Evolutionary Approach to Finding an Optimal Racing Line in a Vehicle Simulator

Christopher Nosowsky*
nosowsky@msu.edu
Michigan State University
East Lansing, Michigan, USA

Kevin Smith*
smit2958@msu.edu
Michigan State University
East Lansing, Michigan, USA

ABSTRACT

Automotive racing is a popular form of entertainment that is enjoyed worldwide. Drivers from all over the world gather to compete in competitions. These competitions drive innovation for vehicle technologies and increase driver skills. Simulators provide drivers with a safe, cost-effective, and efficient way to hone their skills further. This paper introduces the problem of finding an optimal racing line and describes the simulation environment used to conduct our testing. We characterize and simulate two logic controllers using The Open-Source Racing Simulator (TORCS). The collected lap times and vehicle damage are used in a fitness function analyzed by a genetic algorithm (GA) to search for an optimal solution. These results are compared between known starting parameters, a randomized set, and the built-in TORCS simulation drivers. Finally, we conclude with a summarization of our findings and recommendations for future research.

CCS CONCEPTS

• **Computer systems organization** → *Real-time systems*; • **Computing methodologies** → **Vagueness and fuzzy logic**.

KEYWORDS

Genetic algorithm, fuzzy logic, single-objective optimization, minimization, simulation, racing

1 INTRODUCTION

When it comes to popular live television broadcasts worldwide, racing is often up there as one of the most popular entertainment options for viewers. Major sporting events such as the Daytona 500 and Indianapolis 500 have been ranked amongst the top 100 sporting events of 2021 [9]. Its viewership has brought professional racecar drivers worldwide, a plethora of sponsorship opportunities, and a network of funding between teams, stakeholders, and organizations. Safety has also been advanced across racing organizations, which has led to stricter rules and regulations amongst teams. Mixing this with the surge in manufacturing costs, race teams have started to resort to better, more cost-effective ways to save money. One of these ways is through simulation software. Proprietary software like iRacing [2], Assetto Corsa [1], TORCS [4], and others have been built with real-world physics, damage, and environmental models to simulate real-world behavior. Some users are professional racecar drivers looking to get an upper edge in the off-season or before their next race. For others, this might be the closest they get to racing. These simulators play a significant role in giving drivers an upper edge before the races or in the off-season and can reduce the costs of paying for race equipment. While this is nice, it can be challenging to start driving around without first understanding the track layout. Simulators provide aids, such as brake-assist and a "driving line" drawn over the track surface to point out optimal braking and acceleration zones to help drivers understand the track they will be racing on. This not only gives drivers who are learning the track a safer and faster line to follow, but it also avoids drivers from randomly picking a line that "feels right" but is not the overall most efficient way around the track. The driving line gets rid of the guessing game and instead introduces an optimal line for drivers to follow.

2 PROBLEM DESCRIPTION

Currently, simulation software either excludes driving line assistance or fails to provide the best driving line. Drivers using these lines are often confined to the limits of the line itself, where there are often opportunities for improvement since human domain experts often construct the driving lines from scratch. Additionally, it becomes tedious work when one must bring in manual testing for each track, car, and tire model update. It also has been noted that there is a lack of research that addresses the need for an automated design of racing lines using simple heuristics [6]. This paper introduces an evolutionary computation approach to finding an optimal racing line in a vehicle simulator. Our method would eliminate the need for manual testing and instead rely on a GA to determine the best line to follow. First, we will set up a driver that can navigate a track using its central steering, speed, gearing, and braking controls. The GA will be dealing with the speed and steering, where we create two fuzzy sub-controllers inside our driving agent to host our control logic

*Both authors contributed equally to this project.

for outputting the best speed and steering at every given instance around the tracks. Our fuzzy sub-controllers expect parameters that determine the target speed and steering to the driver. These are the parameters that we will be applying to our GA to find the best values that satisfy the following goal: driving fast and driving safely. We will be considering damage when a driver contacts any wall on the track. We will also not be considering the damage from drivers contacting other vehicles in our tests. The vehicles will be transparent to each other, allowing us to test 10 vehicles in one generation of testing as if they were alone on the track. We decided to test with ten vehicles to speed up our genetic algorithm evaluation process. It also gives us an overall view of the progression of our population members in each generation. This progression is changed through the works of our genetic algorithm, which is responsible for adjusting the parameters of our membership functions. Our membership functions are trapezoidal in shape, which defines the boundaries for our controllers' fuzzy rules. Through the evaluation process, the genetic algorithm will ultimately return a solution with the best lap times with minimal car damage.

3 SIMULATION AND TESTING ENVIRONMENT

The following sections detail the simulation and testing environment used in analyzing the GA and the fuzzy logic controllers. All the software execution was in an Ubuntu virtual machine but could also be performed on a Windows-based system. Details of the installation methods are documented in our source code repository, the TORCS [4] project website, the patched version of TORCS with the Simulated Racing Championship (SCR) [11], and the ROS [3] project websites.

3.1 TORCS

Testing for an optimal racing line requires an open track, vehicle, and a driver to gather the needed data. However, many simulators do not provide open-source access to develop or expand upon their proprietary software. The first step would be to get access to the necessary data to test across many generations. One such project that gives us this direct access to their simulation software is TORCS, a multi-platform car racing simulation that is ideal for artificial intelligence research.

A modification of TORCS was created to host is an international Simulated Racing Championship (SCR) competition held for major conferences in the field of Evolutionary Computation and Computational Intelligence, and Games [10] [12]. Competitors would design a controller for a vehicle and compete against unknown tracks and other drivers. There is a number of sensors provided that provide information to the vehicles. The authors of [10] have the sensors' descriptions, and value ranges well documented.

The TORCS application has one major drawback: the game was a stand-alone application. Being a stand-alone application brings the complication that races are not in real-time, execution is blocking, i.e., one bot could stall out of the rest of the execution chain. There is also no separation between the bots and the racing engine. The creation of the SCR patch provided a client/server interface between the game engine and the bots. The SCR patch solves the problem with blocking execution as each bot is processed on a separate computer using User Datagram Protocol (UDP) connections to communicate data between the game engine and the bot. The SCR patch also provides a game-level system tic, that is roughly every 20ms (or 50 Hz), which provides the clients with updated sensor information and for the bots to communicate back to the game engine [10]. Figure 1 depicts the interaction between the client controllers and the TORCS game engine and server bots.

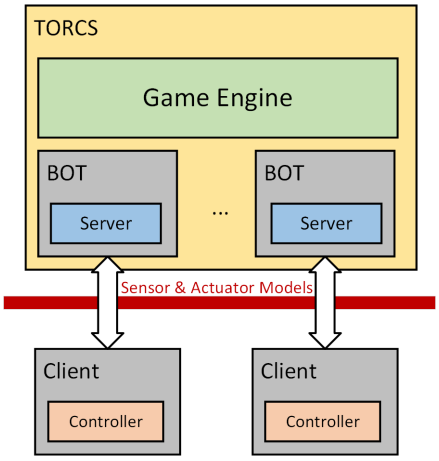

Figure 1: The architecture of the TORCS competition software with client nodes.

The TORCS application and SCR patch were modified to test the fuzzy logic controllers and the genetic algorithm. Typically, vehicles contacting obstructions such as walls and other vehicles cause damage to the vehicle. After receiving enough damage, the vehicle is removed from the race simulation. As stated in the problem statement, vehicle-to-vehicle damage was not considered for this project. This is where we had to go into the simulation's source code and turn off the vehicle-to-vehicle damage. The encoding of the values to be considered for the genetic algorithm was also limited to values that could be modified with the client interface that is communicating between the bots (controllers) and the game engine. Otherwise, changing other parameters of the game would require recompilation of the application

and restarting the simulation for each population member or generation.

3.2 ROS

In addition to the modifications created with the SCR patch for TORCS, the project also uses a client interface that enables the Robotic Operating System (ROS) to communicate to the game engine as a bot. ROS is a framework used primarily for robotic software but can be used for virtually any project outside of robotics. The goals of the design of the ROS framework can be summarized as peer-to-peer, tools-based, multi-lingual, thin, and free and open-sourced [7].

The peer-to-peer part of the framework is naturally an excellent fit for the needs of the genetic algorithm that is tested within this paper. It provides the ability to create multiple clients hosted on the same computer or executed from separate computers within the same network. If the processing is overburdening the host machine, a single driver or a set of drivers can be offloaded to another computer. The multi-computer processing has virtually no impact provided there are no network limitations that prohibit the remote nodes from sending and receiving the information they require to the host machine on the network.

Nodes, or software modules, within the ROS framework are processes that perform computation [7]. Each TORCS client shown in Figure 1 is a ROS node for this project. The nodes within the network communicate with each other with messages. A message is a strictly typed data structure that supports standard primitive types, such as integer, floating-point, boolean, strings, etc. Messages can contain structures formed by other messages, and arrays [7]. The messages are transmitted by publishing them to a provided topic. Any node interested in the data would need to subscribe to the message to view the information transmitted by the other node. Unlike some communication networks, the nodes have no idea of another node's existence.

3.2.1 ROS Message Types. Besides the messages provided from the sensor information defined in [10] and the ROS client, the following messages were also created to allow communication between the TORCS simulator game engine and the client nodes. Figure 2 shows the communication network for the simulation of the genetic algorithm node and one driver node. When multiple drivers are simulated, the grouping /torcs_ros$X$ (X is defined as a value between 1 and 10) would be duplicated for the number of drivers in the simulation. The node /torcs_ros_client_node1 provided the UDP communication interface between the driver (bot in Figure 1) and the game engine and other ROS nodes [11]. The client publishes all the sensor information defined in [10], but in this project, only three of the topics /torcs_ros$X$/speed, /torcs_ros$X$/sensors_state, and /torcs_ros$X$/scan_track are subscribed. The node /torcs_ros$X$/torcs_ga1 is the driver that controls the acceleration, braking, steering, clutch, and gear shifting. All of these values are transmitted by the /torcs_ros$X$/ctrl_cmd topic, which is defined by [11]. The driver node publishes and subscribes to the message, so it always has the previous values transmitted when new values are calculated. There is one signal node called /torcs_ga_alg, which contains the GA definition and evaluation functions.

The following sections describe the custom messages that were created for this project and the analysis of the genetic algorithm.

Table 1: *DriverParams* **message definition.**

Datatype	Name
uint8[]	newParams

3.2.1.1 Driver Parameters. The *DriverParams* message provides the configuration for the fuzzy logic controllers defined in a later section as calculated by the genetic algorithm. The definition provides an array, or list, of values that the subscribing node can receive, unpack, and apply to their controllers. The genetic algorithm node transmits the *DriverParams* message after evaluating the current generation to each driver node with new controller values. The new values are to be applied to the driver nodes to prepare them for the next generation.

Table 2: *LapStatus* **message definition..**

Datatype	Name
Header	header
uint8	nodeId
float64[]	lapTimes
float64	Damage
float64	maxSpeed
uint8	lapsCompleted

3.2.1.2 Lap Status. The *LapStatus* message provides information from the client nodes to the genetic algorithm node at the game tic rate of 20ms (50Hz). The message's header is a ROS-specific data type of ROS that contains a sequence identification and timestamp of the message. While this is not required, it helps debug to ensure that messages are transmitted, and nodes are not receiving stale data. The *nodeId* provides the unique identifier of the node. The *lapTimes* is an array of the completed lap times for the driver. The *damage* provides the amount of damage the vehicle has received by contacting the walls of the track. The *maxSpeed*

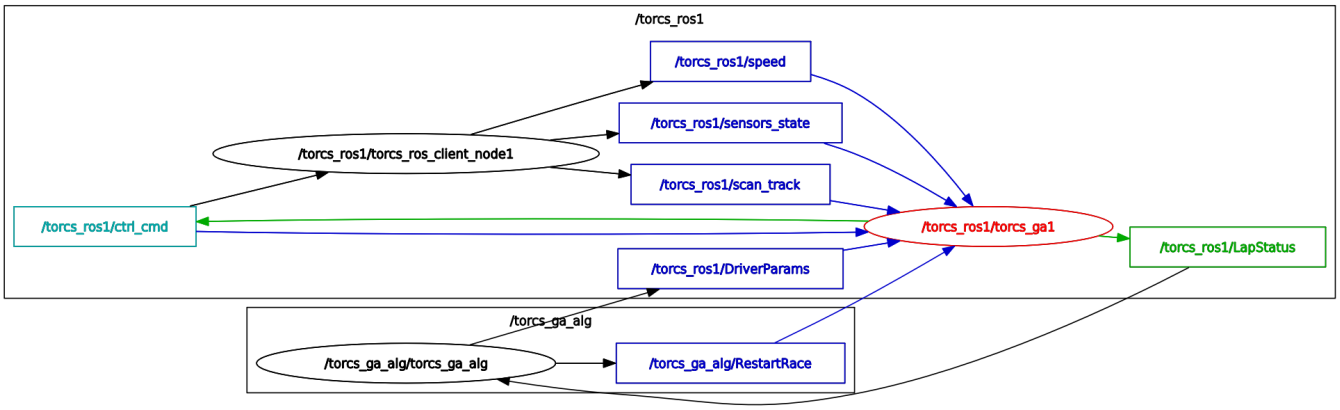

Figure 2: ROS topics for the genetic algorithm node and a single driver node.

provides the maximum speed (in KPH) of the driver over the entire race. Finally, the *lapsCompleted* provides how many laps were completed by the driver node. This information is sent from the driver node to the genetic algorithm node for fitness function processing for the current generation.

Table 3: *RestartRace* **message definition.**

Datatype	Name
Header	header
uint8	mode

3.2.1.3 Restart Race. The *RestartRace* message indicates to the driver nodes that the simulation will restart. The genetic algorithm processing node provides it, and all the driver nodes are subscribed to it. When the mode equals a value of 1, the driving nodes prepare for the restart by no longer updating their *LapStatus* message and clearing their local data. When the mode bit is set back to 0, the TORCS race is restarted, placing all the drivers back to the start of the race with no damage, lap times, or any other information from the previous generation. During this time, the new controller values have been provided and are available for the driver node to use.

3.3 Fuzzy Controllers

There are 19 sensors provided by the SCR that provide the distance from the vehicle's center to the edge of the track. The sensors provide a range of -90 to 90 degrees from the vehicle's left side to the right side. Figure 3 displays a screenshot of TORCS with the information from the sensors overlayed on the image at the moment the screenshot was taken. The red and green axis markers and the red dots are created from the RViz tool built-in with the ROS distribution. The axis origin would be the center of the vehicle, and the distance the points are away from the origin is the distance reading of the sensors to the track's edge. The image in Figure 3 is a screenshot of TORCS and RViz captured at the same time. The two images were combined to create a live view of the measurement. The yellow lines were added as a reader's guide to show the sensor's approximate path from the vehicle's center. The front sensors' measurements are out of frame but are measuring the contour of the curve that the vehicle is currently navigating. These sensors are used in the two fuzzy controllers that were implemented to control the vehicle's speed and steering.

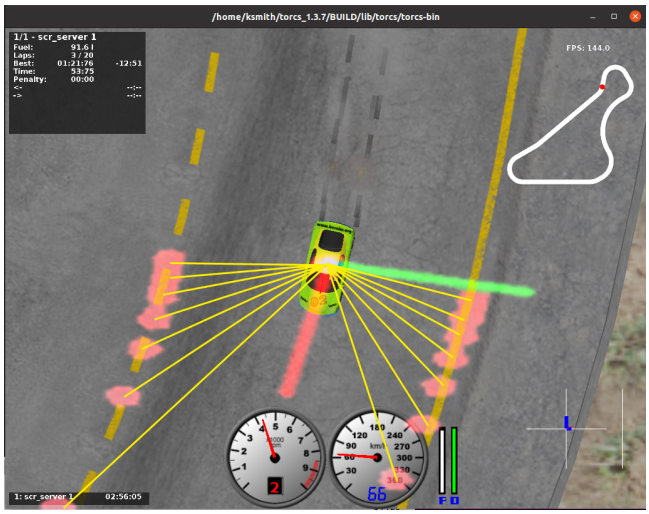

Figure 3: SCR Sensors overlayed on vehicle.

3.3.1 Speed Controller. The speed controller determines the optimal target speed of the car around curves and on a straight segment of the track. This controller has two primary goals: drive fast and drive safe. This is determined by

the genetic algorithm's fitness function that will be described in later sections. These sensors are used to measure the max distance from the vehicle to the edge of the track. Specifically, we are using the 0 degrees (*Front*) and the left and right side angle sensors at intervals ±5 and ±10 degrees of the vehicle's center (*M*5 and *M*10, respectively). The fuzzy logic uses trapezoid membership functions, bounded by Eq 1 and defined by Eq 2, that determine if a sensor is either *High*, *Medium* (or *Mid*), or *Low*. The fuzzy logic rules for this controller are defined in table 4. This is a similar design to the controller rule set the authors in [12] defined, but controller nodes are unique in their implementation and handling of events. These base rules were also created to model the behavior of a human driving expert. In other words, the rules are designed to maximize the car speed depending on the distance to the track border. The speed controller is also responsible for clutch and steering controls. These were based on the controls found in the ROS client implementation located in the source code of this project [11]. The target speed rule for this controller is $Target_{Speed} = [280, 240, 220, 180, 120, 60, 30]$.

$$0 = x_0 \leq x_1 \leq x_2 \leq x_3 \leq x_4 \leq x_5 \leq x_6 \leq x_7 = 200 \quad (1)$$

$$\begin{aligned}
S_{Low}(x) &= \begin{cases} 1, & x_1 \leq x \leq x_2 \\ \frac{x_3 - x}{x_3 - x_2}, & x_2 \leq x \leq x_3 \\ 0, & x > x_3 \end{cases} \\
S_{Medium}(x) &= \begin{cases} 0, & x \leq x_2 \\ \frac{x - x_2}{x_3 - x_2}, & x_2 \leq x \leq x_3 \\ 1, & x_3 \leq x \leq x_4 \\ \frac{x_5 - x}{x_5 - x_4}, & x_4 \leq x \leq x_5 \\ 0, & x > x_5 \end{cases} \quad (2) \\
S_{High}(x) &= \begin{cases} 0, & x \leq x_4 \\ \frac{x - x_4}{x_5 - x_4}, & x_4 \leq x \leq x_5 \\ 1, & x > x_5 \end{cases}
\end{aligned}$$

Table 4: Speed controller fuzzy logic ruleset.

Condition	If True
$Front = High$	$Target_{Speed}[0]$
$Front = Med$	$Target_{Speed}[1]$
$Front = Low$ and $M5 = High$	$Target_{Speed}[2]$
$Front = Low$ and $M5 = Med$	$Target_{Speed}[3]$
$Front = Low$ and $M5 = Low$ and $M10 = High$	$Target_{Speed}[4]$
$Front = Low$ and $M5 = Low$ and $M10 = Med$	$Target_{Speed}[5]$
$Front = Low$ and $M5 = Low$ and $M10 = Low$	$Target_{Speed}[6]$
$Front = MAX$ or $M5 = MAX$ or $M10 = MAX$	$Max_{Speed} = 300$

3.3.2 Steering Controller. The steering controller determines the optimal steering angle and determines the target position of the vehicle on the track. The same sensors are used in the speed controller: *Front*, *M*5, and *M*10. The steering actuator in the client expects a value between -1 and 1 (full left or full right, respectively) as per the definition found in [10]. Like the speed controller, the design closely follows previous designs documented in [12] and [11] and the ruleset is summarized in Table 5. The steering value set is defined as $Target_{Steering} = [0, 0.25, 0.5, 1]$

Table 5: Steering controller fuzzy logic ruleset.

Condition	If True
$Front = High$	$Target_{Steer}[0]$
$Front = Med$ and $M10 = High$	$Target_{Steer}[1]$
$Front = Med$ and $M5 = Med$ and $M10 = Med$	$Target_{Steer}[1]$
$Front = Med$ and $M5 = Low$ and $M10 = Med$	$Target_{Steer}[2]$
$Front = Low$ and $M10 = High$	$Target_{Steer}[2]$
$Front = Low$ and $M5 = Med$ and $M10 = Med$	$Target_{Steer}[3]$
$Front = Low$ and $M5 = Low$ and $M10 = Med$	$Target_{Steer}[3]$

4 GENETIC ALGORITHM

The project utilized the single-objective genetic algorithm to optimize our controllers' parameters. This algorithm is implemented from the Pymoo module, and it implements a basic $(\mu + \lambda)$ genetic algorithm [5]. The $(\mu + \lambda)$family typically initializes a randomly uniform population and, after each generation, creates λ offspring [13]. Figure 4 shows a flow diagram of how the algorithm is processed by the various operators. Details of the operators are discussed in the following sections.

4.1 Encoding

The chromosome for the genetic algorithm is made up of three parts. Each part represents one of the three sensors: *Front*, *M*5, and *M*10, as discussed earlier. Each sensor consists of six data points that describe the shape of the trapezoid functions that define the values of *Low*, *Medium*, and *High*. These points must match the constraints defined in Eq 1. For points, x_0 and x_7 are not considered in the chromosome encoding for our GA. This is because the sensors have a limited range of 0 to 200 [10]. Since the endpoints are fixed, if the sensor reading was a low value, it needed to ensure that the *Low* reading value would always be read as a logical *True*. So the points x_1 and x_6 were also always defined as 0 and 200, respectively. This provided no gaps at the endpoints; the trapezoids for the *Low* and *High* sensor could be as big or small as needed.

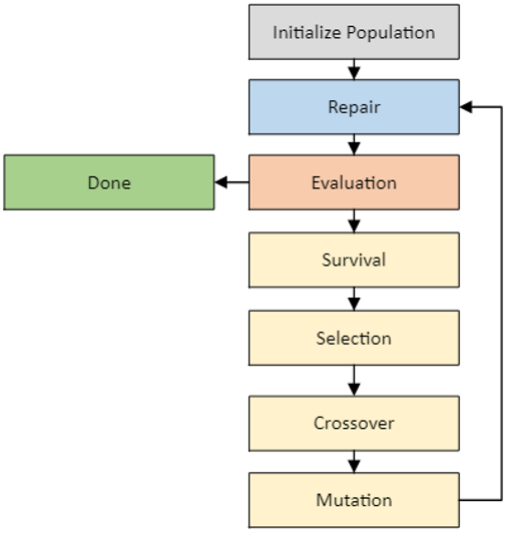

Figure 4: Genetic Algorithm Process.

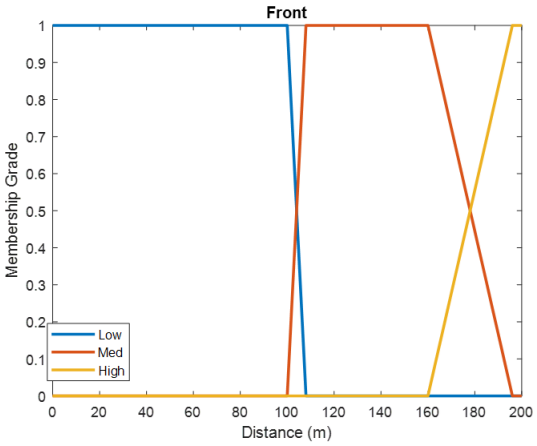

Figure 6: Example for the *Front* sensor trapezoid membership functions.

Figure 5 illustrates a summary of the previous paragraph. The patterned regions would be the fixed endpoints (values 0 and 200 respectively), and the middle six points are the values in the part of the chromosome. The colors represent the fuzzy value that would be represented if the input value falls into that region: The blue band represents the *Low* value, the red band represents the *Medium* value, and the yellow band is the *High* value. This is repeated for the other two sensors, M5 and M10, in the chromosome as well. By selecting random values for each of the data points, the membership functions can be visually analyzed as shown in Figure 6 by following the formulas defined in Eq 2.

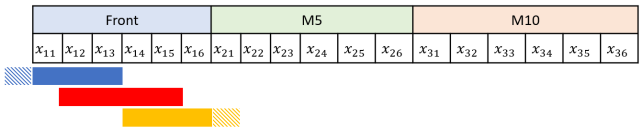

Figure 5: Bit encoding for each sensor.

4.2 Initialization

The population was initialized by two different methods in our testing. The first method was to take parameters of a working set as the base of the population. We keep the base individual as our first population member, then mutate the other nine individuals using the built-in polynomial mutation in the Pymoo library [5]. The mutation rate for this change was 30%, with an η value of 3. This method, starting from a base chromosome, gave us at least one working population member in our first generation to ensure a finishing driver. It also gave us a working starting point so that we could find an optimal solution faster. The other method was using the random integer sampling function provided in the Pymoo library [5]. This method allowed for a better exploration of other solutions instead of relying on a base individual to start. These methods give us the ability to analyze our algorithm's performance from evolution and genetic improvement point of view.

4.3 Survival and Selection

To determine what population members become parents, we will be going with a tournament selection, defaulted to size 2 for the Pymoo library. Since the size of our tournament selection is 2, the default function is a binary tournament selection. This means that for each generation, we will be selecting randomly two individuals to compete in the tournament. The individual that has the highest fitness will be the one selected to move on to the next generation.

4.4 Crossover

For the crossover operator, we went with an integer simulated binary crossover. This will produce new parameters in the offspring, also inheriting the parents' old parameters. Using crossover helps with keeping good characteristics from our selected parents, passing them on to further generations [8]. This will get our GA to converge to reasonable solutions (Exploitation). The determined crossover rate was set to 0.7. This rate determines the number of times a crossover occurs for the chromosomes in one generation. To keep some variation, we did not raise the rate any further.

4.5 Mutation

The mutation operator uses integer polynomial mutation for selecting genes to mutate in our chromosomes. Mutation plays a significant role in making our overall search efficient, allowing for exploration across our search space so that we do not converge to a local optimum. With this in mind, we chose a mutation rate of 0.3. The rate determines how many of our chromosomes we would mutate in one given generation. Increasing this rate would bring in too much diversity. Likewise, decreasing the rate below our set value was causing premature convergence [8].

4.6 Repair

The repair function ensures that the fuzzy logic parameters meet the constraint in Eq 1 for each sensor. If the values do not follow the constraint, the value is replaced with a random integer value between the points that surround this value. This ensures that each node is receiving a properly defined set of values. However, this does have some impact on the size of the search space for new values, but this is a necessary change, or else the controllers will not behave correctly. If the constraint is met, the genetic algorithm has complete control over the values from the tournament, crossover, and mutation phases.

4.7 Fitness Function

We went with a fitness function that establishes our previously mentioned goal: drive fast and drive safely. In order to establish this, we define a function that takes in each driver set of lap times in the current generation, along with the damage they received. Below in Eq 3 we include both of these parameters, taking the average of the lap times, $\overline{L}$, and adding it to the driver's damage, D. Both parts should be minimized as much as possible to achieve optimal results. The driver with the lowest score would be the "fittest" individual in the population as they would have had the fastest lap. We also set a scalar variable, α, that can be adjusted to give more or less weight to the importance of achieving lower lap times compared to achieving lower damage. The fitness function will be residing in the evaluation step of our genetic algorithm once the three laps for the current generation have been completed. Once the laps are complete, we collect all the lap times and damages from each driver to send to our fitness function.

$$F = D + (\alpha\overline{L}) \tag{3}$$

In this case, we are treating the damage as a time penalty, with the damage value being added to the lap time on a scale of seconds. However, another approach could be to change the genetic algorithm to a multi-objective optimization problem and use the lap time and the damage as the two distinct objectives to optimize. The decision to stay with a single-objective optimization problem came from having zero damage on the vehicles. The overall fitness rating would severely penalize even a slight tap on the wall.

5 SIMLUATION RESULTS

Our simulation was initialized and performed with a known driver configuration from experimenting with the design of the driver nodes. The known configuration was duplicated for the population size (10) and was then mutated by the same configuration used within the genetic algorithm. This provided a good starting point to start the drivers for the simulation. However, the mutation also provided some population diversity instead of simply copying the same driver configuration for all ten members. Consequently, this makes the problem a genetic improvement instead of a genetic optimization. The simulation was repeated with the same seeding, but a randomly generated integer sample replaced the initial population. The additional testing goal was to ensure that the simulation results converged without any unintentional bias. The repair function processed the randomly selected members before the first-generation evaluation. Table 6 shows the results of these tests. The known seed values were selected arbitrarily, where a random value was applied for the pre-populated set and the randomly generated sample. Figure 7 shows how the optimal fitness values progressed over the 100 generation tests, while figure 8 shows the average over the 100 generations.

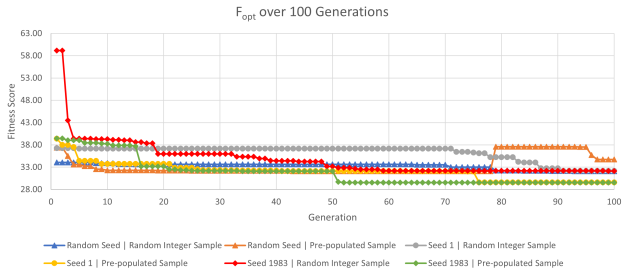

Figure 7: F_{opt} values per generation for 100 generation test runs.

In comparison with the results we obtained, we also performed single races with the built-in drivers driving the exact vehicle in our tests. However, there were complications when running the built-in drivers as to our GA testing. The built-in drivers would load a specific amount of fuel proportionally to the number of laps configured for the race. If the race was configured for over 100 laps, the fuel tank would be full, but only three laps the fuel take would be just about empty. The difference in fuel gave the built-in drivers a boost as there was over a second in lap speed time just based on the

Table 6: GA Best solution results

Test Case	Seed	Num. of Generations	Initial Pop. Set	Best Solution Value [F]
1	1	50	Pre-populated Set	29.58
2	1	50	Random Integer Sample	37.39
3	1	50	Pre-populated Set with 70% mutation rate	34.44
4	1	50	Random Integer Sample with 70% mutation rate	32.44
5	1983	50	Pre-populated Set	32.17
6	1983	50	Random Integer Sample	31.99
7	1	100	Pre-populated Set	29.59
8	1	100	Random Integer Sample	32.10
9	1983	100	Pre-populated Set	29.54
10	1983	100	Random Integer Sample	32.14
11	Random	100	Pre-populated Set	32.02
12	Random	100	Random Integer Sample	32.09

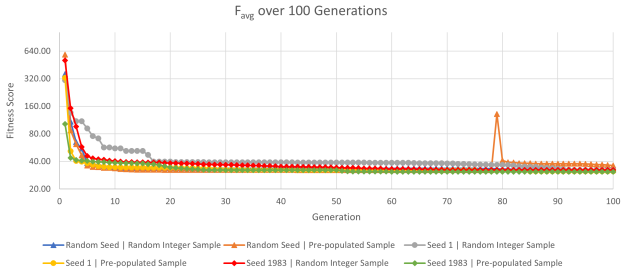

Figure 8: F_{avgt} values per generation for 100 generation test runs.

different fuel amounts. In order to have a fair comparison, the built-in drivers needed to be measured with the same amount of fuel. The built-in drivers were placed in a race of 100 laps and ran ten laps. The average of those ten laps is shown in table 7.

Table 7: Average lap times for built-in TORCS drivers

Driver	Average Lap Time (10 laps)
Berniw 3	29.66
Olethros 3	32.00
Bt 3	31.78
Tita 3	29.67
Inferno 3	29.67
Lliaw 3	29.67

6 CONCLUSIONS AND FUTURE IMPROVEMENTS

In this work, we presented a genetic algorithm approach towards optimizing a vehicle to navigate around a racetrack, choosing the best racing line to achieve the fastest lap and smallest damage. We chose TORCS as the simulator and optimized an already existing driver implemented in the simulator. We also modified the driver to include two fuzzy sub-controllers to calculate the target speed and steering.

The obtained results are very promising, as our driver was able to successfully reduce its lap time through the generations and decrease overall damage. Furthermore, we compared our obtained driver's results with the other AI drivers included in the TORCS application. Our results indicate that we are right in line, if not better than the drivers. Figure 9 shows the *Front*, $M5$, and $M10$ sensor configurations at the first generation and displays them with the configurations after the 100th generation.

The testing included two known seed values and an entirely randomly selected seeding. All three seeds were tested with a known staring configuration and a randomly generated population set for the initial driver configuration. The goals of having a known set with a randomly generated set were to ensure that we have some diversity in our population members and to keep the problem as an optimization instead of becoming a genetic improvement problem. Also, we increased the mutation probability from 30% to 70% for two tests, along with a high crossover rate of 70%. As mentioned in the text, this was to increase the diversity of the population and widen our search scope. However, the run of the 50 results showed very similar results to our other testing. Most likely, the cause is that we are looking for diversity in a limited search space. The population members need to follow a constraint that each of the six points needs to be greater than or equal to the previous point and fit within a value range of 0 to 200. This limitation means that our existing search space for diversity is going to be limited in the number of combinations we can test.

While the results are already satisfactory, the results can be both improved and extended. One of the mechanisms we implemented was giving the car the best angle leading into the turn. This mechanism makes sure that the driver stays close to the wall while on a straight segment of the track. Keeping it in this boundary means that the driver made minor steering adjustments not to leave our defined boundary. This caused the driver to wobble slightly. By adding a PID controller, this strategy could still work more smoothly. As mentioned above, another improvement could be made in dealing with boundary control. The boundary could be refined to consider all tracks, especially the ones where the wall is closer to the track border, to avoid the additional damage received on the straightway. Lastly, extending the project to consider fuel, tire friction, and other vehicle dynamics in the GA calculation could help further lower our lap time and be considerably better across simulators with dynamic track temperatures and environmental considerations.

ACKNOWLEDGMENTS

To Dr. Erik Goodman, for sharing his insights and working with us in defining and implementing our genetic algorithm.

REFERENCES

[1] 2001. Assetto Corsa Competizione. https://www.assettocorsa.it/competizione/

[2] 2021. iRacing. https://www.iracing.com/

[3] 2021. ROS. https://ros.org/

[4] 2021. TORCS. http://torcs.sourceforge.net/index.php

[5] Julian Blank and Kalyanmoy Deb. 2020. Pymoo: Multi-Objective Optimization in Python. *IEEE Access* 8 (2020), 89497–89509. https://doi.org/10.1109/ACCESS.2020.2990567

[6] Luigi Cardamone, Daniele Loiacono, Pier Luca Lanzi, and Alessandro Pietro Bardelli. 2010. Searching for the optimal racing line using genetic algorithms. *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG2010*, 388–394. https://doi.org/10.1109/ITW.2010.5593330

[7] Steve Cousins. 2010. Welcome to ROS topics. *IEEE Robotics and Automation Magazine* 17 (3 2010), 13–14. Issue 1. https://doi.org/10.1109/MRA.2010.935808

[8] Kalyanmoy Deb, Karthik Sindhya, and Tatsuya Okabe. 2007. Self-adaptive simulated binary crossover for real-parameter optimization. *Proceedings of GECCO 2007: Genetic and Evolutionary Computation Conference*, 1187–1194. https://doi.org/10.1145/1276958.1277190

[9] Jon Lewis. [n.d.]. Most-watched sporting events of 2021 so far - Sports Media Watch. https://www.sportsmediawatch.com/2021/07/50-most-watched-sporting-events-2021-so-far/

[10] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. 2013. Simulated Car Racing Championship: Competition Software Manual. (4 2013). http://arxiv.org/abs/1304.1672

[11] Florian Mirus. 2018. Repo for interfacing TORCS with ROS. https://github.com/fmirus/torcs_ros

[12] Mohammed Salem, Antonio Miguel Mora, Juan Julian Merelo, and Pablo García-Sánchez. 2018. Evolving a TORCS Modular Fuzzy Driver Using Genetic Algorithms. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10784 LNCS, 342–357. https://doi.org/10.1007/978-3-319-77538-8_24

[13] Furong Ye, Hao Wang, Carola Doerr, and Thomas Bäck. 2020. Benchmarking a $(\mu + \lambda)$ Genetic Algorithm with Configurable Crossover Probability. (6 2020). https://doi.org/10.1007/978-3-030-58115-2_49

A SOURCE CODE

The source code for this project can be found at the public GIT repository link below. The project is structured as a set of ROS nodes and should be cloned to the user's workspace of the ROS framework. Once the nodes are compiled, the ROS launch scripts can be used to start the GA node and the driver nodes. More specific details on downloading, installing, and executing can be found in the *ReadMe* file within the GIT project.

- Source code URL:
  - https://bitbucket.org/kjslabs/torcs_ga/src/master/
- Pymoo 0.5.0
- ROS 1 - Noetic Ninjemys
- Python 3.8

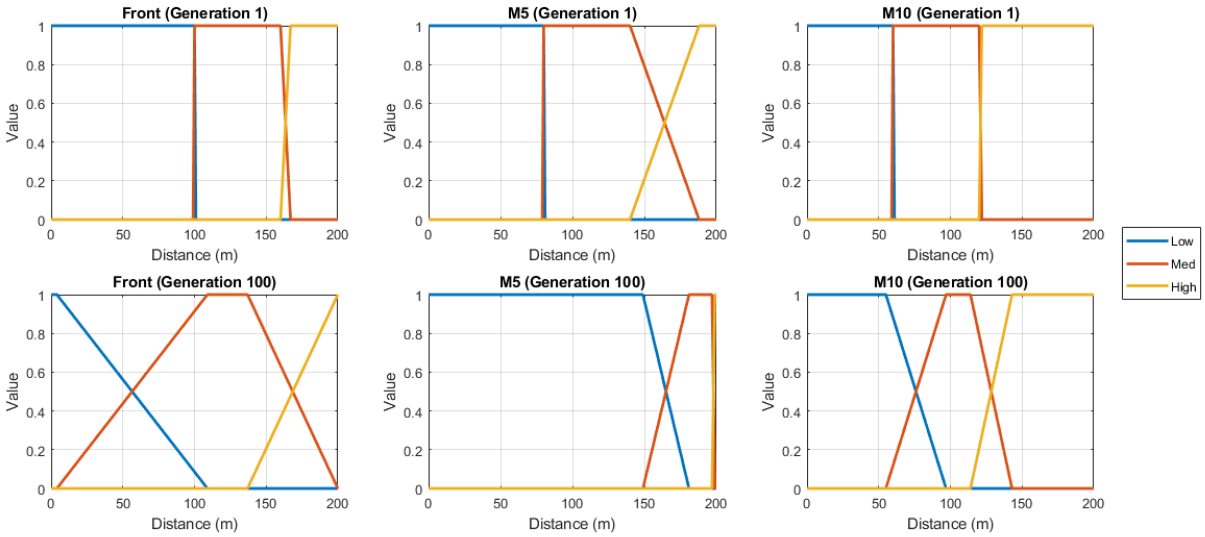

Figure 9: X values after the 1^{st} and 100^{th} generation for the 1983 seed, pre-populated population