Homework 2

Fiorella Maria Romano
FIELD AND SERVICE ROBOTICS (FSR)

April 22, 2025

Exercise 1

First of all, the final model-configuration has been chosen as suggested:

$$q_f = rand(1,3)';$$

$$q_f = q_f / norm(q_f);$$

We're going to consider: $q_f = [0.5293, \ \ 0.4816, \ \ 0.6985]^T$.
The planned path is defined by cubic polynomials as follows:

$$s(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$
$$\dot{s}(t) = a_1 + 2a_2 t + 3a_3 t^2$$
$$\ddot{s}(t) = 2a_2 + 6a_3 t$$

The polynomial coefficients are computed by imposing boundary conditions on initial and final positions, as well as zero initial and final velocity. Solving the resulting linear system gives the coefficients of the polynomials.

Condition	**Equation**
Initial position	$s(0) = s_i \Rightarrow a_0 = 0$
Initial velocity	$\dot{s}(0) = 0 \Rightarrow a_1 = 0$
Final position	$s(T) = s_f \Rightarrow a_2 T^2 + a_3 T^3 = 1$
Final velocity	$\dot{s}(T) = 0 \Rightarrow 2a_2 T + 3a_3 T^2 = 0$

Table 1: Boundary conditions and equations for cubic polynomial coefficients

We get:

$$a_0 = 0, \ a_1 = 0, \ a_2 = \frac{3}{T^2}, \ a_3 = -\frac{2}{T^3}$$

Then, by imposing $T = 5s$ and a sampling time $T_s = 0.01$, it's possible to compute the cubic polynomial.

So far, we're sure that the unicycle will start from q_i and will end in q_f. The conditions on θ are still missing. Therefore, by imposing $k_i = k_f = 2$ it's possible to compute the coefficients $\alpha_x, \alpha_y, \beta_x, \beta_y$ as follows:

$$\alpha_x = Kcos(\theta_f) - 3x_f \qquad \alpha_y = Ksin(\theta_f) - 3y_f$$

$$\beta_x = Kcos(\theta_i) + 3x_i \qquad \beta_x = Ksin(\theta_i) + 3y_i$$

At this point, the desired trajectories and all their time derivatives can be computed:

$$x(s) = s^3 + \alpha_x s^2(s-1) + \beta_x s(s-1)^2$$

$$y(s) = s^3 + \alpha_y s^2(s-1) + \beta_y s(s-1)^2$$

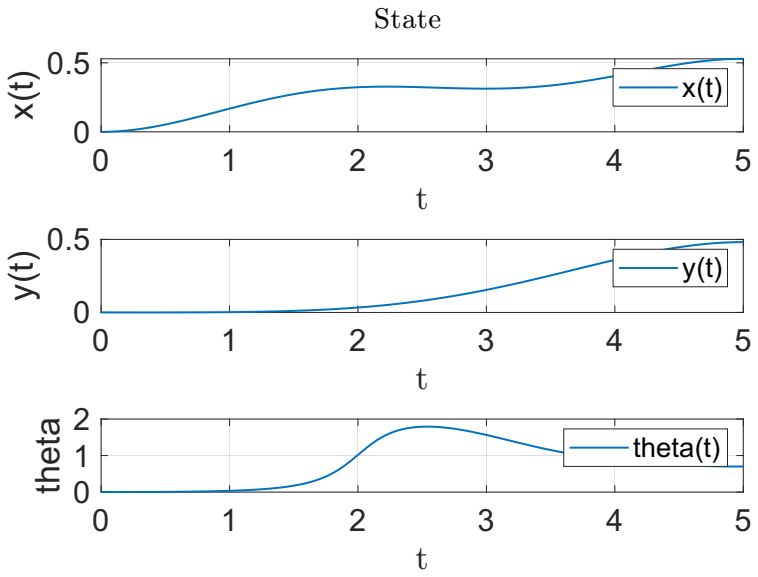


Figure 1: State Trajectories

To satisfy the unicycle kinematic constraints:

$$|v(t)| \leq 0.5 \text{ m/s}, \qquad |\omega(t)| \leq 2 \text{ rad/s}$$

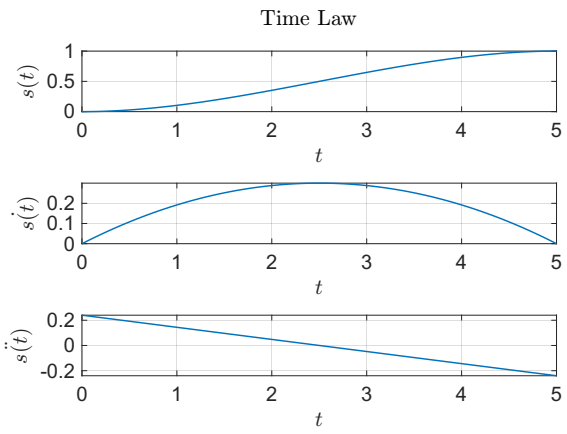
the time law $s(t)$ has been rescaled. The resulting plots of $v(t)$, and $w(t)$ in the following Figures confirm that the velocity and acceleration profiles respect the imposed constraints. The procedure begins with an initial estimate $T_f = 5$. A cubic time scaling function $s(t)$ and its derivatives $\dot{s}(t)$, $\ddot{s}(t)$ are computed using the `cubic_polynomial` function. These are then used to generate the trajectory and corresponding control inputs $v(t)$ and $\omega(t)$. If the resulting velocity or angular velocity exceeds the given limits, the total execution time T_f is iteratively increased by 0.1 seconds at each step. This effectively performs a time-dilation procedure, reducing the magnitude of the derivatives and allowing the trajectory to comply with the constraints. After each update, the time law and resulting trajectory are recalculated. This process continues until the velocity constraints are satisfied or a maximum number of iterations (150) is reached. The final value of T_f obtained at the end of the loop

represents the minimum execution time for which the trajectory is feasible under the given constraints. It is interesting to note that the initial guess $T_f = 5$ s had to be increased by approximately 48% to meet the constraints, resulting in a final feasible duration of $T_f = 7.4$ s. This highlights the trade-off between trajectory duration and admissibility of control inputs.

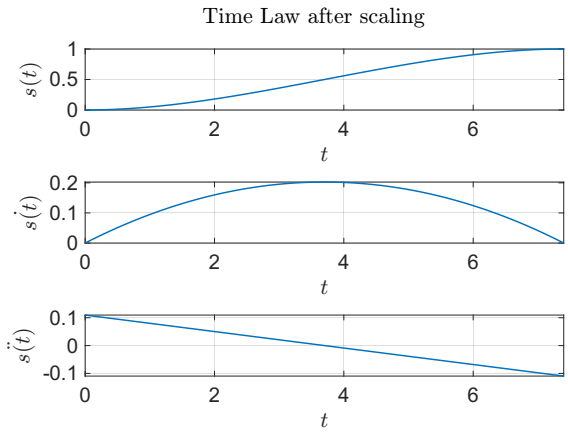
```
Velocity bounds are NOT satisfied.
Iteration 1: velocity limits NOT satisfied. Increasing Tf...
Tf = 5.1
Iteration 2: velocity limits NOT satisfied. Increasing Tf...
Tf = 5.2
Iteration 3: velocity limits NOT satisfied. Increasing Tf...
Tf = 5.3
Iteration 4: velocity limits NOT satisfied. Increasing Tf...
Tf = 5.4
Iteration 5: velocity limits NOT satisfied. Increasing Tf...
Tf = 5.5
Iteration 6: velocity limits NOT satisfied. Increasing Tf...
Tf = 5.6
Iteration 7: velocity limits NOT satisfied. Increasing Tf...
Tf = 5.7
Iteration 8: velocity limits NOT satisfied. Increasing Tf...
Tf = 5.8
Iteration 9: velocity limits NOT satisfied. Increasing Tf...
Tf = 5.9
Iteration 10: velocity limits NOT satisfied. Increasing Tf...
Tf = 6
Iteration 11: velocity limits NOT satisfied. Increasing Tf...
Tf = 6.1
Iteration 12: velocity limits NOT satisfied. Increasing Tf...
Tf = 6.2
Iteration 13: velocity limits NOT satisfied. Increasing Tf...
Tf = 6.3
Iteration 14: velocity limits NOT satisfied. Increasing Tf...
Tf = 6.4
Iteration 15: velocity limits NOT satisfied. Increasing Tf...
Tf = 6.5
Iteration 16: velocity limits NOT satisfied. Increasing Tf...
Tf = 6.6
Iteration 17: velocity limits NOT satisfied. Increasing Tf...
Tf = 6.7
Iteration 18: velocity limits NOT satisfied. Increasing Tf...
Tf = 6.8
Iteration 19: velocity limits NOT satisfied. Increasing Tf...
Tf = 6.9
Iteration 20: velocity limits NOT satisfied. Increasing Tf...
Tf = 7
Iteration 21: velocity limits NOT satisfied. Increasing Tf...
Tf = 7.1
Iteration 22: velocity limits NOT satisfied. Increasing Tf...
Tf = 7.2
Iteration 23: velocity limits NOT satisfied. Increasing Tf...
Tf = 7.3
Iteration 24: velocity limits NOT satisfied. Increasing Tf...
Tf = 7.4
Velocity bounds satisfied.

Minimum time found: Tf = 7.4
```

Figure 2: Minimum Time needed to satisfy the vel and acc constraints

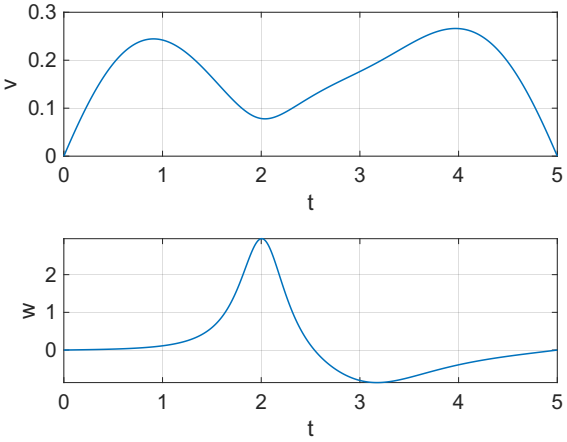
The final values of $v(t)$ and $\omega(t)$ remain strictly within the imposed limits: $\max|v(t)| \approx 0.17$ m/s and $\max|\omega(t)| \approx 2$ rad/s, confirming the effectiveness of the time-scaling procedure.



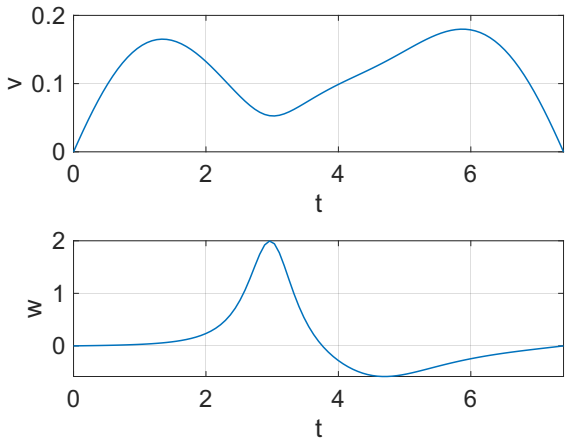
(a) Time Law



(b) Time Law after scaling procedure



(c) Control Inputs



(d) Control Inputs after scaling procedure

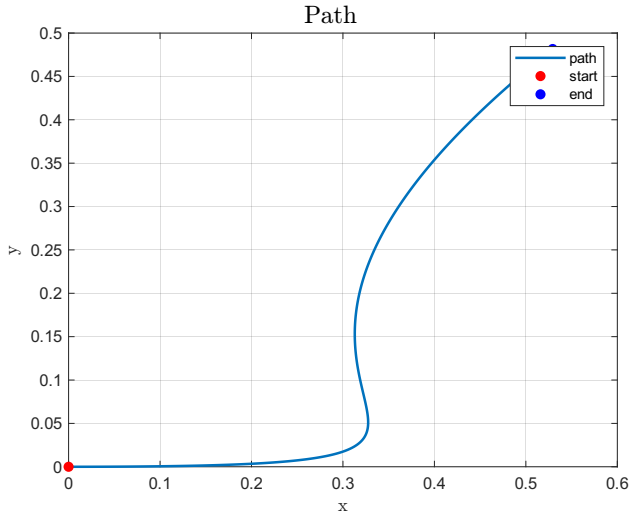


Figure 4: Unicycle Path

Exercise 2

In order to perform an Input/Output linearization, we need to consider another point B of coordinates (y_1, y_2) on the sagittal axis of the unicycle. This point has a distance $|b|$ from the center (x, y). This quantity must be different from zero. In particular, I have considered different values of b, in order to see what happens to the trajectory when it changes

$$b_1 = 0.05m, b_2 = 0.1m, b_3 = 0.2m, b_4 = 0.3m, b_5 = 0.5m, b_6 = 0.6m$$

In this kind of control, the dynamic system of the B point has to be considered r, so the system becomes:

$$\begin{bmatrix} \dot{y_1} \\ \dot{y_2} \end{bmatrix} = \begin{bmatrix} cos\theta & -bsin\theta \\ sin\theta & bcos\theta \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix} = T(\theta) \begin{bmatrix} v \\ w \end{bmatrix}$$

we get:

$$\begin{bmatrix} v \\ w \end{bmatrix} = T^{-1}(\theta) \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{1}$$

In this way:

$$\begin{bmatrix} \dot{y_1} \\ \dot{y_2} \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

Now, the error is referred to (y_1, y_2) and the virtual control inputs are designed as follows:

$$u_1 = \dot{y_{1d}} + k_1(y_{1,d} - y_1) \qquad u_2 = \dot{y_{2d}} + k_2(y_{2,d} - y_2)$$

with $k_1 = 10$, $k_2 = 10$ and where $y_{1,d}$, $y_{2,d}$ and its time derivatives are the desired values of position and velocity of the point B computed starting from the desired values of position and velocity of the center point of the unicycle. Once u_1 and u_2 are computed, they are converted into the real control inputs v, w through eq. 1 and are sent to the unicycle block.

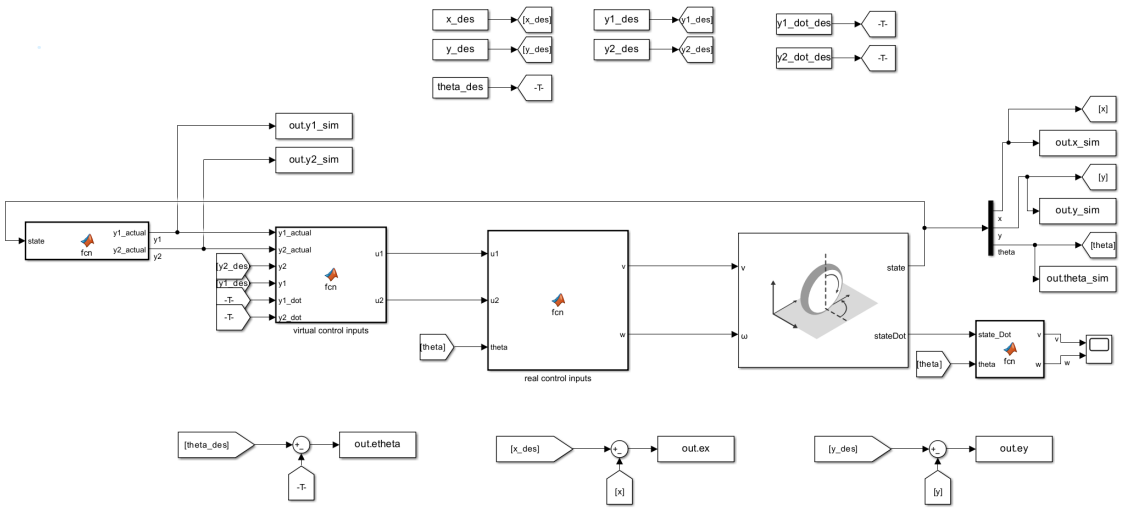


Figure 5: Simulink Scheme for I/O lin

The results are the following:

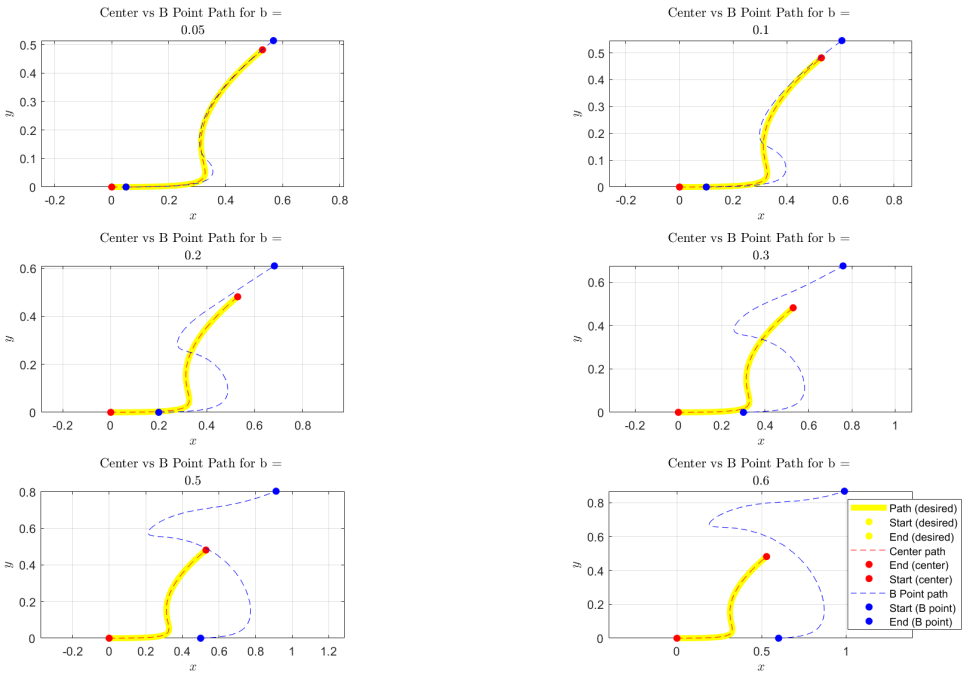


Figure 6: Center point and B point Paths for different values of b

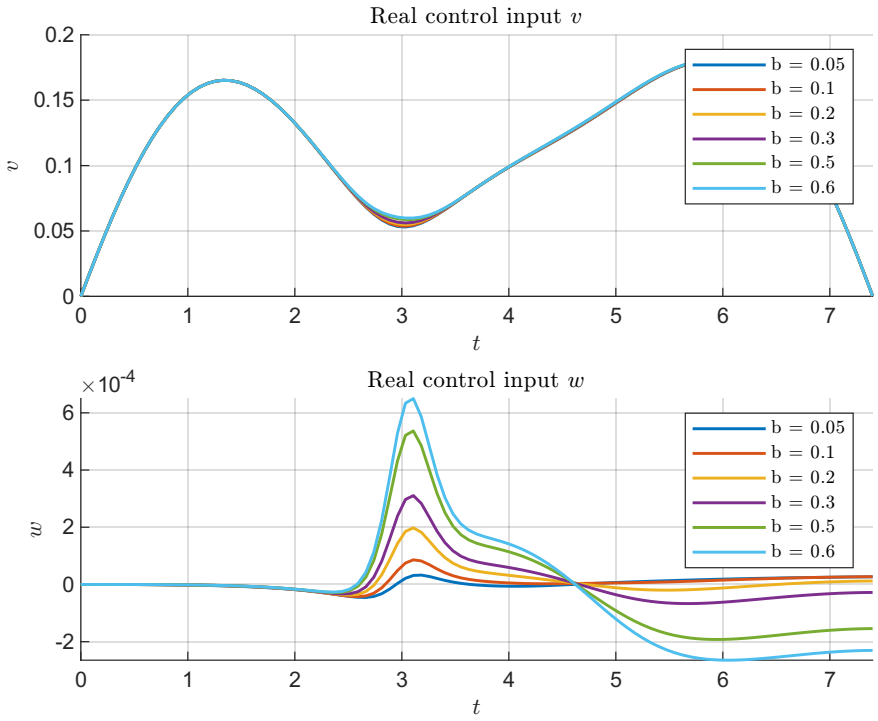


Figure 7: Comparison between Real Control Inputs for different values of b

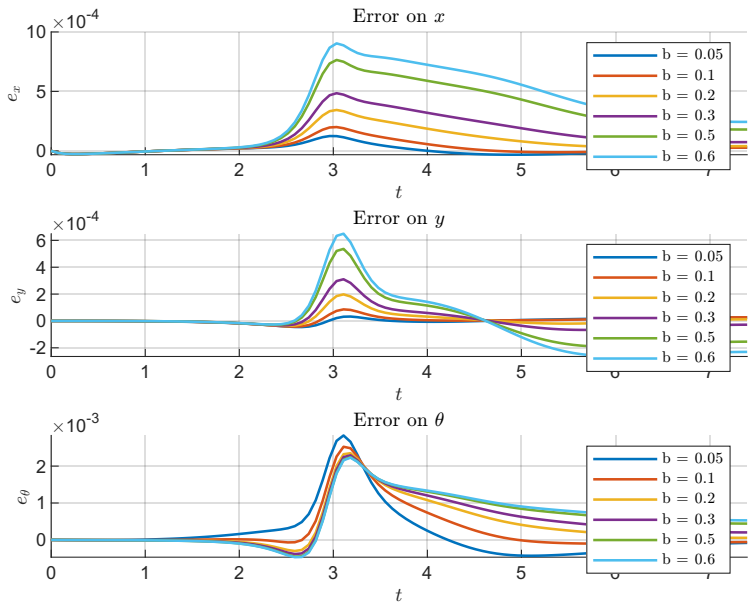


Figure 8: Comparison between errors for Center point and B point Paths for different values of b

From the simulations, it is observed that control based on input/output linearization enables the unicycle robot to accurately follow the desired trajectory in all cases analyzed. However, the effect of parameter b - that is, the distance between the robot center and the control point along the sagittal axis - has a significant impact on the shape of the **trajectory** traveled by the B point of the robot.

In particular, as the distance b increases , the trajectory followed by the center point deviates more from that of point B. This effect is expected, since point B is subject to a kinematic transformation dependent on b and by θ, which introduces a geometric delay in the position of the center.

In terms of the **tracking error**, there is a slight worsening as the value of b increases. This behavior can be explained by the greater dynamic complexity introduced by controlling a point away from the barycenter, resulting in greater sensitivity to angular variations and **control inputs**, as can be seen from Figure 7.

Exercise 3

The first thing I have done was defining the different radius values; in particular I've chosen:

$$r_1 = 0.1 \ m \ r_2 = 1 \ m$$

Then, I've computed the desired trajectory through the following equations:

$$x_d = \frac{r \ cos(\omega t)}{1 + sin^2(\omega t)} \tag{2}$$

$$y_d = \frac{r \ cos(\omega t) sin(\omega t)}{1 + sin^2(\omega t)} \tag{3}$$

where $\omega = \alpha + 1$ and $\alpha = 5$ in my case.

Then, as the trace says, I've implemented a code that, starting from $x_{d,0}$, $y_{d,0}$ computes two random initial conditions x_0, y_0 within $0.5m$.

```
x0_all(rr,1) = xd(1) + r*(2*rand - 1);
y0_all(rr,1) = yd(1) + r*(2*rand - 1);
distance = sqrt( (xd(1) - x0_all(rr,1))^2 + (yd(1) - y0_all(rr,1))^2 );
ok = check(distance);

while (ok == false)
    x0_all(rr,1) = xd(1) + 0.5*(2*rand - 1);
    y0_all(rr,1) = yd(1) + 0.5*(2*rand - 1);
    distance = sqrt( (xd(1) - x0_all(rr,1))^2 + (yd(1) - y0_all(rr,1))^2 );
    ok = check (distance);
end
theta0_all(rr,1) = atan2(x0_all(rr,1), y0_all(rr,1));
```

(a) Code for Initial Conditions

```
xd: 0.1     x0: 0.075521
yd: 0     y0: 0.023594
Distance for r = 0.1: 0.033999m
xd: 1     x0: 0.719
yd: 0     y0: -0.16359
Distance for r = 1: 0.32515m
```

(b) Chosen Initial Conditions

The initial conditions in Figure (b) have been selected, then also θ_0 is computed. The desired trajectories are showed in Figure 10, while the Simulink scheme in Figure 12.

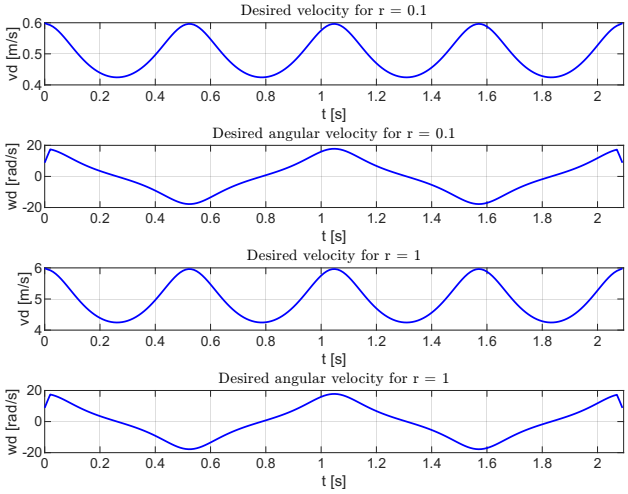


Figure 10: Desired linear and angular velocities

For both linear and nonlinear controllers, the error e is computed and then rotated through the rotation matrix $T(\theta)$ in order to write the error with respect to the current frame of the unicycle.

$$e(t) = q_d(t) - q(t)$$

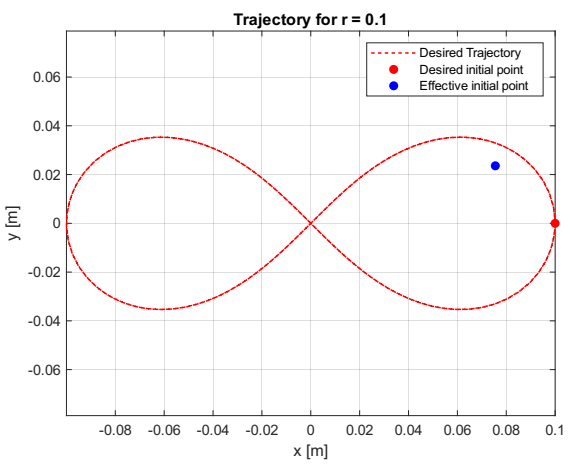
$$e(t) = \begin{bmatrix} e_1(t) \\ e_2(t) \\ e_3(t) \end{bmatrix} = T * \begin{bmatrix} x_d(t) - x(t) \\ y_d(t) - y(t) \\ \theta_d(t) - \theta(t) \end{bmatrix}$$

where:

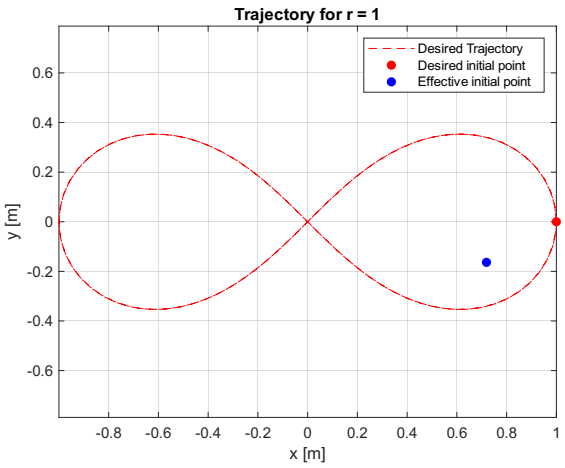
$$T = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

once that the virtual control inputs u_1, u_2 are computed, the real control inputs u, w are derived in the following way:

$$v(t) = v_d(t)cos(e_3(t)) - u_1(t) \qquad w(t) = w_d(t) - u_2(t)$$



(a) Desired path for r = 0.1 m



(b) Desired path for r = 1 m

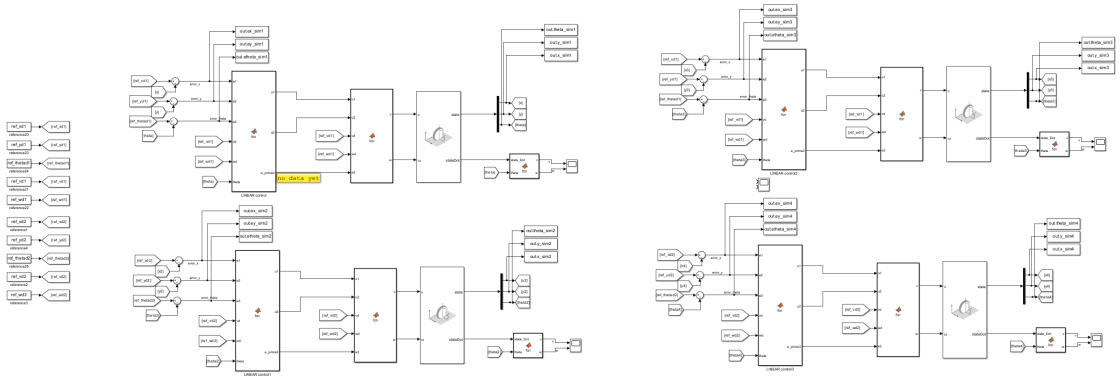


Figure 12: Simulink scheme

Linear Control

With the Linear control we get the following error dynamics:

$$\dot{e}(t) = \begin{bmatrix} 0 & w_d(t) & 0 \\ -w_d(t) & 0 & v_d(t) \\ 0 & 0 & 0 \end{bmatrix} e(t) + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix}$$

where:

$$u_1(t) = -k_1 e_1(t) \qquad u_2(t) = -k_2(t)e_2(t) - k_3 e_3(t)$$

and:

$$k_1 = 2\zeta a, \quad k_2 = \frac{a^2 - w_d}{v_d} \quad k_3 = k_1 \quad a = 70 \quad \zeta = 0.7$$

the results are shown in Figure 13, 14, 15, 16.

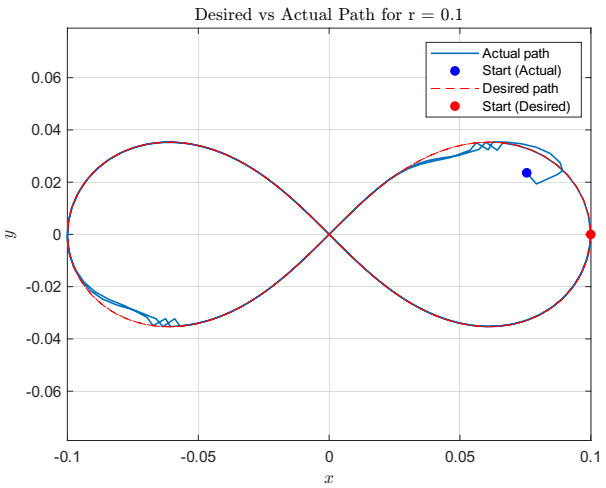


Figure 13: Desired vs Actual path for $r = 0.1$ m, Linear Control

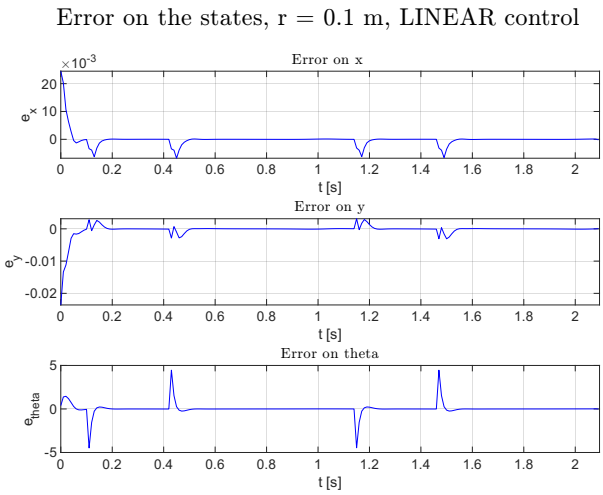


Figure 14: Error on the states for $r = 0.1$ m, Linear Control

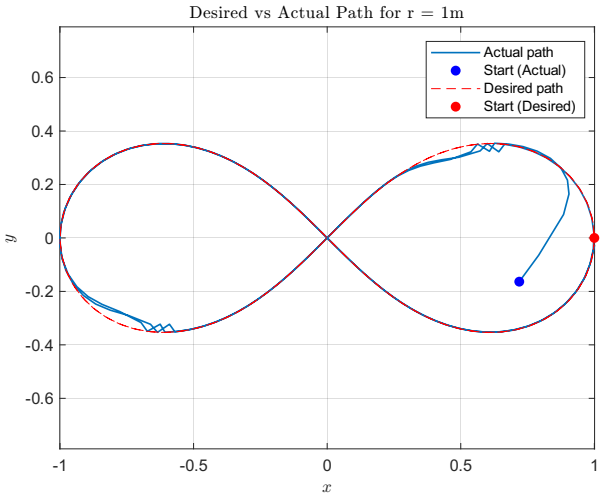


Figure 15: Desired vs Actual path for $r = 1$ m, Linear Control

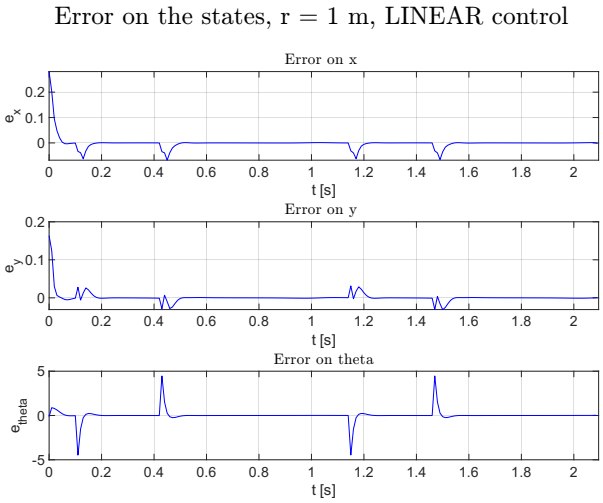


Figure 16: Error on the states for $r = 1$ m, Linear Control

(Almost) Non Linear Control

According to the (almost) non linear control, the error dynamics is the following:

$$\dot{e}(t) = \begin{bmatrix} 0 & w_d(t) & 0 \\ -w_d(t) & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} e(t) + \begin{bmatrix} 0 \\ sin(e_3(t)) \\ 0 \end{bmatrix} v_d(t) + \begin{bmatrix} 1 & -e_2(t) \\ 0 & e_1(t) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix}$$

where:

$$u_1(t) = -k_1(v_d(t), w_d(t))e_1(t)$$

$$u_2(t) = -k_2 v_d(t) sinc(e_3(t))e_2(t) - k_3(v_d(t), w_d(t))e_3(t)$$

and:

- r = 0.1m :

$$k_1 = \frac{a^2 - w_d}{v_d} \quad k_3 = k_1 \quad k_2 = 50 \quad a = 10$$

- r = 1m :

$$k_1 = \frac{a^2 - w_d}{v_d} \quad k_3 = k_1 \quad k_2 = 200 \quad a = 27$$

the results are shown in Figure 17, 18, 19, 20.

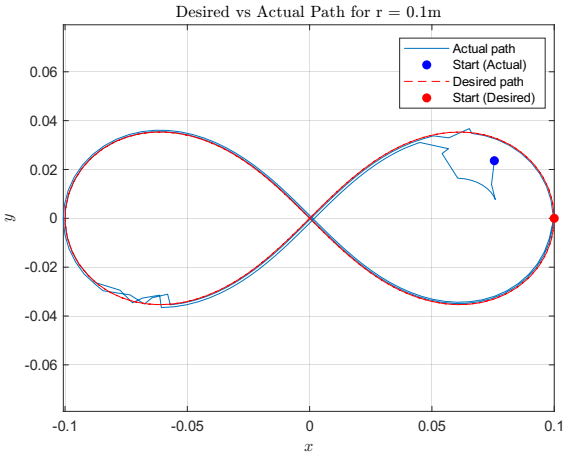


Figure 17: Desired vs Actual path for $r = 0.1$ m, Nonlinear Control

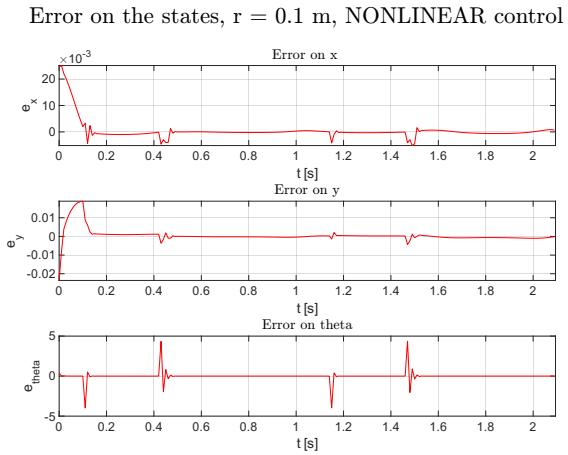


Figure 18: Error on the states for $r = 0.1$ m, Nonlinear Control

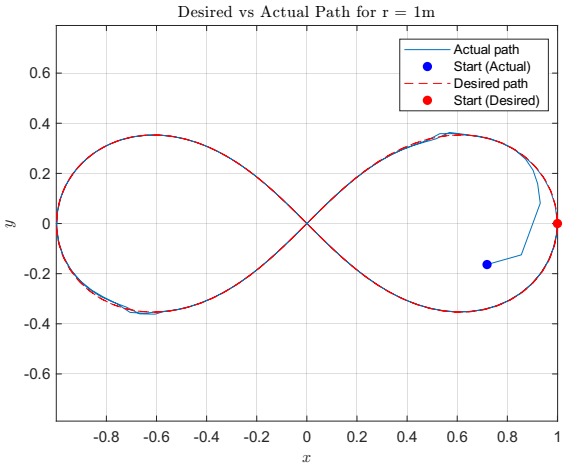


Figure 19: Desired vs Actual path for $r = 1$ m, Nonlinear Control

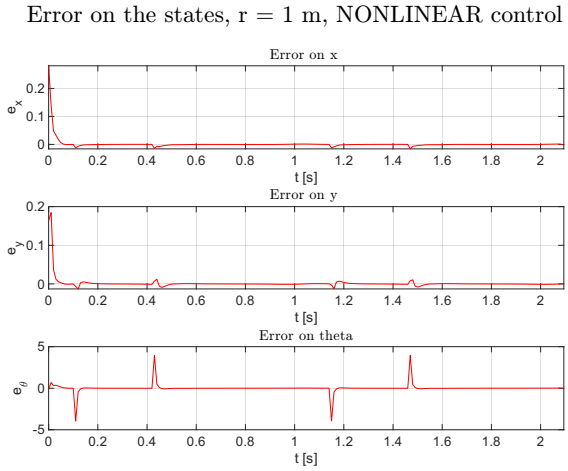
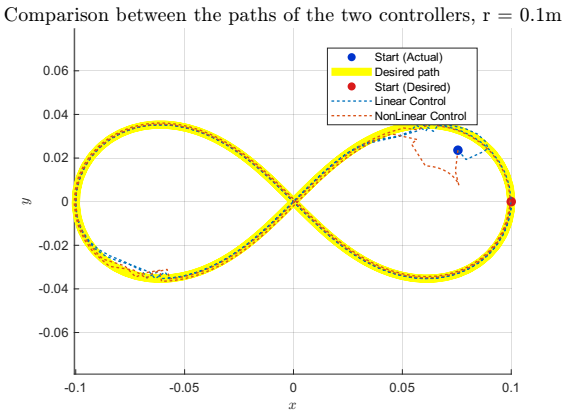


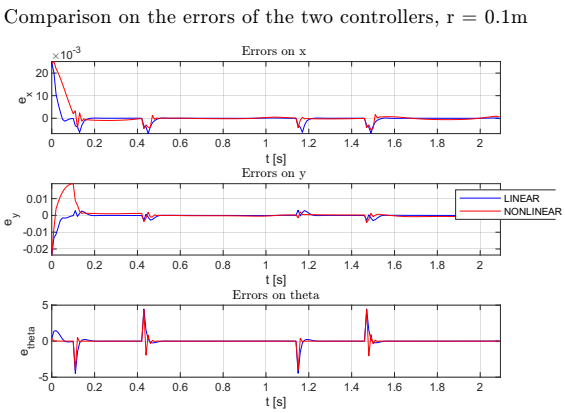
Figure 20: Error on the states for $r = 1$ m, Nonlinear Control

Comparison

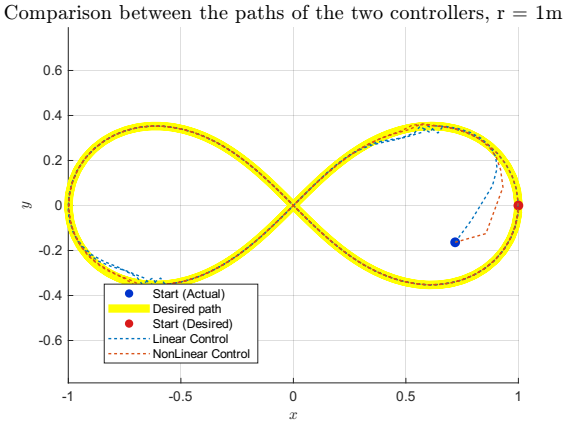
The results show that the (almost) nonlinear controller achieves lower **state-tracking error** than the linear controller for $r = 1$m, while for $r = 0.1$m the linear controller proves more effective.



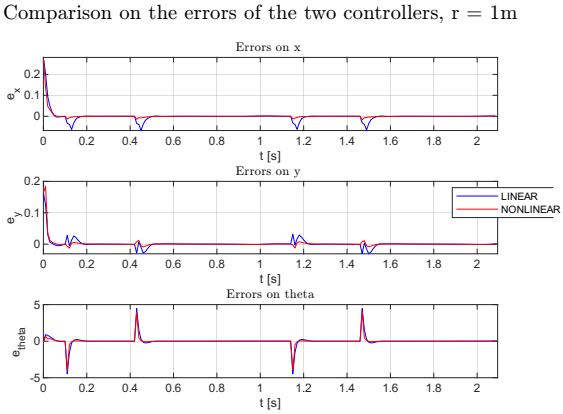
Linear vs Nonlinear path for r = 0.1m



Error on the states for r = 0.1m



Linear vs Nonlinear path for r = 1m



Error on the states for r = 1m

However, the increase in r results in a trajectory with larger curvature and higher velocity demands, especially in the angular component. This amplifies the difference between the two controllers: the linear controller struggles to keep up, especially near high curvature regions, while the nonlinear controller adapts more effectively to the dynamic changes.

Given the short final time $T_f \approx 2$ seconds, the robot is required to complete the trajectory very quickly, leading to inherently high **control inputs** demands. This tendency is further amplified when using the nonlinear controller. This behavior is consistent with the nature of nonlinear control, which typically reacts more aggressively to reduce tracking error, making full use of the available bandwidth. In a real-world scenario, such behavior might raise concerns related to physical feasibility, actuator limitations, or energy consumption, despite the superior tracking performance. Therefore, it may not be implementable on physical systems without saturation.

Regarding the choice of the **control gains**, significant differences can be observed between the linear and nonlinear cases. In the linear case, the same gains were used for both radius configurations ($r = 0.1\,\text{m}$ and $r = 1\,\text{m}$), as they were sufficient to guarantee good performance in both scenarios. In particular, it is noticeable that the gain k_2—which depends on the desired linear velocity v_d and angular velocity w_d—is significantly lower in the case of $r = 1\,\text{m}$. This is consistent with the fact that, when the robot must travel a greater distance in the same amount of time, both the linear and angular velocities increase, thereby

reducing the need for a high gain on k_2.

In the nonlinear case, on the other hand, the gains differ considerably between the two configurations, in order to ensure satisfactory performance in both cases. Specifically, for $r = 1\,\text{m}$, the gains are approximately one order of magnitude higher than those used for $r = 0.1\,\text{m}$. From the plots in Figure 22, it can also be observed that, in each configuration, the gains k_1 and k_3 remain positive and bounded at every time instant, as required by the controller to guarantee system stability.

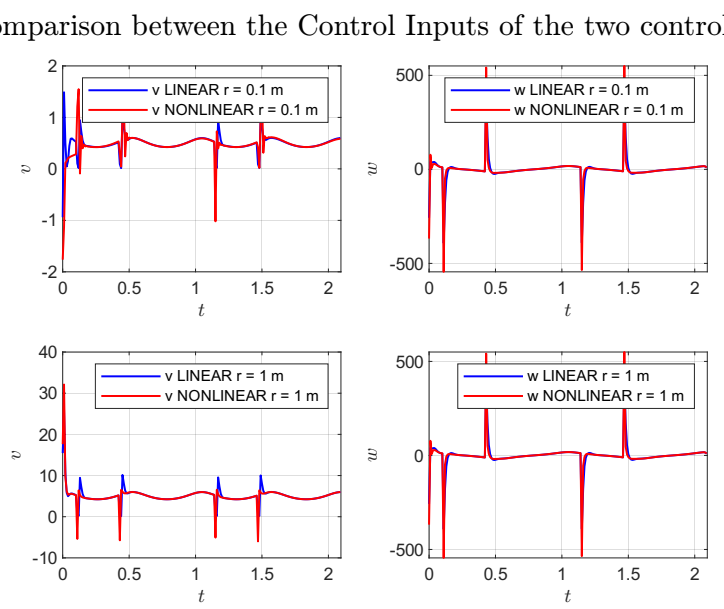


Figure 22: Control Inputs

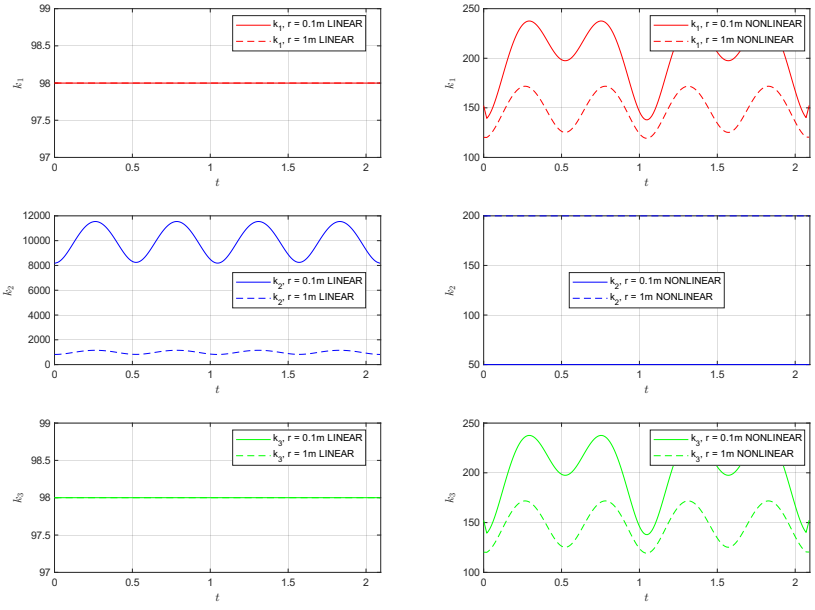


Figure 23: Gains evolution

Exercise 4

The posture regulator in polar coordinates used for this task is defined as follows:

$$v = k_1 \rho \cos\gamma \qquad\qquad w = k_2\gamma + k_1 \sin\gamma \cos\gamma \left(1 + k_3 \frac{\delta}{\gamma}\right)$$

with control gains set to $k_1 = 1$, $k_2 = 1$, $k_3 = 0.15$ and $T_s = 0.1$. The relationship between the polar coordinates (ρ, γ, δ) and the Cartesian coordinates (x, y, θ) is given by:

$$\rho = \sqrt{x^2 + y^2} \quad \gamma = \text{atan2}(y, x) - \theta + \pi \quad \delta = \gamma + \theta$$

The Simulink scheme implemented for this exercise is shown in Figure 24. It includes a block for converting Cartesian coordinates into polar coordinates, followed by the controller that computes the control inputs v and w. The scheme also features an odometric localization block, which implements the following Runge-Kutta-based approximation:

$$x_{k+1} = x_k + v_k T_s \cos\left(\theta_k + \frac{1}{2} w_k T_s\right) \quad y_{k+1} = y_k + v_k T_s \sin\left(\theta_k + \frac{1}{2} w_k T_s\right) \quad \theta_{k+1} = \theta_k + w_k T_s$$

In real robotic systems, the control inputs v and w are not directly known but must be estimated through measurements from wheel encoders using the following formulas: $v_k = \frac{\rho}{2T_s}(\Delta\phi_s + \Delta\phi_l)$, $w_k = \frac{\rho}{dT_s}(\Delta\phi_s - \Delta\phi_l)$ where d is the distance between the wheels of the differential drive robot and ρ is the radius of the wheels. These estimations are subject to quantization errors, noise, and wheel slippage, all of which degrade the localization accuracy. Hence, relying on ideal control inputs, as we do here, overestimates the performance of the localization algorithm. However, since MATLAB is not well suited for simulating encoder behavior, the best approximation is to directly use the values of v and w computed by the posture regulator as inputs to the localization module.

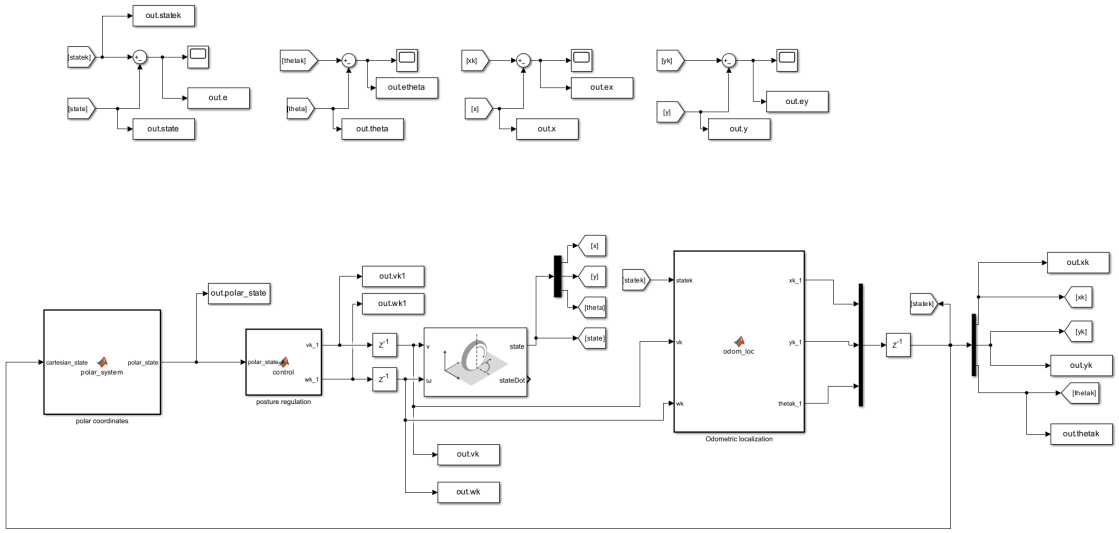


Figure 24: Simulink scheme

As shown in Figure 27, the error between the actual states of the unicycle and those estimated by the odometric localization block does not converge to zero, but rather persists as a small steady-state error. This residual discrepancy is primarily attributed to the numerical approximation introduced by the Runge-Kutta integration scheme and the use of ideal control inputs in the simulation. In a real system, additional factors such as encoder quantization, wheel slippage, and sensor noise would further amplify this error, making accurate localization even more challenging.

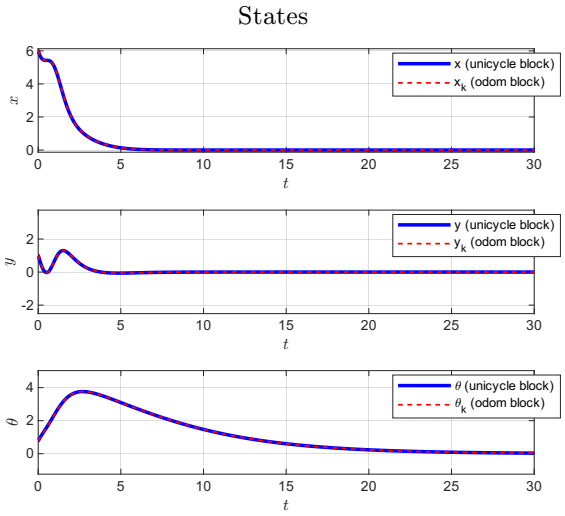


Figure 25: Comparison between the states of the unicycle block and the odometric localization block

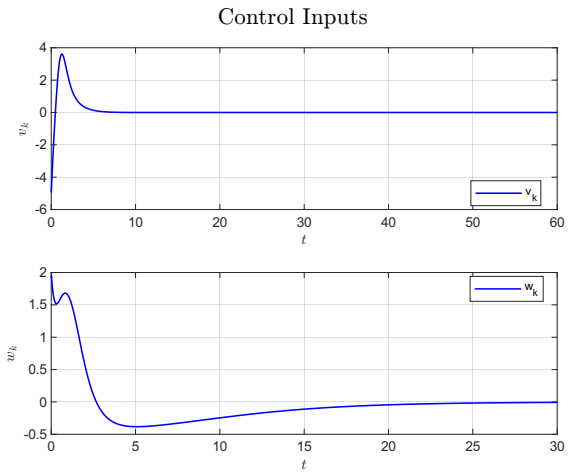


Figure 26: Control Inputs

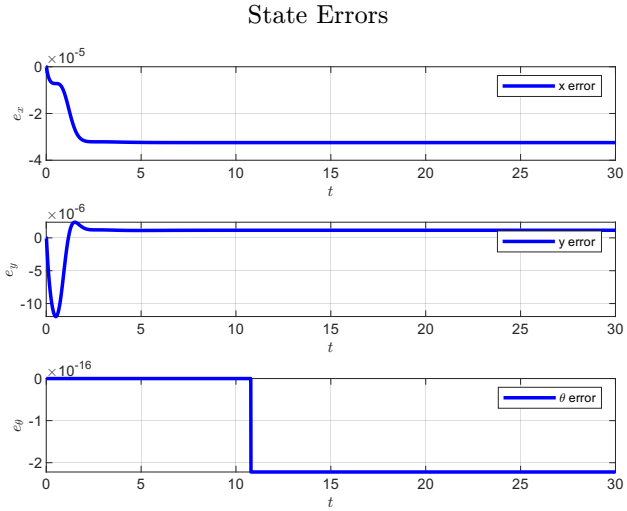


Figure 27: State Errors