1 Team Descriptions

Team A consists of - Eduardo Montes - 94150, Afonso Klier - 96139, Rita Pereira - 96309.

Team B consists of - Afonso Araújo - 96138, Diogo Mucharrinha - 96179, João Santos - 96237.

Team C consists of - Mª Carolina Leite - 96273, Simão Sousa - 96323, Pedro Cary - 105701.

2 Introduction

The following report guides the reader through the **Background** behind this Autonomous Vehicles assignment, the **System Architecture** developed by each group in order to achieve as good a final product as possible, the overall **Experimental Results** the 3 groups obtained and the **Realiability of the Vehicle** on the road.

3 Background

The purpose behind this assignment was to give the students an "*hands-on*" experience with Autonomous Vehicles and all of their nuances and details. To do so, it was advised that the students formed bigger groups with their fellow colleagues in order to tackle such assignment properly. The goal was for each student to research and work on a singular part of the **System Architecture** and in the end, join every individual architecture together to properly conduct the vehicle.

Since the students had a tighter time frame and due to the high request of the vehicle it was allowed for the students to simply show a simulation of the car moving, through software.

4 System Architecture

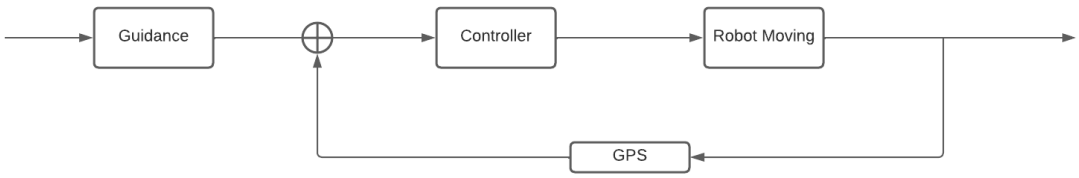


Figure 1: Block diagram for the System Architecture behind the group's implementation

5 Guidance Architecture - Team A

Guidance is the first module of the autonomous car system. Its primary goal is to generate a trajectory from an initial point to a user-specified final point. To accomplish this, our work will be divided into three major sections: image processing, path planning, and trajectory generation.

In the pseudo-code below, it is presented the overview of the guidance block.

Algorithm 1 Overview structure of guidance

1: Get user input in GUI
2: Generate occupancy grid
3: Generate path with RRT*
4: Generate trajectory with cubic spline

5.1 Image Processing

In order to obtain a portion of the map devoid of unnecessary information, we manually removed all information except the road limits, in which the car can circulate.

The result is displayed in figure 4.

Figure 2: Before image processing

Figure 3: After image processing

Figure 4: IST maps

5.2 Map Resolution

In the image provided in the assignment statement, we were given a referential of 40 m for y and 10 m for x. However, upon further inspection on Google Maps, we discovered that those 40 meters are in fact 32.89 meters. So, we used the image manipulation program *GIMP* to determine the number of pixels in the image that represent a certain road.

Due to the thick black lines that mark the world frame, it would be difficult to measure with great accuracy the number of pixels that form those lines. To avoid this lack of accuracy, we measured another road on Google Maps, as shown in figure 5, where we can see that it has 64.89 meters. From figure 6, we can observe that this road is represented by 320.8 pixels.

Therefore, the map we will work on has a resolution of, approximately, 4.94 pixels/meter or, equivalently, 0.20 meters/pixel.

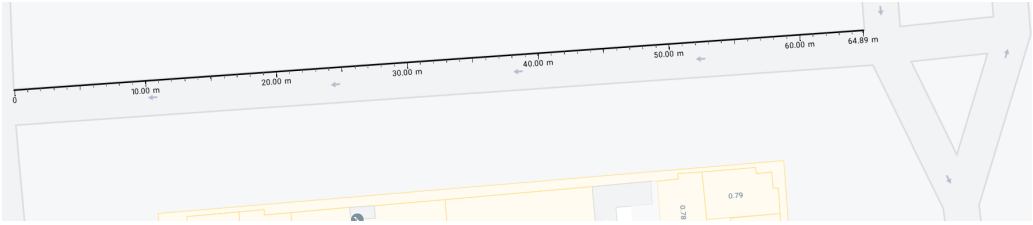


Figure 5: Referential measurement from Google Maps

Figure 6: Referential measurement from *GIMP*

5.3 Graphical User Interface (*GUI*)

An autonomous navigation like the one presented in this assignment, requires the user to specify the final trajectory point, that is, the goal state. So, we developed a graphical user interface (*GUI*) for visualization and simulation purposes, as shown in figure 7.

The user must specify the initial point as well as the goal point, in case of a simulated environment. If we were in a real environment, there is no need to state the initial point, since this is given by the GPS.

To select the initial point, the user needs to click on the Start Point button first and then on the desired place of the map, where it will appear a small red circle indicating that point. In a similar way, to choose the final point, the user needs to click on the Goal Point button and then select the desired location on the map. This time, a small blue circle will appear.

Additionally, the user is required to specify an energy budget on the empty white rectangle. When the initial and final points are defined and the energy budget specified, the user must click on the Done button to start the path planning. Figure 8 shows a possible interaction with the *GUI*.

It is important to note that the points should be placed, approximately, in the middle of the roads. If they are placed by the side of the road, the algorithm will detect a collision, which will be explained in further detail in section 5.4.3.

Figure 7: *GUI* - Initial state

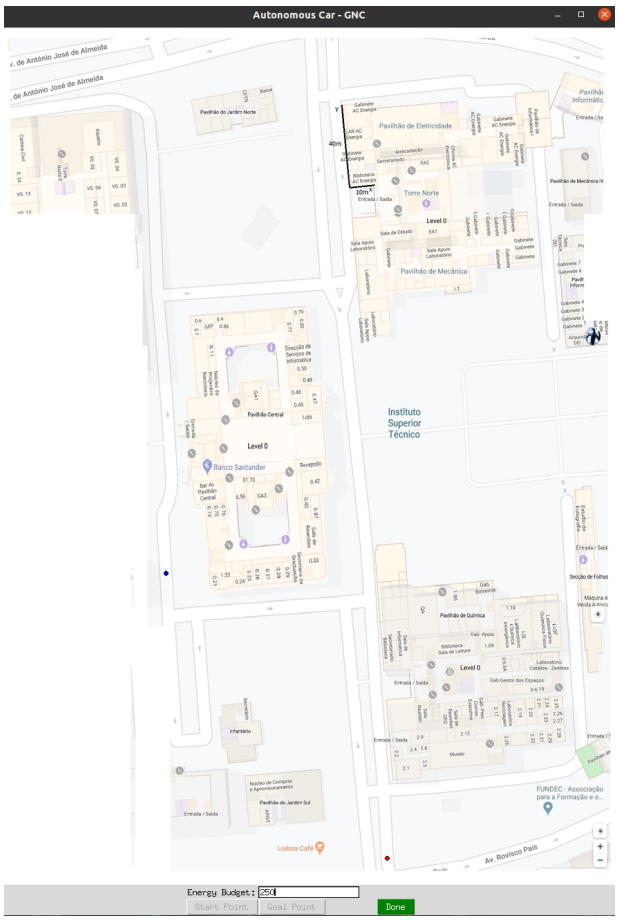


Figure 8: *GUI* - After selecting the points

5.4 Path Planning

Path planning allows an autonomous vehicle to find the shortest and least-obstructed path from a starting point to a goal state. The path can be a collection of states (position and orientation) or a collection of waypoints. This can involve taking into account the robot's physical constraints, such as its size and shape, as well as any obstacles that may be present in the environment.

5.4.1 Occupancy Grid Representation

After obtaining the processed image, the resulting map is represented through an occupancy grid, which discretizes a space into squares of arbitrary resolution and assigns each square a binary value of being full or empty. In this case, black is represented by the number zero, translating to occupied space and white is represented by the number one, indicating unoccupied space.

As a result, the grid is initialized as a matrix of zeros (black) with the size of the input image, thus having the same resolution. The grid equivalent pixels in the input image that represent the roads are filled with ones to indicate free space.

5.4.2 RRT* algorithm

In order to obtain a path from the initial point to the goal point, we used the RRT algorithm. RRT - Rapidly-exploring Random Tree - creates a tree by randomly generating nodes in free space. It begins at the start node and expands until it reaches the desired position (node). The tree grows with each iteration by adding new random nodes, checking whether the added node is inside a region that is considered an obstacle.

Even though RRT finds a feasible motion plan rather quickly, it is not guaranteed that it finds the optimal solution. Taking this into account, we decided to implement a variation of this algorithm, RRT*, which in terms of time complexity and space complexity is the same as RRT, $O(V\,log\,V)$ and $O(V)$ respectively, where V is the number of vertices. However, RRT* achieves asymptotic optimality and converges to the optimal solution faster due to the concept of rewiring. For real-time applications, this is something crucial [5].

The implementation of the RRT* algorithm was done using a code from Github [3]. Because the environment is represented by a discrete occupancy grid, some changes to the original code were required in order for it to work with an occupancy grid. The first is that we need to represent the environment in a discrete way.

Furthermore, we changed the way the nodes were generated. Previously, it was a random pick, but we added a constraint that it could only select a random white pixel, which represents free space.

Finally, to see if there was a collision, we modeled the car as a circle and checked to see if the circle crossed any black pixels. The pseudo-code for the altered RRT* algorithm is presented below.

Algorithm 2 RRT* algorithm

for max number iterations **do**
    $x_{rand} \leftarrow RandomSample()$
    $x_{nearest} \leftarrow Nearest(Tree, x_{rand})$
    $x_{new} \leftarrow Steer(x_{nearest}, x_{rand})$
    if $Collision(x_{nearest}, grid, resolution, circle)$ **then**
        $x_{near} \leftarrow FindNear(x_{new})$
        $x_{parent} = ChooseParent(x_{new}, x_{near})$
        $Rewire(x_{parent}, x_{near})$
    end if
end for

Figure 12 shows a path generated by the algorithm, where the red line is the final path and the green lines represent all the nodes of the tree.

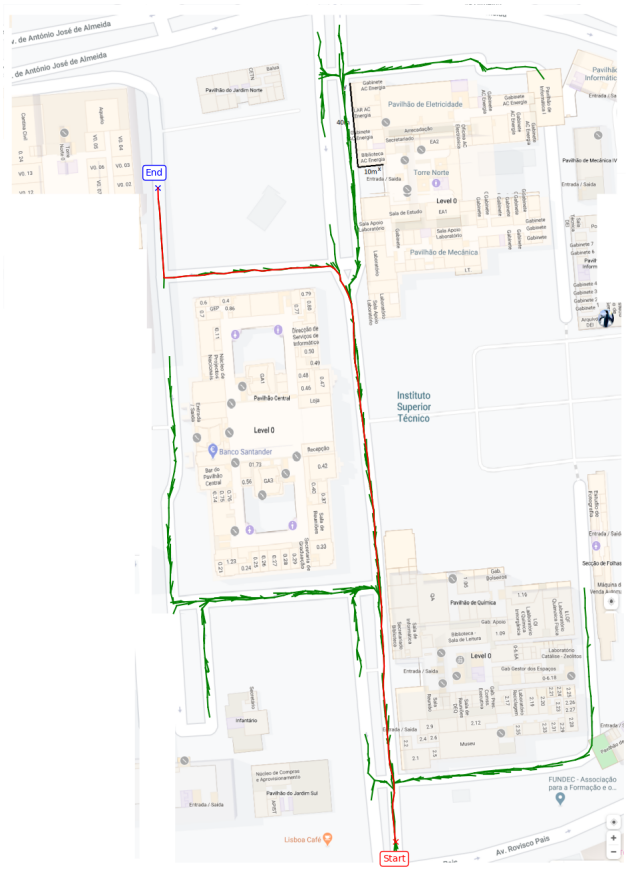


Figure 9: Path generated with RRT*

5.4.3 Collision Checking

As mentioned previously, in order to avoid that the car drives too close to the road's limits, we modeled the car as a circle and not a point. To do that, we used the Bresenham's circle algorithm, which essentially uses integer arithmetic, rather than floating point arithmetic, to calculate the positions of the pixels that should be plotted to produce the circle [1].

The parameter that was needed to produce said circle was the radius. In order to determine the value that should be used, we measured the width of the road, obtaining a value of 4.5 m approximately.

If the the car is modeled with a radius of 1.4 m (2.8 m in diameter), we would have about 1.7 m of free space, which is roughly the width of the car (1.66 m). Hence, we would have a margin of 0.85 m to each side of the road.

In figure 11, we illustrate an example where the vehicle would drive too close to the limit of the road and possibly collide with an obstacle, represented by the black pixels.

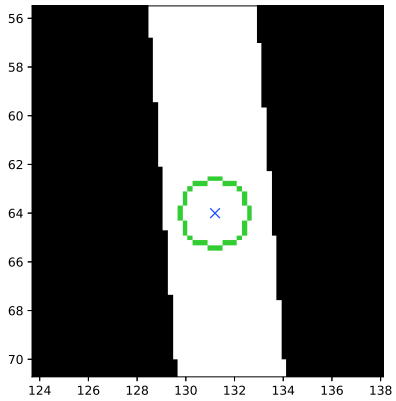


Figure 10: Without obstacle collision

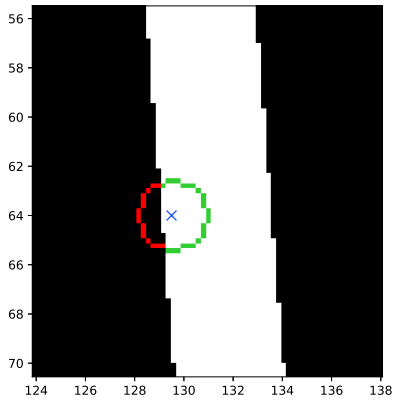


Figure 11: With obstacle collision

5.5 Trajectory Generation

Figure 12: Path after RRT*

Figure 13: Trajectory after cubic spline

Figure 14: Trajectory generated

Observing the planned path represented in figure 12, we can conclude that it is an unfeasible path, due to the fact that the car cannot perform sharp turns of almost 90 degrees. Therefore, in order to generate a realistic trajectory, a cubic interpolation (*cubic spline*) was implemented.

The number of points on the trajectory must be defined for interpolation. The idea was to multiply the total distance of the path produced by RRT* by 2, so that there is a point roughly every 0.5 m.

5.6 Limitations

One of the limitations is that the algorithm doesn't take into account the direction of the roads, meaning that sometimes the generated path may contain one-way roads that are navigated in the wrong direction, as we can see in figure 15.

Figure 15: Road navigated in the wrong direction

To address this limitation we could add one dimension to the occupancy grid that flags the direction of the road. To generate the grid, we would manually add colors to the roads of the input image that represent their direction. Then while the RRT* algorithm is running, when it generates a new node, it checks whether it is on a road with the correct direction taking into account the direction of the path. This feature is fundamental for a safe autonomous navigation.

Ideally we would also have to take into account the orientation of the initial and final positions, since the orientation has an influence on the path produced by the RRT* algorithm. Using again the example of figure 15, if the end state had an orientation of approximately $+3.0°$, even without taking into account the roads' direction, the path generated by the RRT* algorithm would probably go all around the Central Building before arriving at the goal state, unless the car had space to make a U-turn around this point.

With the concept of orientation in mind, another possible limitation that arises is the fact that the car is modelled as a circle. Again, if we had taken into consideration the orientation of the car, the most accurate way of representing it is with a rectangle of 3.8 by 1.66 meters, the real dimensions of the car.

Lastly, another improvement that we can make is to use a map with real latitude and longitude coordinates, which would make the path positions and resolution more accurate.

6 Controller Architecture - Team B

As for our controller architecture, we used the Ackermann model for a bycicle, as proposed in the assignment sheet where:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ \frac{\tanh\phi}{L} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ w_s \end{bmatrix} \tag{1}$$

in such a way that we can control our bycicle model :

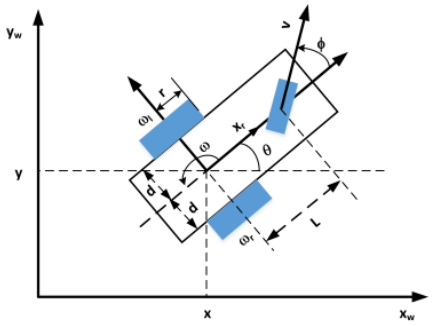


Figure 16: Car Model

As for the important parameters proposed in the assignment we use the length from the main car axis - $L = 2.46$ meters.

6.1 Controller

The vehicle's controller was based on the example provided by professor João Sequeira in lecture 8 of the course. Although it was designed for a unicycle, it can fit our needs for a 4-wheel vehicle.

However, in both cases, the controller must consider the robot's specific constraints and requirements, such as maximum linear and angular velocities and steering angle.

The group proceeded to develop 3 different controllers, finding a good result with the final, and submitted one. Every controller was tuned to take into account the dynamic, and the constraints of a car-like robot, to ensure it would fit the right model.

$$w_e = [x_{ref} - x, y_{ref} - y, \theta_{ref} - \theta] \tag{2}$$

$$b_e = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} w_e \tag{3}$$

Equation 2 represents the error between the desired and current positions for the x and y coordinates and the difference between the desired angle to achieve the following location and the current angle of the car.

6.1.1 Professor's Controller

For this controller, we used the equations below,

$$\begin{aligned} v &= K_v{}^b e_x \\ w_s &= K_s{}^b e_\theta + K_L{}^b e_y \end{aligned} \tag{4}$$

In equation 3 it is applied a rotation on those coordinates, changing the frame where the robot is located. These values are used in the controller, along with the gains, to obtain the desired velocity and angular acceleration.

Then, we conducted a set of tests based on a trial-error process to understand the relationship between the variation of each gain and the robot's behavior.

For the testing below, we maintained every parameter at their default value ($K_v = 0.03$, $K_s = 40$ and $K_l = 1$), tuned the required one accordingly.

The parameter K_v is responsible for the control of the velocity of the vehicle. It is one of the most important parameters since its overshoot can make our controller unstable:

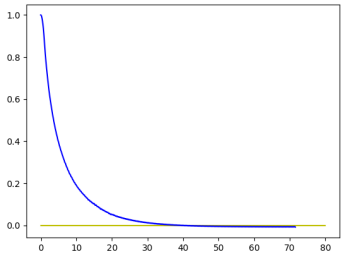


Figure 17: $K_v = 0.006$

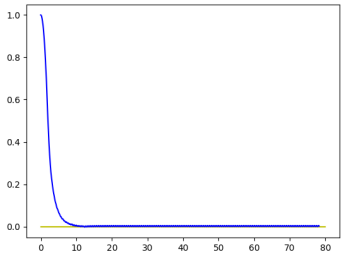


Figure 18: $K_v = 0.03$

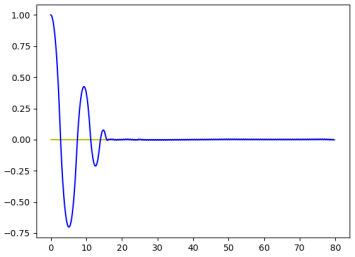


Figure 19: $K_v = 0.15$

As we can see, even though all the values for the parameter K_v achieve the purpose of getting the vehicle on the correct path, it is easy to verify that with the first K_v value, it takes too long whereas, with the last K_v value, it overshoots the path (in the beginning) and has to over-correct itself (oscillatory behavior).

With that, we decided to go with the middle value due to having the best trade-off of the three.

The parameter K_s is related to the angle of the vehicle allowing us to correct the vehicle´s angular velocity:

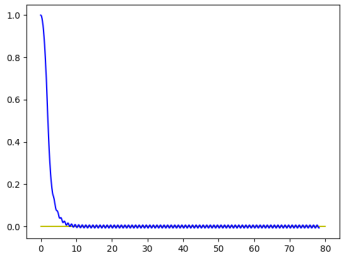


Figure 20: $K_s = 5$

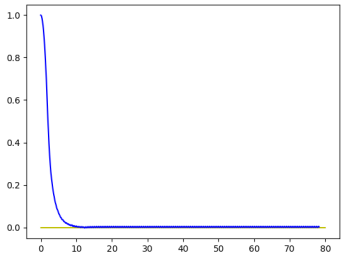


Figure 21: $K_s = 40$

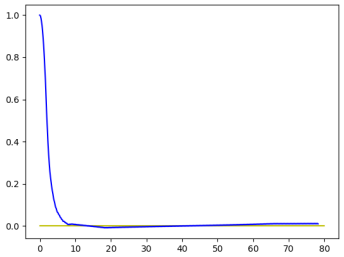


Figure 22: $K_s = 100$

Although they look very similar we can notice some noise forming on the lower K_s value and a bit of overshoot for the higher K_s value, while the middle K_s value seems to be the most stable.

The last controller parameter we decided to optimize was K_l, related to the error on the y-axis that is used to correct the angular velocity of the vehicle in conjunction with the angle θ:

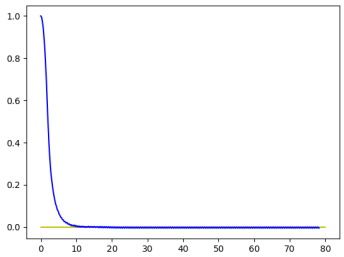


Figure 23: $K_l = 0.05$

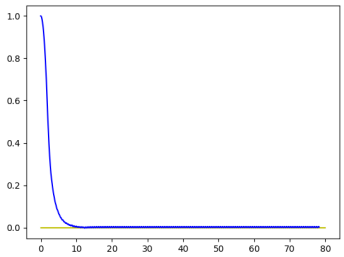


Figure 24: $K_l = 1$

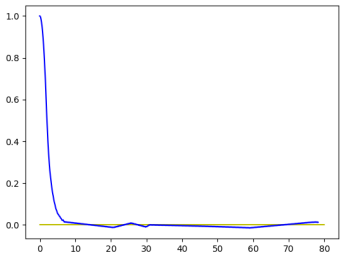


Figure 25: $K_l = 100$

As we were expecting, despite changing this parameter within several orders of magnitude the effect this has on the overall system is very minute (the error related to the angle is way bigger naturally than the error on one axis).

With these parameters in mind, we knew how best to implement the following 2 controllers.

6.1.2 PID Controller

$$
\begin{aligned}
v &= 0.00786 \times k_v \times error \\
w_s &= k_p \times error_{theta} + k_d \times \frac{error_{theta} - error_{previous}}{dt} + k_i \times integral_{theta}
\end{aligned} \tag{5}
$$

The option of a PID controller to control the angular acceleration and a PD controller for the velocity was explored. For this, the generic expression of a PID controller was used, represented in 5, and the gains were fine-tuned to fit the model in question.

The PID that controls the angular acceleration has three components dependent on the error in theta, thus being directly linked to the direction the car intends to take to reach the next point.

For the speed, a simple PD was created that relates the desired speed, which had been obtained in a previous test, depending on the time step used at the time, with the error between the current position of the car and the desired position and its gain.

However, we managed to design a better performing controller, as shown next.

6.1.3 Final Controller

$$
\begin{aligned}
v &= k_v \times error_{position} \\
w_s &= k_s \times error_{theta}
\end{aligned} \tag{6}
$$

In an attempt to improve the car's performance, a more straightforward controller was built, it revealed remarkable results.

In this controller, the linear speed depends on the error between the current position of the car and the next desired position. This error is multiplied by a gain tuned to a broad set of tests to be as robust as possible. Regarding speed, a proportional controller was created that relates the variation of the theta angle and a specific gain. This gain takes inspiration from

the tests done before. We can clearly see this controller is the best in our experimental results - 8.1.

6.2 Updates

As for this section, it is described the *Updates()* method used for every controller, including the submitted one, responsible for updating our variables for every time step.

Algorithm 3 Update()

$\phi_{current} \leftarrow \phi_{previous} + (\omega_{s_{current}} + \omega_{s_{previous}}) \times \Delta t$
if $\phi_{current} > \frac{\pi}{6}$ **then**
    $\phi_{current} \leftarrow \frac{\pi}{6}$
end if
if $\phi_{current} < \frac{-\pi}{6}$ **then**
    $\phi_{current} \leftarrow \frac{-\pi}{6}$
end if
$\theta_{current} \leftarrow \theta_{previous} + \Delta t \times (\frac{\tan(\phi_{current})}{L} \times v_{current} + \frac{\tan(\phi_{previous})}{L} \times v_{previous})$
$\dot{x}_{current} \leftarrow v_{current} \times \cos(\theta_{current})$
$\dot{y}_{current} \leftarrow v_{current} \times \sin(\theta_{current})$
$x_{current} \leftarrow x_{previous} + (\dot{x}_{current} + v_{previous} \times \cos(\theta_{previous})) \times \Delta t$
$y_{current} \leftarrow y_{previous} + (\dot{y}_{current} + v_{previous} \times \sin(\theta_{previous})) \times \Delta t$
$trajectory\_creator(x_{current}, y_{current}, \theta_{current}, \dot{x}_{current}, \dot{y}_{current})$
$previous\_generator(x_{current}, y_{current}, \theta_{current}, \phi_{current}, v_{current}, \omega_{s_{current}})$

As we can see, the code above follows the very simple mathematical rules as stated with the Ackermann model in 1. The group chose a factor of $\pm\frac{\pi}{6}$ since it is hard for a front tire of a car to exceed 30°, upon its steering.

If we are to take 2 different states of our car we can see that for our second state, no matter the state, we always need the information of state 1 to calculate, x, y, θ and ϕ.

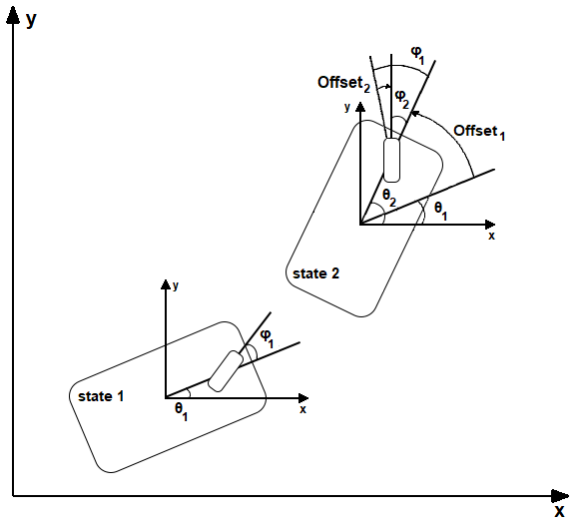


Figure 26: Parameter Variation for 2 Different Car States

In the image above,

$$\theta_2 = \theta_1 + offset_1 \tag{7}$$

$$\phi_2 = \phi_1 + offset_2 \tag{8}$$

Joining the equations above with the image, we can see why we need the information in terms of "v" and "w_s" from the previous state to compute our current state. In this case "$offset_1$" is $\Delta t \times (\frac{\tan(\phi_{current})}{L} \times v_{current} + \frac{\tan(\phi_{previous})}{L} \times v_{previous})$ and "$offset_2$" is $(\omega_{s_{current}} + \omega_{s_{previous}}) \times \Delta t$.

As for the function *trajectory_creator()* its job is to append our newly created parameters to a trajectory vector whereas the function *previous_generator()* allows us to update our new "previous" parameters for the next set of parameters.

7 GPS Architecture - Team C

7.1 Hardware

Before we decided on the final configuration we tried several methods of running our GPS.

- IP address

We started off by using a laptop, however, not every laptop has GPS, and that was the case for our group. With that in mind, we thought about using the **IP address** from our laptop to get our location.

While an IP address can be used to determine an approximation of a device's location, it is not a reliable or accurate method for determining a device's GPS location. IP address localization works by looking at a geolocation database.

However, these databases are not always accurate, and the location information they provide can be off by several meters. Additionally, IP addresses can be easily spoofed, meaning a device's location can be falsely reported.

The main problem is that the IP address is used to identify the device on the internet, not its location. It can give you information on the location of the internet service provider but not the exact location of the device.

Of course, we didn't want this, because we needed GPS technology capable of determining a device's location with a high degree of accuracy (error not larger than a few meters).

- Phone GPS

After that, we thought of **connecting the phone's GPS to our program**. However, this lead to some difficulties and problems like compatibility issues and driver issues.

In the first one, the phone wasn't compatible with the computer's operating system, becoming more difficult to establish a connection between the two devices.

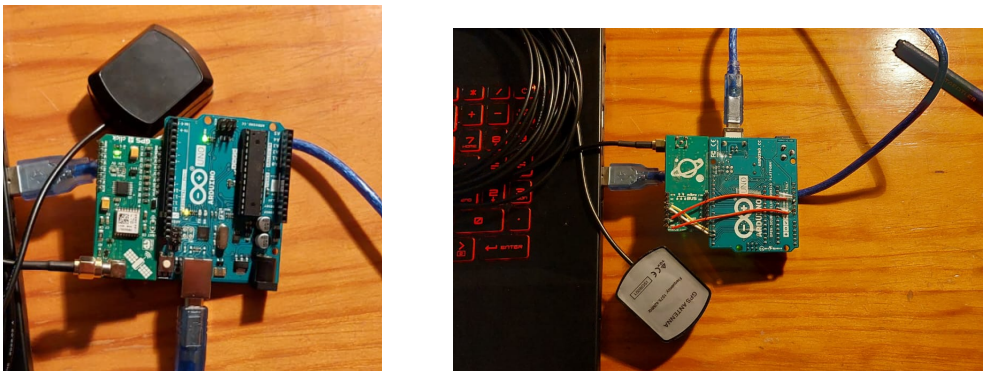
The second one was because we didn't find the correct drivers for the phone to communicate with the computer.

Another reason was the precision. Smartphone GPS use a combination of GPS, Wi-Fi, and cellular network data to determine a user's location. In general, smartphones can provide location data with an accuracy of around 10 to 20 meters, although this can be improved to a few meters in open areas with a clear view of the sky.

- GPS device

Finally, we settled by using a dedicated **GPS device**. These devices typically have a higher-quality GPS receiver and can provide location data with an accuracy of around 2 to 6 meters, or even better.

In the construction of our GPS system, we used different components, all in order to obtain the maximum precision of the location almost in live time. We can see the final result in Figure 27.



(a) front view (b) back view

Figure 27: GPS Installation

7.1.1 Arduino

An Arduino microcontroller is necessary for the GPS system to work as it's the "decoder" of the system, processing the data received from the GPS module and sending it through serial communication to the main code in a computer program.

7.1.2 Antenna

An antenna is necessary for a GPS system in order to receive the signals from the GPS satellites.

The antenna is typically a small device that is connected to the GPS module and is designed to capture and amplify the weak signals emitted by the satellites.

7.1.3 *GPS 4 click*

GPS 4 click carries the L70 compact GPS module from Quectel. The click is designed to run on either 3.3V or 5V power supply. It communicates with the target microcontroller(Arduino) over UART(Serial) interface.

7.2 Code Structure

7.2.1 Libraries

```
#include <Adafruit_GPS.h>
#include <SoftwareSerial.h>
```

Figure 28: Libraries used in code.

The *Adafruit_GPS.h* [2] library used as we can see in figure 28, is an open-source software library for the Arduino platform that allows for easy communication and parsing of data from a GPS module. The library provides a set of functions and variables that can be used to receive and interpret GPS data from a connected module.

Some of the main features of the *Adafruit_GPS.h* library include:

1. Parsing of *NMEA* sentences;
2. Set the ports and baud rate of the serial communication with the board;
3. Support communication with various GPS modules, allowing selective protocol information.

7.2.2 GPS calibration

The results obtained by the external GPS module were not very satisfactory at first. With that in mind, it became evident that were some factors that were significantly influencing its performance. The most important ones were related to the sample frequency, power consumption and the quality of the hardware connections.

The majority of the problems were solved by understanding and sending instruction sets [4] to the Quectel L70, but another crucial detail was the permanent physical contacts by soldering the terminals to their specific ports, which made possible a stable serial connection and a stable power supply.

7.2.3 Arduino code

The Arduino Uno works as an interface between the GPS module and the computer so that, instead of receiving a bunch of NMEA streams the computer will receive pre-processed information that encodes directly position, altitude, velocity and signal quality. To do that the Arduino code consists of an initialisation, to set important GPS parameters and an infinity loop which sends through serial communication the relevant information processed from NMEA protocol every time a new NMEA structure arrives from the GPS module.

The Adafruit library made it easier as it has built-in functions that make the pre-processing of NMEA strings and store relevant information in accessible class variables. The hard part was the initialization and understanding of the L70 instruction set as this library wasn't fully designed for this module. With that, the Arduino is able to restrict the module to send only RMC(position, fix), GGA(position, altitude) and VTG(ground speed) as NMEA structures. It is also possible to increase the precision of the module by setting a higher sample frequency(5Hz instead of 1Hz) and defining the output for every 5 position fixes.

7.2.4 Serial communication in Python

In the computer program, the UART serial protocol has to be matched with the Arduino. This means making the handshake on the baud rate and serial port. With the communication established a function was created that received a serial object and a timeout value.

This way the function will return a refined coordinate formation that can be directly interpreted in Google maps and processed by other libraries such as Geopy. The timeout will make sure the function doesn't get stuck on waiting for a good GPS read, making a smooth integration with other code modules.

This function also implements a save of every successful GPS read in a text file in case a bug need to bee fixed.

7.2.5 Conversion to map coordinates

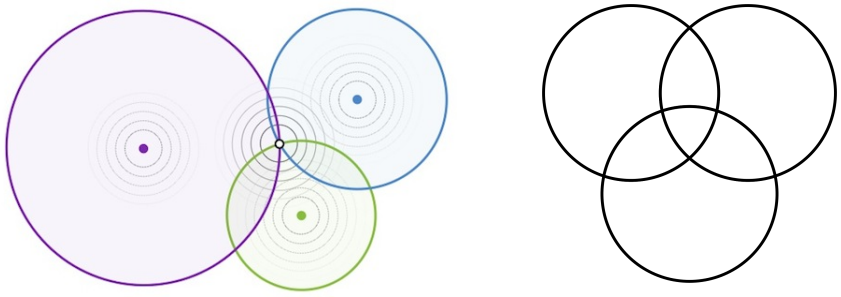
After obtaining the latitude and longitude there needs to be a conversion to the maps coordinates. The following pseudo-code shows how this conversion is done. Before analysing the algorithm it's important to note that we have access to the position of 3 points in the map (map_p_i) and their corresponding real-world coordinates ($coord\_p_i$). The variable *coord* represents the coordinates we want to convert.

Algorithm 4 Conversion()

$ratio \leftarrow \frac{distance(map\_p_1, map\_p_2)}{real\_distace(coord\_p_1, coord\_p_2)}$
for 3 iterations **do**
    $dist_i \leftarrow ratio * real\_distance(coord, coord\_p_i)$
end for
$inter \leftarrow get\_intersection(map\_p_1, dist_1, map\_p_2, dist_2)$
$op_1 \leftarrow distance(inter\_1, map\_p_3) - dist_3$
$op_2 \leftarrow distance(inter\_2, map\_p_3) - dist_3$
if $op1 < op2$ **then**
    $return\ inter_1$
end if

Firstly, we calculate *ratio* which gives us, as the name implies, the ratio between the real distance in meters to the maps units. Having this information we can get the distance in map units between our desired point and the map_p_i points, by calculating the real word distance between $coord\_p_i$ and *coord*. This leaves us with 3 points on the map and the distance to each of these points, which is enough to find the point we are looking for. In a perfect world, the only thing that's left is to calculate the intersection between these 3 circumferences (map_p_i as centres and $dist_i$ as the corresponding radius). However, if there's a small difference in map_p_i and it's corresponding $coord\_p_i$ the circumferences will be slightly shifted not leading to a perfect intersection.

With this in mind, our algorithm gets the intersection between 2 of the circumferences (2 points) and then calculates which one is closer to the remaining circumference. This way, if the intersection is perfect, we get the correct point, and if not, we still get a point with a strong probability of being correct.



(a) perfect scenario (b) imprecise measurements

Figure 29: Circumference intersections

A better solution to this problem would be to calculate the 3 inner points and get an average of the 3. Theoretically, this is a more sound solution, however, in practice, the difference from the resulting points will be minimal since the existing errors are also small. It's important to note that the smallest of imprecisions will lead the circumferences not to intersect on a single point. This creates a need to develop an algorithm capable of dealing with slight errors but doesn't mean the measurements aren't solid.

Lastly, for this algorithm to be correct, we must assume the geographic coordinate system to be in fact a Cartesian coordinate system, since what the algorithm does is convert between 2 Cartesian coordinate systems with different base axes. This is more than a solid approximation since the earth's curvature is irrelevant in such a small map, as the one we are using.

7.3 Future Work

An autonomous car should have a variety of sensors in addition to a GPS to ensure safe and reliable operation. A GPS provides the car with its location, but it alone is not sufficient for an autonomous car to navigate and make decisions.

Additional sensors that an autonomous car may use include:

- LIDAR (Light Detection and Ranging): LIDAR uses lasers to create a 3D map of the car's surroundings, providing the car with information about the location and shape of objects around it.
- Camera: Cameras can be used to detect traffic lights, signs, and pedestrians, as well as to provide a visual representation of the car's surroundings.
- Radar: Radar can be used to detect other vehicles and obstacles in the car's path, even in poor visibility conditions.
- Ultrasonic sensors: Ultrasonic sensors can be used to detect objects at close range, such as when parking a car.

These sensors work together to provide the autonomous car with a comprehensive understanding of its environment, allowing it to make safer and more efficient decisions.

8 Experimental Results

8.1 Path Planning and Controller Results - Team A & B

As we were unable to use the robot-like vehicle to test the integration of the various functional blocks, we decided to approximate the GPS receiver data to the update model data for the position giving us a way to test the integration of the path planning and the controller.

We can see below, from a trajectory defined *a priori*, that the best controller is the Final Controller.

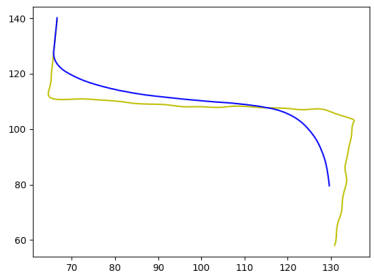


Figure 30: Professor's

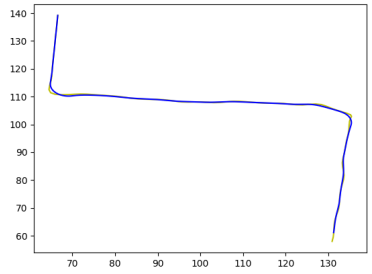


Figure 31: PID

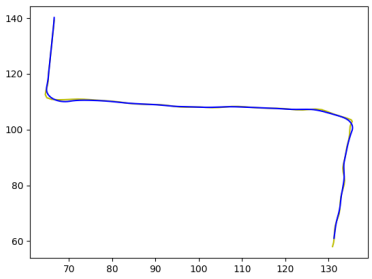


Figure 32: Final

The table below shows that the controller that has produced the best results was the Final one.

	Professor's controller	PID controller	Final controller
MSE[m]	17.387	3.0122	2.8708

However these results were obtained with a trajectory modified enough for our controller to run independently from our Path Planning block.

Even though this did not misconstrue our controller conclusions, we still ran our Final Controller with a direct connection to our Path Planning "root". With this new analysis we discovered that our K_v is 0.03095.

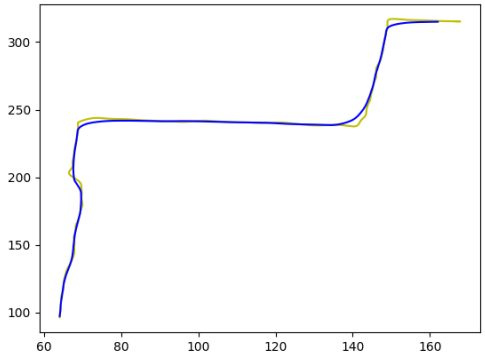


Figure 33: Controller output

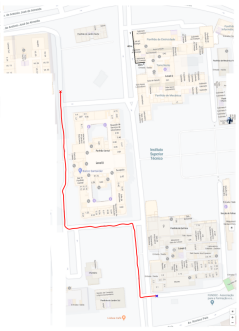


Figure 34: Path Planning output

As we can see our Final Controller behaves accordingly to a complex enough trajectory. However we faced some challenges that we could not overcome when it came to more simple trajectories, such as the straight line represented below,

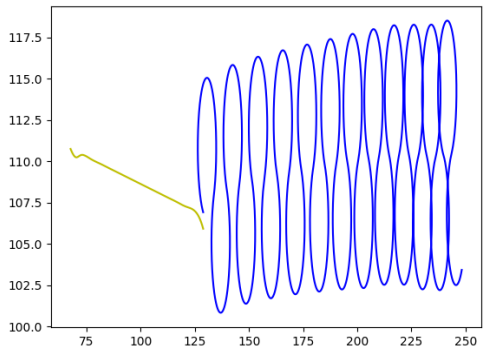


Figure 35: Controller output of a straight line

Figure 36: Path Planning of a straight line

In order to improve this result we would need a better integration between the controller block and the path planning block.

8.2 GPS - Team C

In order to have an idea of the accuracy of the GPS module we did the following test. We stood with the antenna in the same spot over a period of around 60 seconds. After getting the average result and calculating the deviation for each sample we got the following results. Each line represents a different spot on the university campus.

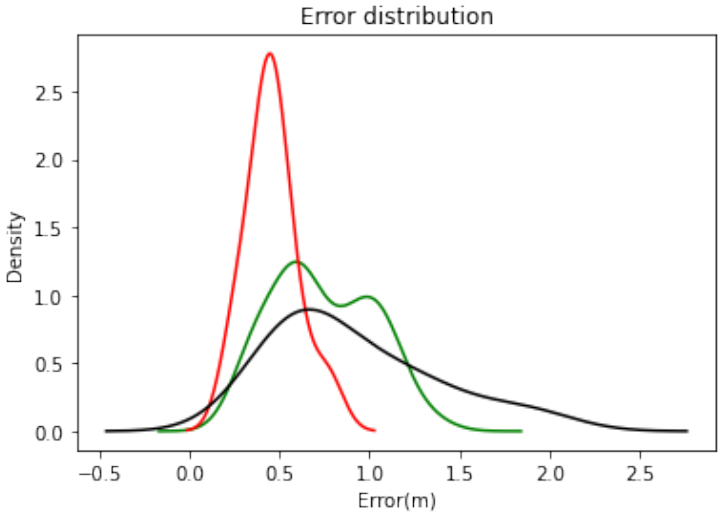


Figure 37: GPS error distribution

As we can see in figure 37 we get a max deviation of a little more than 2 meters and the expected deviation is between around 0.5 to 1 meter.

Another test we did was to simulate, on foot, a similar path to what the car might follow, a full lap around IST´s Central Building.

The first thing to note is that over a period of around 500 seconds (around 350 measurements), we only lost GPS signal one time with a duration of around 4 seconds when we got

close to the building. This leads us to believe that, with the car in the middle of the road, we should have a constant GPS signal.

That being said, it's important to keep in mind that, despite having signal, the coordinates returned might be inaccurate. One simple way to test if the coordinates are plausible is by calculating the velocity at which the antenna needs to move in order to get such coordinates. If we define a max accepted velocity of $35km/h$ and check if at any point there's a calculated velocity bigger than that, for the test described, we get 8 occurrences.

```
39.152099032774245
124.43128622642591
71.15759647168521
111.84611494264986
37.96699425661768
139.62082189811727
519.3204557345776
277.4249879337113
```

Figure 38: Impossible velocities

Our code discards the coordinates that lead to these velocities, handling them the same as when there's no signal.

Finally, as we can see in figure 39, our GPS gives us an overall solid representation of our position.

Figure 39: Example path

Figure 39 was obtained by logging one in each ten GPS outputs during a walk.

9 Reliability of the Vehicle

As for the Reliability of the Vehicle, it's impossible to be sure without doing a real-world test. That being said, our group did manage to get valuable and good results for the proposed circuit for our vehicle, leading us to assume the developed code would suffice and work if we were to implement it in real life. We were able to clearly define the path for the car to follow, the GPS data, as shown in the experimental results, was precise and the controller was able to correctly follow the defined path.

References

[1] Bresenham's Circle Drawing Algorithm in Computer Graphics - TAE — tutorialandexample.com. https://www.tutorialandexample.com/bresenhams-circle-drawing-algorithm. [Accessed 26-Jan-2023].

[2] PythonRobotics/Adafruit_gps Library— github.com. https://github.com/adafruit/Adafruit_GPS. [Accessed 26-Jan-2023].

[3] PythonRobotics/PathPlanning/RRTStar at master · AtsushiSakai/PythonRobotics — github.com. https://github.com/AtsushiSakai/PythonRobotics/tree/master/PathPlanning/RRTStar. [Accessed 26-Jan-2023].

[4] Quectel L70 DataSet. https://docs.rs-online.com/f450/0900766b8147dbe0.pdf. [Accessed 26-Jan-2023].

[5] Sertac Karaman, Matthew R. Walter, Alejandro Perez, Emilio Frazzoli, and Seth Teller. Anytime motion planning using the rrt*. *2011 IEEE International Conference on Robotics and Automation*, 2011.

10 Bibliography

[1] Robotics (Lectures 5-8) - Slides of Theoretical Classes 2022/2023, 1st Semester (MEEC), by Professor João Sequeira