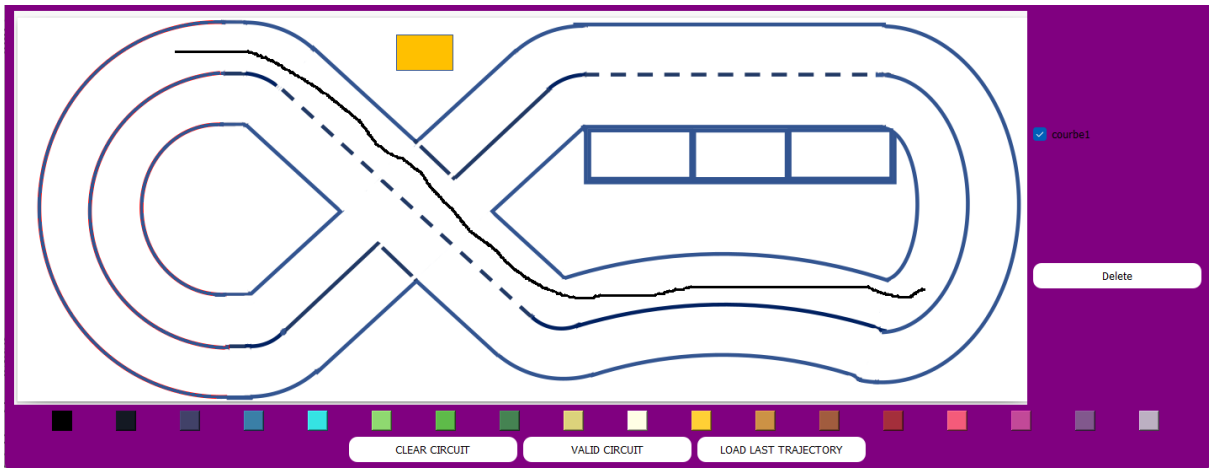
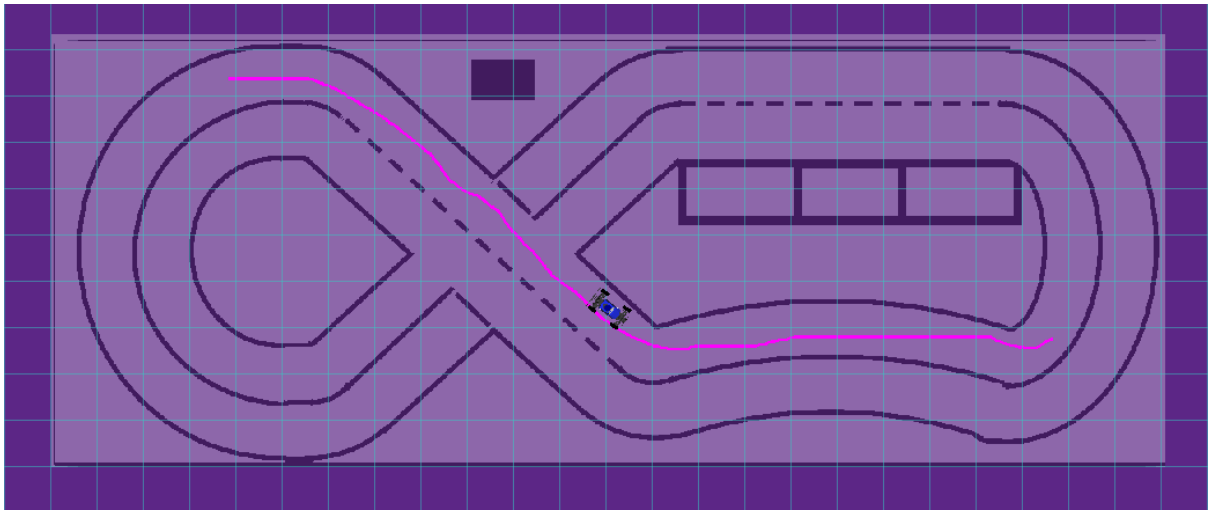
Project report



Autonomous Car simulator



Authors:
Lo'aï GANDEEL, Benoît GENOUD-DUVILLARET

In collaboration with:
Lucie BOURRAS, Naimé MARTINS, The house artificial intelligence (MIA)

Tutors:
Frederic PRECIOSO, Diane LINGRAND

Sommaire

I - Introduction

The new revolution in the automotive world is the introduction of so-called “autonomous cars” on the roads. It is, ideally, a car driving itself, capable of sensing its environment and correcting its trajectory and speed without any human involvement whatsoever. In a self-driving car, no passengers are required in order for the car to drive.

Of course, those cars need a high level of engineering to be safe, as much for other drivers on the road as for passengers of the car. It needs to be capable of making decisions as quickly as us humans do when driving. Mainly, they need to “learn” how to properly drive on circuit before they can drive on normal roads just like any other cars. This is what we call machine learning.

In order to produce the sufficient database for the car, we virtually create paths in an application and add those trajectories to the database. This solution, seemingly easy, is the heart of our project.

The purpose of our project was made clear when we visited a building called the MIA ("Maison de l'Intelligence Artificielle"). We saw the real robotic toy-size car and the real map with our referent Mrs. Lingrand:

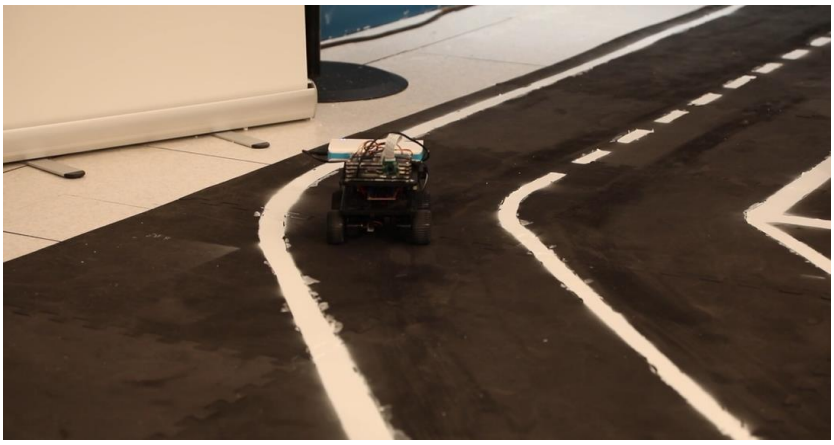


Figure 1.1: The MIA’s toy car

Figure 1.2: The MIA's circuit

Nonetheless we were asked to use ROS (Robot Operating System) and its 3D visualization tool called Rviz to model the car and its path on the circuit for convenience. It allowed us to test our code on a virtual car and a virtual map on a virtual machine.

Our main problem consisted in how to make the car follow a trajectory as close as possible and how to optimize the rendering of the trajectory in order to validate the 2 directional wheels model, an approximation used to reduce a car to a "bicycle" (useful to translate a drawn trajectory to a car movement).

II - Basis of the project

Our project relies principally on two applications to function: the first one is the application made to draw trajectories and the second is our 3D simulator. At the base of the project, those two applications were linked by 1 file: trajectory.txt

This file contained data from the curve that could be understood by the 3D simulator as commands for the car.

To draw our paths and to visualize the car on RViz, we have our own version of the MIA circuit (shown next page). The goal is to have the same circuit for both applications.

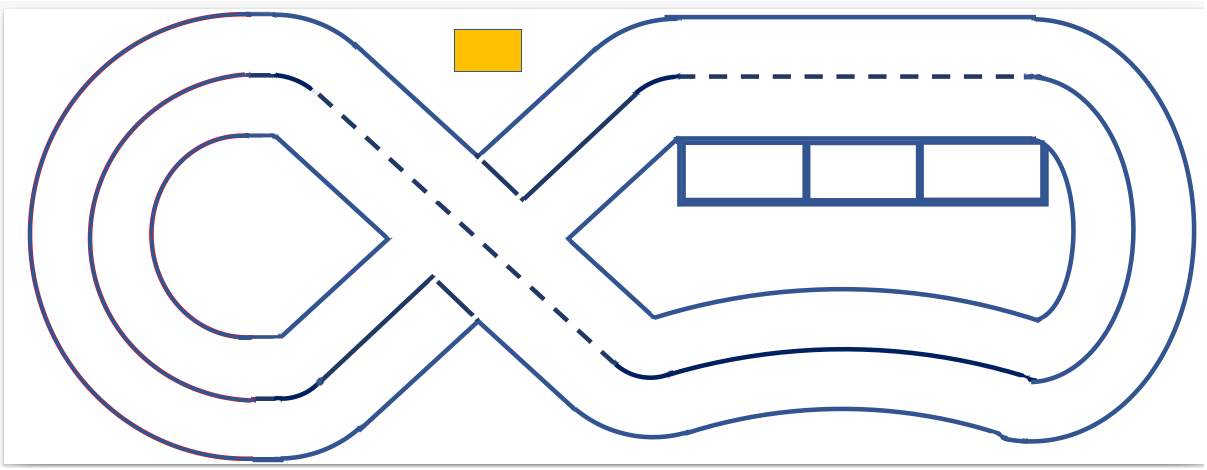


Figure 2.1: The png MIA's circuit

A – ROS visualization: Rviz

1 - Functioning

ROS stands for Robot Operating System. It is a set of software libraries and tools that are made to build robot applications. RViz stands for Ros VIsualiZation. It is a 3D visualization tool for ROS.

Rviz offers multiple tools to visualize the robot and its environment. It can be used to visualize the robot's position, the robot's sensors, the robot's path, and others. The basic interface after installation looks like this:

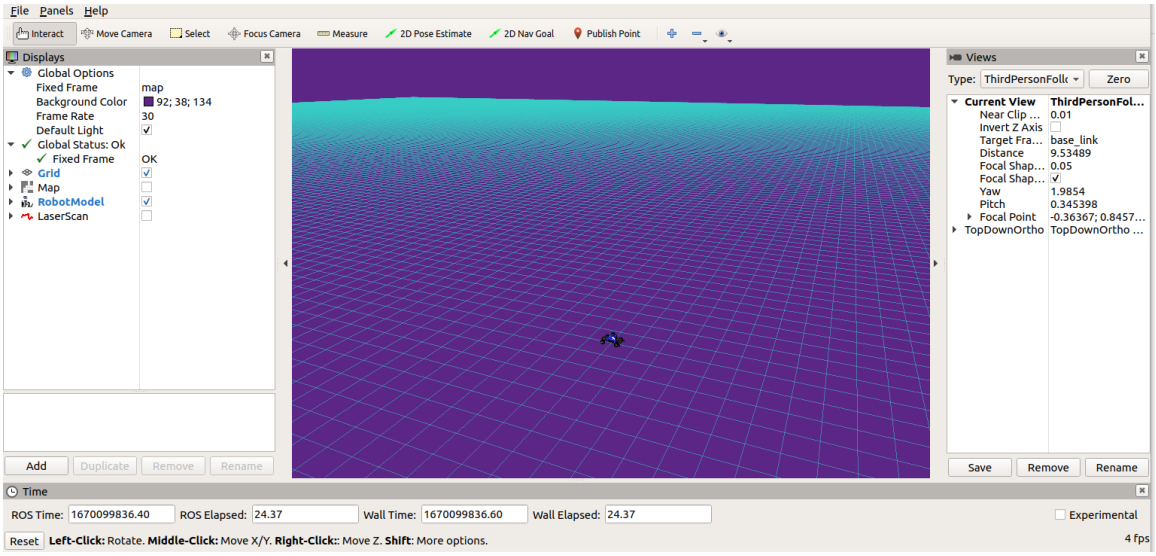


Figure 2.2: RViz initial interface

We are also using MuSHR (Multi-agent System for non-Holonomic Racing) that gives us a premade virtual set to work with (folders where the car we will work with is defined, …). The car looks like:

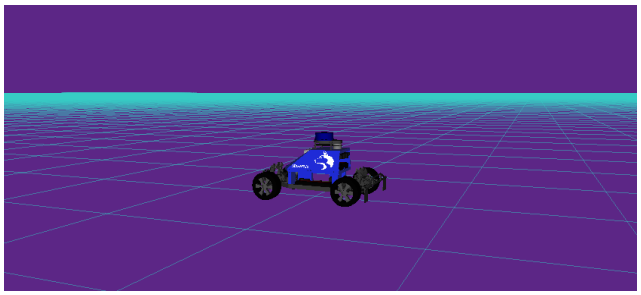


Figure 2.3: MuSHR car

We can drive the car by keeping the mouse cursor on the window shown on the right and using the WASD keys to manually drive the car.

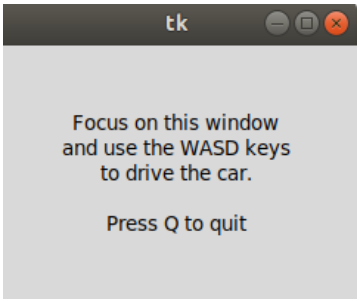


Figure 2.4: 'keyboard' node window

Ros software works with commands and topics. For example the command 'roslaunch' is used to launch a launch file (.launch). This launch file is a file containing all the commands to launch a node (A .cpp or .py file for example). In other words, a node is a program which can be launched by roslaunch (so it includes imports such as ```import rospy```).

- A node can be a publisher, a subscriber or both:
- A publisher is a node which sends data to a topic*.
- A subscriber is a node which receives data from a topic*.

*: A topic is a data stream. It can be a string, a number, a picture, a 3D point cloud, etc.

There is a package called Mushr_sim that contains the virtual car and the virtual map. It also contains a launch file that launches the car and the map simultaneously (```multi_teleop```).

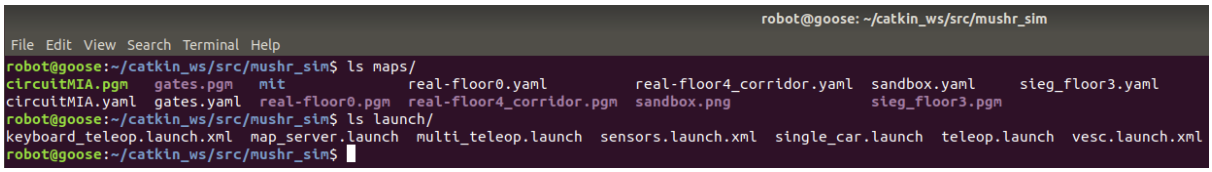


Figure 2.2: Noticeable files in mushr_sim

In our case the map is called 'circuitMIA.pgm'. This image has to be associated with a '.yaml' file that contains the map's metadata (in our case 'circuitMIA.yaml') in order to be set as a background. The metadata is the resolution of the map, the origin of the map, the map's name, and others. It is used to convert the coordinates of the map into coordinates that can be used by the car:

```yaml
image: circuitMIA.pgm
resolution: 0.01
origin: [0, 0, 0]
occupied_thresh: 0.1
free_thresh: 0.09
negate: 0
```

The car and then the map are launched with the command 'roslaunch node file.launch' The '.launch' file contains all the commands to launch the car (which will then call the map.launch file):

```xml
map_server.launch
<launch>
<arg name="map" default="$(find mushr_sim)/maps/circuitMIA.yaml"/>
<node pkg="map_server" name="map_server" type="map_server" args="$(arg map)" />
<param name="/map_file" value="$(arg map)" />
</launch>
```

Having this in mind, we have to underline that we have the possibility to change the map and the car settings. Here's a node containing the settings of the car:

```python
class FakeURGNode:
```

```
def __init__(self):

self.UPDATE_RATE = float(rospy.get_param("~update_rate", 10.0))
self.THETA_DISCRETIZATION                                          =
float(rospy.get_param("~theta_discretization", 656))
self.MIN_RANGE_METERS    =    float(rospy.get_param("~min_range_meters",
0.02))
self.MAX_RANGE_METERS = float(rospy.get_param("~max_range_meters", 5.6))
self.ANGLE_STEP          =          float(rospy.get_param("~angle_step",
0.00613592332229))
self.ANGLE_MIN = float(rospy.get_param("~angle_min", -2.08621382713))
self.ANGLE_MAX = float(rospy.get_param("~angle_max", 2.09234976768))
...
```

It is also good to mention that the car detects obstacles with a laser sensor, making it stop when it detects an obstacle. The car's laser sensor is a node that is launched with the car's launch file.

We also created a package called 'mushr_ros_intro' that also contains launch files and nodes. However, it is not a package that is used to launch the car and the map. It is a package that contains nodes that we used to define the car's path and to visualize the car's path in Rviz. The structure of that package is as follows :

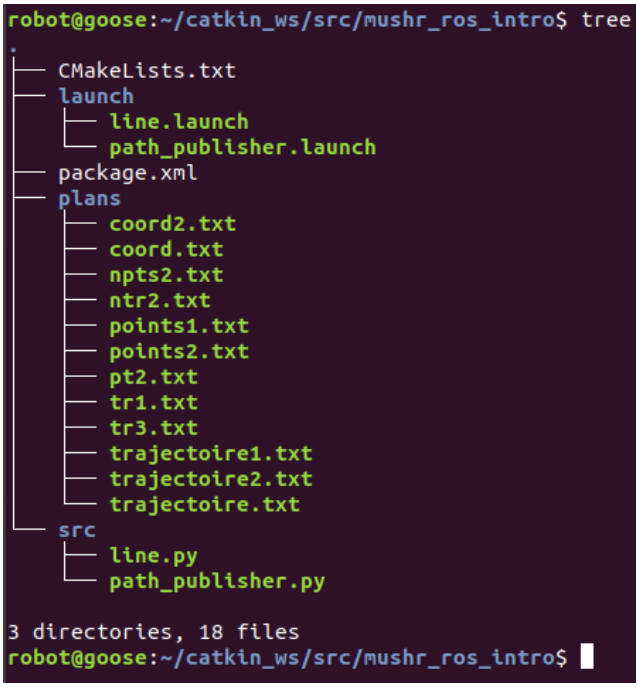


Figure 2.3: mushr_ros_intro

The '.txt' files contain the coordinates of the path that we want the car to follow. There are two types of these (as mentioned previously):

- The coordinates of the path that we want the car to follow (that we will print on the ground), with the format:

```txt
x0 y0, z0
x1 y1
x2 y2
    ...
```

- The values of the car's speed and steering angle (These are the relative values needed for the car to follow a path), with the format (the first line represents the coordinates of the starting point of the car):

```txt
x0, y0, z0
v0, w0
v1, w1
    ...
```

B - Application

The second basis of our project is therefore the application allowing us to draw trajectories. This application was first coded in python by a student from Polytech Nice-Sophia: HUOT-MARCHAND Antoine.

1 - Structure

Before anything else, we need to look at the libraries used. It requires the installation of numpy, PyQT5 and pyqt5_tools. The first one will be used to calculate the instructions given to the car and to manipulate arrays of data. The last two will allow the implementation of the graphical interface of the application.

/!\ Warning /!\ It is necessary to create a particular python environment for this project, because the pyqt5_tools library requires to have python in the 3.9 version at most (The last version of python is 3.11 at the writing of this report).

Now onto the code: The application is based on four different classes :

- Canvas
- QPaletteButton
- QButton
- MainWindow

Class diagram :

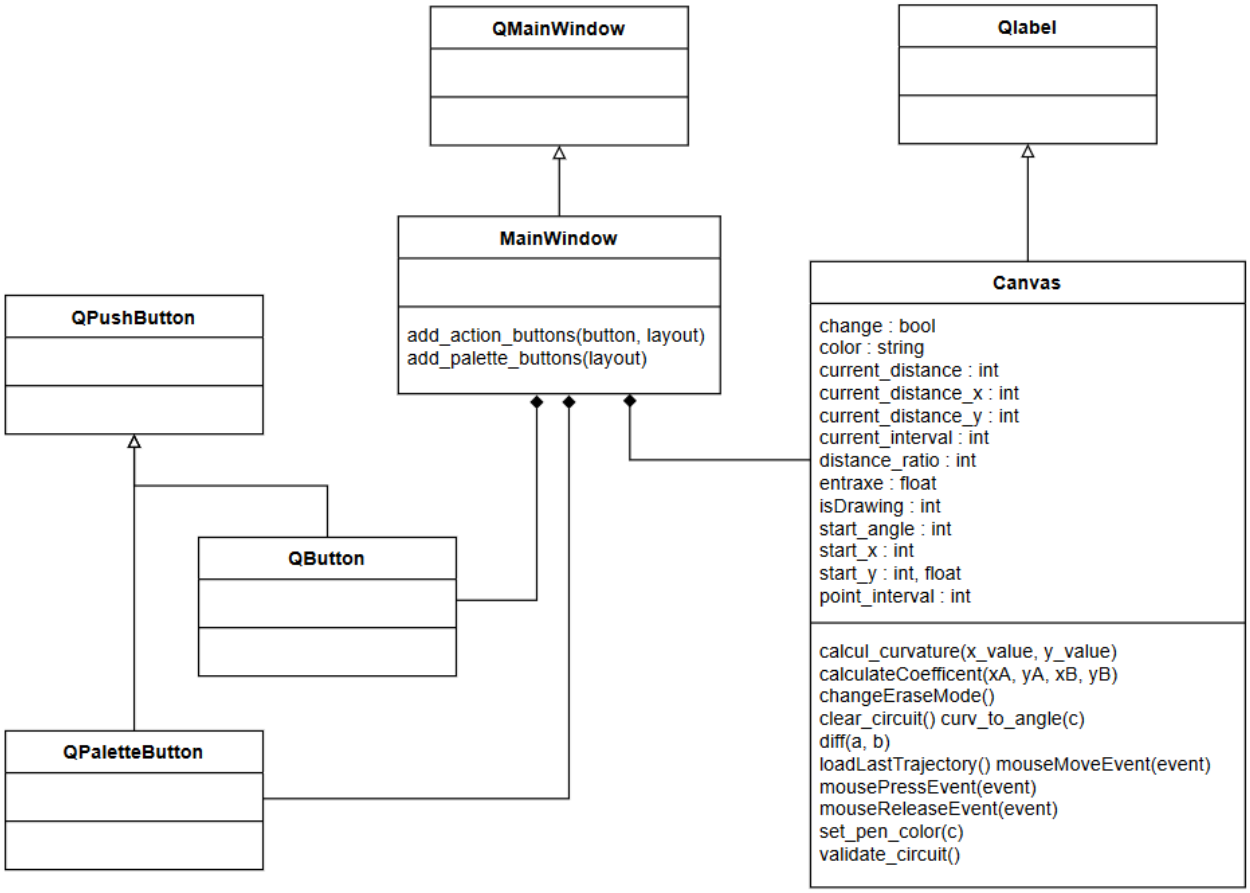


Figure 2.4: Class diagram of Antoine's code

We will mainly focus on Canvas first then on MainWindow because they are the two principal components of the application.

Canvas is the base of the customizable drawing space. It manages the pen used to draw onto the circuit, which is the background of the canvas. It also manages

all actions performed on the trajectories, such as load the previous trajectory, save the trajectory drawn or clear the canvas.

The canvas is the base for recording trajectories and saving them into files. Thus, we found the algorithm used to convert trajectories made of x and y coordinates to coordinates understandable by the car (so by the 3D interface).

We can define 3 mains group for Canvas's functions: the "Drawing" group, the "Calculation" group and the "File" group.

We find in the "Drawing" group every function used to define or alter the drawing:

- *mouseMoveEvent*
- *mousePressEvent*
- *mouseReleaseEvent*
- *set_pen_color*.

We find in the "calculation" group every function used to handle the conversion from the orthogonal coordinates of the drawing area to the coordinates made of speed and curvature:

- *calculate_curvature*
- *calculateCoefficients*
- *curve_to_angle*
- *diff*

Finally, we find in the "File" group every function used to link code and files:

- *Validate_circuit*
- *LoadLastTrajectories*
- *ChangeEraseMode*

Mainwindow is the class responsible for setting up and organizing the graphical interface. It supports all the layouts and widgets of the application. It generates the canvas, described previously, the color palette (because we can draw trajectories in different colors), and 3 buttons: "clear circuit", "valid circuit" and "load last trajectory".

Functions contained in the MainWindow class are used to add the buttons and the color palette to the main window.

Finally, QPaletteButton and QButton are two classes defining the property of the buttons and the color palette.

2 - Calculation of car trajectories

As stated in the introduction, the model we use to translate trajectories is the simple Bicycle model. Its goal is to be able to calculate the speed of the car and the wheels rotation of the car for a given curvature. Those data are required by Rviz's car and therefore the MIA's car to drive.

The model considers the car not as a four-wheel vehicle but as… well… a bicycle. This model models the front wheel of the bicycle as a third wheel placed between the two front wheels of the car and the rear wheel of the bicycle is placed between the two rear wheels of the car. We can schematize this model as follow:

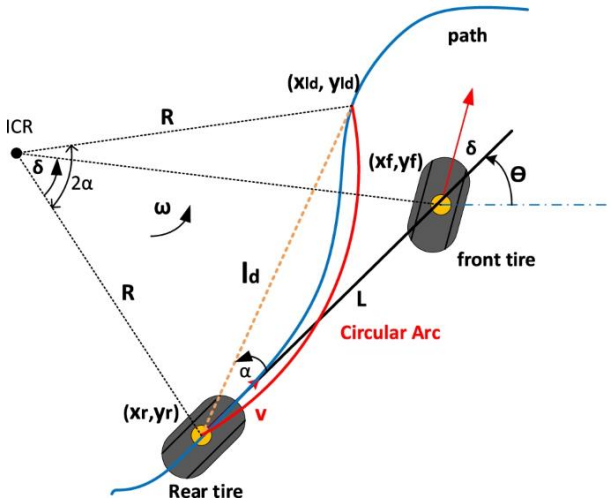


Figure 2.5: the bicycle model

The x and y coordinates of the trajectories are used to first calculate the speed using the formula:

$$v = \sqrt{(x')^2 + (y')^2}$$

Then used to calculate the steering angle of the front wheel of the bicycle. This angle is calculated via the formula:

$$\delta = arctan(\frac{L}{R})$$

Where L represents the interaxial of the car (the space between the front and the rear wheels) which is a unique constant dependant from the car model and R represents the radius of curvature. We can calculate the latter by remembering that:

$$R = \frac{1}{\kappa}$$

Where κ is the curvature, which formula is:

$$\kappa = \frac{|x''y' - y''x'|}{(x'^2 + y'^2)^{3/2}} = \frac{|x''y' - y''x'|}{v^3}$$

The usage of the numpy library becomes obvious when it comes to calculating gradients.

3 - Graphical interface

The basic interface of the application contains only the canvas, the color palette and the tree buttons used to clear the canvas, to validate a trajectory and to load the last trajectory:

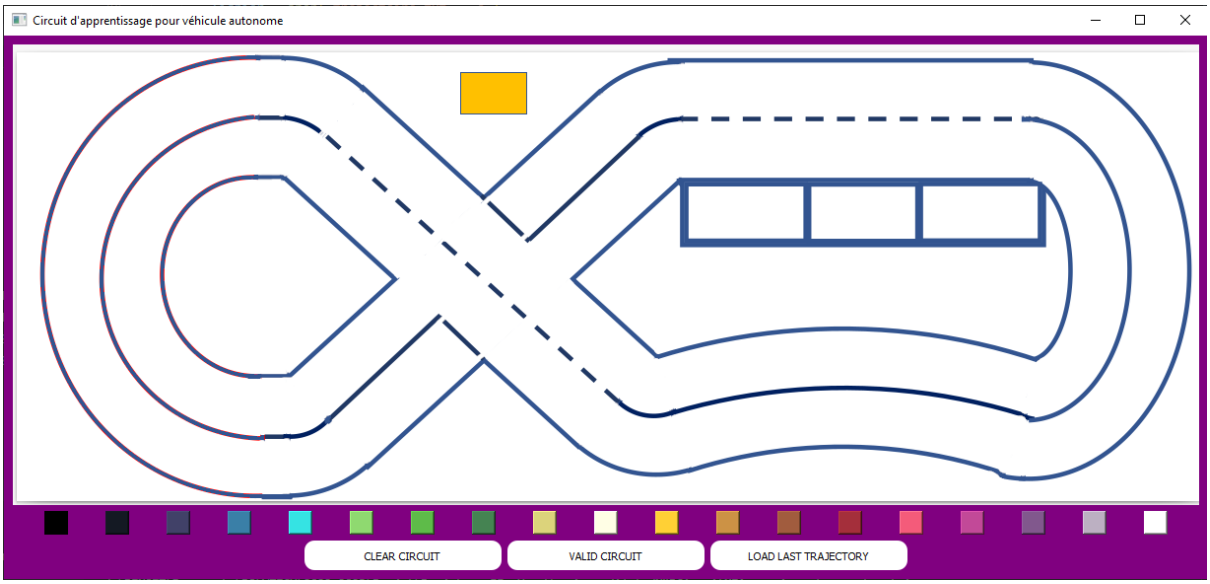


Figure 2.6: Initial interface

On the canvas, it is possible to draw every kind of trajectory we want, even though they are “non valid”, i.e those trajectories are out of the circuit, or even go over a wall. Only one trajectory can be stored at a time. Indeed, In the first version of the code, trajectories are saved in 2 different files: the first one is a file named “points.txt”, which stores the coordinates of the trajectory being drawn and the second one named “trajectories.txt”, which stores commands for the car.

The button clear canvas only deletes what is seen on the screen and doesn't erase the temporary file "points.txt", which can be a problem.

The main window is also not centered, which sometimes results in a part of the application being hidden out of the bounds of the screen.

III - Improvements

A - Application

The first part of our job on this project was to modify and enhance the application provided by Antoine.

1 - Constraints relative to the circuit

In the first version of the code, as stated earlier, all kinds of trajectory can be drawn. Obviously, this versatility can become a problem later on, mainly because the car can only drive unsectioned trajectories and can go through a wall.

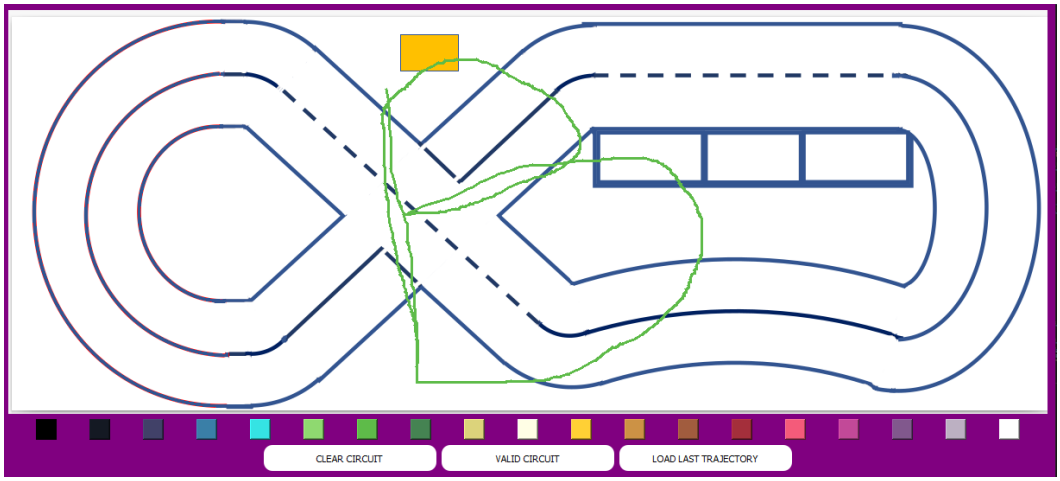


Figure 3.1: Valid trajectory for antoine's application
(Not quite right…)

Impossibility to draw composite trajectory

In order for the trajectory to be correct, it needs to be in a "single piece". Indeed, a car cannot drive multiple segments at a time.

In our first attempt at solving this issue, we thought a good idea was to limit the drawing of a curve to a single mouse press event. Precisely, we thought it would be normal to only click on the mouse, drag it to form a trajectory, release the mouse and to not be able to do anything else.

One of the main problems we experienced with this solution was that it became difficult to form long trajectories. Indeed, drawing with a laptop pad mouse can be exhaustingly difficult, forbidding you from being precise. Unfortunately, the laptop pad mouse was our only option. Moreover, a laptop is usually what is being used by researchers.

So, our answer to the problem was to make the mouse cursor being “teleported” to the last point being drawn. This way, if we stop drawing at a moment, we can always return to the trajectory we were making.

To do that, we had to add this parameter as argument to the canvas:

```
        self.very_last_point = QPoint(-1,-1)
```

Which store the very last point being drawn. This information is thus stored by the function mouseReleaseEvent, that gives it the position of the cursor when the mouse is released. Then the following condition is added to the mouseClickEvent:

```
        if(self.very_last_point != QPoint(-1,-1)) :
            self.cursor().setPos(self.mapToGlobal(self.very_last_point))
```

So that if there exists a very last point, we give its coordinate to the cursor.

Giving the wall their consistency

In order to prevent the car from hitting a wall, we need to prevent a trajectory from being drawn over a wall. The problem is that we are not drawing over a tangible circuit but over a simple image. Our first and only solution was to use the colors of the drawing, as we didn’t have anything else to work with.

In practice, we just needed to verify that the color under the cursor behind dragged is the same as the color of the starting point of the trajectory. This is a simple but effective way to be able to draw only within corridors. We implemented this verification within the mouseMoveEvent method (old version):

```
        if (self.my_qimage.pixel(event.pos()) ==
self.my_qimage.pixel(QPoint(self.start_x * self.distance_ratio,
self.start_y * self.distance_ratio)))
```

Where the left part of the equality represents the current color under the cursor and the left one represents the color behind the starting point of the trajectory. self.my_qimage is the circuit image converted into a QImage.

The main problem with this solution comes from the fact that the image of the circuit we have is composed of a lot of different colors, even for the background (the background color is composed of 2 or 3 different shades of light gray not really perceptible). But we figured out a solution that we will present in the next section.

Taking the car's width into account

Another problem with our model is that the trajectory is made for a bicycle and not a car. So, if the bicycle rides close to a wall, it is possible that the car already drives into the wall. The first solution found to take the width of the car into considerations, was to code a hitbox around the last point drawn:

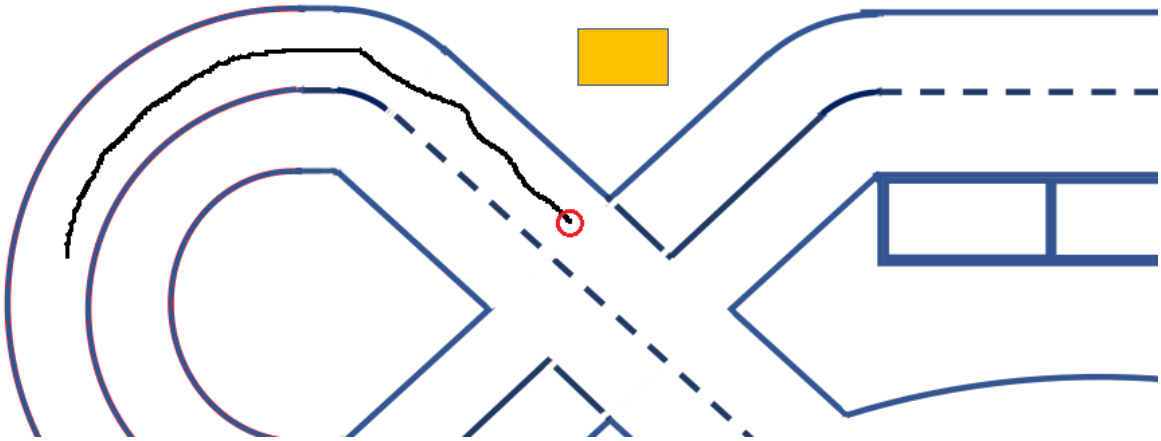


Figure 3.2: Visual representation of the hitbox

This solution would have been great, especially considering cars can have different widths and the radius of the hitbox can be set. The only problem is the calculation time because we need to calculate the hitbox for each point and to verify if some part of a wall is not in the hitbox. A complexity near $O(n^3)$.

The second solution that we chose was to create a modified version of the map. On this version, walls are stretched out to virtually simulate the width of the car. This new circuit is used as a secondary circuit where all tests are conducted. This solution is called a mathematical morphology of the circuit.

In order to achieve this morphology, we used a library called scikit-image, from which we've imported the modules io and morphology. Then, onto the modification: First, we convert that image into a grayscale one:

```
      io.imread('circuitMIA.png', as_gray = True)
```

Then, we calculated the erosion of the circuit (this is the part that transform the circuit) with a dilatation rate of 10 and with a circular form:

```
      self.dilation = 10
      self.my_dilatedimage                                                    =
morphology.erosion(io.imread('circuitMIA.png',    as_gray    =    True),
morphology.disk(self.dilation))
```

If we stopped the transformation here, we would still have the problem described in the last paragraph: we have an image with different gradients of gray. We needed to make this version as clear as possible. The solution: transform this image into a black and white only:

```
      self.my_dilatedimage = np.where(self.my_dilatedimage>=0.5, 1, 0)
```

The result is the following circuit:

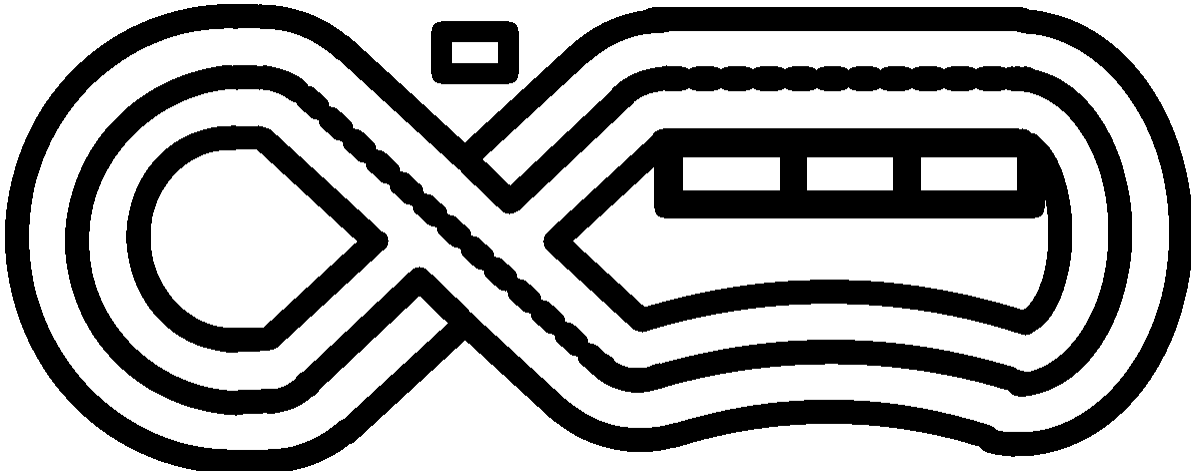


Figure 3.3: Black and white dilated circuit.

This circuit is used in the background of the system for the verifications:

```
      if (self.my_qdilatedimage.pixel(event.pos()) ==
self.my_qdilatedimage.pixel(QPoint(self.start_x * self.distance_ratio,
self.start_y * self.distance_ratio)))
```

The only problem remaining with that part is the management of stops and broken lines. We tried making broken lines disappear with the morphology without any success.

No more forbidden trajectories

Now that we had constrained our drawing area, the next step was to forbid the user from validating a “bad” circuit. In order to do that, we simply added a boolean called *self.valid_circuit*, initialized to false, and turned into true as long as the previous condition is true. If this condition is false, *self.valid_circuit* is turned to false and the function *validate_circuit* no longer saves the trajectory.

2 - Gestion of multiple trajectories

The second main problem encountered with the basic code was the fact that only one trajectory could be stored at a time.

Saving multiple curvature

The first thing we did was to make the application generate multiple trajectory files and point files per validated drawn curvature. Indeed, if we want the car to have a huge database, we need to be able to generate trajectory easily without restarting the system every time.

To generate those files without crushing the previous ones, we simply added a counter called *self.compteur*, which keeps a trace on the total number of files created. Then we create files as followed:

```
        nom_fichier_trajectoire=
"trajectoire"+str(self.compteur)+".txt"
        nom_fichier_points = "points"+str(self.compteur)+".txt"
```

These files are created when validating a trajectory, so the previous code is stored in the *validate_trajectory* method. But we also need to make sure to crush the files if they already exist in order to not generate bugs:

```
if (os.path.exists(nom_fichier_points)) : os.remove(nom_fichier_points)
            self.fichier_point = open(nom_fichier_points,"a")
```

This is also the case for all trajectory files. A good way that we could improve this system would be to ask the user to choose the names of his files directly in the application. It would be a good idea if the application is launched by different people in the same folder.

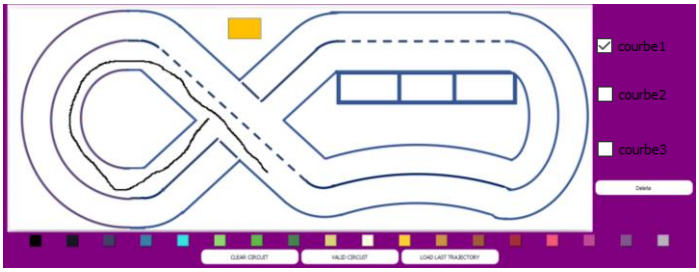
Managing the history of trajectories

Now that we created a system generating multiple trajectories, we needed to manage them directly through our interface. The simplest way we found to do it: a list of checkboxes.

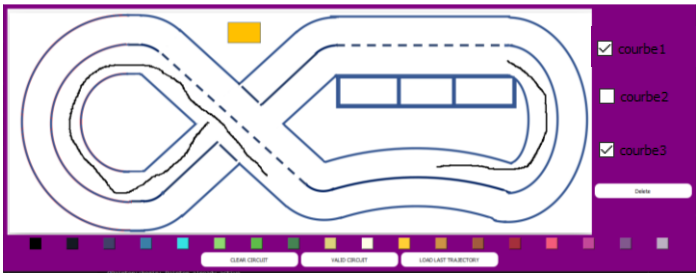
We copied the way Antoine was making his buttons: We created multiple classes in the MainWindow class to add (*addCheckbox*), delete (*deleteCheckBox*) and uncheck (*uncochedcheckBox*) the checkboxes. We know that those functions are still in developpement. Indeed, we still run into errors when trying to delete multiple trajectories at a time.

We found a way to add and delete checkboxes via indexes into the QVBoxLayer where they are stored, instead of recreating every checkbox whenever we add or delete a trajectory (so a checkbox). More about this method in part IV.B.2 of this report.

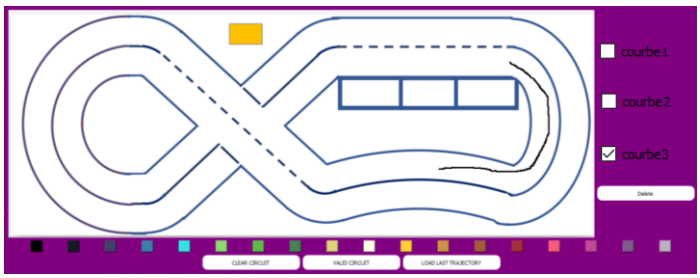
The most interesting part about this management is the function called when the checkboxes are checked: *checkboxChanged*. This method is the link between the MainWindow class and the canvas class. It calls the *loadChosenTrajectory* from the canvas class, with a given trajectory index. This allow us to see every trajectory we want that has been drawn so far:



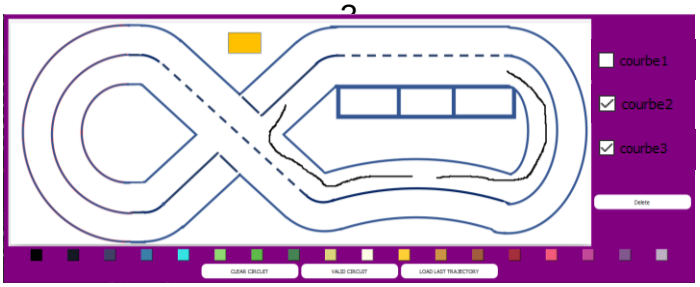
Trajectory 1



Trajectory 1 &
2



Trajectory 3



Trajectory 2 &
3

Figure 3.4: Checkboxes in use

We can also delete a trajectory using the delete button, which calls the *deleteCheckBox* method from MainWindow, which is linked to the *deleteTrajectory* method from canvas class. The latter simply delete all files concerning the specified curve.

B - Ros visualization: Rviz

During the project, we were asked to add a visual representation of a trajectory on “the ground” of our Ros visualization that would not be considered by the car as an obstacle. To do so, we created 2 new files: a launch file and a node. The launch file contains the command to launch the node. The node contains the code to add the trajectory to RViz.

1 - Coding the Launch files

The launch file contains all the commands to launch the node. It is a .launch file which actually is a xml file / uses the xml file syntax.

In the example below we want to plot the trajectory on the circuit. To do so, we need to pass the name of the file containing the coordinates of the trajectory as an "argument" (arg). The argument is passed to the node. The node will then use the argument to extract the coordinates of the trajectory.

We also need to declare the node. The node is declared with the command "node", the name of the package and the type of the node (the type of the node is the name of the file containing the code of the node):

```xml
<launch>
    <arg                name="plan_file"                default="$(find
mushr_ros_intro)/plans/trajectoire.txt" />
    <node      pkg="mushr_ros_intro"      type="line.py"      name="line"
output="screen">
        <param name="plan_file" value="$(arg plan_file)" />
    </node>
</launch>
```

2 - Trajectory not considered by the car as an obstacle

The following code corresponds to our node that draws the car's path onto the ground of the Rviz simulation. Our code includes 2 high level ros packages : visualization_msgs and geometry_msgs.

- visualization_msgs is a set of messages used by higher level packages, such as Rviz, that deal in visualization-specific data. (http://wiki.ros.org/visualization_msgs).
  The main message in visualization_msgs is a Marker. A marker message is used to "plot Markers" onto a visualization environment such as Rviz. We could "plot" things like boxes, spheres, arrows, and in our case, lines.

- geometry_msgs provides messages for common geometric primitives such as points, vectors, and poses. These primitives are designed to provide a common data type and facilitate interoperability throughout the system. (http://wiki.ros.org/geometry_msgs).

Because a line is a set of points, we needed to import the Point message from geometry_msgs.

```python
from visualization_msgs.msg import Marker
from geometry_msgs.msg import Point
```

In our node:

First, we had to initialize the node and create a publisher. The publisher will publish the line on the topic '/visualization_marker'. The publisher will send the line to the topic. The subscriber will receive the line from the topic.

Here our topic will consist in a Marker message - the line.

```python
rospy.init_node('line')
pub_line = rospy.Publisher('/visualization_marker', Marker, queue_size=1)
rospy.loginfo('Notre première trajectoire dessinée') # Allow us to check in the terminal that the node is running
```

Secondly, we had to add a loop to set the time of the execution of the node and initialize the Marker message. The Marker message is initialized with the type of the marker (LINE_STRIP), the scale of the line (0.03), the color of the line (yellow R = 1, G = 1, B = 0):

```python
marker = Marker() # Define the marker
marker.header.frame_id = "/map" # Define the frame - background map - background image
marker.type = marker.LINE_STRIP # could have also been "marker.type = 4" type number of the line
marker.action = marker.ADD

# Initializing the marker position with the initial position of the robot (coordinates are floats)
marker.pose.position.x = 0.0
marker.pose.position.y = 0.0
marker.pose.position.z = 0.0

# marker scale
marker.scale.x = 0.03
marker.scale.y = 0.03
marker.scale.z = 0.03

# marker color
marker.color.a = 1.0 # Else, the marker is transparent
marker.color.r = 1.0
marker.color.g = 1.0
```

```
marker.color.b = 0.0

# marker orientation (case of an arrow)
marker.pose.orientation.x = float(initial_position[0]) # See just below
marker.pose.orientation.y = float(initial_position[1])
marker.pose.orientation.z = 0.0
marker.pose.orientation.w = 1.0
```

Thirdly, we had to extract the points of the trajectory from the .txt file 'Points*.txt'. This file was created by modifying Antoine's code and contains all the coordinates of the trajectory that has been drawn.

```python
plan_file = rospy.get_param("~plan_file")

# coordinates file
with open(plan_file) as f:
    plan = f.readlines()
initial_position = plan.pop(0).split(",") # get the first line and split
it into a list
```

Lastly, we had to add the points of the trajectory to the Marker message and publish it on the topic '/visualization_marker'.

```python
for line in plan:
    pt = Point()
    line = line.split(",")
    pt.x = float(line[0])
    pt.y = float(line[1])
    pt.z = 0.0
    marker.points.append(pt)
    del pt
#rospy.loginfo(marker.points)

# Publish the Marker
pub_line.publish(marker)
```

3 - Results

The first attempts we made to understand the functioning of a Marker were very promising. When we understood how the markers were working, we decided to use : `marker.type = 3`. It was our first result but not quite a line…

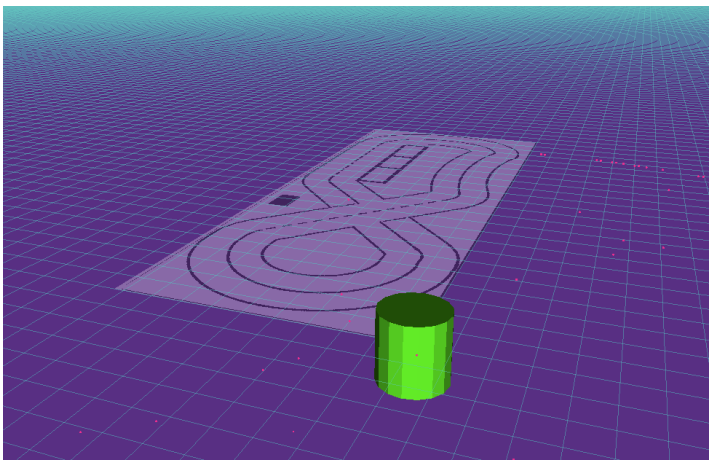


Figure 3.4: Cylinder (not quite a line…)

We then decided to use: `marker.type = 4`. Finally! a line! Looking promising, but we needed a trajectory.

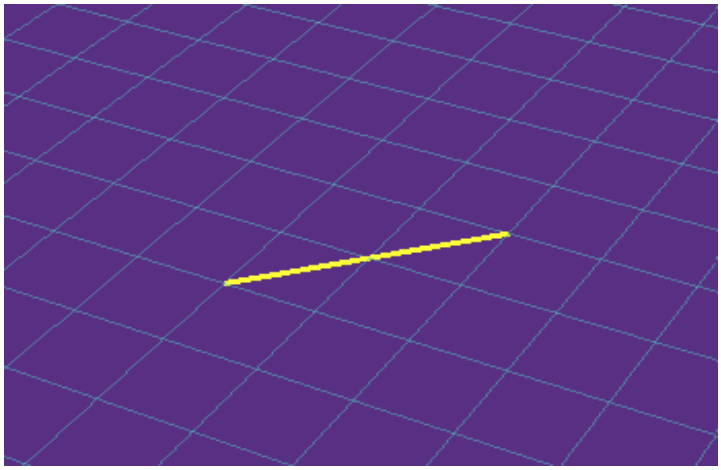


Figure 3.5: Line (not quite a trajectory…)

Then, we used the last loop explained in the previous part, and here is our first trajectory drawn:

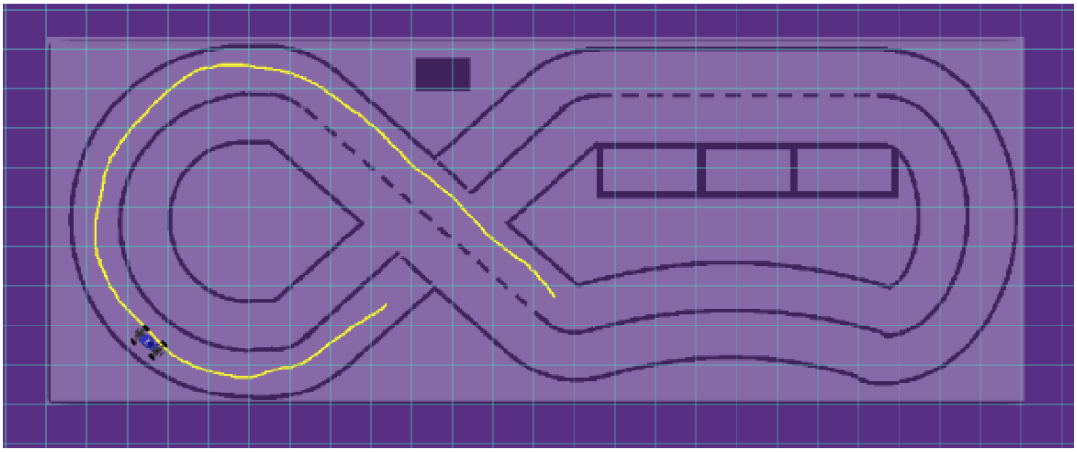


Figure 3.6: Trajectory (finally…)

4 - Testing

All of this had only one goal: to test the veracity of the bicycle model. We tested a lot, made a lot of adjustment, comparing different trajectories, and here are a few, not so good, attempts:

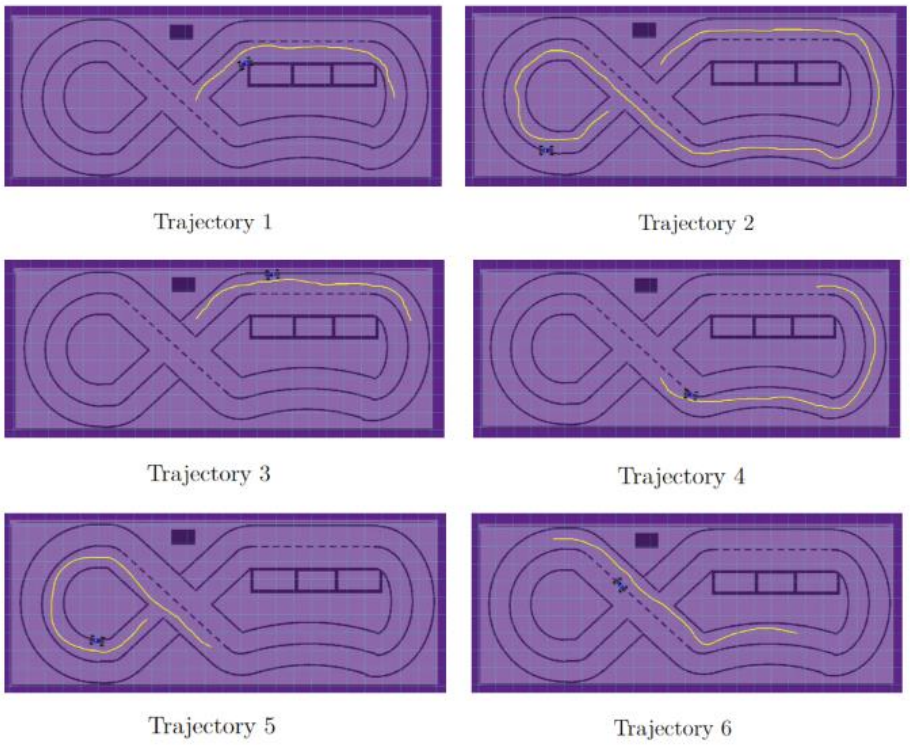


Figure 3.7: Failed attempts

We still manage to get a few good ones out of all of the failed ones:

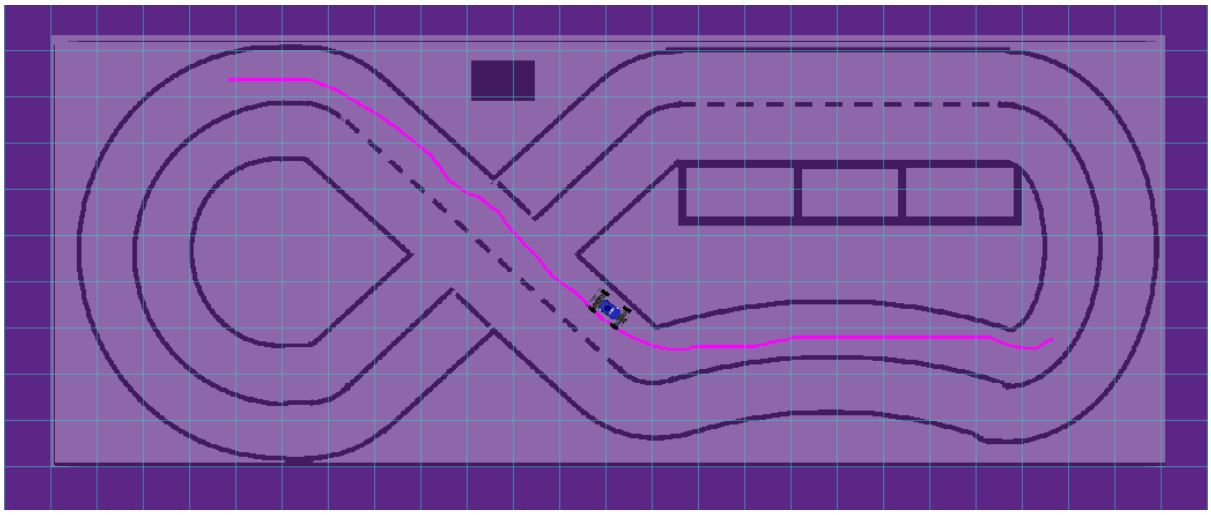


Figure 3.8: Good attempts

Our conclusion out of all the testing is that the imperfection on the curve does really matter because it creates steering angles too important for the car to follow along. Moreover, the speed calculated is too high compared to the size of the car relative to the size of the circuit. A lot of the time, it's what causes the car to crash into a wall.

The model is great. But it needs a reevaluation of the calculation of the speed, or the circuit compared to the size of the car needs to be enlarged.

IV - Theoretical improvement

This section is dedicated to the theoretical improvements that could be brought to the project.

A - Improving the car's "path planning" algorithm

1 - by modifying the existing code

Indeed, Antoine's algorithm is based on the following steps:

- Drawing the car's trajectory into a graphical user interface.
- Computing the input car's trajectory and expressing it with an initial position and some speed and steering angles values (measured every 30 points).

Nevertheless, this algorithm has some flaws.

Smoothing the trajectory

Two problems arise if we want to use this trajectory to compute the car's speed and steering angles with only a small amount of data points:

The complete trajectory is not smooth enough to compute the car's steering angles (this is due to the lack of precision of the human being). Therefore the car's steering angles, computed between two local points, could be too big or too small, making the car "turn too much" or "not enough". Moreover, if we get rid of a big part of the collected data points, we may get rid of a lot of "good" angles while keeping at least one "bad" angle.

Here's an example of a "straight line" made by a user:

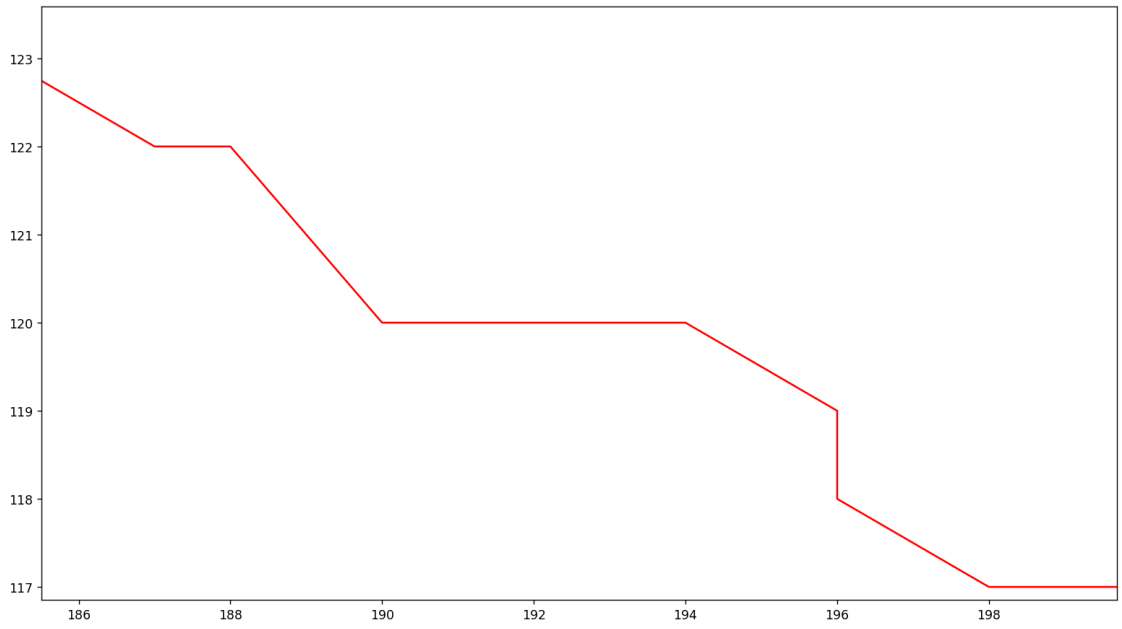


Figure 4.1: Zoom on a straight line made by a user

To dispose of this problem, we could use a smoothing algorithm to smooth the trajectory. In the example below, we used a Bezier curve to smooth the trajectory. We were able to compute this curve by using a specialized github repository (https://github.com/reiniscimurs/Bezier-Curve) which also happens to be able to consider the map's constraints.

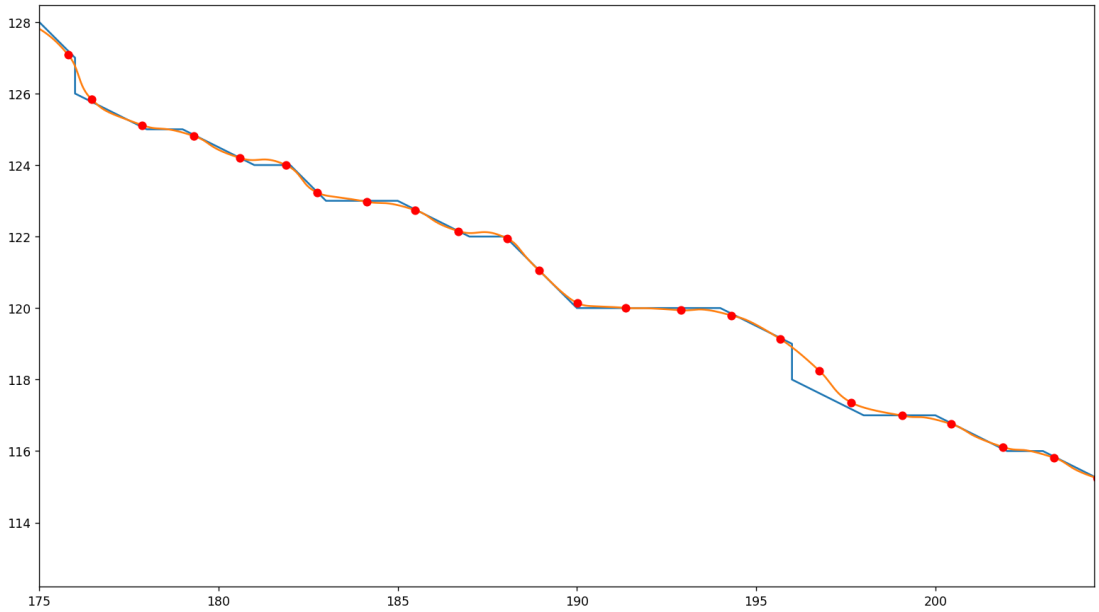


Figure 4.2: Bezier curve of the previous figure

However, this solution is not perfect because of the way the car' speed is computed (local - dividing the distance between two points by the time for the user to draw it). Approaching the car's trajectory with a smooth curve would reduce the angles errors, but it would change the car's speed (considering how much each new point would “diverge” from the old ones).

Improving the car’s speed calculation

The car's speed is computed by dividing the distance between two points by the time for the user to draw it. This is not a good way to compute the car's speed because the user can draw the line very slowly or very quickly. This would lead to a very slow or very fast car.

And our Beziers curves do not help us in this case. Sometimes spacing out the points in the trajectory, sometimes not. This would mean a higher or a lower speed of the car at some points (compared to its "original" speed measured at these points).

Another solution would be to consider the physics variables surrounding each car in an algorithm. We could get the cars properties (length, maximum steering angle, etc.) and the road's properties (friction, etc.) from some ros files, for example:

2 - With Ros's dedicated libraries

Ros/Rviz works with a lot of libraries that could help us improve the car's path planning. In this part we will show you all the libraries we tried to install (unsuccessfully).

Global Planner

https://mushr.io/tutorials/global_planner/

Global planner is a package that can be used to compute a path for a robot. It is based on the OMPL library, which is a C++ library for planning, with complex robot motion constraints. It is a collection of state-of-the-art sampling-based motion planning algorithms. We could use this library to compute a path for the car.

The problems we encountered were concerning the installation of the OMPL library. We tried to install it on our computers, but because of compatibility issues, we were not able to do so. Basically, we tried following the instructions on the MUSHR page however, the expected python version of our virtual environment was not the same as the one expected by the OMPL library (a depreciation error). We then changed some variables in the installation script to “make it fit" and tried to rewrite some parts of the newly installed library (unsuccessfully).

The last thing we wanted to try was to create a new virtual environment with the right python version (not enough time).

sbpl_lattice_planner

We also tried to use the sbplanner package. This package is a ROS wrapper for the SBPL library, a library for motion planning and motion detection in 2D and 3D environments.

In this case, it is a planner that "will generate a path from the robot's current position to a desired goal pose. Paths are generated by combining a series of "motion primitives” which are short, kinematically possible motions. Planning is therefore done in x, y, and theta dimensions, resulting in smooth paths that take robot orientation into account, which is especially important if the robot is not assumed to be circular or has nonholonomic constraints." (http://wiki.ros.org/sbpl_lattice_planner)

B - Improving the application

In this section, we will talk about improvements we wanted to make on the application but didn't have the time to make them happen.

1 - Checkboxes management

As said earlier, the current checkboxes management is not ideal. The solution we thought about to address this issue was to look into the resources given by QVBoxLayout, and by inheritance to the resources given by QBoxLayout.

We found out that there exists a function called *insertLayout*, that can… well… insert a layout into a QVBoxLayout at a given index. If this index is negative, the object is placed at the end. This is perfect to put the *delete* button at the end of the QVBoxLayout. Then, we just need to insert every checkbox at the $max\_index - 1$.

The problem is that we need that max_index. To keep track of the number of layouts that exist in the QVBoxLayout, we can use the *count* method provided by the QLayout class (we think that it doesn't need to be redefined, but we aren't sure 100%).

To load trajectories, there is normally no need to change anything, as that part doesn't affect the checkboxes themselves. But we need a way to solve the problem of deleting all the checkboxes checked when the *Delete* button is pushed. To do that, as we did on our version, we just retrieve the checked checkboxes, and we remove them within a foreach loop from the QVBoxLayer with the *removeItem* function inherited from QLayoutItem.

2 - Trajectory modification

One of the main problems we wanted to answer was "is it possible to modify a specific portion of a curve?". Indeed, with our version, when we hit a wall, it is not possible to retrieve the trajectory drawn. We need to draw it again. This can be quite painful if it happens on a very precise trajectory.

To make the modification happen we thought of a solution: using our trajectory navigation system (with checkboxes) to target a specific trajectory. Once the trajectory and the modifying mode selected, we could draw a rectangle (this is the part we are not sure how to do) to target the specific part of the trajectory we would want to erase.

Another function would go through the points*.txt file of the selected trajectory and search for every coordinate that would be within this rectangle. Then, it would

create two temporary files, one with all the coordinates before the changes and another with all the coordinates after the changes. Of course, we could display them on the application for more clarity.

Then, when drawing a new part for the trajectory, the cursor would be first teleported to the last point drawn for the continuity of the curvature. Then, with the new data, a new points*.txt file would be created. We could ask the user to press a button to finish their modification and replace the existing file or to finish their modification and create a new file as a copy. This would then be considered as a new trajectory.

V - Conclusion

To conclude this project, we can happily say that the bicycle model seems to be a good approximation for the calculation of the steering of the wheels of the car. But we need to find a solution for the speed calculation as it seems causing most of the error concerning the car precisely following the trajectory.

Smoothing the curve drawn seems to be another good idea to explore in order to have more logical steering command to give to the car, as drawn trajectories can be really shaky and imprecise, thus creating too many steering changes for the car to handle.

In regard to the adaptation of the model to multiple cars with different steering angles and interaxials, we can say for sure that the model can be adapted for different interaxials (as it is just a constant in our calculations). We think it can be adapted to different cars with different steering angles by approximating it with Bezier curve as we did in part IV and correcting some points if their position generates high steering angles.

As explained in the last part of our report, it seems highly possible that we will be able to add a trajectory modification functionality to our application. In fact, we are quite sad that a lot of our ideas could not see the light. It was a very interesting project, but not enough time to do it resulted in unfinished ideas

You can find every document we worked with and on on our github: https://github.com/LGPolytech/Projet_S7.