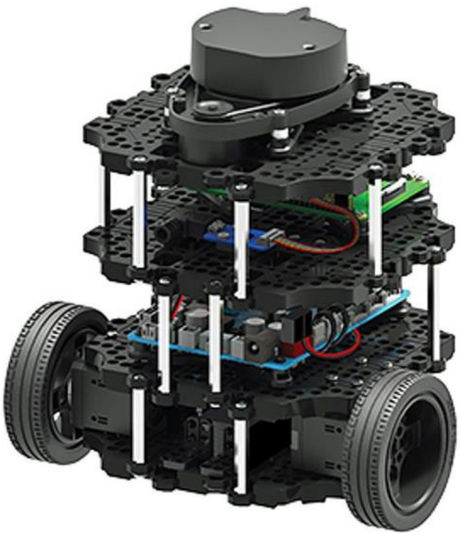
NavigateT project

Weekly updates



Students: Marc CHAMBRÉ, Dhia JENZERI, Loan MICHAUD, Reda TRICHA

Training: Electronics, Informatics & Systems

Teachers: Ionela PRODAN, Simon GAY

School year: 2023 - 2024

TABLE OF CONTENTS

Introduction

The goal of this document is to recap our work week-by-week. It will sum up briefly what we have done in the week table below and detail a bit more, most of the steps further.

This project monitoring is as useful for us as it is for anyone who wants to check the advancement of the project.

1. Week table

Week	Description
[38]	- Meeting with I. PRODAN about the main objectives - Meeting & demonstration with G. SCHLOTTERBECK about ROS and Gazebo - Meeting with S.GAY about the bio-inspired model - Configuring the working environment on the PC - Personal research about ROS & Gazebo
[39]	- Configuring the working environment on the PC and our personal ROS environments - Receipt of material (TurtleBot + cables) - Simulations of Turtlebot using Gazebo
[40]	- Communication between PC and Raspberry configured. - Controlling the real TurtleBot with a keyboard - Reading of existent navigation algorithm - Research about a mathematical model of the robot
[41]	- P controller for static point, linear, quadratic, and ellipsoidal path following - First node to control the TurtleBot
[42]	- P controller python implementation in progress. - Improvements and creations of nodes to retrieve data from the Lidar sensor and control the TurtleBot. - Problem of getting coordinates solved.
[43]	- Simultaneously retrieve coordinates as the turtlebot moves forward - Try to code the P corrector in Python and make it compatible for ROS 2 - Try out a Python algorithm on the TurtleBot to deviate its direction when an obstacle is present.
[44-46]	Done: - Improvements of our ROS node P controller on the real Turtlebot - Generating random trajectories for testing on the real TurtleBot - Controller P trajectory tests with the real TurtleBot - Creation of a GitHub repository Objectives for next week: - Keep improving our self-made nodes. - Try to solve the problems with the PI controller on the Turtlebot - Try to retrieve current location of the Turtlebot at Easynov - Try to implement obstacles avoidance with LIDAR sensor

	- Improve the controller (PID, **feedback linearization**)

2. Week 38 – 39: Initialization of the project

2.1 Main objectives

As a first goal, we will have to understand how to work with ROS and Gazebo, necessary to start controlling or simulating the TurtleBot.

Once we have installed all the prerequisites to set up the workspace on our computer, we will try to make the robot go from a point to another by itself, without any obstacle and with a simple but consistent command.

Then, we will work with Ionela PRODAN to design a more complex command law, such as a PID. In parallel, we will work with Simon GAY to help design navigation algorithms.

2.2 Presentation about ROS & Gazebo

To start, we had a presentation given by Guillaume SCHLOTTERBECK, about ROS and GAZEBO.

The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. For that, we can create nodes that publish or listen to topics, transfer data from sensors to the controller, and control actuators. The working principle is robust because several nodes already exist, including TurtleBot3 which are using, and many others. Then we can assemble nodes to make up our application.

Gazebo is a simulator for ROS nodes that allows us to do 3D simulations, with obstacles and robot models. This is a convenient tool to first test our nodes virtually.

2.3 Installation of the working environment (Ubuntu, ROS & Gazebo)

We started by adding a new Ubuntu session on our project computer, to have a centralized workspace for our simulations and tests.

Then we installed *ROS: humble* and *Gazebo: Garden* to start working on preliminary tests. We successfully controlled the robot simulation in Gazebo, as Guillaume did in his presentation. This is some views of Gazebo interface with the *Turtlebot* mapping its environment thanks to LIDAR sensor:

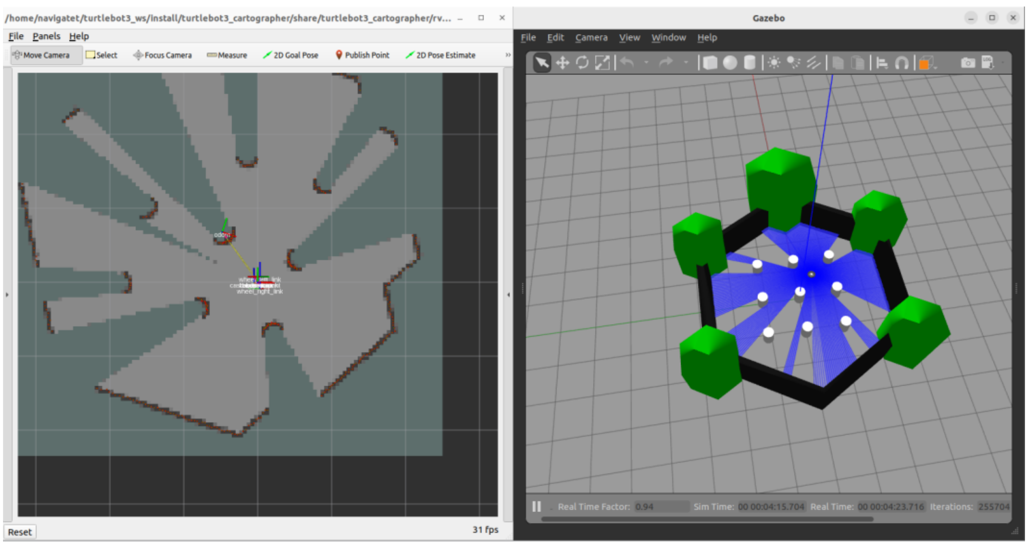


Figure 1 - TurtleBot simulation and mapping thanks to the embedded LIDAR

3. Week 40: TurtleBot control, mathematical model

3.1 Remote TurtleBot control

We started to control the real TurtleBot using a hotspot and a *ssh* link. We also tried to map out our project room with RViz.

One of our main issues was to understand how the connection setup between the hotspot and the Raspberry Pi card should be to work well, we had encountered some problems when trying to connect the Raspberry Pi to the hotspot (sometimes the connection crashes and we have to reboot everything in order for it worked again). We also tried to configure our hotspot on our personal computers to have better graphic performances on Rviz software. For now, it's not working, but it's a future goal.

We started to explore existing navigation algorithms among which we recognize things like 3rd order trajectory equations.

We started exploring the possibility of feedback control of a ROS-enabled robot using *Matlab* and *Simulink*. This will enable us to run a model that implements a simple closed-loop proportional controller for mobile robot trajectory tracking.

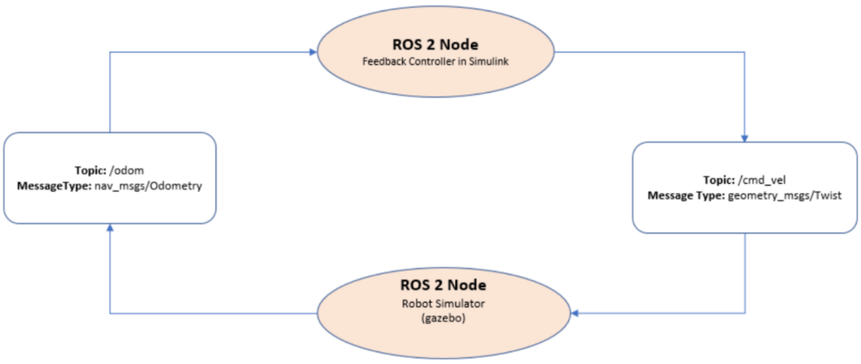


Figure 2 - ROS feedback controller

3.2 Mathematical model

We started to research a mathematical model for the robot, to have a basis for a controller.

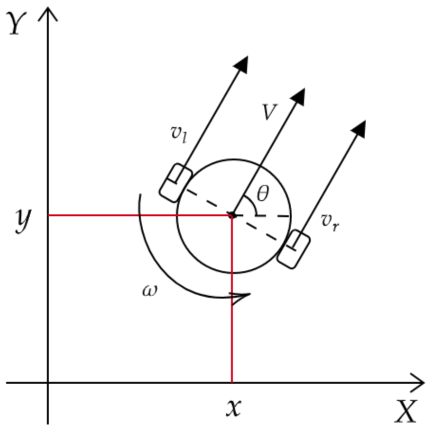


Figure 3 - Turtlebot modelization in a cartesian coordinate system.

We can write the dynamics of the robot as follows:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} V\cos\theta \\ V\sin\theta \\ \omega \end{bmatrix}$$

Then, the state vector would be:

$$x_{\text{state}} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

Let's express the speed V_{wheel} of one wheel:

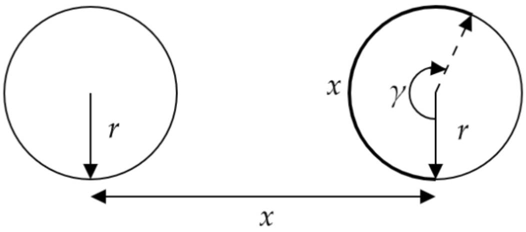


Figure 4 - Distance traveled by a r-radius wheel

We know that the distance traveled by a r-radius wheel is:

$$x(t) = \gamma(t).r \quad \begin{cases} \gamma(t)\text{: angle traveled at time t [rad]} \\ x(t)\text{: distance traveled at time t [m]} \end{cases}$$

Then, we can express the speed as:

$$\dot{x} = \dot{\gamma}r \rightarrow V_{\text{wheel}} = \omega_{\text{wheel}}.r \quad \begin{cases} V_{\text{wheel}}\text{: speed } [\text{m.s}^{-1}] \\ \omega_{\text{wheel}}\text{: angular velocity } [\text{rad.s}^{-1}] \end{cases}$$

Now we can express the components of the state vector as a combination of both right and left motor angular velocities, respectively u_r and u_l. We must consider r [m], the radius of each wheel, and L [m], the length between the center of the two wheels.

We assume that the linear velocity of the robot is written as:

$$V = \frac{V_l + V_r}{2} = \frac{r(\omega_l + \omega_r)}{2}$$

And the angular velocity is written as:

$$\dot{\theta} = \frac{1}{L}(V_r - V_l) = \frac{r}{L}(\omega_r - \omega_l)$$

Finally, we can rewrite the dynamic of the state vector as:

$$x_{\dot{\text{state}}} = \begin{bmatrix} \frac{r(\omega_l + \omega_r)}{2}\cos\theta \\ \frac{r(\omega_l + \omega_r)}{2}\sin\theta \\ \frac{r}{L}(\omega_r - \omega_l) \end{bmatrix}$$

We can validate the model by using cases, like:

$$\begin{cases} \omega_l = u_r \\ \omega_l > 0 \end{cases} \rightarrow \begin{cases} \dot{x} = r.\omega_l \cos\theta \\ \dot{y} = r.\omega_l \sin\theta \\ \dot{\theta} = 0 \end{cases}$$

The robot moves forward at the speed of one motor. There is no angular velocity since the robot goes straight forward.

$$\begin{cases} \omega_l = -\omega_r \\ \omega_l > 0 \end{cases} \rightarrow \begin{cases} \dot{x} = 0 \\ \dot{y} = 0 \\ \dot{\theta} = \frac{2r}{L}\omega_l \end{cases}$$

The robot turns on itself without any linear velocity. As the wheels velocities are opposed, the robot turns at two times a wheel speed.

$$\begin{cases} \omega_l = 0 \\ \omega_r > 0 \end{cases} \rightarrow \begin{cases} \dot{x} = r.\frac{\omega_r}{2}\cos\theta \\ \dot{y} = r.\frac{\omega_r}{2}\sin\theta \\ \dot{\theta} = \frac{r}{L}\omega_r \end{cases}$$

The robot turns around its left wheel since it is immobile. We see that $\dot{\theta} > 0$: As only the right wheel works, the robot turns through the trigonometric way. Only one wheel is working, so the linear velocity is divided by two.

3. Week 41: First node and P-controller

During this week, we decided to split into two teams to increase our productivity. It was a success because we achieved the two main objectives that we fixed, coding our first own node, and designing a controller in Simulink.

3.1 Creation of nodes

Now we know how to control the TurtleBot remotely, we try to code in Python some nodes to give instructions for control. Thus, we have created our workspace in Visual Studio to code nodes in Python. The first node we have created gives to the TurtleBot a command to go straight ahead during 10s along the x axis. It worked both in simulation and with the real TurtleBot. The Python node code is available in the **Appendix 1**.

The next step for us is to map an area thanks to the lidar sensor and the ROS software (NViz). We have done it with Ionela's PC but we noticed that the map result is not really good because of graphical performances. It's for that, we hope the new computer will be available next week to install our software tools and try the mapping again. Also, to save time, we did a script Shell which contains all Linux commands required to install all software tools (ROS2, Gazebo, TurtleBot3 simulation, etc.). It will be useful once we will have the access to the new computer and install all software tools we require.

Finally, once we will map an area, we will try to work on the position of the *Turtlebot* in its environment. So, we will try to create a node for moving the TurtleBot from point A to B.

3.2 P-controller on Simulink

Thanks to our mathematical model, we started to design a controller.

We previously found out this relation:

$$\begin{bmatrix} V \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ \frac{r}{L} & -\frac{r}{L} \end{bmatrix} \begin{bmatrix} \omega_r \\ \omega_l \end{bmatrix}$$

The vector $\begin{bmatrix} \omega_r \\ \omega_l \end{bmatrix}$ being the command of the model, we had to calculate it by reversing the previous expression:

$$\begin{bmatrix} \omega_r \\ \omega_l \end{bmatrix} = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ \frac{r}{L} & -\frac{r}{L} \end{bmatrix}^{-1} \begin{bmatrix} V_{ref} \\ \omega_{ref} \end{bmatrix} = \begin{bmatrix} \frac{1}{r} & \frac{L}{2r} \\ \frac{1}{r} & -\frac{L}{2r} \end{bmatrix} \begin{bmatrix} V_{ref} \\ \omega_{ref} \end{bmatrix}$$

We generated the reference vector $\begin{bmatrix} V_{ref} \\ \omega_{ref} \end{bmatrix}$ like this:

First, the reference velocity is generated using a proportional gain.

$$v_{ref}(t) = \begin{bmatrix} v_x^r(t) \\ v_y^r(t) \end{bmatrix} = k_p\big(p^r - p(t)\big) = k_p \begin{bmatrix} x(t) - x^r \\ y(t) - y^r \end{bmatrix}$$

Then, we take the module and the argument of this $v_{ref}(t)$ to get the linear velocity V_{ref} and the angle θ_{ref}.

$$V_{ref}(t) = \|v_{ref}(t)\| = k_p\sqrt{(x^r - x(t))^2 + (y^r - y(t))^2}$$

$$\theta_{ref}(t) = \arg\left(v_{ref}(t)\right) = \text{atan2}(y^r - y(t); x^r - x(t))$$

Finally, we generate ω_{ref} with another proportional gain:

$$\omega_{ref} = k_\theta\left(\theta_{ref}(t) - \theta(t)\right)$$

Now, we generate the command vector with the relation previously determined:

$$\begin{bmatrix} V_{ref} \\ \omega_{ref} \end{bmatrix} = \begin{bmatrix} \frac{1}{r} & \frac{L}{2r} \\ \frac{1}{r} & -\frac{L}{2r} \end{bmatrix} \begin{bmatrix} V_{ref} \\ \omega_{ref} \end{bmatrix}$$

<u>The block diagram is the following:</u>

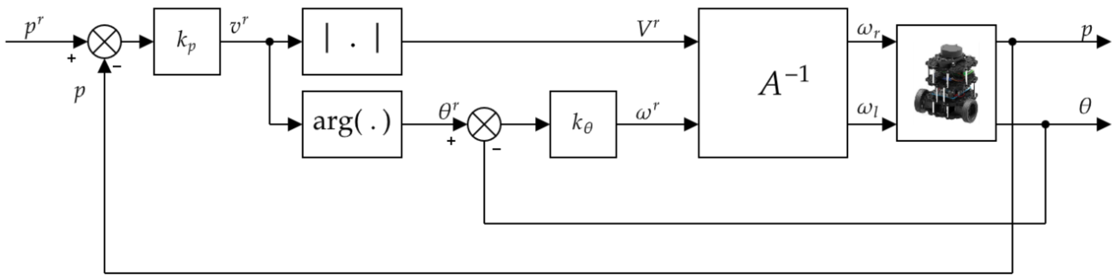


Figure 5 - Schematical control loop for the Turtlebot

<u>We built this control loop in Simulink & Matlab:</u>

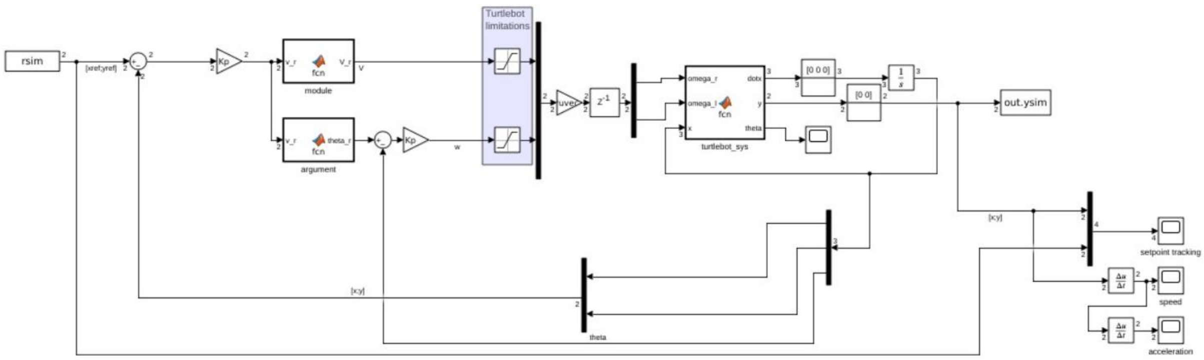


Figure 6 - Simulink model of our P-controller

The Matlab corresponding code is available in the **Appendix 2**.

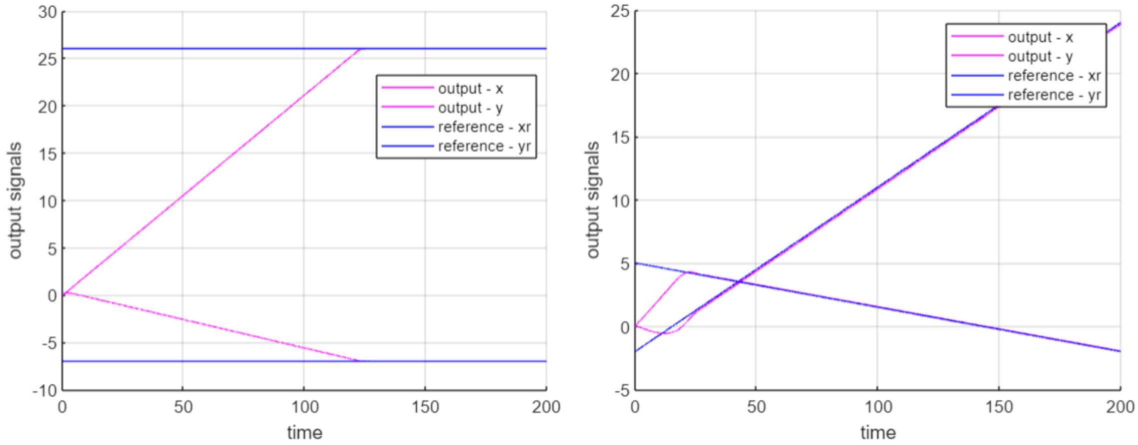


Figure 7 - Setpoint tracking. (On the left, constant setpoint reaching, on the right, linear path following)

<u>This is the final result of the Matlab code:</u>

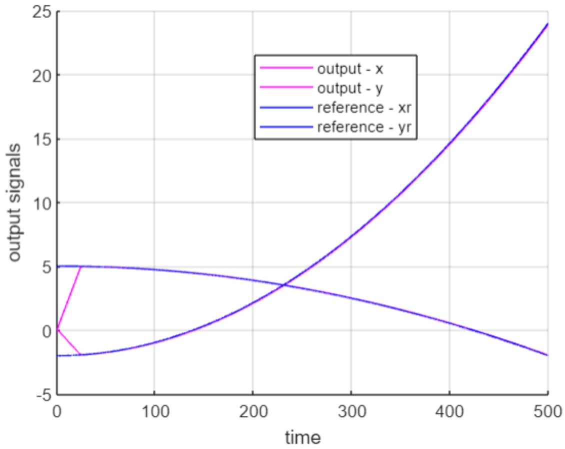


Figure 8 - Quadratic path following

We tried an ellipsoidal path, but it didn't work well because we had a few issues with the $\operatorname{atan}\left(\frac{y}{x}\right)$ function. We decided to use the $\operatorname{atan2}(y, x)$ function because it considers the signs of x and y independently so the output angle is more relevant.

Therefore, there still are some problems like drops from π to $-\pi$. To solve this problem, we used the function $\mathrm{wrapTo2Pi}(\theta)$ so the output angle stays between 0 rad and 2π rad.

The last problem was the drop when the angle was greater than 2π rad so it dropped to 0 rad because of the previous function. The last thing we did to solve the issue is to use the unwrap function. This function adds or retrieves 2π to the angle so that the difference between the current angle and the previous one is less than π. With this function, there is no drop, and the angle is allowed to be bigger than 2π.

You can see below the elliptical path tracking finally working:

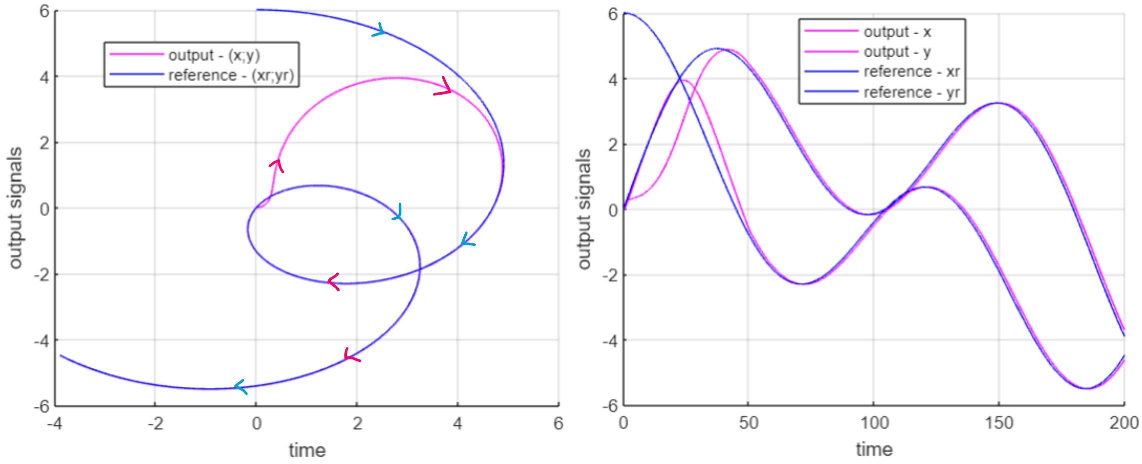


Figure 9 - Ellipsoidal path tracking

4. Week 42: P controller implementation, coordinates supply

This week, the “Forum des entreprises” took place in ESISAR so we couldn’t work together on Tuesday. That’s why we couldn’t progress a lot on the project, but we still tried to fulfill new objectives.

4.1 P-controller implementation

We started to implement the controller in a ROS node. We based on the Simulink model to transcript it into python. We hope that the node will be finished next week and we could test it on the Turtlebot.

We remarked that the step of converting linear speed and angular speed into left and right motor angular speed (Figure 10) is actually embedded in ROS using the class Twist().

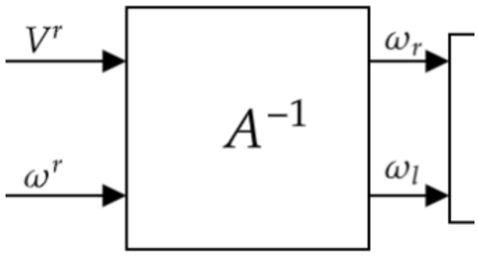


Figure 10 - Conversion between linear and angular speeds into right and left motors angular speeds.

4.1 PID-controller tuning

We tried to tune a PID controller for the Simulink model. But we don’t really see the difference with a P controller and with a PID controller, even for fault rejections. Maybe it is because the model doesn’t consider acceleration limitations, so it works “too well” with a P controller.

Maybe we should try to linearize it with feedback linearization and calculate a custom PID.

4.2 Coordinates supply

We visited the platform at Esynov and we will try to setup the link between the system and the Turtlobot next week, using existing ROS nodes. Thanks to the cameras in the Esynov room, we can accurately estimate the Turtlebot's coordinates in the space in which it is located.

Actually, the turtlebot embeds coordinates computing. We did some experiments with these computed coordinates, and we understood that they are only calculated thanks to the wheels speed. Therefore, it doesn’t consider wheel grip (what we could expect), but it also doesn’t remark whenever the turtlebot is stuck against a wall or whenever we lift it.

This system will nevertheless be useful to do some preliminary tests in our room before testing at Esynov.

5. Week 43: P controller implementation in Python and tries on MPC

This week, Tuesday was dedicated to a session with Jean-Pierre CEYSSON to talk about the notion of innovation, both in the innovation projects we are currently working on, and in the future engineering projects in which we will be involved. As a result, we didn't make much progress on our objectives for the week. Nevertheless, as on every Thursday since the start of the project, we're trying to schedule an afternoon session. The objective of this session, and of the week in general, was to code in Python the P controller that the automation team had set up, and to test some self-made obstacle avoidance algorithms.

The aim over the next few weeks, after the week's teaching break, is to get these algorithms running on the *Turtlebot* and then carry out navigation tests in more complex, unfamiliar environments.

5.1 P controller implementation

We started implementing the P controller last week, but we're having problems running it. This means we can't test the code on the *Turtlebot*. We think these problems are mainly since the PC we're working on doesn't have the same versions we use on our personal computers. It's true that for the code and simulation part, we use our personal computers a lot and it's not easy to manage compatibility between the configurations of each computer, but also the version already implanted in the *Turtlebot*'s SD card.

5.2 Tries on obstacle avoidance

This week, the development team also tried out a Python algorithm on the *Turtlebot* to deviate its direction when an obstacle is present. The prototype code is available in the **Appendix 3**.

This algorithm worked quite well in simulation on our personal computers. However, when we wanted to test it on the computer linked to the *Turtlebot*, the algorithm failed to start. We believe this is due to the previous remarks we made about the differences between the software versions on our personal computers and the tests we carried out on the computer linked to the *Turtlebot*.

5.3 Feedback Linearizing Controller

The automation team started developing a feedback linearization controller for the Turtlebot that will allow it to track a trajectory q(t). We recall that the dynamics of the Turtlebot are described by:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}$$

Where the state vector of the system is $\xi = [x \quad y \quad \theta]^T$ and the input vector is $\xi = [v \quad \omega]^T$, where v is the linear velocity input of the Turtlebot and ω is the angular rate of the Turtlebot. To design a tracking controller for this turtlebot system, we can use the techniques of feedback linearization, which will cancel out the nonlinear dynamics of the Turtlebot.

To get started in designing a feedback linearizing controller, we notice that this system is control-affine, and may be written in the form:

$$\dot{\xi} = f(\xi) + g(\xi)u$$

Where q, u are the state and input vectors described above and $f(\xi) = 0$. Our goal in designing a feedback linearizing controller for this system is to find an input u that creates a linear relationship between the input vector, u, and the output of the system, which we define to be the vector:

$$y_{\text{out}} = \begin{bmatrix} x \\ y \end{bmatrix}$$

If we can accomplish this, we can easily control the $(x; y)$ coordinates of the Turtlebot.

If we were to try and directly design a feedback linearizing controller for this system, however, we would find that a matrix we would need to invert to cancel out the nonlinear terms in the dynamics would not be invertible! To get around this, we apply dynamic extension, and rewrite the system in the following equivalent form:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = f(\tilde{\xi}) + g(\tilde{\xi})w$$

Where we append v to the state vector of the system to form the dynamically extended state vector $\tilde{\xi} = [x \quad y \quad \theta \quad v]^T$. In the extended system, instead of controlling the system with our original input, $u = [v \quad \omega]^T$, we use the new input vector:

$$w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} \dot{v} \\ \omega \end{bmatrix}$$

Where we control the derivative of input velocity instead of velocity itself. Once we have the system in this extended form, we can prove that the following relationship exists between input and output:

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} \cos\theta & -v\sin\theta \\ \sin\theta & v\cos\theta \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = A(\tilde{\xi})w$$

Here, the matrix $A(\tilde{\xi})$ is always invertible if $v \neq 0$. By picking $w = A^{-1}(\tilde{\xi})z$, where $z \in R^2$, we may derive the following linear input-output relationship.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \ddot{x} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} z$$

This is a linear system of the form $\dot{\xi}' = A\xi' + Bz$ where $\xi' = [x \quad y \quad \dot{x} \quad \dot{y}]^T$

The following flow-chart describes the process your controller design should take:

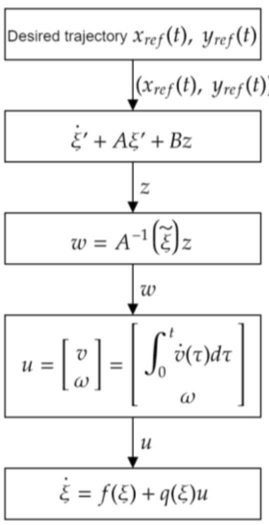


Figure 11 - Feedback linearization control strategy

First, we define the desired $(x_{\text{ref}}(t), y_{\text{ref}}(t))$ trajectory for our system. Both x_{ref}, y_{ref} are differentiable functions of time that describe where we'd like our turtlebot to be at all times. We send this desired trajectory to the feedback linearized system, $\dot{\xi}' = A\xi' + Bz$, which we defined above. We may then use linear control design to pick an input z that allows the system to track the desired trajectory. Once we have this value of z, we convert it back into w using $w = A^{-1}(\tilde{\xi})z$, which came from the feedback linearizing relationship. Once we have w, we may integrate the first component of w,$w_1 = \dot{v}$ in time to find the value of velocity, v, we should send to our original system:

$$u = \begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} \int_0^t w_1\,(\tau)d\tau \\ w_2 \end{bmatrix}$$

Finally, once we have this value for the input, we send it to our original system:

$$\dot{\xi} = f(\xi) + g(\xi)u\dot{\xi} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} u$$

This will enable our original nonlinear system to track the desired trajectory. Following this procedure, we will design a feedback linearizing tracking controller for the Turtlebot.

During the upcoming sessions, we will start implementing a feedback linearizing tracking controller following this procedure.

5.4 MPC tries

Following the *MPC* course, the automation team tested to control the *Turtlebot* with *MPC*. The discretization of the model is the following:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\phi} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v\cos\phi \\ v\sin\phi \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = f(\tilde{q}) + g(\tilde{q})w$$

$$\dot{\xi}(k) = \begin{bmatrix} \dot{x}(k) \\ \dot{y}(k) \\ \dot{\theta}(k) \end{bmatrix} \approx \frac{1}{T_s} \begin{bmatrix} x(k+1) - x(k) \\ y(k+1) - y(k) \\ \theta(k+1) - \theta(k) \end{bmatrix} = \begin{bmatrix} V(k)\cos\big(\theta(k)\big) \\ V(k)\sin\big(\theta(k)\big) \\ \omega(k) \end{bmatrix}$$

$$\xi(k+1) = \begin{bmatrix} x(k+1) \\ y(k+1) \\ \theta(k+1) \end{bmatrix} = \begin{bmatrix} x(k) \\ y(k) \\ \theta(k) \end{bmatrix} + T_s \begin{bmatrix} V(k)\cos\big(\theta(k)\big) \\ V(k)\sin\big(\theta(k)\big) \\ \omega(k) \end{bmatrix}$$

$$u_{\max} = \begin{bmatrix} V_{\max} \\ \omega_{\max} \end{bmatrix} = \begin{bmatrix} 0.22\ \text{m.s}^{-1} \\ 2.84\ \text{rad.s}^{-1} \end{bmatrix}$$

We can then implement this to get *MPC* from *casadi* optimization solver. It worked well for circular reference tracking:

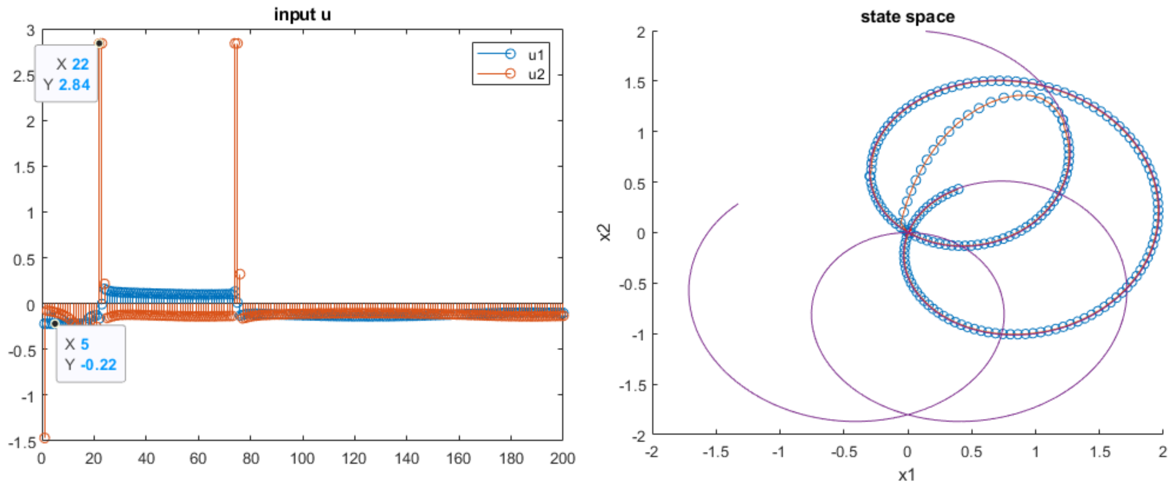


Figure 12 - Reference tracking unsing MPC. On the left we can see that Turtlebot's constraints are fullfiled. On the right, we can use the reference in full line, and the robot's trajectory on dotted.

6. Week 44-46: P and PI controller implementation on the real Turtlebot in Python

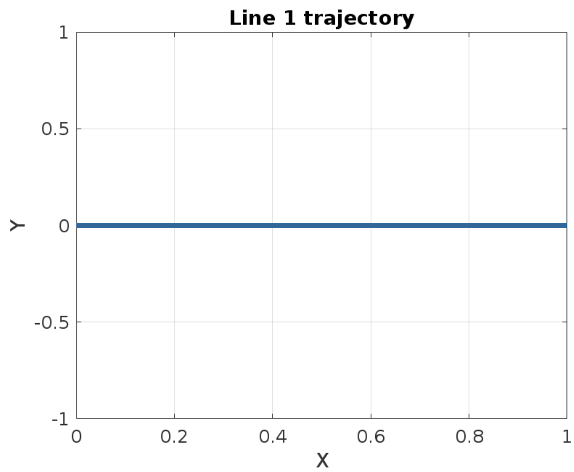
During the last two years, with Jean-Pierre CEYSSON's last intervention and the pedagogical break, we did not have time to schedule many sessions with the whole team. Nonetheless, this week we succeeded in validating several points which constitute a certain progress in the project.

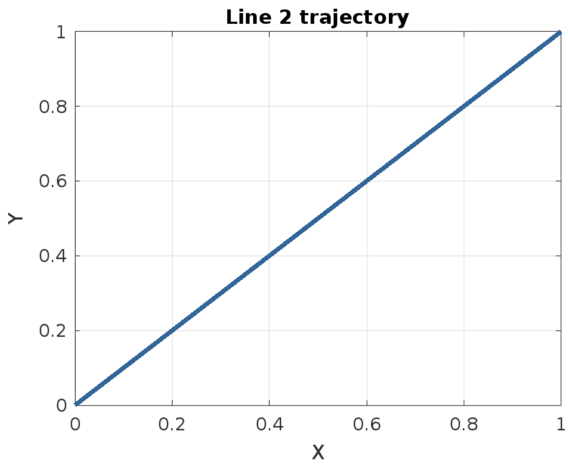
The last few sessions, we had a problem when we executed a trajectory with the *P controller*, the command we passed was not considered by the Turtlebot. What's more, we were obliged to rotate the Turtlebot on itself before it could execute the trajectory, so we had a problem with the *unwrap* implementation in our code.

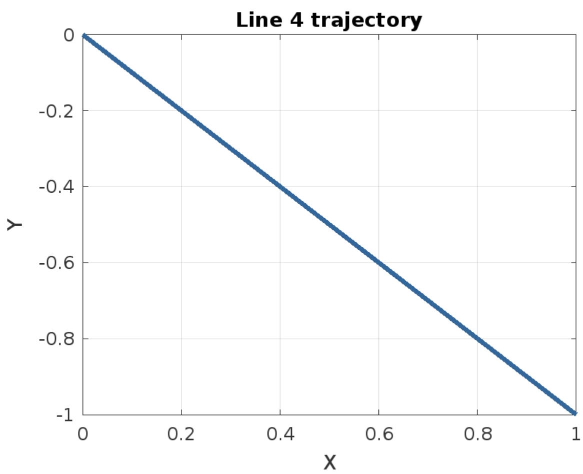
The aim of the next few sessions will be to improve our *P and PI controllers*, but above all to focus on obstacle avoidance and retrieving Turtlebot location data in the *Easynov* room. That's why we'd like to schedule an Easy session for next week.

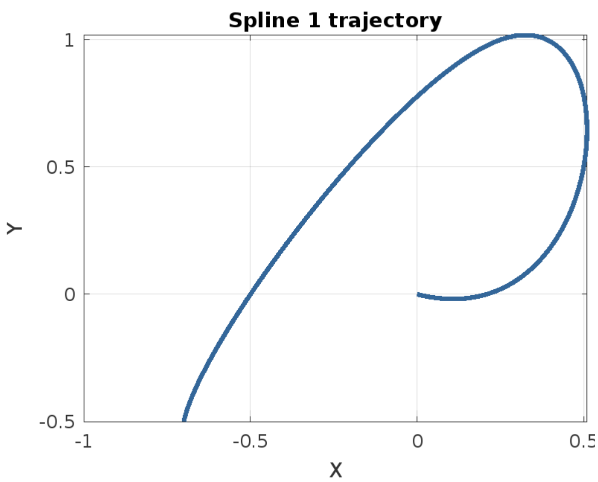
6.1 Generating trajectories

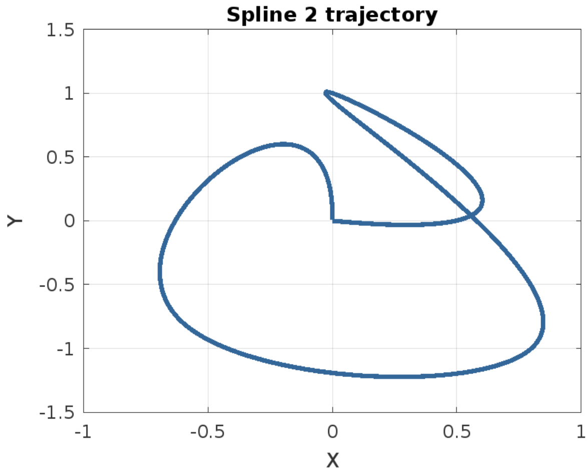
As a first step, we tried to generate different trajectories to test our P controller with different possible paths. We generated both linear and random trajectories to visualize the *Turtlebot's* behavior. Here are some examples of trajectories:











Those trajectories are good to validate our controller, because they include basic linear paths, almost 360° rotations, and tight angles. To improve random generation trajectories, we did a Matlab script available in the appendix **Appendix 4**.

6.2 Improvements of the P controller

As we said earlier, we had several problems with the Python algorithm of the *P controller*. when we executed a trajectory with the *P controller*, the command we passed was not taken into account by the *Turtlebot*. We fixed this error by being more vigilant about the robot's saturation limits. The commands we gave it were above its limits, so it no longer reacted to our algorithm's commands.

In addition, we carried out several tests to obtain simultaneous linear and angular speed limits. In fact, the *Turtlebot* cannot have its maximum linear and angular speed limits simultaneously. The manufacturer announces a maximum linear velocity of $\pm$0.22m/s and angular velocity of $\pm$2.82 rad/s. We therefore found a coefficient of *0.93* to be applied to saturation for maximum efficiency *(so* $v_{\max} = \pm 0.2046$ m/s and $\omega_{\max} = \pm 2.6226$ rad/s). We may revisit this coefficient in the future, in order to improve this controller P as much as possible.

Concerning the fact that the robot must turn on itself before it can react to the algorithm. This bug has been corrected by a more rigorous implementation of the *unwrap* function on the angle. This is the last version of the *Python controller P algorithm* implemented for *ROS 2*:

```python
#!/usr/bin/env python3

import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
import numpy as np
import math
import time
import threading
import pandas as pd
from geometry_msgs.msg import Quaternion
import matplotlib.pyplot as plt
import sys

class MoveController(Node):
    def __init__(self):
        super().__init__('move_controller')

        self.publisher = self.create_publisher(Twist, 'cmd_vel', 10)
        self.subscription = self.create_subscription(Odometry, 'odom',
self.odometry_callback, 10)
```

```
        self.cmd_msg = Twist()
        self.start_time = time.time()
        self.end_time = None

        # Controller constants
        self.kp_distance = 0.9  # Proportional gain for distance control
        self.kp_angle = 0.9  # Proportional gain for angle control
        self.ki_distance = 0 #0.02
        self.ki_angle = 0

        self.kp_angle_init = 1.5

        #Initialize Integral term
        self.integral_distance = 0
        self.integral_angle = 0.05

        # Limits
        # Max linéaire 0.99*0.22
        # Max angulaire 0.93*2.82
        self.MAX_LINEAR_SPEED = 0.22 * 0.93
        self.MIN_LINEAR_SPEED = -0.22 * 0.93
        self.MAX_ROTATION_SPEED = 2.82 * 0.93
        self.MIN_ROTATION_SPEED = -2.82 * 0.93

        # Robot's position updated by odometry_callback
        self.x = None
        self.y = None
        self.z = None
        self.theta = None

        # Memory stamp
        self.x_prev = None
        self.y_prev = None
        self.theta_prev = None

        # Initialize vectors for plot and data saving
        self.xplot = []
        self.yplot = []
        self.error = []
        self.time_ = []
        self.distance = 0
        self.speeds = []

        if len(sys.argv) > 1:
            tracking_type = sys.argv[1]
            if tracking_type == "0" and len(sys.argv) == 4:
                self.xref = [float(sys.argv[2])]
                self.yref = [float(sys.argv[3])]
                print(f"Type: Setpoint tracking -> x={self.xref}
y={self.yref}")
            elif tracking_type == "1"and len(sys.argv) == 3:
                self.file_path = sys.argv[2]
                print(f"Type: Reference tracking -> File:
{self.file_path}")
                self.xref, self.yref = self.importFromCSV(self.file_path)
        else:
            print("No tracking type specified. Tracking (0;0) by
default")
            self.xref = [0.0]
            self.yref = [0.0]
```

```
        # Setpoint
        # self.xref = [0] * 200
        # self.yref = [0] * 200
        self.theta_ref = 0

        # Initialize path from csv file and counter for updating path
        # self.xref, self.yref = self.importFromCSV('spline.csv')
        self.iref = 0
        self.imax = len(self.xref) - 1

        # Parameter for displaying numbers
        self.significant_numbers = 6

        # Start controller thread
        self.control_thread = threading.Thread(target=self.controller)
        self.control_thread.daemon = True   #  Allows the thread to end
when the node is finished
        self.control_thread.start()

        # Sampling time
        self.Ts = 0.05

    def controller(self):
        while (self.x is None) or (self.y is None) or (self.theta is
None):
            self.get_logger().info("Waiting for ODOMETRY to publish its
initial position")

        # Initiallizing start position
        x = self.x
        self.x_prev = x
        y = self.y
        self.y_prev=y
        theta = self.theta

        module_target = math.sqrt((x - self.xref[1]) ** 2 + (y -
self.yref[1]) ** 2)
        argument_target = math.atan2(self.yref[1] - y, self.xref[1] - x)

        time_prev = time.time()

        # Preparing for start angle reaching
        unwrapped_angles = np.unwrap([argument_target, self.theta])
        self.theta_ref = unwrapped_angles[0]
        self.theta = unwrapped_angles[1]

        while abs(argument_target-self.theta)>0.01:
            command_angular_speed = self.kp_angle_init * (self.theta_ref-
self.theta)
            command_angular_speed = min(max(command_angular_speed,
self.MIN_ROTATION_SPEED), self.MAX_ROTATION_SPEED)
            self.cmd_msg.angular.z = command_angular_speed
            self.publisher.publish(self.cmd_msg)

            unwrapped_angles = np.unwrap([argument_target, self.theta])
            argument_target = unwrapped_angles[0]
            self.theta = unwrapped_angles[1]
            print(f"Reference angle: {self.theta_ref}")
            print(f"Current angle: {self.theta}")
            print(f"Error: {self.theta_ref-self.theta}\n")
            print(f"Command: {command_angular_speed}\n")
```

```
        self.get_logger().info(f"Start angle reached !")

        while self.iref < self.imax or module_target > 0.01:
            # Updating time
            current_time = time.time()

            # Computing error
            eps_x = self.xref[self.iref] - self.x
            eps_y = self.yref[self.iref] - self.y

            # Computing module and argument to get to the target
            module_target = math.sqrt(eps_x ** 2 + eps_y ** 2)
            argument_target = math.atan2(eps_y, eps_x)

            # Unwrapping argument to avoid discontinuities

            argument_target = argument_target%(2*math.pi)

            unwrapped_angles = np.unwrap([argument_target, self.theta])
            argument_target = unwrapped_angles[0]
            self.theta = unwrapped_angles[1]

            # Computing Integral term of PI controller
            self.integral_distance = self.integral_distance +
self.ki_distance * (current_time - time_prev) * module_target
            self.integral_angle = self.integral_angle + self.ki_angle *
(current_time - time_prev) * (argument_target - self.theta)

            # Command signal
            command_linear_speed = self.kp_distance * module_target +
self.integral_distance
            command_angular_speed = self.kp_angle * (argument_target -
self.theta) + self.integral_angle

            #Display distance to setpoint and angle error
            self.get_logger().info(f"Distance : {round(module_target,
self.significant_numbers)}")
            self.get_logger().info(f"Angle error :
{round(argument_target-self.theta, self.significant_numbers)}")
            self.get_logger().info(f"Delta t : {round(current_time -
time_prev, self.significant_numbers)}")
            self.get_logger().info(f"step : {self.iref}/{self.imax}\n",)

            # Saturation
            command_linear_speed = min(max(command_linear_speed,
self.MIN_LINEAR_SPEED), self.MAX_LINEAR_SPEED)
            command_angular_speed = min(max(command_angular_speed,
self.MIN_ROTATION_SPEED), self.MAX_ROTATION_SPEED)

            # Command publication
            self.cmd_msg.angular.z = command_angular_speed
            self.cmd_msg.linear.x = command_linear_speed
            self.publisher.publish(self.cmd_msg)

            # Updating counter
            self.iref = min(self.iref + 1, self.imax) # When the terminal
point has arrived, we keep it in time

            # Updating trajectory to plot and datas
            self.xplot.append(self.x)
            self.yplot.append(self.y)
```

```
            self.error.append(module_target)
            self.time_.append(current_time)
            segment = np.sqrt((self.x-self.x_prev) ** 2 +(self.y-
self.y_prev) ** 2 )
            self.distance = self.distance + segment
            self.speeds.append(segment/(current_time-time_prev))

            # Memory stamp
            argument_prev = argument_target
            time_prev = current_time
            self.x_prev = self.x
            self.y_prev = self.y

            time_after_iteration = time.time()
            time.sleep(self.Ts-(time_after_iteration - current_time))

        self.end_time = time.time()
        self.get_logger().info(f"Setpoint reached !")

        # Stop robot
        self.cmd_msg.angular.z = 0.0
        self.cmd_msg.linear.x = 0.0
        self.publisher.publish(self.cmd_msg)

        # # Preparing for final angle reaching
        unwrapped_angles = np.unwrap([self.theta_ref, self.theta])
        self.theta_ref = unwrapped_angles[0]
        self.theta = unwrapped_angles[1]

        while abs(self.theta_ref-self.theta)>0.01:
            command_angular_speed = 2*(self.theta_ref-self.theta)
            self.cmd_msg.angular.z = min(max(command_angular_speed,
self.MIN_ROTATION_SPEED), self.MAX_ROTATION_SPEED)
            self.publisher.publish(self.cmd_msg)

            unwrapped_angles = np.unwrap([self.theta_ref, self.theta])
            self.theta_ref = unwrapped_angles[0]
            self.theta = unwrapped_angles[1]
            print(f"Reference angle: {self.theta_ref}")
            print(f"Current angle: {self.theta}")
            print(f"Error: {self.theta_ref-self.theta}\n")

        self.get_logger().info(f"Reference angle reached !")

        self.get_logger().info(f"\nFinal datas :")
        self.get_logger().info(f"Distance : {round(module_target,
self.significant_numbers)}")
        self.get_logger().info(f"Angle error : {round(argument_target-
self.theta, self.significant_numbers)}")
        self.get_logger().info(f"Average tracking error :
{round(np.mean(self.error), self.significant_numbers)} [m]")
        self.get_logger().info(f"Task achieved in : {round(self.end_time
- self.start_time, self.significant_numbers)} [s]")
        self.get_logger().info(f"Distance traveled :
{round(self.distance, self.significant_numbers)} [m]")
        self.get_logger().info(f"Aveage speed :
{round(np.mean(self.speeds), self.significant_numbers)} [m/s]")

        # Stop robot
        self.cmd_msg.angular.z = 0.0
        self.cmd_msg.linear.x = 0.0
```

```
        self.publisher.publish(self.cmd_msg)

    def odometry_callback(self, msg):

        position = msg.pose.pose.position
        orientation = msg.pose.pose.orientation
        self.x = position.x
        self.y = position.y
        q = orientation
        angles = self.quaternion2euler(q)
        theta = angles[-1]

        # Wrapping theta to 2pi
        self.theta = theta%(2*math.pi)

        # Writing data in file
        # self.file.write(f"Position: ({self.x},{self.y})\nQuaternion:
({q.w,q.x,q.y,q.z})\nTheta: {self.theta}\n\n")


    def quaternion2euler(self, q):
        sinr_cosp = 2 * (q.w * q.x + q.y * q.z)
        cosr_cosp = 1 - 2 * (q.x * q.x + q.y * q.y)
        roll = math.atan2(sinr_cosp, cosr_cosp)

        # pitch (y-axis rotation)
        sinp = math.sqrt(1 + 2 * (q.w * q.y - q.x * q.z))
        cosp = math.sqrt(1 - 2 * (q.w * q.y - q.x * q.z))
        pitch = 2 * math.atan2(sinp, cosp) - math.pi / 2

        # yaw (z-axis rotation)
        siny_cosp = 2 * (q.w * q.z + q.x * q.y)
        cosy_cosp = 1 - 2 * (q.y * q.y + q.z * q.z)
        yaw = math.atan2(siny_cosp, cosy_cosp)

        angles = np.array([roll, pitch, yaw]) #yaw is the angle that we
want
        return angles

    def importFromCSV(self, filename):
        # Read CSV file with data
        df = pd.read_csv(filename, delimiter=';')

        # Extract x and y vectors
        x = df.iloc[:, 0].values  # La première colonne
        y = df.iloc[:, 1].values  # La deuxième colonne

        # Return x and y vectors
        return x, y



def main(args=None):
    rclpy.init(args=args)
    node = MoveController()

    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        # Plot result
        plt.plot(node.xplot, node.yplot, 'xr')
```

```python
        plt.plot(node.xref, node.yref, '-b')
        plt.show()

        plt.plot(node.time_, node.xplot, 'xr')
        plt.plot(node.time_[:len(node.xref)], node.xref, '-b')
        plt.plot(node.time_, node.yplot, 'xr')
        plt.plot(node.time_[:len(node.yref)], node.yref, '-b')
        plt.show()

        node.publisher = node.create_publisher(Twist, 'cmd_vel', 10)
        node.cmd_msg = Twist()

        node.cmd_msg.angular.z = 0.0
        node.cmd_msg.linear.x = 0.0
        node.publisher.publish(node.cmd_msg)
        pass

    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

We were able to test the controller P algorithm on the previously generated trajectories. For the following graphs, blue trajectory represents the reference, and red crossed trajectory represents the real Turtlebot navigation :

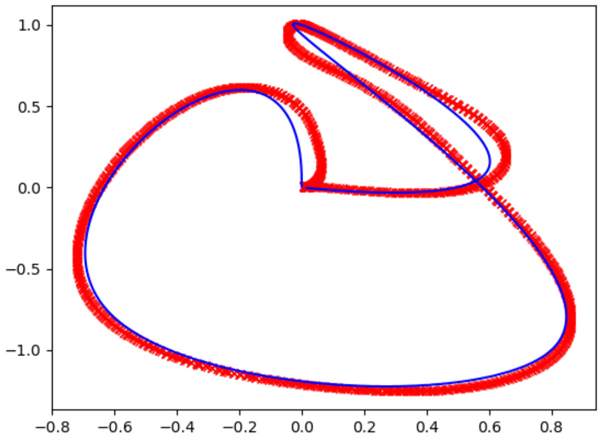


Figure 13 – Real Turtlebot following spline 2 trajectory with P Controller

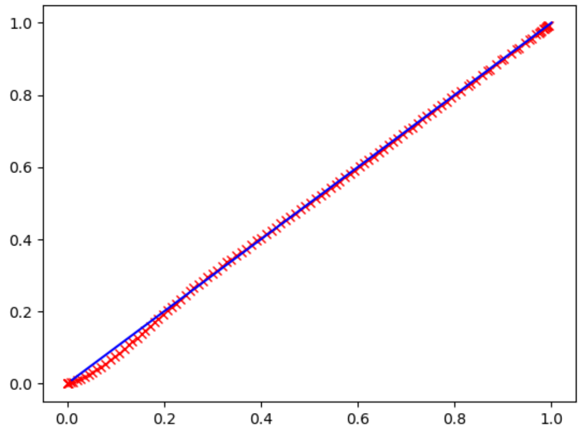


Figure 14 - Real Turtlebot following line 2 trajectory with P Controller

Based on our results and the improvements we have made to the P controller algorithm, we're pretty satisfied with the Turtlebot's trajectory tracking with just one P controller.

6.3 Tests of the PI controller

After thoroughly testing our P controller, we went back to the same algorithm and tried to improve our P controller by adding integrator to make a PI controller. The Python algorithm involves adding integrators for speed and angle. To see the effect of the PI controller, we ran the same trajectory tests as with the P controller:

- **Integrator on the angle ($K_{i_{\text{angle}}} = 0.02$):**

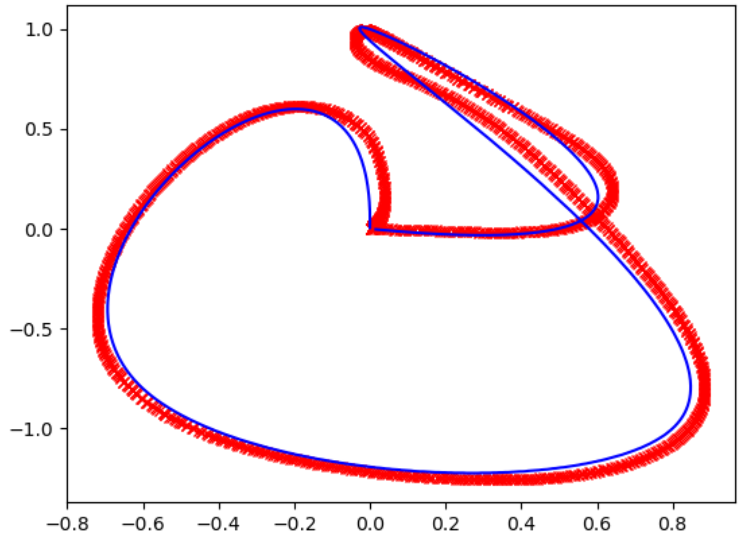


Figure 15 - Real Turtlebot following spline 2 trajectory with PI Controller with $K_{i_{angle}} = 0.02$

- **Integrator on the angle ($K_{i_{\text{angle}}} = 0.02 \; and \; K_{i_{\text{distance}}} = 0.05$):**

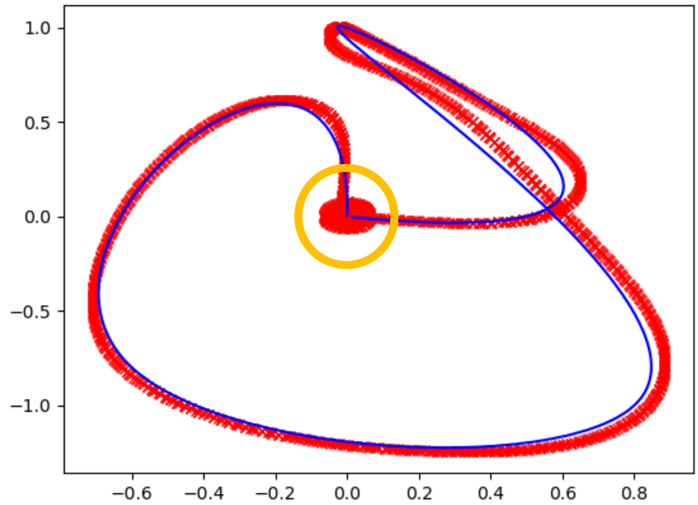


Figure 16 - Real Turtlebot following spline 2 trajectory with PI Controller with $K_{i_{angle}} = 0.02$ and $K_{i_{distance}} = 0.05$

According to our results, the PI controller is currently no better than the P controller in terms of trajectory tracking. What is more, at the end of each trajectory, the Turtlebot tends to turn on itself, and we don't yet know why. On the other hand, the statistics show that the PI controller performs slightly better than the P controller in terms of "average tracking error" after each Turtlebot trajectory:

```
Final datas :
[INFO] [1700054613.779605417] [move_controller]: Distance : 0.009927
[INFO] [1700054613.780145220] [move_controller]: Angle error : 3.855048
[INFO] [1700054613.780751201] [move_controller]: Average tracking error : 0.1627
16 [m]
[INFO] [1700054613.781210492] [move_controller]: Task achieved in : 58.423894 [s
]
[INFO] [1700054613.781680070] [move_controller]: Distance traveled : 8.188985 [m
]
[INFO] [1700054613.782316103] [move_controller]: Aveage speed : 0.14029 [m/s]
```

Figure 17 - Statistics after controller trajectory P

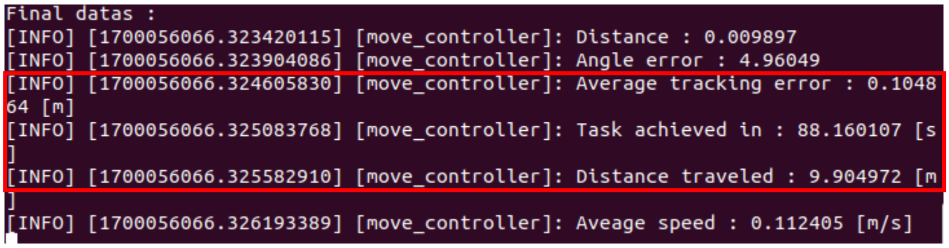


Figure 18 - Statistics after controller trajectory PI

By analyzing some of the output stats the average error is reduced with a PI, but maybe it is only reduced by the fact that the robot has rotated near the point for about 30s… A way to make it work may be to relax the final zone around the final point, but it is not satisfying in terms of precision. For now, the final zone is a circle with a radius of 1cm. As the robot's width is about 18cm, it may be feasible to relax the zone; but it is uncomfortable, knowing that a P controller is able to always reach the target with that 1 cm precision.

It will be a further challenge in the coming weeks to improve this PI controller if possible, but above all to understand why the Turtlebot starts spinning on itself as soon as you put the integrator in the controller.

6.4 User interface for task assignment

For now, the switching between setpoint tracking and reference tracking, and ever reference selection; was manual. It means that util now, we had to modify our code each time we wanted the Turtlebot to achieve a different task.

With that small *UI* ("*User Interface*"), the user can select between reference tracking or setpoint tracking.

When choosing reference tracking, a window appears and proposes the user to select his path as *CSV* file (that he would have built before). There is also an option to tell if the Turtlebot has to start following the path from its current position, of if it has to reach the initial position and angle first.

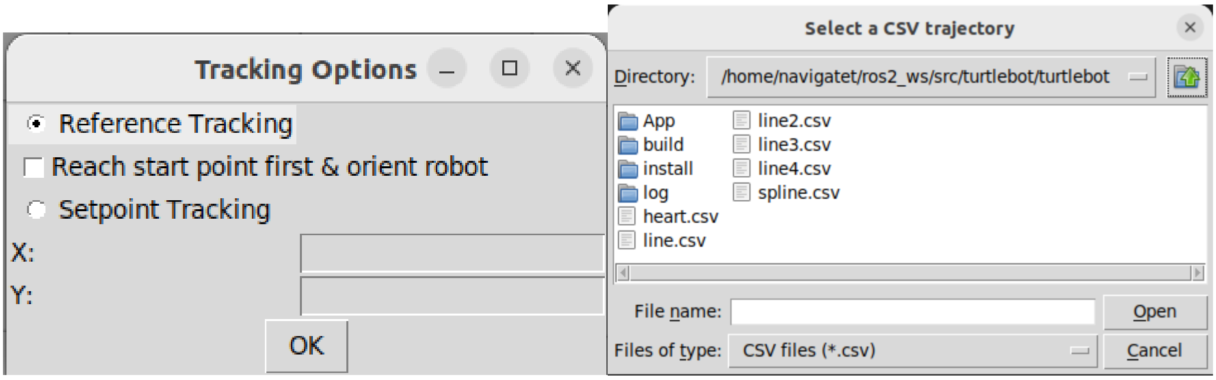


Figure 19 - UI when the user wants to perform reference tracking.

When choosing the setpoint tracking, some fields are enabled to let the user enter his desired setpoint.

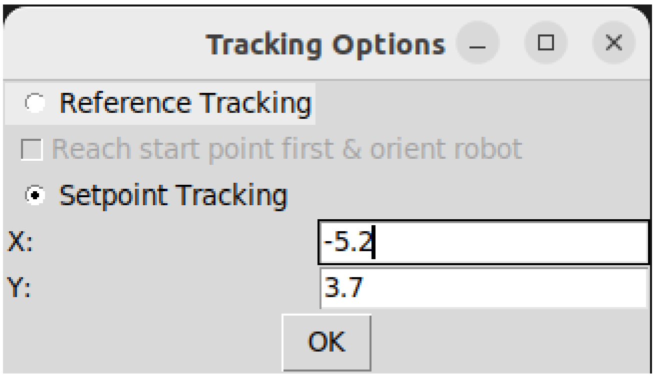


Figure 20 - UI when the user wants to perform setpoint tracking.

It seems nothing, but this *UI* will save our time switching between tests. In fact, each time we wanted to do a battery of tests, we lose time on changing the code to adapt the tracking we wanted.

For now, it works well, except the "*Reach start point first & orient robot*" option, which require some further tunings.

6.5 Creation of a GitHub repository

This week, we have also decided to set up a GitHub repository for our project, so that we can manage the versions of our Python code. Above all, this repository will enable us to leave a maximum of documentary and technical resources to future people working on the Turtlebot project. Here's the link to the GitHub repository that we'll be updating until the end of the project:

https://github.com/Loan5A/NavigateT-Turtlebot.git

ANNEXES

Appendix 1: First node in Python to move

```
#!/usr/bin/env python3

import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist
import time
class  move(Node):
    def  __init__(self):
        super().__init__('test_node')
        self.publisher=self.create_publisher(Twist,'cmd_vel',10)
        self.timer = self.create_timer(0.1,self.timer_callback)
        self.cmd_msg=Twist()
        self.start_time=time.time()

    def timer_callback(self):
        current_time=time.time()
        if current_time - self.start_time < 10.0:
           self.cmd_msg.linear.x=0.2
           self.publisher.publish(self.cmd_msg)
        else:
            self.cmd_msg.linear.x=0.0
            self.publisher.publish(self.cmd_msg)

def main(args=None):
        rclpy.init(args=args)
        node=move()
        try :
            rclpy.spin(node)
        except KeyboardInterrupt:
            pass
        node.destroy_node()
        rclpy.shutdown()

if __name__=='__main__' :
    main()
```

Appendix 2: Matlab code

```
clear all; clc; close all;
%% Definition of parameters
% we construct the structure that can be given as an argument to a block
To Workspace
T=200;                              % simulation time
timp=linspace(0,T,1e4);
Ts = timp(2)-timp(1);

Kp=0.9; %Controller gain
r = 33e-3; %Wheel radius
d = 160e-3; %Distance between wheels
A = [r/2 r/2; r/d -r/d]; %[V;w]=A*[wr;wl]
%% Reference for path tracking
%reference end point for constant and linear references
xref=-7;
yref=26;

%constant reference
%position_ref = [xref*ones(1,length(timp)); yref*ones(1,length(timp))];

%Linear reference
%position_ref = [xref*timp/T+5; yref*timp/T-2];

%Quadratic reference
%position_ref = [xref*timp.^2/T^2+5-1/4*timp/T;
yref*timp.^2/T^2+1/4*timp/T-2];

%Circular references
R=3; f1=0.02; f2=0.05;
position_ref=[R*sin(f1*timp)+R*sin(f2*timp);
R*cos(f1*timp)+R*cos(f2*timp) ];
%position_ref=[(sin(f1*timp));(cos(f1*timp))];
%% turtlebot simulation
load_system('turtlebot');  % we load the simulink model into memory
set_param('turtlebot', 'StopTime', num2str(T)) % set the simulation time

rsim=timeseries(position_ref',timp);    % we build the structure that is
received by the From Workspace block
out=sim('turtlebot');  % we run the simulink model, at the end of the
simulation we have stored the output in the ysim structure
%% plot the results
figure; grid on; hold on
plot(out.ysim.time,out.ysim.signals.values, 'm')
plot(rsim.Time,rsim.Data,'b')
legend('output - x ','output - y ','reference - xr','reference - yr')
xlabel('time')
ylabel('output signals')

figure; grid on; hold on
plot(out.ysim.signals.values(:,1) ,out.ysim.signals.values(:,2), 'm')
plot(rsim.Data(:,1),rsim.Data(:,2),'b')
legend('output - (x;y) ','reference - (xr;yr)')
xlabel('time')
ylabel('output signals')
```

Appendix 3: Obstacle avoidance algorithm prototype in Python

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

class LaserScanSubscriber(Node):
    def __init__(self):
        super().__init__('reading_laser')
        self.publisher = self.create_publisher(Twist, '/cmd_vel', 1)
        self.subscription = self.create_subscription(LaserScan,
'/robot/laser/scan', self.laser_callback, 10)
        self.threshold_dist = 1.5
        self.linear_speed = 0.6
        self.angular_speed = 1

    def laser_callback(self, msg):
        regions = {
            'right':  min(min(msg.ranges[0:2]), 10),
            'front':  min(min(msg.ranges[3:5]), 10),
            'left':   min(min(msg.ranges[6:9]), 10),
        }

        self.take_action(regions)

    def take_action(self, regions):
        msg = Twist()
        linear_x = 0
        angular_z = 0

        state_description = ''

        if regions['front'] > self.threshold_dist and regions['left'] >
self.threshold_dist and regions['right'] > self.threshold_dist:
            state_description = 'case 1 - no obstacle'
            linear_x = self.linear_speed
            angular_z = 0
        elif regions['front'] < self.threshold_dist and regions['left'] <
self.threshold_dist and regions['right'] < self.threshold_dist:
            state_description = 'case 7 - front and left and right'
            linear_x = -self.linear_speed
            angular_z = self.angular_speed  # Increase this angular speed
for avoiding obstacle faster
        elif regions['front'] < self.threshold_dist and regions['left'] >
self.threshold_dist and regions['right'] > self.threshold_dist:
            state_description = 'case 2 - front'
            linear_x = 0
            angular_z = self.angular_speed
        elif regions['front'] > self.threshold_dist and regions['left'] >
self.threshold_dist and regions['right'] < self.threshold_dist:
            state_description = 'case 3 - right'
            linear_x = 0
            angular_z = -self.angular_speed
        elif regions['front'] > self.threshold_dist and regions['left'] <
self.threshold_dist and regions['right'] > self.threshold_dist:
            state_description = 'case 4 - left'
            linear_x = 0
            angular_z = self.angular_speed
        elif regions['front'] < self.threshold_dist and regions['left'] >
```

```
self.threshold_dist and regions['right'] < self.threshold_dist:
            state_description = 'case 5 - front and right'
            linear_x = 0
            angular_z = -self.angular_speed
        elif regions['front'] < self.threshold_dist and regions['left'] <
self.threshold_dist and regions['right'] > self.threshold_dist:
            state_description = 'case 6 - front and left'
            linear_x = 0
            angular_z = self.angular_speed
        elif regions['front'] > self.threshold_dist and regions['left'] <
self.threshold_dist and regions['right'] < self.threshold_dist:
            state_description = 'case 8 - left and right'
            linear_x = self.linear_speed
            angular_z = 0
        else:
            state_description = 'unknown case'
            self.get_logger().info(str(regions))

        self.get_logger().info(state_description)
        msg.linear.x = linear_x
        msg.angular.z = angular_z
        self.publisher.publish(msg)


def main(args=None):
    rclpy.init(args=args)
    laser_scan_subscriber = LaserScanSubscriber()
    rclpy.spin(laser_scan_subscriber)
    laser_scan_subscriber.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Appendix 4: Matlab script to generate splines through specified points

```
clear all; close all; clc;
% Name of the file storing data
file = 'spline.csv'

% Delete "spline.csv" before writing it
% Name of the file storing data
file = 'spline.csv'

% Vérifier si le fichier existe
if exist(file, 'file')
    % Si le fichier existe, le supprimer
    delete(file);
    disp(['Fichier ' file ' supprimé.']);
else
    disp(['Le fichier ' file ' n''existe pas.']);
end

% Define points that the spline will pass through
x = 1/5*[0 3 0 4 -3 -1 0];
y = 1/5*[0 1 5 -5 -4 3 0];

% Sample time
Ts = 0.1;

% Time vector
t = [0 10 20 30 40 50 60]; %Specify time spent reaching each checkpoint
time = 0:Ts:max(t);

% Computing spline
xspline = spline(t,x,time);
yspline = spline(t,y,time);

figure;
plot(xspline, yspline);
hold on;
plot(x,y, 'xr');
title('Trajectory passing through specified points');
xlabel('x_1');
ylabel('x_2');
xlim([min(xspline) max(xspline)]);
ylim([min(yspline) max(yspline)]);

figure;
plot(time, xspline);
hold on;
plot(t,x, 'xr');
title('X component of the trajectory');
xlabel('time (s)');
ylabel('x');
ylim([min(xspline) max(xspline)]);

figure;
plot(time, yspline);
hold on;
plot(t,y, 'xr');
title('Y component of the trajectory');
xlabel('time (s)');
ylabel('y');
ylim([min(yspline) max(yspline)]);
```

```
% Verifying speeds
vx(1) = 0
for i = 1:length(xspline)-1
  vx(i+1) = (xspline(i+1) - xspline(i))/Ts;
end

vy(1) = 0
for i = 1:length(yspline)-1
  vy(i+1) = (yspline(i+1) - yspline(i))/Ts;
end

figure;
plot(time, vx)
title('x-axis speed');
xlabel('time (s)');
ylabel('vx');

figure;
plot(time, vy)
title('y-axis speed');
xlabel('time (s)');
ylabel('vy');

% Exporting to CSV
spline = [xspline', yspline'];
dlmwrite(file,spline,";", "precision", 4);
```