

Genetic Algorithm

Term Project

Differential Evolution for tuning
Complex Controllers
for Autonomous Driving

Algorithm used - Differential Evolution
Github Repository Link

Group Members

- 1.** R. Raghav - 18IM10035
- 2.** Abhay Shukla - 18IM10002

Introduction

Genetic algorithms is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover and selection. John Holland introduced genetic algorithms in 1960 based on the concept of Darwin's theory of evolution, and his student David E. Goldberg further extended GA in 1989.

In most of the genetic algorithms, a set of randomly generated solutions, or a random set of values is taken. There is a function which is to be either maximised or minimised. The solution set is, as mentioned above, mutated, crossed-over and better solutions among them are selected.

Problem Statement

The main aim of our project is to compute the parameters of Stanley and pure pursuit based trajectory tracking controllers for autonomous vehicles using the differential evolution algorithm. The vehicle is controller by 2 combinations of lateral and longitudinal controller

1. PID with Stanley
2. PID with Pure-pursuit

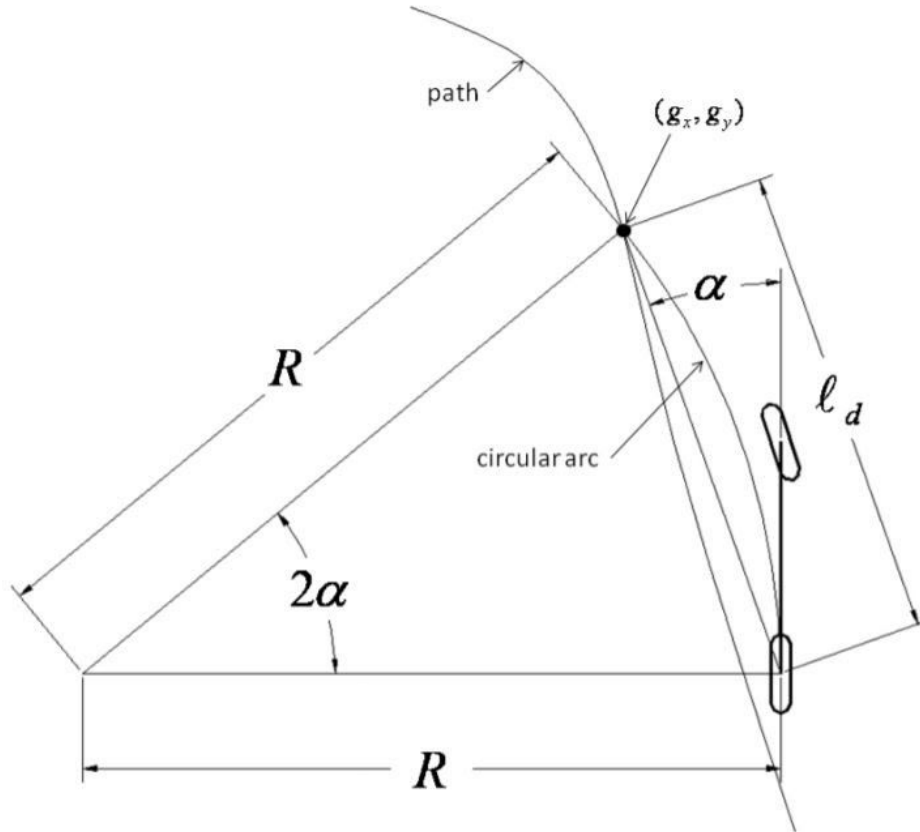
So briefly we are trying to tune complex controllers for **fully autonomous driving scenarios**.

Path Tracking

Path tracking refers to a vehicle executing a globally defined geometric path by applying appropriate steering motions that guide the vehicle along the path. The goal of a path tracking controller is to **minimize the lateral distance between the vehicle and the defined path, minimize the difference in the vehicle's heading and the defined path's heading, and limit steering inputs to smooth motions while maintaining stability**

One of the most popular classes of path tracking methods found in robotics is that of geometric path trackers. These methods exploit geometric relationships between the vehicle and the path resulting in control law solutions to the path tracking problem. These techniques often make use of a look ahead distance to measure error ahead of the vehicle and can extend from simple circular arc calculations to more complicated calculations involving screw theory. This section will describe the geometric vehicle model most commonly used by these methods and two of these methods: **Pure Pursuit and the Stanley Method**.

Pure Pursuit



The pure pursuit method and variations of it are among the most common approaches to the path tracking problem for mobile robots. The pure pursuit method consists of geometrically calculating the curvature of a circular arc that connects the rear axle location to a goal point on the path ahead of the vehicle. The goal point is determined from a look-ahead distance ℓ_d from the current rear axle position to the desired path. The goal point (g_x, g_y) is illustrated in Figure. The vehicle's **steering angle** δ can be determined using only the goal point location and the angle α between the vehicle's heading vector and the look-ahead vector.

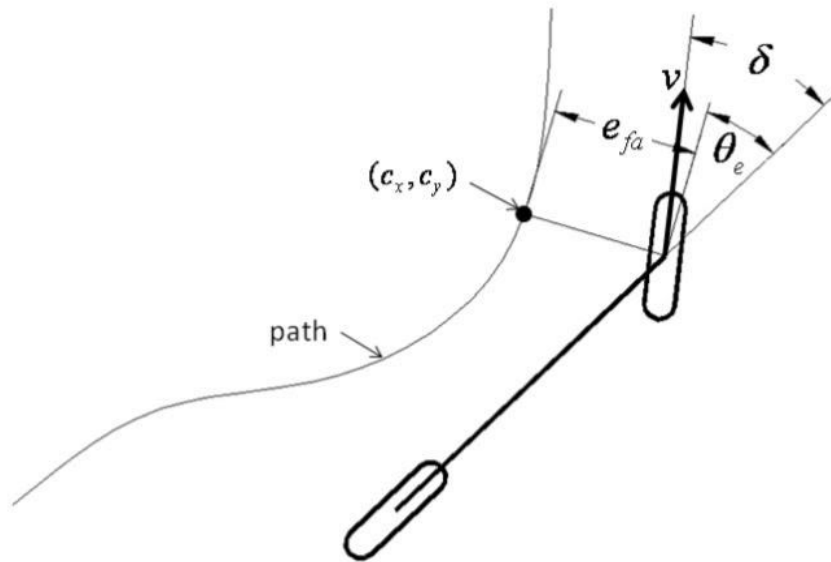
The pure pursuit control law is given as

$$\delta(t) = \tan^{-1} \left(\frac{2L \sin(\alpha(t))}{\ell_d} \right).$$

Here L is the Parameter that we are going to compute using Differential Evolution.

Stanley

The Stanley method is a nonlinear feedback function of the cross track error e_{fa} , measured from the center of the front axle to the nearest path point (c_x, c_y) , for which exponential convergence can be shown. Co-locating the point of control with the steered front wheels allows for an intuitive control law, where the first term simply keeps the wheels aligned with the given path by setting the **steering angle** δ equal to the heading error



$$\theta_e = \theta - \theta_p,$$

where θ is the heading of the vehicle and θ_p is the heading of the path at (c_x, c_y) . When e_{fa} is non-zero, the second term adjusts δ such that the intended trajectory intersects the path tangent from (c_x, c_y) at $k v(t)$ units from the front axle. Figure illustrates the geometric relationship of the control parameters. The resulting steering control law is given as

$$\delta(t) = \theta_e(t) + \tan^{-1} \left(\frac{k e_{fa}(t)}{v_x(t)} \right),$$

Here Ke is the parameter we are going to compute using differential evolution.

PID(Proportional-Integral-Derivative)

We are using PID to control the velocity of the car by calculating the accelerator input using the PID algorithm.

Proportional component:

The proportional term sets accelerator input that is proportional to the error in velocity. However, the end result is a velocity profile which oscillates around the reference velocity. The proportional coefficient (**Kp**) determines how fast the car oscillates(or overshoots) around the reference velocity.

Derivative component:

The derivative component uses a rate of change of error to reduce the overshoot caused by the proportional component. This derivative coefficient (**Kd**) term is used to optimize how far the car overshoots (also known as oscillation amplitude) from the reference velocity.

Integral component:

Over time, the velocity accrues errors due to systematic bias which could drive the car out of reference velocity, but not immediately. The integral component fixes this problem. As this component impacts the error over a period of time, the integral coefficient (**Ki**) should be carefully optimized in small steps as it has a large impact on the overall performance.

$$Out(t) = Kp e(t) + Ki \int e dt + Kd \frac{de}{dt}$$

One could manually tune the Kp, Ki & Kd values with trial and error but it would take a lot of time to get the car to drive around the track smoothly.

So we are using Differential Evolution to compute values of Kp , Ki , Kd.

Idea

The problem of tuning control algorithms is well known. Although mathematical methods are present for computing exact parameters, they are very computationally demanding and resource consuming. Our idea is to use differential evolution for computing the parameters efficiently for both the combinations of trajectory trackers. We are using Differential Evolution to compute

1. K_p, K_i, K_d, k_e (in PID and Stanley)
2. K_p, K_i, K_d, I_d (in PID and Pure-pursuit)

Differential Evolution

Differential Evolution Algorithm is a relatively newer genetic algorithm, quite efficient and popular. It was given by Storn and Prince. They decided that the algorithm should be able to work on real values. In evolutionary computation, differential evolution (DE) is a method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality, also referred to as fitness function. Such methods are commonly known as metaheuristics as they make few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. However, metaheuristics such as DE do not guarantee an optimal solution is ever found. These kinds of algorithms are run for many times, called iterations or generations, until a termination condition is met.

Differential Evolution is generally used in cases where it cannot compute the gradient of the fitness function. Generally, a multi-dimensional and real valued fitness function is used. This means that it is not necessary for the fitness function to be differentiable, as required by classic optimization methods. DE can therefore also be used on optimization problems that are not even continuous, are noisy, change over time, etc.

DE optimizes a problem by maintaining a population of candidate solutions and creating new candidate solutions by combining existing ones according to its simple formulae, and then keeping whichever candidate solution has the best score or fitness on the optimization problem at hand. In this way the optimization problem is treated as a black box that merely provides a measure of quality given a candidate solution and the gradient is therefore not needed.

The main steps involved in DE, during every iteration, are mutation, crossover and selection.

Some Terminologies


- Genes – The characteristics of an individual are called genes.
- Individuals – It is also referred to as chromosomes. It is the collection of all gene values in a particular set.
- Population – The collection of all individuals involved in the process.
- Fitness – The measure of quality of each individual is its fitness.
- Fitness Function – The function which is to be maximized or minimized is called fitness function.
- Generations – The iterations for which some steps are executed repeatedly on loop are called generations
- Mutation – The process of alteration of one or more gene values in an individual from its initial state is called mutation.
- Crossover – Crossover is the recombination of multiple individuals, on their genes.
- Selection – The process of selecting a subset of individuals from a population is called selection.

The Algorithm

Before diving in the steps of the algorithm, it is necessary to know its parameters. The parameters are

- Dimensions (n) – It is the dimensionality of the fitness function. In other words, it can be thought of number of genes, in an individual, or the number of variables in the fitness function
- Population Size (np) – It is the number of individuals in the population. Typically, the population is a little bit more than ten times the dimensionality of the function. Typically, $np \approx 10 * n$
- Mutation Constant (C_μ) – It is also referred to as differential weight. It determines the effect of mutation on an individual. Generally, the values of C_μ lie in the range 0.7-0.8
- Crossover Probability (P_c) – It is the probability which determines whether the crossover should occur at the particular stage. Typically, crossover probabilities are high, for example, in range of 0.7-0.9

Knowing the parameters is of utter importance. The combination of these values can affect the convergence of the algorithm. The choice of DE parameters can have a large impact on optimization performance. Selecting the DE parameters that yield good performance has therefore been the subject of much research.



The following steps gives an intuition of how the algorithm works-

1. Create a random population of the needed size. Lower and upper bounds for each gene value may be implemented and checked.
2. There may be many types of terminating conditions, for example, the number of generations, absolute error between fitness of the fittest individual between two generations, etc. While this terminating condition is reached, repeatedly do
 - a. Mutate the population, with rules which would be discussed later. The original population must be preserved for selection purposes.
 - b. Implement crossover, on the mutated population, between two individuals keeping in mind, the crossover probability (P_c)
 - c. Select between the original individual, or the “modified population” on basis of their fitness values
 - d. Update the population
3. The fittest individual of this population is the solution found out by the algorithm.

Our Use Case

This section covers the values of the parameters of Differential Evolution Algorithm, and the methodologies used by us in the project. It also covers the formula used in mutation.

The parameters values used are -

- Dimensions (n) – 4
- Population Size (np) – 40
- Generations - 100
- Mutation Constant (C_μ) – 0.3
- Crossover Probability (P_c) – 0.8
- Lower Bounds – -1 for each variable
- Upper Bounds – 100 for each variable

Mutation

Mutation is the main crux of the algorithm. It is this, which makes the algorithm unique. For every individual, we choose two different individuals, and use them in mutation.

For every individual 'i' in the population, we choose two other different individuals 'j' and 'k'. Let us consider these individuals as vectors P_i , P_j and P_k . Using C_μ described above,

$$P_{i\mu} = P_i + C_\mu(P_j - P_k) \quad \text{where } P_{i\mu} \text{ is the mutated child/individual of } P_i$$

By using C_μ as the mutation constant, we ensure **controlled perturbation** in this process. The main advantage of this kind of mutation is that it is **self-adaptive**.

Crossover

After mutation, comes crossover. In crossover, a cross between the individual and its mutated child occurs, i.e. in between P_i and $P_{i\mu}$. To implement the crossover with the crossover probability, a function is used which would return a random floating number between 0 and 1. This random number can be used as if this number is greater than the crossover probability, crossover at that particular gene would occur. Let $\text{random}()$ be the function which generates the random number.

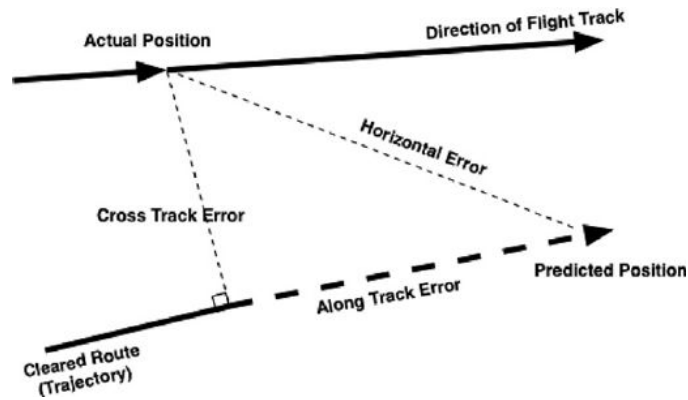
Mathematically, for every gene k in the parent(P_i) and its child($P_{i\mu}$),

$$P_{ik} = \begin{cases} P_{i\mu k} & \text{if } \text{random}() > P_c \\ P_{ik} & \text{otherwise} \end{cases}$$

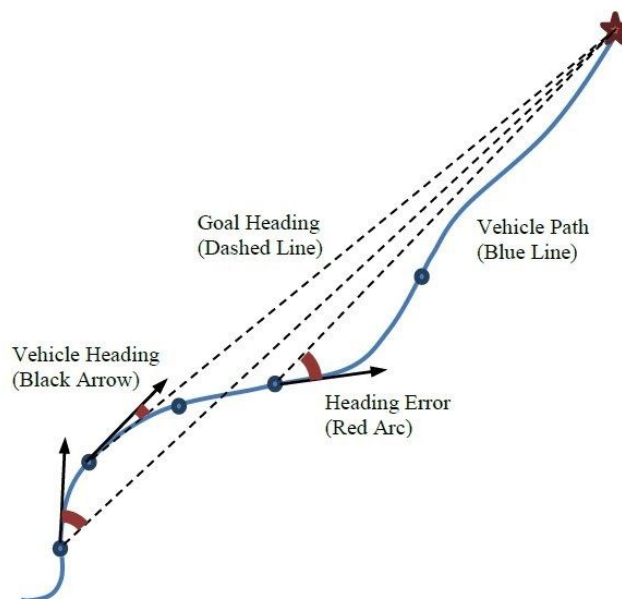
Fitness

Fitness function is used to summarise, as a single figure of merit, how close a given design solution is to achieving the set aims. In this case the fitness of an individual is determined by simulating it in a python environment for tracking a predefined trajectory. The closer the car moves along the desired path, the lesser is the fitness. Although the extent of quality of values is still a subjective question. We need to reduce it down to a number. We achieve this by using 2 quantities.

1. Cross track error - Minimum distance between path and vehicle.



2. Error in heading - Deviation between the car and the path.



At every instance of time we calculate both the quantities and add it the fitness value in a predefined proportion.

Simulation

We have made our own simulator for calculating fitness for reducing complexity and time taken by the algorithm. The complexity is reduced by using a linear bicycle model to imitate car dynamics. The linear bicycle model is described below.

$$x_{t+1} = x_t + v_t \cos(\psi_t) dt$$

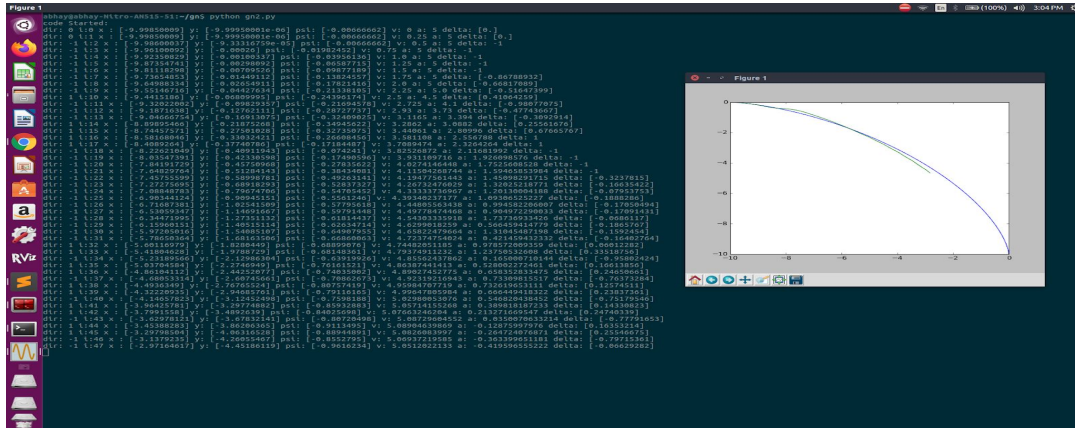
$$y_{t+1} = y_t + v_t \sin(\psi_t) dt$$

$$\psi_{t+1} = \psi_t + \frac{v_t}{L_f} \delta_t dt$$

$$v_{t+1} = v_t + a_t dt$$

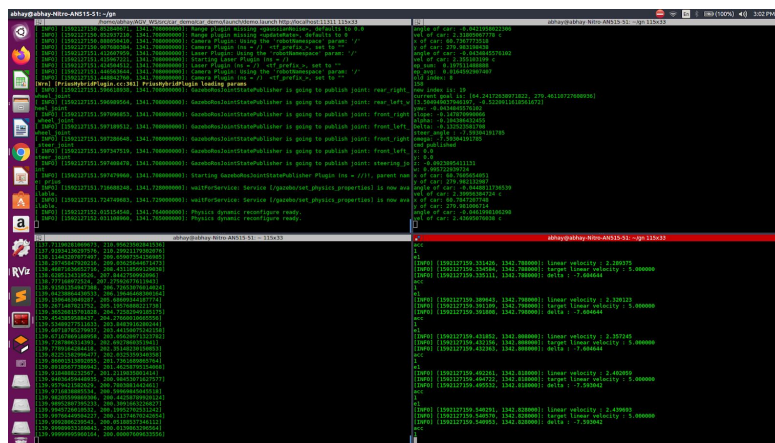
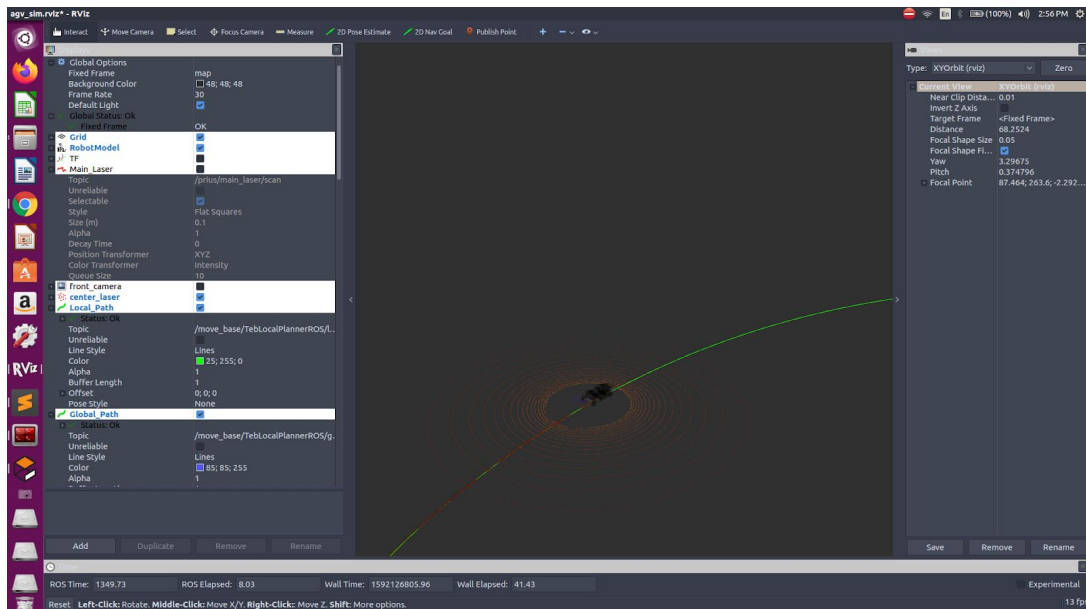
$$cte_{t+1} = f(x_t) - y_t + v_t \sin(e\psi_t) dt$$

$$e\psi_{t+1} = \psi_t - \arctan(f'(x_t)) + \frac{v_t}{L_f} \delta_t dt$$



This is the self written simulation setup to speed up the genetic algorithm

© 2006 The Authors

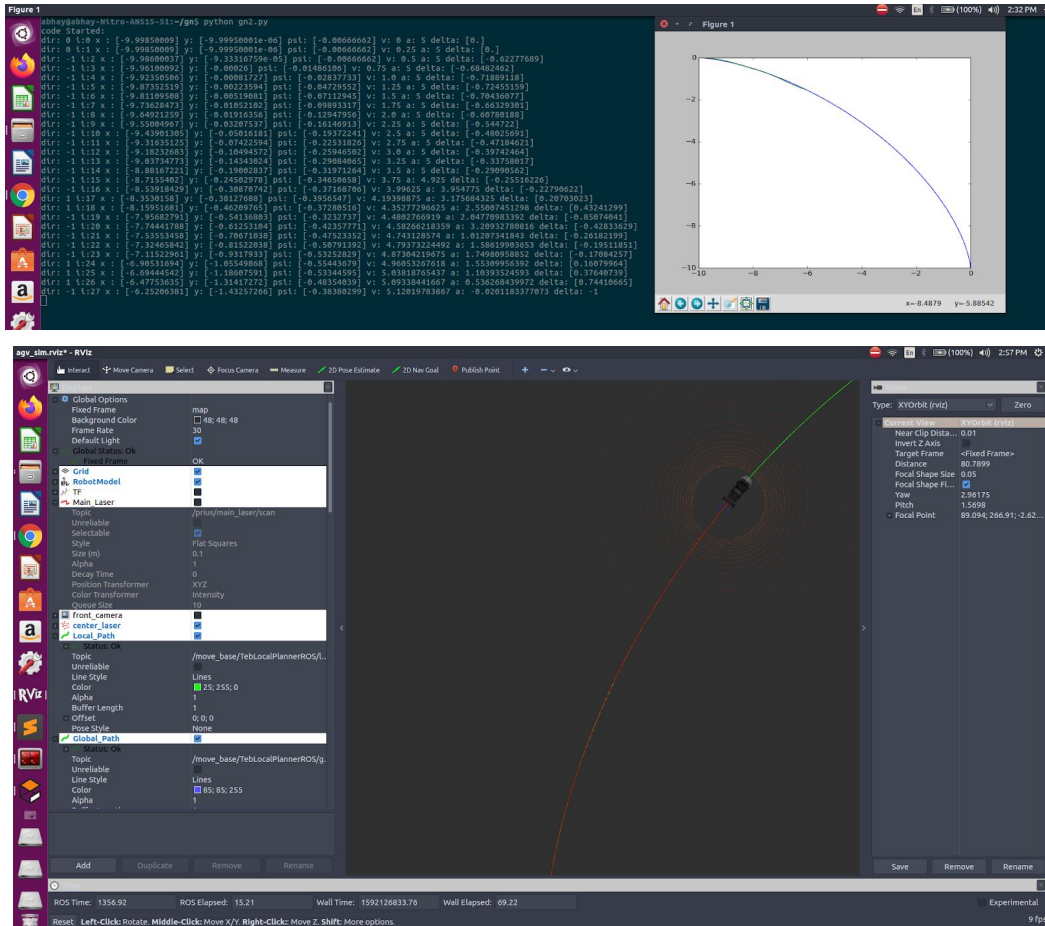


This is the ros based simulation platform Gazebo

Results

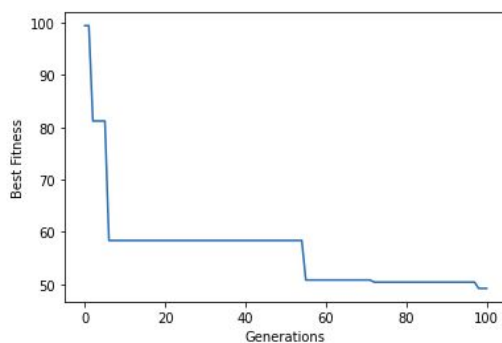
The performance of Stanley and PID based trajectory trackers, by using parameters computed by differential evolution was flawless. We can observe that the vehicle is able to track the path perfectly.

The simulation results are shown below

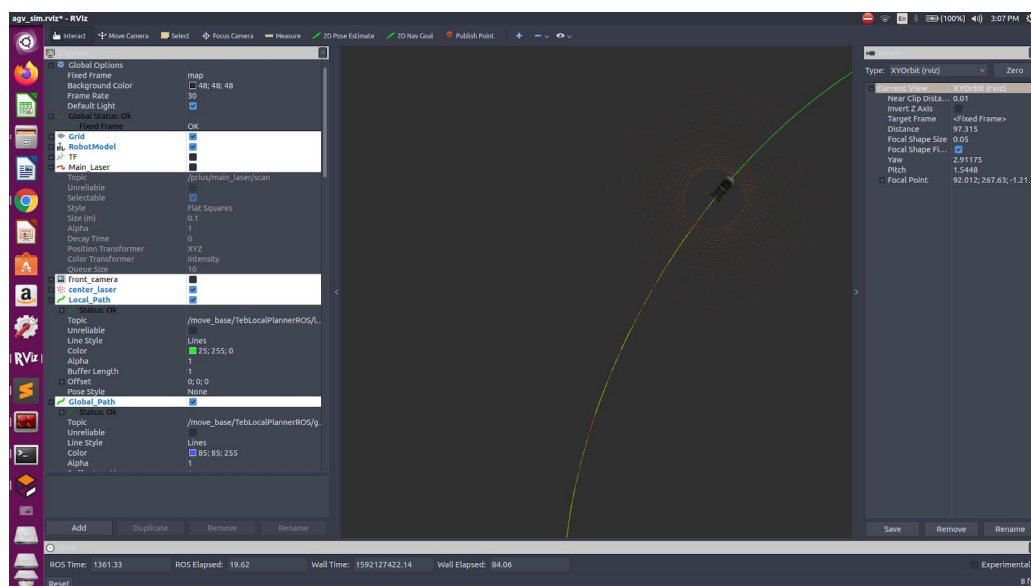
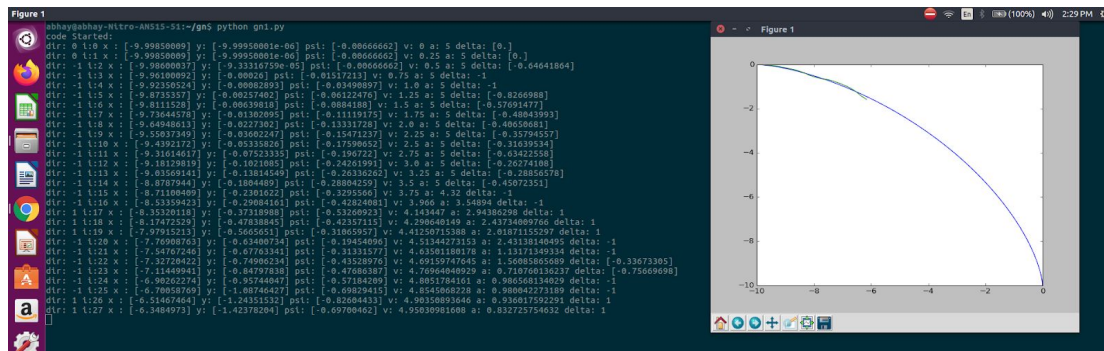


The generation vs fitness graph of the differential algorithm is shown below

Optimal Solution - [3.9931718184326175, 1.8617679832002023, 2.674334095837965, 34.63109735400193]
 Optimal Solution fitness - [49.21382714]
 Fitness(min, max) in current population - [49.21382714] [152.90385151]

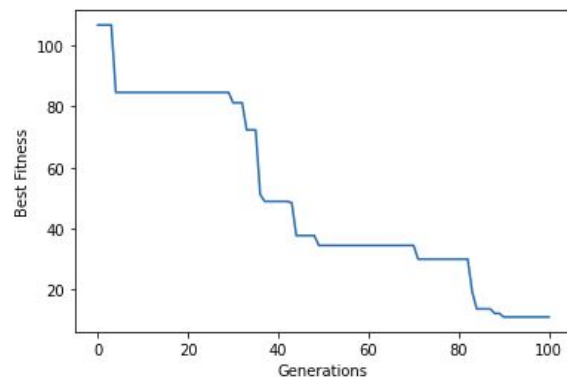


The performance of Pure-Pursuit and PID based trajectory trackers ,by using parameters computed by differential evolution was flawless. We can observe that the vehicle is able to track the path perfectly.



The generation vs fitness graph of the differential algorithm is shown below

Optimal Solution - [3.947328651569062, 3.759661637317892, 0, 5.269012641274907]
 Optimal Solution fitness - [11.09810304]
 Fitness(min, max) in current population - [11.09810304] [119.9904128]



Conclusion

From the results obtained and the simulations, we can see that the performance of the complex controller is flawless. This shows that differential evolution is indeed capable of finding a solution, which may not be the optimal solution, but very close to the optimal solution. We can say this because of the performance of the trajectory tracking controllers. Each generation took approximately 8 seconds, resulting in a total time of approximately 14 minutes.

Future Prospects

There are many ways in which this project can be improved.

1. The run time of the algorithm could be reduced. If the time in calculating the fitness of each individual is reduced, that would create a huge reduction in the total runtime.
2. The ideal values for C_μ and P_c were found out by multiple runs. The typical range of these values are known, but if there could be a process by which we could select the values without trial-and-error method, it would be definitely helpful.
3. This genetic algorithm can be also used to tune parameters of more complex trackers such as linear quadratic regulator or model predictive control.

References

The following resources were very helpful in making us go through this project

1. Wikipedia
2. Paper on Differential Evolution - [Here](#)
3. [Automatic Steering Methods for Autonomous Automobile Path Tracking](#)
4. Softwares and platforms used
 - a. Python
 - b. Robot operating system (ROS)
 - c. Gazebo
 - d. Google Colab