COMP0128: Robotic Control Theory and Systems CW1

Group 2C: Jeffrey Li, Yuzhou Cheng, Bruce Jiang
zcabbli@ucl.ac.uk, yuzhou.cheng.23@ucl.ac.uk, yiheng.jiang.23@ucl.ac.uk

October 30, 2023

1. **Code Instruction:** Please run the file Q1/Sim_DifferentialDriveWithObstacles.m

   Solution: To calculate the radius of the robot wheels r, we first let the robot operate in a straight line along $y = 0$ using PID control towards the obstacle. We keep tracking the **front sensor** readings, $y(3)$ and $y(4)$. When the average of these values reaches two defined points st and en, as outlined in Q1/initQ1.m, We record the odometer readings $y(1)$ and $y(2)$ to compute the $tick$ change in both wheels, $\Delta tick1$ and $\Delta tick2$. We then take the average of these two to get $\Delta tick$. Since we can compute the physical distance between these two points (d_L) using laser readings, with ticks per wheel revolution (tpr) known as 64, the wheel revolutions (rev) through these points can then be given as the following:

   $$rev = \frac{\Delta tick}{tpr} \tag{1}$$

   The wheel radius r can then be given by:

   $$r = \frac{d_L}{2\pi * rev} \tag{2}$$

   We noted the following values:

   $$d_L = 0.5$$
   $$\Delta tick = 50$$

   To handle system noise and sensor deviation, 10 runs are conducted for the calculation (same for wheel distance) and the average value is kept as the final answer. Putting these values in and keeping two decimal numbers results in $r = 0.099678(m) \approx 0.10(m)$ on average throughout 10 runs.

   Now that we have r, the calculation of robot width lw can be done by instructing the robot to travel a full circle, so that lw would be the diameter of the circle drawn. We order the robot to walk in a circular trajectory with the aid of the **front sensor** readings and obtain the change in odometer readings from one of the wheels ΔW_l (in this case we used the left wheel readings). Since tpr and r are known, the wheel distance lw can be given as:

   $$lw = r * \frac{\Delta W_l}{tpr} \tag{3}$$

   By keeping two decimal places, we finally get $lw = 0.24867(m) \approx 0.25(m)$ as the average wheel distance for 10 runs.

2. **Code Instruction:** Please run the file Q2/Sim_DifferentialDriveWithObstacles.m

   Solution: The problem can be completed in 6 stages using a **single PID** while changing to different references appropriately:

   1. Walk in a straight line up to the obstacle
   2. Turning the robot clockwise to face parallel to the obstacle's wall
   3. Orbiting around the obstacle for 2 complete cycles
   4. Rotate back to the initial orientation
   5. Walk backwards in a straight line up to the starting position
   6. Mild amend the robot to initial orientation again

   Throughout the run, the robot keeps calculating its current coordinates (px, py) and its orientation θ using odometer readings and forward kinematics.

   For stage 1, the reference is set to be 0 angle and the error is the difference between left and right odometer readings, similar to what we achieved in Tutorial 2. The robot would walk straight while monitoring the front sensor readings until reaches 0.5 distance away from the obstacle. We denote the straight segment as wL.

   For stage 2, the robot would turn clockwise very slowly at its current position with u being set to $u = [0.5; -0.5]$. y_5 reads the top left sensor measurement and y_6 reads that of the bottom left sensor. The robot would keep turning clockwise while y_5 has not recorded readings less than 0.5 and the readings between y_5 and y_6 are different. This ensures that the direction in which the robot is facing will be in parallel with the side of the obstacle. We recorded the number of ticks after the rotation $after\_tick$ of the left odometer, which we used to calculate the change in ticks that is then used in stage 3.

For stage 3, the reference ensures that the readings of y_5 do not deviate from 0.5 such that the robot will always stay 0.5 away from the obstacle while circulating it, and that the orientation of the robot remains parallel to it. As told that the shape of the obstacle would not change, the number of ticks required for the robot to complete 2 cycles wouldn't change, which we obtained to be 1814 ticks. While circulating, we monitor the tick counts of the left odometer, if it becomes bigger or equal to $(1814 + after\_tick)$, then 2 cycles have been completed and the robot should move onto the next stage, otherwise keep circulating.

For stage 4, the robot would rotate anti-clockwise back to the initial orientation, achieved by minimising the error between θ and the starting orientation. After rotating to the right orientation, in stage 5, the robot would then walk backwards along the line segment wL and stop when the starting position has been reached.

To account for accumulated errors in odometer readings, we set a milestone coordinate inside the line segment wL rather than setting the target coordinate straight to $(0, 0)$, which has improved the quality of the returning trajectory considerably. Finally, when the robot reaches its initial point, in state 6, we order it to mildly amend (to the initial angle) its orientation error due to the shaking in the backward process. Figure 1 shows the resulting trajectory of Q2 by our method, which is quite satisfactory.

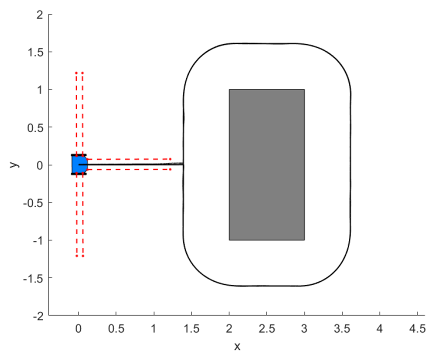
Figure 1: The trajectory of Q2

In the case of the obstacle being in a different position and orientation, our method would also work perfectly as we keep tracking the odometer readings to calculate the position and angle. When the robot completes its two full laps as stated above, it turns around until it is directed to the initial orientation. Then the robot goes backwards to (0,0) through its odometer.

We test our method for different obstacle orientations and positions by relatively moving and orientating the initial robot. So in this case, the robot's goal is to come back to the original point and orient initially. We could notice in Figure2 that indeed the robot succeeded in achieving the goals with positions and orientations $(0.2, 0.2, \frac{\pi}{16})$ and $(-0.15, -0.15, -\frac{\pi}{15})$.

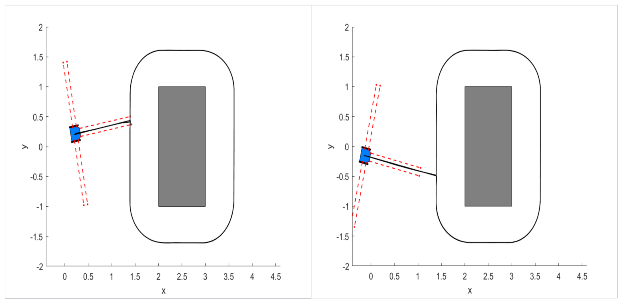
Figure 2: Obstacles set in different orientations and positions

3. **Code Instruction:** Please run the file Q3/Sim_DifferentialDriveWithObstacles.m

Solution: While maintaining the total 100 reference points we set, the problem can be completed using a **single PID**.

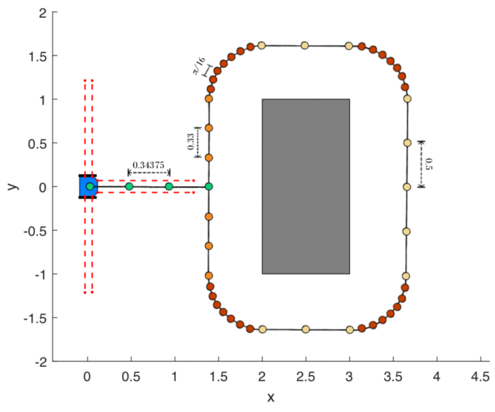

Figure 3: Visualization of the list of target coordinates

Throughout the run, the robot keeps calculating its current coordinates (px, py) and its orientation θ using odometer readings and **forward kinematics** so that it can be controlled to arrive at the indicated target coordinates. Since the location and orientation of the obstacle stay consistent, we calculated and specified a list of target coordinates that the robot should travel to such that it would travel around the obstacles (while keeping an edge distance of 0.5) for two cycles and return to $(0, 0)$.
This list of reference points is stored in the list vari-

able lst in init.m file. The first 3 points are used to guide the robot to reach the turning point, which is $(0.5 + 0.125)$m to the left of the obstacle. When the robot reaches point $(1.375, -1)$, we use the circumferential angle formula to divide the desired arc trail into 8 segments (each corresponds to $\frac{\pi}{16}$ radians) using 7 reference points, the same for the rest 3 corners of the obstacle. When travelling between these arc trails, the robot is guided with 5/3 reference points for the left/right side of the obstacle (See Figure 3). Finally, when the robot came back to its initial position, we used PID control to adjust the robot orientation back to 0 radians.

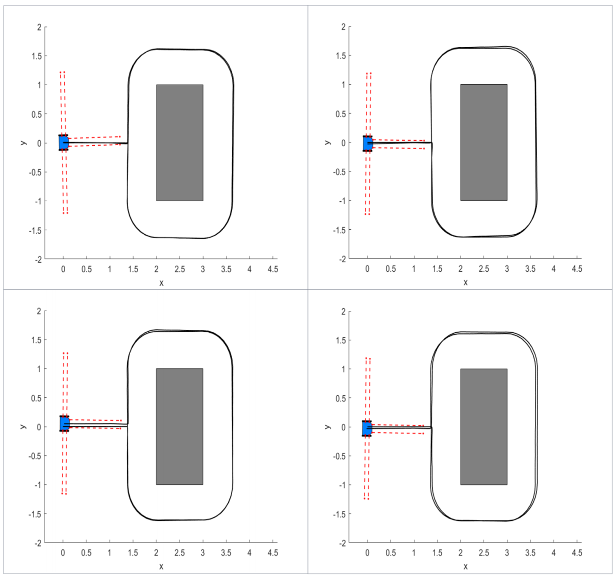
Figure 4: The trajectories of Q3 on various runs

Comparison of Q2 and Q3: 1) The trajectory drawn and the final orientation do not look as perfect as that in Q2 on all runs. Moreover, the trajectory becomes more and more **inaccurate** during the movement of the robot. These are due to the accumulating error in odometer readings. We ran Q3 10 times and showed the results of the first, second, sixth, and tenth quality rankings (See Figure 4). **2)** However, it is worth noting that the robot runs more smoothly while travelling in the arc trails as well as along all 4 sides of the obstacle compared to Q2. This is because, in Q2, laser sensors are used for maintaining the robot's orbit, meaning that the robot has to adjust itself constantly to stay on the desired trajectory. As measure error exists in laser readings, the robot tends to be **shaking** more compared to Q3. In Q3, specific reference points are used such that the robot knows exactly where to go at different stages.

4. **Code Instruction:** Please run the file Q4/Sim_DifferentialDriveWithObstacles.m

Solution: We are given that:

$$\begin{aligned} G(s) = \frac{S_m(s)}{V_m(s)} &= \frac{K_s}{\tau_s s + 1} \\ \implies S_m(s) &= \frac{K_s}{\tau_s s + 1} V_m(s) \end{aligned} \tag{4}$$

Assuming $v_m(t) = c$ is a constant, then $V_m(s) = \frac{c}{s}$, thus:

$$S_m(s) = \frac{K_s}{\tau_s s + 1} \frac{c}{s} \tag{5}$$

The Limit Value Theorem states that:

$$\begin{aligned} \lim_{t\to\infty} s_m(t) &= \lim_{s\to 0} s S_m(s) \\ &= \lim_{s\to 0} s \frac{K_s}{\tau_s s + 1} \frac{c}{s} \\ &= \lim_{s\to 0} \frac{cK_s}{\tau_s s + 1} \\ &= cK_s \end{aligned} \tag{6}$$

As specified in the code, we supplied a constant voltage to both wheels and calculated the speed (denoted as s_f) at which the robot (nearly) converged using the formula $s_f = r\omega$. Here $\omega = \frac{\Delta tick * 2\pi}{nTicks * \Delta t}$, $nTicks = 64$ and $\Delta tick$ are the average change in ticks across both wheels after Δt as the robot reaches terminal speed. Please notice that since we are using the average tick change, we do not need to maintain the robot in a straight line. We experimented with voltage values $[1, 2, 3, 4, 5, 6]$, and on average, $K_s = \frac{s_f}{c} = 0.045046 \approx 0.045$ to 3 decimal places.

5. **Code Instruction:** All code files are stored in Q5 folder. Open pidTuning.m to see code for PD parameter tuning and model.slx for Simulink model.

Solution: The system dynamics are approximated as:

$$\begin{aligned} \dot{d}(t) &\approx s_m \phi(t) \\ \implies \ddot{d}(t) &\approx s_m \dot{\phi}(t) \end{aligned} \tag{7}$$

After transforming the equation into the Laplace domain we get:

$$s^2 D(s) - sd(0) - d'(0) = s_m \dot{\Phi}(s) \tag{8}$$

With $\dot{\phi}(t)$ as input, $d(t)$ as output, $s_m = 0.2$, and zero initial conditions we obtain the following transfer function:

$$Y(s) = \frac{D(s)}{\dot{\Phi}(s)} = \frac{0.2}{s^2} \tag{9}$$

The transfer function of the system as a whole would be the product of the two transfer functions:

$$G_m(s)Y(s) = \frac{0.25}{0.1s + 1} * \frac{0.2}{s^2} = \frac{0.05}{0.1s^3 + s^2} \tag{10}$$

Now let $p(t)$ be the controlling signal (output), $e(t)$ be the error signal (input) of the PD controller, and K_p,K_d being the PD parameters respectively:

$$\begin{aligned} p(t) &= K_d\dot{e}(t) + K_p e(t) \\ P(s) &= K_d s E(s) + K_p E(s) \\ &= E(s)(K_d s + K_p) \end{aligned} \tag{11}$$

Thus the transfer function of the controller $C(s)$ is as follows:

$$C(s) = \frac{P(s)}{E(s)} = K_d s + K_p \tag{12}$$

To enforce the relationship between the proportional and derivative parameters of the PD-controller, we have: $C(s) = 10K_p s + K_p$, thus the transfer function of the whole open loop system is:

$$T(s) = C(s)G(s) \tag{13}$$

Using $T(s)$ and the **margin** function we did an exhaustive incremental search over possible values of K_p that maximises its phase margin. This is achieved by shrinking the range of values to search through and the step size of the search manually, which we found to be $K_p = 2.0$ with phase margin ≈ 78.6 degrees. We have verified this result with **sisotool**(Figure 5).

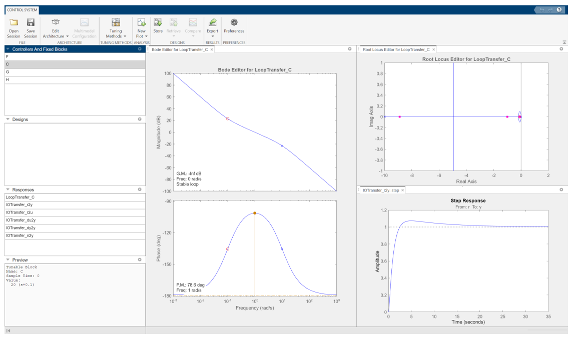

Figure 5: Result examination by sisotool

Lastly, a simulation of the closed-loop system has been created using Simulink in Q5/model.slx (See Figures 6 6 8 9)

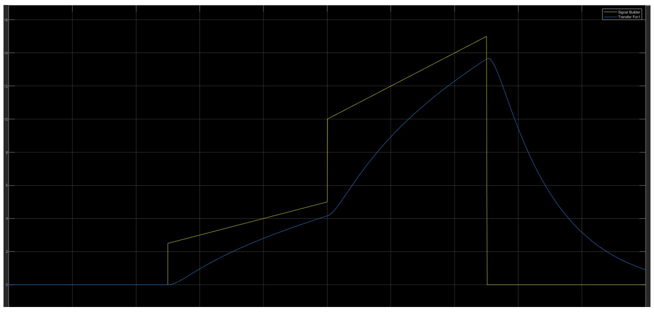

Figure 6: Yellow being $r_d(t)$ and the blue being $d(t)$

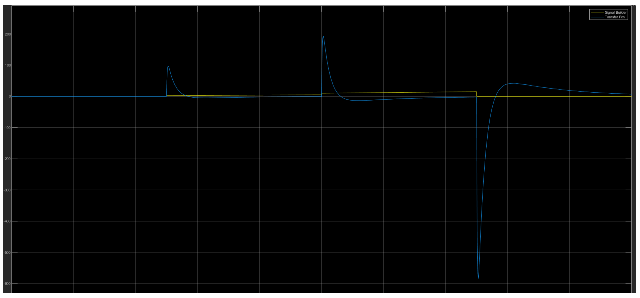

Figure 7: Yellow being $r_d(t)$ and the blue being $\dot{\phi}(t)$

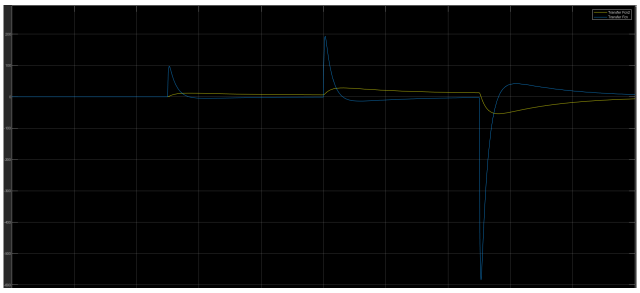

Figure 8: Plot with $\phi(t)$(Yellow) and $\dot{\phi}(t)$(Blue) which we thought was quite interesting

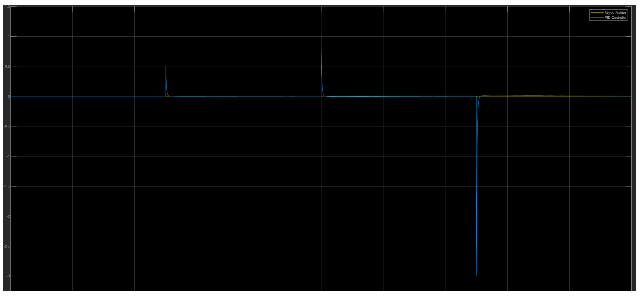

Figure 9: Yellow being $r_d(t)$ and the blue being $v_\Delta(t)$

6. Recall $C(s) = 10K_p s + K_p$, to discretise this PD-controller, we used **Tustin** method with $\Delta t = 0.02s$ as follows:

$$s \approx (\frac{2}{\Delta t})(\frac{z-1}{z+1})$$

$$\begin{aligned} C(z) &= 10K_p(\frac{2}{\Delta t})(\frac{z-1}{z+1}) + K_p \\ &= 1000K_p(\frac{z-1}{z+1}) + K_p \\ &= \frac{1000K_p z + K_p z - 1000K_p + K_p}{z+1} \end{aligned} \tag{14}$$

$K_p = 2$, thus $C(z) = \frac{2002z-1998}{z+1}$. We now convert it into an IIR filter:

$$C(z) = \frac{P(z)}{E(z)} = \frac{2002z - 1998}{z+1} = \frac{2002 - 1998z^{-1}}{1 + z^{-1}} \tag{15}$$

Rearranging the equation, we get:

$$P(z)(1+z^{-1}) = (2002 - 1998z^{-1})E(z) \tag{16}$$

$$\begin{aligned}\implies p[k] + p[k-1] &= 2002e[k] - 1998e[k-1] \\ p[k] &= 2002e[k] - 1998e[k-1] \\ &\quad -p[k-1] \end{aligned} \tag{17}$$

To compute distance $d(t)$ from the range sensor measurements, assuming we can compute the relationship between the range sensor measurements and actual distance and we know the width of the robot l_w, then we could do the following. Let the actual distance measured by the sensor be $q(t)$, then the following relationship holds (See Figure 10):

$$\cos(\phi(t)) = \frac{d(t)}{0.5l_w + q(t)} \tag{18}$$
$$\implies d(t) = \cos(\phi(t))(0.5l_w + q(t))$$

For instance, if we are in the scenario of a robot in Part (1), then $q = (y_8 + y_9)/2$ where y_8 and y_9 are the two right range sensor measurements.

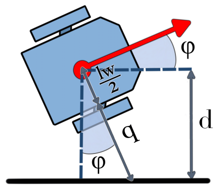

Figure 10: Geometric relationship of the scenario

The remaining task is to get $\phi(t)$. Notice that we have:

$$\frac{\dot{\Phi}(s)}{V_\Delta(s)} = \frac{0.25}{0.1s+1} \tag{19}$$
$$\implies 0.1\ddot{\phi}(t) + \dot{\phi}(t) = 0.25v_\Delta(t)$$

Since $V_\Delta(t)$ is **constant** between t_i to t_{i+1} for $\forall i$, we solve this ordinary differential equation piece-wisely:

$$\forall t \in [t_i, t_{i+1}), \phi(t) = C_1 + C_2e^{-10t} + 0.25v_\Delta(t_i)t \tag{20}$$

We first solve for $i = 0$ by substituting initial conditions:

$$\begin{cases} \phi(t_0) = C_1 + C_2e^{-10t_0} + 0.25v_\Delta(t_0)t_0 \\ \phi'(t_0) = -10C_2e^{-10t_0} + 0.25v_\Delta(t_0) \end{cases} \tag{21}$$

Using this we can solve for C_1 and C_2 as with zero initial conditions and $v_\Delta(t_0)$ is what we control:

$$t_0 = \phi(t_0) = \dot{\phi}(t_0) = 0$$

$$\begin{cases} 0 = C_1 + C_2 \\ 0 = -10C_2 + 0.25v_\Delta(t_0) \end{cases} \tag{22}$$

The ordinary equation ensures C_1 and C_2 would remain constant through the same coloured highlight $[t_i, t_{i+1})$ (See Figure 11). Since $\phi \in \mathbb{C}^1$ (continuously differentiable), we could obtain $\phi(t_1), \dot{\phi}(t_1)$ by Equation 23.

$$\forall t \in [t_0, t_1), \phi(t) = C_1 + C_2e^{-10t} + 0.25v_\Delta(t_0)t \tag{23}$$

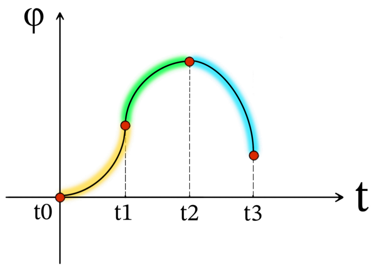

Figure 11: Demonstrating how ϕ varies over time

Using this reasoning we can solve C_1 and C_2 for the following time step where we know $\phi(t_1), \dot{\phi}(t_1)$ and that $v_\Delta(t_1)$ is what we control which can then be used to obtain $\phi(t_2), \dot{\phi}(t_2)$:

$$\begin{cases} \phi(t_1) = C_1 + C_2e^{-10t_1} + 0.25v_\Delta(t_1)t_1 \\ \phi'(t_1) = -10C_2e^{-10t_1} + 0.25v_\Delta(t_1) \end{cases} \tag{24}$$

Same process can be carried out to calculate $(\phi(t_3), \dot{\phi}(t_3))$,$(\phi(t_4), \dot{\phi}(t_4))$,$(\phi(t_5), \dot{\phi}(t_5))$ and so on ... knowing that step function v_Δ is what we can control and thus we could solve for all $\phi(t)$. Substituting this back to Equation 18 so we can solve for $d(t)$.

Finally, the reasons why the controller would behave differently are that in Part 1, the simulation is done using a user-defined way of simulation, meaning the simulation environment can be very different to Simulink. **1)** The most important example is that in Simulink, no restrictions on the voltage value have been applied to the system, where its values are at the scale of 10^4 (See Figure 9), which also explains the rapid changes in $\dot{\phi}$ (See Figure 7). Comparatively, in the Part 1 simulator, voltage values have been capped within the range [-6,6], thus limiting such abrupt controls. **2)** Another example is the simulation environment in Part 1 has some user-defined noises to sensor measurements whereas in Simulink there aren't any or could be vastly different causing the sensor measurements to differ hence resulting in different controller responses. **3)** Other unknown internal parameters in Part (1), as said, that can change at each simulation run influence the controller's behaviour.