# 01_Motors control

In this robot project, 3 modes of functioning were defined. The approach for Mode 1 and Mode 2 in terms of movements is similar. We need to have the robot go forward, backward, turn right and left. In Mode 3, we want the robot to know exactly his position in real time, and be able to go to a particular position that would be sent to him. In this wiki page, I explain the approach I had regarding the robot's movements from the beginning.

The very first step was to control the motors using an H-Bridge. In order to have the motors going straight, I tested various values of PWM with each motor until I had a pair of values that would make the robot go straight. This approach was not good at all. Any external factor would change this calibration (such as different floor, different battery level etc.).

The second approach (used for Mode 1 and Mode 2) was to use the encoders data to enslave one motor to the other. I designed a proportional controller that would make sure both motors are going at the same speed. Using a feedback loop, the controller could adapt to external disruptions.

The third approach (used for Mode 3) was to implement odometry to compute the robot's position in real time and to implement a more advanced controller. This controller would take in parameter a position and correct the angular and the linear error of this robot to make go it to a precise location.

All the details regarding the structure, the materials, the sensors I used are below.

## Introduction

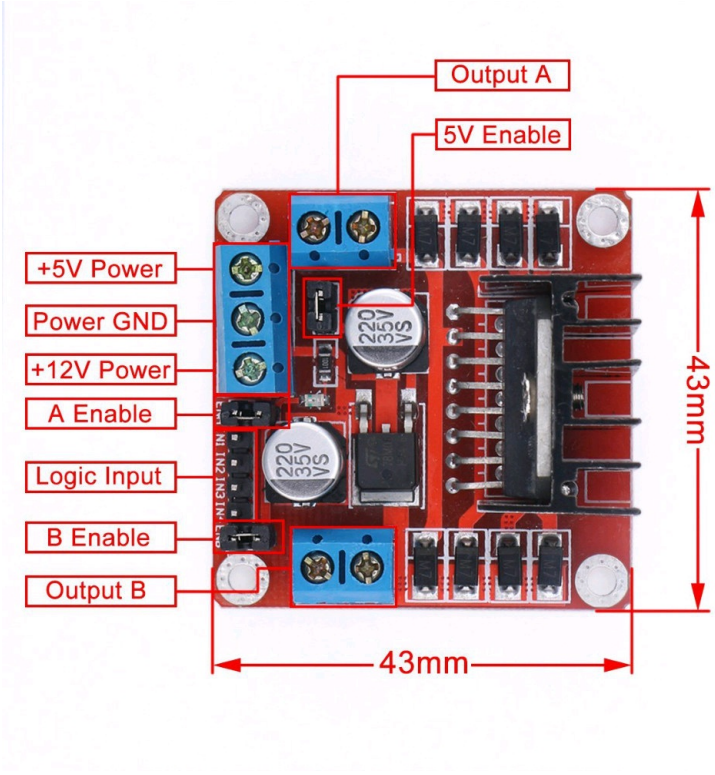
The first thing I started to work on is the robots movements. The equipment we had is consists of 2 wheels with DC motors (Ref TFK-280SA-22125 MOTOR), a frame, one H-Bridge (Ref STM L298N), 2 encoders (Ref FC-03, one for each wheel) and an AURIX™ TC297 B-step Application Kit.

At first, we want to control the motors, to do so we just need to supply each motor with current. The problem is the that the output pins of a microcontroller do not supply enough current to trigger the motors, which is normal. Therefore we decided to power supply the wheels' motors using an H-Bridge that would be powered by an external Voltage Source (a battery in our case) and controlled with the GPIOs of the microcontroller.

## The H-Bridge

Code: Motors files

The H-Bridge we use is the STM L298n, you can find the L298 datasheet in the link below, here is an image showing its ports




Link to the L298n datasheet

As you can see there are 4 logic inputs, 2 enable pins and 2 outputs (one for each motor). In order to get a voltage in the output A (or B)

you need to put the EnA pin (EnB) at a high state and depending on the polarity of the output A (output B) pins, you need to send to the logic input In1 = 1 and In2 = 0 (In3 = 1 and In4 = 0) or In1 = 0 and In2 = 1 (In3 = 0 and In4 = 1). This is what we use to make the motors go forward or backward.

Once you are supplying the H-Bridge with an external power supply and you have configured the enable pins and logic input, you will notice that you will have a voltage equal to the supply of the H-Bridge in the output pins.

Once the H-bridge is configured and we can supply the motors with enough current, we realize that the two wheels don't go at the same speed even if the motors are similar. Here is a link with the Motor's Datasheet

## Pulse Width Modulation

code: PWM_config

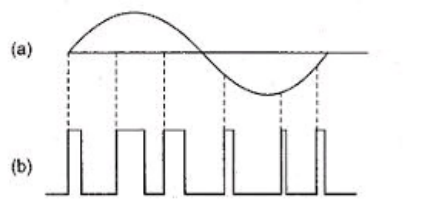



**Fig. 18.8** Pulse Width Modulation (a) Signal (b) PWM

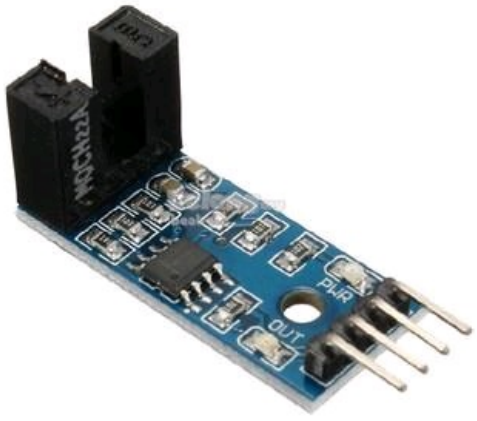
In order to get the robot moving forward, we need to get each wheel going at the same speed. The solution we used in this case was using a Pulse Width Modulation. By controlling the duty cycle of the PWM going to the logical inputs of the H-Bridge, we can change the voltage supplied to the motors. With the basic configuration of the H-Bridge specified previously, the outputs were supplying the voltage that the H Bridge was supplied with. For example, if you supply the H-Bridge with 9V and configure the H-Bridge as said, you will get 9V between the output pins.

The Pulse Width Modulation signal allows us to change that value to a wanted percentage of the supply voltage of the H-Bridge. Let's now say that the H-Bridge is supplied with 9V and we put EnA = 1, In1 = 50% PWM signal, In2 = 0. With this configuration, we will get an average of 4.5V between the A output pins.

In a nutshell, a PWM signal allows you to control the speed of a motor.

## Encoders

code: Encoders_config




Now that we can make our robot move in the directions we want, it would be useful to know the distance it travels and the angle it turns to. To do so, we use encoders. Here you can find a bit of information about how the Encoder works. Basically, we have a code disk with 20 holes, each time the wheel rolls and the encoders sees one hole, the encoder sends a pulse. By counting the amount of pulses the encoder sends to the microcontroller, we can calculate the distance the robot has traveled.

To collect the pulses sent from the encoder, we set up an external interrupt on the External Request Unit of the microcontroller, each time a specific input pin receives a pulse, it increments a counter.

Now that we know how to collect the pulses from the encoder, we can calculate the distance that the robot traveled. We measured the diameter of a wheel: d = 6,5cm. Therefore we know that when the encoder sent 20 pulses, the robot traveled a distance equal to the perimeter of the wheel. This is what helps us to calculate the amount of pulses we need to receive from the encoder to travel a certain

distance.

Since 20 pulses gives us a very poor precision, we used a little trick to increase the resolution. Instead of just counting the rising edges sent by the encoders, we also count the falling edges. This means that instead of receiving 20 ticks each time one wheel travels a distance equal to its perimeter, we receive 40 ticks. This is a very important step as it will lead us to having a better precision when counting the error between the two wheels for when we will start working on the enslavement of the robot.

For turning right and left:

We know that when the robot turns to a 90 degree angle for example, it travels a distance equal to 1/4 of a circle. Therefore, when a wheel travels a distance equal to divided by 4, our robot has turned to a 90 degree angle.

A more efficient way of doing that is to make the other wheel go in the other direction at the same time. Let's say we want to turn right, we can either make the left wheel travel a distance while the right wheel is not moving (or moving with a speed inferior to the Left wheel), or we can make the left wheel go forward for half that distance and make the right wheel go backwards for the same half distance. This is the method we used to make our robot turn in every direction.

## Proportional Control

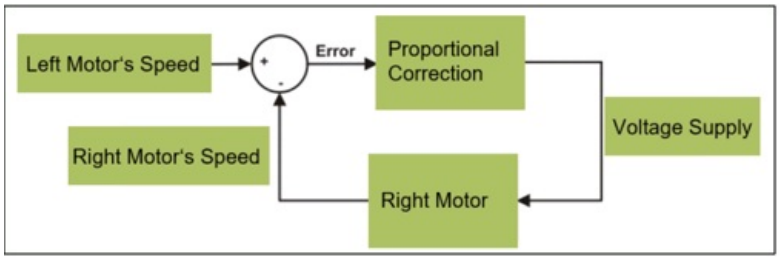
The single big problem that remains with the robot is that the two identical motors when fed the same voltage don't have a similar rotation per minute ratio. This is due to many reasons including the fact that when mounted on the robot, for it to go forward one motor has to have a rotation in the opposite direction than the other one, this means that the friction and various losses of energy of the motors are not similar. At first, to make the robot go forward, we had to test various PWM duty cycles for each of the motors until we could find one with which the robot goes forward with a straight trajectory. That was when we implemented the PWM to the project. We quickly realized that the right motor was going faster that the left one (thanks to the data of the encoders). So we fixed it's PWM duty to 40% and tried out many other duty cycles with higher values with the other motor. One good enough balance for going forward was found with the left wheel being given a 80% duty cycle and the right wheel a 46% duty cycle. For the going Backwards, new values had to be found since we change again the rotation of the motors.

Note : These duty cycle values are given as example. Since then, we changed the motors to find a better match.

### Proportional control version 1: Wheel enslavement

The values found are very specific, and do not permit to adapt to any disruption. The best/easiest way we found to make the robot go forward/backwards with a straight trajectory was to enslave one motor to the other one. The concept is fairly easy, we give a constant voltage value to the motors, which makes them rotate at a certain speed, and we measure the error between the two motors thanks to sensors (encoders) and with a feedback loop we keep changing the value of the voltage fed to the weaker motors so that we tend to have the smallest error possible.

Here is an illustration of what a proportional control enslavement looks like :




In the case of our robot, the set point is the PWM value given to the strongest motor (Right motor, around 40%). The error that we calculate is the difference of values captured by the encoder of each wheel. The equation for our system is the following:

PWM_Left = PWM_Right - (Proprotional_Factor * Error)

In general, as said earlier, the right wheel goes faster than the left wheel, therefore if we want to have a positive error we can calculate:

Error = EnoderRightCounter - EncoderLeftCounter

Once we have the value of the error, we have to multiply it by a proportionate factor. This one is almost entirely determined from testing. You can try to predict a range of values by knowing that for example: when going forward, the Right Wheel goes usually about 1,6 times faster than the Left wheel. Which means that the PWM value the Left Wheel should receive is around PWM_Right * 1,6. Then you can choose an average value of the error (usually between 2 and 5 with a good enough enslavement) to estimate the value of your coefficient. Please bear in mind that the coefficient values are different for every direction.

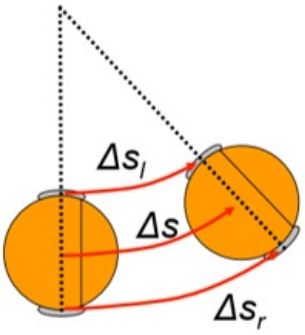
Let's now see how this is structured in the code for the Mode 1 of the robot. There is a Forward (Backward, Right and Left) function that sets the H-Bridges inputs and sets the PWM values. Meanwhile, there is a timer interrupt that adjusts the Left PWM according to the error

that is measured each 10ms. When you send a command via Bluetooth, it calls one of the functions which sets the robot to go towards your desired direction and activates the enslavement in the interrupt for that specific direction.

### Mode 3: Advanced Control Algorithm

For the third mode of the robot, a more advanced control algorithm was needed. A Convolutional Neural Network (CNN) detects symbols on the floor and sends their position to the microcontroller. The microcontroller then has to make the robot move to the location of the symbol.

In this case, the instruction sent to the microcontroller is the location of the symbol. For the microcontroller to process and understand this type of instruction, it has to know the location of the robot in real time. Therefore, I used odometry to compute the position of the robot. Odometry is the use of the data from motion sensors to estimate change in position over time. You can find information regarding odometry by clicking in this link. In that document, the odometry is done with the encoders data only. Here is what it looks like:




Here, 's' represents the variation in encoders ticks (which can be converted into centimeters), and the Δ means "variation". The formula used to update the position of the robot using encoders only if the following.

$$\Delta x = \Delta s \cos(\theta + \Delta\theta/2)$$

$$\Delta y = \Delta s \sin(\theta + \Delta\theta/2)$$

$$\Delta\theta = \frac{\Delta s_r - \Delta s_l}{b}$$

$$\Delta s = \frac{\Delta s_r + \Delta s_l}{2}$$

$$p' = f(x, y, \theta, \Delta s_r, \Delta s_l) = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \frac{\Delta s_r + \Delta s_l}{2} \cos\left(\theta + \frac{\Delta s_r - \Delta s_l}{2b}\right) \\ \frac{\Delta s_r + \Delta s_l}{2} \sin\left(\theta + \frac{\Delta s_r - \Delta s_l}{2b}\right) \\ \frac{\Delta s_r - \Delta s_l}{b} \end{bmatrix}$$

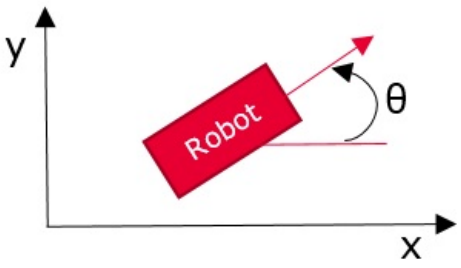
In my case, I was able to use the data from encoders and from a 9-axis Microelectromechanical Systems (MEMS) sensor containing an accelerometer, magnetometer and gyroscope (The sensor implementation is explained in the another wiki page). The 9-axis MEMS sensor gives information regarding the angle orientation of the robot in three dimensions. Therefore, I could use the value from the encoders to compute the distance with a straight trajectory, and the value from this sensor to measure the angle variation while turning. After adapting the angular variation in the formula shown above, I had a good estimation of the robot's location in real time.

Here is an another picture describing the calculation of the robot's location.

$$X = X + \Delta X * \cos(\theta + \Delta\theta)$$

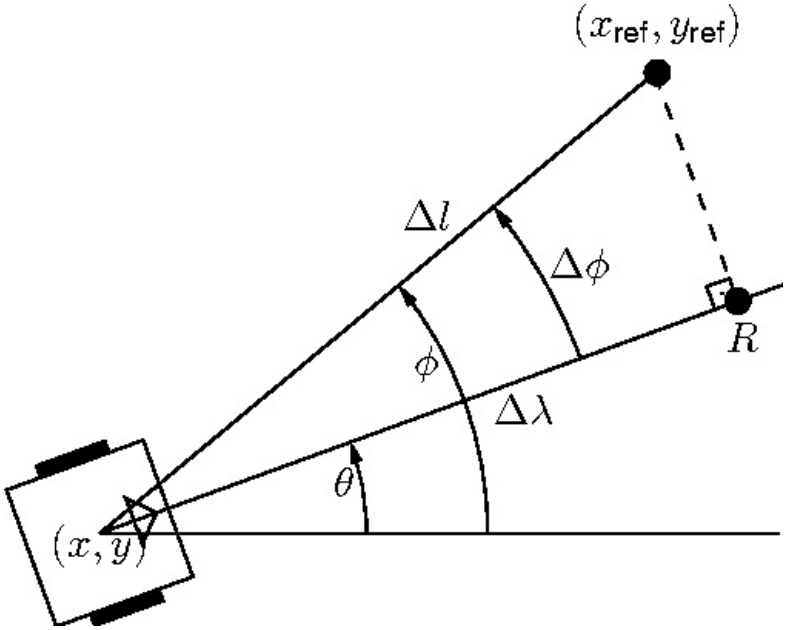
$$Y = Y + \Delta Y * \sin(\theta + \Delta\theta)$$

$$\theta = \theta + \Delta\theta$$




Now that the robot can process a location, we can implement a control algorithm that takes in parameter a location. Here is the link containing all the information I used to implement this enslavement.

The idea is simple. The robot knows its current location and is given in parameter a location (in X, Y Cartesian coordinates) to which he has to go. Using this information, we can compute an overall error called Δl in the picture below.




This error is broken down in two parameters: * Δλ represents the linear error. This is the distance the robot should travel with a straightforward trajectory. * ΔΦ represents the error in orientation. The robot should correct this error before moving anywhere.

Indeed, if the robot has an orientation error superior to 180 degrees, if it moves it will only increase its linear error.

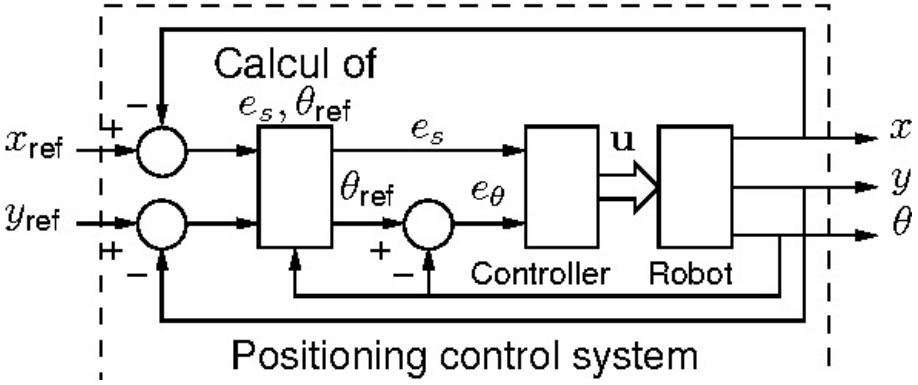
The formulas used to calculate the linear error and the error in orientation are shown below: 'eΘ' is the orientation error and 'es' is the linear error

To make $e_\theta = \Delta\phi$, we just need to define $\theta_{\text{ref}} = \phi$, so $e_\theta = \theta_{\text{ref}} - \theta = \phi - \theta = \Delta\phi$. For this, we make:

$$\theta_{\text{ref}} = \tan^{-1}\left(\frac{y_{\text{ref}} - y}{x_{\text{ref}} - x}\right) = \tan^{-1}\left(\frac{\Delta y_{\text{ref}}}{\Delta x_{\text{ref}}}\right) \quad (6)$$

$$e_s = \Delta\lambda = \Delta l \cdot \cos(\Delta\phi) = \quad (7)$$
$$\sqrt{(\Delta x_{\text{ref}})^2 + (\Delta y_{\text{ref}})^2} \cdot \cos\left[\tan^{-1}\left(\frac{\Delta y_{\text{ref}}}{\Delta x_{\text{ref}}}\right) - \theta\right]$$

Once we have calculated, we just need to implement a control algorithm following this example:




In order to convert this in the code, we used a state machine. An timer interrupt occurring every 100ms calculates the location of the robot and determines the errors. If a new location is sent to the robot, the state machine is reset. The steps of the state machine are the following: * Aurix_State_Wait: Calibrating the angle value * Aurix_state_ROTATION: correcting the orientation up to a threshold of 0.15 radians * Aurix_State_Forward: the robot is set to go forward, the linear and orientation errors determine the trajectory.

The formulas controlling each motor's speed are shown below:

$$Right\ Motor\ Speed = Coef1 * Linear\ Error + Coef2 * Angular\ Error$$

$$Left\ Motor\ Speed = Coef1 * Linear\ Error - Coef2 * Angular\ Error$$

Amine Gaizi, Last modification 18/06/2019

---