­­­LAB4

The goal of this laboratory was to combine the line sensor from Lab 1 and the motor controller from Lab 3 to make the Turtlebot follow a given line autonomously.

The infrared line sensor:

We are using a Pololu QTR reflectance sensor which can detect how much infrared light has been reflected from a surface. With a certain threshold we can than determine if a surface is dark or light. The installed sensor is located on the lower front of the robot. It is about 10 cm wide and consists of an array of 8 photo diodes with each one photo transistor. The distance between every sensor is 8 mm. This gives us a total of 8 individual sensors that can differentiate between dark and light areas. This data is retrieved by the controller MCU through a port expander board using a I2C connection. The sensor data encoded in a binary format and received as an integer. To retrieve the original data, it is necessary to convert the integer back to a binary number. This binary number has 8 bits, each storing the value of one sensor.

Properties of the line:

The line is assumed to be wide enough to trigger at least one of the infrared sensors and at most two at the same time. The goal is to create a controller algorithm that keeps the line between the two middle sensors 3 and 4. We are implementing a tracking error and set this position to tracking error 0. We then define the tracking error to be greater than zero when the line is on the left side of the sensor. If the tracking error is less than zero, the line is on the right side of the sensor.

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

Automatisch generierte Beschreibung

Figure 1 Spacing of the infrared sensors



Figure 2 Main structure of the line following code

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Figure 3 Function to convert integer to binary

In LAB1 we already created a function to get the binary values of the integer we received by the line sensor. However, the function would give us an integer again with all the bits in a single number. This was not a very useful implementation. We needed to get the bits in an array. We define an empty array in the callback function (figure 4) and give the pointer to the findBinary function. The basic decimal to binary converter in form of a while function would then store every bit in this array, figure 3.

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Figure 4 Top part of the timer callback function

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Figure 5 Function to calculate the line error based on a binary array

The calc\_error\_line function shown in figure 5 does the calculation of the line error. The input of this function is an 8-dimensional array, containing the state of each infrared sensor coded in binary. A logical one means, that the corresponding sensor has detected a dark spot. In the for loop we iterate through every value in the binary[] array, ergo through all sensors data. We calculate the sum of the binary[] array to get the total number of triggered sensors. Furthermore, we are filling the distance\_from\_middle array with the opposing distance of every sensor from the middle. The distance between every sensor is 4mm. These values get negative to indicate a distance to the left and positive to indicate a distance to the right. The sum\_dist value is the sum of every distance from the center if the sensor was activated. Calculating the line\_error by dividing the sum of the distance of each activated sensor with the total number of activated sensors. This line\_ error ten represents the offset of the black line from the center, regardless of how wide the line is. Even if only one sensor is not activated, we still get a useful line error value.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Figure 6 Conversion from line error to yaw error

The calc\_yaw\_error function in figure 6 calculates a much stabler result for our error value. It includes the wheelbase dimension (D) and the distance between the line sensor and wheels (H). This determines the position of the turning point of the bot, which is between the two wheels, shown in Figure 7.



Figure 7 Schematic of the wheelbase and line sensor placement

D = 165 mm

H = 85 mm

Calculating the reference for the motors:

Based on the yaw error, we can now calculate our reference for the two motors. Since our motor controller is designed to compute rotations per minute, we set a base speed of 100 rpm and depending on the direction of the motor add or subtract our yaw error. To give the yaw error more effect on the base speed, we incorporated an additional gain. Testing different gains, we first found a relay able solution with a base speed of 100 rpm +- yaw\_error \* 12. By Increasing the gain, we got better performance in sharper corners. The drawback was increased instability on the straight segments. Small changes in the yaw error multiplied with a high gain gave the bot a very shaky ride in straight segments.

To furthermore improve the performance a linear or even an exponential gain was necessary. This would increase the impact of the yaw error in tighter corners and make sure to decrease the gain when the line is going straight. The robot would be very stable on straight lines and keep its agile performance in sharp corners. We tried implementing different if statements to check if the yaw error was in a certain range. Then we could set a different gain accordingly. This worked but needed a lot of tuning of all the values. In the end we saw that the values where basically the same as the yaw error itself.

A very simple solution was to square the yaw error. This would increase the gain linearly. To keep the sign of the yaw error, the absolute value of the yaw error was multiplied by itself, see figure 8. For tuning this setup to use different base speeds it is possible to also add a gain to increase the impact of the squared yaw error.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Figure 8 Calculating the reference signals for the motors

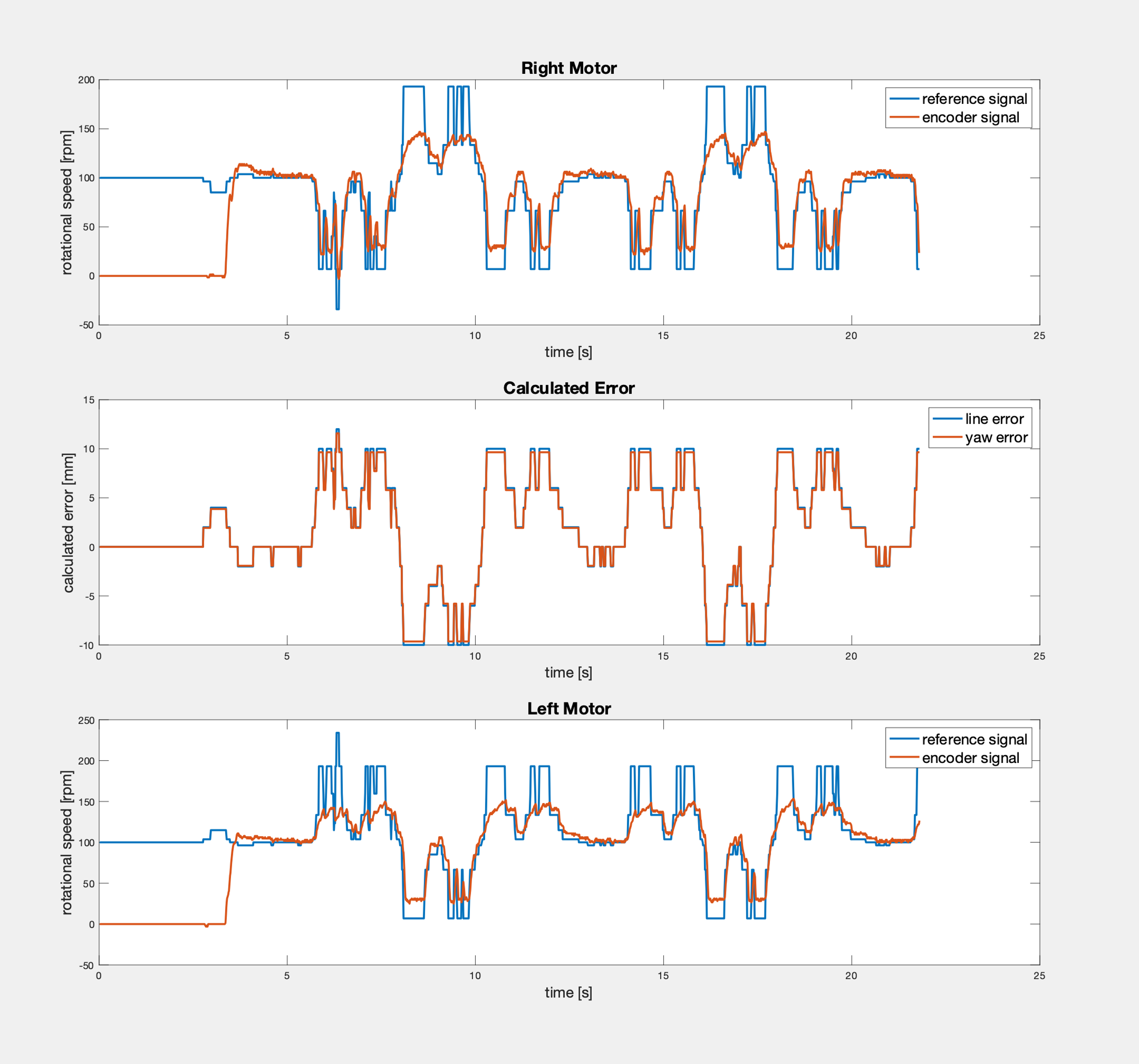


Figure 9 Plot of tracing one round on the line parkour

The plot of figure 9 shows the signals of the left and right motor. There is also shown the line and yaw error calculated from the data collected by the line sensor. The reference signal for each motor is then set by the functions shown in figure 8. It is clear to see that the encoder value of each motor is tracing the reference signal great. It is also clear to see the limitations of the motors. When the reference exceeds a given speed of around 150 rpm, the motors are limited by the duty cycle implemented in LAB 3. The solution for this would be to set a limit to the reference signal when calculated. Because of the geometry of the robot, calculating the yaw error results in a very similar value of the line error. This is because the ratio between wheelbase (D) to distance of line (H) sensor to wheels is almost one half, shown in figure 7.

During this course we worked with a TurtleBot that you can see in the Figure \ref{fig:turtlebot}.

During this course we worked on a mobile robotics platform equipped with motors, sensors, interfaces, and different processing units. In the following paper it will be referenced as TurtleBot shown in Figure \ref{fig:turtlebot}. The main processing unit is the STM32F767, it is connected directly or indirectly to all inputs and outputs via GPIOs.

Acknowledge

The SX1509 is complete ultra

low voltage 1.2V to 3.6V General Purpose parallel

Input/Output (GPIO) expanders ideal for low power

handheld battery powered equipment.

The SX1509 is a general purpose GPIO expander board, able to parallel handle several inputs and outputs. It has ultra-low voltage capabilities of 1.2 to 3.6 V, ideal for battery powered equipment.

Reliable

use