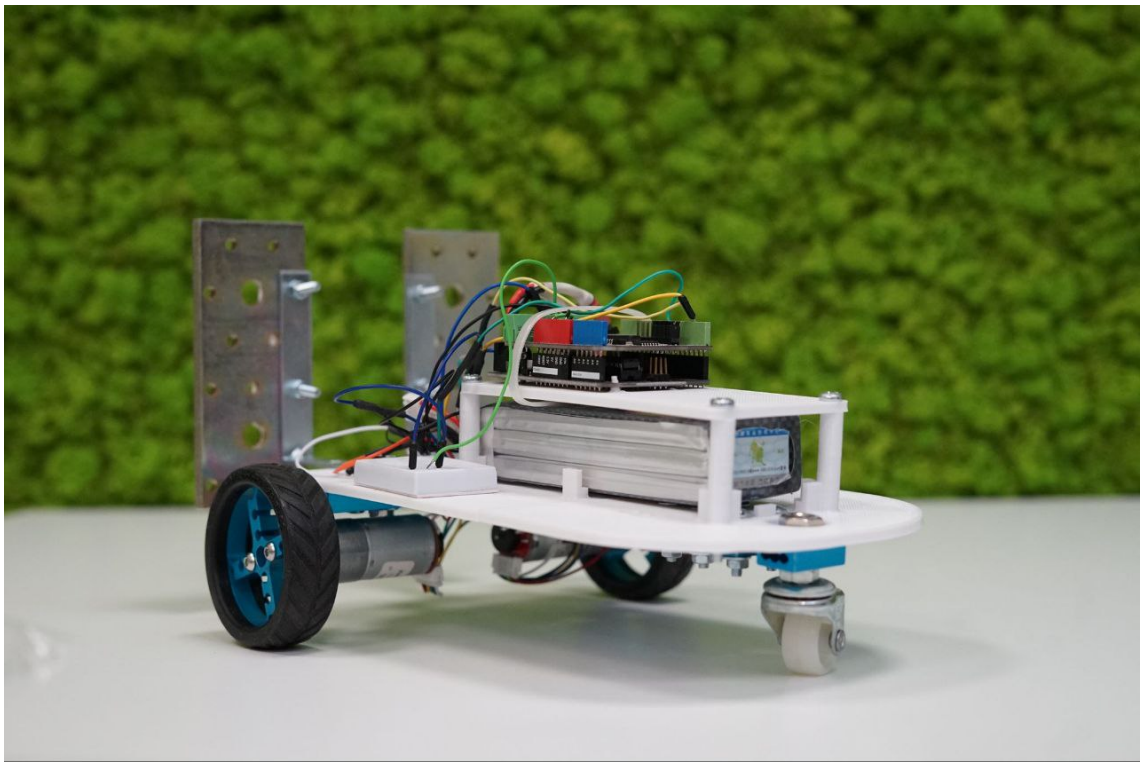
Politecnico di Torino

DRAWBOT



DrawBot - Project Report

01PEEQW - Robotics

Matthias Bonnart - 294197
Leonardo Ferrari - 295045
Carlo Maria Foglia - 301379
Riccardo Giacchino - 301168
Gregory Sacco - 303267

2022

Contents

1 Introduction

As part of the Robotics course in mechatronics engineering at the Politecnico di Torino, a project could be realised to familiar ourselves with some practical aspects in the field of robotics, to give a few realised with the Drawbot project: CAD design, design and build of the electronics on board, control algorithms, coding using ROS, autonomous Drive.

For this project, a mobile robot with a differential drive configuration was chosen with an autonomous algorithm using ROS. The detailed project idea will be presented in the next section, "General Description". Following this section, the electronics will be described, then the mechanics of the robot and the programming. And in the last section before the conclusion, the final assessment will present the strengths and weaknesses of the robot, the leads "to go further, if we had more time" and some applications of a Drawbot robot in real world situations.

This project was realised from the start of the semester with firstly a brainstorming to determine the best approach to the project goals. The implementation of the hardware was the next logical step with the order of components and construction of the robot followed by the coding and testing to get. This report presents the results of the work that was put in during the semester.

2 General Description

In this section, the objectives of the project will be explained. Firstly the main idea of the project and initial objective will be detailed and then the achievements will be presented briefly, before going in to detail in the next sections.

2.1 Drawbot project - presentation of the idea

The goal of this project was to design, build and code an autonomous robot, using Robot Operating System (ROS), that would replicate the pattern a user would draw on a computer, on the ground/table it would be placed on. The schematic in figure 2.1 illustrates this idea.

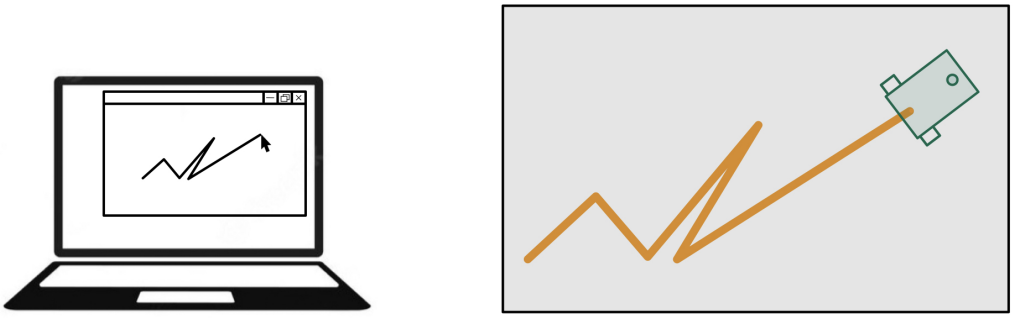


Figure 2.1: Schematic showing the initial objective of the project

2.2 Achievements

The first step that was done after discussing the project objectives was to design and build the mechanical parts and the electronics. For this purpose, a CAD design was created, some 3D components were printed and adjusted to the available electrical and mechanical components to fit them on the bot. Then the code was created step by step, to give a few:

1. creating the ROS structure;
2. testing nodes with arduino;
3. implementing a remote control via cable from the PC;
4. recovering data from the encoders;
5. implementing the low-level control;
6. parallel working on the odometry, path planner and math node with their integration to the ROS structure.

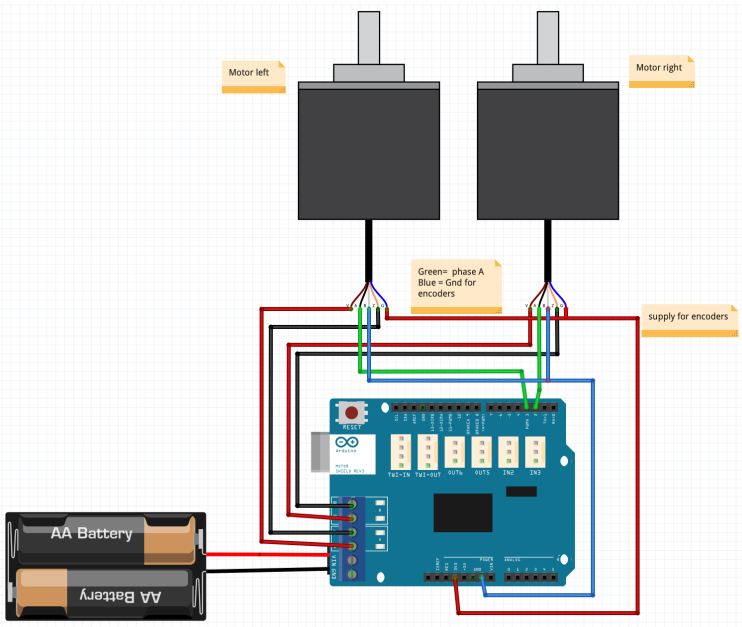
This, of course, came with a lot of experiments and testing to adapt the code. The electrical, mechanical and coding parts will be detailed in the coming sections. To summarise the achievements this group has done: the robot was designed on CAD, the hardware part was designed and created to match our needs, while also using 3D printed parts. This hardware then allowed to test a simple autonomous algorithm that allows the robot to get from an initial position to a given goal position using an odometry localisation code.

3 Electronics of the robot

For the electronic part of our Drawbot, several components have been implemented. Here's the list:

- Micro Controller: Arduino UNO
- Motor driver: L298P shield v1. Vendor: OFROBOT
- 2 Motors: 12V rated voltage with encoder integrated.
- Power supply: 11.1V rated voltage and 5200mAh. LiPo battery.

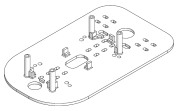
In the following figure is shown a simple schematic figuring how we connected all the components:



The motor driver is mounted on top of the Arduino. They share almost all the pins, but the driver has two more inputs for the higher voltage supply and four more outputs to control the motor velocities. In all the tests, since the WiFi module was still not set, the Arduino has been supplied through the USB cable from the computer. As clear from the scheme, each motor has 5 pins. V and Z pins are used to power up the motors, while A corresponds to the Phase A output of the encoder and B to the ground pin of the encoders. Each encoder is supplied by 3.3 V provided by the Arduino. Actually real motors+encoders have 6 pins each, but some problems occurred trying to use Phase B of the encoders so in the end, phase B hasn't been connected. In general the encoders, through phase A and phase B outputs, can let know the micro controller how fast the motor is rotating and in which direction. For example, from the software point of view, the pins connected to the two phase A of the encoders are configured in INTERRUPT, which means that at each rising edge of the output square wave of phase A, the Arduino will call a function able to elaborate this input, allowing us to count the edges in order to compute the velocity. Since the phase B hasn't been connected, the rotation info was retrieved by means of a simple code in the main software.

4 Mechanics of the robot

The mechanics of the Drawbot need to be as rigid as possible in order to attain a good accuracy. Because of that, the structure is based on an older car robot with metallic components, that serve as the structure of the robot, like the bones in the body.
Several parts were designed using Solidworks and printed from scratch in order to be able to place on the top of it all the wanted components (Arduino, battery, breadboard...).



The first part to be designed is the base, its goal is to connect the two metallic parts of the old robot and, at the same time, to have enough space for all the new components. The first version realized for the base had just the holes necessary to connect all the pieces plus other holes to fix the battery; after carrying out some tests a new upgraded version was finally designed, with buttonhole instead of hole in order to have better tolerances for the other parts, plus other holes to make it easier for cables coming from the motors to pass through. The new version presents also a special section for the battery with four cylinders at the extremity for the upper base.

The upper base is designed for the Arduino and eventually for the WiFi module or other modules, it is then connected with the base with screws and bolts. After the printing of this structure, in order to counterbalance the weight distribution of the Drawbot, several metal sheets were placed in the back park. In this way it is possible to move the center of gravity as close as possible to the centre of the axis of the wheels.

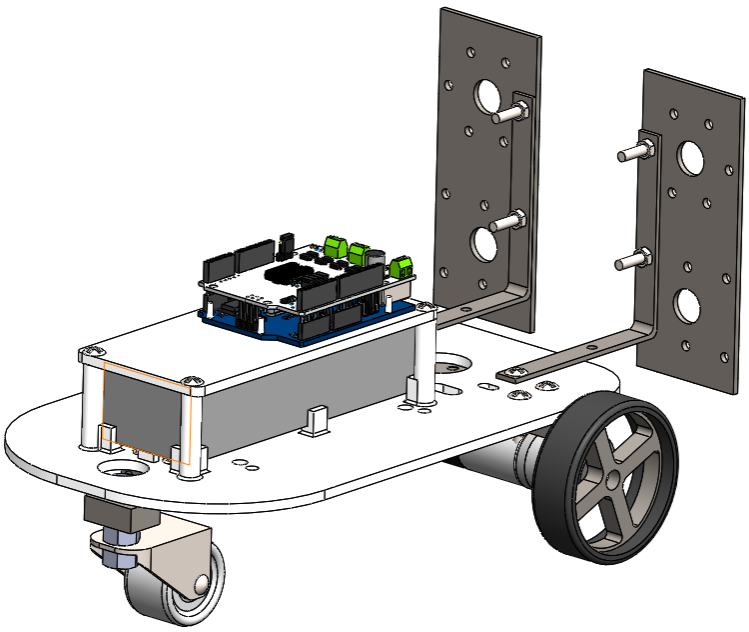


Figure 4.1: Cad 3D Solidworks

5 Programming

5.1 Description of the general structure of the program

The first step of the programming phase is the functional decomposition of what should be implemented. In figure 5.1 is proposed a theoretical scheme representing the main functional blocks, pointing out their I/O variables and interconnections.

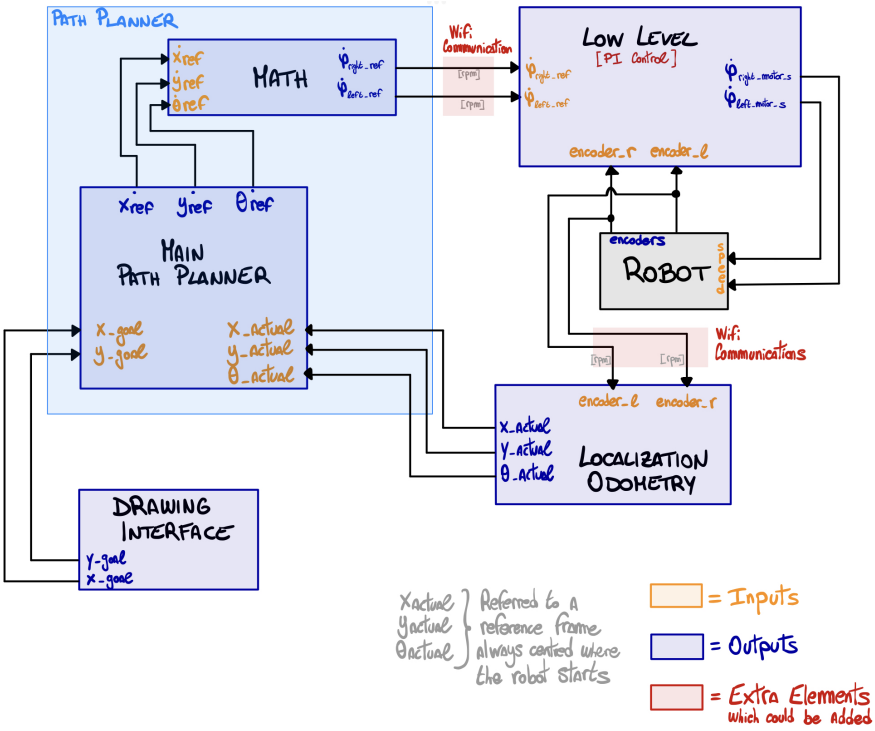
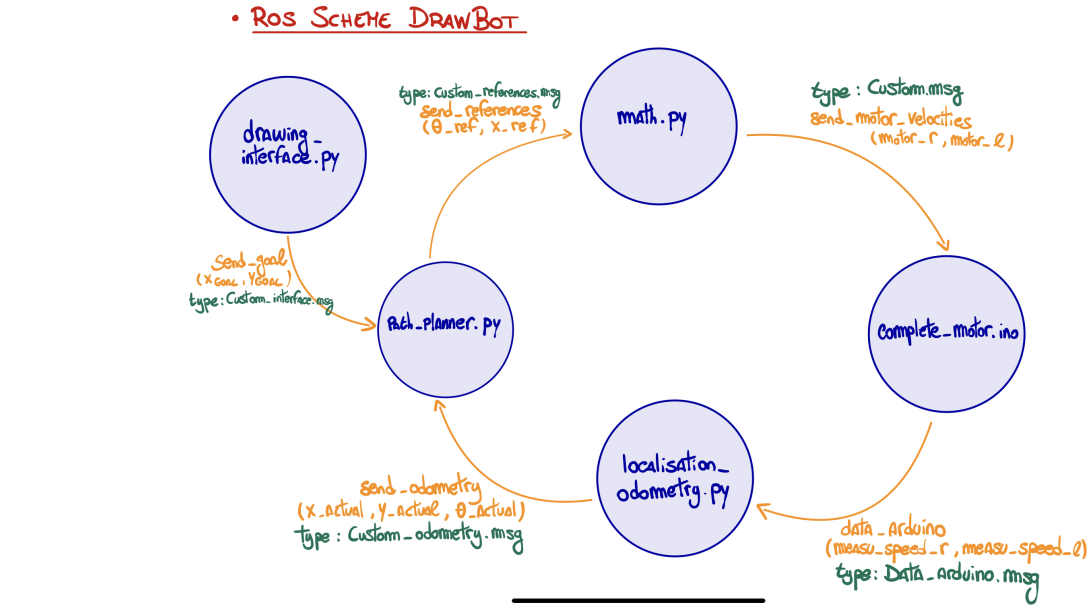
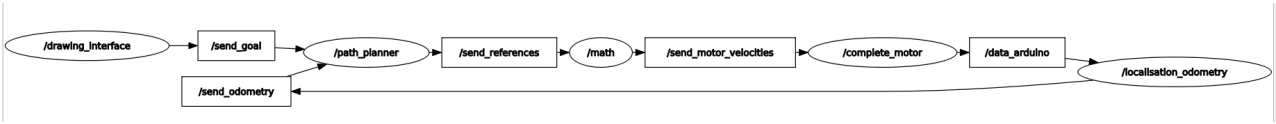


Figure 5.1: Theoretical scheme representing the functionalities to be provided

Starting from this overview, it is then possible to move to the implementation of the code. The code implementation is handled exploiting ROS. The main structure is composed by five nodes, each one with a particular task, communicating each other through the same number of nodes. The idea is to realize a modular structure in which every node handles a different part of the general process needed to govern the robot. In figure 5.2 the graphic concept and the actual ROS nodes visualization are both proposed:



(a) General scheme of ROS structure for the nodes and the topics



(b) actual ROS scheme obtained through the command "rqt_graph" once that all the nodes are active

Figure 5.2: ROS schematics

As can be seen from figure 5.2a, the nodes interconnection has a circular structure, since they recursively have to send and receive information which will constantly be updated. The only extra node, which only sends information is *drawing_interface* node. A brief node explanation will now be provided and properly analyzed in the following chapters:

- **drawing_interface** : opens a graphic interfaces which allows you to trace with the cursor a line (or a series of lines) and sends the (x_{goal},y_{goal}) ending point of that line;
- **path_planner** : takes as input the (x_{goal},y_{goal}) goal coordinates coming from the node drawing_interface and the (x_{actual},y_{actual},θ_{actual}) actual coordinates (actual position and attitude of the robot) and provides as output how much the robot should rotate (θ_{ref}) and how much the robot should go forward (x_{ref}) after the rotation;
- **math** : gets as input the reference coordinates and computes the two motor velocities (motor_r, motor_l) in rpm, providing them as output;
- **complete_motor** : Arduino node, takes the two reference motor velocities as input and send the command to the motors in PWM. Then, it takes the values read by the encoders and convert them from pulses to rpm velocities. Finally, it sends these measured speed (measu_speed_r , measu_speed_l) to the next node;
- **localisation_odometry** : takes as input the two speed (in rpm) measured from the encoders and, through some algebraic manipulation obtained from the odometry analysis, provides as output the actual configuration of the robot expressed by (x_{actual},y_{actual},θ_{actual}).

Therefore, by simultaneously exploiting the aforementioned nodes, it's possible to give to the Drawbot a certain goal and let him reach it, having a continuous update of its actual position (measured through the odometry) and its distance from the goal.

5.2 Description of the different tasks and their management

The structure of the programming was described in the previous section. Here, in this section, the explanation, the functioning and the coding of each node will be described. The low-level control node *complete_motor* receives the reference speeds in rpm and takes care of controlling the rotation of both motors. It also publishes the measured speeds, which will be used by the other nodes to handle the next speed to provide: the *localisation* node uses odometry to estimate the current position of the robot; the *path_planner* calculates the direction and trajectory of the robot and gives these values to the *math* node which will provide the reference speeds to the low-level controller. In this section, some of the code will be referenced as listings, the codes can be found in the appendixes A to E.

5.2.1 Low level controller (Arduino) - *complete_motor*

As previously stated, the low-level controller receives the reference speeds in rpm for both motors and from there, it converts the reference speeds into a PWM (pulse width modulation) wave to be sent to the motors, working in an open loop fashion. The low-level control node also reads, from the encoder, the number of ticks and converts them into an estimated speed. The first part of this section will go over the conversion of encoder readings into a measured speed in rpm, then the code of all the Arduino structure will be presented. For reasons explained later the PI controller was not used in the final code. The work on the PI control is presented in section 5.3.

Measuring the speed

The DC motors that were used on the Drawbot were chosen with encoders. Each motor has two encoders, A and B, which are out-of-phase. The code exploits an interrupt function that detects the rising edges of signal A, when this happens, it increases a counter variable and checks if the signal B is high or low. This gives us two crucial information: the number of ticks since the last reading and the direction the motor is spinning. As shown in figure 5.3, there are two sensors measuring the speed in each motor, they are out-of-phase, making it possible to see which direction the motor is spinning by checking which rising edge occurred first.

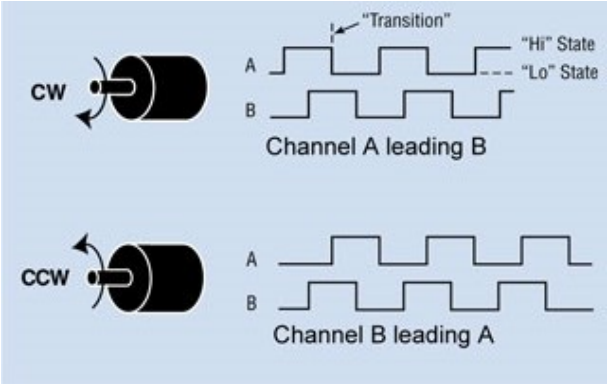


Figure 5.3: Sensory reading during a clockwise and anti-clockwise rotation [2]

Once that the direction of the rotation is known, it's necessary to retrieve the speed. Using the counter variable that sums the rising edges, an estimation for the rotation speed can be found. In particular, by turning the wheel by hand and printing the readings it is possible to obtain the number of ticks per rotation. The figure 5.4 shows the experimental setup (tape was used to better visualize each rotation). This was done ten times for both wheels (right and left) and gave the following results:

$$Ticks_R = \frac{103+102+101+103+103+102+102+102+102+102}{10}$$
$$= 102.2$$
$$Ticks_L = \frac{101+102+101+102+102+103+102+102+103+103}{10}$$
$$= 102.1$$

Approximating this value by 102 and setting an interval of 0.1 seconds between the measurements, the speed in rpm can be given by: $v = 60\frac{X}{102 \cdot 0.1}$, with $X =$*Sum of encoder readings during the 0.1s interval*. An experiment was conducted to see if the interval had an impact on the precision of the speed measurements, since it was necessary to determine if an interval of 0.1 second might give too imprecise measurements. This is shown in table 5.1. The noise on the measurements was estimated experimentally previously at [-6,+6] rpm, so these results are within this margin. Seeing these results it was determined that the time interval of 0.1 second would not be an issue on the accuracy of the measurements.

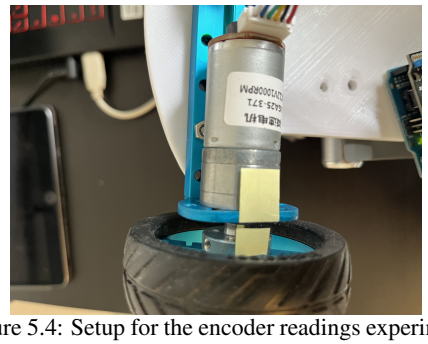


Figure 5.4: Setup for the encoder readings experiment

Time Interval	Velocity measurements [rpm] for pwm=50					
10 seconds	154.08	153.84	153.06	154.02	154.92	155.28
1 seconds	152.94	158.22	158.22	157.08	158.82	157.62
0.1 seconds	158.82	164.7	158.82	158.7	158.7	164.7
10 seconds	Average: 154.20					
1 seconds	Average: 157.15					
0.1 seconds	Average: 160.74					

Table 5.1: Speed measurements for different time intervals

Functioning of the Low level code

Although the Arduino node is in open loop, the whole ROS structure realizes a closed loop control for the position. After the computation done in the *math* node, the velocity of the motors (in rpm) are received by the Arduino node, which job is to convert these values in PWM, send them to the respective pin of the motors and provide back all the needed data to restart the cycle.
The first part of the code contains the definition of all the pins used in the Arduino, such as the motor pins, which value was already defined in the datasheet of the motor shield, with the following configuration:

	PWM pin	DIR pin
Right motor	5	4
Left motor	6	7

For what regards the encoder, the correspondent pins are defined as well as the variables needed to compute the velocity, in particular a variable for each motor to count the number of rising edge of the signals. Since the high rate with which the velocities are computed by the code, the values are published only after a specific time interval of 0.1 sec, by implementing an *if* condition.
- Code Reference: from line 139 to line 156 of A.1
Moreover, there's a function able to elaborate and publish all the needed data to the node *localization_odometry*, by using a customized message file *Data_arduino.h*. In particular, the function memorizes inside the variable that has to be published all the correspondent values (most of them were created for debugging purposes, so now are set to zero). Since the phase B of the encoders isn't used, it is implemented an *if* statement that compare the measured velocity and the reference one. If there is a difference in the sign, the measured speed is inverted, in this way it's realized an "artificial" determination of the wheels direction. The publish rate of the *void loop* is set to 0.05s. This control is crucial in order to send to the node *localization_odometry* the velocity with the correct rotation verse, that will influence the result of all the computations.
- Code Reference: from line 215 to line 253 of A.1

The last part of the code works in order to send the values of the motors to the respective pins. Since all the velocities are in rpm, they have to be converted in PWM using the aforementioned conversion formula based on the empirical tests. There's also a "boundary check" function, realized to ensure that the PWM value isn't greater than the allowed upper limit of 255 or lower than the lower limit of 0.
- Code Reference: from line 160 to line 181 of A.1

5.2.2 Localisation - *localization_odometry*

Another crucial part of mobile robotics is its localisation in an environment. For this project, a local localisation method was used. The odometry algorithm uses the readings from the encoders to estimate its trajectory and by extension its position and orientation. This information is then published on the node *path_planner*, which is in charge to compute the control algorithm.
The initial position of the robot is given in figure 5.5a. It moves then in the plane and the odometry node computes the coordinates $(x,y)_I$ of the robot in the inertial frame. The angle θ is also computed giving its orientation from the angle between X_I and X_R. An example of this is shown in figure 5.5b.

(a) Initial position

(b) Example position

Figure 5.5: Configurations of the robot frame in the inertial frame

The algorithm

The algorithm is based on the material present in [5] and the code is improved with inspiration from [1]. After each new measured speed is published by the Arduino node *complete_motor*, it is read by the subscriber *localisation_odometry*. Then, the algorithm calculates the new current position (x_{actual},y_{actual},θ_{actual}). The algorithm works as follows:

* some useful information first:

- R : radius of the wheel
- b : distance separating the center of both wheels
- the letter *i* is used to denote the right or the left wheel when the formulas are the same for both

1. The linear speed of each wheel is computed from their respective rotational speed (from rpm to m/s), acquired from the topic "data_arduino":
$$V_{lini} = 2 \cdot \pi \cdot R \cdot V_{roti}/60$$
2. The linear and angular speeds of the robot are then retrieved (V_y is 0, the robot is non-holonomous):
$$V_x = \frac{V_{linR} + V_{linL}}{2} \qquad V_y = 0 \qquad V_\theta = \frac{V_{linR} - V_{linL}}{b}$$
- Code References points 1 and 2: from line 38 to line 42 of B.1
3. Since it's necessary to have the cognition of time, in particular of the elapsed time *dt* between the former and the actual computation of the position, it's necessary to include in the Python file the library dedicated to the time. Then, defining two variables "current_time" and "elapsed_time" and updating them every cycle is possible to obtain the elapsed time *dt*, useful to keep track of the current position of x,y and θ.
- Code References point 3: from line 44 to line 46 of B.1
4. Finally, the new position and angle is computed: the math is presented below and uses the graphical representation in figure 5.6. These values will then be published on the topic "send_odometry" for the path planner to use.
$$x = x + (V_x cos(\theta) - V_y sin(\theta)) * dt \qquad y = y + (V_x sin(\theta) + V_y cos(\theta)) * dt \qquad \theta = \theta + V_\theta * dt$$
- Code References point 4: from line 48 to line 58 of B.1

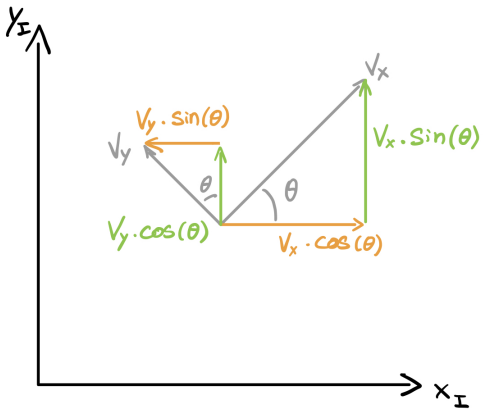


Figure 5.6: Graphical representation using vectors

The results

Testing the odometry without testing the path planner is no easy task. A test was conducted to check the accuracy of these nodes. It will be detailed further in the next section when discussing the path planner. This test is shown below in figure 5.7. The robot starts at the corner and goes to its goal position, a point on its diagonal, designated by the cross in the figure in 5.7a. This test was forced by using a rostopic pub on the node to provide as goal the coordinates (0.5,0.5), i.e. without the graphic interface. It reaches its goal position in figure 5.7b. The figure 5.8 shows the readings taken from the encoders. The system looks quite precise but this result is to be taken with caution. The error of the odometry accumulates over time. This is a simple test with the robot turning once and then going straight. To analyse the quality of the localisation system more tests should be conducted. A way to fix the accumulated error is to recalibrate the robot

during its functioning[1]. Another way to solve this would be using a global localisation system on top of the local system such as beacons and according sensors, a lidar... .

(a) Initial position (b) Final position

Figure 5.7: Test of the odometry and path planner

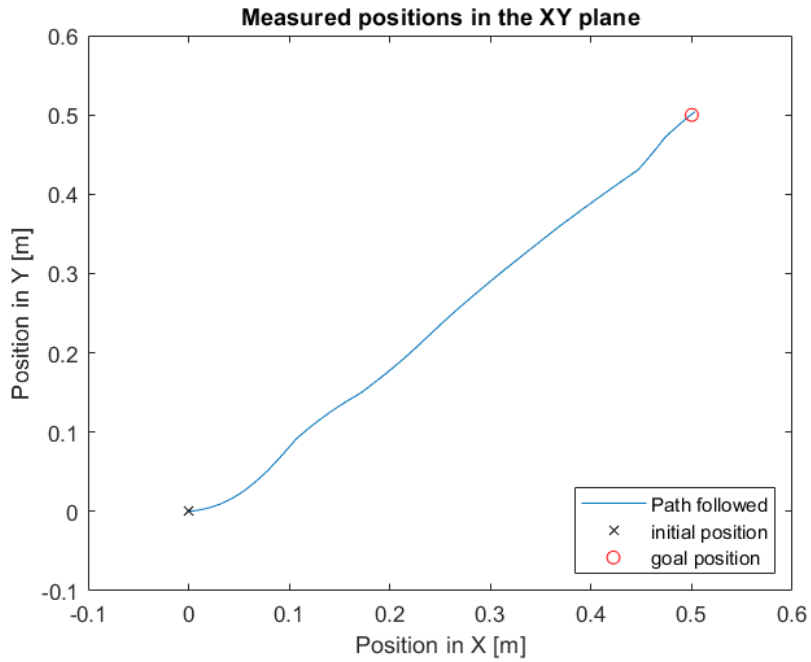
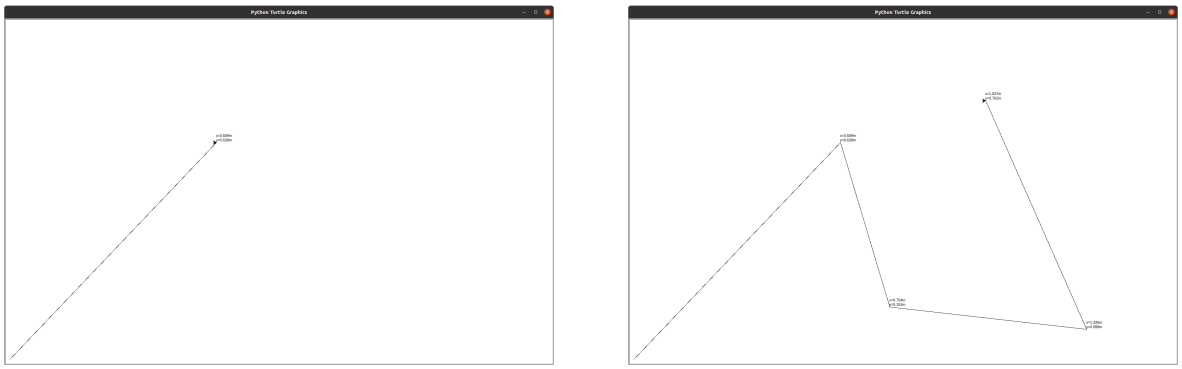


Figure 5.8: XY position given by the odometry

[1]This can be done several ways, some relatively easy and low cost like adding corners to the map and using switches on the side of the robot to detect the walls of corner, hence its position. Some less trivial and more costly methods exist like using ARUCO markers on the floor and a camera to scan them and actualise the robot position

5.2.3 User Interface - *drawing_interface*

This is the interface that allows the user to generate the list of desired position based on the drew lines. The GUI (graphic unit interface) works opening a window in which the user is able to select any point by using the cursor. Then, just by clicking on the desired point, it will automatically draw the line between the former point and the new one that has been selected, as shown in figure 5.9



(a) One point (b) Multiple point

Figure 5.9: Drawing interface

The code is based on the *Turtle* graphic library, through which it is possible to generate a window with customized dimensions, which correspond to the number of pixel, and interact with it. In this scenario the dimension were $1600 * 1000$ pixels, which correspond to $1,6 * 1,0$ meter of the actual workspace of Drawbot. The values can be chosen arbitrarily. By default, the starting point were the center of the window, so it was changed to the bottom-left corner for practical reasons. Once the window is generated, by clicking on it, the code generates the correspondent point through an object characterized by the x-y coordinates of the pixel. These coordinates have to be divided by 1000 to scale them from meters in millimeters, since the workspace is $1600 * 1000$mm.
-Code Reference: lines 16 to line 21 from C.1

Each time a new point is selected, it is saved and published on the *send_goal* topic, which is read by the respective subscriber node: *path_planner*.
-Code Reference: line 23 to line 36 C.1

5.2.4 Computing the distances - *path_planner*

Once that the goal has been set in the graphic interface (*drawing_interface*), and that the actual positions has been determined (*localisation_odometry*) both these information are sent to this node through different topics. Firstly, in path_planner all the information coming from the topics are acquired through callback functions.
-Code Reference: from line 33 to line 55 of D.1
Then, by subtracting to the goal quantities the actual ones, it's possible to obtain the distance that separates it from the desired goal. Once that x_todo and y_todo are computed, the actual angle of which the Drawbot should rotate to reach the correct orientation is determined by exploiting the following formula:

$$\theta\_todo = \arctan(\frac{y\_todo}{x\_todo})$$

-Code Reference: from line 53 to line 63 of D.1

Then, to optimize the rotation, if the angle to do is greater than 180 degrees the Drawbot spins clockwise instead of counterclockwise, since the rotation is shorter.
Finally, once that the angle around which it has to rotate is known, it's possible to compute the actual distance that the robot has to do after such rotation:

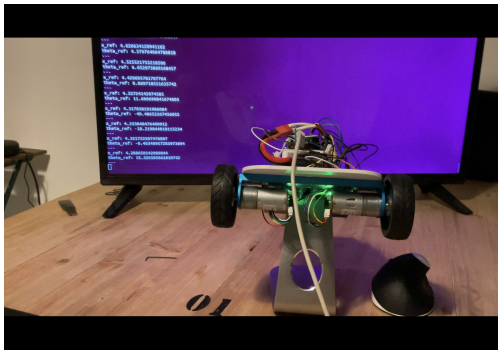
$$x\_ref = \frac{y\_todo}{x\_todo}$$

If the x_ref is smaller than 2 cm, it is assumed that the robot is closed enough to the goal, then x_ref is set to zero as well as θ_todo.
-Code Reference: from line 64 to line 75 of D.1

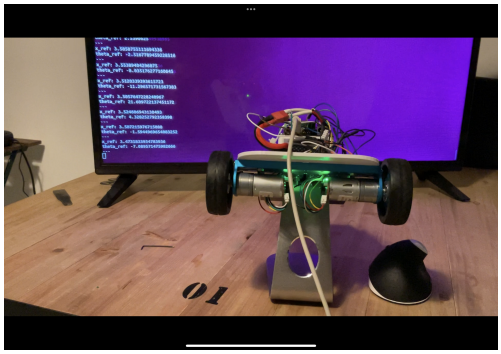
Here, two brief tests are proposed to evaluate the capability of the encoders to determine the actual position and to determine if the implemented nodes are able to process the receive data as expected.

No Road Test

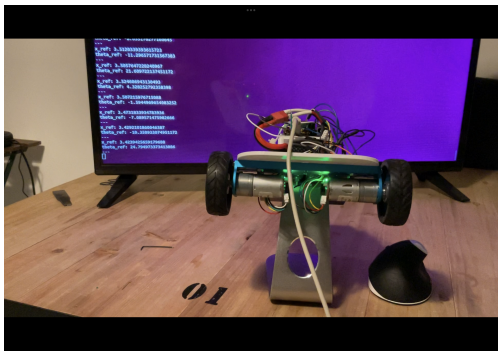
The goal is set by forcing a topic on *send_goal* through *rostopic pub*. Once that the total distance and angle are computed, the Drawbot starts to move accordingly to these parameter, decreasing step by step its distance.



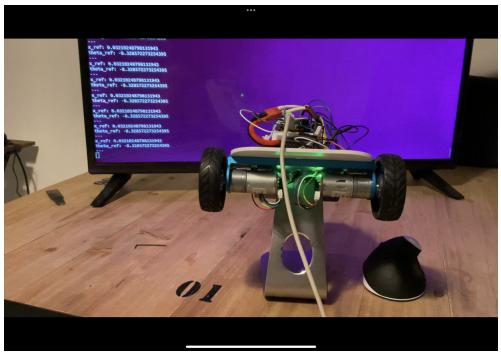
(a) Goal received, computed the first angle and distance



(b) orienting and shortening the distance (from 4.5m to 3.5m)



(c) getting closer to the goal 1.9m



(d) end phase once that the goal zone is reached

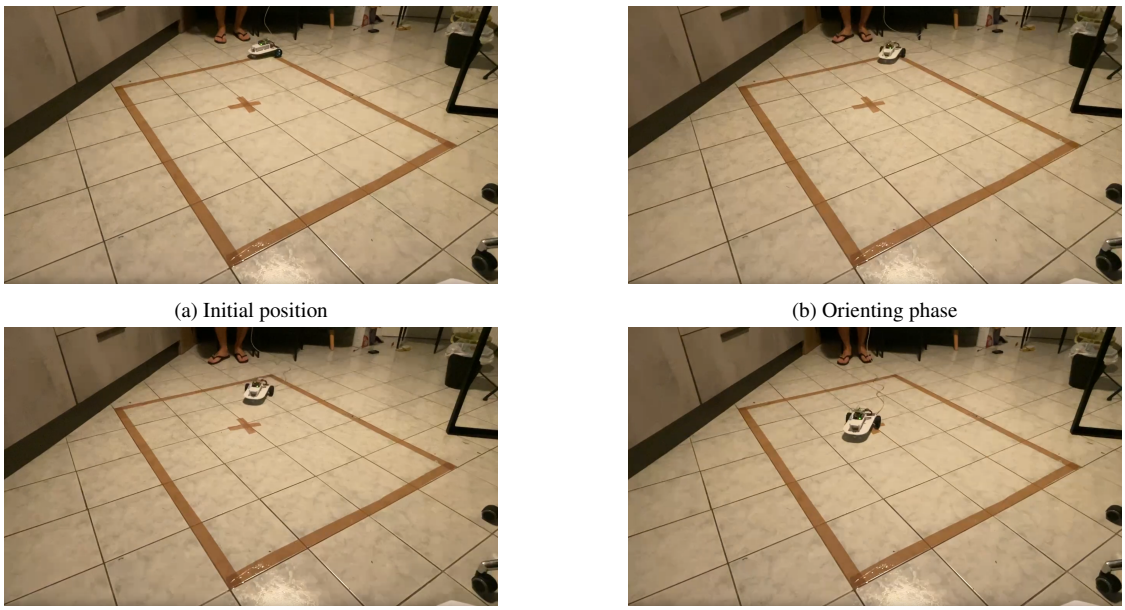
Figure 5.10: First test of autonomous assessment of the path to do

Road Test - single goal

The road test allows to determine if the algorithms works also in "non ideal" conditions, i.e. with friction intertia and a non smooth surface.

A single goal is then provided to the node *path_planner*, in this case (x=0.5cm , y=0.5cm). Several test are needed to properly set the internal and external speed that the robot during the rotation, since providing both wheels the same velocity won't rotate it as desired. This issue can be fixed by using a PI control, which regulates the speed of both the wheels.

Of course, using just the odometry in some configuration it won't be extremely precise and for what regards multiple goals it must be taken into account a possibly cumulative error. A possible way to improve this global localisation system, which will work in pairs with the odometry to provide a more accurate localisation. The road test is shown in figure 5.11.



(a) Initial position

(b) Orienting phase

(c) Straight line phase

(d) Final position

Figure 5.11: Test of the path planner

5.2.5 Determine velocities for the motors - *math*

The node *math* contains the whole algorithm able to determine the velocities to be sent to the motors, which will be managed by the Arduino node *complete_motor*. According to the reference distance and angle received by the node *path_planner* it is able to decide the type of movement (i.e. to rotate or to go straight) and to set the parameters dedicated to the right and left motor velocities according to the type of motion. The algorithm works as follows:

- **Rotation :** Once that the reference variables (angle of which to rotate and distance to do after the rotation) are taken from the topic, there is a first check on the angle, since the first adjustment is always on the angle, then on the distance to cover: if the angle is greater than 10° (or lower than -10°) and the distance to do is greater than 10 cm, then the robot will rotate. This is done to give a minimum range for the rotation, since if the robot rotate every time that there is even a small angle to do, then it will have a very unregular and jerky trajectory. On the other hand, setting a range like the aforementioned one it is able to go straight for a while if there's a small angle to do, which will increase by going forward and then rotate when the threshold θ is reached. Notice that, the condition on the distance (i.e. distance > 0.1) is realized to determine if the Drawbot already reached the "goal zone", defined by this range, that is the zone in which the robot is considered to be arrived to the goal. Going more into the code, once that the algorithm detects that there's a rotation to do, there's a second check on the sign of the θ, in this way is possible to determine the verse of rotation, setting as consequence the rpm for each motor. Different velocities are set for the internal and external motors, to compensate non ideal aspect such as friction, centre of gravity and so far and so on.
  - Code References Rotation: from line 49 to line 78 of E.1
- **Straight Movement :** Once that the rotation is done, there's the check for the forward movement: if the distance to cover is greater than 10cm the robot will proceed with the movement, setting both the wheels at the same speed [rpm], otherwise it will stop, setting all the velocities to zero, since it'll already be in the "goal zone".
  - Code References straight movement: from line 80 to line 108 of E.1
- Once that the velocities to be sent are computed, they are sent to the Arduino node *complete_motor* which will convert the rpm velocities in PWM and use them to move the motors.

5.3 Extra features developed but not incorporated in the final code

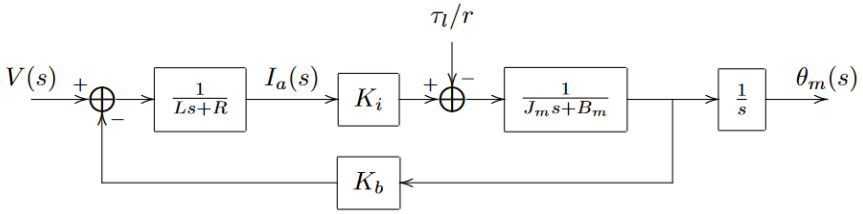
5.3.1 PI control

A PI control was chosen to not have a steady state error and a PID control was not selected to lower the complexity of the controller as no derivative gain was applied. In this section, two methods will be explained to obtain the PI gains: model based approach and trial and error. Both of these methods were tested, although one made more sense in the setting we were working with, this will be detailed later. The last part will show and explain the results of the PI developed for this project. Before starting, it is necessary to say that the motors did not come with a datasheet (only a table with very little information was given), therefor a lot of experiments had to be realised to retrieve the needed information.

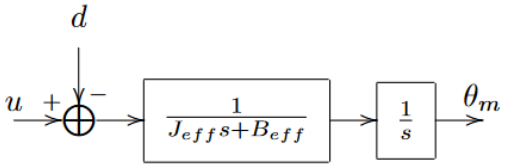
PI: Model Based Approach

The first approach that was chosen was to use control theory studied in others courses. To do this the motor had to be modelled. The schematic in figure 5.12 shows the complete model of a DC motor. However, because no datasheet was available, the parameters that are in the model had to be estimated experimentally. Here are the two main assumptions that were made in order to model the motor:

1. The electrical time constant can be neglected as it is way smaller than the mechanical one: $L/R << J_m/B_m$ with L and R, the armature inductance and resistance and J_m and B_m, the inertias and damping [6].
2. The motor can be approximated by a first order system using the effective inertias and damping terms [6].



(a) Complete block diagram for a DC motor system (Laplace domain)



(b) Simplified Open Loop block diagram (Laplace domain)

Figure 5.12: Block diagram of a DC motor (V is the voltage, θ_m is the motor velocity) [6]

Using these assumptions, the transfer function of the motor can be approximated graphically with a step response. A very sparse datasheet was given but the table in figure 5.13 gives us the velocity we should be seeing for a 12V step signal.

Model: JGA25-371		Data Sheet										
Voltage		No Load		Load				Stall		Reducer		Weight
Workable	Rated	Speed	Current	Speed	Current	Torque	Output	Torque	Current	Ratio	Size	Unit
Range	Volt.V	rpm	mA	rpm	mA	kg.cm	W	kg.cm	A	1:00	mm	g
6-24V	12	977	46	781	300	0.11	1.25	0.55	1	4.4	15	99
6-24V	12	463	46	370	300	0.23	1.25	1.1	1	9.28	17	99
6-24V	12	201	46	168	300	0.53	1.25	2.65	1	21.3	19	99
6-24V	12	126	46	100	300	0.85	1.25	4.2	1	34	21	99
6-24V	12	95	46	76	300	1.1	1.25	5.5	1	45	21	99
6-24V	12	55	46	44	300	1.95	1.25	9.7	1	78	23	99
6-24V	12	41	46	32	300	2.5	1.25	12.5	1	103	23	99
6-24V	12	25	46	20	300	4.2	1.25	21	1	171	25	99
6-24V	12	19	46	15	300	5.6	1.25	28	1	226	25	99
6-24V	12	11	46	8.8	300	9.45	1.25	47	1	378	27	99
6-24V	12	8.6	46	6.8	300	12	1.25	60	1	500	27	99

Figure 5.13: Information on the motor [3]

With the expected value of $977rpm$ with no load at 12V, the steady state error can be retrieved and with the second parameter, the time constant, the first order model can be retrieved. This was done in the electronics laboratory to make sure the input was 12V as the battery gives a slightly lower voltage then the wanted 12V. Here are the model values (J and B are the effective inertias and damping of the motor) obtained from the graph in figure 5.14.

$$G(s) = \frac{V_\infty}{\tau s+1} \qquad G(s) = \frac{0.9304}{0.232s+1}$$
$$= \frac{1}{Js+B} \qquad = \frac{1}{0.2494s+1.0748}$$

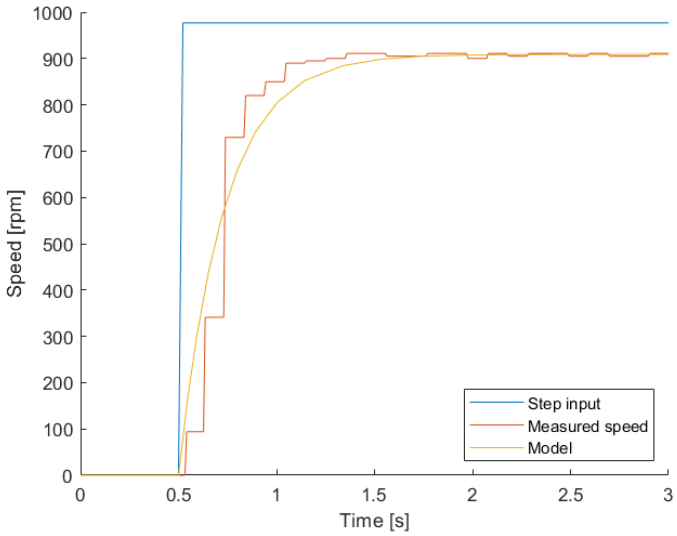


Figure 5.14: Step response of the motor for a 12V step input

Now using the values from the experiment, the PI gains can be retrieved by doing the following math based on the controller in figure 5.15.

$$V = \frac{1}{Js+B}(K_p + K_i/s)(V_R - V)V(1 + \frac{K_p + K_i/s}{Js+B} = \frac{1}{Js+B}(K_p + K_i/s)V_R V = \frac{sK_p + K_i}{Js^2 + (B+K_p)s + K_i}V_R$$

with V the velocity of the motor and V_R the reference velocity. The characteristic polynomial can be tuned with K_i and K_p:

$$s^2 + \frac{B+K_p}{J}s + \frac{K_i}{J} = s^2 + 2\zeta\omega s + \omega^2$$

The gains can be found with this formula:

$$K_p = 2\zeta\omega J - B \qquad\qquad K_i = J\omega^2$$

By choosing the damping coefficient ζ to 1 (to not get an overshoot) and taking 10 for the natural frequency of the system ω to get a relatively fast system, we get our two gains: $K_p = 3.91$ and $K_i = 24.93$.

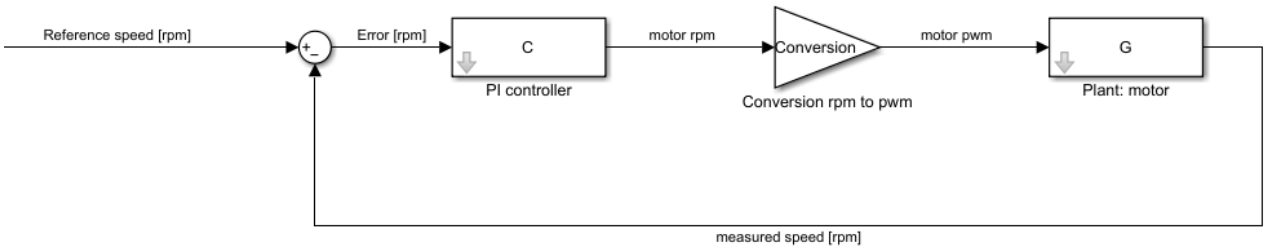


Figure 5.15: Schematic of the controller

When implementing this controller in the code, a problem was encountered. The error and sum of error terms are calculated in rpm but the input sent to the motor is a pwm value between 0 and 255. An extra block should be implemented in the control to bring the rpm value to a pwm one. This proved more difficult than anticipated as this was highly non linear and mapping this experimentally was difficult. For this reason, the gains obtained by model based approach were not used,instead a trial and error method was employed. This will be described in the next section.

PI: Trial and error Method

The model based approach was not successful in giving us a functional controller, therefor the trial and error tuning method will be adopted to fix this. A model based approach seems more relevant and interesting than this one, however designing a controller for an unknown plant is still something common when the plant can not be characterized. The methodology that was used was still well thought through. It is based on [4].
As the system has a process variable (measured speed) that changes rapidly when the control variable (reference speed) changes, the proportional gain was set to 0.1 (the method worked better with the integral term set to 0 at the start). The

gain was then incremented until the output oscillated around the reference value. This gain was then reduced a bit (from 0.9 to 0.5) and the integral term was then adjusted to give almost no oscillations. This method takes time but gave us a relatively good controller.

Results and usage of the PI control

All the tests were done with one motor but the obtained gains were tested on the other wheel also. This was done because of a hardware issue that prevented the testing of both wheels at the same time. To demonstrate the good functioning of the PI for one wheel without touching the ground, the reference speed (in rpm) was changed a lot in order to see if the measured speed would follow. This is shown in figure 5.16. The results were promising and the decision was made to keep these gains even if the response of the system could be made faster when the motor starts at 0 rpm.

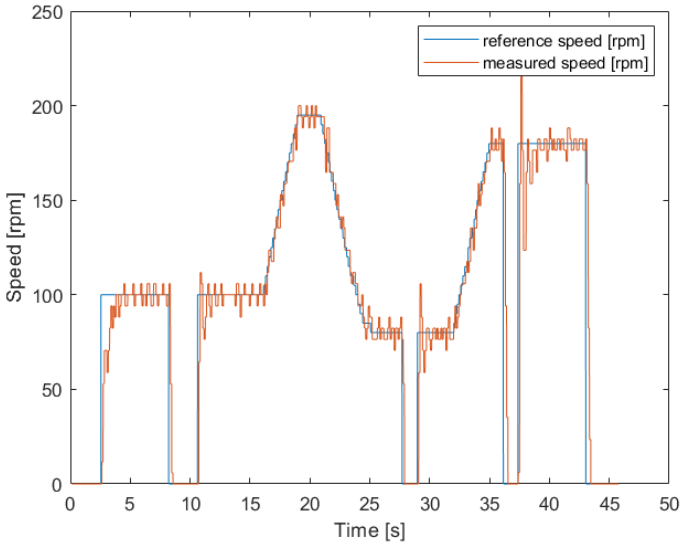


Figure 5.16: Test of the PI for different reference speeds

However, these tests were conducted without the wheel touching the floor. When solving the hardware issue and testing the robot on the floor with the PI, the results were not convincing. This problem was probably due to the gains of the PI and not the rest of the coding, as the added disturbance from the inertia of the robot and extra friction were not present previously. Because of a lack of time and the will to test the other blocks (odometry and path planning), solving this issue was set aside and the other blocks were tested without the PI. The low-level control used to test the other blocks was a simple open loop with a pwm value sent to the motors. With more time, more work could be put in solving this issue, after assuring the good functioning of the other blocks. With more time, the gains of the PI controller could be changed to get a better behaviour and to use a closed loop control.

5.3.2 Wireless control

A further implementation which might be done is to get rid of the cable connection between the PC and the Arduino, by using a more convenient wireless connection. This option was studied but not implemented. The best solution was to equip the Arduino with a Wi-Fi module, in particular the ESP8266, which is a low-cost Wi-Fi microchip, with also micro controller capability as the Arduino. Through a function of the ROS Serial node it is possible to establish a TCP/IP connection between the two devices, sending data also using a customized message. The main issue is to set up a two way communication between the module and the Arduino, in order to send the data received from the module to the Arduino and vice versa. Since both the micro-controller work with serial communication, the solution is to implement a I^2C communication protocol using the SDA (Serial data pin) and SCL (Serial clock pin), present on both the modules.

6 Final Assessment

6.1 Pros and Cons

Pros:

- Autonomous Drive;
- Mechanical Robustness;
- High battery autonomy;
- Intuitive GUI;
- High modularity of the code nodes with ROS.

Cons:

- No Wireless communication. Many hurdles had been encountered trying to put together the WiFi module and the Arduino.
- Encoders accuracy. Moreover, only one output of the encoders have been used; the one to understand the speed but not the one used to understand direction of rotation.
- Position of the center of mass (CoM). For this differential mobile robot is important to have the center of mass exactly in the middle of the wheels axis. In this way the CoM will coincide with the instantaneous center of zero velocity (ICZV). Here, due to the battery weight and position the Drawbot has a ICZV slightly. This issue is partly compensated by adding some weights on the back.
- Only broken-line paths. Not possible to draw paths with curves.
- No feedback on the velocity at the low level control. Only position feedback by the odometry.

6.2 To go further - If we had more time

With more time, the next logical step for this project would have been implementing a closed loop low-level control for the wheels using the PI code that was implemented, but adapting the gains to the robot. This would allow the wheels to turn at the desired speed. Then, the implementation of the wireless module could be a good addition for ease of testing and to prevent introducing errors from the user following the robot with the computer. Finally, to get more complex drawings, the code could be adapted to draw curved lines instead of only straight line profiles.

Another possible upgrade for the Drawbot, is the usage of the already implemented variable *change*, present in the node *math*. This variable is a boolean which is set to zero by default and updated each cycle if the type of rotation changes from a rotation to a straight movement or vice versa. This might represent a consistent upgrade, since in general when the robot is doing a movement (e.g. a straight movement) the wheels are spinning and if ever the type of movement change (e.g. from straight to a rotation) one of the wheel due to inertia will keep spinning while the other one has to stop its forward rotation and start to rotate on the other side. This will lead to a misalignment of the desired motion, which causes inaccurate trajectories. This might be solved by stopping the motion for a while (e.g. 0.5 seconds) whenever change=1 occurs, in this way the robot has the time to stop all the movements and then move the wheels in a symmetrical way.
Furthermore, one could also imagine fixing a pen to a servo on the robot to allow it to replicate a drawing from the interface on a larger surface on the floor. The servo would move the pen up in between different lines and down when the line has to be drawn.

However, for all wider and more complex drawings, it will become necessary to use a global localisation system as the error with the odometry accumulates over time. This could be done with a lidar or beacon recognition system and combine this information from the global localisation with the local localisation using kalman or markov algorithms. The main

drawback from this extra goal, aside from the time to implemement it, is the price of the extra components such as sensors (and possibly beacons). However, the fact ROS is used and allows high modularity means the integration of this new feature in the already existing code would be very simple.

6.3 Some real world implementation for Drawbot

Pushing the idea further, for a robot like Drawbot with a drawing/painting feature attached to it, our group came up with a few ideas of where such a robot could be useful. Implementing a function that could replicate an uploaded image instead of a live drawing by the user, would be an addition that allows even more applications. Aside from the artistic possibilities of replicating a drawing or painting on a much wider scale, there are practical ideas where the Drawbot could be useful: to paint the pedestrian crossing lines and other types of road markings, to draw the boundaries of a playing field (football, tennis...) or game lay-outs for children in parks, to trace temporary directions on the ground for large events such as festivals, to trace temporary art work or messages (the robot could be rented by companies), and many more ... .

7 Conclusion

In this report, the work of this semester was explained. After presenting the idea for Drawbot, the different sections go over the electronics, the mechanics and the coding part of the robot. Then the final assessment presented the strengths and weaknesses of the robot, the areas we would work on if we were to continue the project and lastly some ideas of real world scenarios where the Drawbot could be useful.

Before the closure of this project, we would like to thank the professor Rizzo for the opportunity to dive deeper into the subject of Robotics by working on the practical aspects connected to the project. We would also like to thank the Professor Primatesta for the advice and the reflection induced by our discussion when planning the work and different aspects on this project.

The project allowed us to learn a lot on the topic of robotics and more specifically the electronics and mechanics needed onboard and how they connect to each other but most importantly the working of an autonomous algorithm using localisation, path planner and control in general. It was important to us to use ROS, to familiar ourselves with it and to get comfortable with it in order to implement future projects with it.

This project was a success for us. The initial goal of creating from scratch an autonomous robot that replicates a drawing from an interface, using ROS for the coding, was reached and we are proud to present the work of these last months in this report.

Bibliography

[1] atotto. ros_odometry_publisher_example.py, 2016.

[2] Dynapar. Quadrature encoder overview, 2022. Last accessed 02 June 2022.

[3] Open Impulse. Jga25-371 dc gearmotor with encoder (126 rpm at 12 v), 2022. Last accessed 02 June 2022.

[4] Realpars. How to tune a pid controller, 2021. Last accessed 02 June 2022.

[5] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[6] M.W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. Wiley, 2005.

A Arduino Code

```
1
2  #include <ros.h>
3  #include <std_msgs/Int16.h>
4  #include <std_msgs/Float32.h>
5  #include <drawbot/Custom.h>
6  #include <drawbot/Data_arduino.h> //data_arduino
7
8  // Takes from the node math.py through the topic "send_motor_velocities" the
       reference speed for the two motors (in rpm).
9
10 ros::NodeHandle nh; //ROS node handler setup
11
12
13 //------------ PIN ASSIGNEMENT ------------
14 // Motor encoder output pulse per rotation (change as required)
15 #define ENC_COUNT_REV 102.0 //Each 102 triggers the wheel makes a complete
       rotation (.0 since we need a float)
16
17 // Encoder output to Arduino Interrupt pin (careful only 2interupt pins on
       arduino 3 and 2)
18 #define ENC_IN_r 3
19 #define ENC_IN_l 2
20
21 // Encoder phase B
22 //#define ENC_B_r  5
23 //#define ENC_B_l  4
24
25 //LEFT MOTOR --> M2
26 // MD10C PWM connected to pin 10
27 #define PWM_l 6
28 // MD10C DIR connected to pin 12
29 #define DIR_l 7
30
31 //RIGHT MOTOR --> M1
32 // MD10C PWM connected to pin 10
33 #define PWM_r 5
34 // MD10C DIR connected to pin 12
35 #define DIR_r 4
36
37
38 //-------------- VARIABLE ASSIGNEMENT ------------------
39 // Pulse count from encoder
40 volatile long encoderValue_r = 0;
41 volatile long encoderValue_l = 0;
42
43 // One-second interval for measurements
44    int interval = 100; //sampling time 0.1s
45    int interval2 = 70; //sampling time 0.05s
46
47 // Counters for milliseconds during interval
```

```
48  long previousMillis = 0;
49  long previousInternalMillis = 0;
50  long currentMillis = 0;
51  long internalMillis = 0;
52
53  // Variable for RPM measuerment
54  int rpm_l = 0;
55  int rpm_r = 0;
56  double speed_encoder_l = 0;
57  double speed_encoder_r = 0;
58  double reference_speed_l = 0; //reference speed in rpm
59  double reference_speed_r = 0;
60  double motor_pwm[2]={0,0};
61  int motor_inputs_pwm[2]={0,0};
62
63  double dir_r;
64  double dir_l;
65
66  long currentTime = 0;
67  long previousTime = 0;
68  long StartRecordTime = 0;
69  double dt=0.0;
70
71  //end variable definition section
72
73
74
75  void messageCb( const drawbot::Custom& toggle_msg){ //just one function to take
        all the data from the topic Custom
76    reference_speed_l = toggle_msg.motor_l; //rpm vel
77    reference_speed_r = toggle_msg.motor_r; //rpm vel
78  }
79
80
81  //Ros subscribers
82
83  ros::Subscriber<drawbot::Custom> sub("send_motor_velocities", &messageCb);
84
85  //Ros publishers
86  drawbot::Custom rpm_values;
87  drawbot::Data_arduino data;
88  ros::Publisher pub_encoder("pub_encoder", &rpm_values); //publisher which
        publishes the encoder values
89  ros::Publisher data_arduino("data_arduino", &data); //publisher named
        data_arduino which publishes data through the topic Data_Arduino
90
91
92  void setup()
93  {
94    //----------------ARDUINO
95    // Setup Serial Monitor
96    Serial.begin(9600);
97
98    // Set encoder as input with internal pullup
99    pinMode(ENC_IN_r, INPUT_PULLUP);
100   pinMode(ENC_IN_l, INPUT_PULLUP);
101
102   // Set encoder phase B as input with internal pullup
103   //pinMode(ENC_B_r, INPUT_PULLUP);
104   //pinMode(ENC_B_l, INPUT_PULLUP);
105
```

```
106   // Set PWM and DIR connections as outputs
107   pinMode(PWM_l, OUTPUT);
108   pinMode(DIR_l, OUTPUT);
109
110   pinMode(PWM_r, OUTPUT);
111   pinMode(DIR_r, OUTPUT);
112
113   // Attach interrupt
114   attachInterrupt(digitalPinToInterrupt(ENC_IN_r), updateEncoder_r, RISING);
115   attachInterrupt(digitalPinToInterrupt(ENC_IN_l), updateEncoder_l, RISING);
116
117
118   // Setup initial values for timer
119   previousMillis = millis();
120   StartRecordTime = millis();
121
122
123   //---------------ROS
124   nh.initNode();
125   nh.subscribe(sub);
126   nh.advertise(pub_encoder); //setting the Arduino as a publisher named "
          pub_encoder"
127   nh.advertise(data_arduino); //setting the Arduino as a publisher named "
          data_arduino"
128
129 }
130
131 void loop()
132 {
133
134   // Update RPM value every second
135   currentMillis = millis();
136   internalMillis = millis();
137
138
139   if (currentMillis - previousMillis > interval) { // count 0.1 second
140     previousMillis = currentMillis;
141
142     speed_encoder_r = (float)(encoderValue_r*600.0/(ENC_COUNT_REV)); //it gives
          you the rotations per minutes for the right wheel
143     speed_encoder_l = (float)(encoderValue_l*600.0/(ENC_COUNT_REV)); //it gives
          you the rotations per minutes for the left wheel
144
145     rpm_values.motor_l = speed_encoder_l;
146     rpm_values.motor_r = speed_encoder_r;
147
148     //use this to show in output the direction read by the encoders instead of
          the rpm value
149     //rpm_values.motor_l = dir_l;
150     //rpm_values.motor_r = dir_r;
151
152     //Debug encoder value
153     pub_encoder.publish( &rpm_values );    //Publishes the encoders values
154     encoderValue_r = 0;
155     encoderValue_l = 0;
156   }
157
158   double * motor_inputs = computeMotorInput(speed_encoder_r, reference_speed_r,
          speed_encoder_l, reference_speed_l);
159
160   //direction
```

```
161   if(motor_inputs[1] >= 0){
162     digitalWrite(DIR_l, LOW);
163   }
164   else{
165     digitalWrite(DIR_l, HIGH);
166   }
167
168
169   if(motor_inputs[0]>=0){
170     digitalWrite(DIR_r, LOW);
171   }else{
172     digitalWrite(DIR_r, HIGH);
173   }
174
175   motor_inputs_pwm[0]=abs(motor_inputs[0])*255/1200; //conversion of the speed
          command from rpm to pwm
176   motor_inputs_pwm[1]=abs(motor_inputs[1])*255/1200; //conversion of the speed
          command from rpm to pwm
177
178
179   //boundary check for the motor PWM values
180   boundaryCheck(motor_inputs_pwm[0], PWM_r);
181   boundaryCheck(motor_inputs_pwm[1], PWM_l);
182
183   nh.spinOnce();
184   delay(5);
185 }
186
187
188
189 // FUNCTIONS
190
191 void updateEncoder_r()
192 {
193   // Increment value for each pulse from encoder
194   encoderValue_r++;
195   //if(digitalRead(ENC_B_r) == HIGH){    //1 is
196   //  dir_r=1;
197   //}
198   //else{
199   //  dir_r=0;
200   //}
201 }
202
203 void updateEncoder_l()
204 {
205   // Increment value for each pulse from encoder
206   encoderValue_l++;
207  // if(digitalRead(ENC_B_l) == HIGH){    //1 is
208  //    dir_l=1;
209  // }
210  // else{
211  //    dir_l=0;
212  // }
213 }
214
215 double * computeMotorInput(double measured_speed_r, double ref_speed_r, double
      measured_speed_l, double ref_speed_l) //computes rpm value needed for the
      motor
216 {
217
```

```
218         motor_pwm[0]=ref_speed_r;
219         motor_pwm[1]=ref_speed_l;
220
221 // Setting variables to send through the topic "data"
222         data.err_speed_r=0;
223         data.err_speed_l=0;
224         //data.err_speed_r=dir_r;
225         //data.err_speed_l=dir_l;
226         data.err_sum_speed_r=0;
227         data.err_sum_speed_l=0;
228         data.ref_speed_r=ref_speed_r;
229         data.ref_speed_l=ref_speed_l;
230         data.controller_speed_r=motor_pwm[0];
231         data.controller_speed_l=motor_pwm[1];
232
233   //since we do not have the second phase from the encoder we have to set the
        sign of the velocities in this way
234   if (ref_speed_r<0 && measured_speed_r>0 ){
235     measured_speed_r=-measured_speed_r;
236
237   }
238   if (ref_speed_l<0 && measured_speed_l>0 ){
239     measured_speed_l=-measured_speed_l;
240   }
241         data.mesu_speed_r=measured_speed_r;
242         data.mesu_speed_l=measured_speed_l;
243
244
245   if (internalMillis - previousInternalMillis > interval2){
246     previousInternalMillis = internalMillis;
247     data_arduino.publish( &data );
248
249    }
250
251
252         return motor_pwm;
253 }
254
255 void boundaryCheck(double mot_pwm, int PWM)
256 {
257  if(mot_pwm<255 & mot_pwm>0){
258             analogWrite(PWM, mot_pwm);
259  }
260  else{
261             if(mot_pwm>255){
262                     analogWrite(PWM, 255);
263             }
264             else{
265                     analogWrite(PWM, 0);
266             }
267  }
268 }
```

Listing A.1: Arduino code

B localisation_odometry Code

```
1  #! /usr/bin/env python3
2
3  import rospy
4  import csv
5  from drawbot.msg import Data_arduino
6  from drawbot.msg import Custom_odometry
7  import time
8  import numpy as np
9
10 """
11 takes the data sent from the arduino node "complete_motor" through the topic "
       data_arduino" and computes the odometry of the robot, allowing to determine
       its position in terms of coordinates x,y and theta which are gonna be
       transmitted through the topic "send_odometry"
12 """
13
14
15 #INITIALIZATION VARIABLES
16
17 #internal variables initialization
18 wheel_rad = 0.0325                                  #wheel radius [m]
19 wheel_dist = 0.178                                  #length between the center of
       both wheels [m]
20 init_node_time = time.time()    #takes note of the time [s]
21 previous_time = init_node_time  #previous time (needed to compute the elapsed
       time) [s]
22 current_time = 0
23
24 #output initialization
25 x = 0                                                          #actual position
        x [m]
26 y = 0                                                          #actual position
        y [m]
27 theta = 0                                                      #actual angle
       theta [rad]
28
29
30 def callback(data):
31
32     global previous_time
33     global x,y,theta
34
35     odom=Custom_odometry()
                  #Defining an object of the type custom_odometry
36     rpm_speed_r = data.mesu_speed_r
          #Retrieve speed left motor from encoders [rpm]
37     rpm_speed_l = data.mesu_speed_l
          #Retrieve speed right motor from encoders [rpm]
38     lin_wheel_r = 2*np.pi*wheel_rad*rpm_speed_r/60              #find the linear
           speed of the right wheel [m/s]
```

```
39      lin_wheel_l = 2*np.pi*wheel_rad*rpm_speed_l/60                  #find the linear
            speed of the left wheel [m/s]
40      vx = (lin_wheel_r+lin_wheel_l)/2                                        #get the
            linear speed vx of the robot [m/s]
41      vy = 0
                                    #non holonomous robot -> vy=0
42      vtheta = (lin_wheel_r-lin_wheel_l)/wheel_dist                   #get the angular
            speed of the robot [rad/s]
43
44      current_time = time.time()
45      dt = current_time-previous_time
           #elapsed time [s]
46      previous_time = current_time
47
48      x = x+(vx*np.cos(theta)-vy*np.sin(theta))*dt                    #compute the new
            actual position x [m]
49      y = y+(vx*np.sin(theta)+vy*np.cos(theta))*dt                    #compute the new
            actual position y [m]
50      theta = theta+vtheta*dt
                #compute the new actual angle theta [rad]
51
52      print(time.time()-init_node_time,x,y,theta)                     #just for
            visualization purposes, prints  (time, x, y, angle)
53
54      odom.x_actual = x
55      odom.y_actual = y
56      odom.theta_actual = theta
57
58      pub.publish(odom)
                #publish the stored values on the topic "send_odometry"
59
60
61
62  #ROS
63
64  rospy.init_node('localisation_odometry')        #define a node called
        localisation_odometry
65
66  rospy.Subscriber('data_arduino', Data_arduino, callback)        #set this node
        as a subscriber to "data_arduino" of the type Data_arduino
67
68  pub = rospy.Publisher('send_odometry', Custom_odometry , queue_size=1)  #set
        this node as publisher which publishes on the topic "send_odometry" of the
        type Custom_odometry
69
70  rospy.spin()
```

Listing B.1: localisation_odometry code

C drawing_interface Code

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

# Source: https://pythonguides.com/python-turtle-mouse/

"""
Generates a window in which through the cursor it's possible to set the path
    that the robot has to follow. In particular it traces a straight line
    starting from the origin of the axes (0,0) to the wanted point and provides
    the coordinates of such point. It is possible to set multiple goals one after
     the other. The size of the window can be regulated according to the
    workspace's dimension of the robot.
"""

from turtle import *
import turtle
import rospy
from drawbot.msg import Custom_interface
from std_msgs.msg import Int16

#VARIABLES
WIDTH, HEIGHT = 1600, 1000


screen = turtle.Screen()
screen.setup(WIDTH + 4, HEIGHT + 8)                                    # fudge factors
    due to window borders & title bar
screen.setworldcoordinates(0, 0, WIDTH, HEIGHT) # setting the origin form which
    the robot starts to move, in this way wherever we place it it will always be
    in the (0,0) of our window's frame

#ROS
rospy.init_node('drawing_interface')
pub = rospy.Publisher('send_goal', Custom_interface, queue_size=10)

#FUNCTIONS
def func(i, j):

  goal = Custom_interface()
  turtle.goto(i, j)
  goal.x = int(i)/1000
  goal.y = int(j)/1000
  turtle.write("x="+str(goal.x)+"mm\ny="+str(goal.y)+"mm")
  print(goal.x, goal.y)
  pub.publish(goal)                                                    #
      provides the goal coordinates on the topic "send_goal"


screen.onclick(func)
screen.mainloop()
```

Listing C.1: drawing_interface code

D path_planner Code

```
1  #! /usr/bin/env python3
2
3  import rospy
4  import numpy
5  from math import sqrt
6  from math import pi
7  from std_msgs.msg import Int16
8  from std_msgs.msg import Bool
9  from drawbot.msg import Custom_odometry
10 from drawbot.msg import Custom_references
11 from drawbot.msg import Custom_interface
12
13 """
14 Receives the measured data from the topic "send_odometry" and compare them with
       the goal variables obtained from the topic "send_goal", providing as output (
       as a custom message through the topic "send_references") the ref angle and
       position (theta_ref, x_ref)
15
16 theta_ref = how much to rotate before to go forward
17 x_ref = how much to go forward before to reach the goal position
18 """
19
20
21 #INITIALIZATION VARIABLES
22
23 #output initialization
24 x_ref = 0
25 theta_ref = 0
26
27 #input initialization, these coordinates should be provided by the topic "
       send_goal"
28 x_goal=0
29 y_goal=0
30
31
32
33 #CALLBACK FUNCTIONS
34
35 #callback funtion for the subscriber to "send_goal"
36 def callback(msg):
37
38         global x_goal, y_goal
39
40         x_goal = msg.x                  #taking the x_goal coordinate from the
               topic
41         y_goal = msg.y                  #taking the y_goal coordinate from the
               topic
42
43         print(x_goal, y_goal)   #print of the goal coordinates for visualization
               purposes
```

```
44
45
46 #callback function for the subscriber to "send_odometry"
47 def math_tmp(msg):
48
49         global x_goal, y_goal
50
51         Ref=Custom_references()
              #variable of the customized type, it'll use to send the needed data
              through the topic "send_references"
52
53         x_act=msg.x_actual
                    #actual position x [m]
54         y_act=msg.y_actual
                    #actual position y [m]
55         theta_act=msg.theta_actual
              #actual position theta [rad]
56
57         x_todo=x_goal-x_act
              #x distance from the actual goal
58         y_todo=y_goal-y_act
                    #y distance from the actual goal
59
60         theta_total_rad=numpy.arctan2(y_todo,x_todo)     #total angle to do [rad]
61         theta_todo_rad = theta_total_rad-theta_act              #effective angle
               to do [rad]
62         theta_todo_deg = theta_todo_rad*360/(2*pi)              #effective angle
               to do [deg]
63
64         #if the angle is greater than 180 degrees, instead of rotating for such
              angle it's faster to rotate in the opposite direction with a lower
              angle (i.e. 360-x)
65         if(theta_todo_deg > 180):
66                 #print("%f prima -- " %(theta_todo_deg))        #just for
                       visualization purposes
67                 theta_todo_deg=-(360-theta_todo_deg)
68                 #print("%f dopo/n" %(theta_todo_deg))           #just for
                       visualization purposes
69
70         x_ref = sqrt(x_todo**2+y_todo**2)                                  #
              effective distance to do [m]
71
72         #reached goal area, once that the distance from the goal is less than 2
              cm stop (set to 0 the references)
73         if(x_ref <= 0.02):
74                 x_ref = 0

75                 theta_todo_deg = 0
76
77         Ref.x_ref=x_ref
                    #reference position [m]
78         Ref.theta_ref=theta_todo_deg                                       #
              reference angle [deg]
79
80         pub.publish(Ref)
                    #publish the stored values on the topic "send_references"
81
82
83
84 #ROS
85
```

```
86 rospy.init_node('path_planner')
                                                    #define a node called
    path_planner
87
88 sub_from_odometry = rospy.Subscriber('send_odometry', Custom_odometry, math_tmp)
       #set this node as a subscriber to "localisation_odometry" of the type
    Custom_odometry
89
90 sub_from_interface = rospy.Subscriber('send_goal', Custom_interface, callback)
           #set this node as a subscriber to "drawing_interface" of the type
    Custom_interface
91
92 pub = rospy.Publisher('send_references', Custom_references, queue_size=1)
                    #set this node as publisher which publishes on the topic "
    send_references" of the type Custom_references
93
94 rospy.spin()
```

Listing D.1: path_planner code

E math Code

```
1  #! /usr/bin/env python3
2
3  import rospy
4  import time
5  from geometry_msgs.msg import Twist
6  from std_msgs.msg import Int16
7  from std_msgs.msg import Bool
8  from drawbot.msg import Custom_references
9  from drawbot.msg import Custom
10
11 """
12 takes from "send_references" the two referece values of the velociy: theta_ref (
       i.e. how much to rotate) and x_ref (i.e. how much to go straight). It
       determines if the former command was a rotation or a translation in order to
       understand if we had a change between the former and the actual command (in
       this case change will be set to 1) and compute the command to send to arduino
        for the rotation or for the translation according to the case. The priority
       will always be given to the rotation and iff it is satisfied, we'll proceed
       in the forward movement. Finally it will provide as output the two velocities
        in rpm to send to the motor: ref_speed_r, ref_speed_l ,through the topic
       Send_motor_velocities.
13 """
14
15
16
17 #INITIALIZING VARIABLES
18
19 #initializing internal variables
20 form_rot = 0    #if 1 then the former command was a rotation
21 form_lin = 0    #if 1 then the former command was a translation
22 vel_lin = 0     #linear velocity [m/s]
23 change = 0
24
25 internal_vel = 580
26 external_vel = 465
27 linear_vel = 465
28
29
30
31
32 # dobbiamo evitare che al primo cambio change si attivi magari con una seconda
       variabile "first_change"
33
34 #FUNCTIONS AND CALLBACKS
35
36 def set_velocity(pos_lin, pos_ang):
37
38         global form_rot, form_lin
39         global vel_lin
40         global flag
```

```
41          global previous_time, interval
42          global change
43          global internal_vel
44          global external_vel
45          global linear_vel
46
47          Info=Custom()
48
49          if((pos_ang<-10.0 or pos_ang>10.0) and (pos_lin>0.1)): #in this way we're letting the robot rotate only from a certain treshold value otherwise it'll stop every time there's even a degree of rotation to do
50
51          # We are in a rotational motion
52
53                  vel_lin = 0             #null linear velocity, since we have to rotate
54                  form_rot = 1    #we are rotating so set to 1 for the next iteration
55
56                  #check for the variable "change"
57                  if(form_lin) :
58                          change=1
59                          form_rot=1
60                          form_lin=0
61
62                  #left rotation
63                  if(pos_ang>0) :
64
65                          right_motor = int(external_vel)
                                    #right motor speed [rpm]
66                          left_motor = int(-internal_vel)
                                    #left motor speed [rpm]
67                          print("right %d -- left %d" %(right_motor, left_motor))
                                                        #print of the two sent speed, just for visualization purposes
68                          Info.motor_r=right_motor
69                          Info.motor_l=left_motor
70
71                  #right rotation
72                  else:
73
74                          right_motor = int(-internal_vel)
                                    #right motor speed [rpm]
75                          left_motor = int(external_vel)
                                    #left motor speed [rpm]
76                          print("right %d -- left %d" %(right_motor, left_motor))
                                                        #print of the two sent speed, just for visualization purposes
77                          Info.motor_r=right_motor
78                          Info.motor_l=left_motor
79
80          else:
81
82          # We are in a linear motion
83
```

```
84                  #boundary for the distance, if the distance is lower than 10cm
                        then do not go forward
85                  if pos_lin > 0.1 :
86
87                          vel_ang=0                                   #null angular
                            velocity since we have to go forward
88                          vel_lin=linear_vel                  #linear velocity not
                            null since we are in a linear motion
89                          form_lin=1
90
91
92                  #the boundary for the distance is not respected, so we should
                        stop
93                  else:
94                          vel_ang= 0
95                          vel_lin= 0
96                          form_lin= 1
97
98                  if(form_rot):
99                          change=1
100                         form_rot=0
101                         form_lin=1
102
103                 val = int(vel_lin)
104
105                 print("right␣%d␣--␣left␣%d" %(val, val))        #print of the
                        two sent speed, just for visualization purposes
106
107                 Info.motor_r=val                            #right motor speed [rpm]
108                 Info.motor_l=val                            #left motor speed [rpm]
109
110
111         pub.publish(Info)                                   #publish the two speed
                of the topic [rpm] on the topic
112
113
114 #callback function of the topic "send_references"
115 def callback(msg):
116         set_velocity(msg.x_ref, msg.theta_ref)
117
118
119
120 #ROS
121
122 rospy.init_node('math') #define a node called "math"
123
124 sub = rospy.Subscriber('send_references', Custom_references, callback)  #set
    this node as a subscriber to "send_references" of the type Custom_references
125
126 pub = rospy.Publisher('send_motor_velocities', Custom , queue_size=1)   #set
    this node as publisher which publishes on the topic "send_motor_velocities"
    of the type Custom
127
128 rospy.spin()
```

Listing E.1: math code