|15|3|8$$

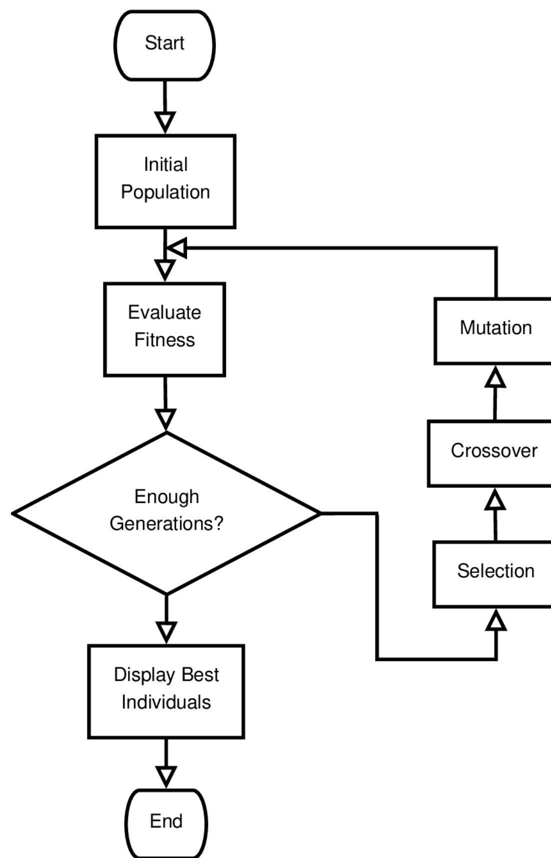


Figure 4.1: Genetic algorithm scheme

4.1.3 Floating point representation

An individual is represented by a string of floating point numbers. Floating point representation can be useful when we need to represent real valued problems like continuous parameter optimization. An example of a floating point represented chromosome:

$$c_3 = 4.5|0.2|3.3|0|2.1$$

4.1.4 Permutation representation

The permutation representation is typical for ordering/sequencing problems e.g. the Hamiltonian path or the TSP problem. Ordinary, if a problem has n variables then it is represented as a string of n integers where each occurs exactly once. This encoding also needs special recombination operators e.g. Partially Mapped Crossover (PMX) or Cycle crossover [21].

4.2 Population

Population is a set of chromosomes which encodes the current set of candidate solutions in one iteration of the algorithm. The next population is formed by selecting a couple of candidate solutions from the current population.

Much emphasis needs to be placed at the initial population creation. As all other populations are based on the initial one, a large diversity of individuals is required. Usually, the random individuals are created to include the whole range of possible solutions. Sometimes, the solutions may be seeded in the area of the expected optima.

The key parameter is a population size which depends on the nature of the problem. Ideally, the algorithm outcome will be better with a higher number of populations. On the other hand, the computational time grows with quantity of solutions. Ordinarily, we are trying to find a trade-off between number of populations and computational time.

4.3 Fitness evaluation

All individuals of the current population are being scored during the fitness evaluation. This is done by a fitness function which evaluates a quality of a given solution. Generally, better solutions have a higher value.

A well designed fitness function is a crucial step for good algorithm outcomes. It is also the most time consuming part of GAs [22].

4.4 Selection

Selection is one of the basic operators which is applied on each population. By selection, a set of individuals from the current population is chosen and inserted into a mating pool. Individuals from the mating pool are used to generate new offspring which will form a new generation.

As a new generation is based on the selected individuals, it is desirable that the mating pool consists of "good" individuals. Usually, the better individuals (with higher fitness function) are favored.

The *Selection pressure* determines a degree to which the better individuals are favored. The higher the selection pressure, the better individuals are chosen, which leads to a faster convergence of the algorithm. On the other hand, the chance of the convergence into an incorrect (suboptimal) solution is increased. However, if the selection pressure is too low, the algorithm execution time will be unnecessarily longer [23].

There are many different methods to select individuals for the next generation. Some of the most popular ones are described further.

4.4.1 Fitness proportional selection

Fitness proportional selection (FPS) is the selection technique based only on the individual's fitness function. Each member of the population has a certain probability to become a parent:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{4.1}$$

where N is the number of individuals in the population and f_i is the fitness of the ith member. This technique has the following disadvantages:

1. One individual with very high fitness can rapidly decrease the probability for the other individuals.

2. The selection pressure may be lost at the end of runs when fitnesses are similar.

Roulette-wheel selection

Fitness proportional selection is sometimes called the Roulette-wheel selection due to the similarity to a roulette game. Each individual gets a part of the wheel proportionally corresponding to its fitness value. A random number $n \in [0, f_{sum}]$ is generated which determines the chosen individual. This represents the spinning of the roulette.

Analogically, the Rank-based selection (see 4.4.2) can be associated with Roulette-wheel selection. The only difference is the use of selection probability instead of the fitness value.

4.4.2 Rank-based selection

Rank-based selection improves the Fitness proportional selection (Section 4.4.1) where the selection is based on the rank instead of the fitness. All individuals of the population need to be sorted by fitness value. The selection probability is then allocated according to the individual rank. In comparison with the FPS, the influence of the individual with rapidly high fitness is reduced. Also the selection pressure is kept up at the end of runs when the fitness variance is low. The biggest disadvantage of Rank-based selection is potentially time-consuming sorting which is needed for rank assignment. The mapping of the rank to the selection probability can be done either by linear or exponential function.

The linear ranking is based on the following equation:

$$p_i = \frac{1}{N}\left(\eta^- + (\eta^+ - \eta^-)\frac{i-1}{N-1}\right): \qquad i \in \{1, \dots N\} \tag{4.2}$$

where $\frac{\eta^-}{N}$ is the probability of the worst individual, $\frac{\eta^+}{N}$ is the probability of the best individual to be selected and N is the number of the individuals in the population.

The exponential ranking selection differs from the linear ranking selection by weighting probabilities of the ranked individuals:

$$p_i = \frac{c-1}{c^N - 1}c^{N-1}; \qquad i \in \{1, \dots N\} \tag{4.3}$$

where base of the exponent is a parameter $0 < c < 1$. The closer c is to 1, the lower the “exponentiality” of the selection method [24].

The exponential ranking should be preferred if individuals above the average fitness values need to be selected more frequently.

4.4.3 Tournament selection

Tournament selection of the size s provides the selection pressure by holding a tournament among s competitors. The winner of the tournament is the individual with the highest fitness value. The winner is then inserted into the mating pool. The mating pool, being comprised of tournament winners, has a higher average fitness than the average fitness value of the population. Due to this, the selection pressure is being increased by each generation which drives the GA to improve the fitness faster. Increased selection pressure can also be provided by increasing the tournament size s as the winner from the larger tournament will have a higher fitness value [23]. The tournament selection is illustrated in figure 4.2.

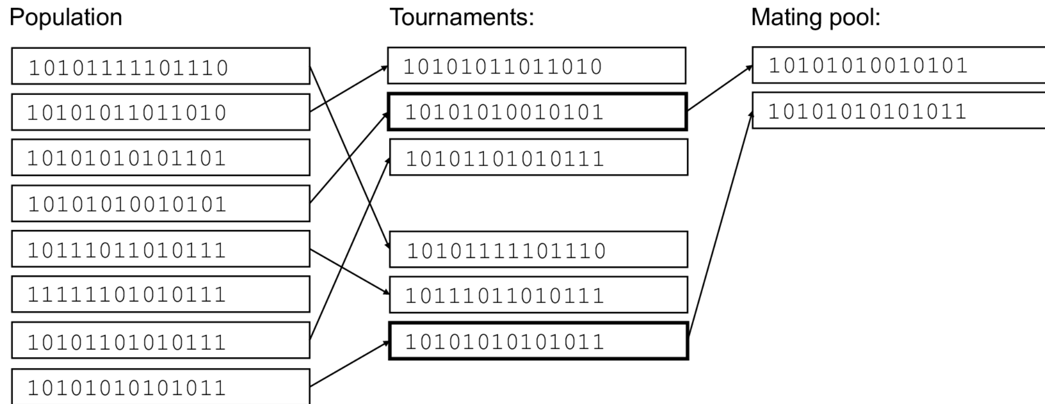


Figure 4.2: Tournament selection

4.4.4 Elitism

Elitism is a special case of the selection operator which ensures that the present best solution will be preserved in the next population. Such individuals can be lost if they are not selected or destroyed by mutation or crossover. It has been discovered that elitism significantly improves the GAs performance [19].

4.5 Crossover

Once the individuals have been selected into a mating pool, the crossover operator can be applied. The main idea of the crossover is a combination of two good solutions from the mating pool into an even better solution. Since we do not know which features make the individuals good, the recombination of genes is done randomly. Apparently, this may also lead to worse solutions by combining poor features of the chromosomes. The typical types of crossover have been introduced:

4.5.1 One-point crossover

One-point crossover is the simplest crossover operator. A random number smaller then the length of the chromosome is generated and the parts of two parents behind the crossover position are exchanged.

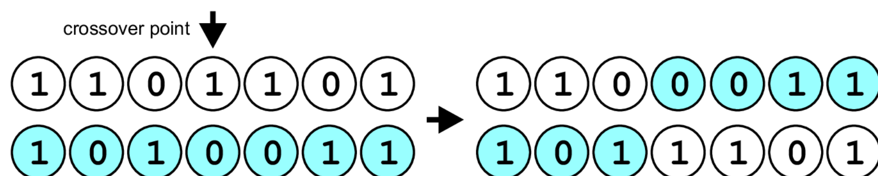


Figure 4.3: One-point crossover

4.5.2 N-point crossover

N-point crossover follows the same idea as the one-point crossover with a difference that n crossover point are generated instead of one. The chromosomes are exchanged at each crossover point (see 4.4).

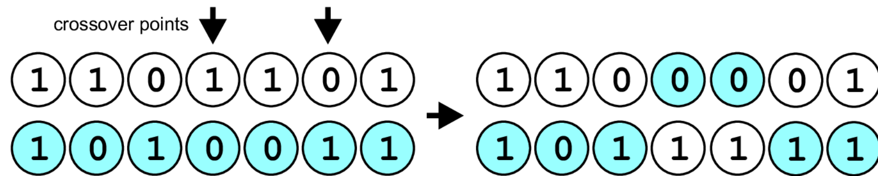


Figure 4.4: Two-point crossover

4.5.3 Uniform crossover

Uniform crossover combines the chromosomes by the binary crossover mask of the same length as the chromosomes. The mask is generated by a uniform distribution over $[0, 1]$. Although the uniform crossover have been refused as a correct crossover for a long time it can introduce the demanded diversity into the population.

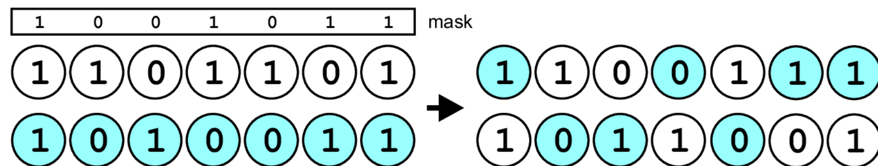


Figure 4.5: Uniform crossover

4.5.4 Permutation crossover

Due to the fact that applying ordinary crossover operators as described before would lead to invalid solutions, the special crossover operators need to be introduced.

Partially mapped crossover

The partially mapped crossover (PMX) is a method which at first generates two crossover points and exchanges the substrings between them. The rest of the chromosome is then amended by the mapping based on the corresponding genes of the chromosomes.

Cycle crossover

The cycle crossover process can be divided into several parts. At the beginning, the cycles need to be identified. Each gene of the new child keeps the same position as in the parent, however, the genes for each cycle are taken alternately from both parents (see figure 4.6). Cycles are created in the following way:

1. start with the first gene of the first parent
2. look at the same position of the second parent
3. go to the position with the same gene in the first parent
4. add this gene to the cycle
5. repeat step 2 until you get back to the first gene of first parent

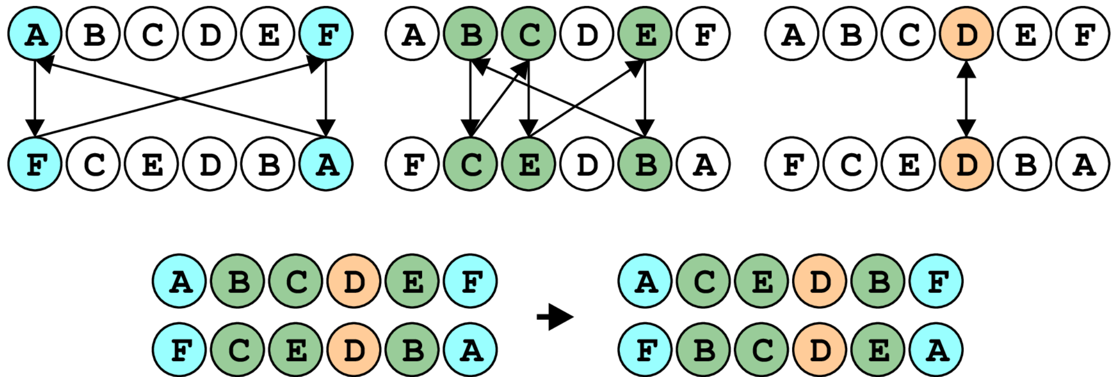


Figure 4.6: Cycle crossover (Inspired by [21])

4.6 Mutation

The main characteristic of the mutation is that it operates on only one individual. The purpose of the mutation is to preserve and introduce diversity into chromosomes. This prevents the permanent fixation at any particular locus and thus playing more of a background role. Mutation was the only source of variation in some early versions of evolutionary programming or evolution strategies [19].

In comparison with crossover in terms of disruption, mutation is more powerful than crossover, although it lacks ability to preserve alleles common to individuals. However, in terms of construction, crossover is more powerful than mutation [25].

The mutation operator is applied with a probability specified by the mutation rate parameter. A good choice of the mutation rate parameter belongs to one of the crucial steps of GA setting. High mutation rate will lead to losing good solutions and on the other hand a low mutation rate will decrease diversity.

The basic types of mutation for the ordinary representations are introduced below.

4.6.1 Mutation for binary representations

Only one mutation operator exists for binary represented problems and this flips the gene at randomly generated positions (Figure 4.7).

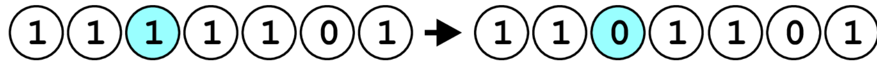


Figure 4.7: Binary mutation

4.6.2 Mutation for integer representations

Mutation of integer represented individuals can be done either by *Random resetting* or by *Creep mutation*. Both type of mutations are displayed in the figure 4.8.

Random resetting turns the value of each gene with probability p_m into a new random value chosen from the set of permissible values.

Creep mutation changes each gene with probability p_m by increasing it by a small value which is usually obtained from symmetric distribution with the center at 0.

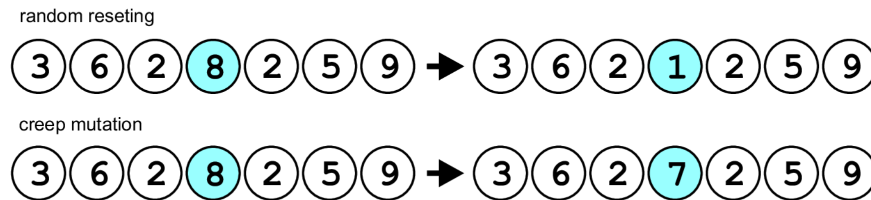


Figure 4.8: Integer mutations

4.6.3 Mutation for floating-point representations

As well as integer representations, floating-point representation can be mutated in two ways (Figure 4.9):

Uniform mutation assigns to a randomly chosen gene a new value chosen from the set of permissible values.

Nonuniform mutation does a small change of the each gene where the addition value is generated by the Gaussian distribution.

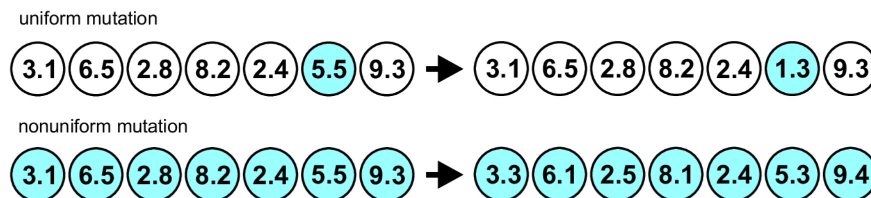


Figure 4.9: Floating-point mutations

4.6.4 Mutation for permutation representations

As it was discussed in section 4.1.4, the permutation representation doesn't allow occurrence of two genes with the same value. Due to this fact, the classic mutation operators cannot be used. All described mutation types are displayed in the figure 4.10.

Insert mutation selects the genes at random positions. Move the second after the first and shifts the rest along to accommodate.

Swap mutation picks two genes at random positions and swap their positions.

Inversion mutation selects two alleges at random position and inverts the whole substring.

Scramble mutation selects two genes at random positions and mixes all substring genes.

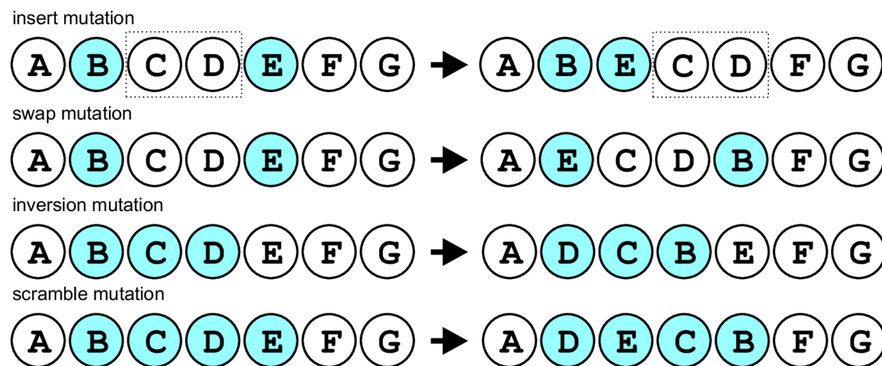


Figure 4.10: Permutation mutations

Chapter 5

Racing line

The performance of each driver depends on many miscellaneous factors. The sense of throttling, judgment of speed, smooth shifting or well-timed braking are some of them but no driver will achieve the best results without following the optimal racing line. But what is the “best” trajectory the driver can follow? In terms of racing the “best” means the trajectory driven in the least time at the greatest average speed [26] [27] [3].

Firstly, let’s focus on commonly known path optimization problems such as the *Shortest path* and the *Minimal curvature path*:

Shortest path is a trajectory with the least distance possible (the red line in the figure 5.1). In this case, the driven distance will be the shortest possible, however the average speed will not be high due to sharp turns on the way.

Minimal curvature path is defined as a trajectory with the least curvature possible (the blue line in the figure 5.1). Curvature is a major factor in the determination of the optimal trajectory [28].

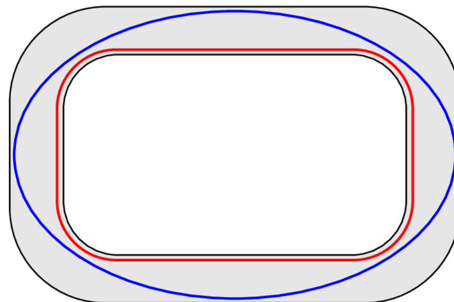


Figure 5.1: Comparison between Shortest path (red line) and Minimal curvature path (blue line)

Neither the shortest path nor the minimal curvature path by itself usually comprise the optimal racing line. As it was mentioned in [29], the optimal racing line is a trade-off between the shortest path and the minimal curvature path:

$$r = (1 - \epsilon) \cdot sp + \epsilon \cdot mcp \tag{5.1}$$

Chapter 6

Controller design

The aim of this chapter is to describe the design of the proposed controller. On the note of the already used techniques discussed in chapter 3, the controller based on the Genetic algorithms summarized in the chapter 4, was created.

Most of the controller logics can be divided into two main parts: the warm-up logic and the race logic. The proposed controller follows the same idea while the warm-up stage is used for track model creation and racing line (see chapter 5) optimization. The optimized race line is being used during the race stage.

6.1 Track model representation

The track model representation needs to be introduced first as it is a fundamental element of both stages. Almost each of the autonomous drivers racing in the TORCS nowadays, use a certain model of a track. As a human gets better while driving through the same segment of the track again, the performance of an autonomous driver can be improved too. This is caused by having more details about the track. With more details, better path planning and car control can be done which lead to better lap times. In comparison with a human player, all information about the track can be stored by the autonomous driver at once.

A controller proposed by [28] uses a Cartesian coordinates to preserve the track model. In [30], the track is represented by a set of segments (start, end, direction) of turns or narrow parts. Each segment is moreover divided into smaller parts (start, end, type) which specify the segments more precisely. The solution proposed in [12] does a track sampling by storing information about the distance from the start line of the car, the angle with the track, the distance of the car to the center of the track, the distance of the car with the track edges and the distance the rear wheels cover between two game ticks.

The proposed controller uses a track model represented by vectors. Each vector determines a relative direction from the current point and is represented by a direction and a magnitude in the polar coordinates.

An example of four vectors $v_1 = (2.5, 0)$, $v_2 = (2.5, 1)$, $v_3 = (2.5, 0)$ and $v_4 = (2.5, -0.4)$ displayed by *Progressive vector diagram* [31] via MATLAB [32] can be found in figure 6.1.

6.2 Warm-up stage

The warm-up stage is usually used for the track exploration and optimization. Each controller has the $100,000$ game ticks [7] (approximately 30 minutes of real time) to learn the

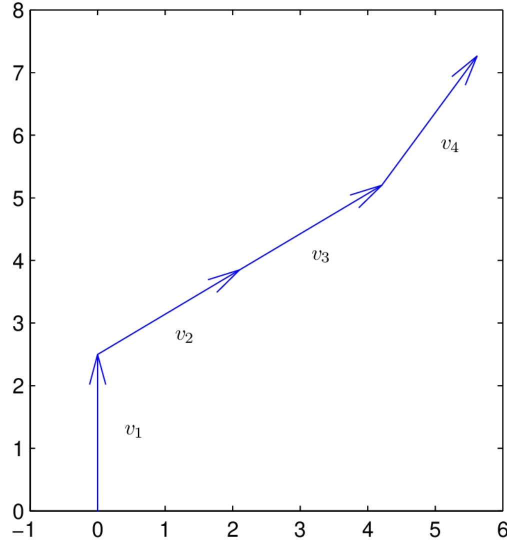


Figure 6.1: Track model representation

track, optimize car settings and enhance performance. Behaviour of the proposed controller can be separated into several parts which are illustrated in figure 6.2 and described further in the following section. The main purpose of this stage is to build the track model which will be optimized by the Genetic algorithm in the rest of the time. Design of the warm-up stage was inspired by [28] and [12].

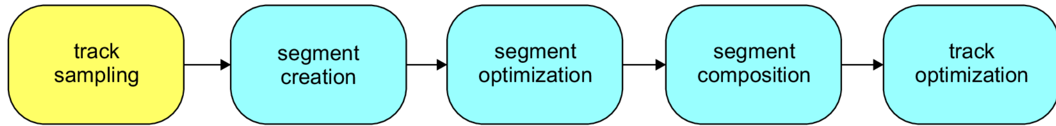


Figure 6.2: Warm-up stage process

6.2.1 Track sampling

As it was discussed in the section 6.1, the track is represented by the set of vectors while each set reflects the next relative direction. The track model is being build by the constant speed in the middle of the road. The constant speed without any high acceleration or sharp braking is important otherwise the wrong track model will be built.

Once the complete lap is sampled, the car stops and the optimal race line evaluation begins. Let's have a look at the controller logic which leads to the smooth track sampling at first.

The constant speed is reached by a simple logic which takes into account only the current and the desired speed.

$$acceleration = \frac{desiredSpeed - actualSpeed}{desiredSpeed} \tag{6.1}$$

Steering control is based on the angle which the car contains with the track axis. This will force the car to drive along the track. The track position is important as well to keep the car in the middle of the road.

$$steer = angle - trackPosition \tag{6.2}$$

Track sampling is done by two sensors s_1 and s_2 containing angle β. By using the values of these sensors we are able to compute the direction of the next vector α (see Equations 6.3 and 6.4). One sample is taken after driven distance which equals to the l value on the narrow road. This can be simply computed by the track width (Equation 6.5) and represents the magnitude of the vector. The sampling process is displayed in the figure 6.3.

$$l = \sqrt{{s_1}^2 + {s_2}^2 - 2 \cdot s_1 \cdot s_2 \cdot \cos\beta} \tag{6.3}$$

$$\alpha = \arcsin\frac{s_2 \cdot \sin\beta}{l} \tag{6.4}$$

$$magnitude = s_1 \cdot \tan\beta \tag{6.5}$$

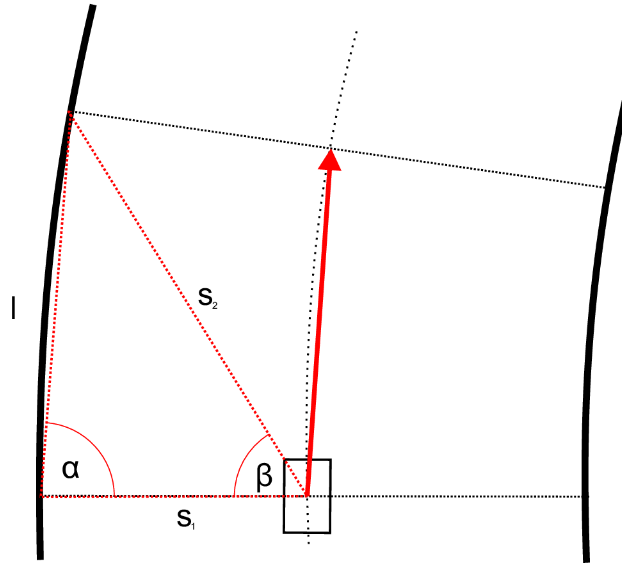


Figure 6.3: Sampling scheme

In comparison with [12] the track model is computed on the fly and the only stored information are the vectors. Also the car width doesn't need to be known.

6.2.2 Segment creation

Segment creation is the first stage of the optimization process. Due to the problem complexity, the track needs to be divided into smaller parts and optimized separately. The division is done in the middle of each narrow part of the track which leads to the separation of each turn on the track.

Since the track is already sampled the Segment creation process is very straight forward. Suppose that α is a minimum angle of a turn. A position p_1 of the first angle bigger than

α represents the beginning of a turn. The next first angle lower than α represents the end of a actual turn. After all the position of p_2 of the next first angle bigger than α is the position of the beginning of the next turn. The position $p = \frac{p_2 - p_1}{2}$ is the new segment division point.

6.2.3 Segment optimization

The most important and complicated part of the warm-up stage is the Segment optimization. Every segment is optimized separately by a Genetic algorithm.

Problem encoding

Issued from the track representation described in 6.1, the way of the alternate path representation needs to be introduced. Supposing the sampled trajectory as a center line of the road, the trajectory can be displaced for a certain length to the left or to the right side limited by the $\frac{width}{2}$ for each side. Let's say that we have n positions on the track for each vector starting point where the alternate paths can go through. Whole segments composed of j vectors will need $j+1$ points due to the last vector of segment.

To make it clear, see figure 6.6. The red arrows represent the original vectors. Every vector has 9 points the new path can go through (represented by the black dots). The green arrow illustrates the new path represented by values -1 for the beginning of the first vector and -2 for the end of the vector.

A segment which consists of j vectors will then be represented by string of $j+1$ integers. Assuming that we have n positions, each gene range will be $\left(-\frac{n}{2}, \frac{n}{2}\right)$ where 0 represents a point in the middle of the road.

Initial seed

Initial seed is done at the beginning of the algorithm to create the initial population. Based on the knowledge of the optimal racing line (chapter 5), the most outer orbit of a turn would be a good initial solution as it partly corresponds to the optimal path by the entrance and exit of the turn. The fact that only a part of the initial population can be seeded this way to keep diversity in the population, a seed which includes these solutions is introduced.

For each individual a random number $r \in \left(-\frac{n}{2}, \frac{n}{2}\right)$ will be generated, where n is a number of division points for each vector. All genes of the individual will be seeded by the r value. This ensures that all initial individuals will consist of a continuous path and the most outer orbit will probably be included into the initial population.

Crossover

A mean crossover operator will be used for a recombination of individuals as it's able to combine two solutions with preservation of a continuous path. A one-point crossover is not suitable as it may produce gaps during the recombination of two various individuals. An offspring is created by averaging the parents alleles while the son is rounded down, and daughter rounded up. A proposed crossover is displayed in figure 6.4.

Mutation

A mutation based on the *Creep mutation* (see 4.6.2) is introduced to bring diversity into solutions. A random number of genes in a row starting at the random position in the

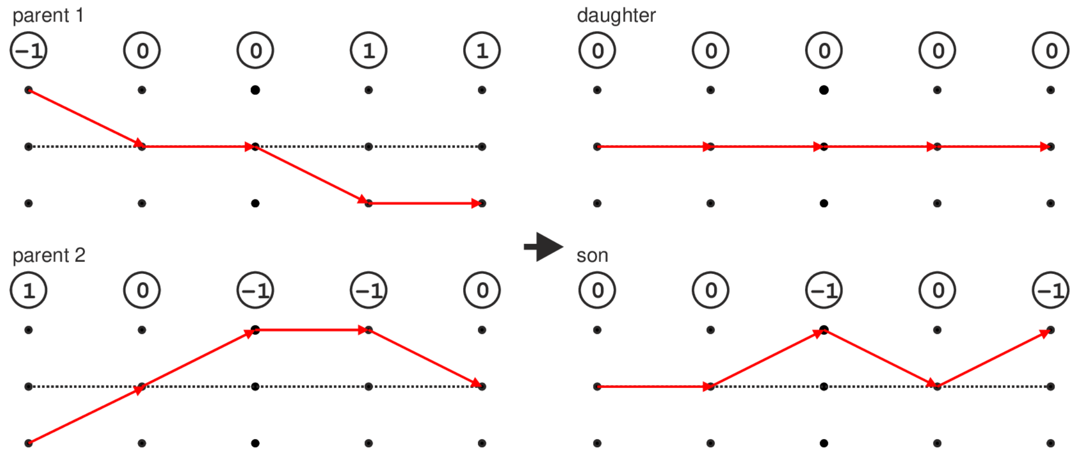


Figure 6.4: Segment crossover

chromosome are increased/decreased by a small value. The advantage of this operator is that the mutated chromosome preserves the original continuity. An example of this mutation is displayed in figure 6.5, where chosen genes are decreased by 1.

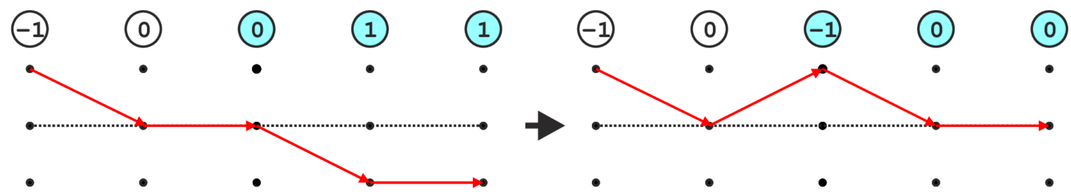


Figure 6.5: Segment mutation

Fitness function

Fitness function usually belongs to the key part of Genetic algorithm. As this approach uses the racing path as an evaluative criterion, it is fundamental to calculate a new path from the track vectors and a chromosome. Considering figure 6.6 which displays a relation between two segments and the detail of this figure at 6.7, a new angle β and new length can be computed by following computational procedure:

A dimensions of the basic triangle (formed by the red arrow) displayed in figure 6.7 needs to be computed first:

$$\gamma = \pi - (\frac{\pi}{2} - \alpha) \tag{6.6}$$

$$x = \frac{l}{\tan \alpha} \tag{6.7}$$

$$y = \frac{l}{\sin \alpha} \tag{6.8}$$

where l is the actual vector length and α is the direction of the next vector. By these values, $trackWidth$ and the current chromosome o we are able to compute the length ln and angle β:

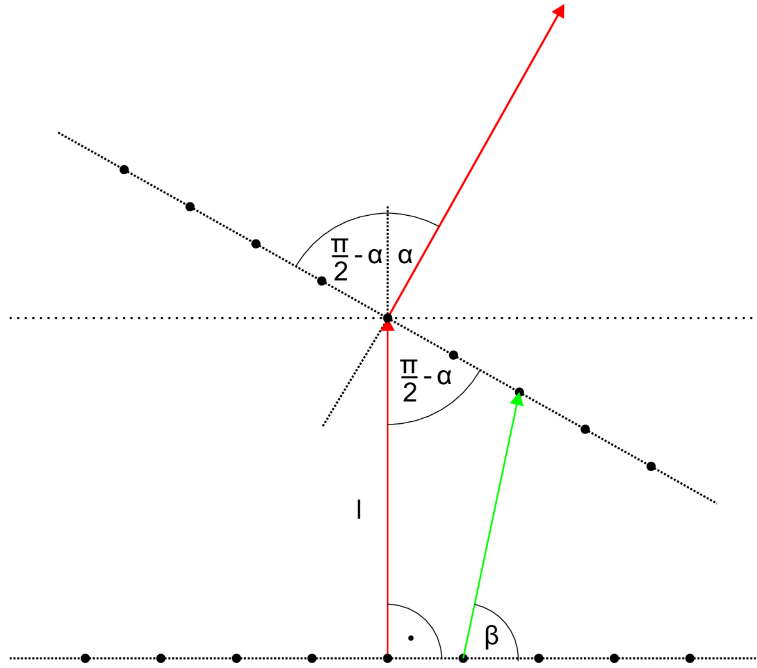


Figure 6.6: A relation between two vectors used during the fitness evaluation

$$xn = x + o_i \cdot trackWidth \tag{6.9}$$
$$yn = y + o_{i+1} \cdot trackWidth \tag{6.10}$$
$$ln = \sqrt{xn^2 + yn^2 - 2 \cdot xn \cdot yn \cdot \cos\alpha} \tag{6.11}$$
$$\beta = \arccos \frac{nl^2 + xn^2 - yn^2}{2 \cdot nl \cdot xn} \tag{6.12}$$

To get the final vector angle we need to take into account also the initial vector angle α_{old} and the addition from the previous vector $addition_{prev}$.

$$addition = \frac{\pi}{2} - \beta \tag{6.13}$$
$$\alpha_{new} = \alpha_{old} + addition - addition_{prev} \tag{6.14}$$

Finally, the new vector is formed by the direction α_{new} and by the magnitude ln. The fitness function for a segment with n vectors $\vec{v}$ is computed as:

$$f = \sum_{i=0}^{n} \frac{\vec{v}^2_{i_{\alpha_{old}}}}{\vec{v}^2_{i_{\alpha_{new}}}} + c \cdot \sum_{i=0}^{n} \frac{\vec{v}_{i_l}}{\vec{v}_{i_{ln}}} \tag{6.15}$$

6.2.4 Segment composition

Once all segments are optimized the composition is done to form the whole track again. As the segment with n vectors is formed by a chromosome of $n+1$ length the overlapping genes (ending gene of a segment and beginning gene of the following segment) are combined into one by their average. The process is displayed in the figure 6.8.

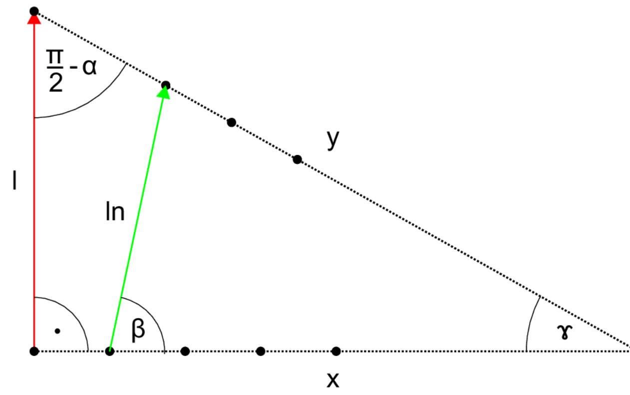


Figure 6.7: Two segment relation detail used during the fitness evaluation

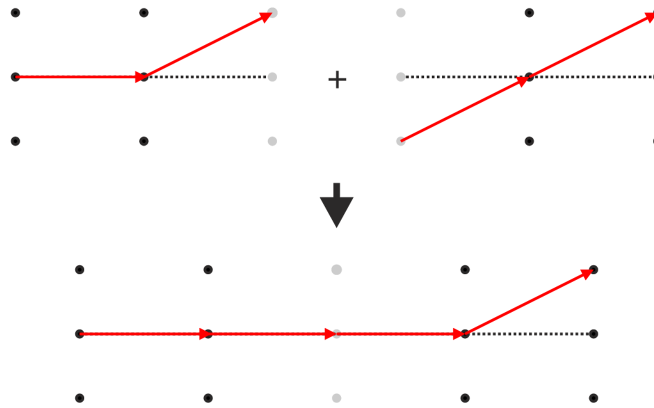


Figure 6.8: Composition of two segments into one

6.2.5 Track optimization

Track optimization is carried out to smooth out the gaps between segments. A gap may occur for example if a left turn follows after a right one. Track optimization carried out in the same way as Segment optimization. The same genetic algorithm is used with a difference of the initial seed while each of the individuals is seeded as composed solution from a previous step.

6.3 Race stage

During the race stage the controller benefits from the race line obtained in the warm-up stage. A simple logic was proposed to follow the optimized path and change the speed according to the following curvature.

Steering control is based on the steering proposed in the warm-up stage (see Equation 6.2):

$$steer = angle - \left(trackPosition - \frac{o_{i+1}}{\frac{n}{2}}\right) \tag{6.16}$$

where o_{i+1} is the following optimization gene and n is a number of division points.

The acceleration is computed by the same equation as in the warm-up stage 6.1, however the $desiredSpeed$ is predicted from the following set of vectors according to the curvature. The curvature for n following vectors $\vec{v}$ is computed as:

$$curvature = \frac{\sum_{i=0}^{n} |\vec{v}_{i_{\alpha_{new}}}|}{n} \tag{6.17}$$

6.4 Interaction between warm-up stage and race stage

The warm-up stage and the race stage are formed by two separate client runs thus the data obtained in the warm-up stage needs to be persistently saved to disk. For each warm-up run three different files are created:

6.4.1 Initial vectors file

An initial vector file contains all vectors sampled during the warm-up stage. However these vectors are not used any way during the race stage they are well used during debugging to compare the initial path with the optimized one.

6.4.2 Final vectors file

Similar to the *Initial vectors file*, this file contains vectors also with the difference being that these vectors represent the final optimized path. The final path is obtained from the initial vectors by the same logic that uses the fitness function. This file is important for the race stage because the maximum speed prediction is done based on the final vectors.

6.4.3 Optimization values file

The last file contains the optimization values of the best chromosome which is used for the prediction of the correct way during the race stage.

Chapter 7

Implementation

This chapter present the implementation details of the proposed controller described in chapter 6. At first the application structure is introduced followed by the stage work-flow description. The implementation details of each stage are presented afterwards.

7.1 Application structure

The application was implemented in the C++ programming language based on the client provided for *The Simulated Car Racing Championship* (section 2.1). The package can be downloaded from the CIG project page[1] and provides a stand-alone console application which purveys the UDP communication with a server and composes the sensors/actuators wrapper. The Genetic Algorithm Utility Library (GAUL) [33] has been used for the race line optimization.

The application structure is displayed in figure 7.1 while the main class of the autonomous driver is providing the car control a `Driver` class. The vehicle control is done via virtual methods from the base class `BaseDriver`, the purpose of this class was already discussed in section 2.2. The `WrapperBaseDriver` class provides a virtual method `CarControl wDrive(CarState cs)` which is used instead of `string drive(string sensors)` method from the `BaseDriver`. The `wDrive()` method encapsulates all sensors into `CarState` object by parsing the input string obtained from server. The `CarControl` object then builds the output string from given drive directives.

The `TrackDetail` is a class which holding all information about the track and providing input/output file control. A method for the initial trajectory, final trajectory and the saving of optimization values is used at the end of the warm-up stage to preserve these values for the other stages of the race. These files are loaded during the initialization of the stage (see 7.2.1). This class is also used during the track sampling of the warm-up stage to preserve the sampled vectors by the `_vectors` attribute and during other stages to obtain the maximum speed for the next sector of a track or the next optimization value.

The optimization is done via `Optimization` class which allows optimization by segments or the entire track. In this approach, the segments are optimized before the track optimization.

The `SegmentOptimization` and `TrackOptimization` classes contain Genetic algorithms for segment/track optimization.

[1] http://sourceforge.net/projects/cig/

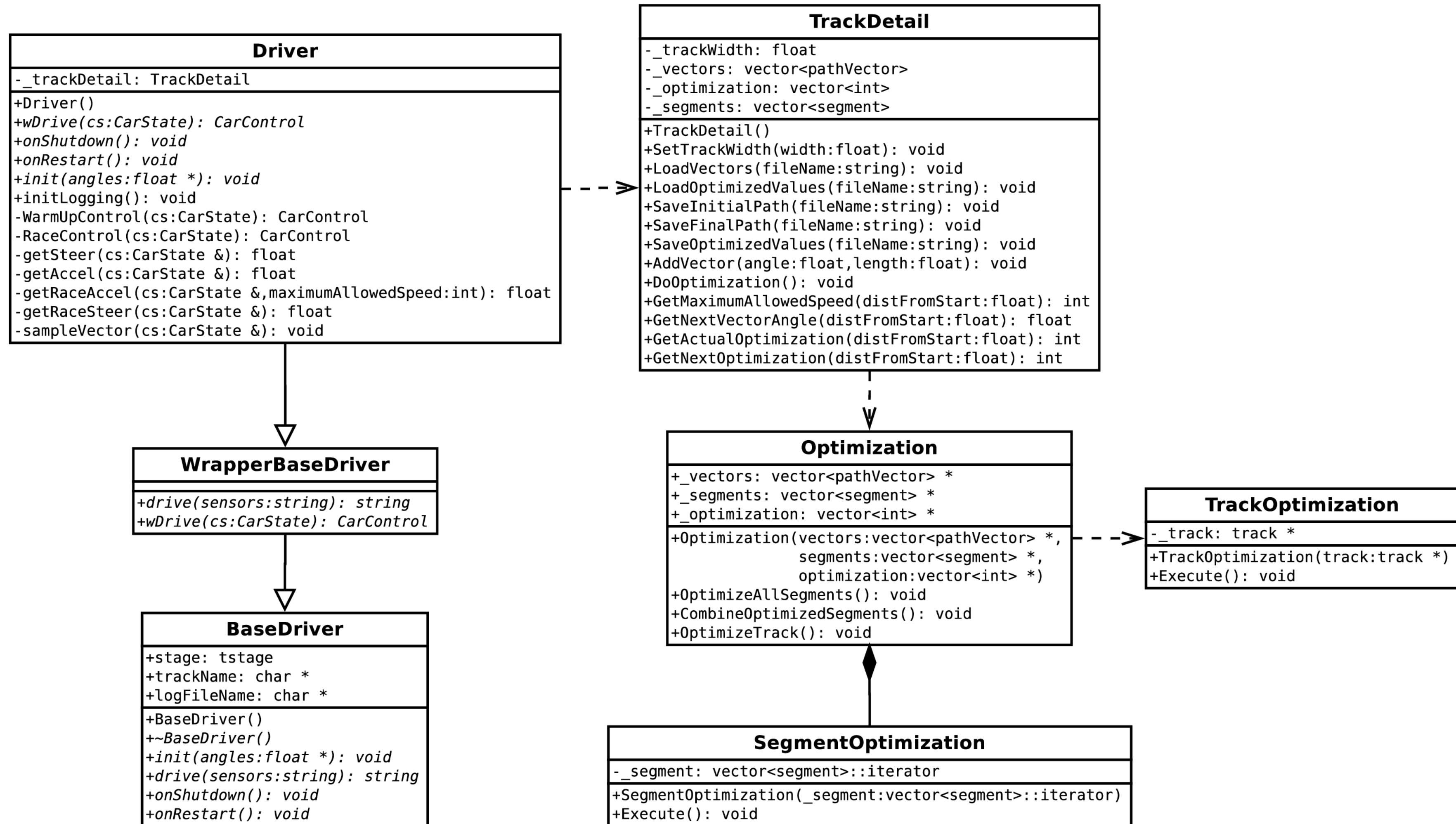


Figure 7.1: Application class diagram

7.2 Stage work-flow

Each client run can be divided into three main parts displayed in figure 7.2 while different methods are called during each of them.

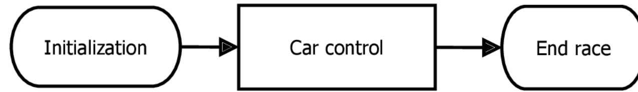


Figure 7.2: Stage work-flow

7.2.1 Initialization

At the beginning of each stage the driver needs to be initialized. Sensor initialization is mandatory and is done by the `void init(float *angles)` method during the UDP client identification. The car has 19 range sensors which need to be initialized. The initialization is done the same way for all stages of the race:

- 9 sensors in the middle by 5 degrees
- 10 side sensors by 15 degrees

A new method `void initLogging()` was added to facilitate loading of the information about the track which was sampled during the warm-up stage. This is done during the qualification or the race stage by `LoadVectors()` and `LoadOptimizedValues()` methods. For the file structure description see section 7.5.

7.2.2 Car control

The car control is the main part of the stage work-flow when the server provides information about the environment and driver reacts to it by effectors. According to the stage of the race either `WarmUpControl()` method for the warm-up stage or `RaceControl()` for all other stages is used. Both of the implemented controls follow the same basic idea mentioned below:

1. get gear
2. get steer
3. get throttle/brake
4. calculate clutching

Gear change and clutching calculation is kept unchanged from the client proposed for *The Simulated Car Racing Championship* while methods to obtain steer and throttle have been changed to conform the requirements.

7.2.3 End of the run

At the end of each run either `void onShutdown()` or `void onRestart()` method is called to inform the driver that the run has ended. This is used by the warm-up stage to store all obtained information into the files (see 7.5).

7.3 Warm-up stage

This section and the following section clarify information in the Controller design chapter (see 6) and complement the implementation details.

7.3.1 Track sampling

Track sampling is being done while the car is driving at a constant speed in the middle of road. The speed of 40 kilometers per hour was experimentally chosen because it is sufficiently slow speed to do track sampling and on the other hand fast enough to complete one lap in a specified time. This means that the car is able to take as many vector samples as desired and to react sufficiently fast to keep the car still in the middle of the track. With a higher speed the car goes off the center of the track by going through sharp turns. This causes wrong track sampling.

The steering control is provided by the `getSteer()` method while the equation in section 6.2 was slightly modified to provide faster reaction to the track curvature:

$$steer = 7 \cdot angle - 3 \cdot trackPosition \tag{7.1}$$

Sample are taken by the `sampleVector()` method which uses sensors at positions 0 and 2 for the left side of the track and 18, 16 for the right side of the track. According to the sensor initialization in 7.2.1 this pair of sensors cover a $\frac{\pi}{6}$ angle. It is essential to take a sample from the outer side of the turn otherwise a wrong model will be created. This is explained in figure 7.3 where the right side sensor s_{16} will obtain an incorrect value.

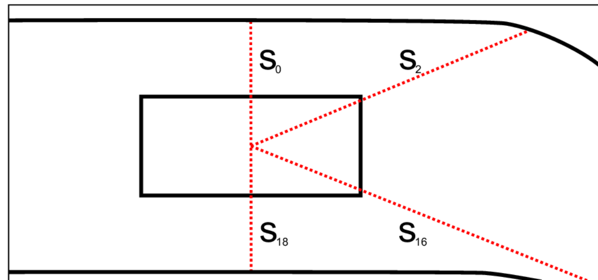


Figure 7.3: Wrong sampling by the right side of a car

7.3.2 Segment creation

The track is divided in the middle of each narrow part of the track based on the turn identification. A constant `TURN_THRESHOLD` determines the angle in which a current vector is considered as a part of the turn or not. The angle needs to be provided in radians and the number of segments created relies on this constant.

7.3.3 Segment optimization

Segment optimization is done by the *The Genetic Algorithm Utility Library* (GAUL) while the original Darwinian Genetic algorithm is being used. Many different rules for the passage of parent individuals into subsequent generations exist while in this approach all parents that rank sufficiently highly will pass to the next generation. This will preserve the currently

best solution into the next generation. The integer representation of individuals is limited by $\left(-\frac{n}{2}, \frac{n}{2}\right)$ where n is defined by a constant `NUMBER_OF_POSITIONS`. This constant determines the number of division points for each vector and it need to be set to the same value during the warm-up stage and the race stage otherwise the race prediction will not work correctly in the race stage.

In the initial seed, all genes of each individual are set to the same random value from the range $\left(-\frac{n}{2}, \frac{n}{2}\right)$ which will gain the required diversity.

A crucial aspect of GAs is the selection operator which is used to choose individuals from the population for crossover and mutation. For this purpose GAUL allows the definition of a single individual selection operator (for mutation) and a double selection operator (for crossover). The single individual selection operator in this solution selects the best solution from the population. A roulette-wheel algorithm is used to select two individuals. Both of these operators are built in the GAUL.

The algorithm convergence is most influenced by the mutation operator which was built according to the design described in 6.2.3. At the beginning, the direction is chosen to determine whether genes will be decremented or incremented. Then, a couple of genes in a row are modified. The length of mutation is generated randomly from interval $(0, l)$ where l is a length of the chromosome. A mean crossover from the GAUL library is used as a crossover operator.

The fitness function was implemented along the design described in 6.2.3 where the racing line is used as an evaluative criterion. The constant `SEGMENT_FITNESS_TRADE_OFF` determines the trade-off between *Minimal curvature path* and *Shortest path*. The range of this constant is $[0, 1]$ where 0 stands for the *Minimal curvature path* and 1 for the *Shortest path*. Fitness function source code can be found in the Appendix A.

The Genetic algorithm used for segment optimization can also be modified by the following parameters located in the `Constants.h` file:

`SEGMENT_POPULATION_SIZE` determines the size of the population

`SEGMENT_MAX_GENERATIONS` defines the maximum number of generations

`SEGMENT_CROSSOVER_PROBABILITY` defines the crossover probability, where $p_c \in [0, 1]$

`SEGMENT_MUTATION_PROBABILITY` defines the mutation probability, where $p_m \in [0, 1]$

7.3.4 Track optimization

The same Genetic algorithm as for the Segment optimization is used for the track optimization where the only difference is the initial seed. Each individual is seeded as a composed optimized segment from previous step while each gene is increased by a random number from the range $\left(-\frac{n}{2}, \frac{n}{2}\right)$. Due to the long length of chromosome the length of mutation is limited by a constant `MAXIMUM_TRACK_MUTATION_LENGTH`. A fitness value trade-off can be adjusted by `TRACK_FITNESS_TRADE_OFF`. As well as the segment optimization, the track optimization can be also modified by following parameters:

`TRACK_POPULATION_SIZE` defines the size of the population

`TRACK_MAX_GENERATIONS` determines the maximum number of generations

`TRACK_CROSSOVER_PROBABILITY` defines the crossover probability, where $p_c \in [0, 1]$

`TRACK_MUTATION_PROBABILITY` defines the mutation probability, where $p_m \in [0, 1]$

After the track optimization ends the initial and optimized track represented by vectors and optimization values are saved into output files (see section 7.5).

7.4 Race stage

A simple controller logic which utilizes the path evolved during the race stage has been implemented. At the start of the warm-up stage the optimized path and optimization values are loaded from files.

A steering control is obtained by the `getRaceSteer()` method while the steer value is computed by equation 6.16 shown in design. The optimization o_{i+1} is retrieved by the `GetActualOptimization(int distFromStart)` method and n is defined by the constant `NUMBER_OF_POSITIONS`. The `distFromStart` sensor value (see table 2.1) is used to identify the position at the lap thus finding a corresponding optimization value. Since the position -1 or 1 of the *trackPosition* denotes the position of the car's center to the roadside, the maximum position needs to be limited. The `MAX_POSITION` constant defines the maximal value of $\frac{o_{i+1}}{\frac{n}{2}}$.

The acceleration is computed by the equation 6.1 like in the warm-up stage with a difference that *desiredSpeed* is a variable obtained by the `GetMaximumAllowedSpeed(int distFromStart)` method. The speed prediction is done based on the curvature of the following path (equation 6.17). The length of the following path taken into account is defined by the `FUTURE_LENGTH` constant. The predicted speed based on the curvature is described in table 7.1. These values were experimentally chosen to provide the best controller performance.

Table 7.1: The maximal speed based on the following curvature

Curvature	< 0.008	< 0.01	< 0.02	< 0.03	< 0.06	> 0.06
Maximal speed	200	160	130	100	50	40

7.5 Files specification

The information about the track obtained in the warm-up stage is stored into files to preserve it for the next stages. At the end of each warm-up stage three files are created:

`LOGFILENAME` preserves the initial sampled track where each vector is stored at a new row. The vector is stored as an angle and length while these values are separated by a blank space.

`LOGFILENAME_final` keeps the optimized track composed by vectors. The file format is the same as in the case of `LOGFILENAME` while each row consists of one vector composed by an angle and length separated by a blank space.

`LOGFILENAME_optimized` holds the data of the final chromosome. Each gene is stored on a separate row.

The `LOGFILENAME` can be specified by a parameter of the application (the implicit value is `default`):

```
$ ./client logFileName:<l>
```

7.6 Track visualization

The visual comparison of the evolved solution with the initial one is important because it is difficult to evaluate the better solution using just the values of the vectors. The best way to compare two solutions is to display them in one figure. The visualization was essential during debugging of the track sampling and it is also well used during segment and track optimization.

The visualization is done in MATLAB [32] while the *2D Progressive Vector Diagram* [31] function is used to plot vectors. The input of this function demands vectors in Cartesian coordinate system[2]. The `getVectorComponentsFromFile()` function was implemented to obtain Cartesian components of the vectors and display them. Function `comparePaths()` allows comparisons between the initial path and optimal paths. Both functions can be found in Appendix B.

[2]Cartesian coordinate system specifies each point by pair of numerical coordinates

Chapter 8

Experiments

The behaviour of the proposed car driver was tested on five chosen tracks all of which can be found in table 8.1. The selection consists of the mountain road track *Alpine 2* where the reliability of the Track sampling (see section 8.1) was tested or the oval track *Michigan Speedway* where the Segment creation (see section 8.2) is difficult due to the distinction for turning.

Experiments follow the logical process of the controller's behaviour described in chapter 6. Since just a certain amount of time is dedicated to a controller in the warm-up stage, hence the time elapsed from the start of the warm-up stage is displayed in table 8.1 for each track. The longest run is $7'15''90$ which means that the controller still has a lots of free time to reach the limit 30 minutes (see 6.2).

Table 8.1: The time elapsed from the start of the warm-up stage

Track	Track sampling	Segment creation	Segment optimization	Track optimization
Aalborg (2567.54m)	$3'46''54$	$3'46''72$	$4'33''66$	$4'55''86$
Alpine 2 (3773.57m)	$5'30''89$	$5'30''93$	$6'37''00$	$7'09''92$
CG Speedway 1 (2057.56m)	$3'07''47$	$3'07''50$	$3'31''36$	$3'43''28$
Michigan Speedway (2311.79m)	$3'27''18$	$3'27''30$	$3'59''05$	$4'01''79$
Wheel 1 (4257.62m)	$6'04''30$	$6'04''43$	$6'52''30$	$7'15''90$

8.1 Track sampling

Track sampling is the first step of the warm-up stage while the car is going by a constant speed of 40 kilometers per hour. The sampling was tested on the tracks mentioned in table 8.1. As it can be seen from the table, track sampling is the most time consuming part of the warm-up stage. The longest sampling time has was on the *Wheel 1* track and this was caused by the length of the track.

During sampling, the main criteria which needs to be met is the minimal bias of the central line. However the controller logic is not able to keep the strict constant speed which can be seen on the left in figure 8.1, the bias of the central line is still minimal (see the figure on the right in figure 8.1). The largest speed deflection was logged during the *Alpine 2* sampling due to mountainous terrain.

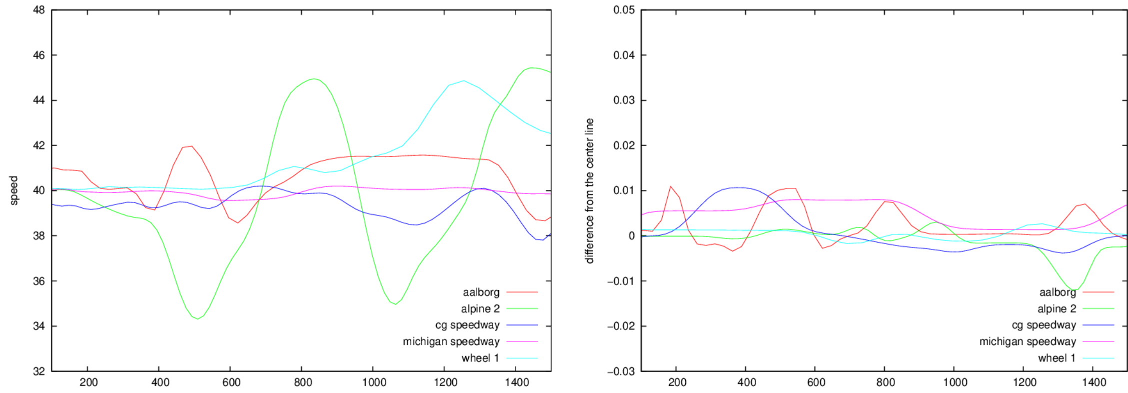


Figure 8.1: Speed and distance from the centre of the road during sampling

Two of the sampled tracks are displayed in figure 8.2. It is apparent that the end of the track does not meet with the beginning of the track which is caused by the inaccurate angle values. This probably happens due to vector sampling using the same driving distance all the throughout despite different lengths at corners. This can be fixed either by a dynamic change of vector length sampling or by angle adaptation at corners. Beyond that, the sampled track is still sufficient due to track representation by vectors which express the relative direction from one point to the next one. This means that the end of the track links smoothly to the beginning of the track.

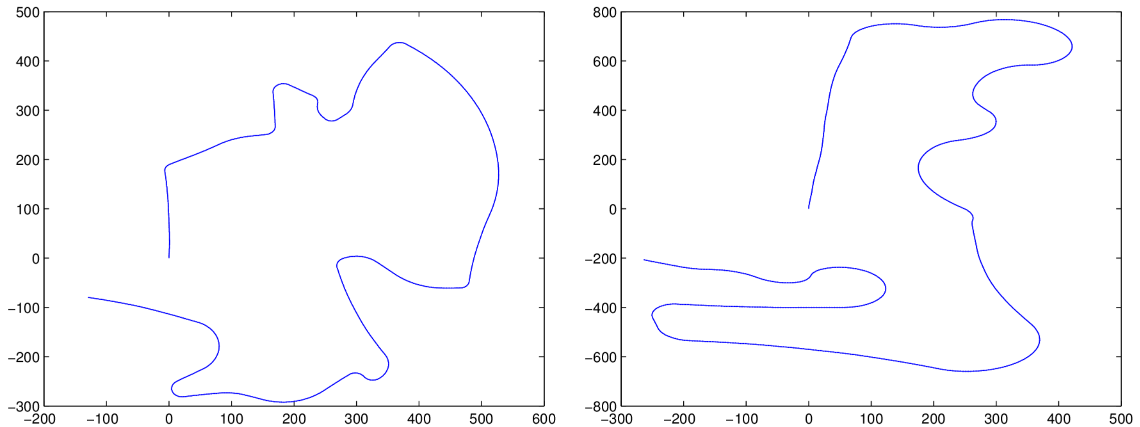


Figure 8.2: Sampled tracks *Aalborg* and *Wheel 1*

8.2 Segment creation

Division of the track into segments is done in the middle of each narrow part between corners. The number of created segments can be affected by the TURN_THRESHOLD constant. The segments in picture 8.3 were created from the *Aalborg* track with a TURN_THRESHOLD value 0.016 rad. The effect on the created segments by the TURN_THRESHOLD is displayed in table 8.2.

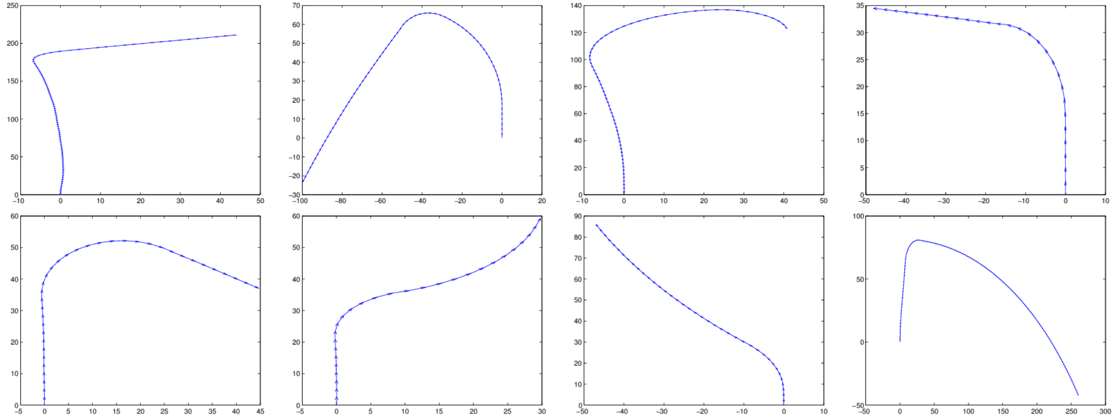


Figure 8.3: A few segments created from track *Aalborg*

Table 8.2: Number of segments created in relation to the TURN_THRESHOLD constant

	0.010	0.016	0.5
Aalborg	18	17	13
Alpine 2	15	16	17
CG Speedway 1	9	9	7
Michigan Speedway	3	2	2
Wheel 1	9	10	15

8.3 Segment optimization

Segment optimization is the most important part of the warm up stage while several configurations were explored to obtain the best results. The Segment optimization process is demonstrated at the first segment of the *Aalborg* track (see figure 8.6) while the track width is set to 5 meters.

In all figures the fitness function proposed in equation 6.15 is used which expresses the ration between the initial and evolved track. This means that a fitness value of 1 represents the initial path or the path as good as initial one. A number higher than 1 then represents the better solution. The tests below are based on the following parameters:

NUMBER_OF_POSITIONS: 50

SEGMENT_POPULATION_SIZE: 50

SEGMENT_MAX_GENERATIONS: 10000

SEGMENT_CROSSOVER_PROBABILITY: 0.7

SEGMENT_MUTATION_PROBABILITY: 0.3

SEGMENT_FITNESS_TRADE_OFF: 0.7

The left graph in figure in 8.4 compares the separate runs with default parameters. As can be seen from the figure, fitness function values of each run grow until the 5000th generation and then the solution does not evolve further.

The right graph in figure 8.4 displays the fitness convergence with the effects of the track width. The wider the track is, the lower the fitness value is reached. By increasing the NUMBER_OF_POSITIONS to 100 for a track 10 meters wide, almost the same fitness value is acquired (refer to the purple line in the figure).

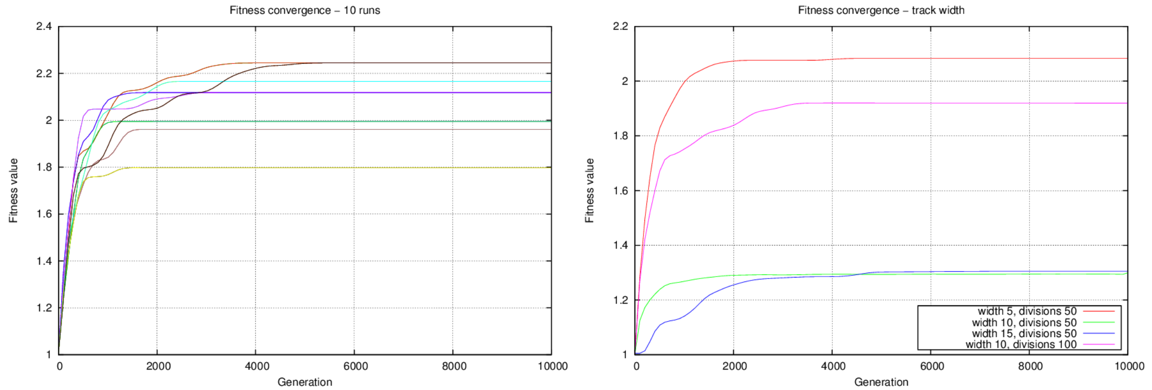


Figure 8.4: A comparison between fitness values on separate runs and the relationship between fitness values and differing track widths

Fitness convergence of the algorithm is largely dependent on the number of division points for each vector, which is displayed in figure 8.5. For a track 5 meters wide, 50 division points achieves the fastest convergence of the algorithm.

On the basis of the graph on the right in figure 8.4 it can be deduced that the best ratio between the track width and number of division points is $d = w \cdot 10$ where d is a number of division points and w is the track width.

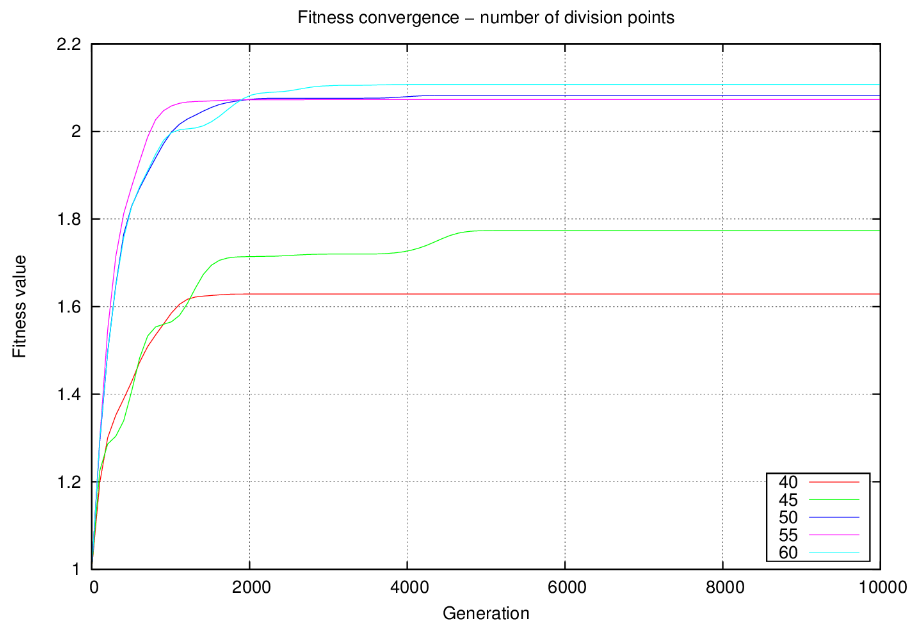


Figure 8.5: The fitness convergence of the track 5 meters wide with the effects of a number of division points

The difference between the same segment of the track 5 meters wide evolved for 30

and 50 points is displayed in figure 8.6 where the blue line represents the initial path, the green line shows the path that has evolved for 30 points and red line shows the path that has evolved for 50 points. The green line therefore represents the worse solution as the advantage of the inner part of the turn is not used. Some of the other evolved segments can be found in figure 8.7.

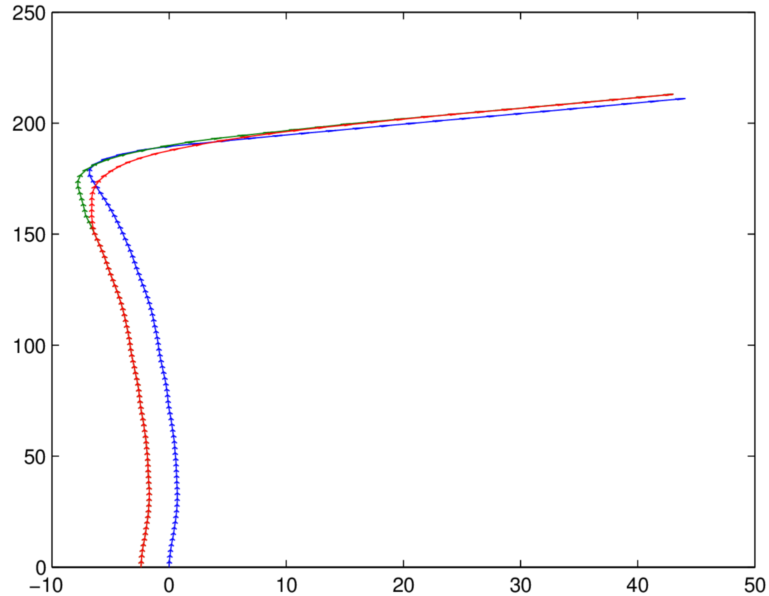


Figure 8.6: Optimized segment of the track 5 meters wide for 30 division points (green line) and 50 division points (red line) division points. The blue line represents the initial path.

8.4 Track optimization

The track optimization is applied to the composed path by the segments when the overlapping end points of each segment are averaged. The main purpose of the track optimization is to smooth out the gaps between segments. Various settings of the genetic algorithm were tested to achieve some passable results. The parameters which give the best results are described below:

`NUMBER_OF_POSITIONS:` 50

`TRACK_POPULATION_SIZE:` 30

`TRACK_MAX_GENERATIONS:` 1000

`TRACK_CROSSOVER_PROBABILITY:` 0.5

`TRACK_MUTATION_PROBABILITY:` 0.3

`TRACK_FITNESS_TRADE_OFF:` 0.7

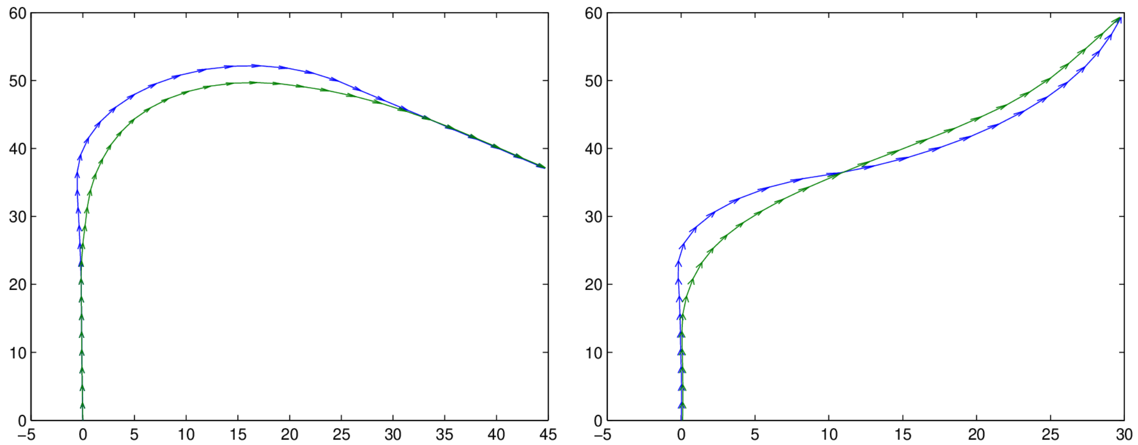


Figure 8.7: Optimized segments of the *Aalborg* track

A comparison of the gap between two segments before and after optimization is displayed in figure 8.8. The segment interpolation achieved via the segment composition (see section 6.2.4) is displayed on the left side. The figure on the right side then shows the image of the gap enhanced by the track optimization. The whole optimized track *Aalborg* can be found in figure 8.9 and the rest of the evolved tracks are in Appendix C.

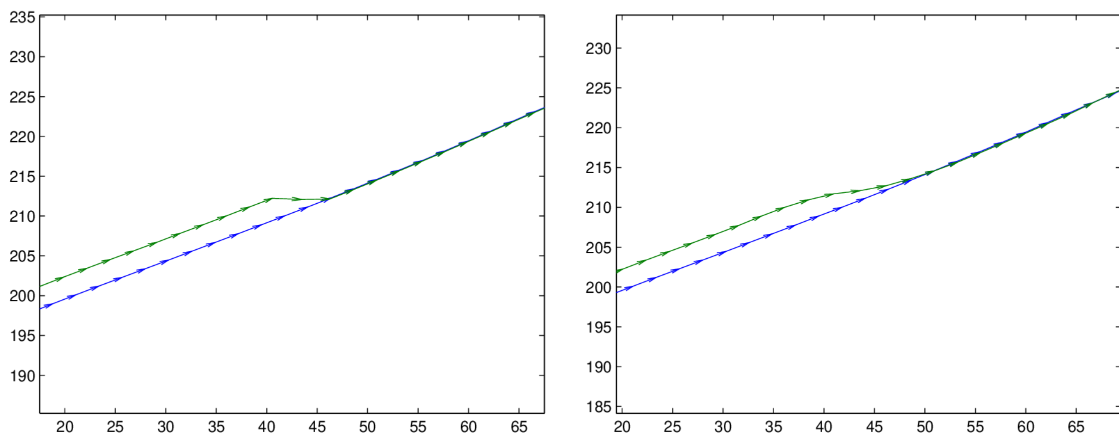


Figure 8.8: A gap between two segments smoothed out by the track optimization (the *Aalborg* track).

The Genetic algorithm is not the best approach to do small enhancements but it is still able to optimize the problem and produce a solution. The better approach would be, for example, *Hill climbing*[1] applied to the segment transitions.

The track optimization could be excluded from the process in the case where the each segment is optimized based on the fact the starting point is fixed to the last gene of the previous segment.

[1]Hill climbing is an optimization technique good for a local optimum finding

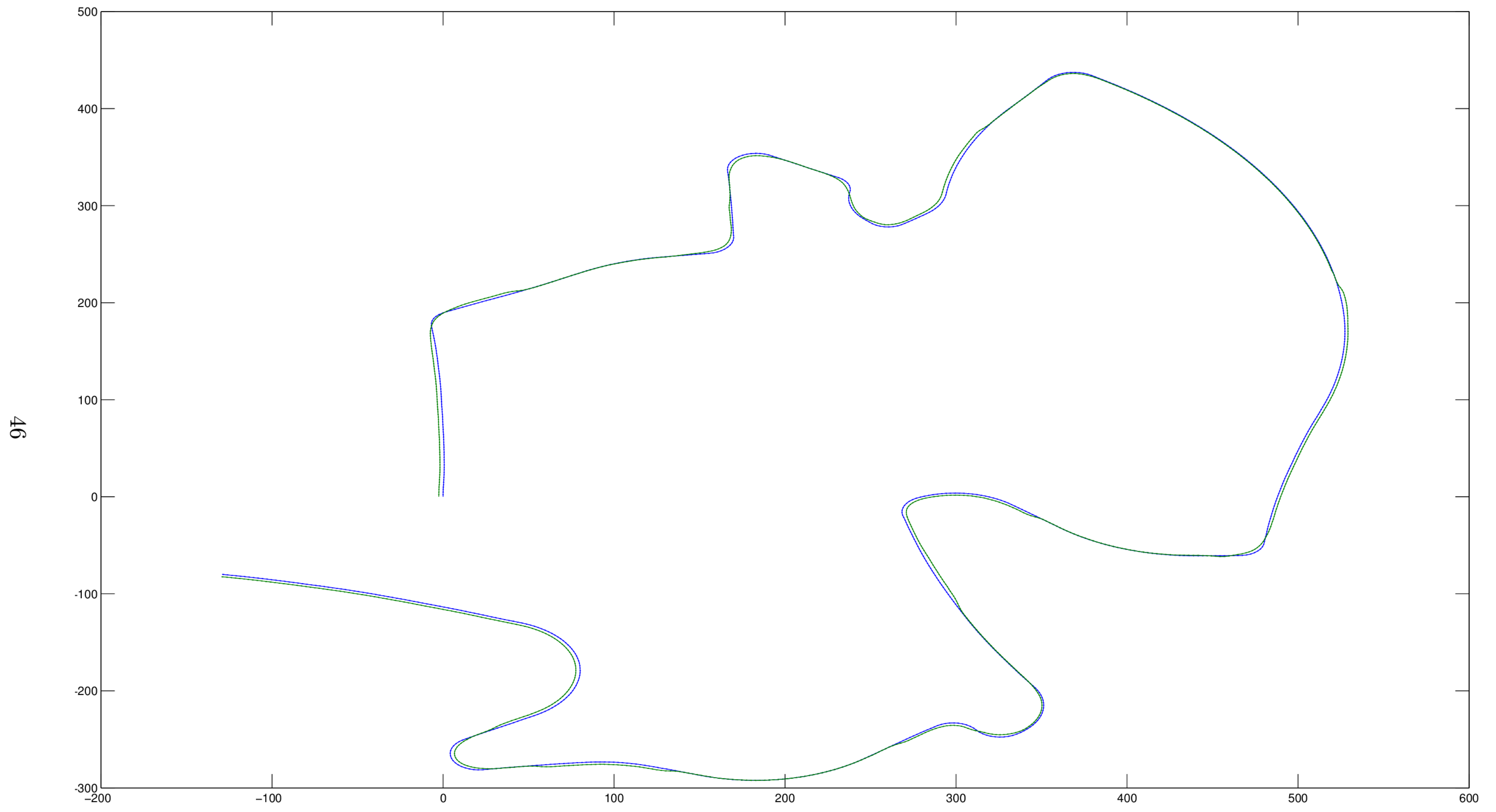


Figure 8.9: Optimized the Aalborg track

8.5 Comparison with other solutions

The performance of the proposed controller was compared with the Bernhard Wymann controller (called berniw in TORCS) and a human player during the qualification stage. This means that each of the tested controllers run on the track alone and the best lap was recorded. The best lap times are displayed in table 8.3.

From the results it is evident that the proposed controller cannot outperform neither the berniw's controller nor the human player. This is caused by the fact that the simple race stage logic is not able to fully utilize the optimized path from the warm-up stage. The controller also has problems with deceleration on the turns following long narrow parts that sometimes deviate from the track. Despite all of that, the optimized path could form the fundamental part of the controller which will be able to outperform the other controllers.

Table 8.3: The best lap times during the qualification stage

Track	Human	berniw's controller	proposed driver
Aalborg	1′38″80	1′40″68	2′28″10
Alpine 2	1′48″30	1′55″15	3′33″10
CG Speedway 1	43″47	46″34	2′03″10
Michigan Speedway	42”53	41″32	2′21″30
Wheel 1	1′48″10	1′42″16	3′38″42

Chapter 9

Conclusion

The main aim of this thesis has been to propose an autonomous controller for the TORCS simulator which would take advantage of algorithms inspired by biology. On the basis of already proposed techniques, a new controller based on the GAs is carried out.

The controller is implemented in the C++ programming language while the Genetic algorithm optimization is done by the GAUL library. The controller behaviour can be divided into two main parts which are exploited during the different stages of the competition. The warm-up stage servers for the track sampling and the race line optimization. The race stage logic then benefits from the data obtained in the warm-up stage.

Track sampling is being done while the car drives at a constant speed in the middle of road. The track model is being built on the fly and it is represented by a set of vectors. Each vector determines a relative direction from the current point and is represented by a direction and a magnitude of polar coordinates.

Once the single lap has been sampled, the obtained track model is divided into smaller parts called segments and all segments are optimized separately. The division is done in the middle of each narrow part of a track and it is important because the entire track optimization would be unsuccessful due to complexity of the problem.

The segment optimization is done by the GA which uses as an evaluative criteria of the racing line. When the GA is applied to the proposed track representation, a new mutation and fitness function which computes the final trajectory is introduced.

However the division of the track into segments yields gaps between the segments, these gaps need to be smoothed out by another GA utilizing the whole trajectory which is composed of segments. The optimized trajectory is used during the race stage while an optimal position on the track and maximum allowed speed are obtained.

The implemented controller was tested on five tracks of the TORCS simulator and compared with other solutions. However, the implemented controller is not as fast as other controllers the optimized trajectory with a proper race logic has a big potential.

9.1 Own contribution

The main findings of this work are a new track model representation, different track sampling and race line optimization. Although various solutions have been proposed, most of the implementation details are kept hidden due to the competition held in the field of autonomous driving. This work proposes the complete solution which can be used to build a highly competitive driver.

9.2 Future work

The major enhancement which can be done on the warm-up stage is the segment composition. Although the segment gaps are enhanced by the GA, this approach is not able to fully remove them. One solution could be to start the optimization of each following segment with a fixed starting point to the last optimized gene from the previous segment.

The main focus of future work should be on the race stage. Based on the track representation, the Artificial Neural Networks (ANN) trained beforehand might form a race stage logic which will fully use the potential of the obtained racing line. The ANN might also be very effective to overtake other drivers. As well as trajectory optimization, the gear shifting can also be improved by evolutionary techniques.

Bibliography

[1] Q. Chen, U. Ozguner, and K. Redmill, "Ohio state university at the 2004 darpa grand challenge: Developing a completely autonomous vehicle," *IEEE Intelligent Systems*, vol. 19, pp. 8–11, September 2004.

[2] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*, vol. 12 of *Natural Computing Series*. Springer, 2003.

[3] F. Beltman, "Optimization of ideal racing line,"

[4] Torcs, "Torcs official website," 2011. [Online; Visited by 23.11.2011].
URL http://torcs.sourceforge.net/

[5] D. Loiacono, L. Cardamone, and P. Lanzi, "Simulated Car Racing Championship Competition Software Manual," Tech. Rep. February, Technical Report, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy, 2011.

[6] SIGEVO, "Genetic and evolutionary computation conference," 2011. [Online; Visited by 23.11.2011].
URL http://www.sigevo.org/gecco-2011/

[7] D. Loiacono, L. Cardamone, M. V. Butz, and P. L. Lanzi, "The 2011 Simulated Car Racing Championship @ Evo*-2011." [conference presentation], 2011.

[8] D. Loiacono, L. Cardamone, M. V. Butz, and P. L. Lanzi, "The 2010 Simulated Car Racing Championship @ CIG-2010." [conference presentation], 2010.

[9] D. T. Ho and J. M. Garibaldi, "A Fuzzy Approach For The 2007 CIG Simulated Car Racing Competition," *2008 IEEE Symposium On Computational Intelligence and Games*, pp. 127–134, 2008.

[10] D. Perez, G. Recio, Y. Saez, and P. Isasi, "Evolving a fuzzy controller for a car racing competition," in *Proceedings of the 5th international conference on Computational Intelligence and Games*, CIG'09, (Piscataway, NJ, USA), pp. 263–270, IEEE Press, 2009.

[11] E. Onieva, L. Cardamone, D. Loiacono, and P. Lanzi, "Overtaking opponents with blocking strategies using fuzzy logic," in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pp. 123–130, IEEE, 2010.

[12] J. Muñoz, G. Gutierrez, and A. Sanchis, "A human-like torcs controller for the simulated car racing championship," 2010.
URL http://e-archivo.uc3m.es/bitstream/10016/10179/1/human_munoz_2010_ps.pdf

[13] J. Togelius, P. Burrow, S. M. Lucas, and S. Member, "Multi-population competitive co-evolution of car racing controllers," *Computer*, 2007.

[14] J. Muñoz, G. Gutierrez, and A. Sanchis, "Controller for TORCS created by imitation," *2009 IEEE Symposium on Computational Intelligence and Games*, pp. 271–278, Sept. 2009.

[15] Brooks and R. A., "A Robust Layered Control System For a Mobile Robot," tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.

[16] Hiong and T. A. N. Chin, "Enhancing player experience in computer games: A computational intelligence approach," *Autonomous Robots*, pp. 53–63, 2010.

[17] D. Perez, G. Recio, and Y. Saez, "Evolving a fuzzy controller for a Car Racing Competition," *2009 IEEE Symposium on Computational Intelligence and Games*, pp. 263–270, 2009.

[18] A. Agapitos, J. Togelius, and S. M. Lucas, "Evolving controller for simulated car racing with object-oriented genetic programming," in *Proceedings of the Genetic and Evolutionary Computing Conference GECCO*, 2007.

[19] M. Melanie, "An Introduction to Genetic Algorithms," *Computers Mathematics with Applications*, vol. 32, no. 6, p. 133, 1996.

[20] M. Hrušovský, "Genetic Algorithm Acceleration Using OpenCL," Master's thesis, Brno University of Technology, 2010.

[21] A. E. Eiben and J. E. Smith, "Introduction to Evolutionary Computing - Genetic Algorithms." [presentation], 2004.

[22] A. Di Pietro, L. While, and L. Barone, "Applying evolutionary algorithms to problems with noisy, time-consuming fitness functions," in *Evolutionary Computation, 2004. CEC2004. Congress on*, vol. 2, pp. 1254 – 1261 Vol.2, june 2004.

[23] L. M. Brad and D. E. Goldberg, "Genetic Algorithms , Tournament Selection , and the Effects of Noise," vol. 9, pp. 193–212, 1995.

[24] T. Blickle and L. Thiele, "A Comparison of Selection Schemes used in Genetic Algorithms," vol. 2, no. 11, 1995.

[25] S. W. M, "Crossover or Mutation ?," no. 1973, 1993.

[26] B. B. Beckman, "The physics of racing," *Physics*, pp. 1–2, 2002.

[27] G. Strang and K. Willcox, "Racing Line Optimization," *Science*, no. 2009, pp. 1–114, 2010.

[28] L. Cardamone, D. Loiacono, P. L. Lanzi, and A. P. Bardelli, "Searching for the optimal racing line using genetic algorithms," *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pp. 388–394, Aug. 2010.

[29] F. Braghin, F. Cheli, S. Melzi, and E. Sabbioni, "Race driver model," *Computers & Structures*, vol. 86, pp. 1503–1516, July 2008.

[30] J. Quadflieg and M. Preuss, "Learning the Track and Planning Ahead in a Car Racing Controller," *Intelligence and Games*, pp. 395–402, 2010.

[31] I. M. Mosquera, "Progressive vector diagram," 2009. [Online; Visited by 10.1.2012]. URL http://www.mathworks.com/matlabcentral/fileexchange/25940-2d-progressive-vector-diagram/content/ipvd.m

[32] MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[33] S. Adcock, "The genetic algorithm utility library," 2009. [Online; Visited by 6.5.2012].
URL http://gaul.sourceforge.net

Appendix A

Fitness function

```
boolean segment_score(population *pop, entity *entity)
{
    float curvature = 0;
    float totalLength = 0;

    segment *seg = (segment *)pop->data;

    float sectionWidth = seg->width / NUMBER_OF_POSITIONS;

    float alpha, beta,newAngle,angleAddition, newLength, x,
       xn, y, yn;

    int i =0;
    float oldAddition = 0;

    for (vector<pathVector>::iterator itVect =
       seg->vectors.begin(); itVect < seg->vectors.end();
       itVect++, i++ )
    {
        alpha = (itVect + 1) < seg->vectors.end() ? (itVect +
           1)->angle : 0.000001;

        x = itVect->length / tan(alpha);
        y = itVect->length / sin(alpha);

        xn = x + (((int
           *)entity->chromosome[0])[i])*sectionWidth;
        yn = y + (((int
           *)entity->chromosome[0])[i+1])*sectionWidth;

        newLength =
           sqrt(pow(xn,2)+pow(yn,2)-2*xn*yn*cos(alpha));

        beta = acos((pow(newLength,2)+pow(xn,2)-pow(yn,2)) /
```

```
            (2*newLength*xn));

        angleAddition = rightAngleInRads - beta;

        newAngle = itVect->angle + angleAddition -
            oldAddition;

        oldAddition = angleAddition;

        curvature += pow((fabs(newAngle)*RAD_TO_DEGREE),2);
        totalLength += newLength;
    }

    entity->fitness = (1 - SEGMENT_FITNESS_TRADE_OFF) *
       seg->totalLength/totalLength +
       SEGMENT_FITNESS_TRADE_OFF *
       seg->totalCurvature/curvature;
    return TRUE;
}
```

Appendix B

Matlab functions

B.1 comparePaths function

```
function [] = comparePaths(fileName1,fileName2,startPosition)

if nargin < 3
    startPosition = 0;
end

getVectorComponentsFromFile(fileName1)
hold on
getVectorComponentsFromFile(fileName2,startPosition)
hold off
```

B.2 getVectorComponentsFromFile function

```
function [vx,vy] =
   getVectorComponentsFromFile(fileName,startPosition)

if nargin < 2
  startPosition = 0;
end

A = importData(fileName);

angle = 0;

rows = size(A,1);

vx = zeros(1,rows);
vy = zeros(1,rows);

for  i = 1 : rows
    a = A(i,1);
    length = A(i,2);
```

```
    vx(i) = sin(a+angle)*length;
    vy(i) = cos(a+angle)*length;

    angle = angle + a;

end
ipvd(vx,vy,startPosition,0)
```

Appendix C

Optimized track figures

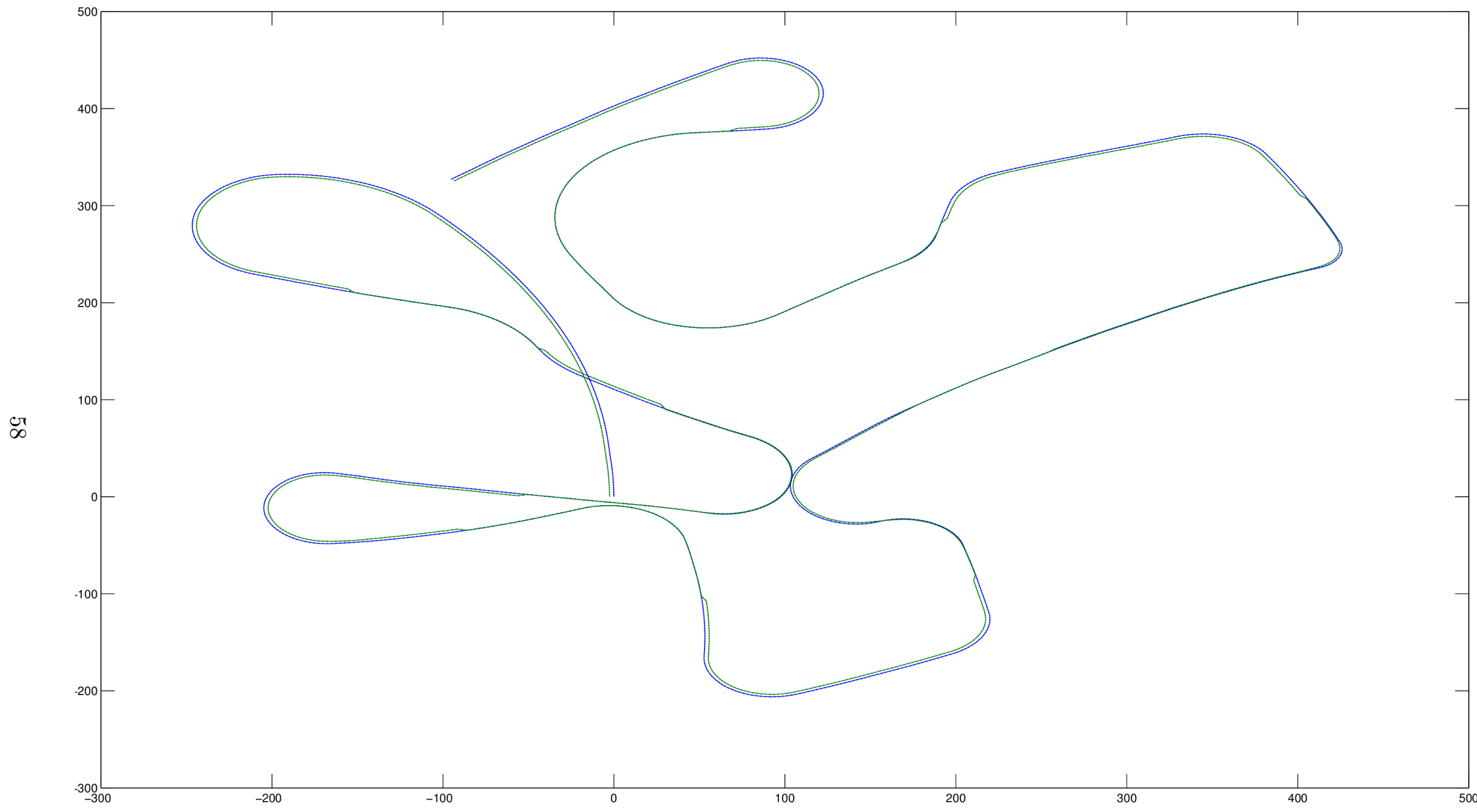


Figure C.1: Optimized Alpine 2 track

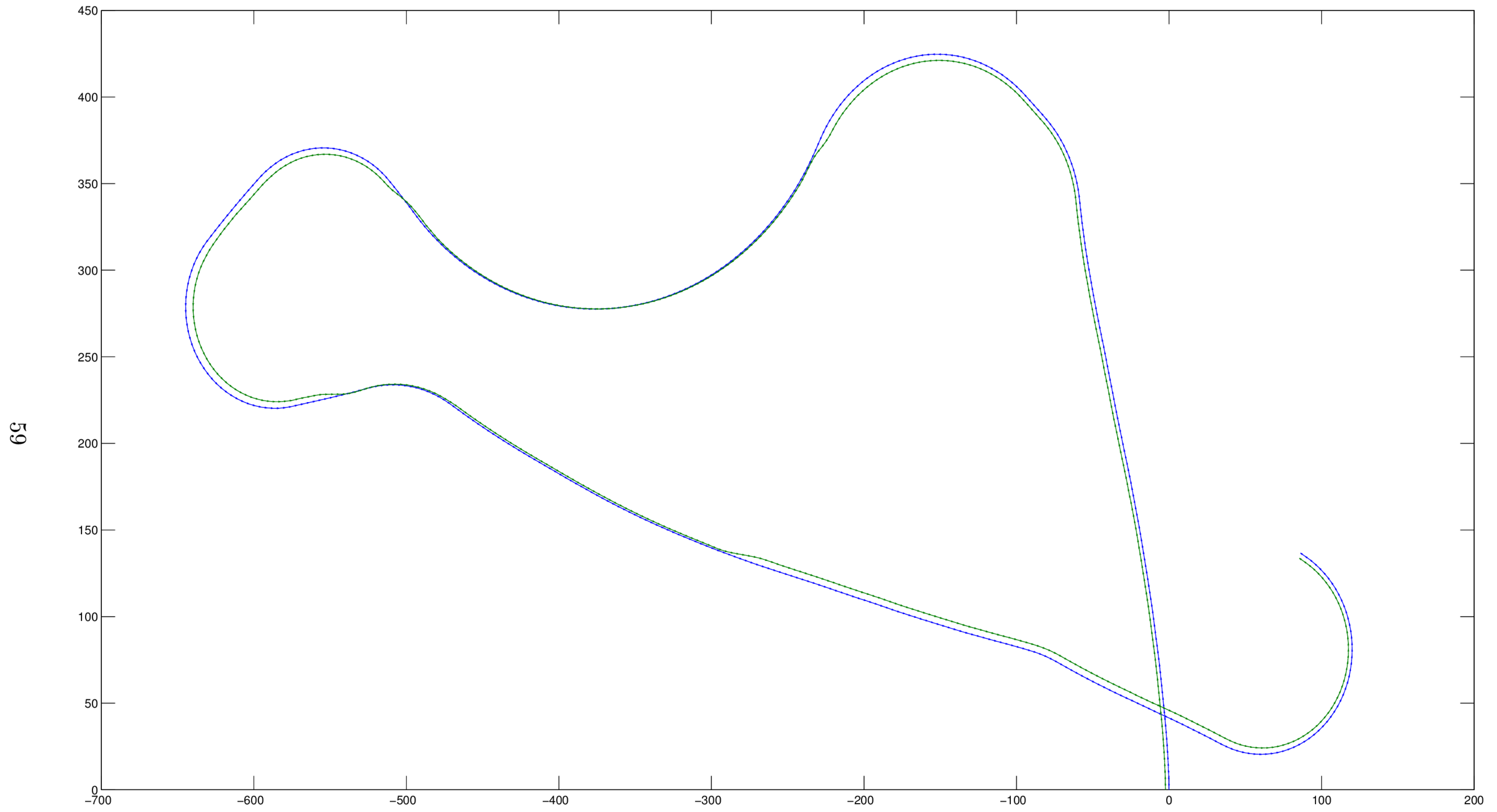


Figure C.2: Optimized CG Speedway number 1 track

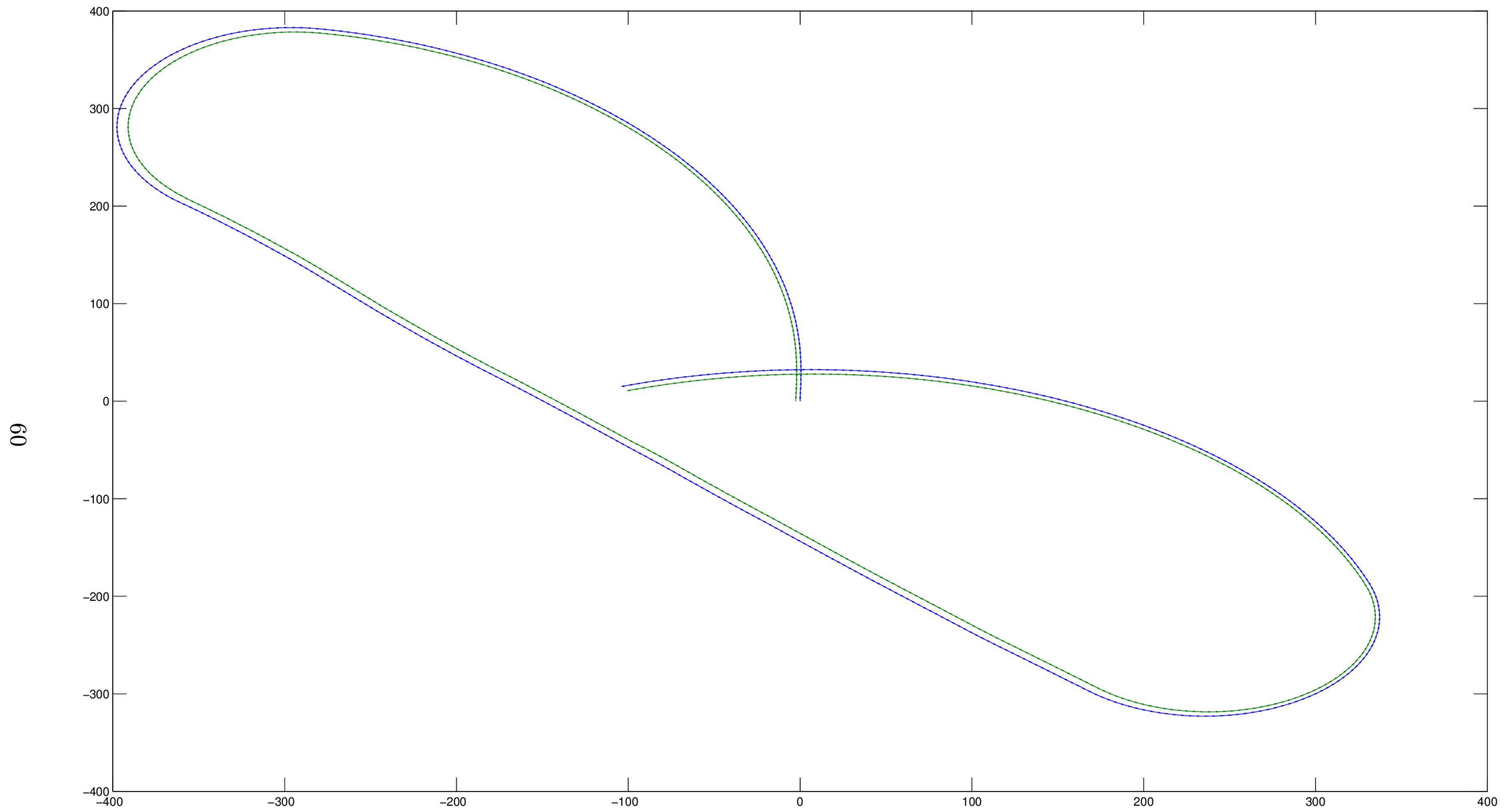


Figure C.3: Optimized Michigan Speedway track

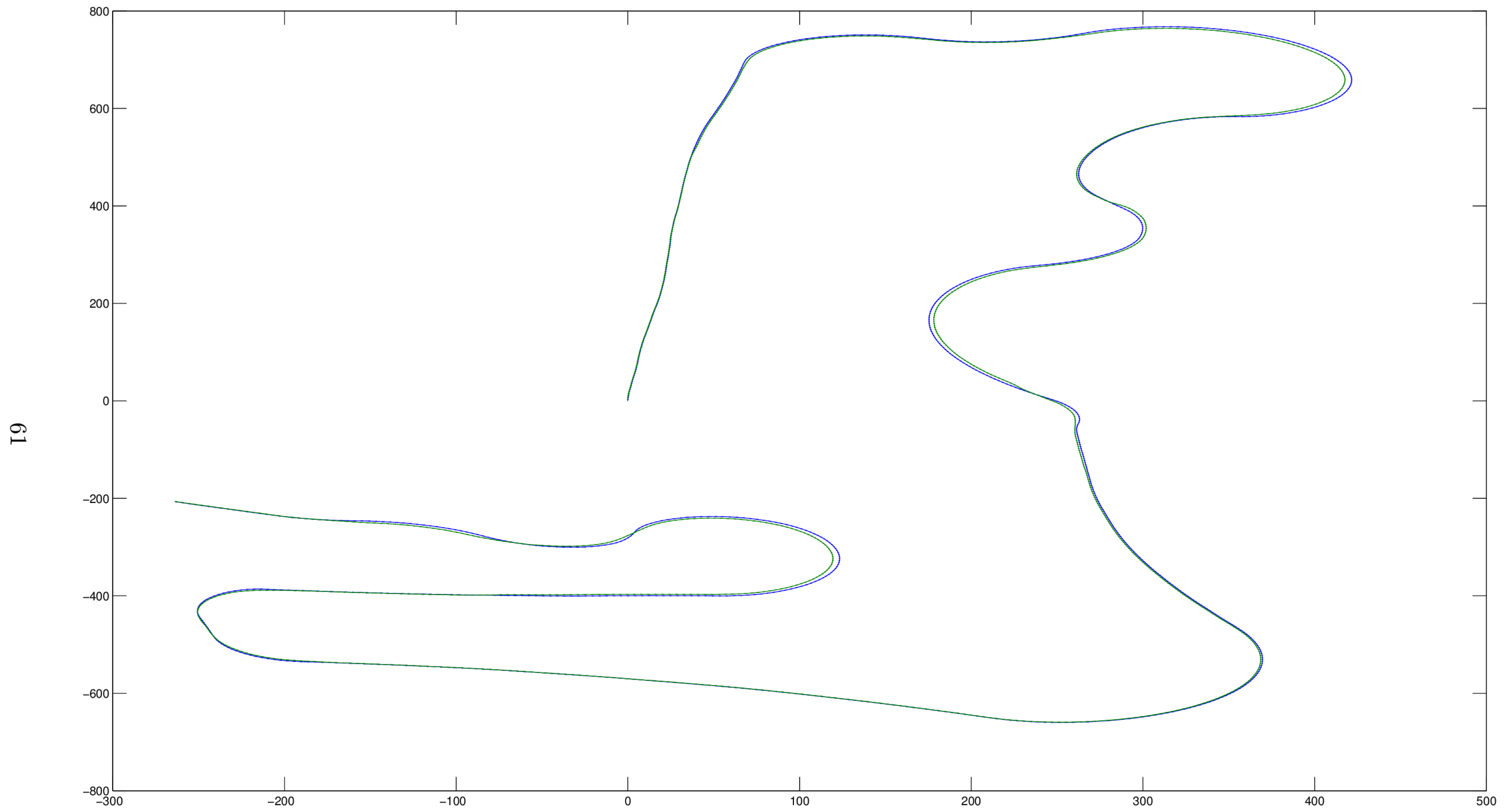


Figure C.4: Optimized Wheel 1 track