Final Project - Introduction to Robotics

by Fabian Becker, Florian Remberger & Quang Thanh Lai

General Project Requirements

The aim of this project is for the robot to find its way around a given map and navigate to two of the three marked destinations by the shortest possible route. The map has a size of 3 x 3 metres and is divided into tiles of 25 cm. A target is reached when the robot is positioned at one of the adjacent target tiles, while the position of the target itself is perceived as an obstacle. After reaching a target, the robot has to navigate back to the start position. The starting position of the robot is downwards.

Additionally, our group implemented a live view of the current map, the calculated path and the current position of the robot on the device's display.

Member Contributions

Fabian came up with the ideas for path finding and converting the shortest path into driving instructions. He then implemented and optimised these concepts with the help of Thanh. He also came up with the idea of displaying this information on the robot's display and coded the first working prototype, which was later refined by all three group members. Florian was responsible for the complex logic of the driving function. Together with Thanh, he implemented and optimised the logic, which we'll explain in more detail later. Finally, Thanh helped with debugging and optimising various parts of the code. He also refactored a lot of code to improve the uniformity of our code.

Importing

Importing the map is quite simple. We don't need to read it directly from a text file. A const char* variable called mapString is used to store the map string. It is located in the main.c file. It must be changed manually if you want the robot to use a different map.

The 2D array mapStringMatrix is also declared in main.c and can be filled with the characters from mapString using a simple for-loop.

We need to iterate from i = 0 to i < 196, incrementing i after each loop. This is because the 14x14 map stored in the string contains 196 characters (14 * 14 = 196). For each character, we must calculate the correct row and column to store in the 2D array.

The row can be determined by dividing i by 14. Due to the datatypes of i (uint8_t) and 14 (int by default), this calculation is evaluated as an integer. The result is an integer between 0 and 13, which determines in which row of the array the character at position i in mapString will be stored.

The column is calculated by the operation i % 14. This will also return an integer between 0 and 13 that represents the column of the array.

Using both calculations, we can now combine them to get the following equation. mapStringMatrix[i/14][i%14] = mapString[i]

Path finding

For path finding, we used a simpler version of Dijstra's algorithm with lower time complexity, so that the robot can only move in four directions. The algorithm doesn't need to check every other node for a possible shorter path, but only the four adjacent tiles.

All the information for the pathfinding algorithm is contained in an array of the Structure tile. A tile contains the information required for one tile of the map, which is referred to as a node in this context. The structure tiles contains the following information:

- visited | has the tile been processed by the algorithm (bool)
- distance | the current shortest distance to the starting position (uint8_t)
- prev | the index of the previous tile on the current shortest path as 1D index (uint8_t)

In order to save memory, all indexes are stored as 1D indexes. We have written a function to convert these coordinates into 2D indexes and vice versa. Hence to this we can always use the index that's more convenient for the current calculation.

The other parts of the algorithm are pretty much equal to a "normal" Dijkstra. In order to explain other interesting logic in more detail, a detailed explanation of the algorithm is omitted.

Conversion to driving instructions

To find the shortest path, we first look for the neighbouring tile of the current destination with the shortest distance to the start. Then a route array of that particular distance is constructed. This array contains the coordinates of the tiles from the start (excluded) to the destination (included). This array is used to generate the driving instructions, with the forward function being executed only if a turn is required or if there are no more coordinates to process.

When the robot arrives at the destination, the makeSound() function is called and the robot turns 180 degrees.

For the way back, a similar array is constructed, now containing the coordinates of the start tile. The execution of the motion commands is done in the same way, the only difference being that the array is now processed from the back to the front.

As the robot arrives at the starting index, it is now turned, based of its celestial direction, in a way that the robot is facing south again, in order to be ready for driving to the next target.

Driving Controls

function driveTiles(uint8_t tiles)

This function is located in the motion.c file. It makes the robot move forward in a straight line. The input parameter tiles represents the number of tiles (each tile has a length of 25 cm) that the robot should move forward. Using a 14x14 map, the tiles datatype uint8_t will be sufficient because we cannot move more than 12 tiles before hitting a wall. The second parameter "anfahren" (=approach) is of boolean type and determines whether the robot should approach the last point more slowly

without stopping abruptly at the end. It also changes the value of the buffer variable, because we are stopping the motors at a much lower power, so the robot will not travel as far as if we were stopping at a higher speed.

Calculating the distance to travel is the first thing this function does. This calculation is quite simple, because we only have to multiply the tiles by 25 (cm) to get the total distance we want to move. We also multiply the already calculated distance by 1000 because we do not want to use floating point values.

After calculating the distance, we declare nine variables. We will use them to calculate the difference in degrees.
degL and degR display the current measured degrees. prevDegL and prevDegR store the last measured degree values. motorL and motorR store the motor power to be applied to each of the two motors. They are both initialised to 50. diffL and diffR store the absolute difference between prevDegL/ prevDegR and degL/ degR. The offSet variable determines how large the difference between diffL and diffR must be before the motor power is increased to prevent the robot from drifting in one direction.

Now we get the current degree of the wheels. We store this value in the prev_deg variable.

The general logic of this drive function is: First we start the motors and then we check how much distance we have traveled since the last check. From the total distance we have to travel, we subtract the distance traveled. We stop the motors and set Motor_stop_float if the total distance is less than the buffer value (3000 which represents 3cm if anfahren paramter is false, else buffer equals zero).

At each iteration of the while loop, we check whether we already need to slow down the robot because we want it to approach a table. Logically, we first need to check that the 'approach' parameter is true. Then we want to check that the distance for both sides is less than 50000, so each wheel has to move 25000 more, which is 25cm and therefore 1 tile. The last thing that has to be true so that we can decrement the motor force for both sides by two is that the motor force is not already less than 20. If we did not check this, the motor force could possibly be decremented to 0, which would lead to an infite while loop because the robot would stop moving, so the distance travelled will never be less than or equal to the buffer.

We now start the motors with the motor force of the current value of motorL and motorR. After starting the motors we use the Motor-Tacho_GetCounter function which gives us the current value of the motor degree. Now that we have the new degree value and have stored it in the deg variable, we need to calculate the distance that the robot has moved. This calculation is quite complex. So I want to break it down into smaller steps:

1. **Calculate the difference between the old degree value and the new degree value** . This can be done by subtracting the old value stored in prev_deg from the current degree value stored in deg. In our code we use the getAbsDiff function which calculates the absolute difference so we do not get any negative numbers.
2. **Multiply** the number we just calculated **by 2**. The Motor_Tacho_GetCounter function will return 1 for every two degrees the motor has rotated.
3. **Divide it by 360.0**. 360 degrees would be a full turn of the motor and therefore a full turn of the wheel. What we want to calculate here is how many times the wheel has turned. The .0 guarantees, that the result will be of type double so we do not lose precision.
4. **Multiply by CIRCUMFERENCE**. CIRCUMFERENCE is the circumference of the wheel. After this step, we now know excactly how many centimeters the robot has moved since the last check.
5. **Multiply by 1000**. If we did not multiply by 1000, we would lose the decimal places because we are subtracting a double from an int. Because of this step we also had to multiply by 1000 when we first calculated the total distance.

After calculating the distance we can simply subtract it from the total distance and set prev_deg to deg.

We now check wether one of the wheels has rotated more than the other by comparing the diffL and diffR variables. If one of them is greater than the other, plus our previously specified offSet, we increase the motor force for the motor on the side where the wheel has rotated significantly less. The last thing inside the while loop has to be a delay. So we will not get the same degree value because the wheels will move a bit before we check again.

This procedure is repeated until the distance is less than or equal to 2cm. If this is the case, the motors will stop and the function will end.

function turn(uint8_t dir)

The motion.c file also contains the funtion turn. It makes the robot turn left or right according to the input parameter dir. If dir is equal to 1, the robot will turn left and if dir is equal to 0, the robot will turn right. We need to declare several variables that will be used later to store the distance to be covered by the left and right motors (distanceL, distanceR). To assign these variables we have to calculate the distance that both wheels have to turn. We have measured, that the distance between the two wheels is 8 cm. We also know, that the robot has to turn 90 degrees. Using this information we can approximate the distance one wheel has to travel by calculating the circumference for a circle with a diameter of 8 cm. Then we have to multiply the calculated value by 1/4 because the wheels only have to travel 1/4 of the circle.

1/4 * (8 * pi) = 2 * pi

We want to avoid floating-point values so we multiply the result of the above equation by 1000.

After calculating the distance that both wheels have to travel, we declare variables that will store the current degree value of both motors as well as the previous degree values (prev_degL, prev_degR, degL, degR). Then we define the variables motorL and motorR with 21 for motorL and 20 for motorR (the left motor of our robot is slightly weaker), which represent the motor force for the left and right motors. Then we define the diffL, diffR and offSet variables, which serve exactly the same purpose as they did in the driveTiles function.

The next thing this function does is get the current motor degree for the left and right motors and assign that value to the prev_degL and prev_degR variables.

We then start both motors using the ternary operator to evaluate the direction each motor must move using the dir parameter.

The while loop makes sure that each motor travels the distance it has to travel. Therefore, the condition of the while loop will be evaluated as true as long as one of the distances (distanceL or distanceR) is greater than 0.

Inside the while loop we first check if the distanceL variable is less than or equal to 0. If so, we stop the left motor because we have already moved enough distance on that side. Else if the distanceL variable is greater than 0, we get the current motor degree and store that value in the degL variable. Now that we have the previous and current degree value we can calculate the exact distance the wheel has turned since the last check. This can be done in the following way:

1. **Calculate the amount the wheel has turned**: First we calculate the difference between prev_degL and degL. Then we multiply this value by 2 because the motor sensor returns 1 for every 2 degrees it turned. Dividing this result by 360 gives us the exact amount the wheel has turned. ((degL - prev_degL) * 2) / 360

2. **Multiply the result by the CIRCUMFERENCE of the wheel**: By multiplying the amount the wheel turned by the CIRCUMFERENCE of the wheel we get the exact distance it traveled.

3. **Multiply the result by 1000**: This must be done because we want to avoid floating point values.

After calculating the distance the wheel has turned we subtract that value from the total distance it has to turn.

The last operation of the else part is to assign the current degree value to the prev_degL variable so that we can do the same calculation steps in the next iteration.

The same procedure is used for the right motor. The following instructions are exactly the same as those used in the driveTiles function for fine-tuning motor power. We check whether one of the wheels has rotated more than the other by comparing the diffL and diffR variables. If one of them is greater than the other, plus our

previously specified offSet, we increase the motor force for the motor on the side where the wheel has rotated significantly less. The last thing inside the while loop has to be a delay. So we will not get the same degree value because the wheels will move a bit before we check again.

Displaying the progress

Once the map has been converted from the input string to a 2D array and the start and end points have been processed by the system, this information is printed to the display. This view is updated when the robot starts driving to its first target. Once the robot starts moving, the route is only updated when the robot makes a turn. This design choice was made to keep the display logic out of the driving functions. After each update, you can also see the robot's current position and the destination it's heading towards. The same behaviour can be observed either when the robot navigates back to its starting position, or when the device travels to the second target.
