CEG 4158: Computer Control in ROBOTICS

Department of Electrical Engineering and Computer Science

University of Ottawa
Fall 2023

Presented to:
Dr. Farzan Soleymani

Completed by:

Student Name	Contribution
William Wang - 300128012	Q1
Guangyu Niu - 300084738	Q1
Natasha Tremblay - 300168364	Q2 – EKF (non-weighted)
Fatmah Bayrli - 300159193	Q2 – WEKF (weighted)
Ines Mansouri - 300191449	Q2 - UKF

Submission Date: December 4th, 2023

Table of Contents

1. Introduction

For the first part, Accurate control of multiple degrees of freedom (DOF) systems is critical in robotics to provide precise trajectory tracking and placement. In this research, proportional-integral (PI), proportional-derivative (PD), and proportional-integral-derivative (PID) controllers are used in the design and study of a control system for a 3-link, 3-DOF robot. Deriving torques, linearizing the system, and figuring out the best controller gains for every configuration are the objectives. Simulation tests using step and sinusoidal inputs and trajectory tracking for a prescribed trajectory between defined locations with a trapezoidal velocity profile will be used to assess the controllers. MATLAB will be used to implement the design and analysis. The project is to uncover the nuanced behaviors and performances of PI, PD, and PID controllers in addressing the complexities of a 3-link, 3-DOF robot. Through rigorous analysis and simulation experiments, we seek to draw meaningful conclusions that shed light on the efficacy of each controller type under various input scenarios.

For the second part, In the rapidly evolving landscape of autonomous vehicles, achieving precise and reliable positioning is paramount for ensuring safe and efficient navigation. Sensor fusion, a critical component of autonomous systems, involves integrating data from multiple sensors to enhance accuracy, robustness, and overall performance. This process allows autonomous vehicles to overcome challenges such as sensor limitations, environmental variability, and unforeseen obstacles. In this section, we delve into the concept of sensor fusion and its pivotal role in optimizing the positioning of autonomous vehicles. Sensor fusion plays a pivotal role in advancing the capabilities of autonomous vehicles by amalgamating information from diverse sensors to create a comprehensive and accurate representation of the surrounding environment. This section explores the principles and methodologies of sensor fusion, shedding light on how it contributes to the enhanced positioning of autonomous vehicles. From combining data from LiDAR, radar, and cameras to utilizing advanced algorithms for real-time processing, sensor fusion holds the key to addressing the complexities associated with autonomous navigation. Through an in-depth examination of sensor fusion techniques, the aim is to provide insights into the mechanisms that empower autonomous vehicles to navigate safely and effectively in dynamic and unpredictable environments.

2. Discussion Question One

2.1 Task 1 Simulation with Sine wave and step function as Input

In this part we are going to use Simulink to simulate using PID/PI/PD controller to control a 3 DOF robot which does not include any rotational motion. Before we go deeper, we need to know what these three controllers represent.

Basically, each letter represents different parameters that is included in the controller. P stands for proportional (Kp), it controls the speed of the system response. I stand for integral which controls the speed of steady-state error removal. And D stands for derivative, which is for predicting change. PID and PI controllers are mostly using in controlling systems since the PD controller mainly depends on prediction and it will be easily influenced by noises.

2.1.1 Previous work and adjustments

Before the presentation our block diagram includes the rigid body tree, controller block and calculation blocks for speed velocity and position shows in the following figure:

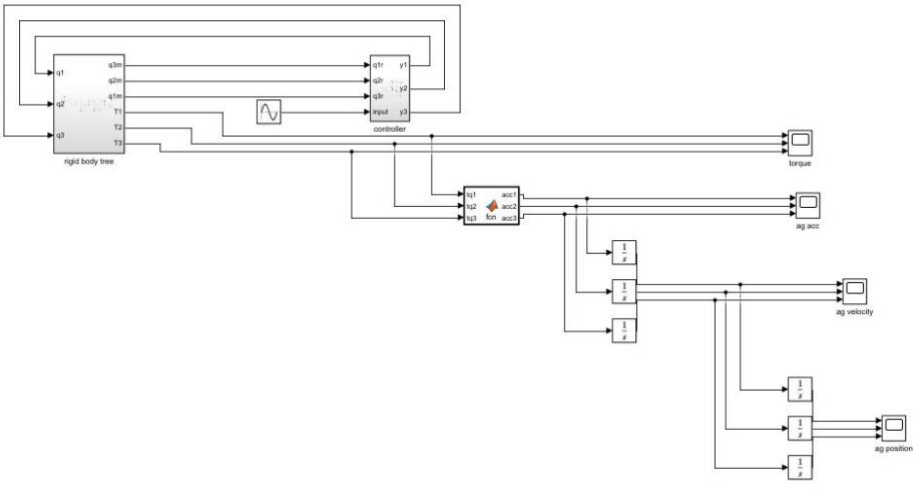

Figure 1: block diagram for overview

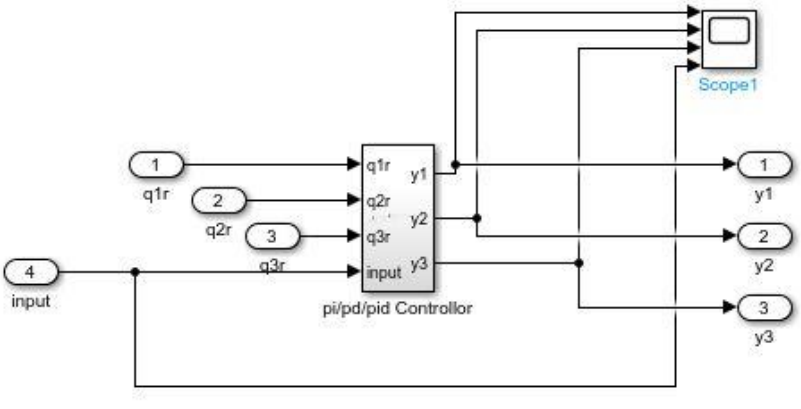

Figure 2: Controller block

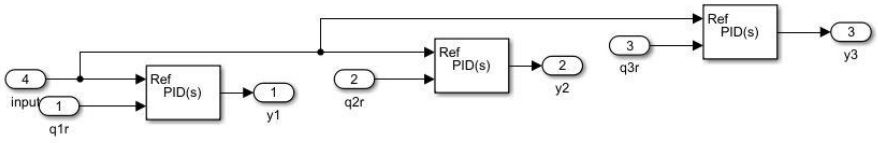

Figure 3: Controllers inside the block

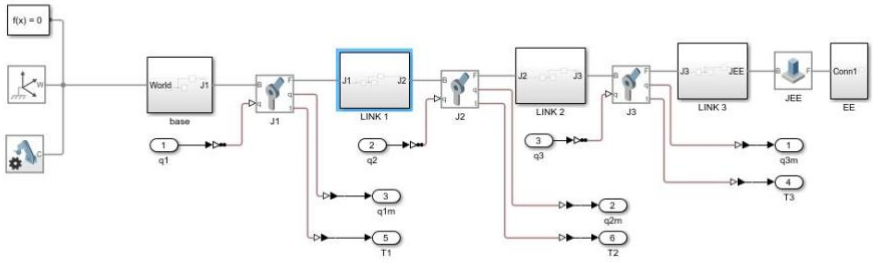

Figure 4: Rigid body tree

With the help from professor Farzan we noticed that we included a rotational motion which is not suppose to be in this project and also there are some errors in the results shown as follow:

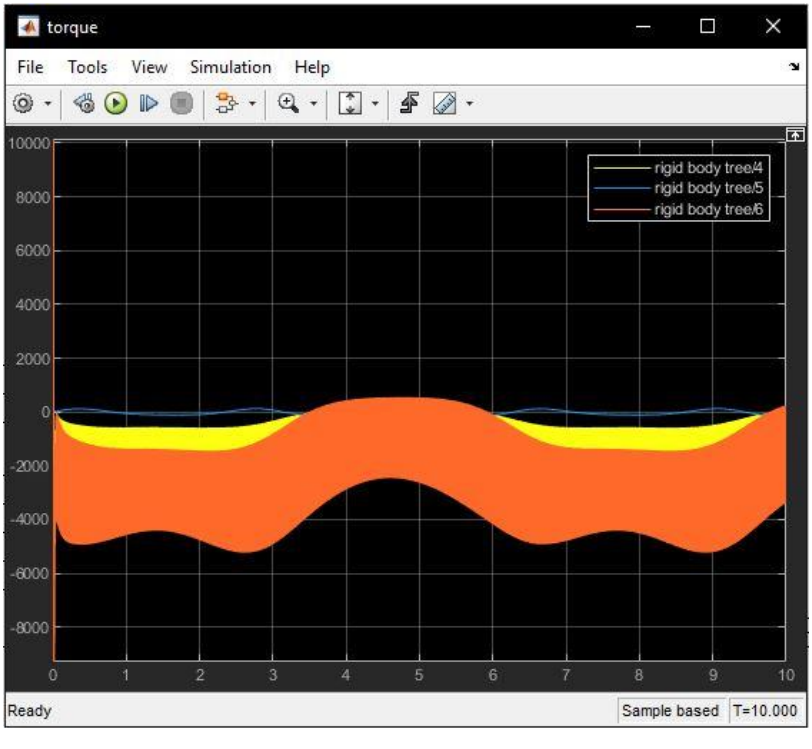

Figure 5: Result with error for sine wave input with PID controller

We can see that the torque varies between hundreds to thousands, this is due to the error in adjusting the controller parameters. We also notice that the calculation part is unnecessary since all the calculations can be automatically export by editing those blocks.

To fix the problem that we found in the presentation, which is torque varies in a huge range, we came up with two solutions: one is to control the torque instead of the motion, and another one is to slow down the motion by changing parameters of controllers. Unfortunately, both solutions have some problems that we cannot solve.

The updated block diagram is shown as follow:

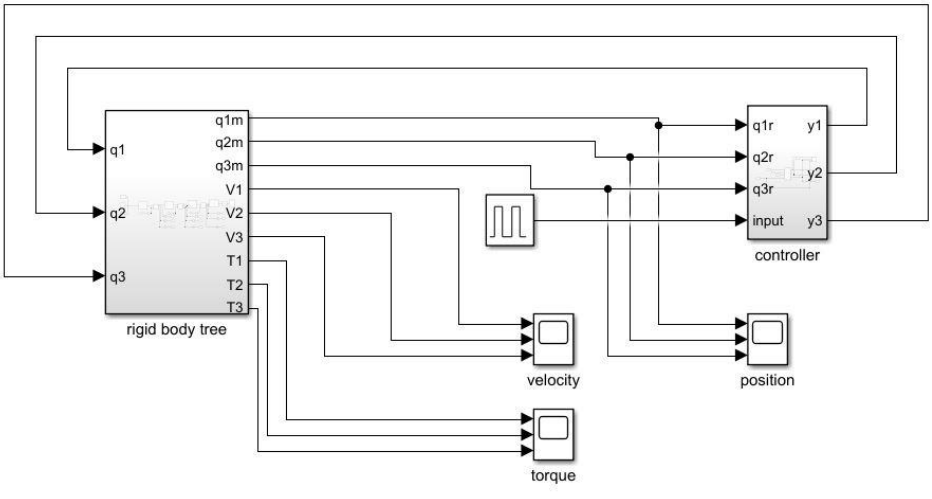

Figure 6: Updated verson of overview

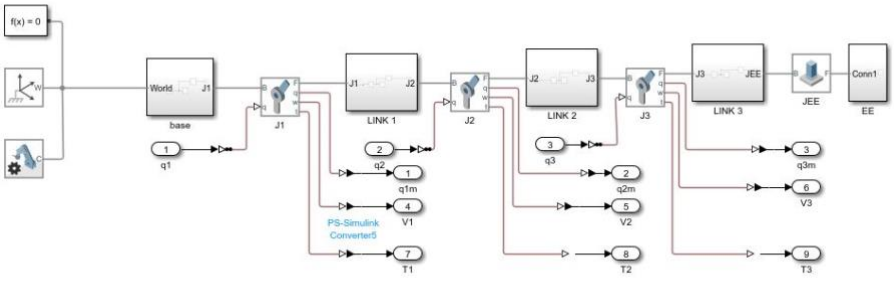

Figure 7: Rigid body tree

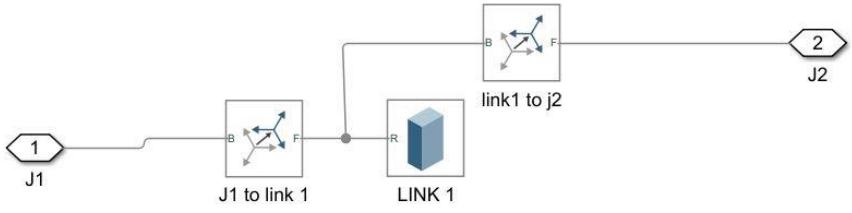

Figure 8: Example of a link

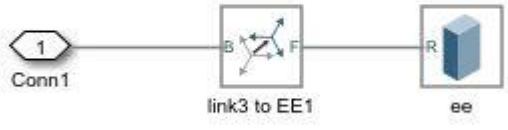

Figure 9: End effort

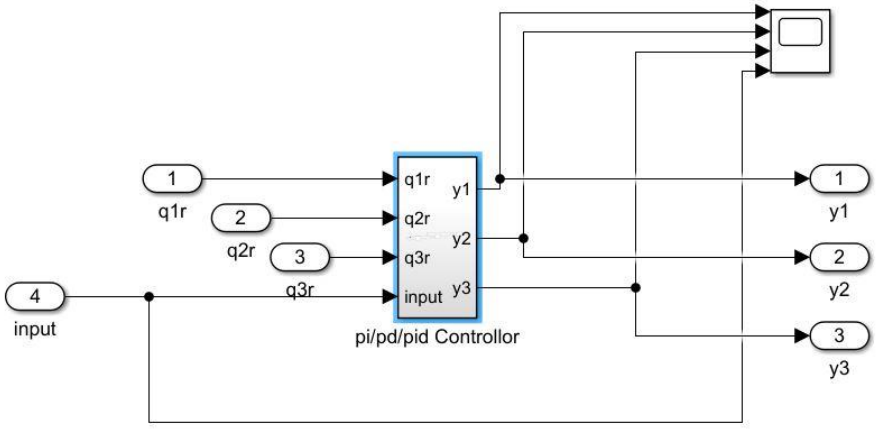

Figure 10: Controller block

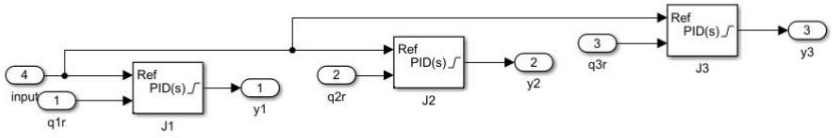

Figure 11: Details of the controller

From the graph we can see that we are mainly using the Robotic system toolbox to build up the rigid body. To make it a 3 DOF robot we need to create the base first which needs to include a solver configuration, a world frame, and a Mechanism configuration to connect those to the base.

To connect the base, we firstly need to move the frame to locate where the base is supposed to be since the base does not have any motion, so we have no need to add a joint under the base. Then we locate the position of the joint. Since joints will only do a rotational motion around the z-axis and we also only want it to move on the x-z plane. Therefore, we also need to move the z-axis pointing out the to the screen which is doing a 90-degree rotation around the x-axis as the same when we locate the joint position. After that we need to locate the next joint center and rotate the x-axis back to the original. The following buding steps will be the same as we mentioned before. At the end we need to add an end effect to the robot.

2.1.2 Two solutions

First we tried to control the torque instead of the motion but we met an unsolveable problem. When we start to simulate the block, the robot just fall down eventhough we adjust those controllers several times. The simulation model is shown in the following graph.

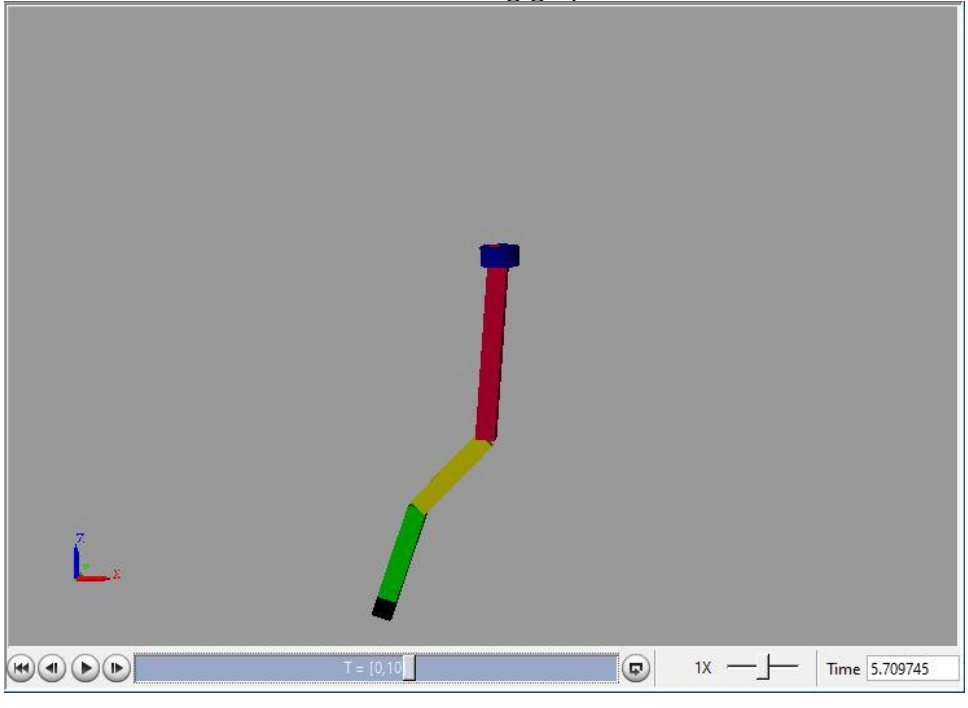

Figure 12: Failure in Torque control

In this simulation we even cannot generate any result. So, we gave up with this and tried to edit the parameters in controllers. To slow the motion down we changed the acting time to 5 seconds and limited the range of the output form –100 to 100. The adjustments are shown as follow:

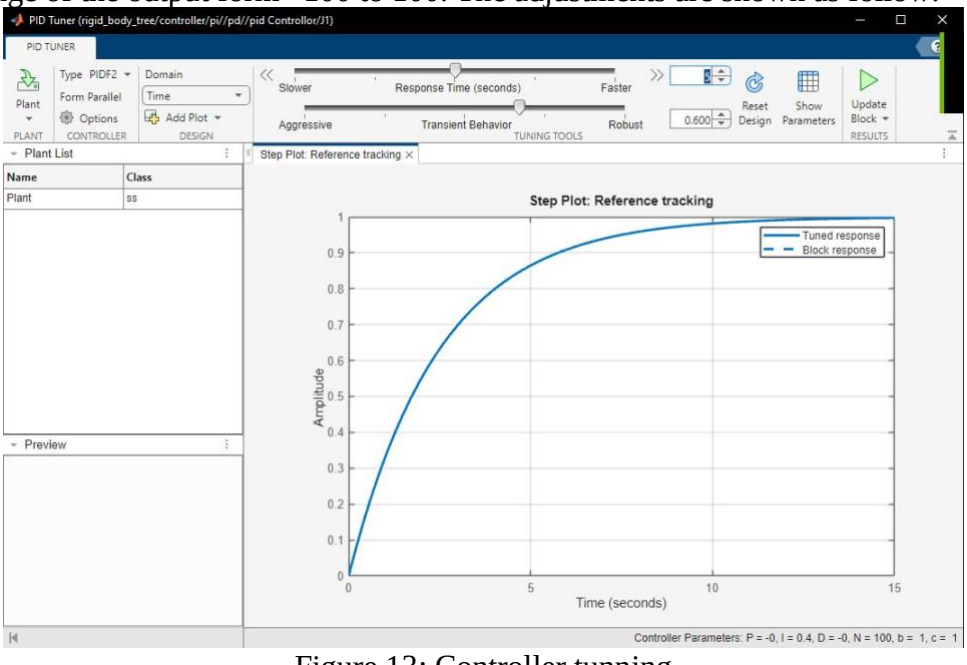

Figure 13: Controller tunning

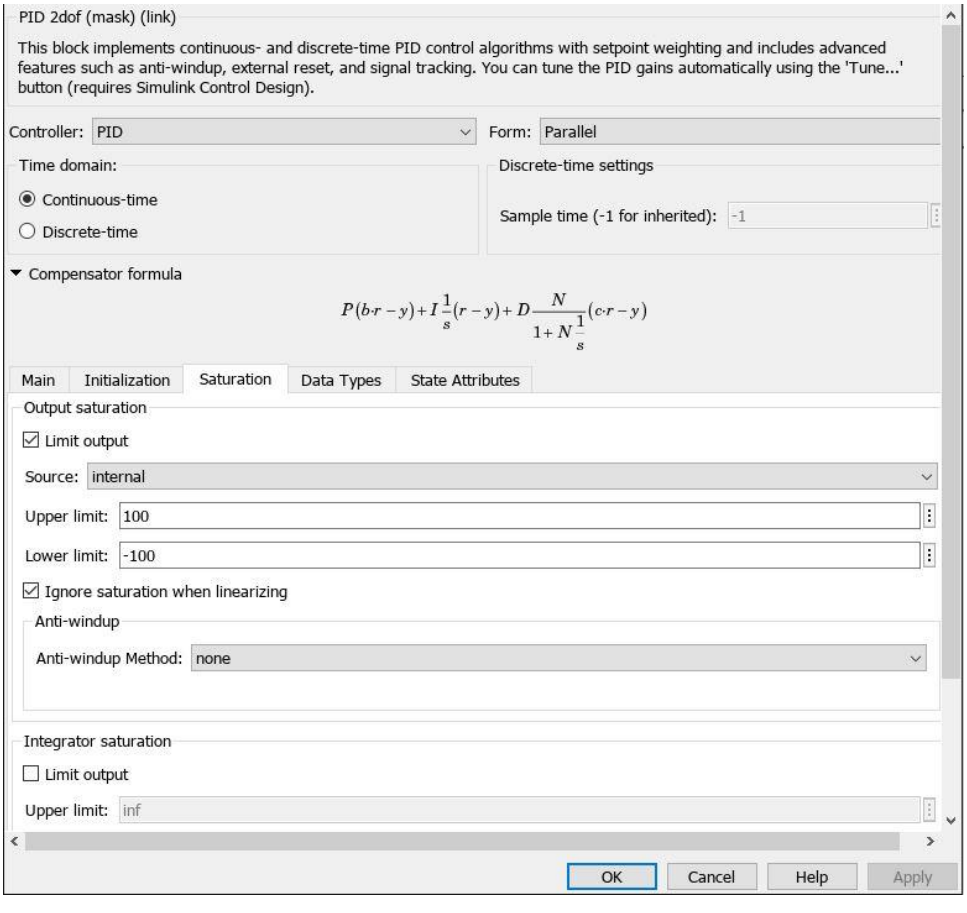

Figure 14: limit of controller output

The output of torque is still not good enough yet but if we keep increasing the acting time it will not move at all.
We are still using the method that using the sine wave first tune the controller and get the result, then change the input without tune the controller and get the result, after than we change back to sine wave input change the controller and tune it also get the result.

2.1.3 Result

2.1.3.1 Sine wave input PID controller

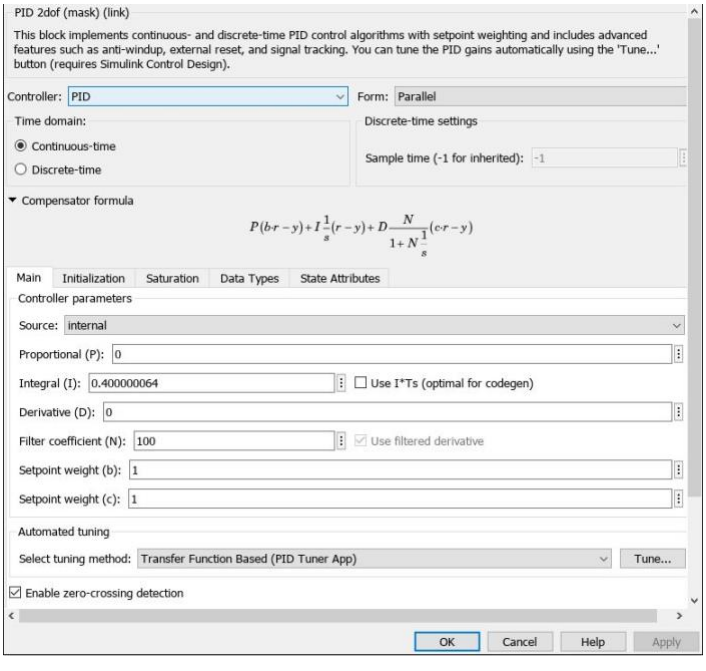

Figure 15: Joint 1 controller tunning

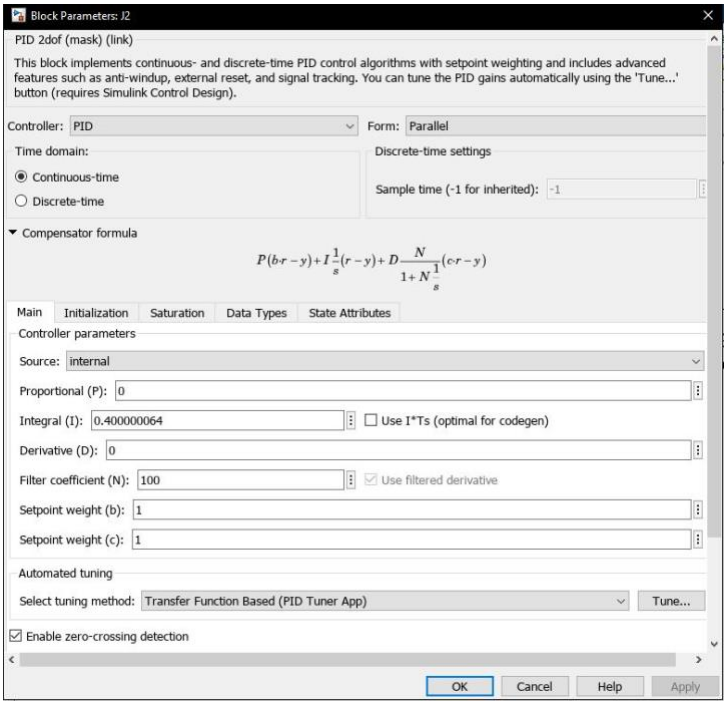

Figure 16: Joint 2 controller tunning

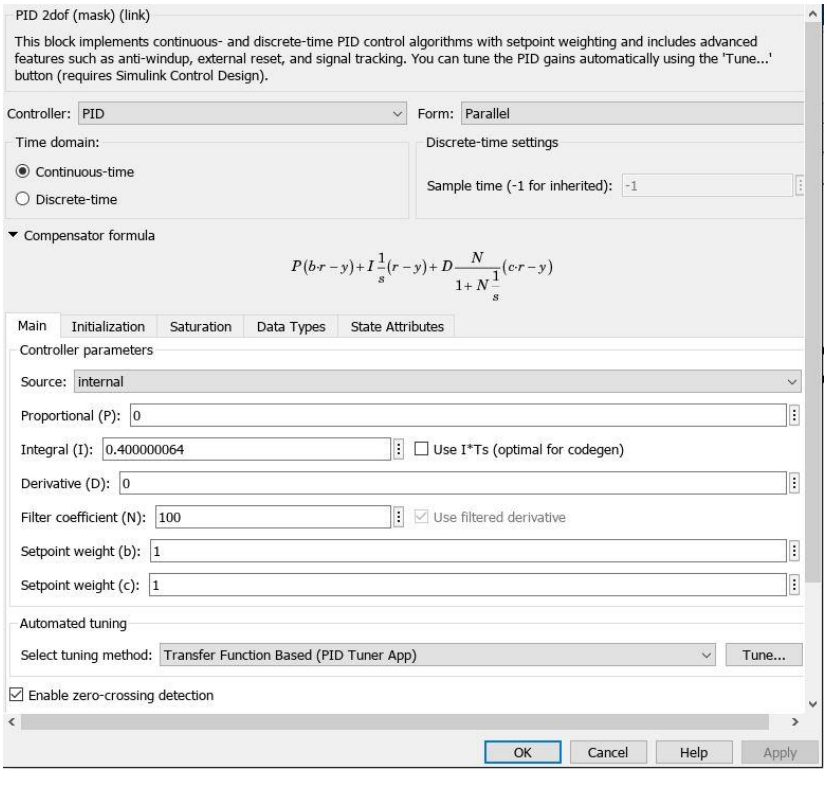

Figure 17: Joint 3 controller tunning

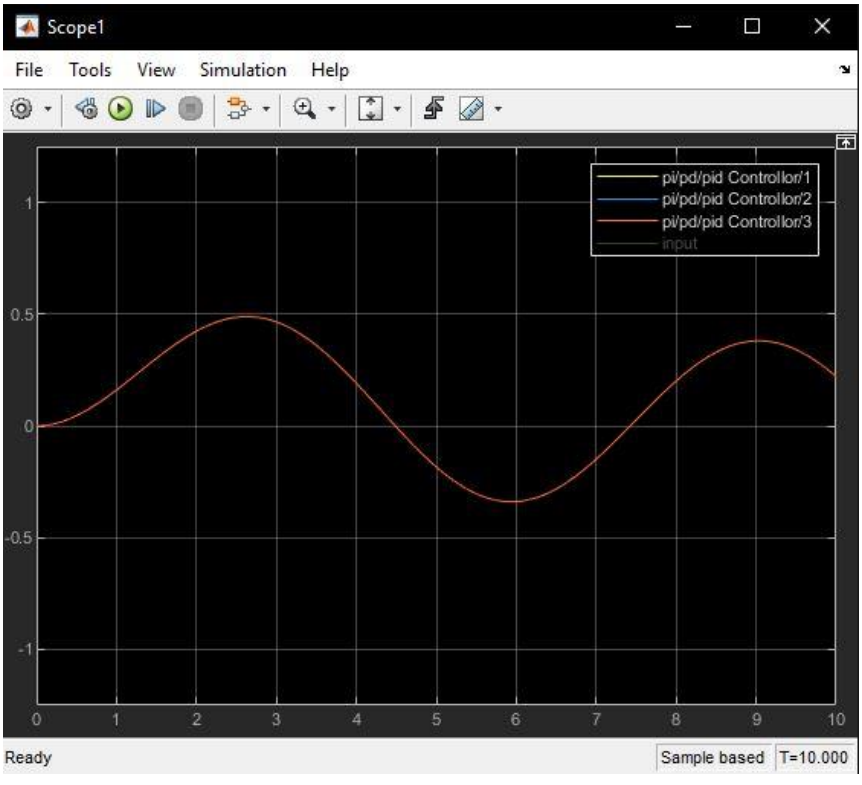

Figure 18: Controller response

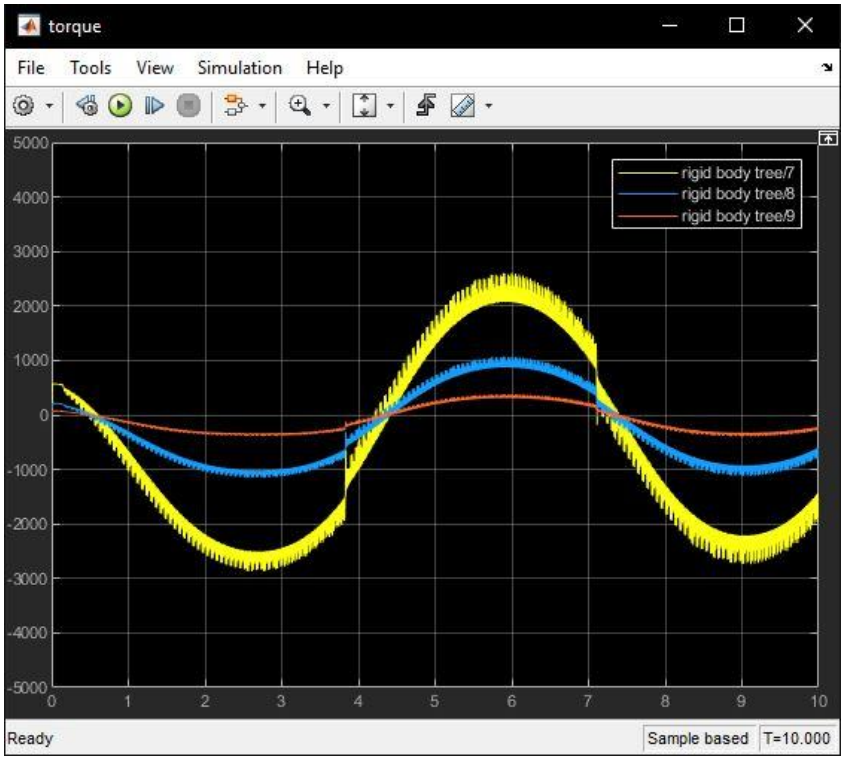

Figure 19: result for torque

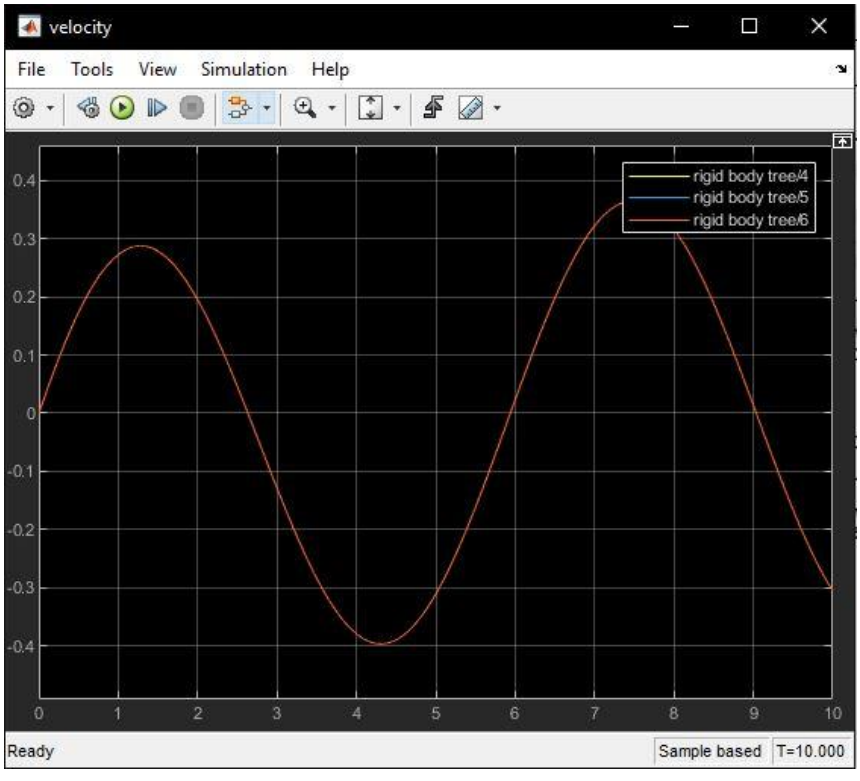

Figure 20: result for velocity

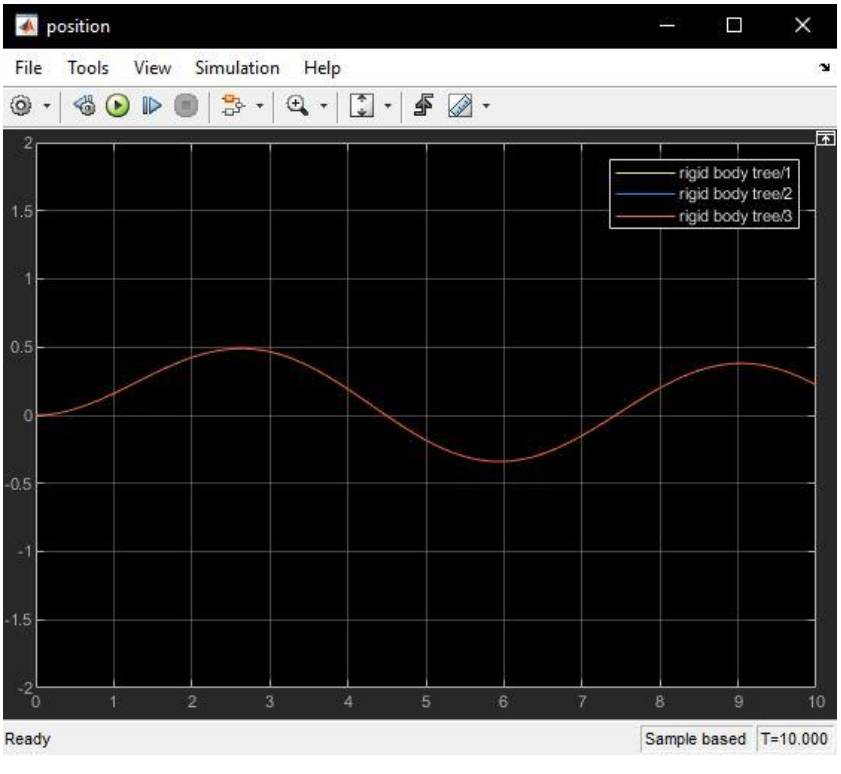

Figure 21: result for position

2.1.3.2 Step input PID controller

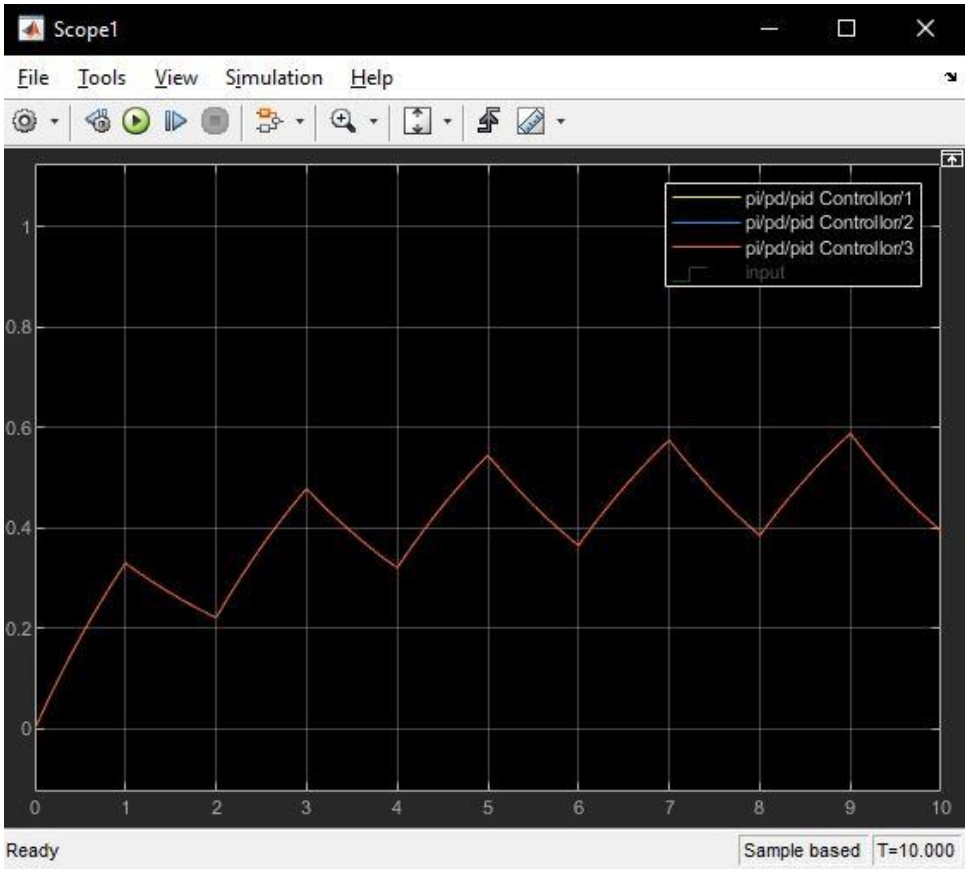

Figure 22: controller response

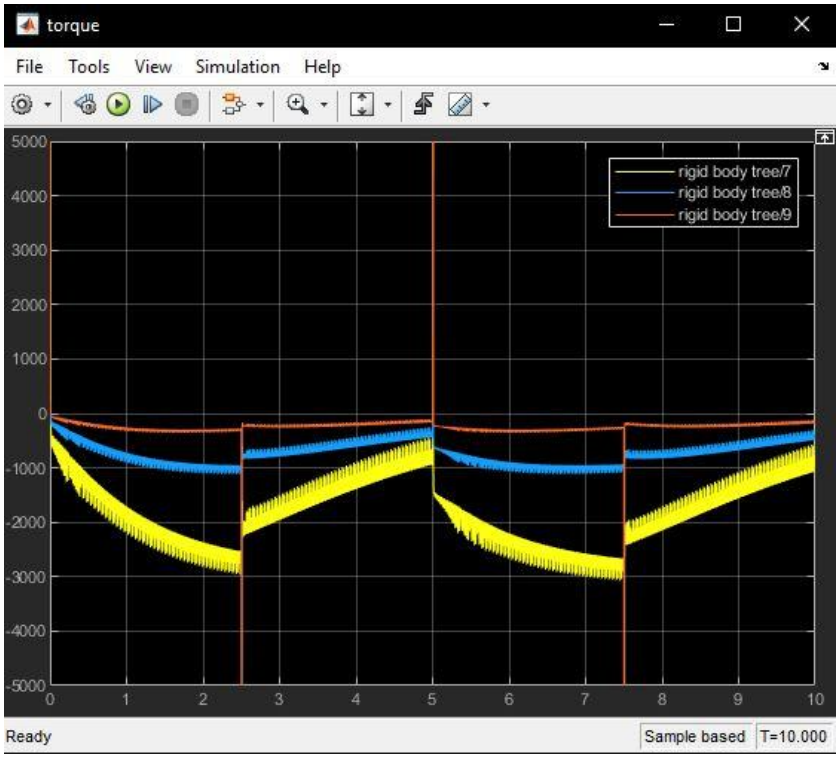

Figure 23: result for torque

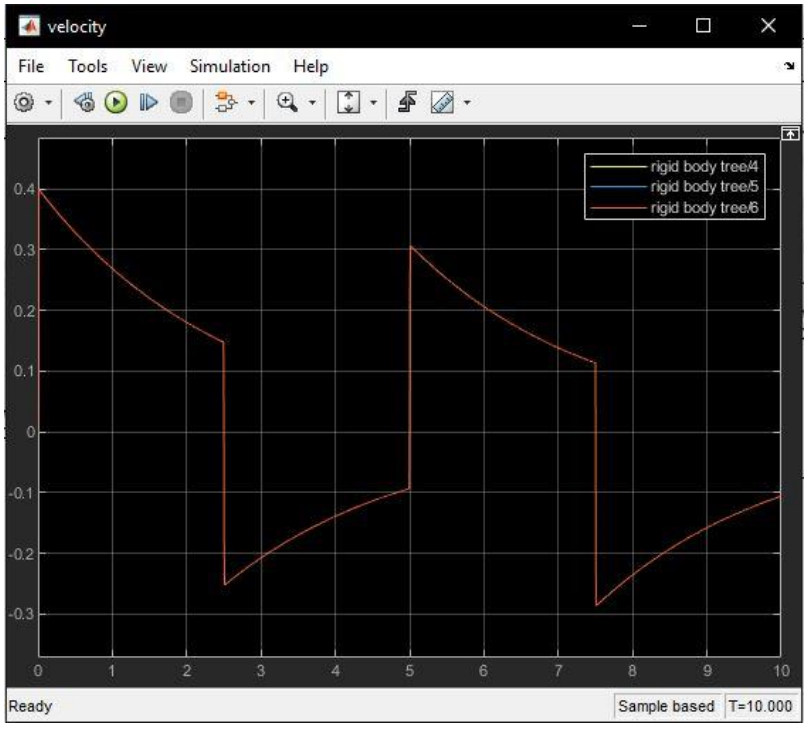

Figure 24: result for velocity

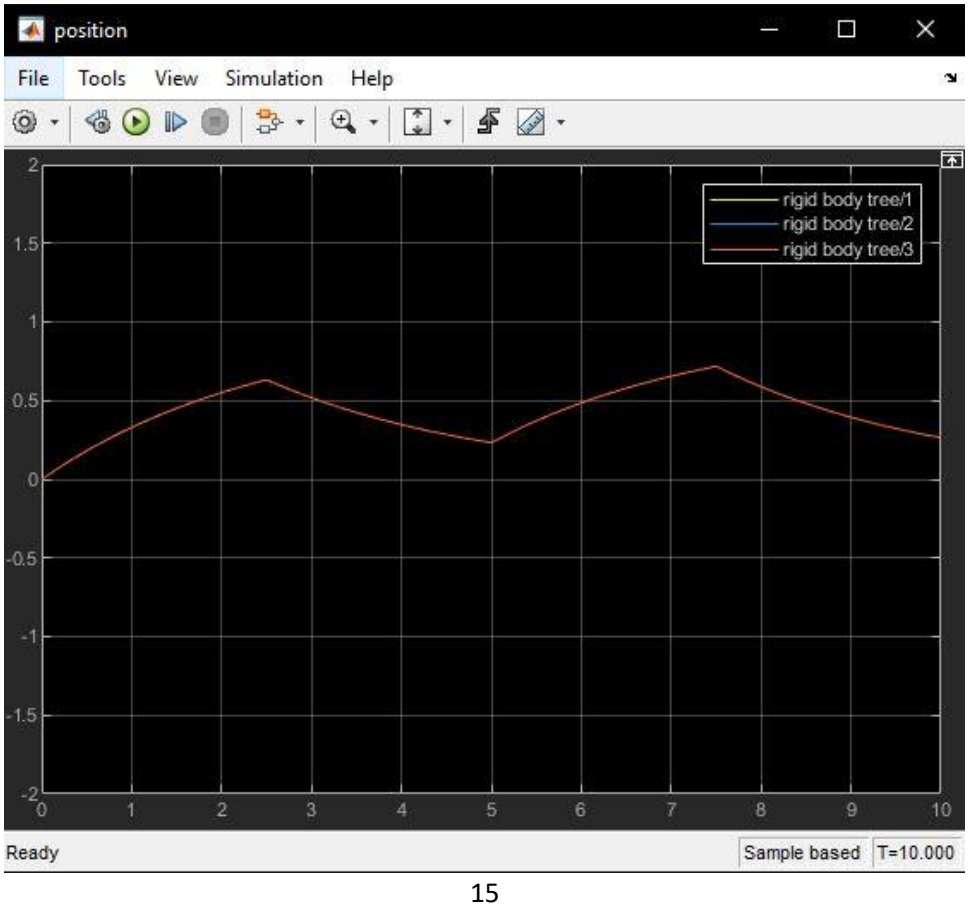

Figure 25: result for position

2.1.3.3 Sine wave input PI controller

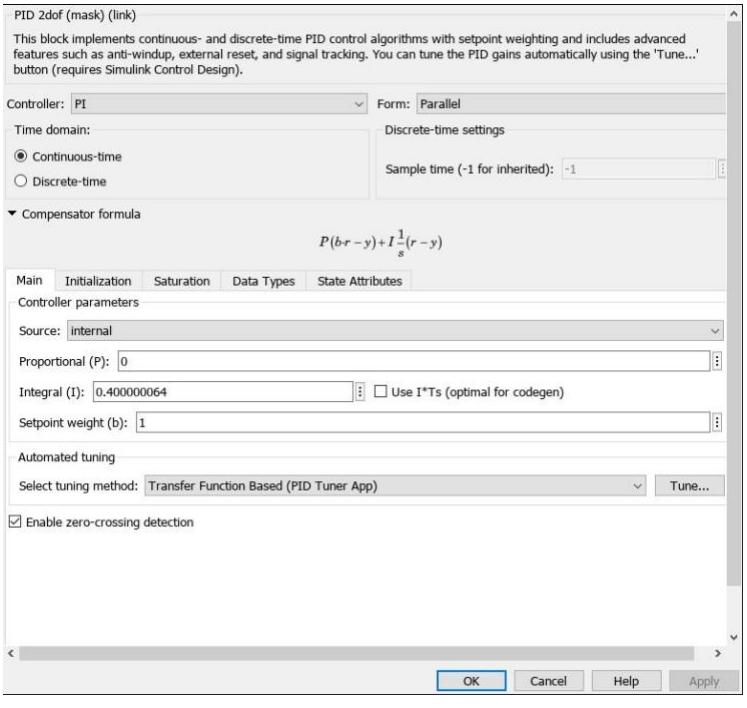

Figure 26: Joint 1 controller tunning

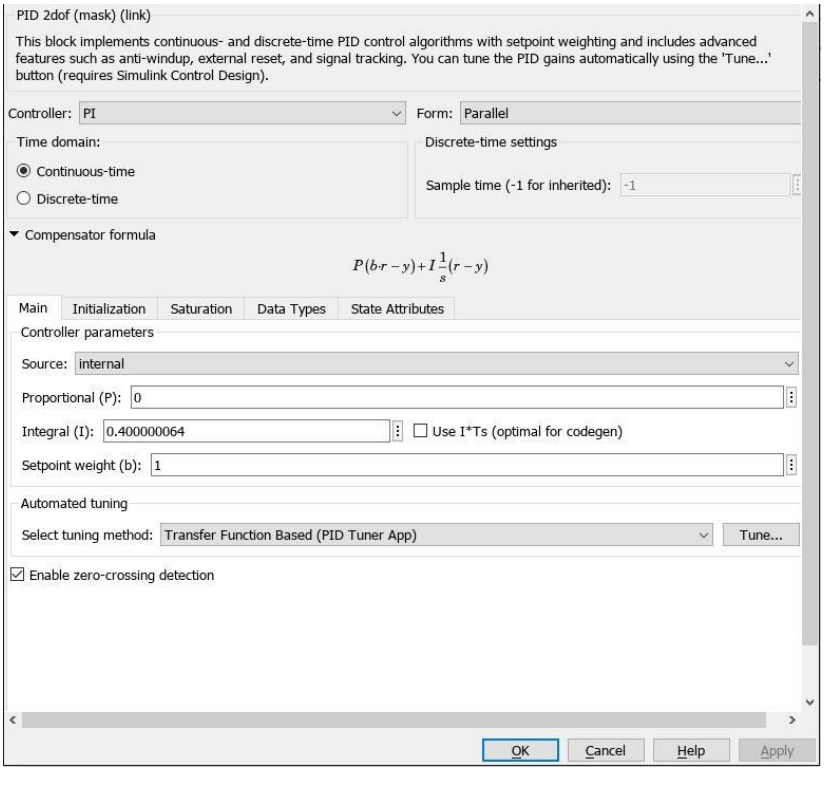

Figure 27: Joint 2 controller tunning

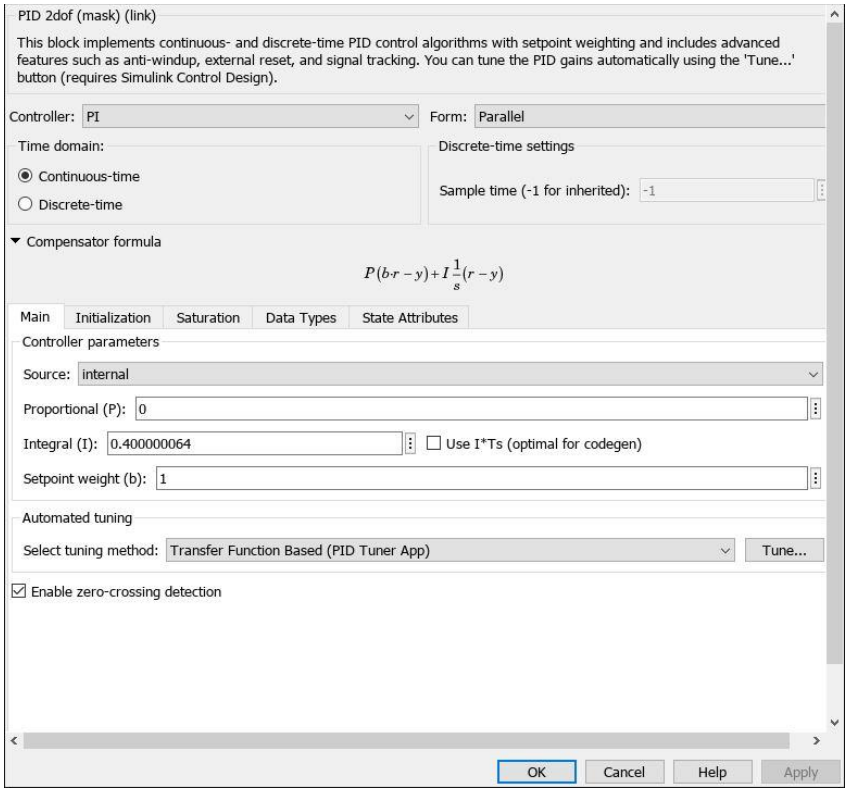

Figure 28: Joint 3 controller tunning

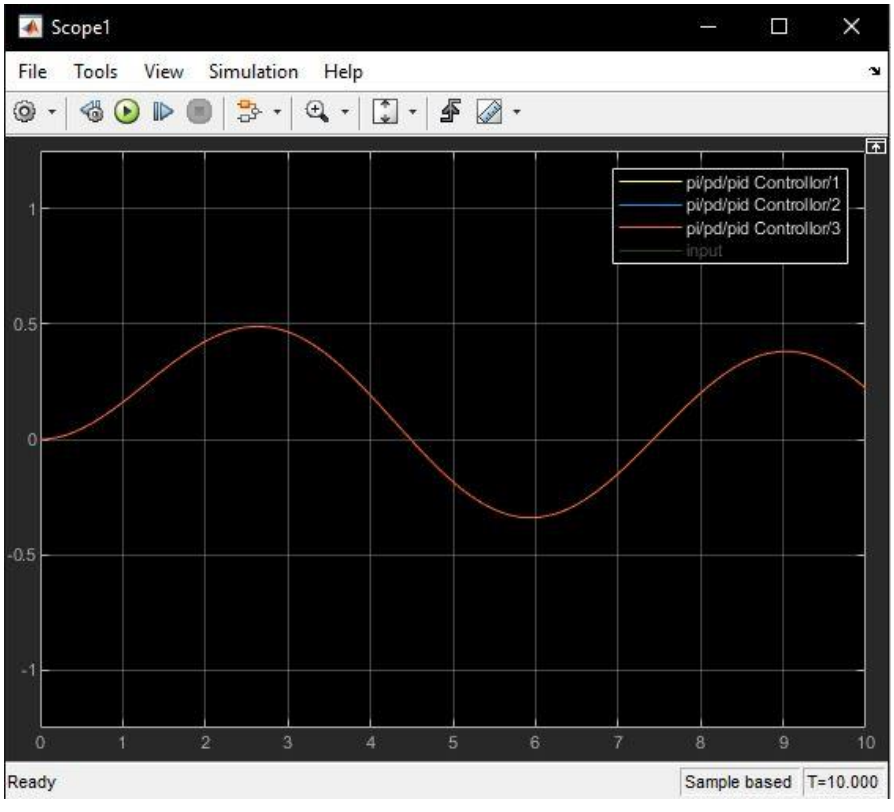

Figure 29: controller response

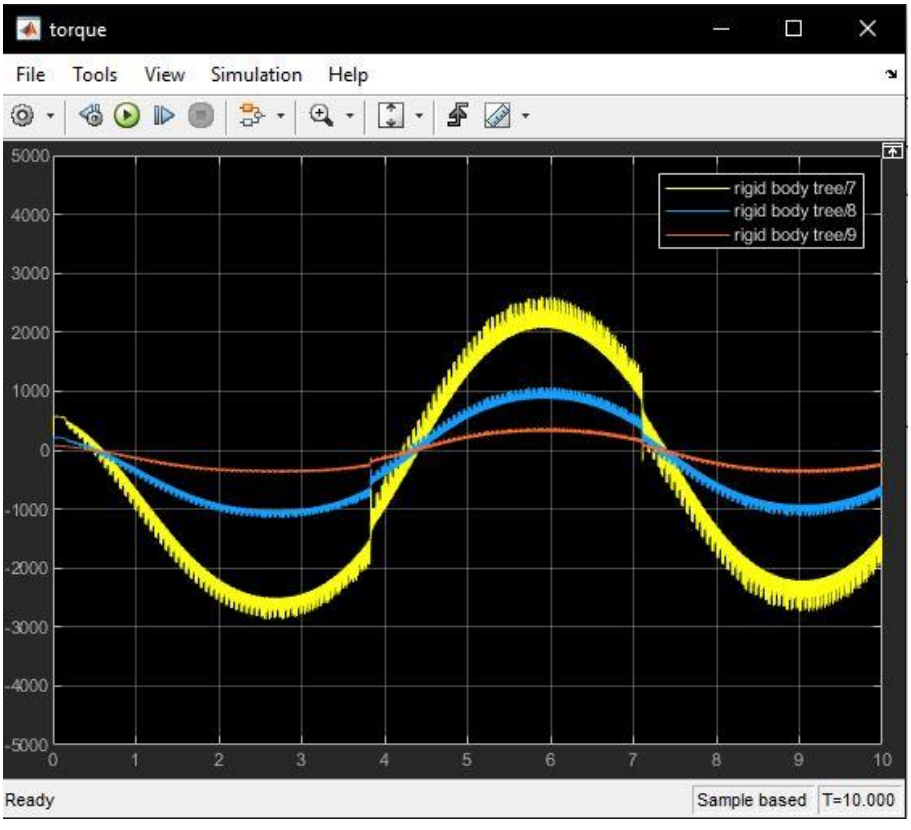

Figure 30: Result for torque

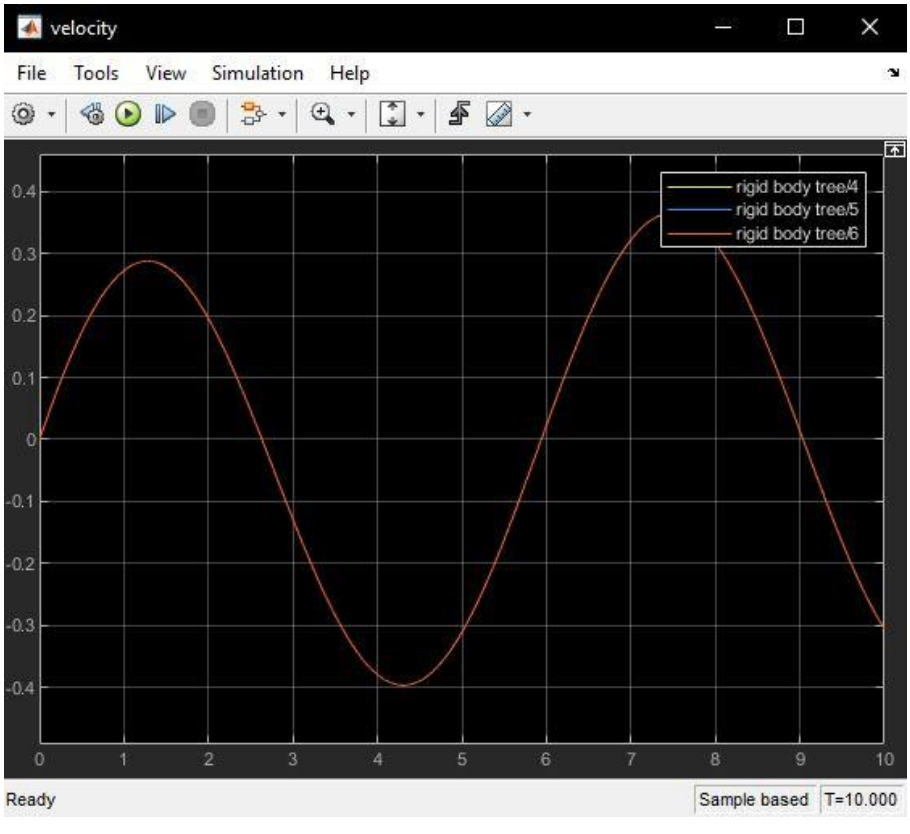

Figure 31: Result for velocity

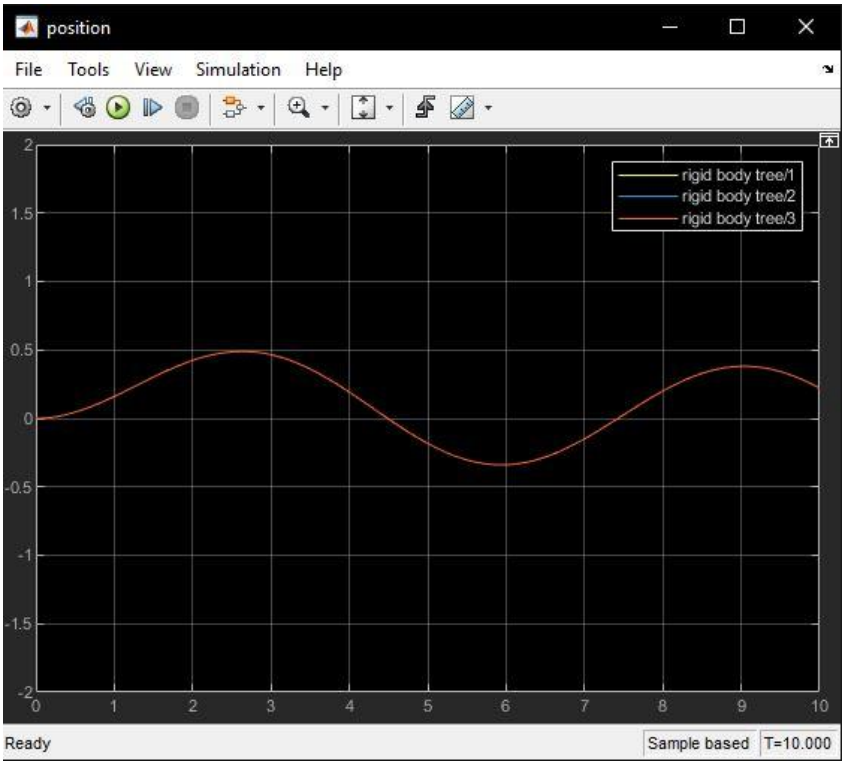

Figure 32: result for Position

2.1.3.4 Step input PI controller

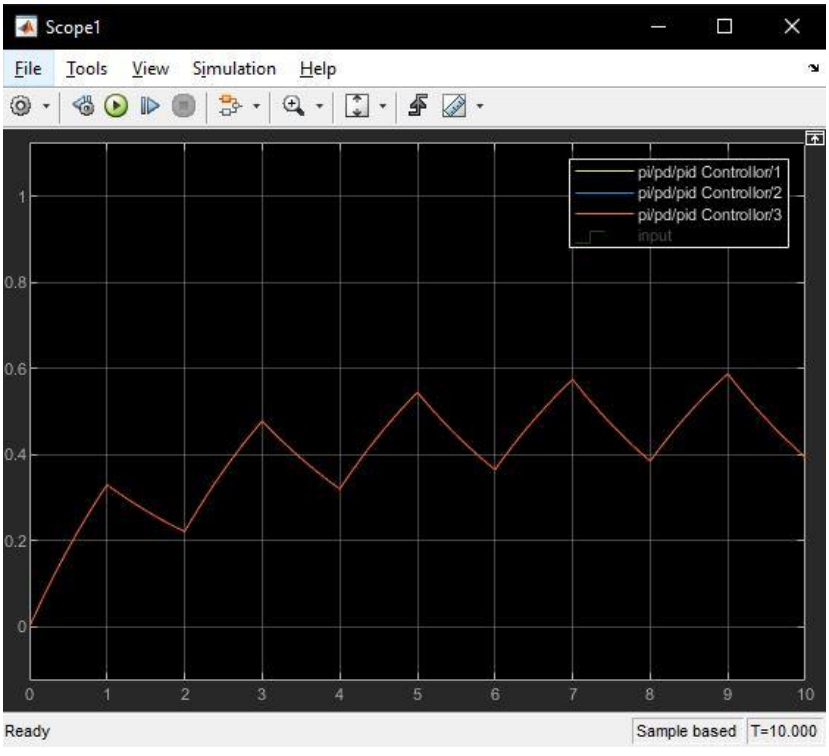

Figure 33: controller response

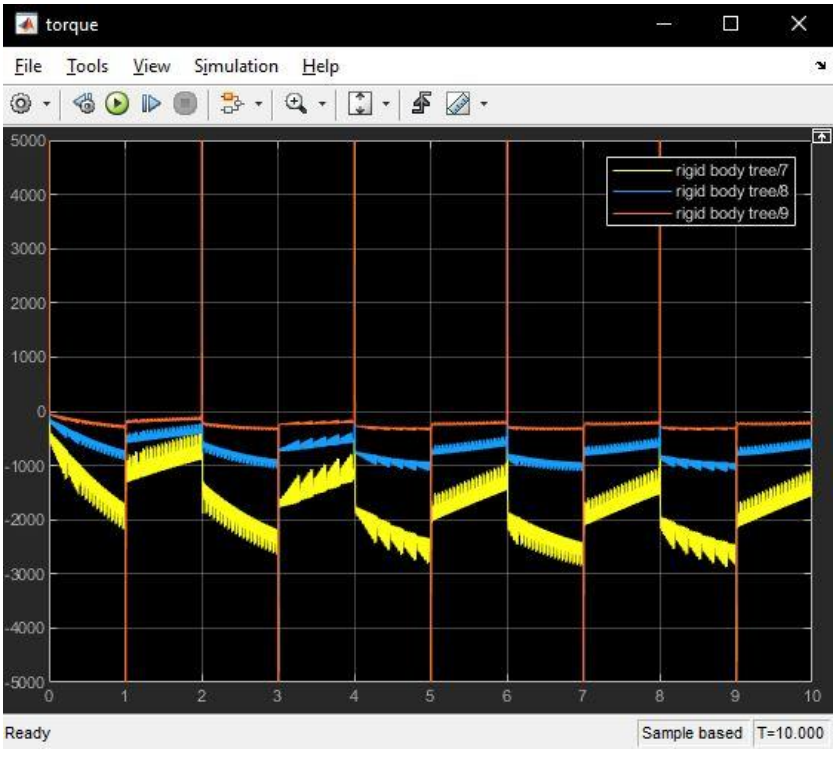

Figure 34: result for torque

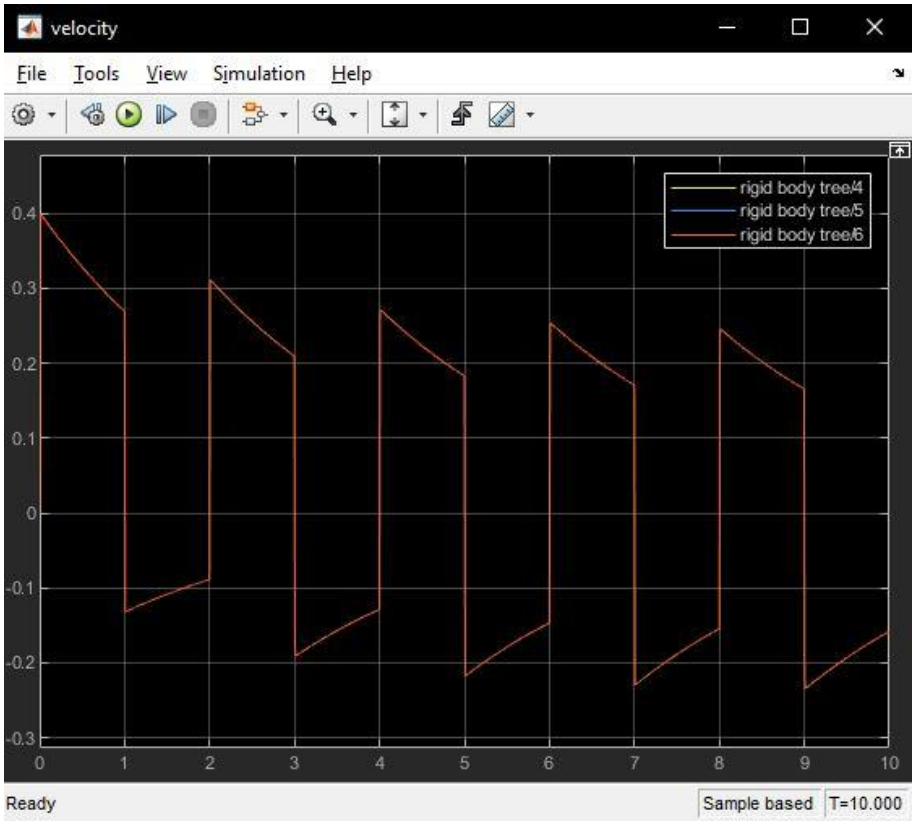

Figure 35: result for velocity

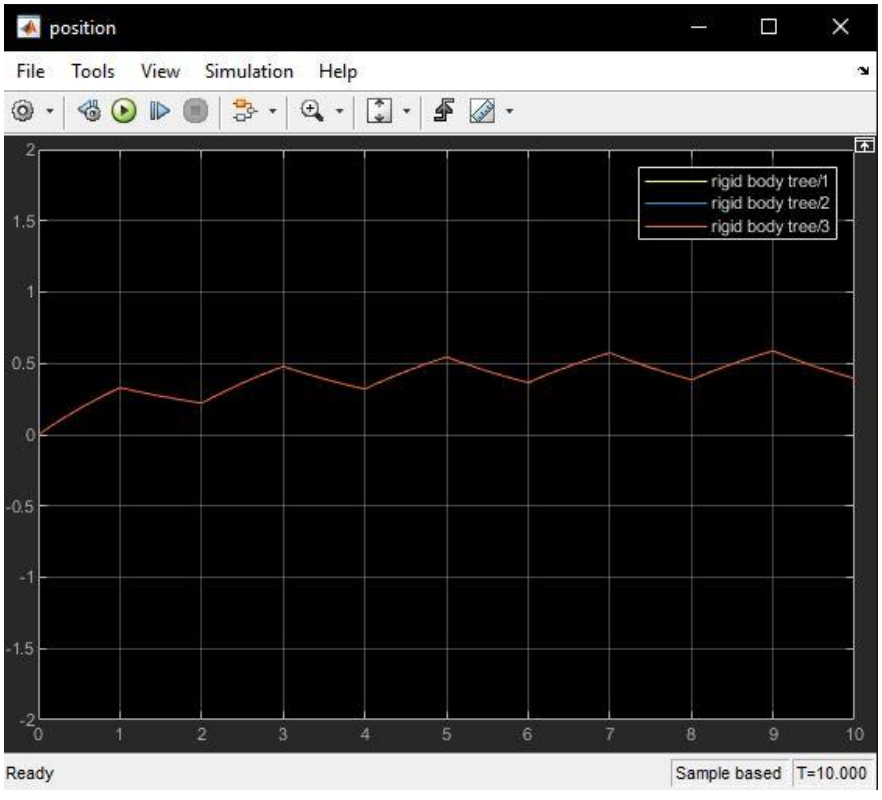

Figure 36: result for position

2.1.3.5 Sine wave input PD controller

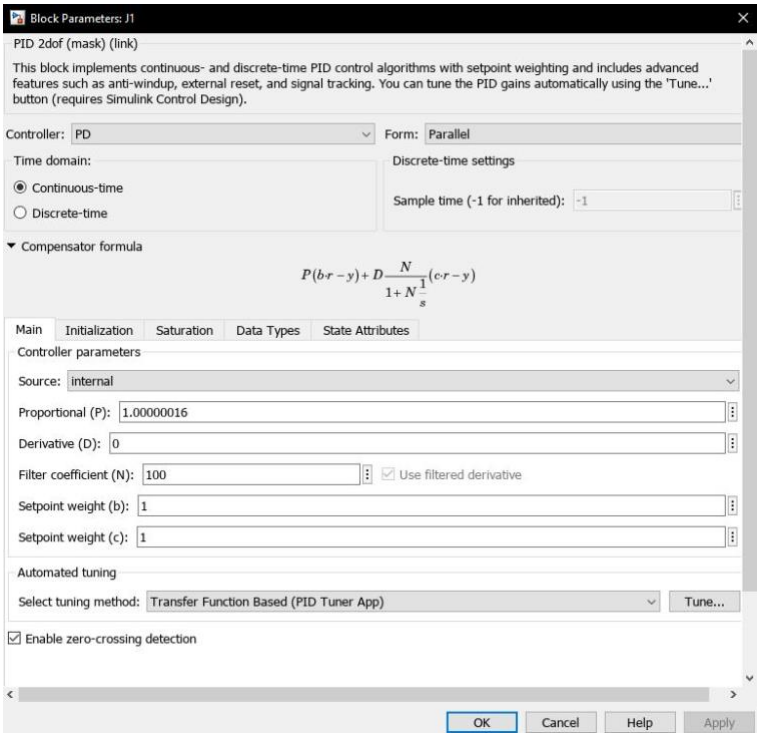

Figure 37: Joint 1 controller tunning

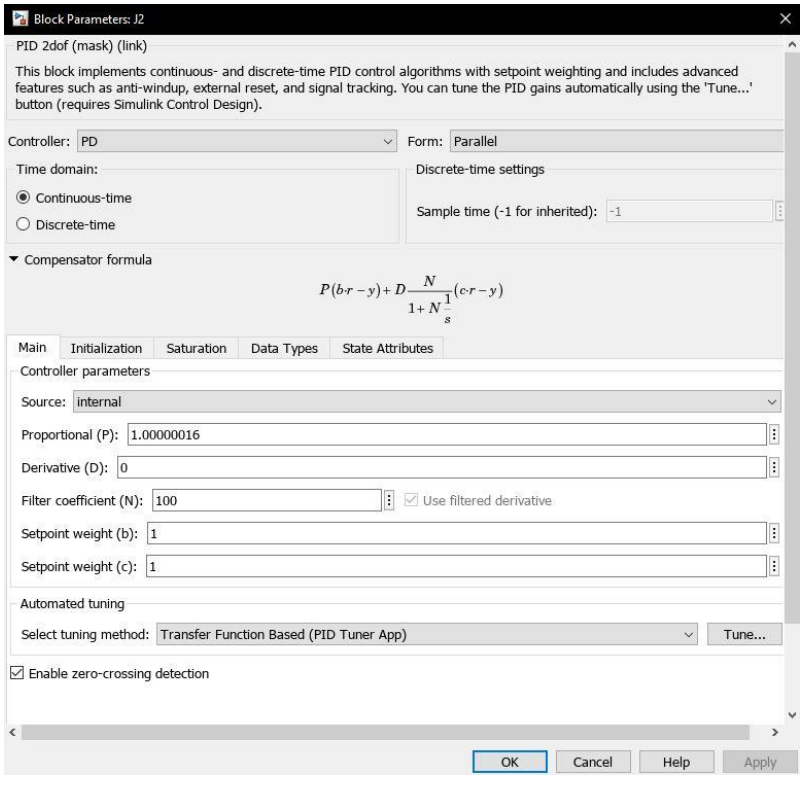

Figure 38: Joint 2 controller tunning

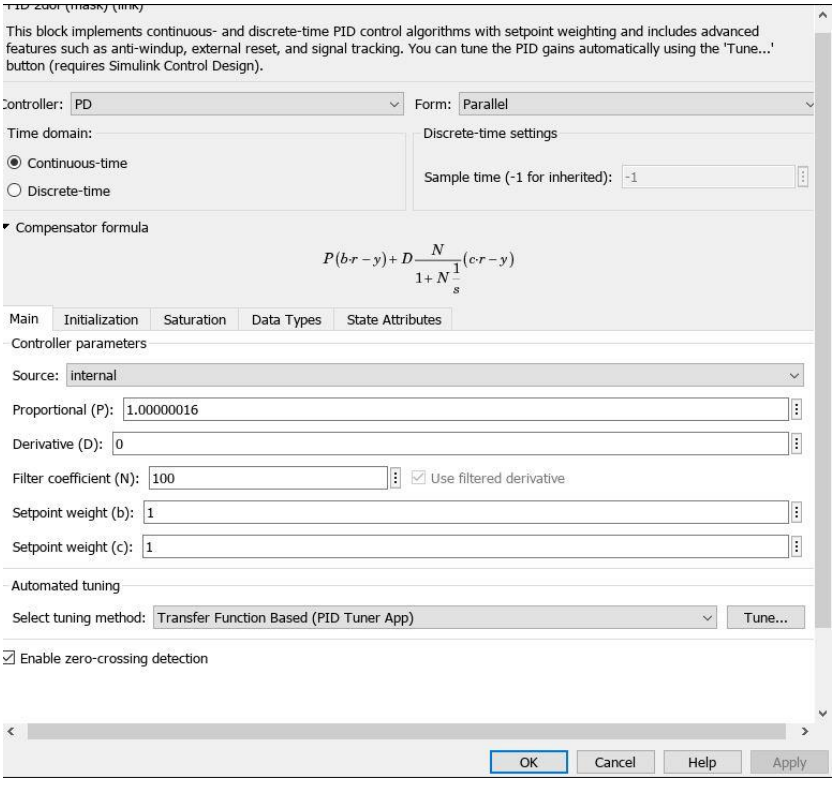

Figure 39: Joint 3 controller tunning

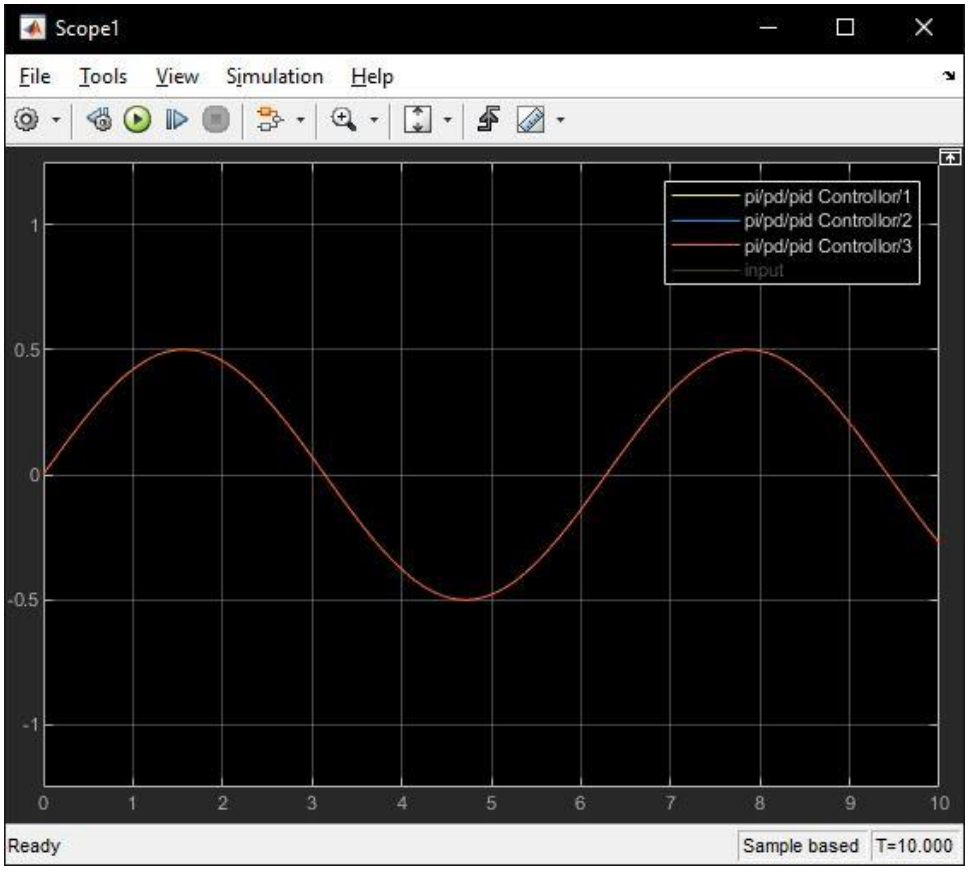

Figure 40: Controller response

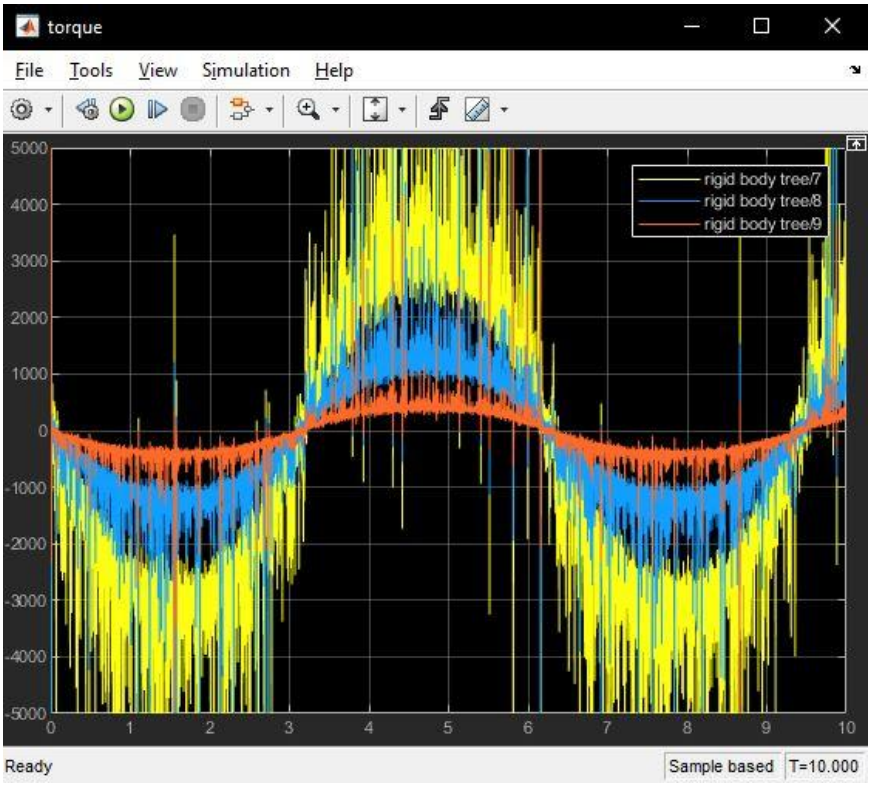

Figure 41: Result for torque

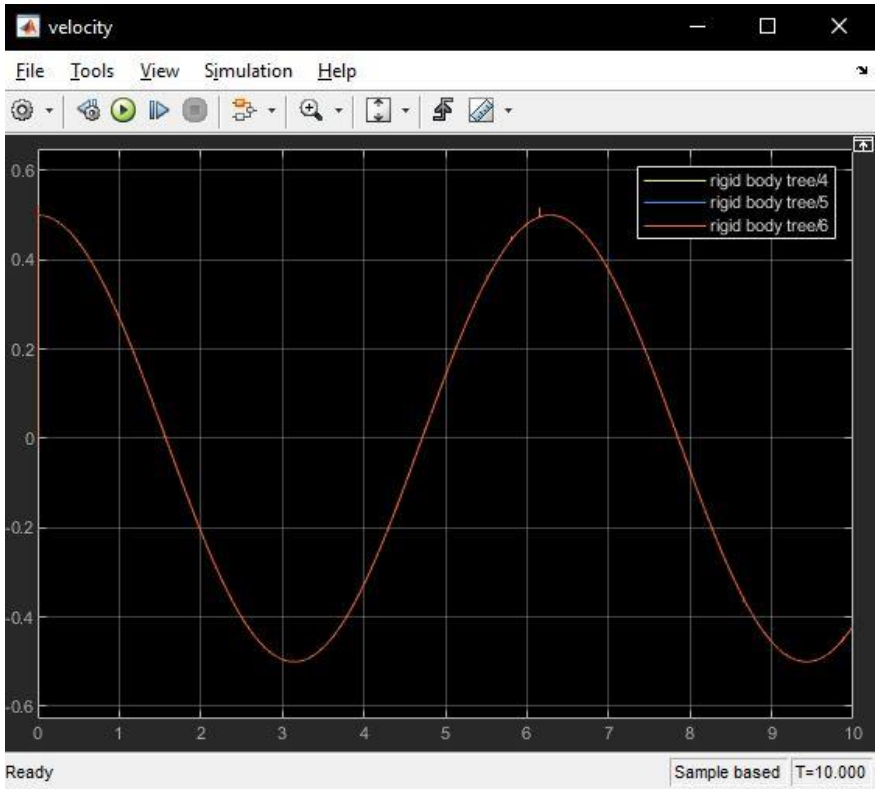

Figure 42: Result for velocity

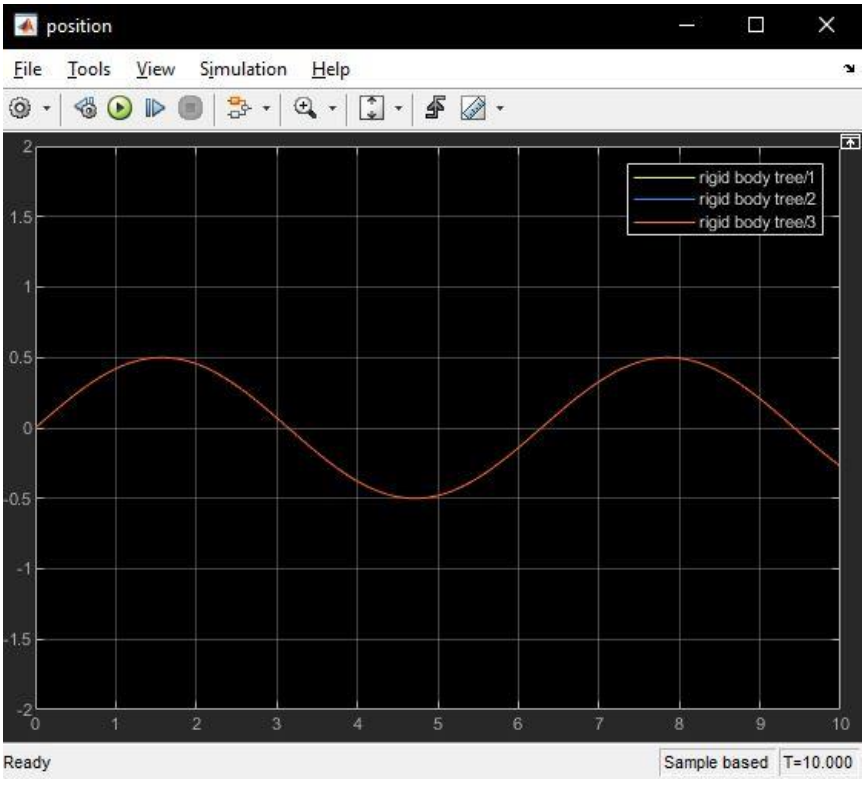

Figure 43: Result for position

2.1.3.6 Step input PD controller

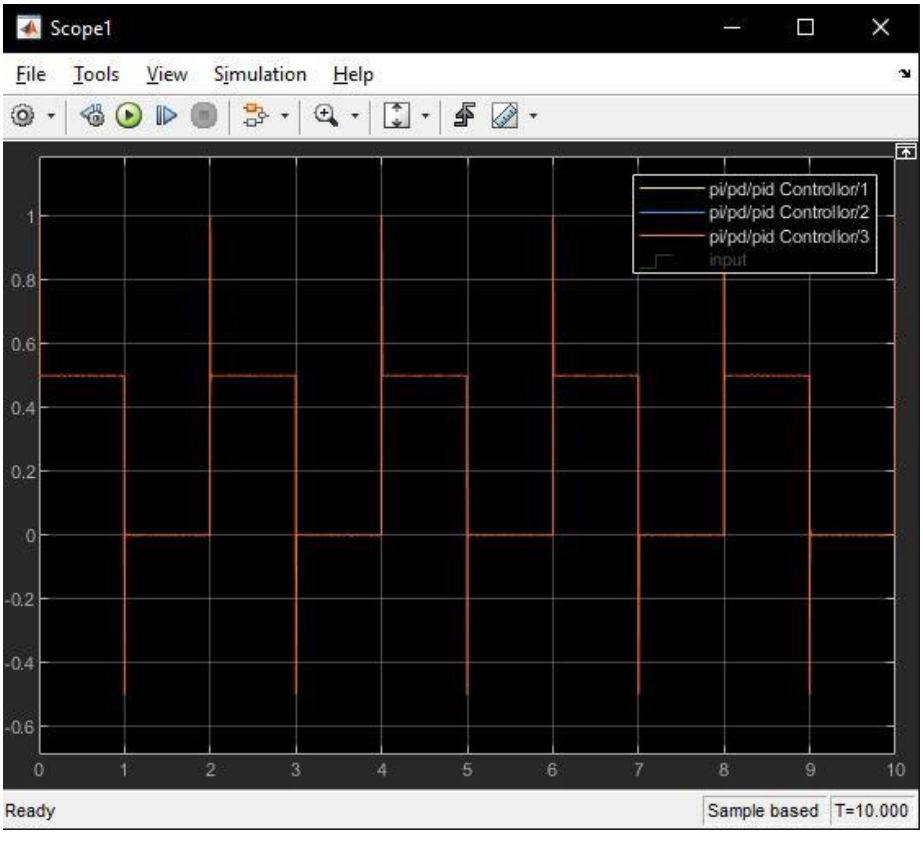

Figure 44: controller response

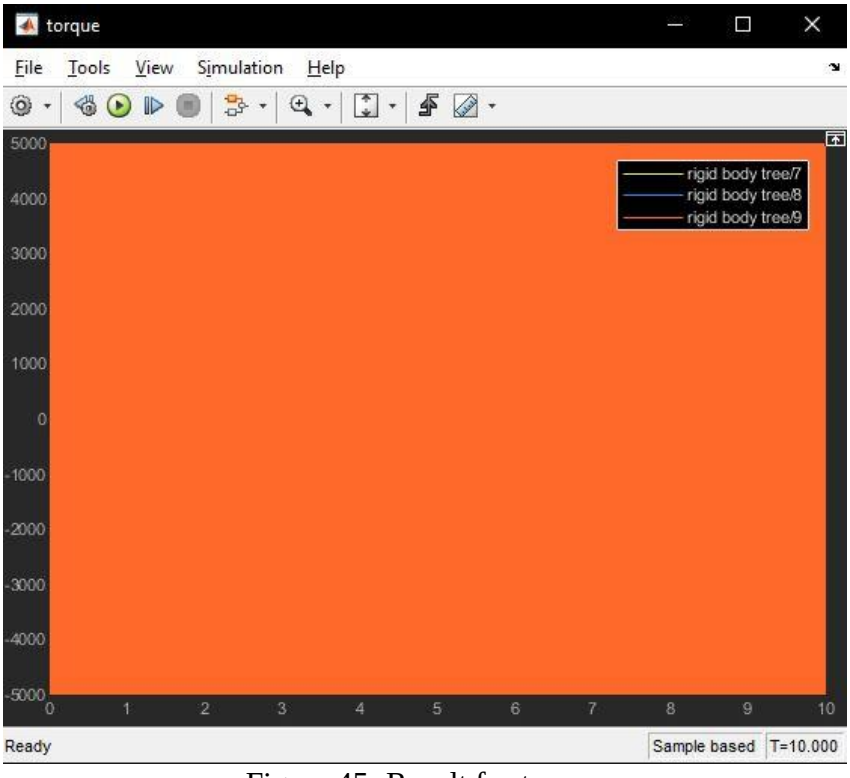

Figure 45: Result for torque

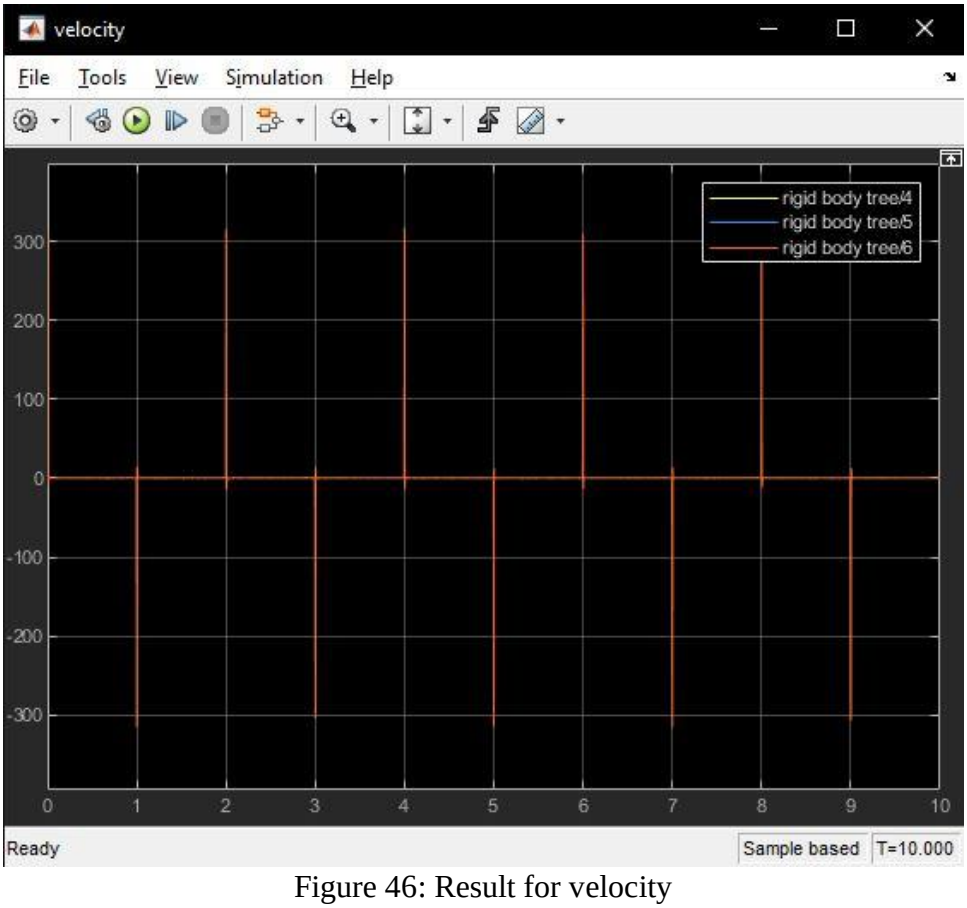

Figure 46: Result for velocity

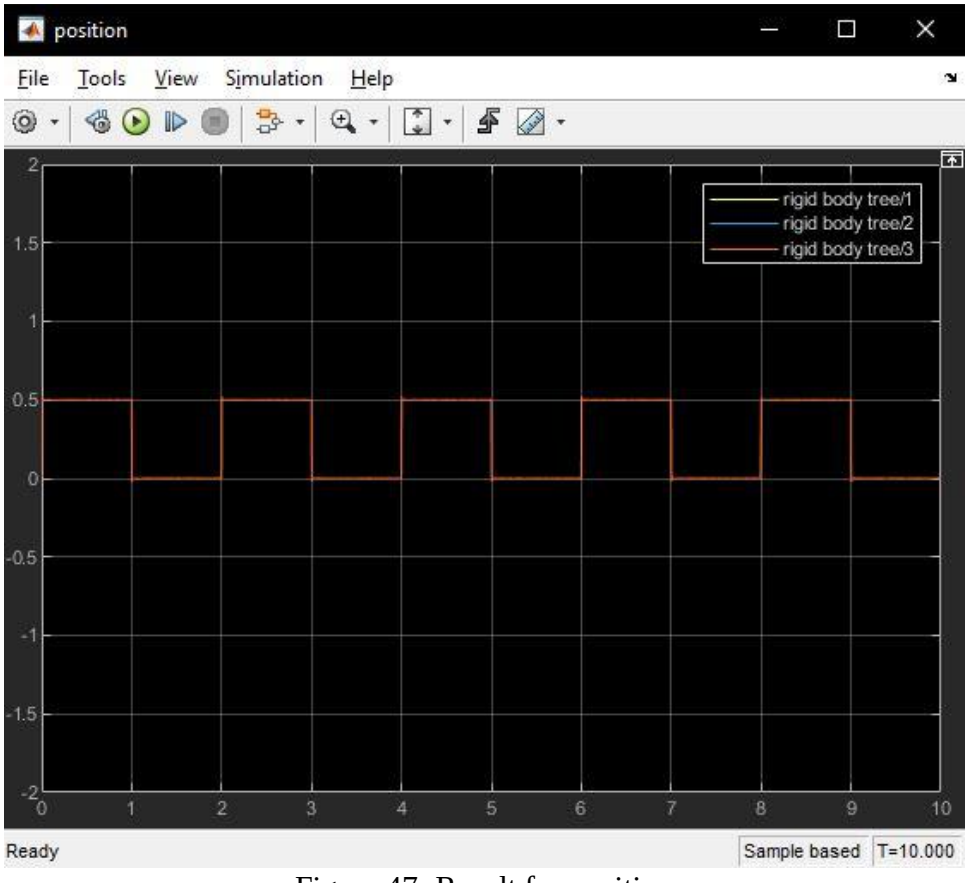

Figure 47: Result for position

2.1.4 Discussion

In this part, we successfully control the velocity not to be so fast and, torque does not vary in a large range. But we did not successfully control the torque under 100. However, that is the best result that we can find. For the result we noticed that whatever the input is the torque is not varying in a large range for PI and PID controllers which are the two that engineers mostly use But for PD controllers there are much error in the result, it is even worse in the step input result, we think is due to all the signal that output from controller is based on prediction and noises influencing the result, it matches our research that PD controller has lower accuracy than other controllers. And the best controller is the PID since from the result it is the most accurate.

2.2 Task 2 Trajectory Tracking

In this task we came up with two methods to Track the trajectory. One is using the block that is provided in the Simulink library (Inverse Kinematics, Polynomial Trajectory and Coordinate Transformation Conversion). But in this method, we need to import a NxP matrix which confused us a lot, since we don't have enough time to understand how does it work we came up with another solution which using the geometric method solve for angles of each joint. This method also has a problem which cannot determine the effect from the weight for each link.

2.2.1 Geometric Method

Based on the lectures, we are not quite sure how to solve it using the end point. Instead of that we can slove it once we know the end position of joint 3 and the total angle that moved.

2.2.1.1 Block Diagram

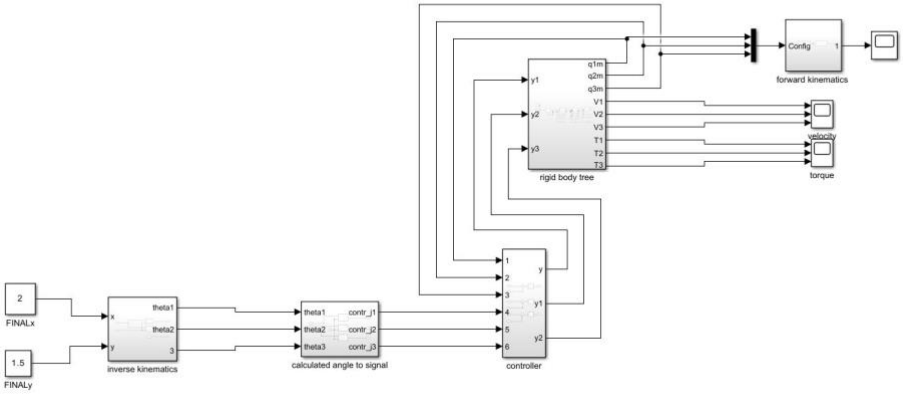

Figure 48: Block diagram for trajectory

We are doing it in several separate parts which are calculation for inverse kinematics, calculated angle to signal, controller, rigid body tree, forward kinematics, and data collection. We set the joint 3 final position at x=2 and y=1.5. Since the rigid body tree will be the same as the previous task so we will not discuss that in this section.

2.2.1.2 Inverse kinematics

Since it will be the same if we provide the total angle that moved or the third joint angle, therefore, to reduce the work (which can reduce a calculation step) we choose to provide theta 3 at 60 degree which is around 1.047 rad. The block diagram is shown as follow:

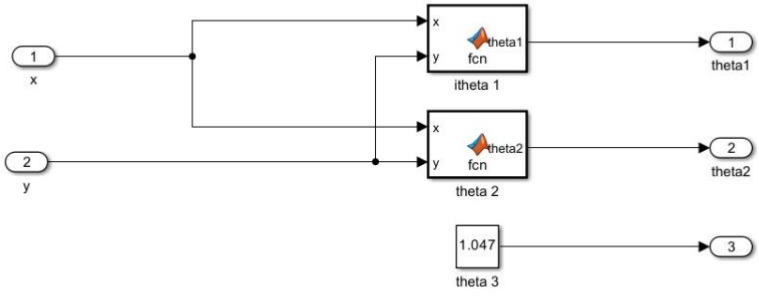

Figure 49: Inverse Kinematics Block

Then we need to calculate the value of theta 1 based on the lecture we need to find beta and psi first the calculation block is shown as follow:

```
function theta1 = fcn(x,y)
l1 = 2;
l2 = 1;
l3 = 1;

beta = atan2(y,x);
psi = acos((x^2+y^2+l1^2-l2^2)/(2*l1*sqrt(x^2+y^2)));
theta1 = beta +psi;

```

Figure 50: Calculation for theta 1

We also need to be using the function to find theta 2, and it will be calculated as follow:

```
function theta2 = fcn(x,y)
l1 = 2;
l2 = 1;
l3 = 1;

theta2 = acos((x^2+y^2-l1^2-l2^2)/(2*l1*l2));
```

Figure 51: Calculation for theta 2

2.2.1.3 Angle to signal

Since the description says we need to implement a sine wave as input, it is not clarified what is that input for, so we decided to when the wave reaches the peak, the joint will move to the set angle. The design is shown as follows:

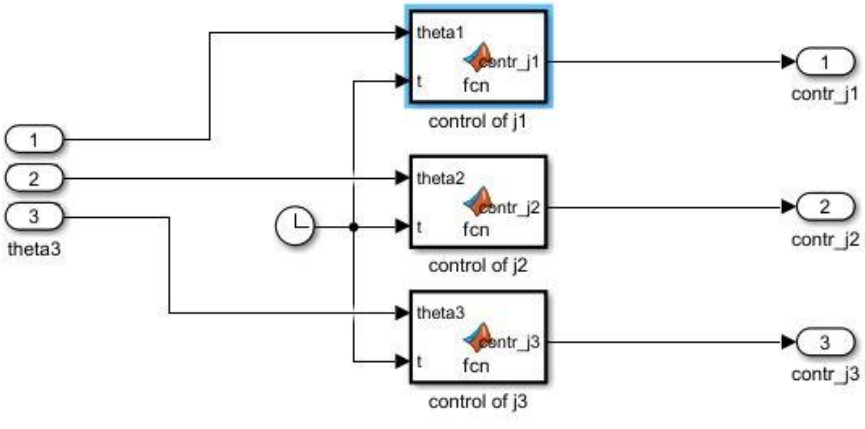

Figure 52: Angle to signal

We also implement a clock which is used to make the graph oscillate according to time. The sine wave won't give the signal that we want so we change it to cosine wave which can reduce the work for phase, and the function and signal output are shown below:

```
function contr_j1 = fcn(theta1, t)

contr_j1 = theta1*(1-cos(pi*t))/2;
```

Figure 53: Function that used to transform the angle to signal.

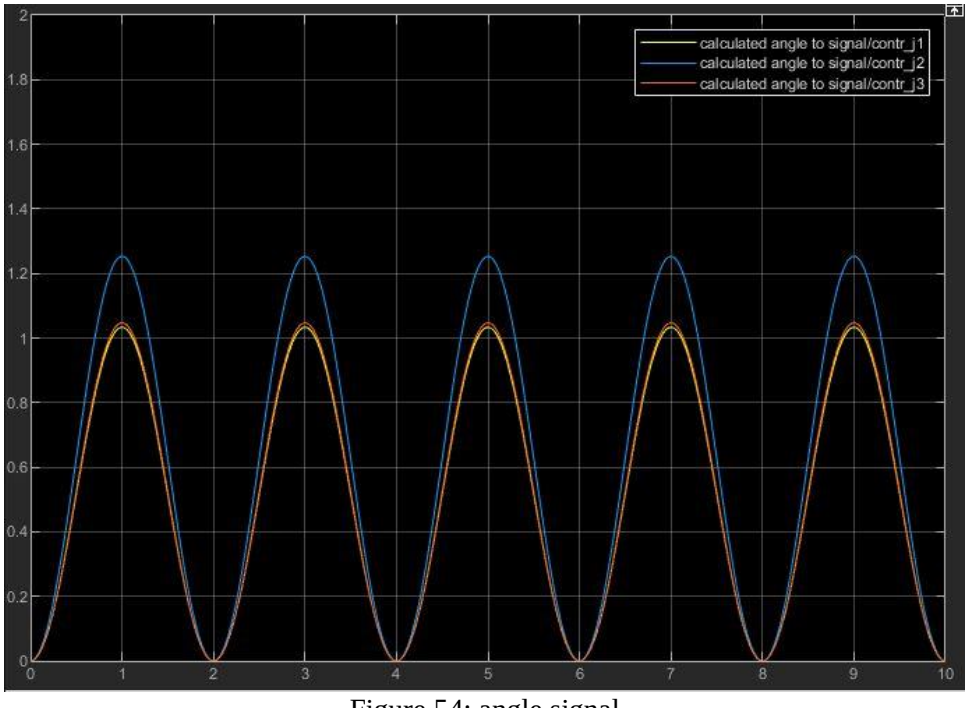

Figure 54: angle signal

2.2.1.4 Controller

For the controller, since PID is the best controller as we discussed in task1, so we are using it in this part. We wanted to use the same one as task one, but we don't know why it doesn't work at all, so we tried another method which the one doesn't have the feedback and we add it in. Moreover, the angle signal has already controlled the speed of action, so we leave it as original condition. There is another reason that we were not change the parameter that is we don't have enough time to adjust it to a motion that looks good, and the presetting is good enough. The block is shown below:

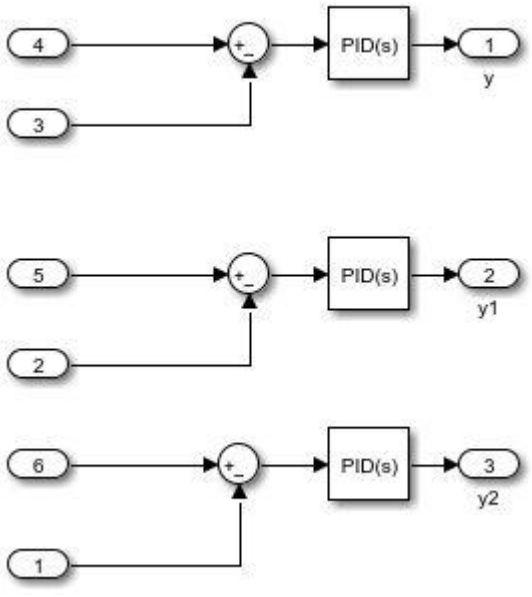

Figure 55: controller

2.2.1.5 Forward kinematics

In this part we are going to convert the position to a specified coordinate transformation representation we also want to implement this to task one but unfortunately, we don't have time to do that. There are two blocks used in this part which are get transformation and coordinate transformation conversion. This first one is used to find the transformation, we need to link that to the rigid body tree and define the source body (base) and target body (end effect). And the coordinate transformation conversion is used to cover the transformation to vectors which is a x y z matrix. The block diagram is shown as follow:

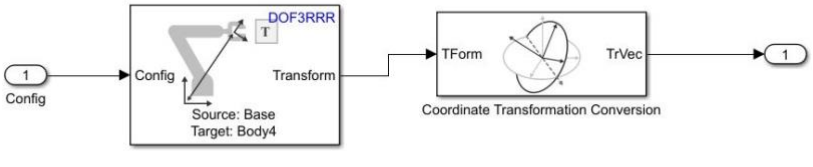

Figure 56: forward kinematics

2.2.2 Result

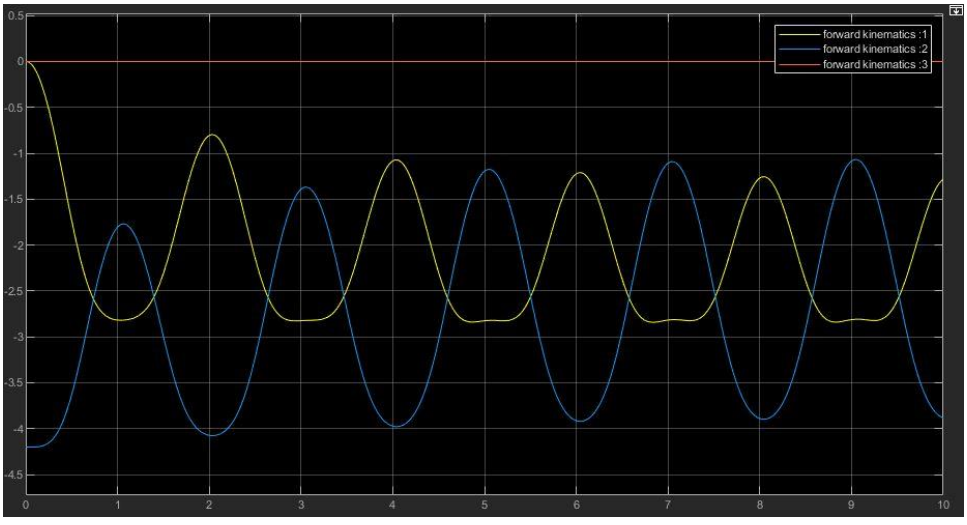

Figure 57: forward kinematics result

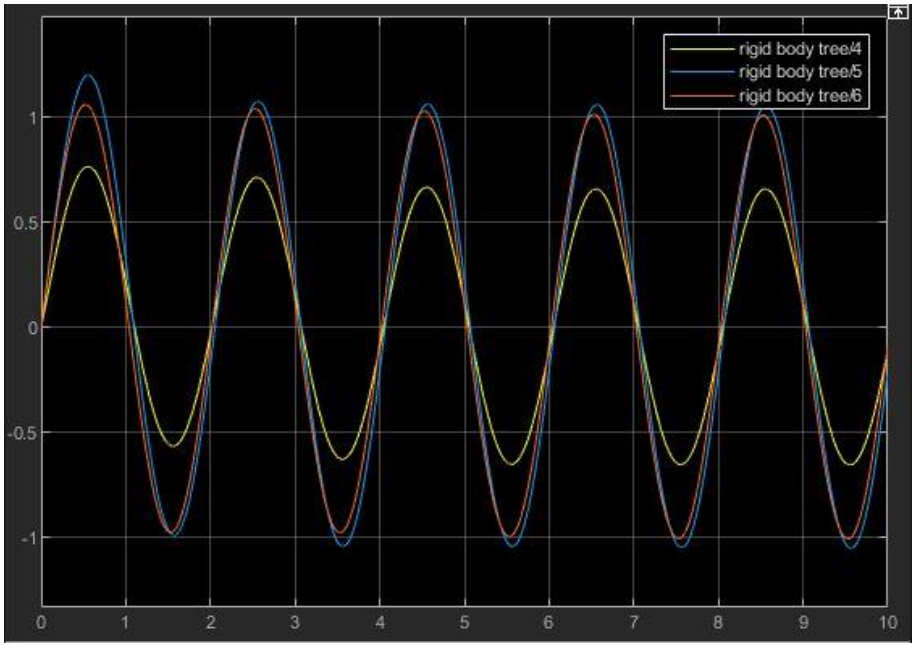

Figure 58: velocity

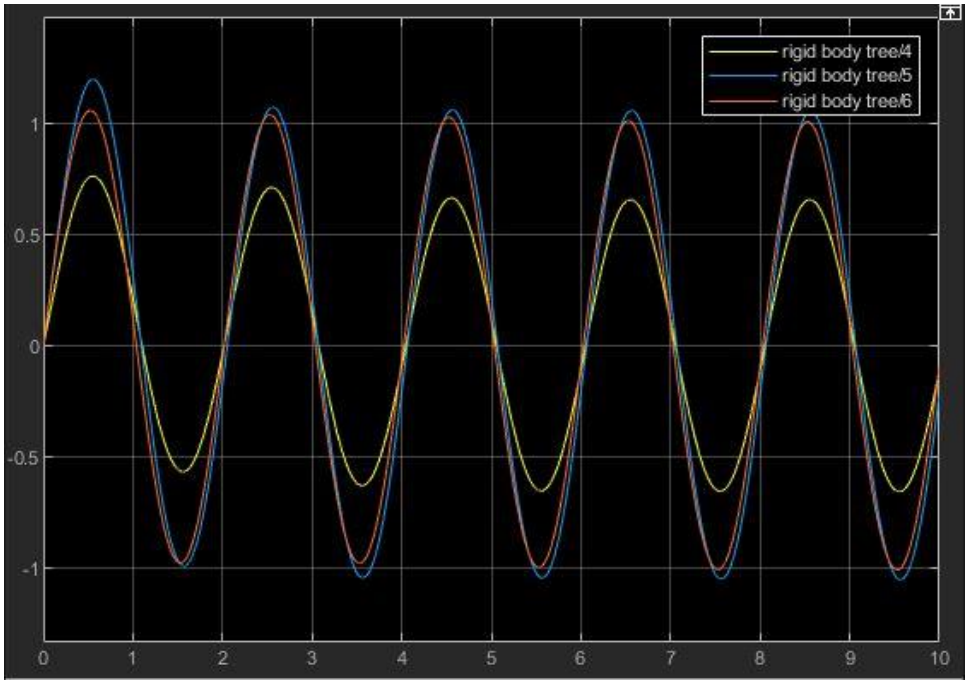

Figure 59: torque

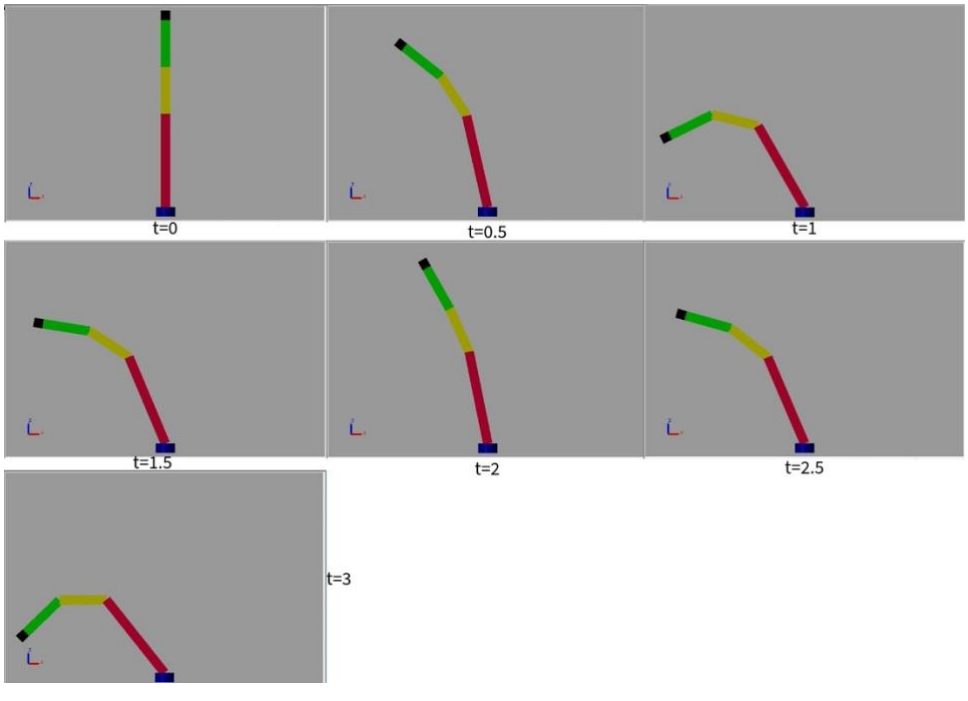

Figure 60: Position change in range time = 0 ~ 3

2.2.4 Discussion

In this task we successfully make the end effect move to the position that we want, to varify that we measured it by hand it shows as follow:

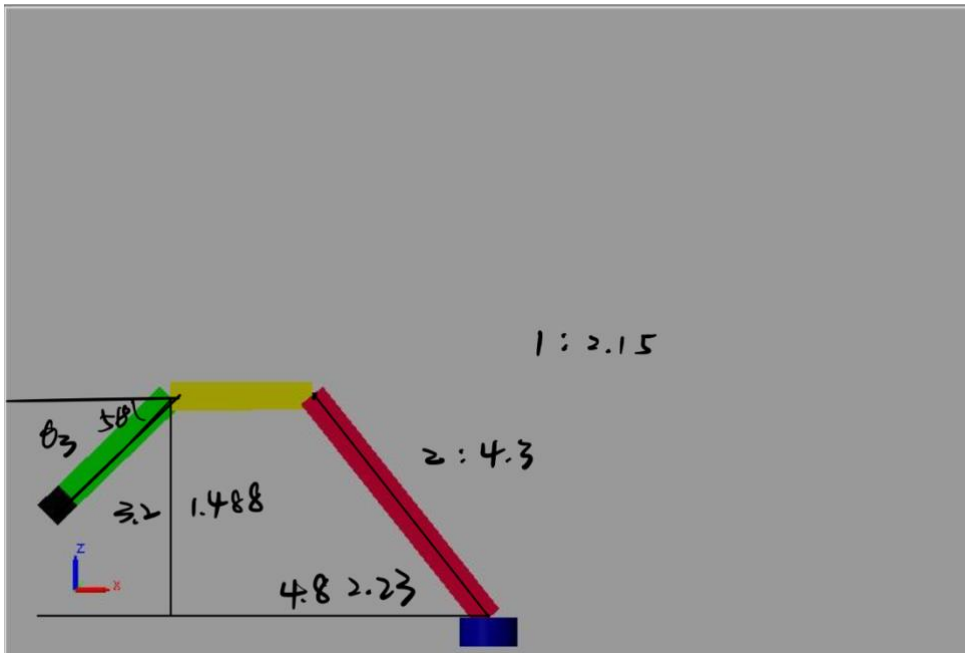

Figure 61: Verification of position when t = 3

From the simulation when t = 3 is close to the position that we set so we used that to verify。 We can see when we use the length of link one as our legend since we set link 1 at 2meter then we can find the position of joint 3 which is x = 1.488m y=2.23 and theta 3 = 56°. It is not that accurate since we are measuring by hand, however it is close. It proves that our simulation is successful. We also can see that the result from forward kinematics looks good, but we cannot use that to verify our result since it is measuring the end effort position, and we are setting the joint 3 position. Moreover, we noticed that after few curves the graph goes to stable, we think it is due to the control of the PID controller. We verified that by remove the controller and the forward kinematics look like the following graph:

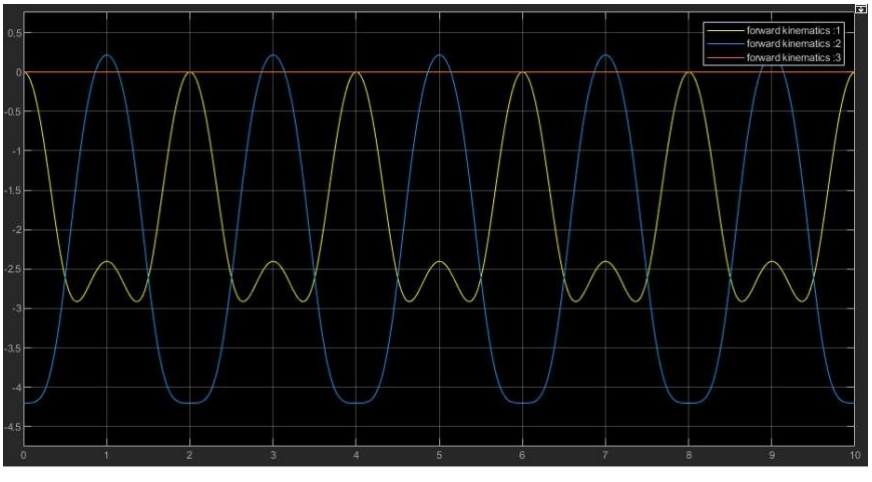

Figure 62: forward kinematics without PID controller

From the graph we can see it is the same as what we thought, but we don't know how without the controller the motion is not same as previous, but we know it is due to the removal. Other than that, the velocity is pretty good, but the torque is not good enough. To make the torque more stable we need to adjust the PID controller react in a longer period and extend the period of the sine wave, furthermore, to make the body that can reach the position we also need to adjust the period of simulation. Overall, this simulation successfully served its purpose.

3. Discussion Question Two

In this section we estimate the position of an autonomous vehicle using dead reckoning navigation (inertial sensors and odometry) and sonar sensors. Sensor fusion techniques to be demonstrated include the Non-Weighted Extended Kalman Filter (EKF), Weighted Extended Kalman Filter (WEKF) and Unscented Kalman Filter (UKF).

> *Sensor fusion* is a crucial aspect of autonomous vehicle navigation, integrating data from multiple sensors to improve accuracy and reliability. In the context of estimating vehicle position, sensor fusion combines measurements from inertial sensors (accelerometers and gyroscopes) and odometry with sonar sensor data.
>
> *Dead reckoning* is the process of estimating the vehicle's position based on its previous known position and the data from inertial sensors (accelerometers and gyroscopes).
>
> *Inertial sensors* provide information about the vehicle's acceleration and angular rate, allowing for the calculation of changes in position and orientation over time.
>
> *Sonar sensors* are used to measure distances to nearby objects, providing information about the vehicle's surroundings. Sonar data is valuable for detecting obstacles and refining position estimates, especially in scenarios where GPS signals may be unreliable or unavailable.

The three sensor fusion loop methods follow a similar algorithm:

1. Initialization
   a. Begin by setting the initial position and velocity of the vehicle using the provided dead reckoning measurements, obtained from either inertial sensors or odometry.

   b. Establish the initial state of the system by initializing the covariance matrices for processing noise, measurement noise, and state variables.

   c. Configure the sonar sensor by defining its measurement characteristics and specifying the desired level of positioning accuracy.

2. Prediction

3. Update

3.1 Define the robot system model

The non-linear kinematics model is represented by the following equations:

$$\dot{x}(t) = v(t)cos(\theta(t)) + wx$$
$$\dot{y}(t) = v(t)sin(\theta(t)) + wy$$
$$\theta(t) = \omega(t) + w_\theta$$

State of the system:

$X = [x, y, \theta]^T$

Input to the system:

$U = [v, \omega]^T$

White Gaussian process noise:

$w = [w_x, w_y, w_\theta]^T$

$w \sim N(0, Q)$

Odometry sensor equation:

$Y = CX + n = [100|010|100] + n$

White Gaussian sensor noise:

$n \sim N(0,R)$.

3.2 Instantaneous Center of Curvature

After some research we chose to derive the linear and angular velocities as well as the angle theta based on the ICC - Instantaneous Center of Curvature.

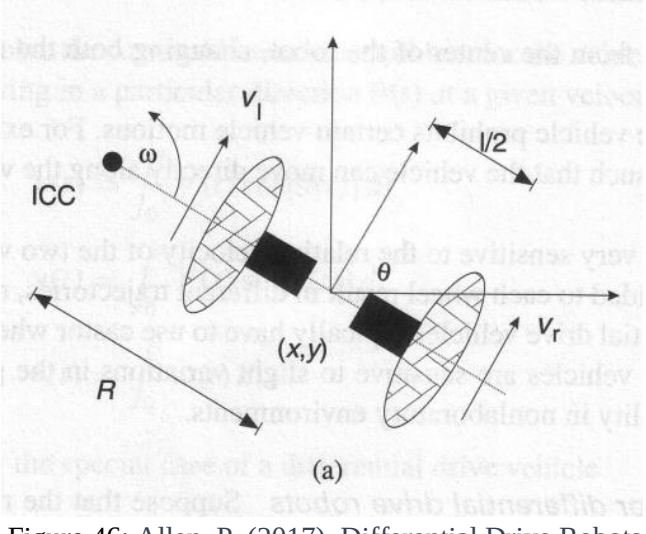

Figure 46: Allen, P. (2017). Differential Drive Robots

Our system inputs were d1(left wheel distance traveled), d2 (right wheel distance traveled) and L (distance between wheels). These are used to determine the following:

vr=d1/dt (left wheel velocity)
vl=d2/dt (right wheel velocity)
v= (vr+vl) / 2 (linear velocity)
w= (vr-vl) / L = dθ/dt (angular velocity)

3.3 EKF – Extended Kalman Filter Implementation

The general Extended Kalman Filter 'predict-update' algorithm for non-linear models is demonstrated below. This model was obtained from the lecture module on Kalman Filters. The system is initialized with estimates for the state vector ($\hat{X}$) and the covariance matrix (P). After this it cycles between 'predicting' new values for $\hat{X}$ and P based on the previous inputs, and 'updating' these values using the Kalman gain (K) and measurement model (Z).

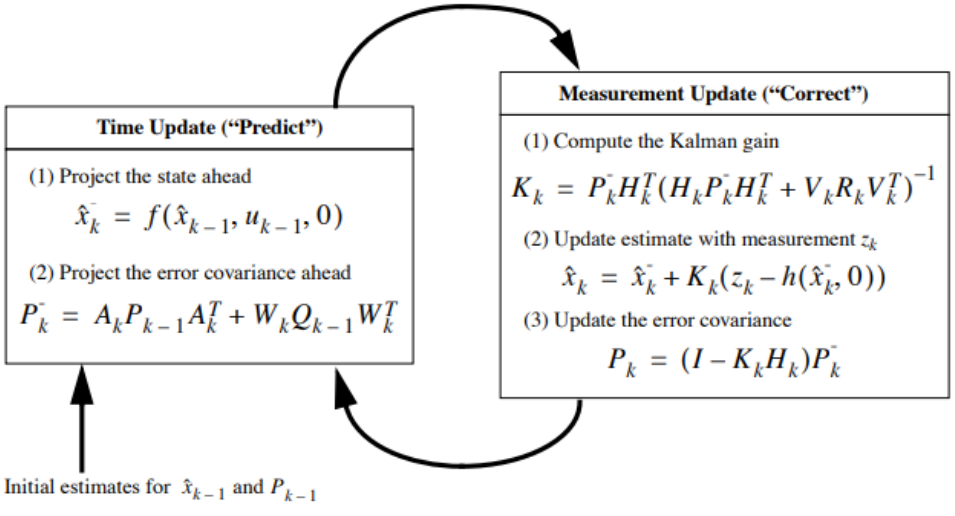

Figure 63: Extended Kalman Filter Algorithm Welch, G., & Bishop, G., An Introduction to the Kalman Filter

In our initial attempt to simulate the Kalman Filter on an autonomous vehicle navigation system that uses dead reckoning, we followed the derived version of this mathematical model presented in section 3.3 of the 'Sensor Fusion' reading.

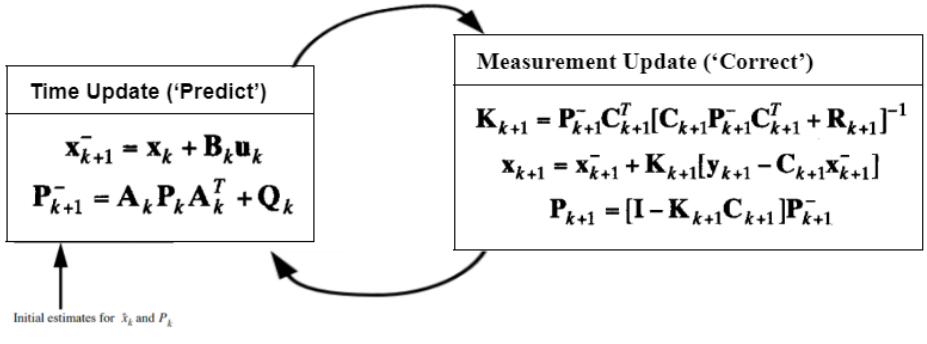

Figure 64: EKF Sensor Fusion for Dead Reckoning Mobile Robot Navigation, Sasiadek, J. Z., 'Sensor Fusion' Section 3.3

It is important to note some key differences between the more general and derived sets of equations. The Jacobian matrices used to linearize the EKF model are represented by A, H, V, and W in the first flowchart. Their equations are shown below for reference.

$$A_{[i,\,j]} = \frac{\partial f_{[i]}}{\partial x_{[j]}}(\hat{x}_{k-1}, u_{k-1}, 0)$$

$$W_{[i,\,j]} = \frac{\partial f_{[i]}}{\partial w_{[j]}}(\hat{x}_{k-1}, u_{k-1}, 0)$$

$$H_{[i,\,j]} = \frac{\partial h_{[i]}}{\partial x_{[j]}}(\tilde{x}_{k}, 0)$$

$$V_{[i,\,j]} = \frac{\partial h_{[i]}}{\partial v_{[j]}}(\tilde{x}_{k}, 0)$$

Where the true state and measurement vectors relate to the functions *f* and *h* taking for parameters the state vector ($x=[x,y,\theta]^T=[v(t)\cos(\theta(t))+wx,\ v(t)\sin(\theta(t))+wy,\ w(t)+w\theta]^T$), control vector, which in our case are the angular and linear velocity inputs, ($u=[u,v]^T$), the Gaussian process noise ($w \sim N(0,Q)$), and Gaussian sensor noise ($v \sim N(0,R)$).

$$x_k = f(x_{k-1}, u_{k-1}, w_{k-1})$$

$$z_k = h(x_k, v_k)$$

In the Jacobian matrices the Gauss noise values are set to zero since the individual values are not known for each step and thus approximations must be done without them.

In the second derived model the Jacobian matrices are shown below for B and A.

$$\mathbf{B}_k = \begin{bmatrix} T\cos\left(\theta_k + \frac{\Delta\theta_k}{2}\right) & 0 \\ T\sin\left(\theta_k + \frac{\Delta\theta_k}{2}\right) & 0 \\ 0 & 1 \end{bmatrix}$$

$$\mathbf{A}_k = \begin{bmatrix} 1 & 0 & -v_k T\sin\theta_k \\ 0 & 1 & v_k T\cos\theta_k \\ 0 & 0 & 1 \end{bmatrix}$$

To simplify the model, it is assumed that the process and sensor noise are zero-mean, and approximations can be made without the w and v values. Thus, if the measurement model relating to the h(x,u,v) function is as follows and v is ignored then h(x,u,0) holds the same values as f(x,u,0) in a 3x3 diagonal matrix as opposed to a 3x1. We then assumed that the Jacobian matrix for C was equal to that of A:

Y = CX+v = [100;010;001]X + v (from problem equations)

Additionally, the second model does not show the W and V Jacobian matrices since the W and V matrices are equal to the identity matrix and thus if Q and R (from the Gause noise equations) are both 3x3 matrices with constants on the diagonal, the V and W matrices will have no effect on them.

Using the above mathematical equations and assumptions above the following Matlab code was used for the simulation where R, Q, and the angular and linear velocities are all measured as constants to simplify the simulation for more readable results. (See Appendix for MATLAB code)

Through the for loop the graphs points were plotted as the x(k) against y(k) of the state vector X on each iteration. This plot shows the estimated (x,y) coordinates of the vehicle at time k, demonstrating the path taken by the robot. Results obtained either followed a curved to linear or circular trajectory.

In the first curved graph, d1=d2, therefore both wheels are moving at the same velocity and the robot has no angular velocity. Smaller R values increased the linearity of the graph. Meanwhile smaller Q values resulted in a more curved graph.

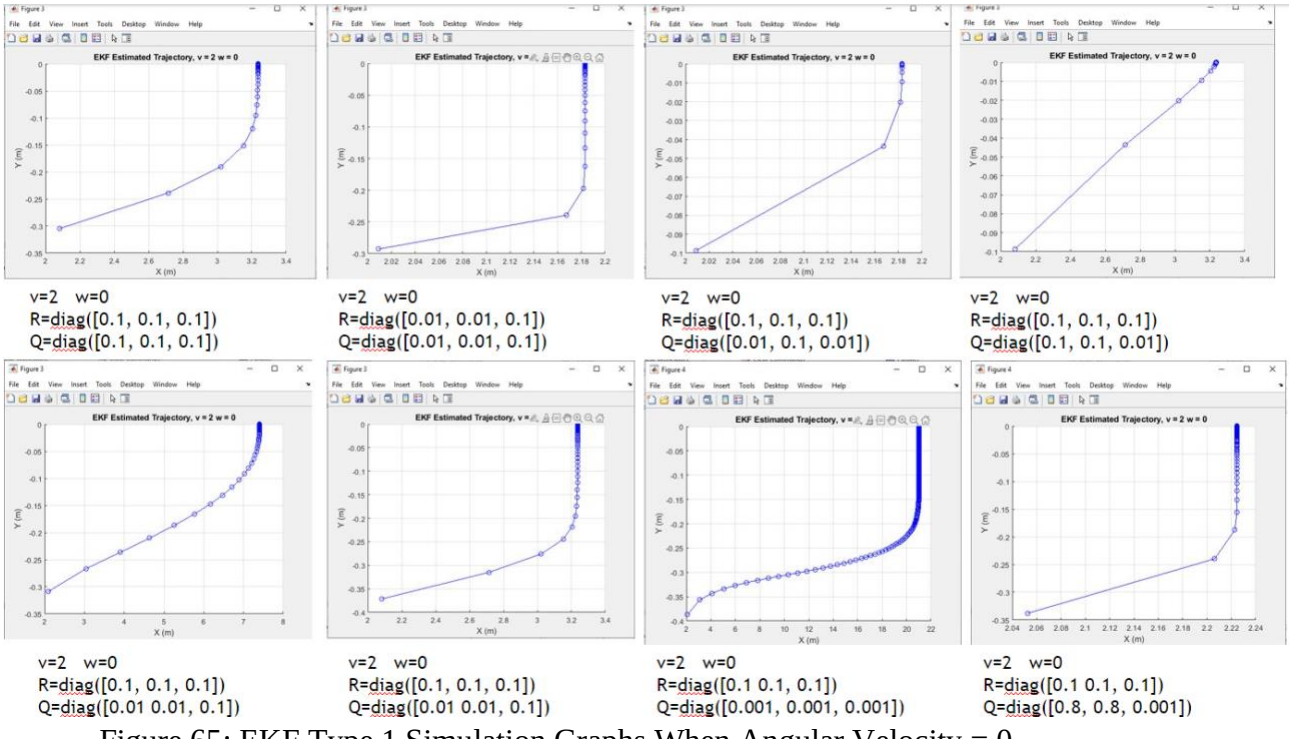

Figure 65: EKF Type 1 Simulation Graphs When Angular Velocity = 0

In the second circular graph the following variable effects on the graph were observed:

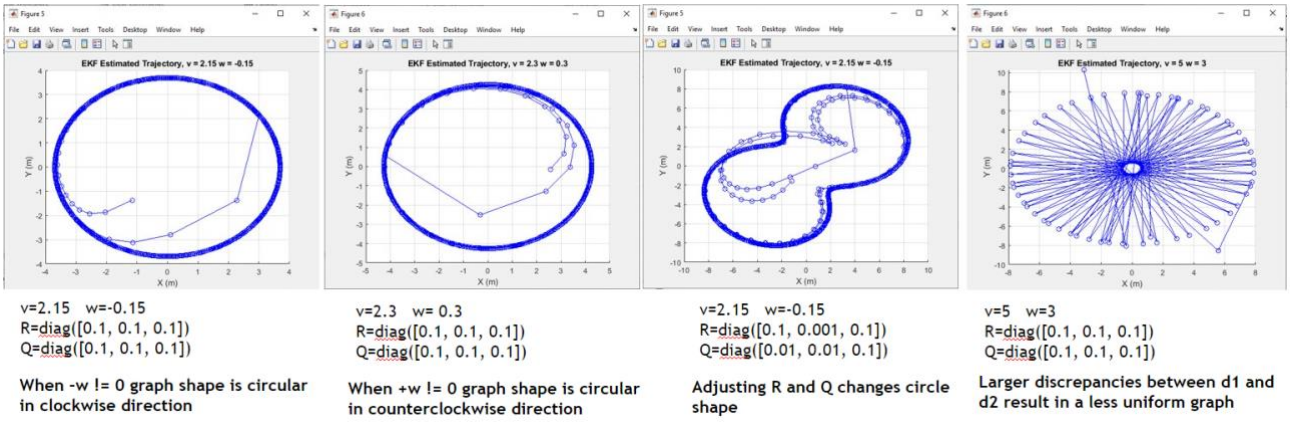

Figure 66: EKF Type 2 Simulation Graphs When Angular Velocity != 0

The covariance matrices Q and R represent the system's noise and as these values increase, the more skewed the results become from the actual path. Angular velocity affects the graph's direction and the degree at which the robot turns (figure 50).

3.4 WEKF - Weighted Extended Kalman Filter Implementation

The Weighted Extended Kalman Filter (WEKF) is a more sophisticated variant of the standard Extended Kalman Filter (EKF). It's designed to offer better accuracy, particularly in systems characterized by significant non-linear dynamics or uncertain measurement and process models.

The mathematical equations used in the algorithm:

1. State Estimation Problem

The goal is to estimate the state x_k of the sytem over time

Inputs:

x_k state vector at time k

u_k control input at time k

y_k Measurment at time k

2. Weighted Extended Kalman Filter

1. Creating Weighted matrices

Equation Used:

$$\mathbf{Q_k} = \mathbf{Q_a} . \mathbf{a^{-2(k+1)}}$$

$$\mathbf{R_k} = \mathbf{R_a} . \mathbf{a^{-2(k+1)}}$$

Where Q_a, R_a: Base noise covariance matrices

a: weighting factor

k: time step

2.WKEF Step

- state update: $\boldsymbol{x_{k|k-1} = A_k . x_{k-1} + B_k . u_k}$
- Covariance update: $\boldsymbol{P_{k|k-1} = A_k . P_{k-1} . A_k^T + Q_k}$
- Kalman Gain: $\boldsymbol{K_k = P_{k|k-1} . C_k^T . (C_k . P_{k|k-1} . C_k^T + R_k)^{-1}}$
- State Estimate Update: $\boldsymbol{x_k = x_{k|k-1} + K_k . (y_k - C_k . x_{k|k-1})}$
- Covariance Estimate Update: $\boldsymbol{P_k = (I - K_k . C_k) . P_{k|k-1}}$

3. Implementation Steps

1.initialize the state vector x_k , estimation covariance P_k, and other parameter

2. update the state and the covariance estimates using the WEKF equations

3. Plot the results and then simulate

The code for the WEKF

```
import numpy as np
import matplotlib.pyplot as plt

def create_weighted_matrices(Qa, Ra, a, k):
    """ Create weighted noise covariance matrices """
    Qk = Qa * a ** (-2 * (k + 1))
    Rk = Ra * a ** (-2 * (k + 1))
    return Qk, Rk

def wekf_step(x_k, P_k, u_k, y_k, A_k, B_k, C_k, Qk, Rk):
    """ Perform one step of the Weighted Extended Kalman Filter """
    # State Update
    x_k1_k = np.dot(A_k, x_k) + np.dot(B_k, u_k)

    # Covariance Update
    P_k1_k = np.dot(A_k, np.dot(P_k, A_k.T)) + Qk

    # Kalman Gain
    S_k1 = np.dot(C_k, np.dot(P_k1_k, C_k.T)) + Rk
    K_k1 = np.dot(P_k1_k, np.dot(C_k.T, np.linalg.inv(S_k1)))

    # State Estimate Update
    y_k1_pred = np.dot(C_k, x_k1_k)
    x_k1 = x_k1_k + np.dot(K_k1, (y_k - y_k1_pred))

    # Covariance Estimate Update
    I = np.eye(len(x_k))
    P_k1 = np.dot((I - np.dot(K_k1, C_k)), P_k1_k)

    return x_k1, P_k1
# Adjusted Q and R Values
Qa = np.diag([0.2, 0.2, 0.1])  # base process noise covariance
Ra = np.diag([0.3, 0.3, 0.15]) # base measurement noise covariance

initial state vector
```

```
x_k = np.array([1, 1, 0.5])  # Initial state [x, y, theta]

initial estimation covariance
P_k = np.diag([0, 0.1, np.pi/180])  # Initial estimation covariance

# Other parameters
A_k = np.eye(3)  # State-transition model
B_k = np.eye(3)  # Control-input model
C_k = np.eye(3)  # Measurement model
Qa = np.eye(3)   # Base process noise covariance
Ra = np.eye(3)   # Base measurement noise covariance
a = 1.2          # Weighting factor
k = 0            # Time step

# Trajectory data storage
estimated_trajectory = [x_k]
expected_trajectory = [x_k]

# Simulation steps
for step in range(20):
    # Assume some control input (linear and angular velocity)
    u_k = np.array([0.01, 0.04, np.pi/180 * 5])  # Move forward and turn slightly
    expected_state = x_k + u_k  # Expected state without noise

    # Add noise to simulate real measurements
    y_k = expected_state + np.random.normal(0, 0.1, 3)

    # Create weighted matrices
    Qk, Rk = create_weighted_matrices(Qa, Ra, a, k)

    # Perform a WEKF step
    x_k, P_k = wekf_step(x_k, P_k, u_k, y_k, A_k, B_k, C_k, Qk, Rk)

    # Store trajectories
    estimated_trajectory.append(x_k)
    expected_trajectory.append(expected_state)

    # Increment time step
```

```
    k += 1

# Plotting
estimated_trajectory = np.array(estimated_trajectory)
expected_trajectory = np.array(expected_trajectory)

plt.figure(figsize=(10, 6))
plt.plot(estimated_trajectory[:, 0], estimated_trajectory[:, 1], label='Estimated Trajectory')
plt.plot(expected_trajectory[:, 0], expected_trajectory[:, 1], label='Expected Trajectory', linestyle='--')
plt.scatter(x_k[0], x_k[1], color='red', label='Final Estimated Position')
plt.xlabel('X Position')
plt.ylabel('Y Position')
plt.title('Robot Trajectory - Estimated vs Expected')
plt.legend()
plt.grid(True)
plt.show()
```

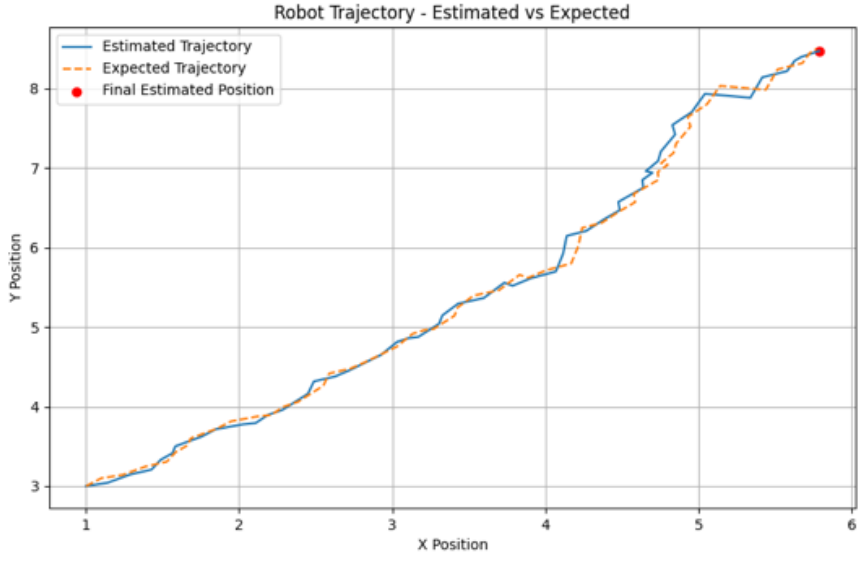

Figure 67: WEKF Simulation I Graph

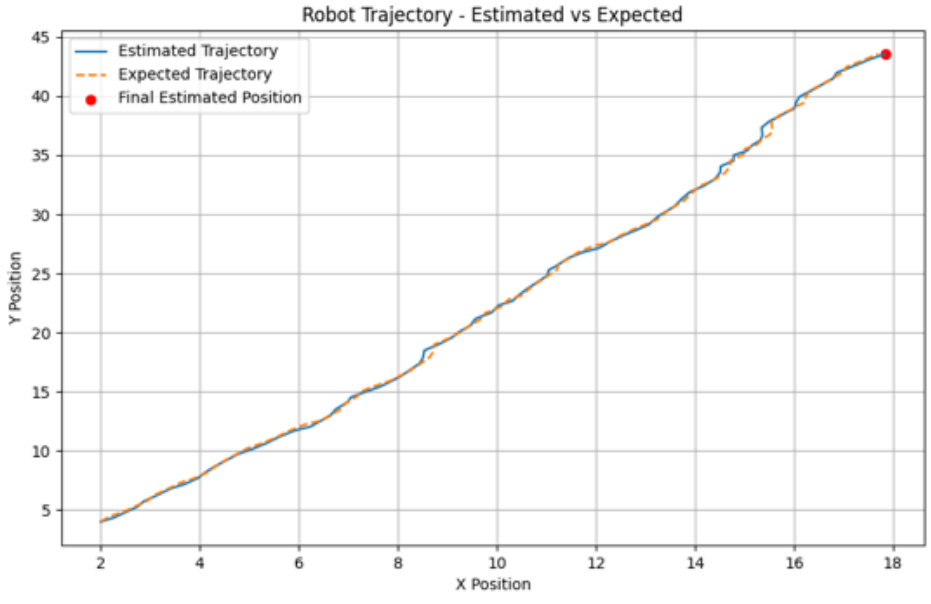

Figure 68: WEKF Simulation II Graph

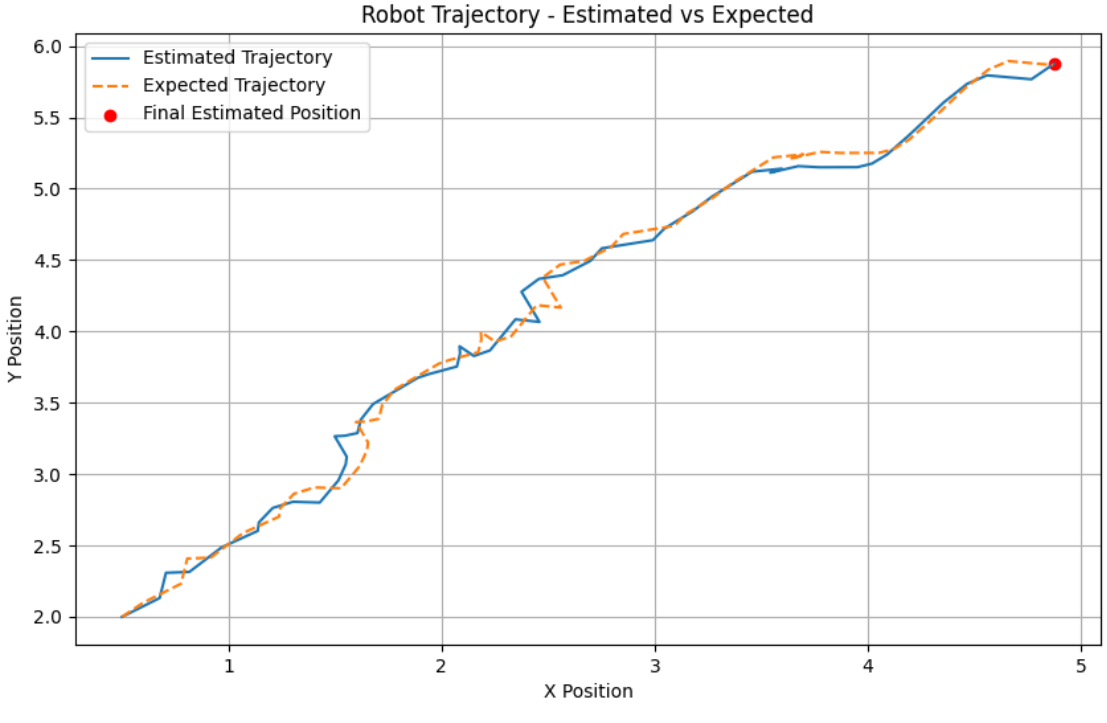

Figure 69: WEKF Simulation Graph when $\theta = 90^o$

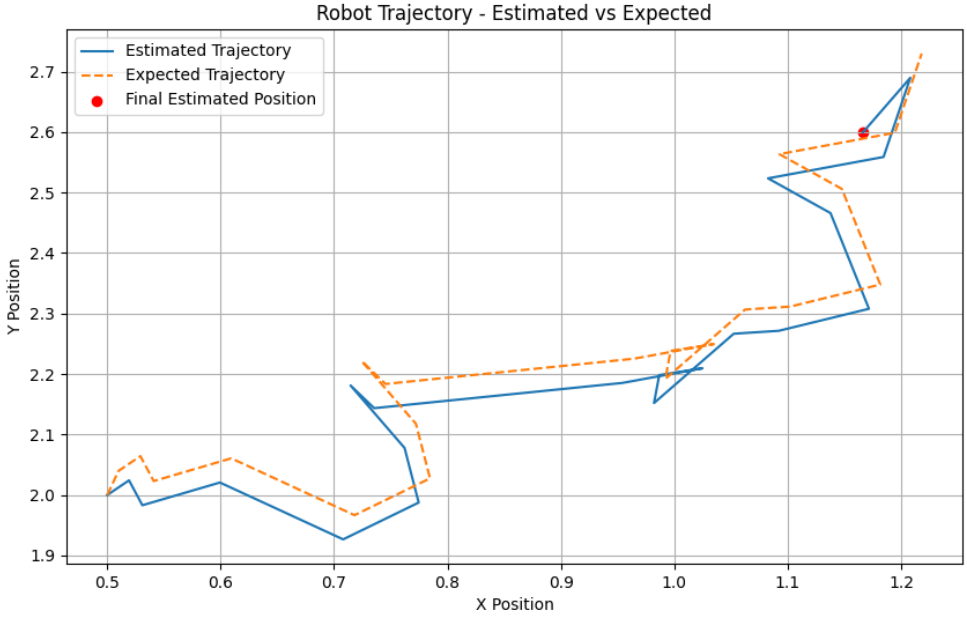

Figure 70: WEKF Simulation Graph when Qk and Rk changed

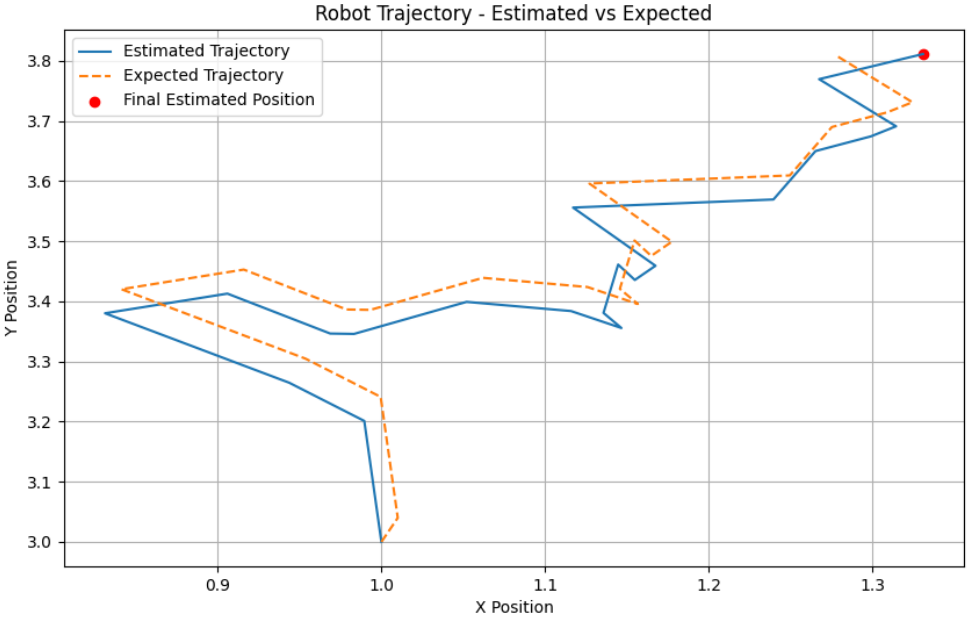

Figure 71: WEKF Simulation Graph when the initial state change to [1,3,0.5]

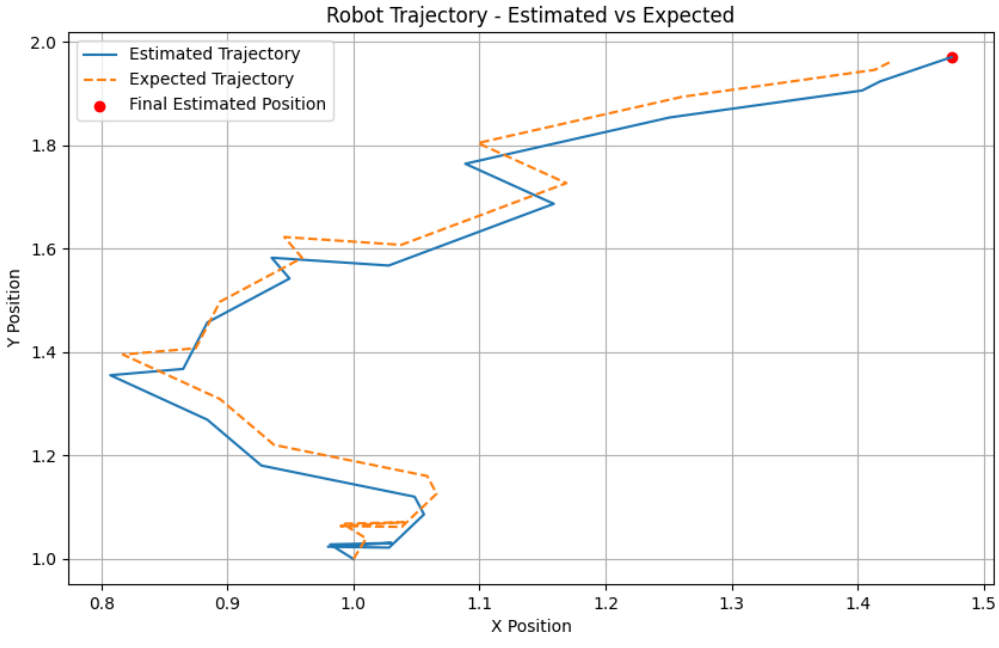

Figure 72: WEKF Simulation Graph when the Qk = [0.2, 0.2, 0.1] and Rk = [0.3, 0.3, 0.15]

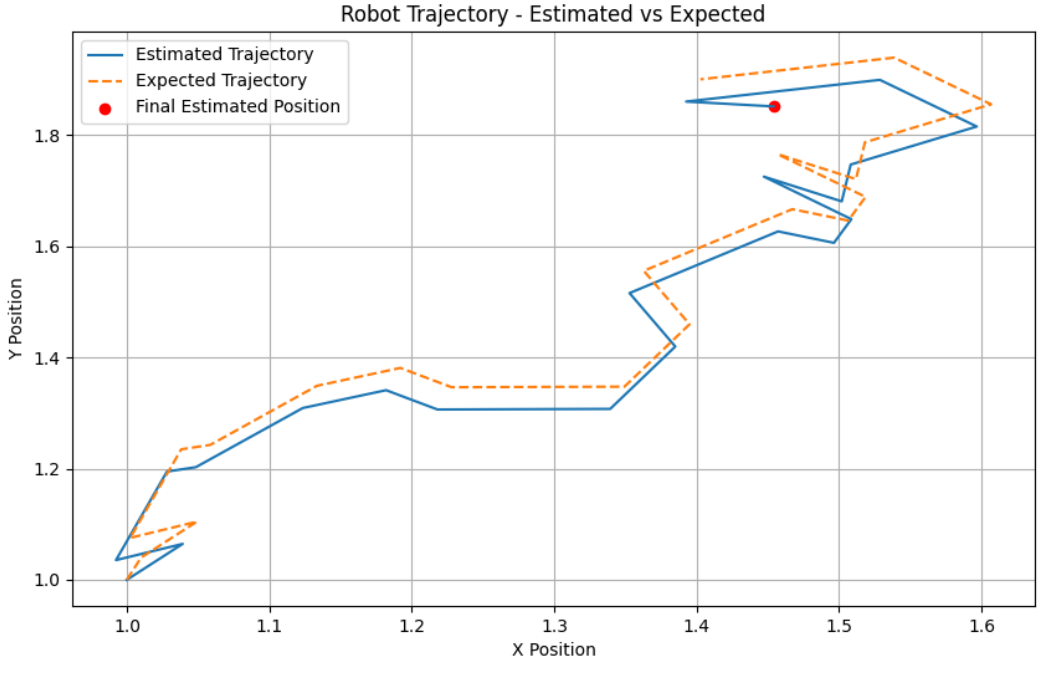

Figure 73: WEKF Simulation Graph when the initial estimation covariance change to [0.01, 0.5, np.pi/180]

several factors affect the simulation of the robot trajectory using the Weighted Extended Kalman Filter (WEKF): Table 1: Different test cases for multiple factors that change together.

	Case I	CaseII	CaseIII
Initial State Vector(xk)	[1,3,270]	[5,7,180]	[2,4,45]
Initial Estimation Covariance (Pk)	[0.01, 0.2, np.pi/180]	[0.05,0.7, np.pi/180]	[0.5, 0.9, np.pi/180]
Process Noise Covariance (Qa)	[0.2, 0.3, 0.7]	[0.5,0.6,0.2]	[0.8, 0.4, 0.5]
Control input(uk)	[0.1, 0.3, np.pi/180 * 5]	[0.6, 0.7, np.pi/180 * 5]	[0.2, 0.5, np.pi/180 * 5]
Measurement Noise Covariance (Ra)	[0.5, 0.7, 0.10]	[0.4,0.8,0.50]	[0.7, 0.9, 0.20]
Weighting Factor(a)	1.2	2.0	5
Simulation Duration	50	60	100

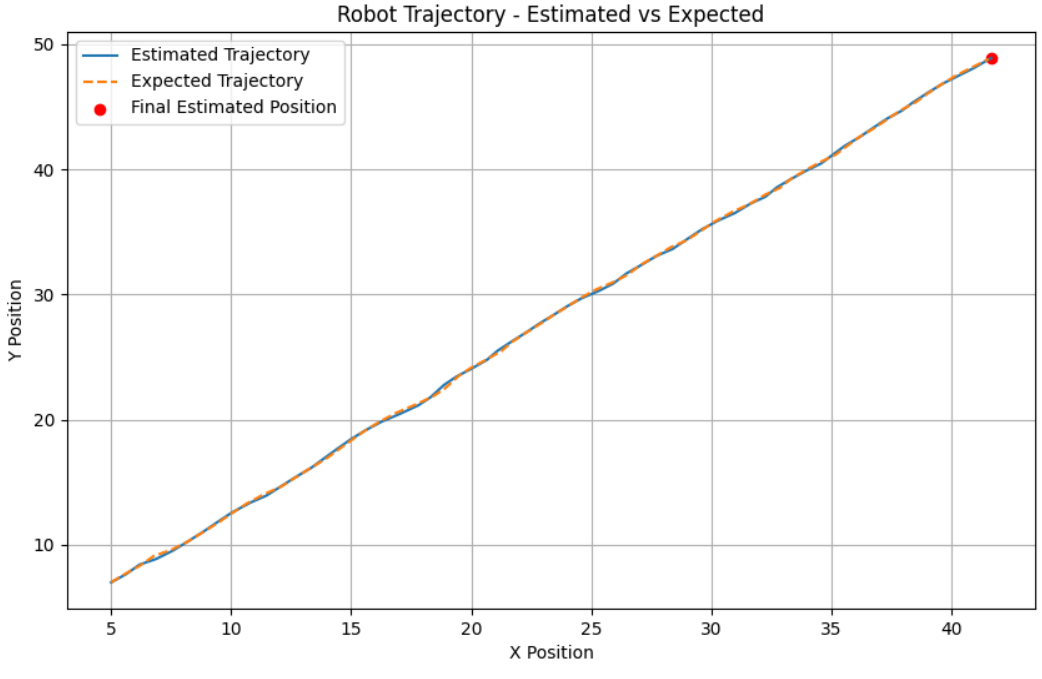

Figure 74: WEKF Simulation Graph for Case I

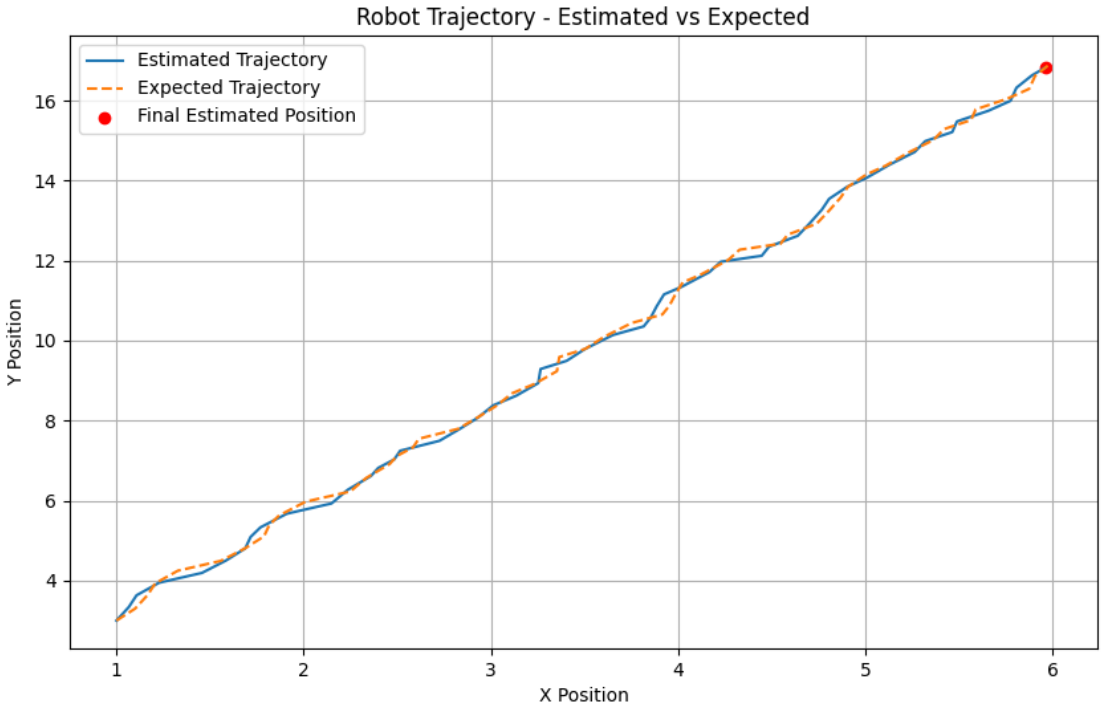

Figure 75: WEKF Simulation Graph for Case II

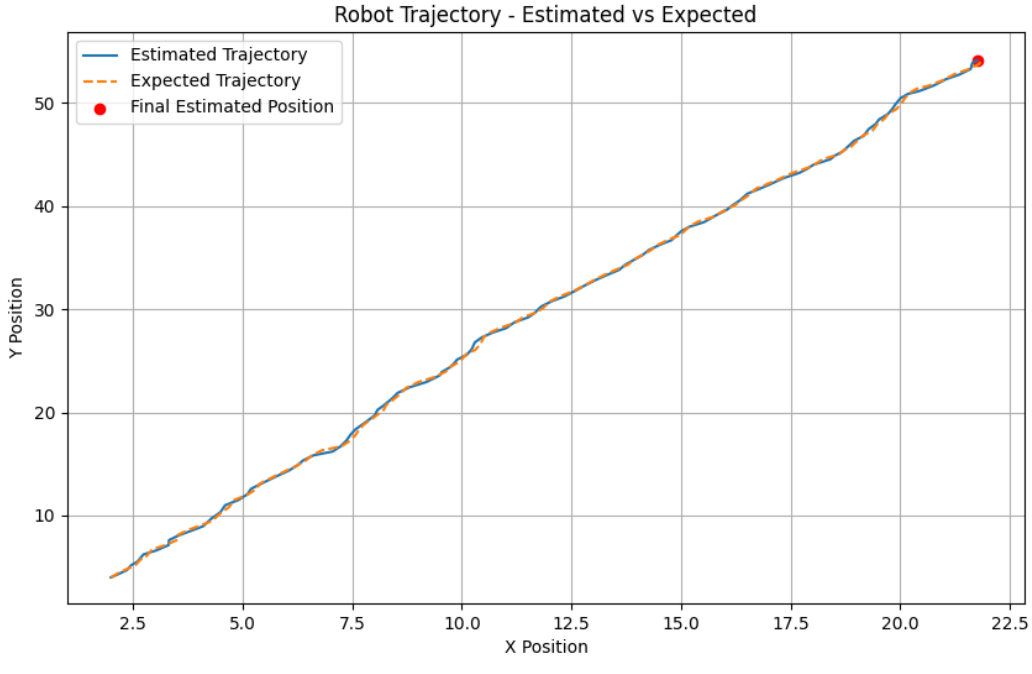

Figure 76: WEKF Simulation II Graph for Case III

Discussion of the WEKF:

Case I is likely to be the most stable due to the lower noise and uncertainty levels. while Case III will be the most responsive to changes in measurements due to the high weighting factor but may also be the most susceptible to measurement noise.

Furthermore, Case II and Case III have longer durations, which may allow the filter more time to converge to the true state, but the higher noise levels could make this convergence less smooth.

The robustness of each filter will largely depend on the accuracy of the model and the characteristics of the measurement noise. Case I seems to be the most balanced and could be expected to be the most robust if the model and measurements are reasonably reliable.

The accuracy of the state estimates will depend on the balance between the model's predictions and the measurement updates. With its moderate settings, Case I might yield the most accurate estimates, assuming the model is fairly accurate, and the measurement noise is not overly problematic.

How the EKF and WEKF differ:

The EKF is suitable for systems with moderate non-linearities and where initial estimates and noise statistics are reliable. However, for systems with high non-linearity or uncertain initial conditions, the WEKF provides a more robust solution at the cost of increased computational effort. It fine-tunes its process by weighing state estimates and adapting to noise variances in real time, offering better performance in complex scenarios.

3.4 UKF – Unscented Kalman Filter Implementation

The unscented Kalman filter approximates the probability distribution. From the priori Gaussian distribution, sample points are calculated so that their mean and covariance are the same as the probability distribution. These points are symmetrically distributed along the original mean, each holding a weight. UKF uses more than 1 point to approximate the next position, as opposed to EKF who uses only the mean to approximate a new linear function from nonlinear function. These points are then transformed through the non-linear function and a new mean and covariance matrix is computed from the sigma points. Below are the steps taken to implement the UKF.

3.4.1 Algorithm Implementation

1. *Initialization*

First in the algorithm, some variables are initiated such as the state vector with the initial position and the angle of rotation, the covariance matrixes Q and R respectively for the process and the sensor noise as well as the variables to determine the scaling parameter lambda as shown in the figure below.

$$\kappa \geq 0 \quad \text{Influence how far the sigma points are away from the mean}$$
$$\alpha \in (0, 1]$$
$$\lambda = \alpha^2(n + \kappa) - n$$
$$\beta = 2 \quad \text{Optimal choice for Gaussians}$$

Figure 77: Scaling parameter variables [3]

Each sigma points for the calculation of the mean have a weight attributed to it. The weights for the mean and the covariance differ only for the first column, the previous mean and covariance in the system as shown in the figure below. In the equation, m stands for the weight mean matrix of 1x(2n+1) where n is the dimension of the system, and the c represents the weight of the covariance matrix. The sum of all points should be 1.

$$w_m^{[0]} = \frac{\lambda}{n + \lambda}$$
$$w_c^{[0]} = w_m^{[0]} + (1 - \alpha^2 + \beta)$$
$$w_m^{[i]} = w_c^{[i]} = \frac{1}{2(n + \lambda)} \quad \text{for } i = 1, \ldots, 2n$$

Figure 78: Weights of the sigma points for the UKF

2. *Prediction*
    i. Compute the sigma points
        a. Number of sigma points

The more sigma points, the more accurate the results of the approximation will be. However, if we would take all points, a lot of resources and computational power is needed. Therefore, a total of 2n+1 is generated for this implementation. The final matrix for sigma points will then be n x (2n+1).

        b. Sigma points calculation

The first sigma point is the previous mean in the system, while the rest are determined by the matrix square root of the dimensionality and the scaling parameter with the previous covariance.

$$\begin{aligned}
\mathcal{X}^{[0]} &= \mu \\
\mathcal{X}^{[i]} &= \mu + \left(\sqrt{(n+\lambda)\,\Sigma}\right)_i \qquad \text{for } i = 1, \ldots, n \\
\mathcal{X}^{[i]} &= \mu - \left(\sqrt{(n+\lambda)\,\Sigma}\right)_{i-n} \qquad \text{for } i = n+1, \ldots, 2n
\end{aligned}$$

Figure 79: Sigma points calculation

    ii. Calculate the mean and variance of the new Gaussian

Once the previous steps are completed, we can predict the mean and the covariance matrix by first computing each sigma point through the non-linear function. Then, to calculate the predicted mean, for each transformed sigma point through the non-linear function, it is multiplied by its respective weight in the summation. To calculate the predicted covariance matrix, the difference between each transformed sigma point with the previous mean with its respective covariant weight is accumulated.

$$\bar{\mu}_t = \sum_{i=0}^{2n} w_m^{[i]} \bar{\mathcal{X}}_t^{*[i]}$$

$$\bar{\Sigma}_t = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\mathcal{X}}_t^{*\,[i]} - \bar{\mu}_t)(\bar{\mathcal{X}}_t^{*\,[i]} - \bar{\mu}_t)^T + R_t$$

Figure 80: Predicted mean and covariance

3. *Update state of the system*

We are not using Jacobian because we are not linearizing the function. Instead, the sigma points are transformed through the measurement equation before calculating the result with their respective mean weight for the state of the system in the measurement space. Calculating the covariance in the measurement space requires the difference of the sigma points with the new calculated state in measurement space.

$$\hat{z}_t = \sum_{i=0}^{2n} w_m^{[i]} \bar{\mathcal{Z}}_t^{[i]}$$
$$S_t = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)(\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)^T + Q_t$$

Figure 81: Formulas for the measurement space

In order to calculate the Kalman matrix, it is needed to find the result of the cross-correlation between the predicted stage and the measurement stage. With this result and the previously calculated covariance matrix in the measurement space, it is possible to find the Kalman matrix.

$$\bar{\Sigma}_t^{x,z} = \sum_{i=0}^{2n} w_c^{[i]} (\bar{\mathcal{X}}_t^{[i]} - \bar{\mu}_t)(\bar{\mathcal{Z}}_t^{[i]} - \hat{z}_t)^T$$
$$K_t = \bar{\Sigma}_t^{x,z} \; S_t^{-1}$$

Figure 82: Formulas for the cross-correlation and the Kalman matrix

Once all have been calculated, the next mean and the covariance can be calculated for the posterior state, and another iteration can be processed.

$$\mu_t = \bar{\mu}_t + K_t(z_t - \hat{z}_t)$$
$$\Sigma_t = \bar{\Sigma}_t - K_t \; S_t \; K_t^T$$

Figure 83: Formulas for the mean posterior and the covariance posterior

3.4.2 Simulations

The code for the implementation of the UKF can be found in the Appendix. The simulation time was set to 100 with each step set to 1. Therefore, 100 iterations would be executed. The covariance matrixes were assumed to be constant for consistency with the previous EKF examples and to simplify the calculations for the UKF points, though in theory, the new covariance matrix would be used to calculate the next iterations sigma points. Four different cases were experimented with for the UKF as seen below representing a trajectory for the AGV to follow in red.

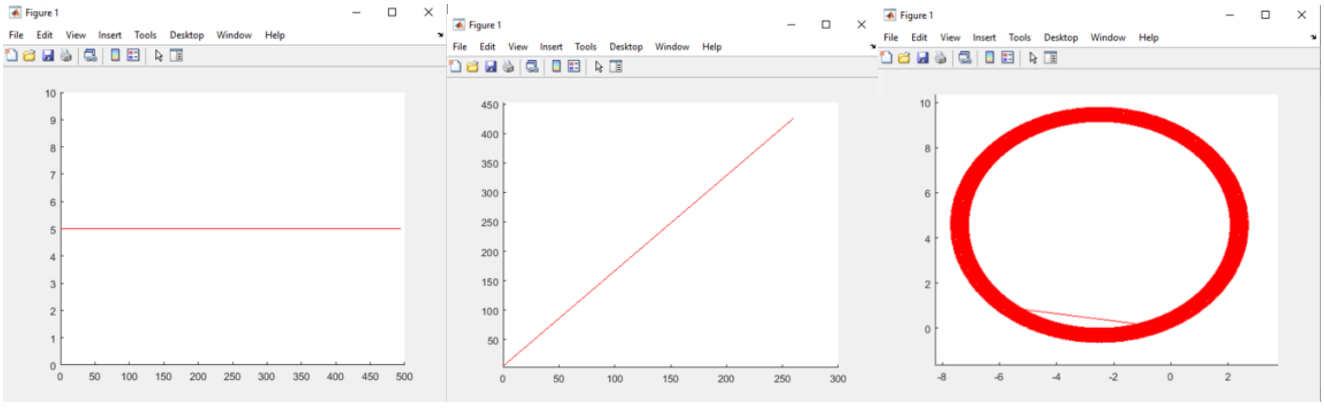

Figure 84: Expected trajectories for the AGV using UKF

I. Constant Trajectory

For the first test case, the constant velocity and the angular velocity are respectively 5 and 0 and the initial state is at [0;5;0]. The initial configuration of the covariance matrixes is Q_0 = [0.1 0 0; 0 0.1 0; 0 0 0.1]; R_0 = [0.1 0 0; 0 0.1 0; 0 0 0.1].

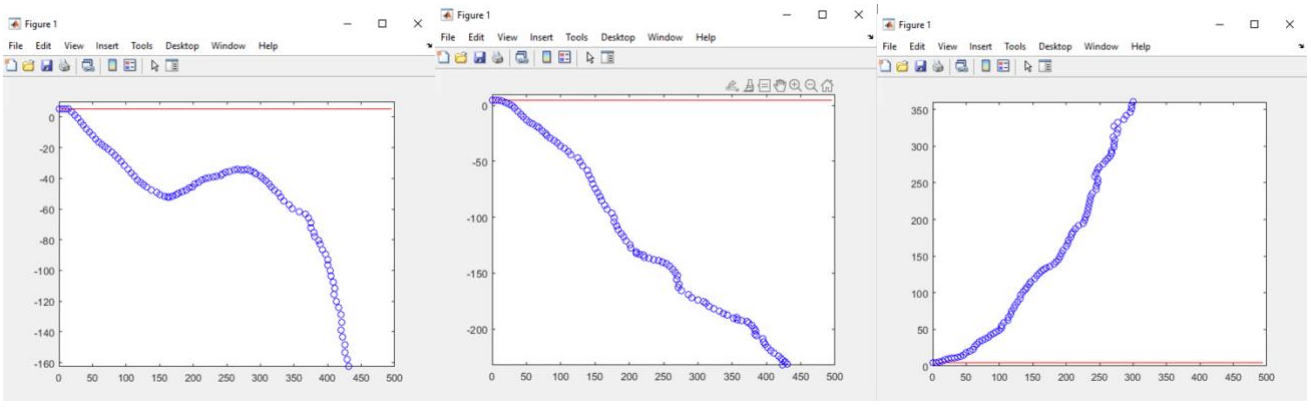

Figure 85: Constant trajectory with a decrease of R_0 by 10 times (0.1 -> 0.01 -> 0.001) and Q_0 being constant with its values at 0.1.

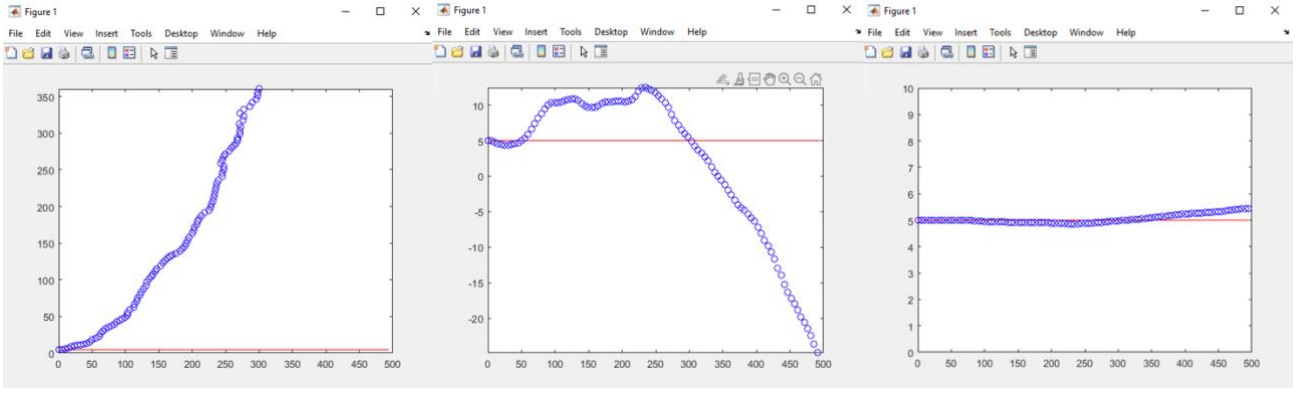

Figure 86: Constant trajectory with a decrease of Q_0 by 10 times (0.1 -> 0.01 -> 0. 000001) with R_0 to be the smallest value from the last simulations.

II. Ramp trajectory

For the second test case, the constant velocity and the angular velocity are respectively 5 and 45 with an initial position at the origin 0. The initial configuration of the covariance matrixes is Q_0 = [0.1 0 0; 0 0.1 0; 0 0 0.1]; R_0 = [0.1 0 0; 0 0.1 0; 0 0 0.1].

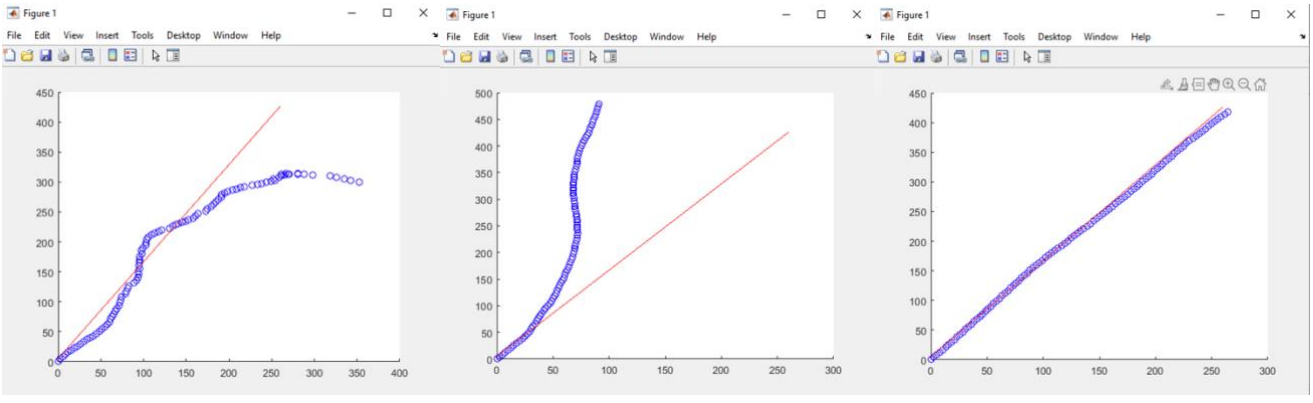

Figure 87: Ramp trajectory - with a decrease of Q_0 by 10 times (0.1 -> 0.01 -> 0. 001) with R_0 to be constant of its initial values.

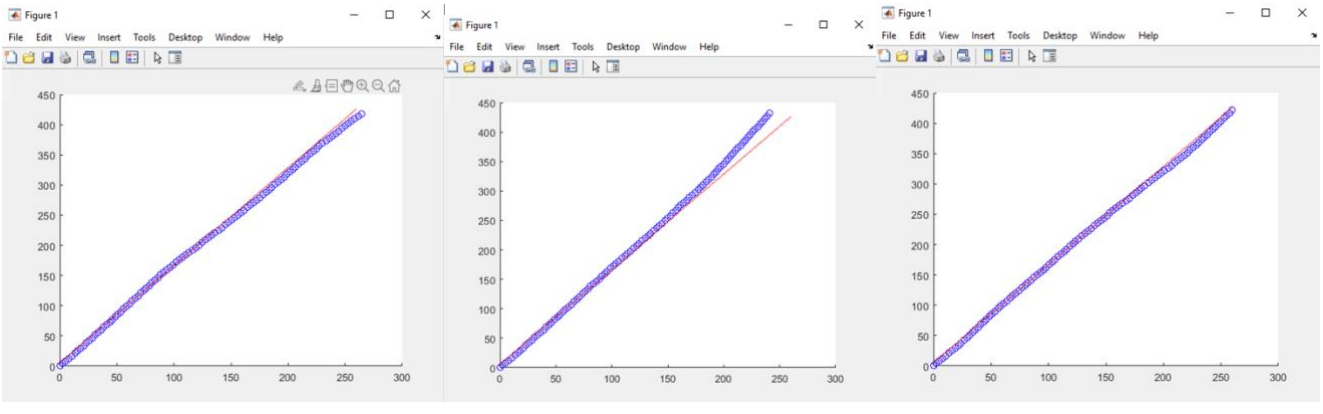

Figure 88: Ramp trajectory - with a decrease of R_0 by 10 times (0.1 -> 0.01 -> 0. 001) with Q_0 to be the smallest value from the previous simulations.

III. Circular trajectory

For the third test case, the constant velocity and the angular velocity are respectively 5 and varies with an initial position at the origin 0. The initial configuration of the covariance matrixes is Q_0 = [0.1 0 0; 0 0.1 0; 0 0 0.1]; R_0 = [0.1 0 0; 0 0.1 0; 0 0 0.1].

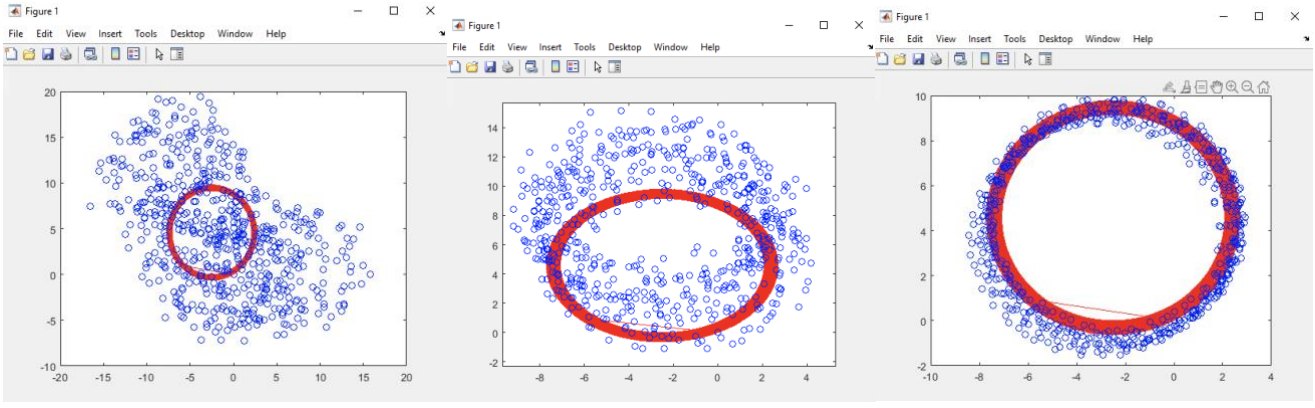

Figure 89: Circular trajectory - with a decrease of Q_0 by 10 times (0.1 -> 0.01 -> 0. 001) with R_0 to be constant of its initial values.

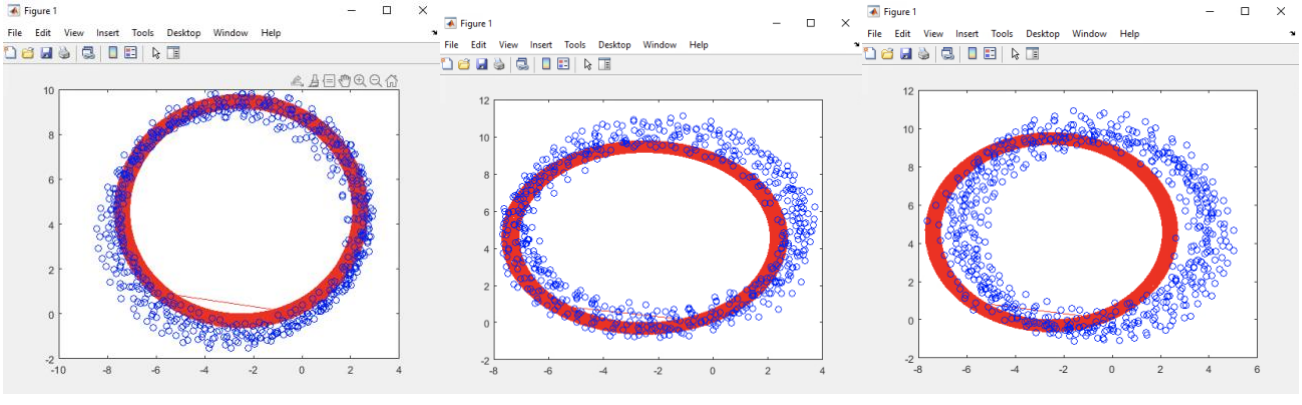

Figure 90: Circular trajectory - with a decrease of R_0 by 10 times (0.1 -> 0.01 -> 0. 001) with Q_0 to be the smallest value from the previous simulations.

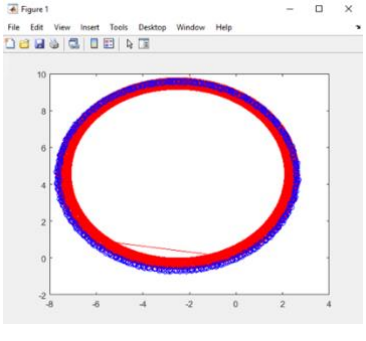

Figure 91: Circular trajectory - Q_0 = [0.00001 0 0; 0 0.00001 0; 0 0 0.00001]; R_0 = [0.001 0 0; 0 0.001 0; 0 0 0.001]

From the simulations of the different trajectories, it is observed how greatly the influence of the covariance matrix Q impacts the prediction results of the AGV trajectory. For both covariance matrixes, the more the value is decreased, the more accurate the predictions are. Logically, since the covariance matrixes are associated with the noise in the system, the smaller they are, the more accurate the predictions are estimated. It is also noticed that optimal predictions are a result of the R matrix being greater than the Q matrix since R is a measurement noise which is greater than the process noise.

3.6 Factors that influence the estimation and accuracy of EKF and UKF in the simulation

The algorithm used for the EKF, WEKF and UKF are prediction algorithms. With these algorithms, accuracy in the implementation influences the results of the system. In this scenario, the AGV position is the principal factor to monitor and estimate its next position in its motion. Various factors influence the results of the estimations observed in the simulations, notably the values for the covariance matrixes, the non-linearity of the system, the sample time with the propagation of uncertainty.

The values of Q and R directly impact filter performance since they are associated with the noises influencing the accuracy of the system. The covariance matrixes represent the difference between the true position and the prediction with each iteration and therefore, carrying the noise and error of the system.

EKF make linear approximations of the system. The accuracy depends on how well this linearization estimates the true system. UKF uses the average of a set of sigma points to predict the posterior using the non-linear function. The accuracy for this factor for UKF depends on the number of sigma points used.

The frequency of measurements determines the steps between the sample in time. As the sample are closer to each other in time, the more accurate the prediction are since there is little difference in time. The distance between the two points in time is less the closer they are in time.

The uncertainty propagates over time can affect the performance negatively since, in the UKF, the newfound covariance matrix is generally used to calculate the next set of sigma points.

3.7 Discussion and Conclusion

Both EKF, WEKF, and UKF improve position estimation in non-linear models by combining information collected from different sensors. EKF and WEKF use Jacobians matrices to linearize the system, however WEKF uses weighted values based on time for its Q and R covariance matrix calculations. UKF simplifies this process by selecting sigma points that capture the mean and variance of the state distribution.

Despite their effectiveness, the three sensor fusion algorithms may face challenges in scenarios with significant sensor noise in the Q and R covariance matrices, outliers, or abrupt changes. High levels of sensor noise can lead to an overall lower filter performance, causing the state estimations to deviate from the true path of the vehicle.

Since there was an assumed zero-mean white Gaussian noise, the models did not need to include fuzzy logic to rationalize the mean of the white Gaussian noise. In real application however, the exact values for the covariance matrixes are unknown, resulting in a non-white noise which causes the diverge. Adaptive fuzzy logic system (AFLS) is useful for the Kalman filter to adapt to the new measurements of the sensors. AFLS uses the process mean and covariance values to weigh the data.

4. References

Allen, P. (2017). Differential Drive Robots. *Differential Drive Kinematics*. https://www.cs.columbia.edu/~allen/F17/NOTES/icckinematics.pdf

C. -L. Wang and D. -S. Wu (2008). Decentralized Target Tracking Based on a Weighted Extended Kalman Filter for Wireless Sensor Networks (pp. 1-5). IEEE. https://ieeexplore.ieee.org/document/4697838

D'Alfonso, L. et al. (2015) Mobile robot localization via EKF and UKF: A comparison based on real data. Robotics and Autonomous Systems (Vol. 24, Part A). https://www.sciencedirect.com/science/article/abs/pii/S0921889015001517

Fariña, B. et al (2023). Sensor Fusion Algorithm Selection for an Autonomous Wheelchair Based on EKF/UKF Comparison. *International Journal of Mechanical Engineering and Robotics Research* (Vol. 12, No. 1). https://www.ijmerr.com/2023/IJMERR-V12N1-1.pdf

[MATLAB]. (2017, May 17). *Nonlinear State Estimators | Understanding Kalman Filters, Part 5* [Video]. YouTube. https://www.youtube.com/watch?v=Vefia3JMeHE

Sasiadek, J. Z. (2002). Sensor fusion. *Annual Reviews in Control*, 26(2), 203-228. https://doi.org/10.1016/S1367-5788(02)00045-7

Singh Chadha, H. (2017, April 27). *The Unscented Kalman Filter: Anything EKF can do I can do it better!* Medium. https://towardsdatascience.com/the-unscented-kalman-filter-anything-ekf-can-do-i-can-do-it-better-ce7c773cf88d

Stachniss, C. (2022, December 1). *Unscented Kalman Filter (UKF)* [Power Point]. UNI FREIBURG. http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam05-ukf.pdf

Wan, E. A. and Merwe, R. (2000). The Unscented Kalman Filter for Nonlinear Estimation. *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium* (Cat. No.00EX373). https://groups.seas.harvard.edu/courses/cs281/papers/unscented.pdf

Welch, G., & Bishop, G. (2006). An Introduction to the Kalman Filter (TR 95-041). Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175.

5. Appendix

*EKF.m (*Natasha Tremblay)

```
% Initialize Variables
   syms t wx wy wtheta n
   dt = 1;
   T= 2000;
   d1=8;d2=2;L=2;
   I =  diag([ 1, 1, 1]); %identity matrix
   Q = diag([ 0.1, 0.1, 0.1]); %process noise matrix
   R = diag([ 0.1, 0.1, 0.1]); %sensor noise matrix
   x=[1;1;1];P=diag([1,1,1]);
   [v, w] = velocities(d1,d2,L,dt);
%Graphing
   figure;
   title(['EKF Estimated Trajectory, v = ',num2str(v),' w = ', num2str(w)]);
   xlabel('X (m)');
   ylabel('Y (m)');
   grid on; hold on;
   estimated_positions = zeros(T / dt + 1, 2);
   theta_curr=0;


   for k= 1:T/dt + 1
         [theta_next] = angle(theta_curr, w, dt);
         theta_delta=theta_next-theta_curr;
         theta_curr=theta_next;
         display(v+":"+w+":"+theta_curr+":"+theta_next);
         [B_pred, A_pred, C_up] = jacobians(v,theta_delta, theta_next, dt);
         [x_sq, z_sq] = xz(v, w, theta_next, I);
         u=[v; w];
         [x,P] = EKF(u, x, P, A_pred,B_pred,C_up, Q, R,z_sq, I);
           estimated_positions(k, :) = x(1:2)';
   end
plot(estimated_positions(:, 1), estimated_positions(:, 2), 'bo-');
hold off;


function [v, w] = velocities(d1,d2,L, dt)
         vr=d1/dt; vl=d2/dt;
         v= (vr+vl) / 2;
         w= (vr-vl) / L;
end
function [theta_next] = angle(theta_curr, w, dt)
         theta_next = mod(w*dt + theta_curr,180);
end
function [B_pred, A_pred, C_up] = jacobians(v,theta_delta,theta,dt)
   %Jacobian Matrices
         A_pred= [1 0 -v*dt*sin(theta); 0 1 v*dt*cos(theta); 0 0 1];
         C_up= [1 0 -v*dt*sin(theta); 0 1 v*dt*cos(theta); 0 0 1]; % Ck=Ak+1
         B_pred= [dt*cos(theta+((theta_delta)/2)) 0; dt*sin(theta + ((theta_delta)/2)) 0; 0 1];
end
function [x_sq, z_sq] = xz(v,w,theta,I)
         theta_dot = w;
         x_dot=v*cos(theta);
         y_dot=v*sin(theta);
         x_sq= [x_dot; y_dot; theta_dot];%state model with zero mean(approximate)
         z_sq= I*x_sq;     % measurement model with zero mean(approximate)
end
function [x,P] = EKF(u,x,P, A_pred,B_pred,C_up, Q, R, z_sq, I)
   %Predict
         x_pred = x + B_pred*u;
         P_pred = A_pred*P*transpose(A_pred)+Q;
   %Update
         K_up= P_pred*transpose(C_up)*inv(C_up*P_pred*transpose(C_up)+R);
         x=x_pred + K_up*(z_sq - C_up*x_pred);
         P=(I-K_up*C_up)*P_pred;
end
```

*UKF.m (*Ines Mansouri)

```
clear all;

%Initialization
dt = 1;
T = 100; %200*pi for circular trajectory
state_0 = [0;5;0]; %[0;0;45] for ramp and circular trajectories
Q_0 = [0.1 0 0; 0 0.1 0; 0 0 0.1];
R_0 = [0.1 0 0; 0 0.1 0; 0 0 0.1];
alpha = 1;
beta = 2;
k = 1;
n = length(state_0);
mean = [0 0 0];



trajectory_ukf = zeros(2,ceil(T/dt));

%calculate weights
weight_mean = sigmaWeight_mean(n,alpha,k);
weight_cov = sigmaWeight_cov(n,alpha,beta,k);

expected_trajectory1 = zeros(2,ceil(T/dt));
et_x = 0;
et_y = 5; %0 for ramp and circular trajectories

for k_t = 1:dt:T

    %example on how the velocities are found at the base level
    %d1 = 5;
    %d2 = 5;
    %vr = d1/dt;
    %vl = d2/dt;
    %v = (vr+vl)/2;
    %w = (vr - vl)/2;


    %constant values for the velocities for simple implementation
    v = 5;
    w = 0; %45 for circular trajectory
    w_et = 45; %45 for ramp trajectory and k_t for circular trajectory
    expected_trajectory1(1,k_t) = et_x;
    expected_trajectory1(2,k_t) = et_y;
    et_x = v*cos(w_et) + et_x;
    et_y = v*sin(w_et) + et_y;


    ci = [v,w];


    trajectory_ukf(1,k_t) = state_0(1);
    trajectory_ukf(2,k_t) = state_0(2);

%Prediction

    %calculate sigma points
    S = sigmaPoints(state_0,Q_0,alpha,k);


    display(S);


    %calculate predicted_mean
    predicted_m = predicted_mean(weight_mean,S,ci,n,Q_0,mean);


    %calculate preddicted_covariance matrix
```

```
    predicted_c = predicted_cov(n,weight_cov,S,predicted_m);




%Update

    %calculate measurment space
    Z = hmeasurment(n,S,R_0,mean,Q_0,ci);

    %calculate mean_measurmentSpace
    z = mean_measurmentSpace(n,weight_mean,Z);

    %calculate covariance measurment space
    cov_mesSpcae = cov_measurmentSpace(n,weight_cov,Z,z,R_0);

    %calculate cross corelation
    T = cross_corelation(n,weight_mean,predicted_m,z,Z,S);

    %calculate K
    K = T.*transpose(cov_mesSpcae);

    display(K);

    %return new mean and covariance
    mean_postori = predicted_m + K*(S(:,1) - z);
    cov_postori = predicted_c - K*cov_mesSpcae*transpose(K);

    disp(k_t);
    trajectory_ukf(1,k_t) = state_0(1);
    trajectory_ukf(2,k_t) = state_0(2);

    state_0 = mean_postori;
    %Q_0 = cov_postori; %if the covariance were assumed to not be constants

end

plot(expected_trajectory1(1,:),expected_trajectory1(2,:),'r');
hold on;
scatter(trajectory_ukf(1,:),trajectory_ukf(2,:),'b');
%axis([0 500 0 10]);

graph;
```

sigmaWeight_mean.m

```
function [W] = sigmaWeight_mean(n,alpha,k)

    l = 2*n + 1;
    W = zeros(1,l);
    lambda = (alpha^2) * (n + k) - n;

    W(1,1) = lambda/(n+lambda);

    for c = 1:2*n

        W(:,c+1) = 1/(2*(n+lambda));

    end
```

```
end
```

sigmaWeight_cov.m

```
function [W] = sigmaWeight_cov(n,alpha,beta,k)

    l = 2*n + 1;
    W = zeros(1,l);
    lambda = (alpha^2) * (n + k) - n;


    W(1,1) = (lambda/(n+lambda)) + (1-(alpha^2)+beta);

    for c = 1:2*n


        W(:,c+1) = 1/(2*(n+lambda));
    end
end
```

sigmaPoints.m

```
%state vector x is of size n=3 (3x1)
%covariance matrix Q is of size 3x3
%assumed 0 mean Gaussian
%S has a size of n x 2n+1 = 3 x 7
%Q is a lower triangle matrix



function [S] = sigmaPoints(state,Q,alpha,k)

    n = length(state);
    l = (2*n)+1;
    lambda = (alpha^2)*(n+k) - n;


    display(lambda);


    S = zeros(n,l);


    C = (n+lambda) * Q;



    Q_chol = chol(C); %used for the squared root



    S(:,1) = state;



    for c = 1:n
        S(:,c+1) = state + Q_chol(:,c); % i = 1, ... , n
        S(:,c+1+n) = state - Q_chol(:,c); % i = n+1 , ... , 2n
    end

end
```

process_model.m

```
function [PM_matrix] = process_model(v_p,w_p,theta_prev,Q,mean,n)

   PM_matrix = zeros(n,1);
```

```
    process_noise = mvnrnd(mean, Q);

    x_next = v_p*cos(theta_prev) + process_noise(1,1);
    y_next = v_p*sin(theta_prev) + process_noise(1,2);
    theta_next = w_p + process_noise(1,3);

    PM_matrix(1,1) = x_next;
    PM_matrix(2,1) = y_next;
    PM_matrix(3,1) = theta_next;

end
```

predicted_mean.m

```
function [M] = predicted_mean(W,S,contr_input,n,Q,mean)

    M = zeros(n,1);
    M = S(:,1);

    for c = 1:2*n+1
        p_m = process_model(contr_input(1,1),contr_input(1,2),S(n,1),Q,mean,n);
        M(1,1) = M(1,1) + W(1,c) * p_m(1,1);
        M(2,1) = M(2,1) + W(1,c) * p_m(2,1);
        M(3,1) = M(3,1) + W(1,c) * p_m(3,1);
    end
End
```

predicted_cov.m

```
function [C] = predicted_cov(n,W,sigma_predicted,mean_predicted)

    C = zeros(n,n);

    for c = 1:2*n +1

        diff = (sigma_predicted(:,c) - mean_predicted(:,1)).*transpose(sigma_predicted(:,c) -
mean_predicted(:,1));
        C(1,1) = C(1,1) + W(1,c) * diff(1,1);
        C(2,2) = C(2,2) + W(1,c) * diff(2,1);
        C(3,3) = C(3,3) + W(1,c) * diff(3,1);
    end
end
```

mean_measurmentSpace.m

```
function [z] = mean_measurmentSpace(n,W,Z_i)



    z = zeros(n,1);



    for c = 1:2*n +1
        z(1,1) = z(1,1) + W(1,c) .* Z_i(1,c);
        z(2,1) = z(2,1) + W(1,c) .* Z_i(2,c);
        z(3,1) = z(3,1) + W(1,c) .* Z_i(3,c);
    end
end
```

hmeasurment.m

```
function [Z] = hmeasurment(n,S,R,mean,Q,contr_input)
```

```
    Z = zeros(n,2*n+1);
    R_t = mvnrnd(mean, R);

    for c = 1:2*n +1
        p_m = process_model(contr_input(1,1),contr_input(1,2),S(n,1),Q,mean,n);
        CX = eye(3).*p_m;
        Z(1,c) = CX(1,1) + R_t(1,1);
        Z(2,c) = CX(2,1) + R_t(1,2);
        Z(3,c) = CX(3,1) + R_t(1,3);
    end
end
```

cross_corelation.m

```
function [T] = cross_corelation(n,W,pm_cr,z,Z,S)

    T = zeros(n,1);
    i = 1;

    for c = 1:2*n+1
        weightedCR = (S(:,c) - pm_cr(:,1)) .* transpose(Z(:,c) - z(:,1));
        T(1,1) = T(1,1) + (W(1,c) * weightedCR(1,1));
        T(2,1) = T(2,1) + (W(1,c) * weightedCR(2,1));
        T(3,1) = T(3,1) + (W(1,c) * weightedCR(3,1));

    end
end
```

cov_measurmentSpace.m

```
function [C] = cov_measurmentSpace(n,W,Z_i,z,R)

    C = zeros(length(R),length(R));

    for c = 1:2*n +1
        diff = (Z_i(:,c) - z(:,1)).*transpose(Z_i(:,c) - z(:,1));
        C(1,1) = C(1,1) + W(1,c).*diff(1,1);
        C(2,2) = C(2,2) + W(1,c).*diff(2,2);
        C(3,3) = C(3,3) + W(1,c).*diff(3,3);
    end

end
```