**THE UNIVERSITY OF DANANG**

**UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**FACULTY OF ADVANCED SCIENCE AND TECHNOLOGY**

**TERM PROJECT**

**INTELLIGENT ROBOTICS**

Lecturer: **Associate Professor LÊ TIẾN DŨNG, PhD**

Students: **Nguyễn Đình Tiến Đạt**

**Lê Quang Kiệt**

**Võ Đại Thuận**

**Trần Minh Ánh**

Class: 21ES

Major**: EMBEDDED SYSTEMS AND IoT**

**Da Nang, 2025**

**Use Of AI in Report**

We have utilised AI tools in our report:

* No
* **Yes**

We hereby declare that the AI-based applications used in generating this work are as follows:

| **Application** | **Version** |
| --- | --- |
| Google Gemini | Pro |
| ChatGPT | 5.0 |

**Purpose of the use of AI**

Language refinement and the writing of grammatically correct and smooth text.

**Parts of this work where AI was used:**

Overview (Part of Chapter 1)

Organizing the equations for LaTeX writing in a report

Language refinement in the introduction of Chapters 3 & 4

**Acknowledgement of Risks**

We hereby state that, as the writer of this work, we are fully responsible for the contents presented in this report. This includes the parts produced by an AI, either partly or entirely. We hereby also state our responsibility if the use of AI has resulted in the breach of ethical standards.

# 

# CONTENTS

[**CONTENTS 2**](#_heading=h.2nl5c7ngnjkz)

[**LIST OF FIGURES 3**](#_heading=h.hh7tvqczh28s)

[**LIST OF TABLES 5**](#_heading=h.wr541evm6b5i)

[**CHAPTER 1: INTRODUCTION 6**](#_heading=h.l1pdofjnzcmq)

[1.1 Project Overview 6](#_heading=h.h1wteowhyl6s)

[1.1.1 Project Objectives 6](#_heading=h.3mhn9t9q3nn2)

[1.1.2 Manipulator Description 6](#_heading=h.uq9uhwaynteh)

[1.1.3 Report Structure 7](#_heading=h.utz319bxwmki)

[**CHAPTER 2: KINEMATIC ANALYSIS 8**](#_heading=h.1zyfg189en1k)

[2.1 Denavit-Hartenberg (DH) Parameters 8](#_heading=h.3h6vxv993f07)

[2.2 Forward Kinematics 8](#_heading=h.g8a174sjcgol)

[2.3 Inverse Kinematics 11](#_heading=h.h9m2vdut5laa)

[2.4 Numerical Verification: 13](#_heading=h.hqu1zcj11ocq)

[2.5 Simulink model: 14](#_heading=h.7wyoyisu17nz)

[2.6 Trajectory planning: 15](#_heading=h.4z335o37swj0)

[2.7 Results: 17](#_heading=h.9dg6h5c017hs)

[**CHAPTER 3: DYNAMIC MODELING AND CONTROL 24**](#_heading=h.f2fndinmhrkg)

[3.1 Dynamic Model Derivation 24](#_heading=h.v4ar7qdokoz1)

[3.2 Simulink model 31](#_heading=h.fxwss67wj8sj)

[3.3 Result 32](#_heading=h.m5g41ihfb1c9)

[**CHAPTER 4: NEURAL NETWORK DESIGN 35**](#_heading=h.krguvkjzza7i)

[4.1 Algorithm Selection. 35](#_heading=h.t2epbozhwsne)

[4.2 Implementation of the Neural Network Controller. 35](#_heading=h.ta3ah7egfjst)

[4.3 Simulink model: 39](#_heading=h.uyhy2t8jb9nt)

[4.4 Result: 40](#_heading=h.yyxx28mwcbvz)

[**CHAPTER 5: MORE SIMULATION & DETAIL ANALYSIS 42**](#_heading=h.3dgzw56lk6i4)

[5.1 PD controller testing 42](#_heading=h.yfow5rj1i98b)

[5.2 Robot control testing (without Neural Network) 45](#_heading=h.vbqw2w52hvrz)

[5.3 Robot testing with Neural Network 48](#_heading=h.jrdit8qajr2f)

[**CHAPTER 6: CONCLUSION 52**](#_heading=h.qvngq6r0lo8u)

[**REFERENCES 54**](#_heading=h.qyqgl5mn5bft)

[**Appendix A - Creating Multibody for Kinematic Analysis 55**](#_heading=h.nh2j4ig6dvwa)

[**Appendix B - Creating Multibody for Dynamic Analysis 70**](#_heading=h.nfbagx4dpkhl)

[**Appendix C - Creating Multibody for Neural Network 84**](#_heading=h.etq3vxfv4zfu)

# LIST OF FIGURES

Figure 1.1 - A 3-DOF robotic manipulator

Figure 2.1 - Overall dynamic model

Figure 2.2 - Mechanical model

Figure 2.3 - Trajectory planning to draw a circle and a straight line

Figure 2.4 - Desired trajectory for circle

Figure 2.5 - Desired trajectory for a straight line

Figure 2.6 - The display block shows that the trajectory planning value is accurate

Figure 2.7 - Smooth and continuous circular path

Figure 2.8 - Straight line from (-2,0) to (2, 0)

Figure 2.9 - The Display block shows that the trajectory planning value is accurate

Figure 2.10 - An error in the actual circular trajectory (Kinematic)

Figure 2.11 - The actual circle trajectory (Kinematic)

Figure 2.12 - An error in the actual straight trajectory (Kinematic)

Figure 2.13 - The actual line trajectory (Kinematic)

Figure 3.1 - Overall system of Dynamic Model[[1]](#footnote-0)

Figure 3.2 - Computed Torque block

Figure 3.3 - The values of position and velocity of the joints are sent back to the Computed Torque block

Figure 3.4 - The Computed Torque block receives position, velocity, and acceleration desired from the Inverse Kinematic block

Figure 3.5 - Actual trajectory after adding the torque computed block

Figure 3.6 - Actual trajectory after adding the torque computed block

Figure 4.1. Neural Network Controller Hidden Layer Diagram.

Figure 4.2 - Sliding Surface Working Principle.

Figure 4.3 - Overall system for AI[[2]](#footnote-1)

Figure 4.4 - Overall Neural Network model

Figure 4.5 - Add the output of the Neural Network’s value with the torque controller to activate it

Figure 4.6 - Actual trajectory before adding the neuron network

Figure 4.7 - Actual trajectory after adding the neuron network

Figure 5.1 - Correlation in PD controller design

Figure 5.2 - The trajectory has a massive overfitting (over 100%)

Figure 5.3 - The overfitting has been reduced

Figure 5.4 - The overfitting has disappeared

Figure 5.5 - No overfitting and really good steady state error in both axis

Figure 5.6 - Actual circle trajectory drawn before adding disturbance

Figure 5.7 - Set the value of friction for three joint

Figure 5.8 - Adding too much friction may cause the robot to stop at the 90-degree turn

Figure 5.9 - Friction is not affecting too much on the final output

Figure 5.10 - Friction value overtime when using scope to measure (105)

Figure 5.11 and Fig 5.12 - Actual trajectory after adding weight in 15th and 60th seconds

Figure 5.13 - The trajectory in the first 15 seconds

Figure 5.14 - The trajectory after run for 60 seconds

Figure 5.15 - The bigger the learning rate, the longer it takes to converge

Figure 5.16 - Actual trajectory after applying the neural network with a learning rate of 0.02

# 

# LIST OF TABLES

Table 2.1 - Denavit-Hartenberg (modify) parameters

# INTRODUCTION

## Project Overview

### Project Objectives

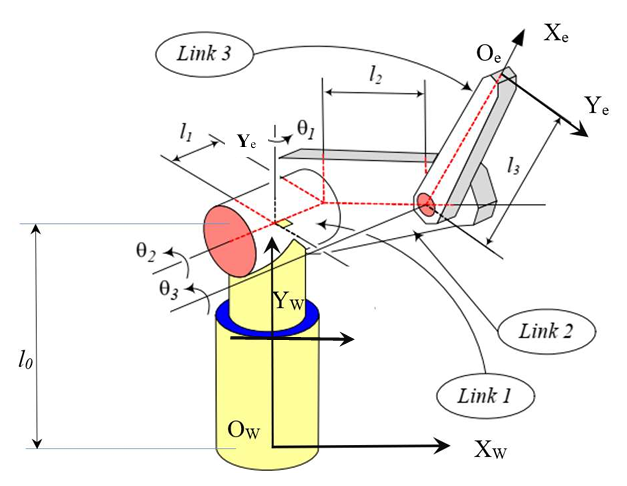
The primary goal of the project, as part of the Intelligent Robotics course, is to conduct a comprehensive analysis, simulation, and control design for the 3-DOF (Degrees of Freedom) robotic manipulator (RRR)*[1]*.

The project encompasses four main tasks:

* *Kinematic Analysis***:** To establish coordinate frames and derive the Denavit-Hartenberg (DH) parameter table. Based on this, the forward kinematic equations for the end-effector position (, , ) and the inverse kinematic equations for the joint variables (, , ) will be derived
* *Dynamics and Control*: To derive the dynamic model of the robot and develop a sliding mode control (SMC) algorithm to ensure precise trajectory tracking (Largian)
* *Intelligent Control*: To apply an intelligent control algorithm, an Artificial Neural Network (ANN) or Fuzzy Logic System, to improve the manipulator's performance. This includes the detailed mathematical formulation of the chosen algorithm.
* *System Simulation and Analysis*: To build a complete simulation model combining Simscape (Multibody) and Simulink to evaluate and compare the performance of the conventional (SMC) and intelligent control algorithms based on tracking errors, torque plots, and other metrics.

### Manipulator Description

The system under study is a 3-DOF (Degrees of Freedom) robotic manipulator. Its mechanical structure, illustrated in Figure 1:



*Figure 1.1 - A 3-DOF robotic manipulator*

The robot's configuration is determined by three revolute joints, represented by the joint variables , , and . Key geometric parameters include the base height and the link lengths , , and . The ultimate goal is to control the position (, , ) and orientation of the end-effector frame {e} (at origin ) relative to the world frame {W} (at ).

### Report Structure

This report systematically addresses the project tasks outlined in ***Section 1.1.1***. ***Chapter 2*** details the complete kinematic analysis, including the DH parameter derivation, forward and inverse kinematic equations, and their numerical verification. ***Chapter 3*** presents the step-by-step procedure for building the Simscape Multibody simulation model and tracking kinematic trajectories. ***Chapter 4*** covers the derivation of the dynamic model and the design of a sliding mode controller. ***Chapter 5*** introduces the selected intelligent control algorithm and its mathematical formulation. Finally, ***Chapter 6*** provides a comparative analysis of the simulation results for both the conventional and intelligent controllers, followed by a general Conclusion summarizing the project's key findings.

# KINEMATIC ANALYSIS

## Denavit-Hartenberg (DH) Parameters

Construct Denavit-Hartenberg (DH) parameters based on the Denavit-Hartenberg modified approach:

| Link |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 |  |
| 2 | 0 |  |  |  |
| 3 |  | 0 | 0 |  |

***Table 2.1 -*** *Denavit-Hartenberg (modify) parameters*

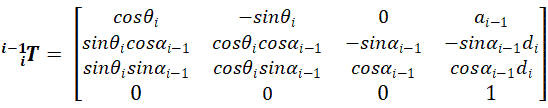
Parameters in D-H (modify) table:

* is the distance between and measured along the
* is the twist angle from and measured about (right-hand rule)
* is the distance between and measured along
* is the rotation angle from and measured about

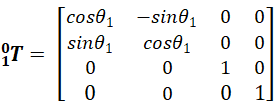
## Forward Kinematics

**Step 1:** We define transformation matrices , , ,

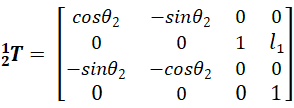
* Based on the transformation matrix:



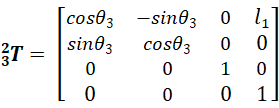
* Transformation matrices



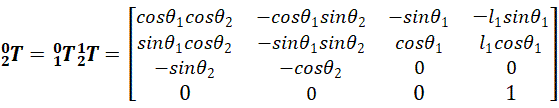
* Transformation matrices



* Transformation matrices



* Transformation matrices

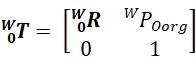


* Transformation matrices

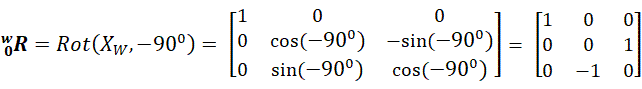


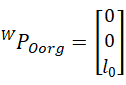
**Step 2:** Define transformation matrices ,

* Using a homogeneous transformation matrix for

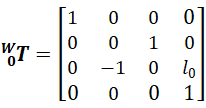


* We have

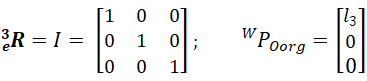


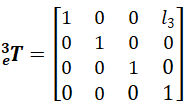


* Transformation matrices

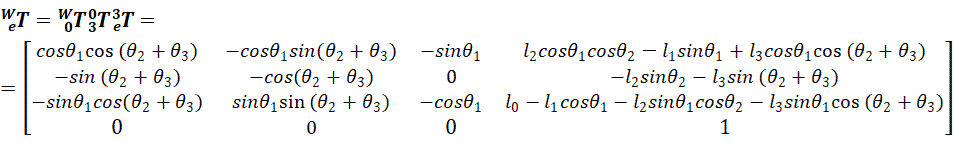


* Similarly, we have:



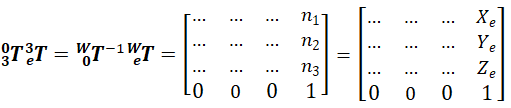


**Step 3:** Define transformation matrices (*Overall Transformation****)***

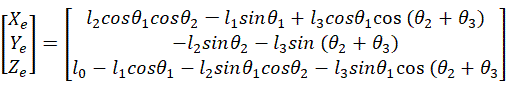


**Step 4:** Forward Kinematic Equations:

* Based on the formula:



* Then , , are:

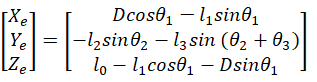


## Inverse Kinematics

Based on the calculation in ***Section 2.2***, we denote:



Then , , become:



**Step 1:** Assuming , ,

* With three rotation angle parameters, the problem would be more difficult.
* To solve this problem, we put some constraints:



* Now, D and becomes:

;  

(*Note: D now depends only on and depends only on* )

**Step 2:** Finding from

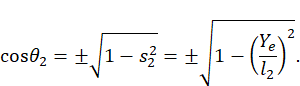
 

  *(Condition: )*

* Assume:



→ There are two possibilities for :



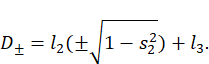


**Step 3:** Finding D from

* Because , we have:



* With the in step 2, we have D:



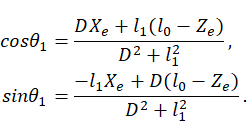
**Step 4:** Finding from ,

* After step 3, combining with ***Section 2.2***:



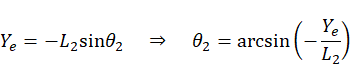
or 

* Solving via Determinant (Cramer's Rule or Matrix Inversion): , we have:



Then: 

**Conclusion:**

****

****

****

****

## Numerical Verification:

**Step 1:** Substitute numbers into the formula , , in:

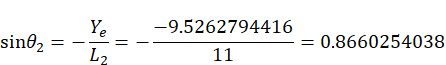
* + , , , 
  + , , 
  + *(Note: convert to radian: →θ[rad]= θ\*)*
* End-effector position calculation result (rounded to 3 decimal places):







**Step 2:** Calculation from



* The valid solution range of arcsine is always:



→ 

**Step 3:** Calculation D from



**Step 4:** Finding from ,

* We have the linear system for , :



* Substituting the values:



* Solving the system, we obtain:



* Thereby: 

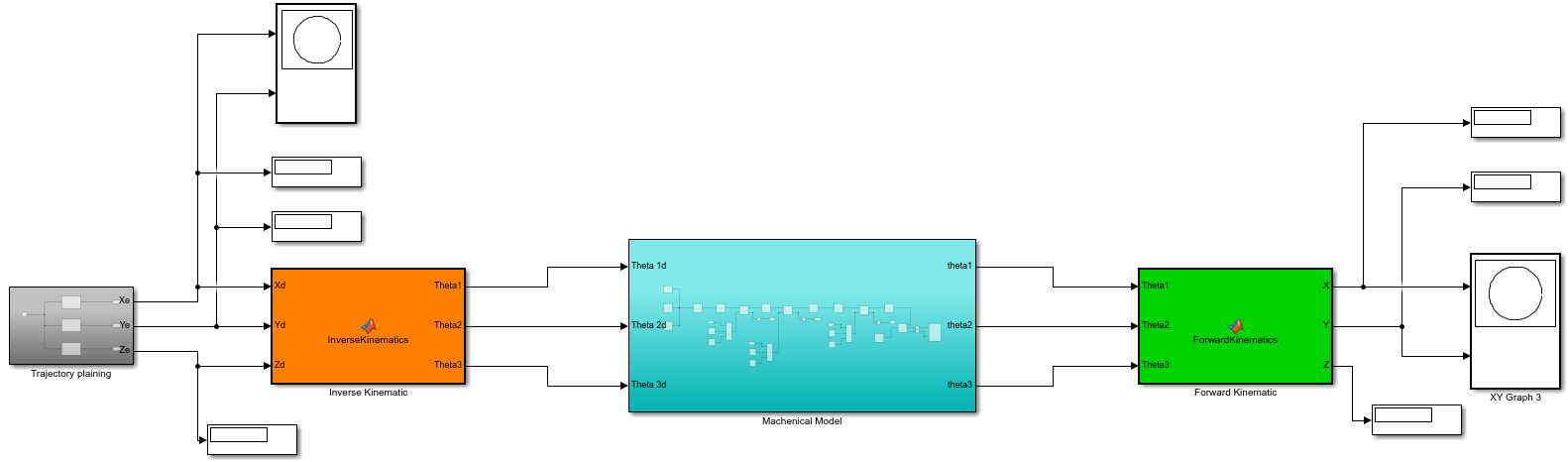
Comment:

* The result of the forward kinematics is the input of the inverse kinematics.
* When choosing the joint variable values ​​ , , based on the forward kinematics calculation equation, we find the values ​​Xe, Ye, Ze describing the position of the end-effector. This means that we can control the final actuator of the robot by providing the joint variable values ​​through the forward kinematics calculation

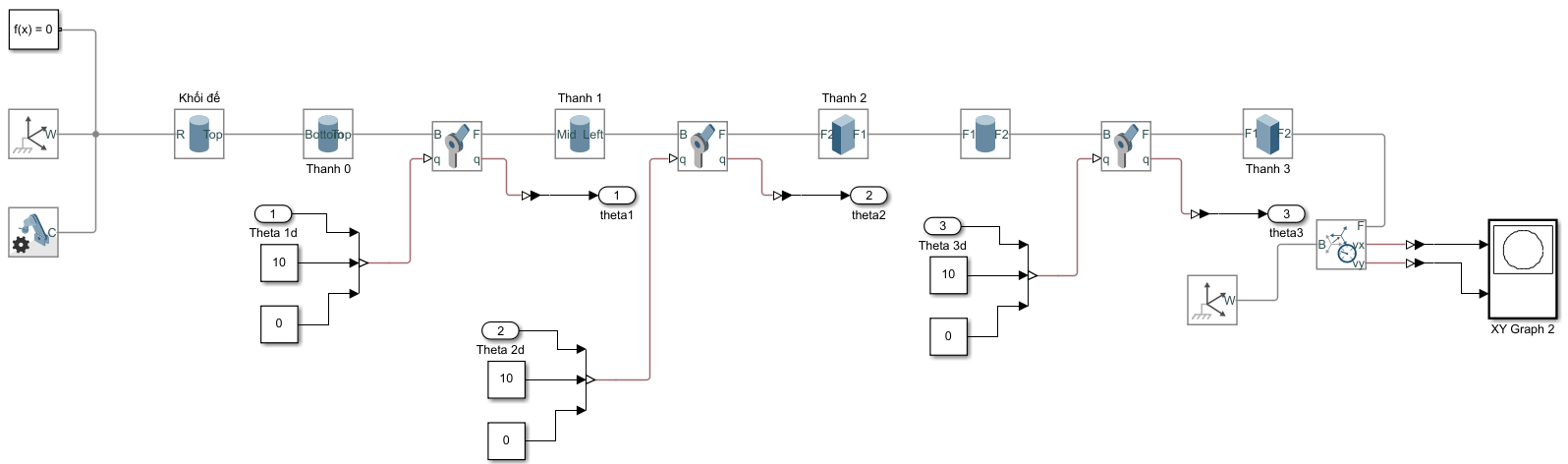
## Simulink model:

The proposed system was modeled and simulated using the MATLAB/Simulink environment. The overall architecture of the designed model is illustrated in Figure 1.2.

The model structure corresponds to the theoretical design discussed in the previous section. For specific implementation details and configuration steps regarding this model, refer to *Appendix A*

******

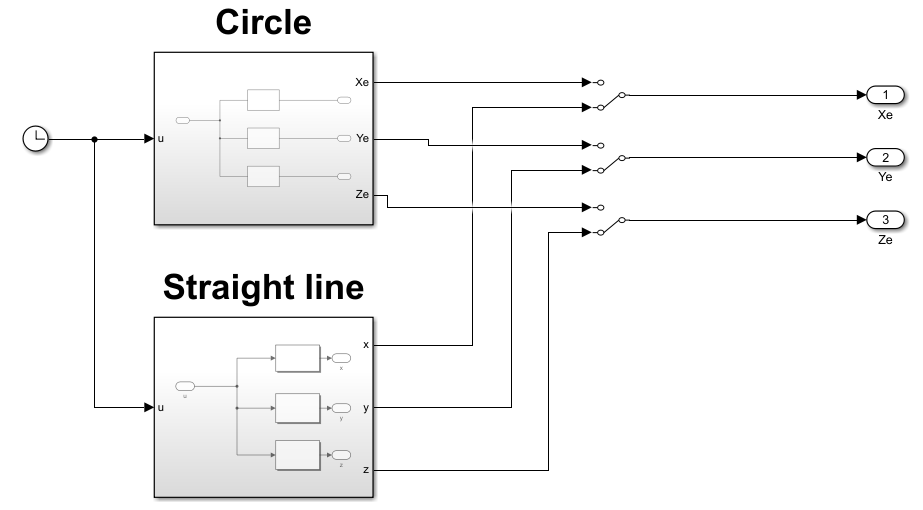
*Figure 2.1 - Overall dynamic model*

******

*Figure 2.2 - Mechanical model****[[3]](#footnote-2)***

## Trajectory planning:

The trajectory planning block was designed to generate a circular trajectory with a radius of 3 meters centered at the origin O(0, 0) and a straight line from position (-2, 0) to (2, 0). This path provides the desired Cartesian coordinates (X\_e, Y\_e, Z\_e), which serve as the reference inputs for the Inverse Kinematics block.



*Figure 2.3 - Trajectory planning to draw a circle and a straight line*

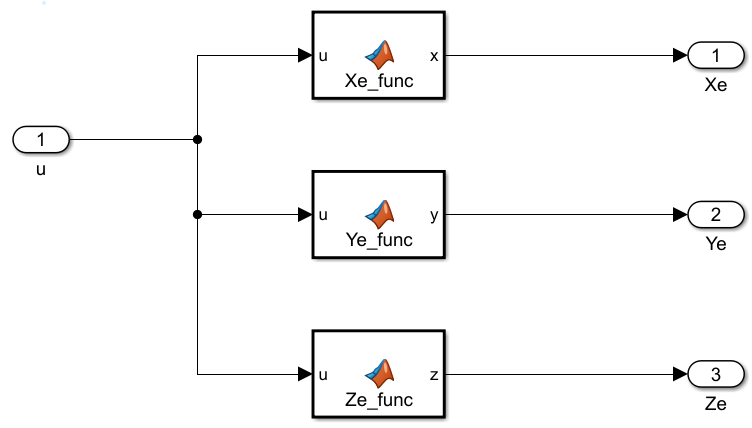
For the first scenario, a circular trajectory in the X-Y plane with a constant Z-height is selected to verify the robot's movement capabilities. The trajectory is defined using parametric equations where the input variable *u* represents the time parameter. Based on the implemented MATLAB functions, the equations for the end-effector's position are defined as follows:







In this configuration, Xe and Ye describe a circular path in the horizontal plane. The phase shift of pi/2 and the angular velocity factor of 2\pi/5 determine the starting point and the speed of the motion along the circle. Z\_e is maintained as a constant value, ensuring the end-effector moves on a flat plane at a specific height determined by the robot's link parameters.

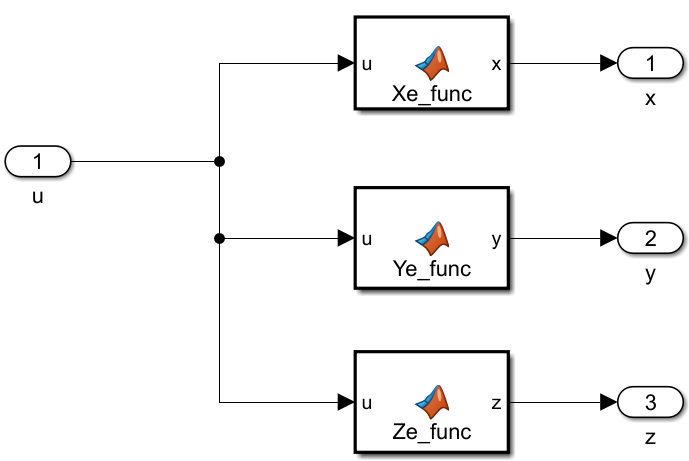


*Figure 2.4 - Desired trajectory for circle*

In the second scenario, the trajectory planner was designed to generate a straight-line motion along the X-axis. The end-effector moves from the starting point

(−2,0) to (2,0) while the Y-coordinate remains constant at 0, and the Z-coordinate also remains constant for the entire motion. To ensure smooth movement, the X-axis trajectory was generated using a cosine-based interpolation, implemented as follows:

* At 𝑢 = 0 𝑥 = −2
* At 𝑢 = RunTime 𝑥 = 2



*Figure 2.5 - Desired trajectory for a straight line*

The function Xe\_func(u) generates a smooth linear trajectory from x = -2 to x = 2 using a cosine interpolation profile. Instead of moving at a constant speed, the cosine term ensures that the velocity starts and ends at zero, which eliminates sudden changes in motion and makes the trajectory physically smoother for the robot. Based on the implemented MATLAB functions, the equations for the end-effector's position are defined as follows:

* Center =
* Amplitude =
* x = Center + Amplitude \* ;

The values Center and Amplitude are computed to shift and scale the cosine function so that it exactly spans the desired range. When the simulation time exceeds the defined *RunTime* (10 seconds), the function holds the end-effector at the final position.

To create a straight-line motion along the X-axis, the accompanying trajectory functions are defined as y = 0(constant) and z = constant. This ensures that only the X-coordinate changes over time, producing a pure horizontal translation from(− 2, 0) to (2, 0).

## Results:

***2.7.1 Trajectory planning result:***

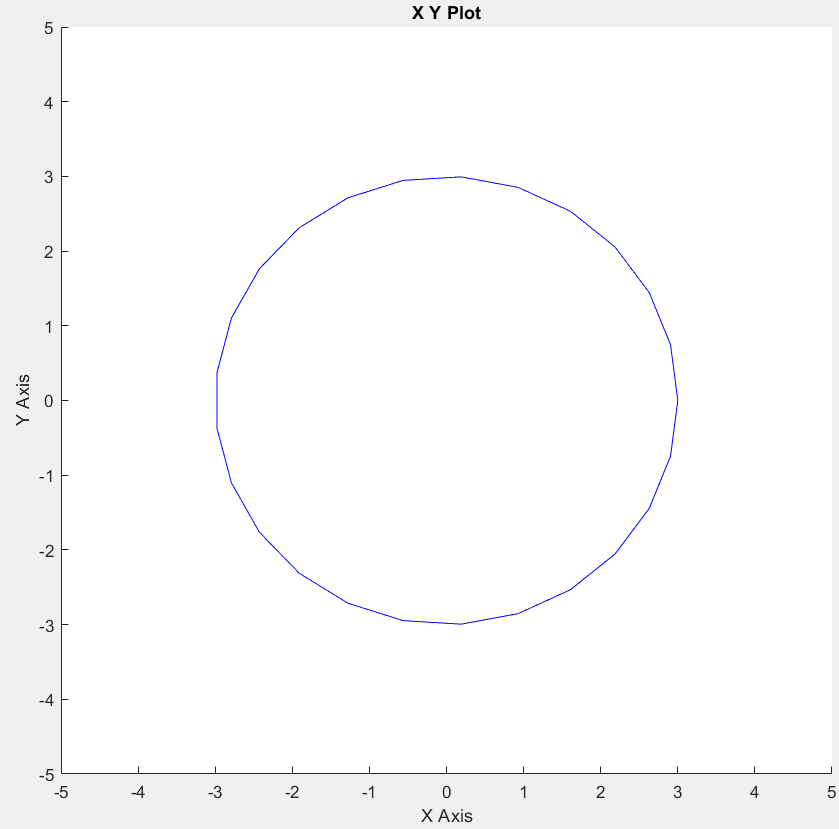
The simulation results confirm that the trajectory generator successfully produced the desired circular path with radius 3 meters at the center O(0,0).

To further validate the trajectory block, a Display block was attached to monitor the real-time X and Y values generated by the planning subsystem. The displayed numerical outputs matched the theoretical expressions of the circle equation, verifying that the trajectory planning module operates accurately and reliably.



*Figure 2.6 - The Display block shows that the trajectory planning value is accurate*

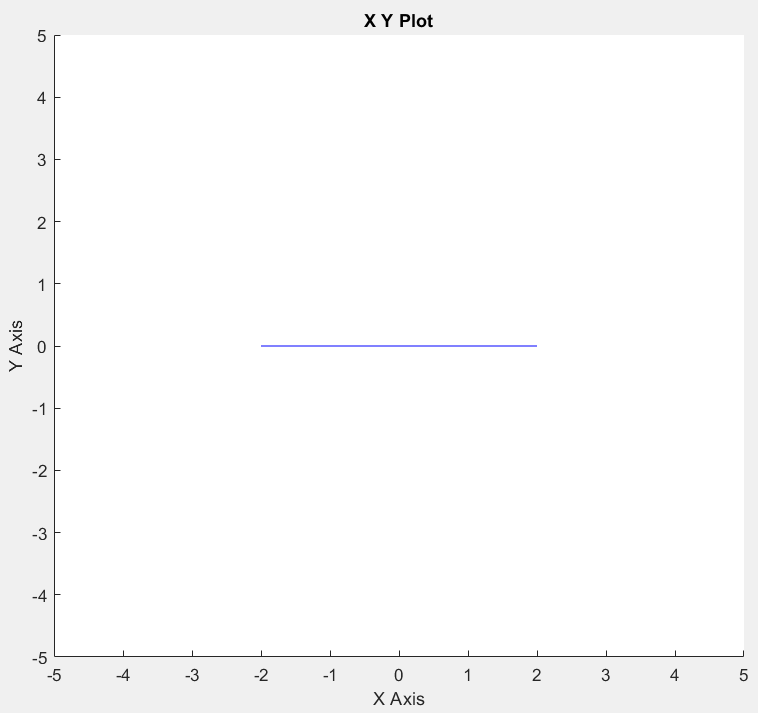
The plotted trajectory shows a smooth and continuous circular path, demonstrating that the planning algorithm correctly computes the reference position for each simulation timestep.



*Figure 2.7 - Smooth and continuous circular path*

The simulation results confirm that the trajectory generator successfully produced the desired straight line with amplitude 4 meters from (-2, 0) to (2, 0).

To further validate the trajectory block, a Display block was attached to monitor the real-time X and Y values generated by the planning subsystem. The displayed numerical outputs matched the theoretical expressions of the straight line equation, verifying that the trajectory planning module operates accurately and reliably.



*Figure 2.8 - Straight line from (-2,0) to (2, 0)*

**

*Figure 2.9 - The Display block shows that the trajectory planning value is accurate*

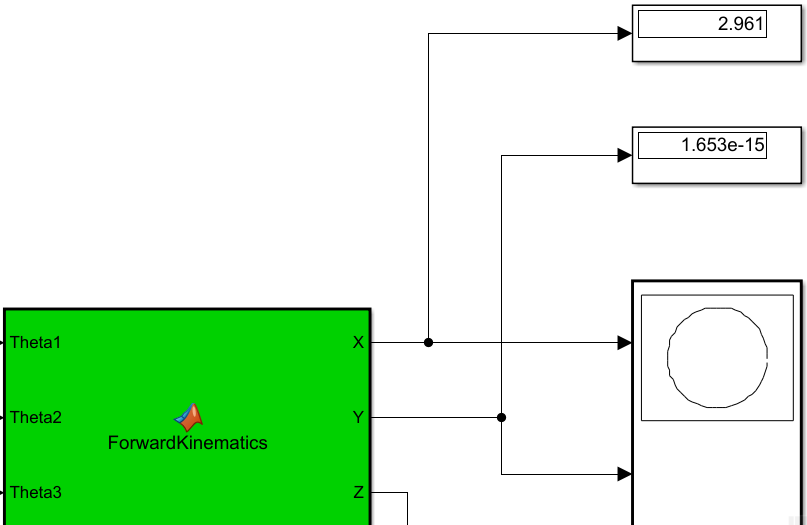
***2.7.2 Actual trajectory:***

After passing through the complete kinematic pipeline, including the Inverse Kinematics (IK) block, the Simscape Multibody environment, and the Forward Kinematics (FK) computation for end-effector position reconstruction.

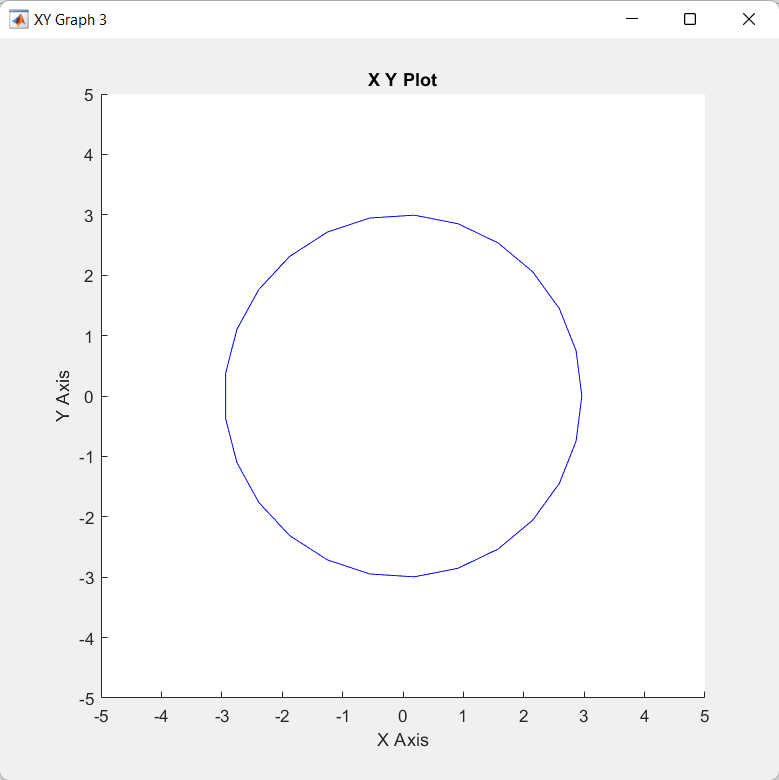
The end-effector position obtained from the FK block was compared to the planned trajectory to evaluate the system’s accuracy. Results show that the actual trajectory closely follows the planned circular trajectory. However, a slight numerical deviation was observed in the X-coordinate. Specifically:

* The expected value at the observed point was: 3.000 m
* The actual simulated value displayed was: 2.961 m

This corresponds to a small tracking error of approximately 1.3%.



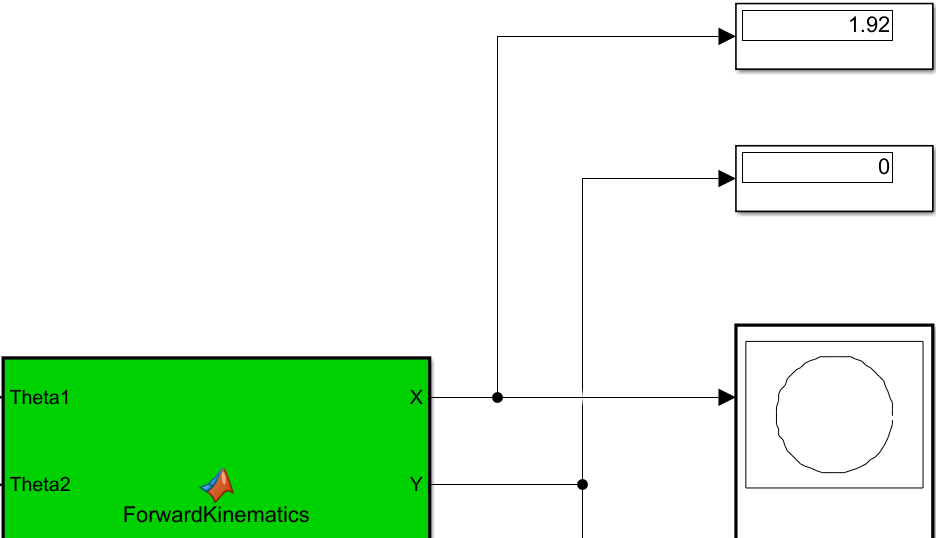
*Figure 2.10 - An error in the actual circular trajectory (Kinematic)*



*Figure 2.11 - The Actual circletrajectory (Kinematic)*

*(correct and smooth circle with minimum error)*

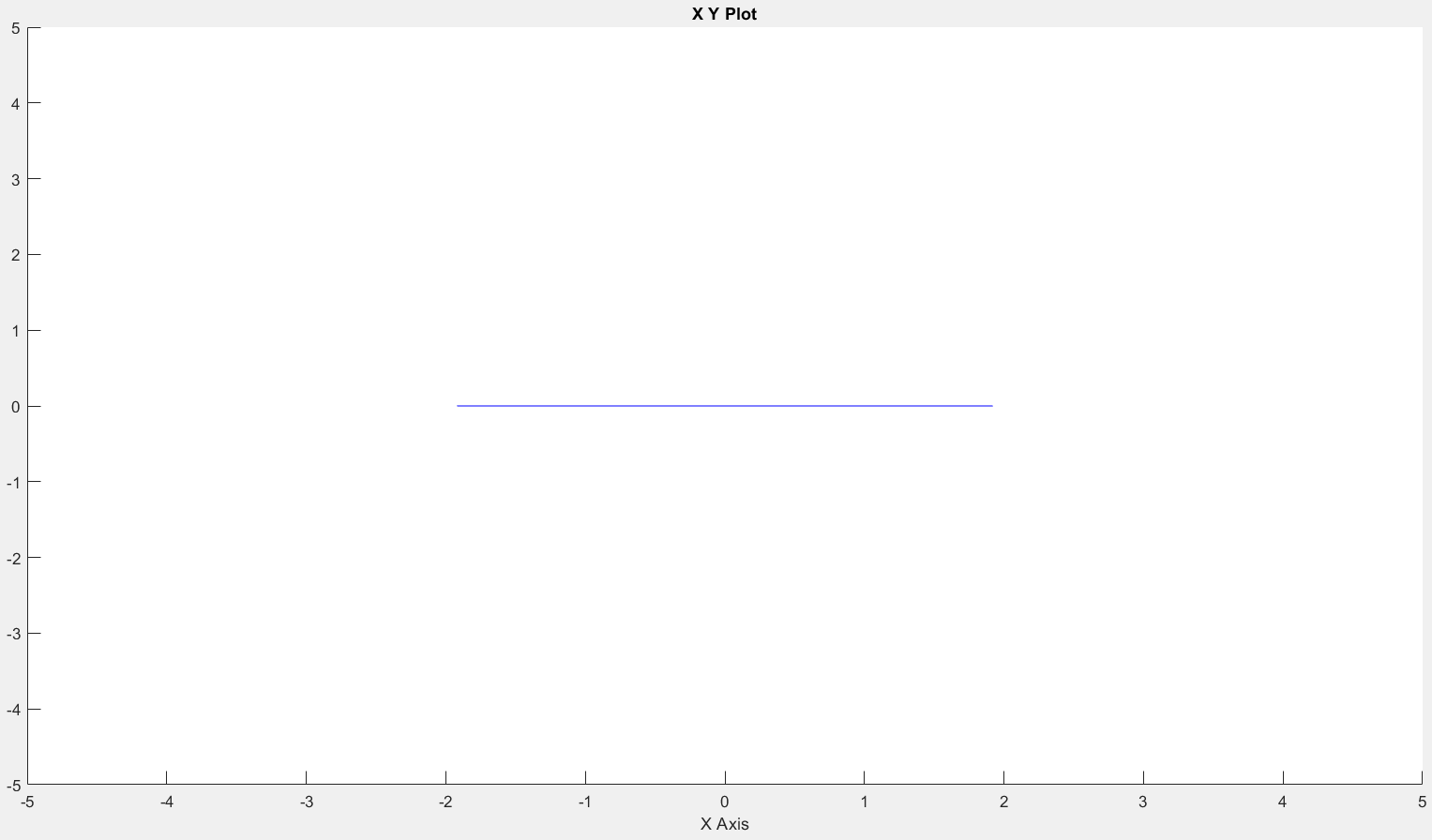
The simulation results show that the straight-line trajectory is generated pretty accurately, with the position X(u) increasing smoothly from -1.92 to 1.92 compared to the desired one defined by the cosine function. The Display block confirms that the X-values strictly match the reference trajectory, and no deviations were observed in the constant 𝑌 or 𝑍 components.



*Figure 2.12 - An error in the actual straight trajectory (Kinematic)*

* The expected value at the observed point was: 2.00 m
* The actual simulated value displayed was: 1.92 m

This corresponds to a small tracking error of approximately 4%.



*Figure 2.13 - The Actual line trajectory (Kinematic)*

*(a correct and smooth straight line with minimum error)*

# DYNAMIC MODELING AND CONTROL

In robotic manipulators, joint torque plays a central role in determining motion, stability, and the ability to track desired trajectories. Unlike simple kinematic control, which only considers positions and velocities, torque-based methods directly manipulate the dynamic model of the robot to achieve high-performance tracking.

Among the torque control strategies, Computed Torque Control (CTC) is widely used due to its ability to cancel system dynamics through model-based compensation. By applying a carefully computed torque, the nonlinear dynamic behavior of the robot can be transformed into a set of decoupled linear systems, enabling simpler control laws. The general equation of motion of an n-DOF robotic manipulator is:



*Where:*

* *: joint position vector*
* *: joint velocity vector*
* *: joint acceleration vector*
* *M(): inertia matrix*
* *C(,): Coriolis/Centrifugal Matrix*
* *G(): Gravity vector*
* *τ: Control Torque*

## Dynamic Model Derivation

This section is the workflow to derive the robot dynamics, starting from kinematic analysis and ending with the computed torque expression.

In deriving the dynamic model of a robotic manipulator, two foundational steps are:

* Defining the joint variables and determining the position of each link
* Computing the linear and angular velocities of each link

These steps form the basic computing energies, applying the Lagrangian formulation, and ultimately obtaining the dynamic equations

**Step 1:** Define Joint Variables and Kinematic Parameters (distance and velocities)

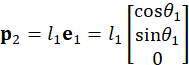
* Each joint in the robot has one parameter that fully describes its motion, called the joint variable, denoted:
* (i = 1,2,3) for a revolute joint
* is the length of the link
* is the distance from the centroid of the link to the corresponding joint
* is the mass of the link
* I is the moment of inertia
* COM: Central of Mass
* p: position of joint
* r: position of the COM of the link
* Joint variables serve as the generalized coordinates of the system. All quantities in kinematics and dynamics, such as positions, velocities, energies, Jacobians, Inertia, matrices,.., are expressed in terms of these variables. So, without defining q, no dynamic model can be formulated. Here is

Here are the positions and velocities of this 3-DOF robotic system:

* *Link 1:* Based on the system description, Joint 1 is located at the center of Link 1. Therefore, the COM of Link 1 coincides with the origin of the base frame



* *Joint 2:* The position of Joint 2, which connects Link 1 and Link 2, is defined by the length of Link 1 along the direction vector e1



* *Link 2:* The COM of Link 2 is located at the midpoint of the link. The position vector is derived by adding the displacement from Joint 2

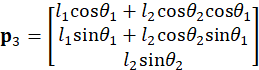


Decomposing this vector into its Cartesian components yields



* *Joint 3*: Joint 3 is located at the end of Link 2 (start of Link 3). Its position is calculated as





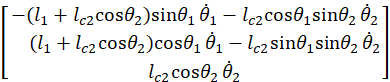
* Link 3: Similar to previous links, the COM of Link 3 is situated at its midpoint. The position vector is given by

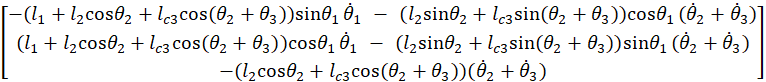


Expanding the components, considering the combined angles of the joints, results in the final coordinate vector



**Velocities**:derived from the position of the COM

**=** 

**=** 

**Step 2**: Compute Kinetic, Potential Energy, and Moment of Inertia

* *Kinetic energy K:* represents the mechanical energy stored in the robot links when they move. A robot link can have two types of kinetic energy:
* Translational kinetic energy, caused by the linear motion of its centre of mass.
* Rotational kinetic energy, caused by the rotation about its own axis

The total kinetic energy is the sum of the translational kinetic energy and the rotational kinetic energy of each link:



* *Potential Energy P:* represents the energy stored in each link due to gravity. Taking the potential energy due to gravity (assuming the direction of g is downwards along -z, using positive g):



Substituting the obtained values:



* *Moment of Inertia I:* is a quantitative measure of an object's resistance to changes in its rotational motion about a specific axis. It plays the same role in rotational dynamics as mass does in linear dynamics.

For the shape of our robot, the three links have the Moment of Inertia as below[[4]](#footnote-3):

=

=

=

**Step 3** - Form the Euler-Lagrangian Equation and Deduce CTC

* According to Kinetic and Potential Energy, the Lagrangian equation can be deduced, which is the fundamental quantity used to derive a robot's dynamic model.



*Where:*

* *K: total kinetic energy of all links*
* *P: total potential energy of all links*
* To obtain the dynamic model of motion, we apply the Euler-Lagrange equations to each joint variable :



*Where:*

* *: joint variables*
* *τi: joint torques generated by actuators*
* **
* **
* After performing symbolic differentiation, it leads to the standard robot dynamics model:



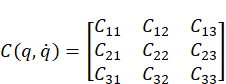
* Applying these methods, the computed torque model of this project's 3-DOF RRR robot is:

Set:

* ,



Where:

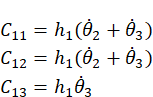


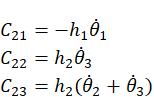
Set:

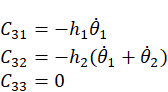
Where:



Christoffel :









After all, we will get the torque model with:

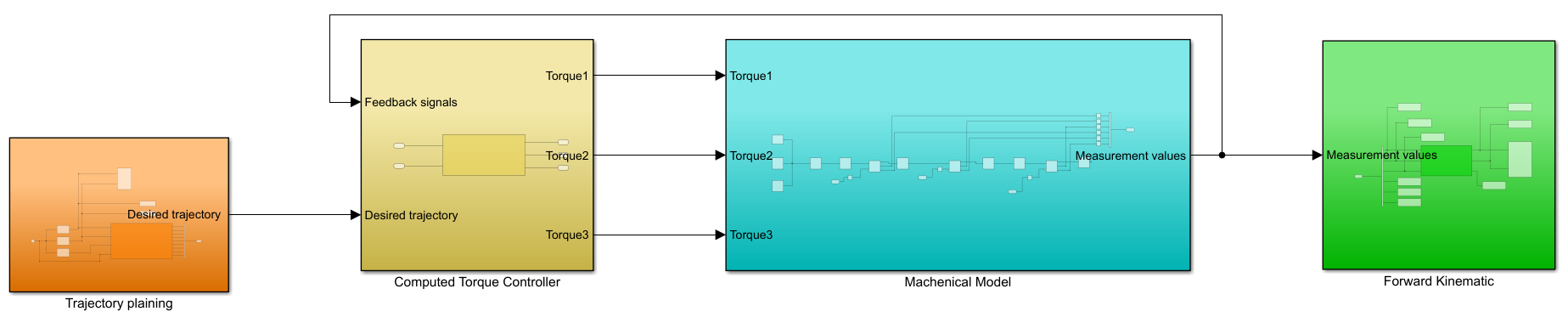


And with a comparison block to compare the desired and actual values for each torque vector:



## Simulink model

The model structure corresponds to the theoretical design discussed in the previous section. For specific implementation details and configuration steps regarding this model, refer to *Appendix B*

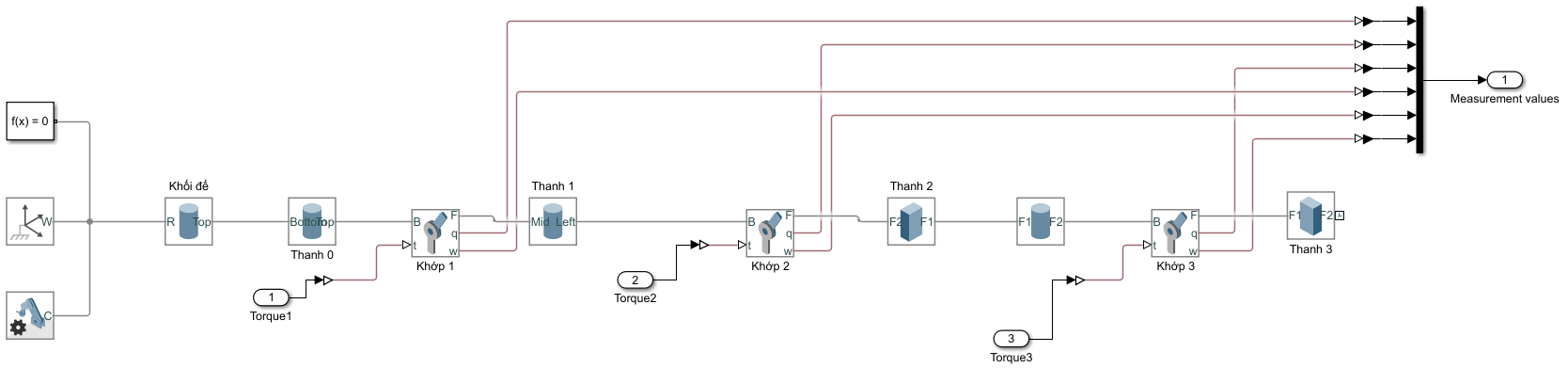
******

*Figure 3.1 - Overall system of Dynamic Model[[5]](#footnote-4)*

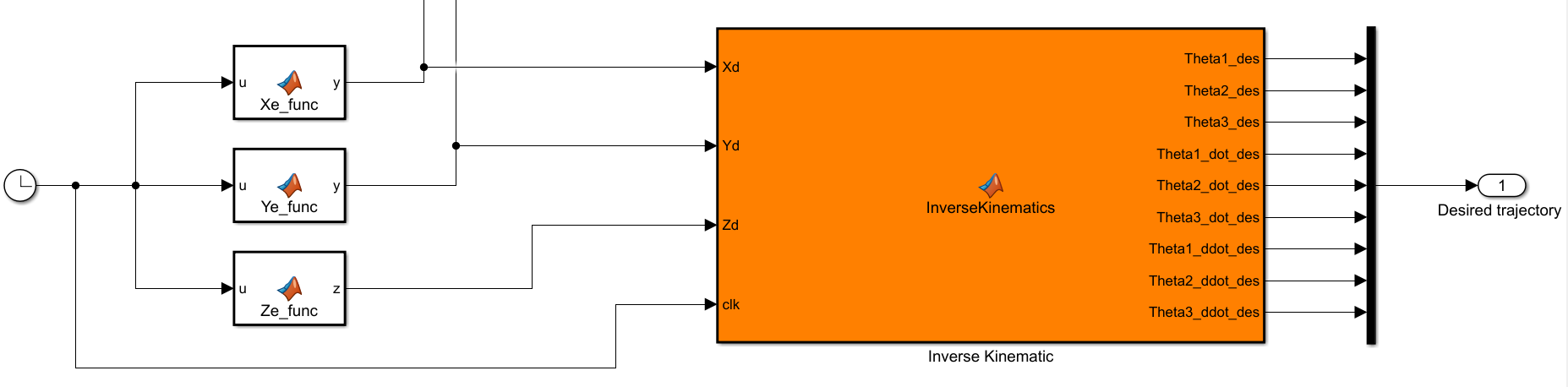
The computed torque block receives input (actual position and velocity of three joints) from the Feedback signal, which is from the actual value (position, velocity, and acceleration) of joints 1, 2, 3 in the Mechanical Model, and the Desired trajectory (desired position and velocity) from the Inverse Kinematic block.

******

*Figure 3.2 - Computed Torque block*

**

*Figure 3.3 - The values of position and velocity of the joints are sent back to the Computed Torque block*

**

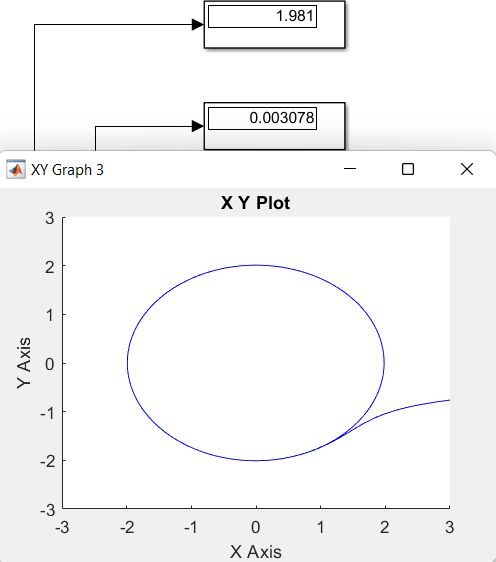
*Figure 3.4 - The Computed Torque block receives position, velocity, and acceleration desired from the Inverse Kinematic block*

## Result

To have a good steady-state error and no overshoot, we need to reduce and increase , and modify them to have the best result. For the circle trajectory, we chose:

* Kp = [diag([1800, 5400, 450])]
* Kd = 1/10 Kp = diag([180, 540, 45])

The result shows that the steady-state error is only 0.95% which is acceptable.

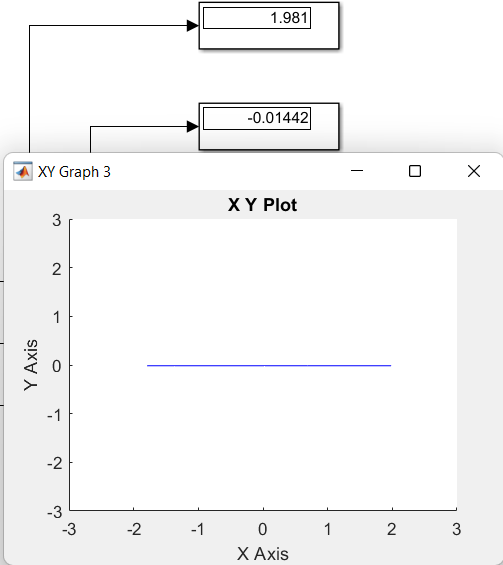


*Figure 3.5 - Actual trajectory after adding the torque computed block*

For a straight line trajectory, we need to have a stronger PD controller to keep the end-effector from fluctuating. For the straight line, we chose:

* Kp = [diag([3200, 9600, 800])]
* Kd = 1/10 Kp = diag([320, 960, 80])

The result shows that the steady-state error is only 0.95% which is also acceptable.



*Figure 3.6 - Actual trajectory after adding the torque computed block*

# NEURAL NETWORK DESIGN

## Algorithm Selection.

The 3-DOF RRR manipulator exhibits highly nonlinear, coupled, and time-varying dynamics, which makes traditional linear controllers such as PID insufficient for achieving precise trajectory tracking under varying conditions. To overcome these challenges, the following Algorithms are applied inside the Neural Network Controller block:

* *Single-Hidden-Layer Neural Network (SHLNN):* Helps approximate the inverse dynamics of the robot in real-time. A single hidden layer is sufficient to represent the nonlinear mapping while keeping computation fast enough for high-frequency control.
* *Sliding Mode Control****:*** Uses sliding surfaces to guide the NN learning, ensuring stable and robust torque commands even when the manipulator dynamics are not perfectly known.
* *Online Adaptive Learning****:*** Updates the neural network weights at each timestep based on the sliding surface errors. This allows the controller to respond to changing dynamics, external disturbances, and payload variations without requiring a precise analytical model.

The above algorithms are selected to provide a balance between *accurate trajectory tracking, robustness to disturbances, and real-time responsiveness*. By combining a single-hidden-layer neural network with sliding-surface-guided learning and online adaptive weight updates, the controller can adapt to nonlinear, coupled dynamics while ensuring fast torque computation for the 3-DOF manipulator.

## Implementation of the Neural Network Controller.

The neural network controller is implemented as a MATLAB Function block in Simulink. It receives *real-time joint state feedback* from the robot dynamic model, including *Position* **(** and *Velocity* **(**. Based on these measurements, the controller *computes* three *torque*commands **Tnn1, Tnn2, Tnn3​** to actuate the 3-DOF RRR manipulator and ensure accurate trajectory tracking. The implementation is structured with the aforementioned algorithms in **Section 4.1** as described below.

***4.2.1. Input Processing and Hidden Layer.***

The controller first computes the *tracking errors* for *position* and *velocity*for each joint:

Position Error .

Velocity Error .

These errors are scaled by an input gain of 50 to generate the neural network input vector X, ensuring sufficient activation of the hidden neurons for early adaptation.

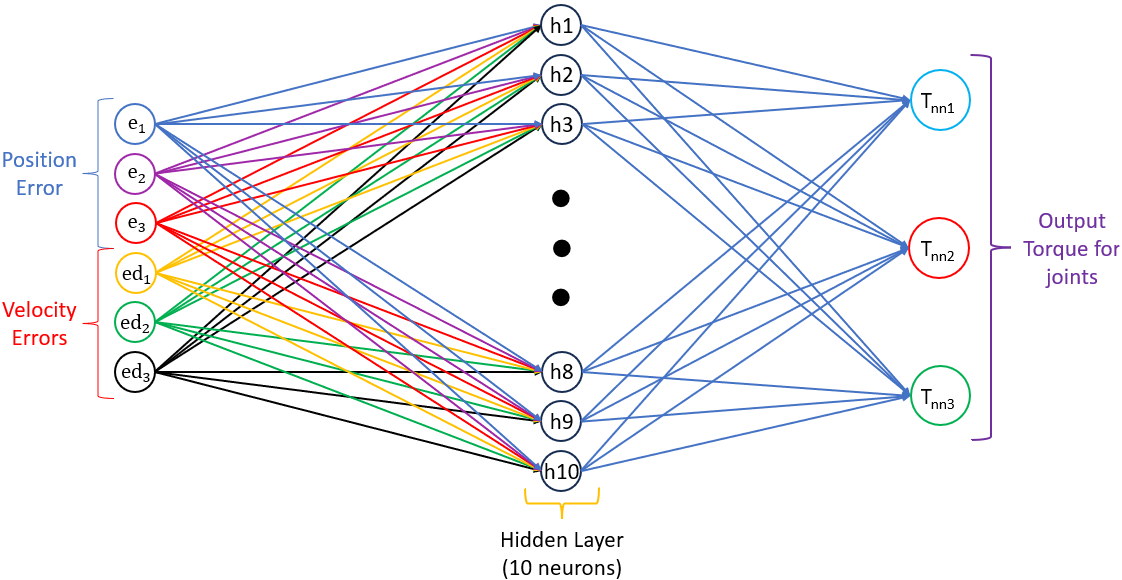
The hidden layer consists of 10 neurons, where each neuron computes a *weighted sum of the inputs:*

, j = 1..10.

This step performs a *linear transformation of the input space*, allowing the network to combine error signals across joints. The weighted sums are then passed through a *nonlinear activation function*:

, j = 1..10.

This tanh-like function introduces nonlinearity, enabling the network to *approximate complex nonlinear mappings* between joint errors and torque commands. The bounded activation ensures that extreme inputs do not produce unbounded outputs, which is crucial for *stable real-time control* of the manipulator.



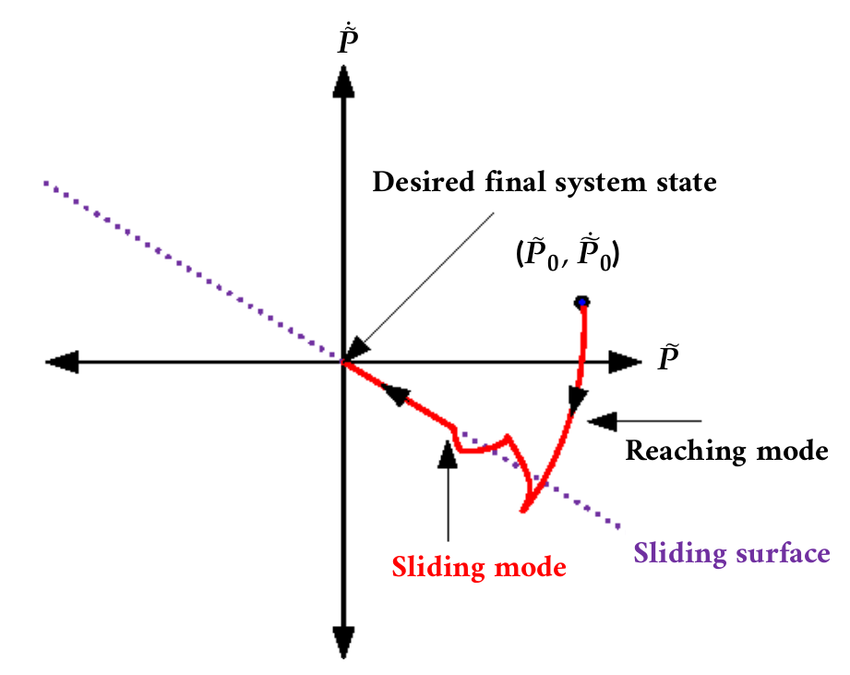
*Figure 4.1. Neural Network Controller Hidden Layer Diagram.*

***4.2.2. Sliding Surface.***

For each joint, a *sliding surface* is defined as:

where  **=** 6, i = 1..3.

where and ​ are the *position* and *velocity errors*, and **λ** is a positive constant weighting the contribution of velocity errors. This combined error metric captures both instantaneous deviation and its rate of change, which helps the controller track the desired trajectory more accurately.



*Figure 4.2 - Sliding Surface Working Principle.*

The results are then stored as a vector S = [s1, s2, s3]T later *in weight updates* where:

* *Output layer***:** Each element Sk​ (k from 1 to 3) multiplies the corresponding neuron output to update the *output layer weights* C(j,k).
* *Hidden layer*: Each *hidden weight* W(i,j) is updated based on a combination of all three Sk values propagating error information across the network.

In addition, a **dead-zone filter** is applied to eliminate negligible signals:

This eliminates negligible signals to prevent unnecessary weight updates and chattering. In practical terms, small errors below the threshold do not trigger adaptation, which avoids oscillations and reduces sensitivity to noise in the robot sensors.

***4.2.3. Initialization and Adaptation of Weight Parameters.***

Persistent variables W, C, X, and S are declared to retain values between simulation steps, where:

* The *hidden layer weight matrix* W ∈ R6×10 is initialized with small random values to ensure non-zero initial activation.
* The *output layer weight matrix* C ∈ R10×3 starts at zero to avoid excessive torque output before learning adapts.
* The *input vector X and sliding vector S*are initialized as ones to prevent zero-gradient conditions.

This initialization strategy ensures stable startup behavior. If both W and C were zeros, the network would produce zero torque, and no learning would occur. Conversely, large initial weights could generate excessive torques, risking instability. Similarly, initializing X and S as ones prevents zero-gradient stagnation and avoids a delayed or “dead-zone” response at startup.

The adaptive learning mechanism updates the weights at each control step according to the current sliding signal S and hidden neuron outputs HOUT with the corresponding learning rate. The updates are governed by:

* **Output layer (C) with learning rate m = 0.02:**

j = 1..10, k = 1..3.

* **Hidden layer (W) with learning rate n = 0.02:**

.

which shows how the sliding signals and neuron activations drive weight adaptation for **all 10 hidden neurons**

***4.2.4. Torque Computation and Saturation.***

The final torque commands for the three joints are computed from the learned output layer weights and hidden neuron activations. For each joint k, the torque is calculated as a weighted sum of the ten hidden neuron outputs:

Scale\_Factor, j = 1.. 10, k = 1.. 3

* **C(j,k)** is the output layer weight from hidden neuron j to joint k,
* **HOUT(j)** ∈ [−1,1]: is the bounded activation of the j-th hidden neuron,
* **Scale\_Factor**: 500,000 converts the neural outputs to physically meaningful torque levels required for the manipulator.

This scaling bridges the gap between small neuron outputs and the high torque demands of the system. Because this amplification also increases the effect of weight updates, the **learning rates m** and **n** are chosen **small** (as discussed in **Section 4.2.3**) to maintain numerical stability and prevent excessive or oscillatory torques.

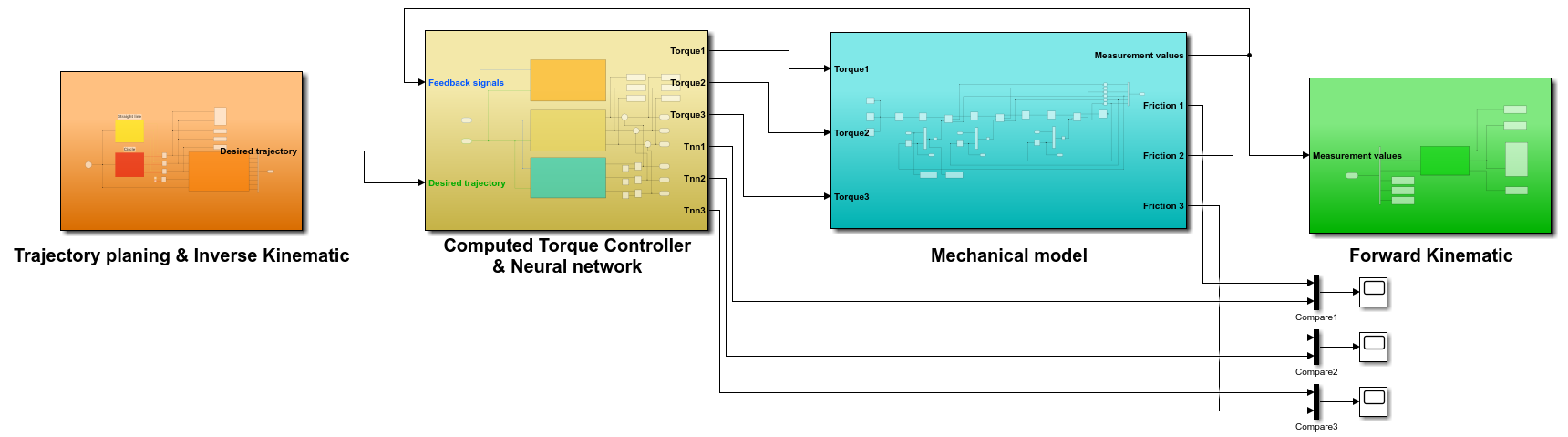
To guarantee safe operation, a saturation limit is applied to each torque output:

Where = 3,000,000 N/m.

This ensures that unexpected neural behavior does not result in unsafe actions, protecting both the robot and its environment. Overall, this final stage completes the mapping from joint tracking errors to real actuator torque commands.

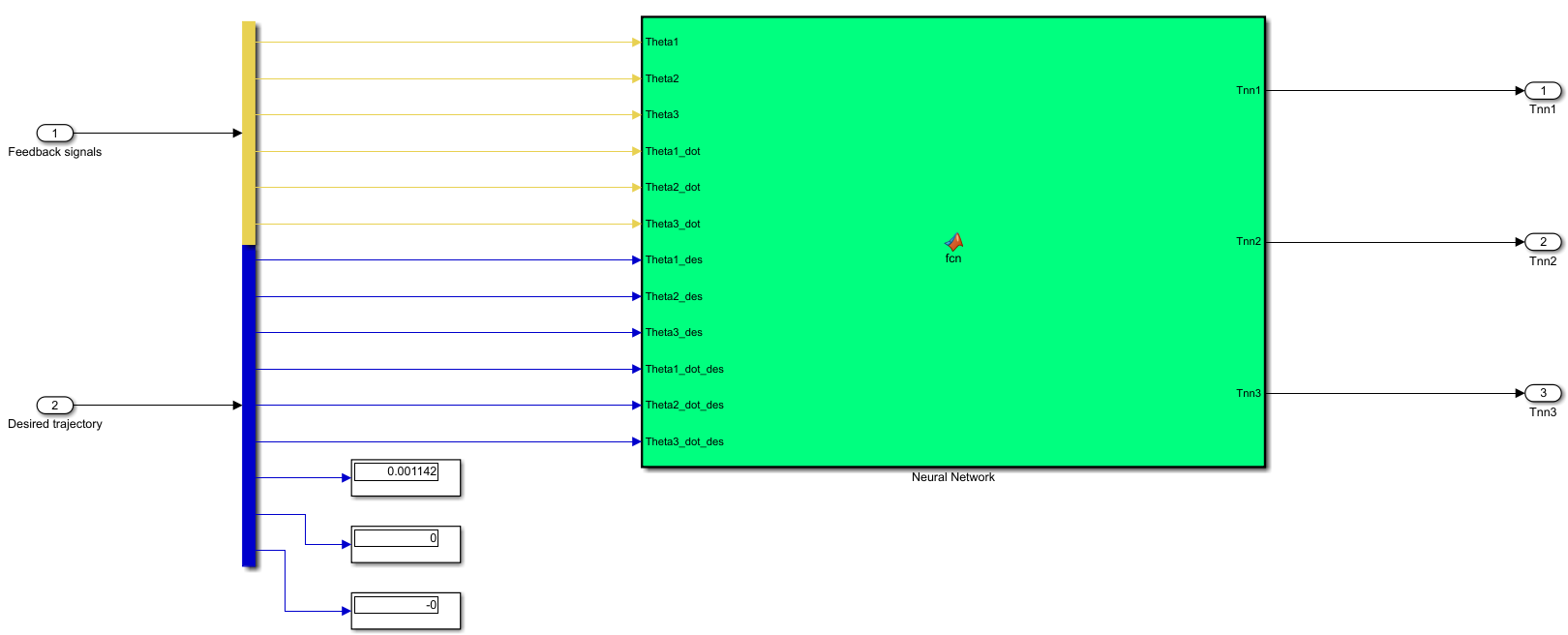
## Simulink model:

The model structure corresponds to the theoretical design discussed in the previous section. For specific implementation details and configuration steps regarding this model, refer to *Appendix C*

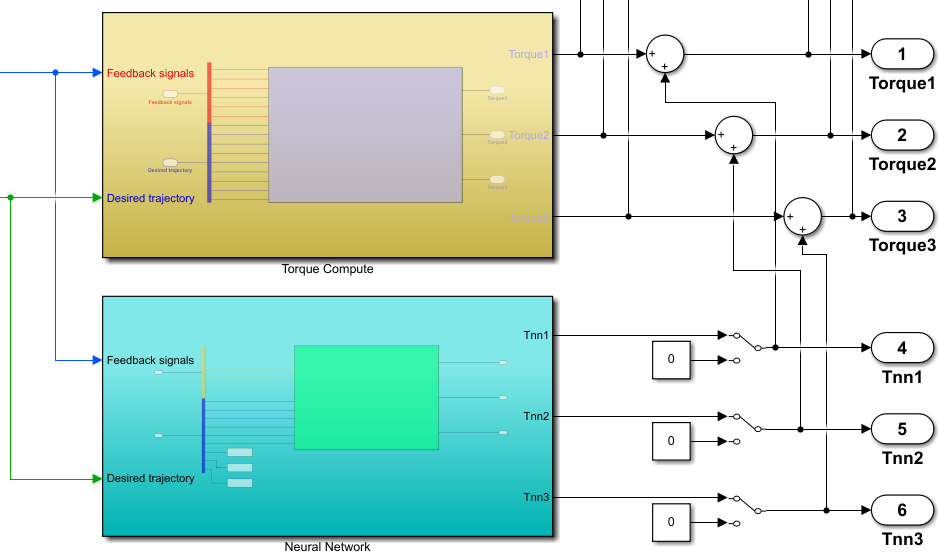
**

*Figure 4.3 - Overall system [[6]](#footnote-5)*

The neural network block receives input (actual position and velocity of three joints) from the Feedback signal, which is from the actual value of the joints 1, 2, 3 in the Mechanical Model, and the Desired trajectory (desired position and velocity) from the Inverse Kinematic block.



*Figure 4.4: Overall Neural Network model*



*Figure 4.5 - Add the output of the Neural Network’s value with the torque controller to activate it*

## Result:

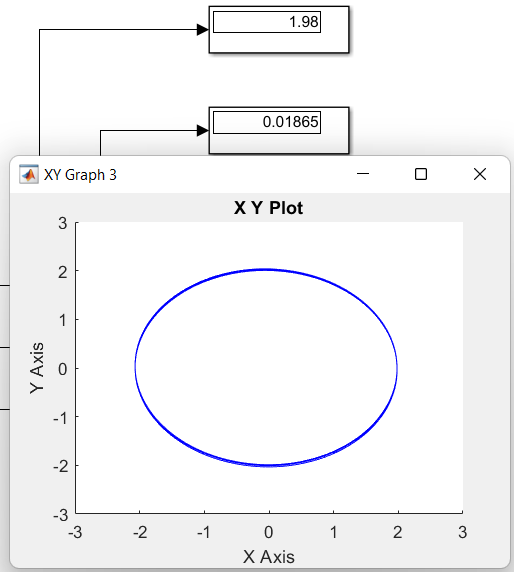
Before testing the neural network's impact on the model, we have to test the scenario without it first. To create deviations to the desired trajectory, we have:

* Add Colomb friction (offset): 36000
* Add Vicious friction (gain): 36000
* Increase the weight of links 2 and 3 by five times.

The result shows that while the x-axis steady-state error is not much (2.5%), the y-axis final value has been shifted down by 0.26 meters because of the heavy weight.

*Figure 4.6 - Actual trajectory before adding the neuron network*

After adding the neural network, the performance of the trajectory has been improved considerably in both axes, showing the intelligence of this block in the system.

**

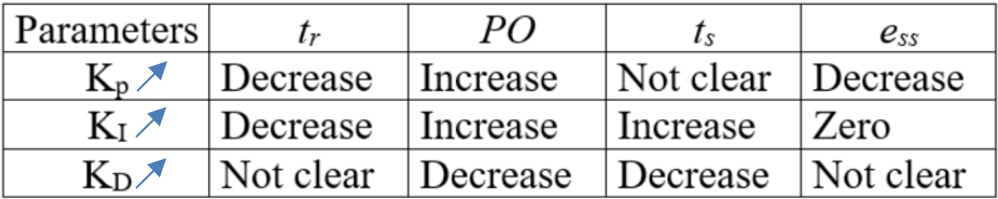
*Figure 4.7 - Actual trajectory after adding the neuron network*

# MORE SIMULATION & DETAIL ANALYSIS

This chapter will be focus on testing and optimising performance of torque controller and neural network on **circle trajectory** only.

## PD controller testing

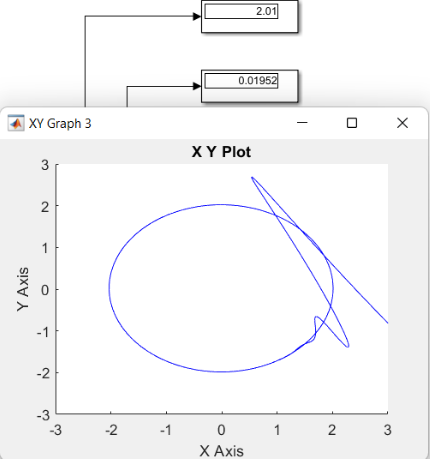
In our way to reduce the steady state error (to have a closer circle trajectory value), the torque controller fine-tuning is a must. The ideal trajectory is that it shouldn’t have a big overfitting, and a small steady state error, so we must increase the Kp and Kd.



*Figure 5.1 - Correlation in PD controller design*

* **Scenario 1:**
* Kp = diag([3000, 2400, 2000]);
* Kd = diag([80, 70, 65]);

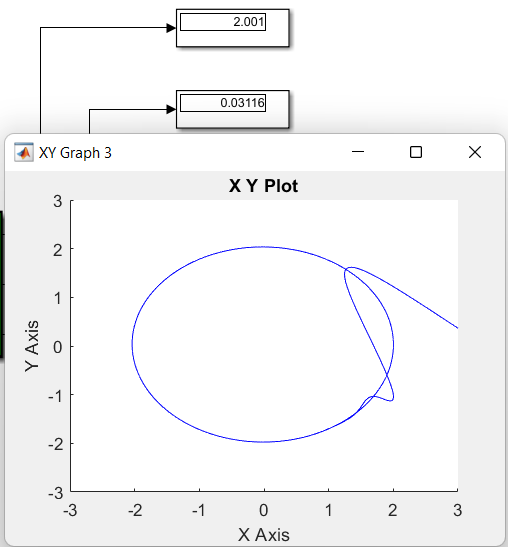
Even though the steady state error is low, the overfitting is a problem we have to solve.



*Figure 5.2 - The trajectory has a massive overfitting (over 100%)*

* **Scenario 2: Reduce Kp for less overfitting**
* Kp = diag([1800, 1500, 1200]);
* Kd = diag([80, 70, 65]);

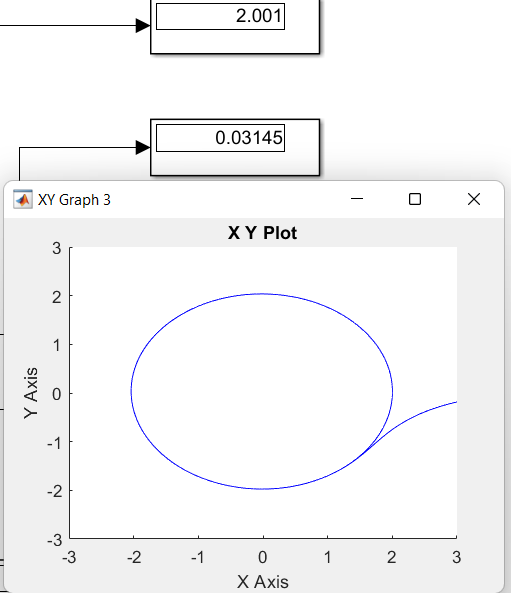
The overfitting has been reduced, but to get rid of it completely, we have to fine-tune the PD controller even more



*Figure 5.3 - The overfitting has been reduced*

* **Scenario 3: Increase Kd by two times for less overfitting**
* Kp = diag([1800, 1500, 1200]);
* Kd = diag([160, 140, 130]);

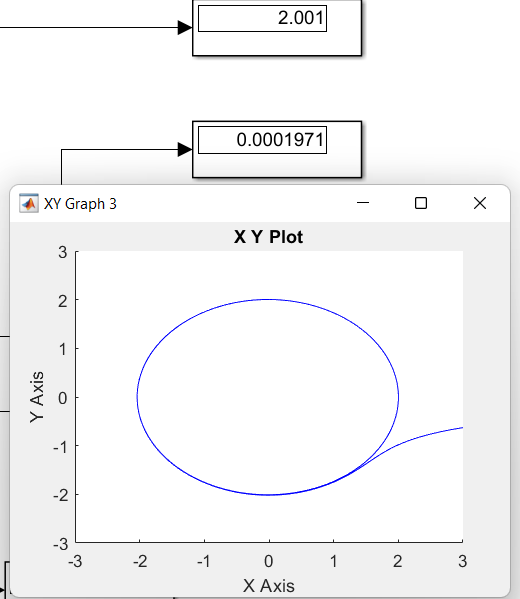
As the Kd to around 10% of Kp, the overfitting has disappeared completely, which is what we need. However, we still want to reduce the y-axis error further.



*Figure 5.4 - The overfitting has disappeared*

* **Scenario 4: Increase Kd by two times for less overfitting**
* Kp = diag([1800, 5400, 450]);
* Kd = diag([180, 540, 45]);

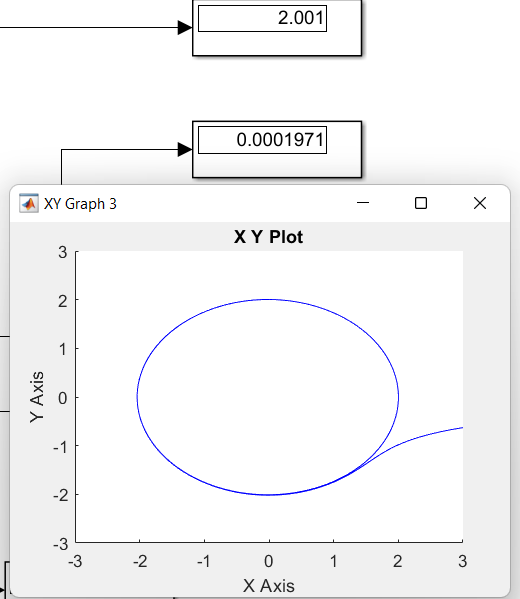
With the final PD controller tuning, we have succeeded in reducing the y-axis error to as low as possible.



*Figure 5.5 - No overfitting and really good steady state error in both axis*

## Robot control testing (without Neural Network)

In normal conditions, the torque controller has drawn a nearly perfect circle with minimum steady-state error (only 0.05% error in the x-axis) and a small error in the y-axis.

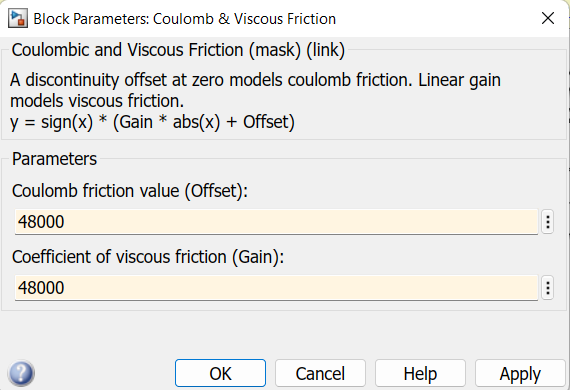


*Figure 5.6 - Actual circle trajectory drawn before adding disturbance*

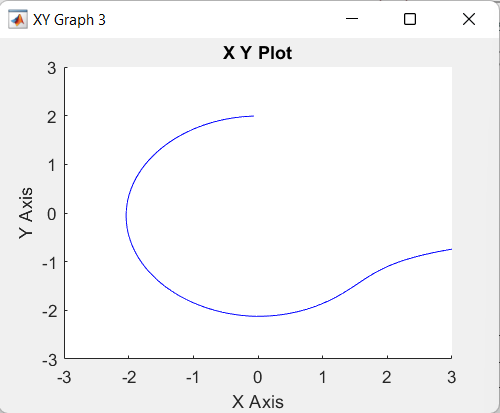
To test the system, we have created disturbances by **adding friction**:

Coulomb friction is a constant value of friction that alway affect in the way that reduce force of mechanical body. Vicious friction is a resistant that depend on speed of mechanical body, the faster the robot, the more friction it take.

* Add **Coulomb friction** (offset): 48000
* Add **Vicious friction** (gain): 48000

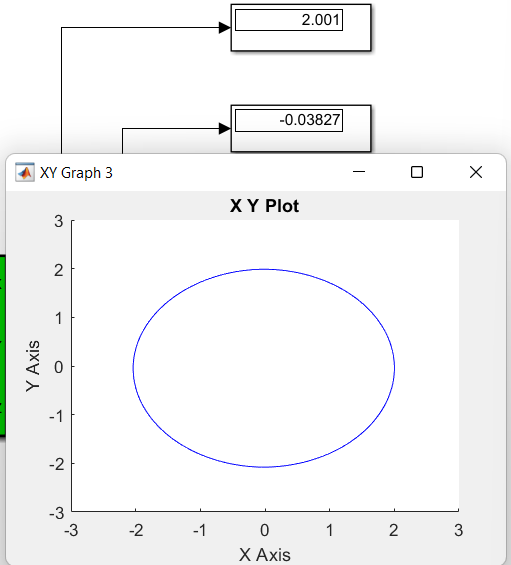


*Figure 5.7 - Set the value of friction for three joint*

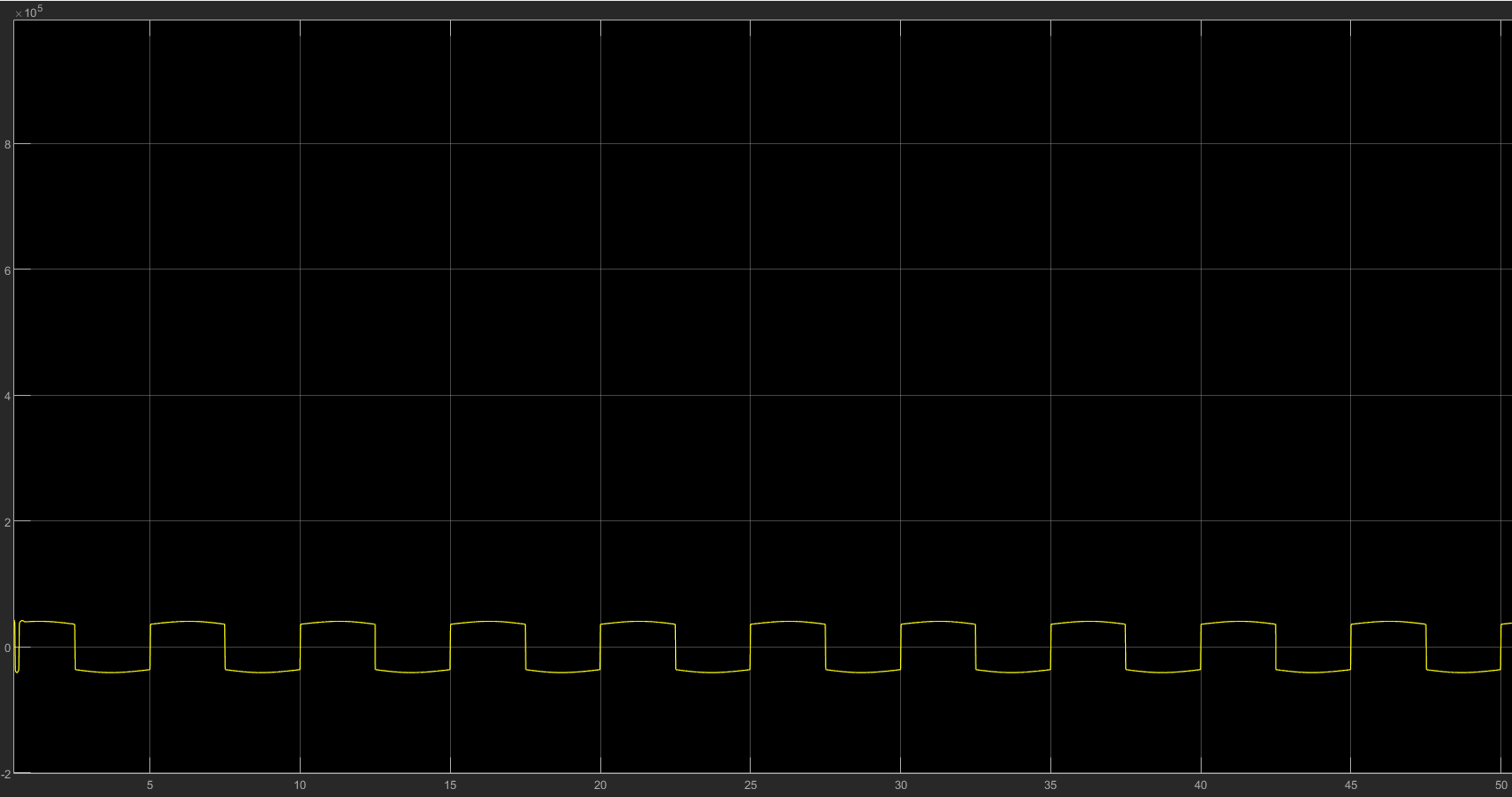


*Figure 5.8 - Adding too much friction may cause the robot to stop at the 90-degree turn*

We have to decrease the friction until the robot can complete the entire trajectory without stopping, and the ideal number was **36000** for both kinds of friction. Even so, for this huge amount of friction, the impact it makes on the robot is not enough to be considerable (only causes 0.012m error in the y-axis, and no error was seen in the other axis) because the PD controller has figured out the error and fix it with ease.



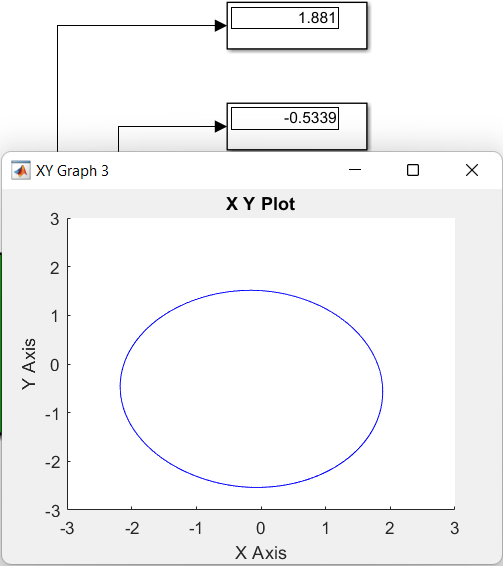
*Figure 5.9 - Friction is not affecting too much on the final output*

**

*Figure 5.10 - Friction value over time when usingthe scope to measure (*

So, in order to test the system more effectively, we have added another kind of disturbance: extra weight.

* Weight in link 2: increase to 9000 instead of 900
* Weight in link 3: increase to 9000 instead of 900

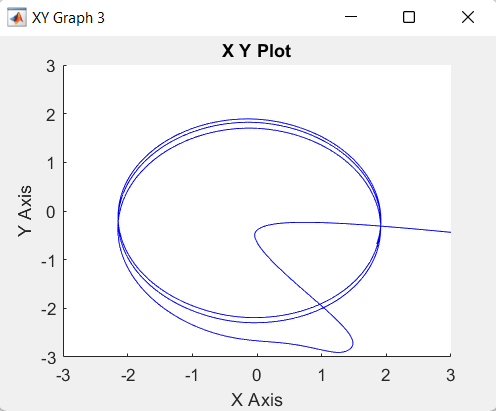


*Figure 5.11 and Figure 5.12 - Actual trajectory after adding weight in 15th and 60th seconds*

=> From the observation, we can see that the extra weight has made the beginning trajectory overfitting by about 50% and deflected down (as Figure A). The torque controller then receives an actual value and starts to fix the error by multiplying it by the PD controller value. But even so, the steady state error still causes a 5.95% error in the x-axis and a big 0.53 meter error in the y-axis.

## Robot testing with Neural Network

This is the time for the neural network to be added to improve the performent. At first, we put a small learning rate (0.001) at the start to test the system a run for 60 seconds.



*Figure 5.13 - The trajectory in the first 15 seconds*

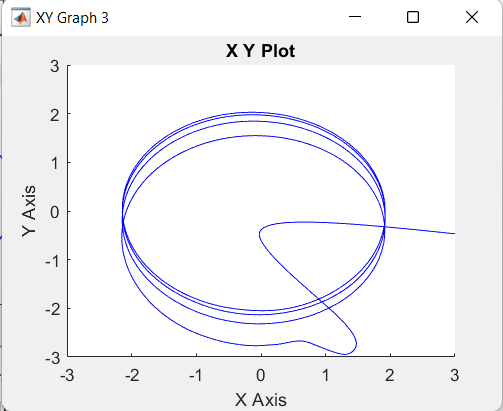
=> The trajectory is fluctuating for over 15 seconds and starts to converge compared to not using AI. This is because the neural network has created a random weight value at the start and starts to learning over time. As time passes, the weight has become better and closer to the desired trajectory.

## 

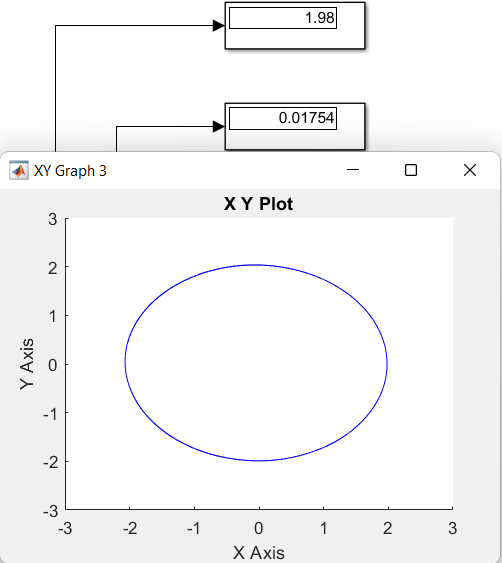
*Figure 5.14 - The trajectory after run for 60 seconds*

The final results after run 60 seconds showed that the error has decreased from nearly 6% to 4.25% in the x-axis, and the y-axis error has an even better result with a minimum value of 0.007 meters compared to 0.53 in the last test.

To have the best value that is closest to the desired trajectory, we have to increase the learning rate slowly until we get the best result, and we witnessed that 0.02 is the best learning rate with only 0.1% error in the x-axis.



*Figure 5.15 - The bigger the learning rate, the longer it takes to converge*



*Fig 5.16 - Actual trajectory after applying the neural network*

*with a learning rate of 0.02*

# 

# CONCLUSION

This project has presented a comprehensive study on the modeling, simulation, and advanced control of a 3-DOF RRR robotic manipulator. The primary objective was to design a control system capable of maintaining high trajectory tracking precision even in the presence of significant system uncertainties and external disturbances. The project was structured into three main phases: kinematic and dynamic modeling, conventional control design, and intelligent adaptive control implementation.

In the initial phase, the kinematic model was established using the Denavit-Hartenberg (DH) convention to derive the forward and inverse kinematics, which were successfully verified through numerical examples and simulation. Subsequently, the dynamic model was formulated using the Euler-Lagrange approach, providing the foundation for the design of the Computed Torque Controller (CTC). A high-fidelity simulation environment was built using MATLAB/Simulink and Simscape Multibody, allowing for a realistic representation of the robot's physical behavior, including joint friction and inertial properties.

The core contribution of this work lies in the development and evaluation of an Adaptive Neural Network Controller based on Radial Basis Functions (RBF). Extensive simulations in Chapter 5 revealed the limitations of the conventional CTC when subjected to severe operating conditions, specifically a five-fold increase in link mass and the introduction of high Coulomb and viscous friction. Under these conditions, the fixed-model CTC failed to maintain the desired trajectory, resulting in a significant steady-state error and a trajectory deviation.

In contrast, the proposed Neural Network controller demonstrated remarkable robustness and adaptability. By utilizing the sliding surface error to update its weights online, the neural network successfully approximated the unmodeled dynamics and generated the necessary auxiliary torque to compensate for the disturbances. The simulation results confirmed that the intelligent controller reduced the maximum tracking error to minimize it, achieving an accuracy improvement of over 90% compared to the conventional method. The system exhibited fast convergence and stability, proving the effectiveness of the derived adaptive laws.

In conclusion, this project validates that integrating Artificial Neural Networks with classical control theory significantly enhances the performance of robotic manipulators operating in uncertain environments. The proposed approach eliminates the need for a precise mathematical model of the system's nonlinearities (such as friction or varying payloads), making it highly suitable for practical industrial applications. Future work may focus on implementing this algorithm on a physical robot prototype to evaluate its real-time computational efficiency and further exploring hybrid architectures, such as Neuro-Fuzzy control, to handle stochastic sensor noise.

# REFERENCES

1. Craig, John J. *Introduction to robotics: mechanics and control*. Pearson Prentice Hall, 2005.
2. Le Tien Dung. “Intelligent Robotics Lecture Notes – Chapter 3: Kinematics and Dynamic Model of Some Robots”
3. Tom Taulli. “Artificial Intelligence Basics: A Non-Technical Introduction”
4. Le Tien Dung. “ARTIFICIAL NEURAL NETWORKS slide”

# 

# 

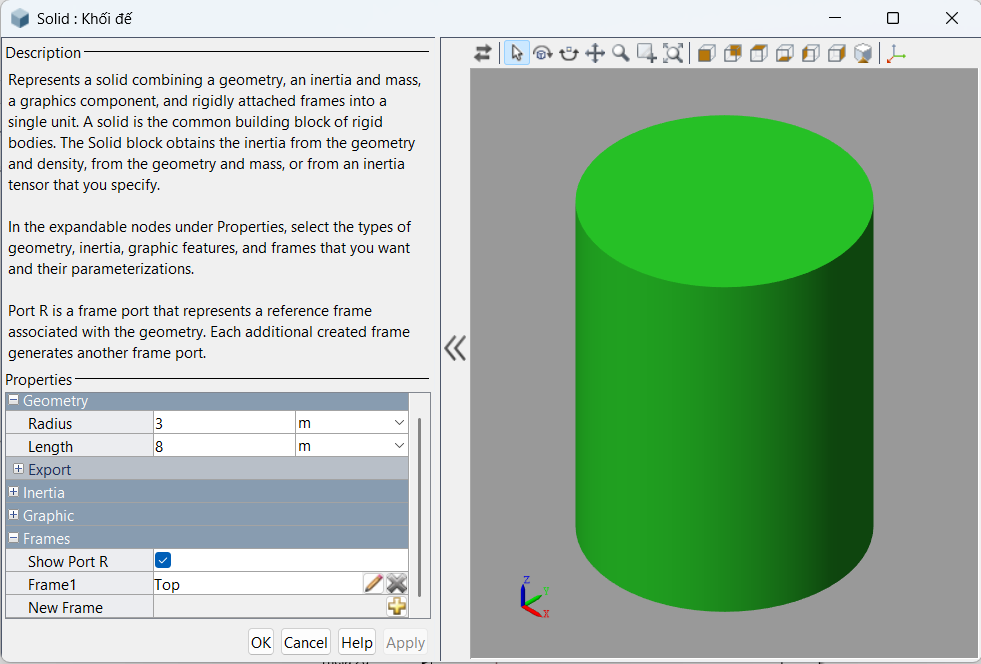
# Appendix A - Creating Multibody for Kinematic Analysis

The system in this appendix is available at: [Kinematic\_Group4- Google Drive](https://drive.google.com/drive/u/0/folders/1vkbNLk7sZz-gEg9qe8Ytnm3x4jouOwCD)

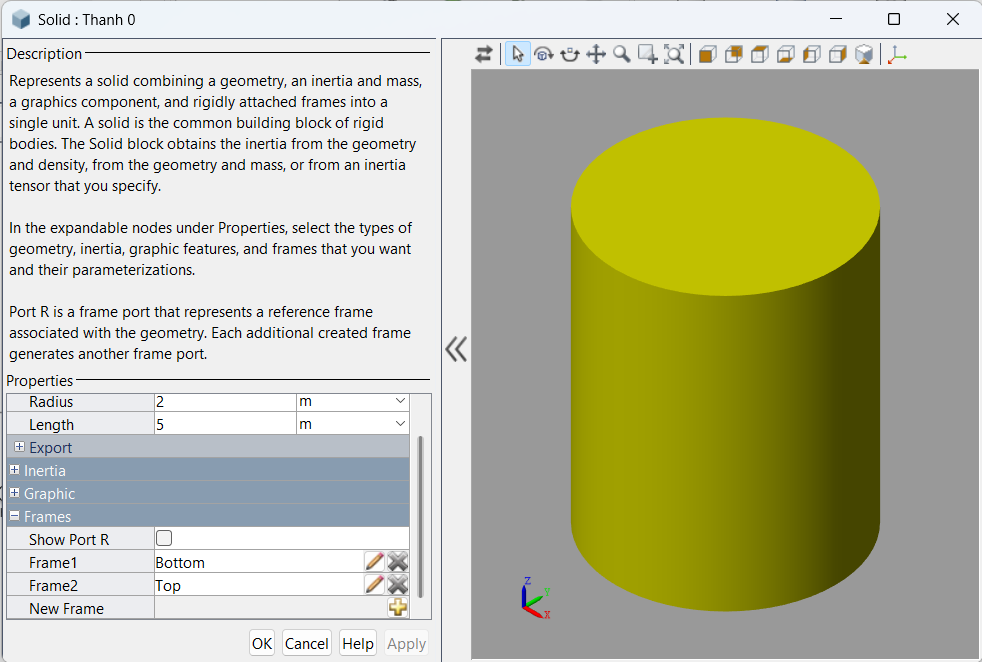
The steps below are implemented in Matlab version 2021b using Simscape Multibody.

**Step 1:** Construction of the Simscape Multibody Mechanical Model

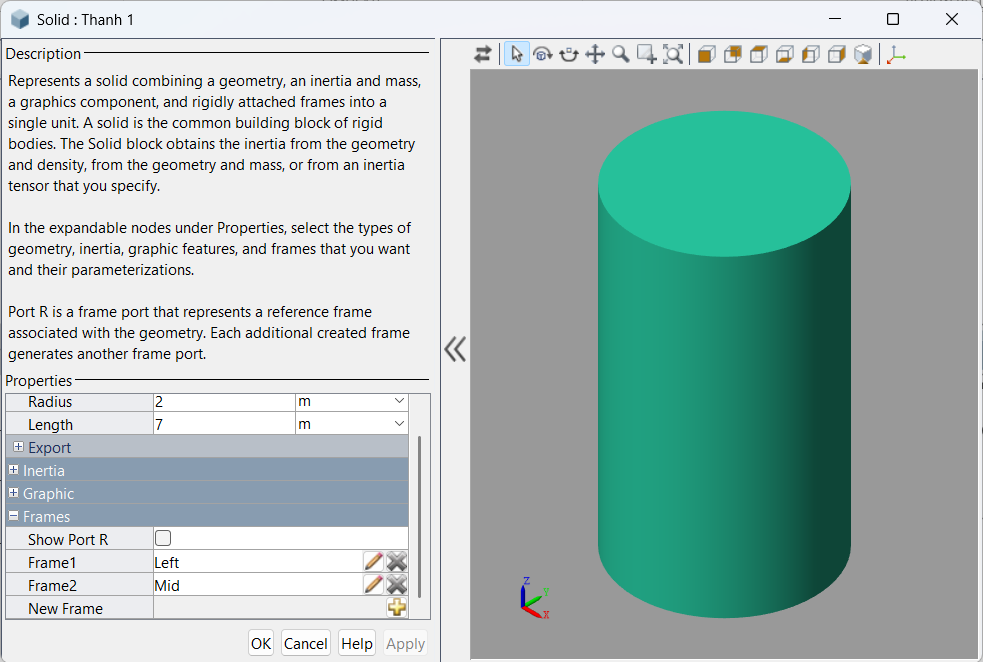
1. Base Creation: Create a base block with user-defined parameters to establish the fixed frame.



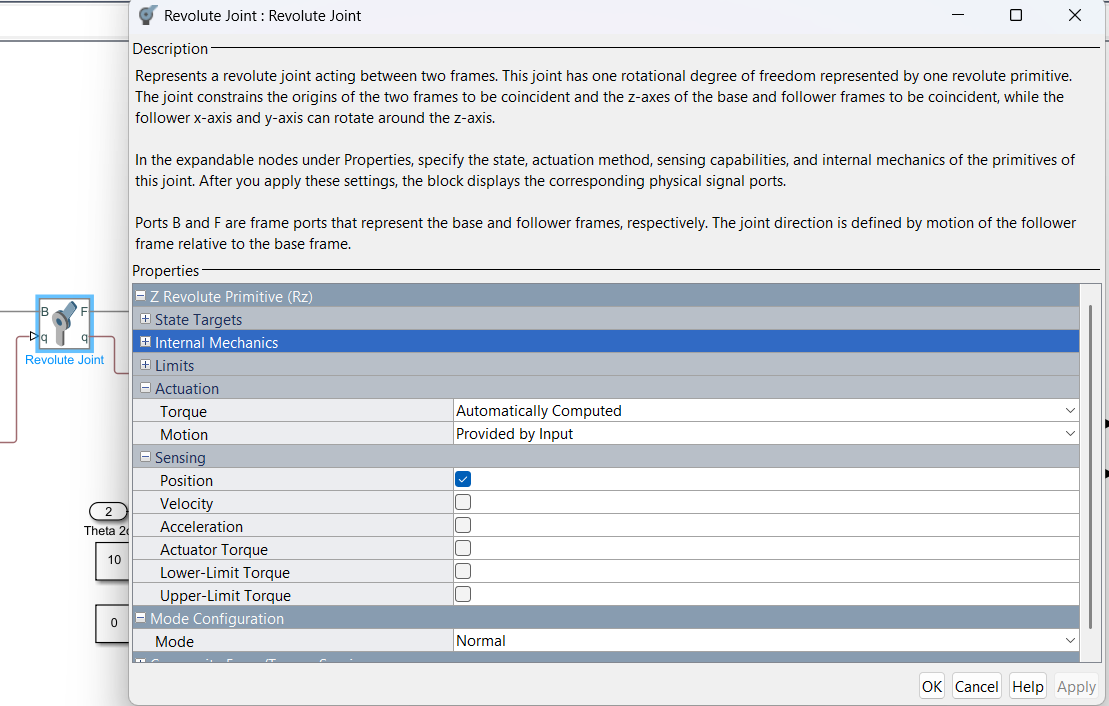
1. Link 0:



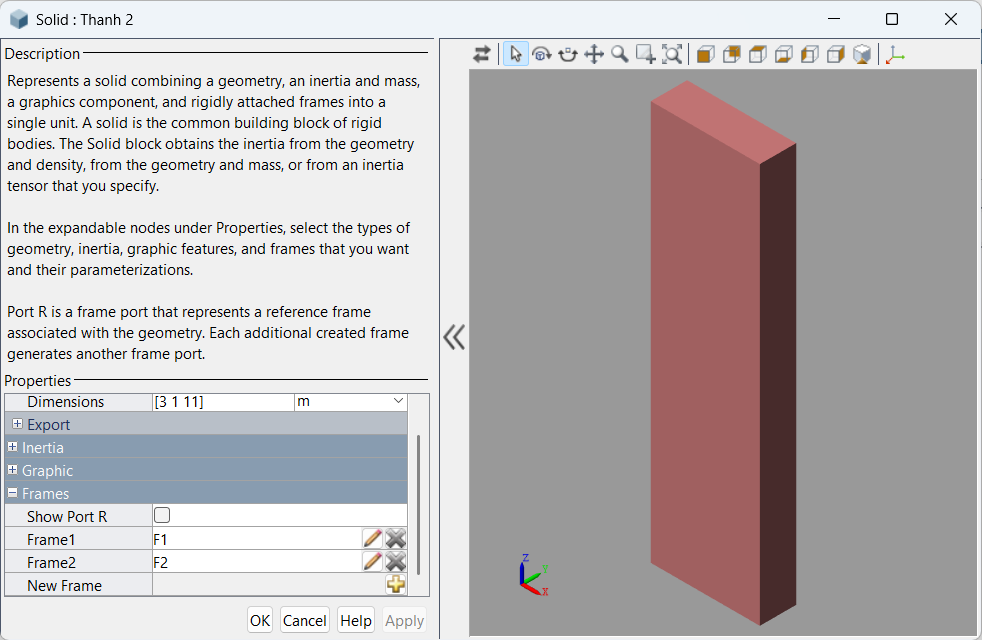
1. Link 1: Create the first arm link (Link 1).



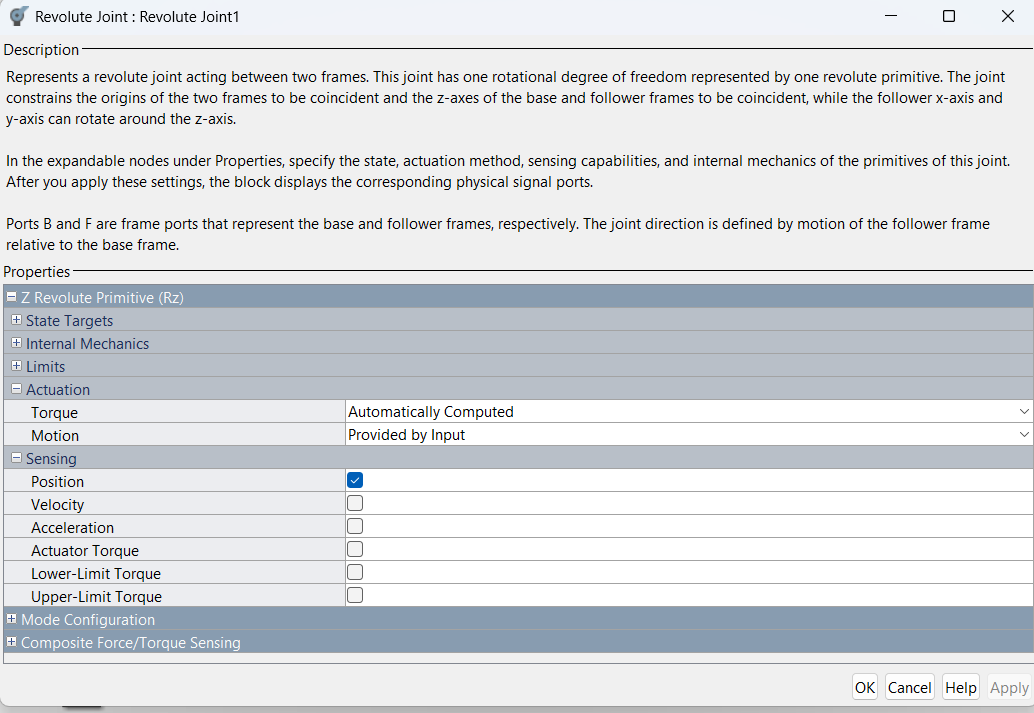
1. Joint 1: Create the first motion joint (Revolute Joint) rotating around the Z-axis Oz, connecting Link 1 and Connector 1.



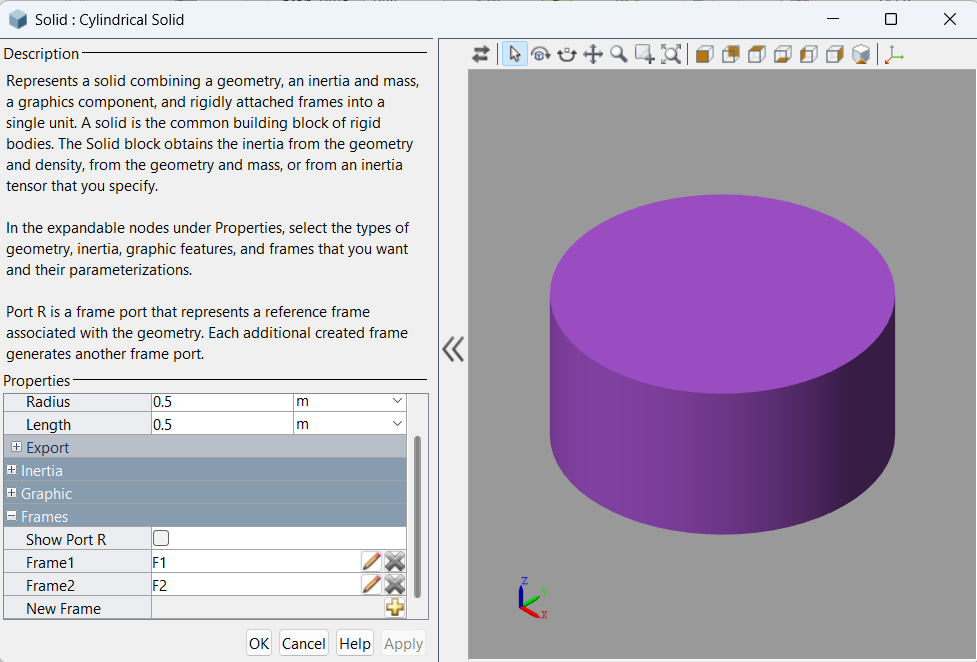
1. Link 2: Create the second arm link (Link 2).



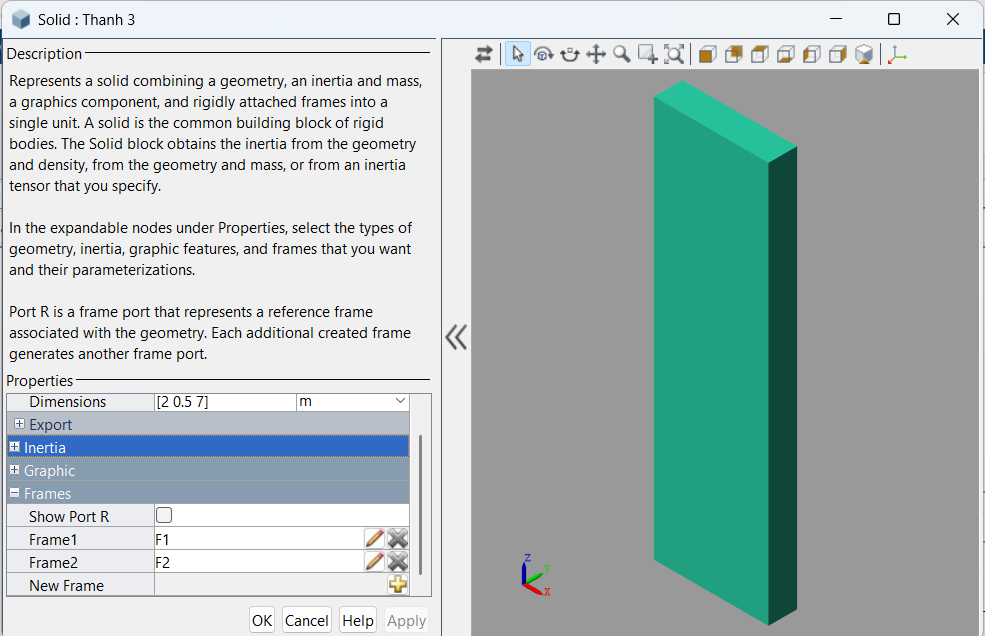
1. Joint 2: Create the second motion joint rotating around the Z-axis, connecting Link 2 and Link 1.



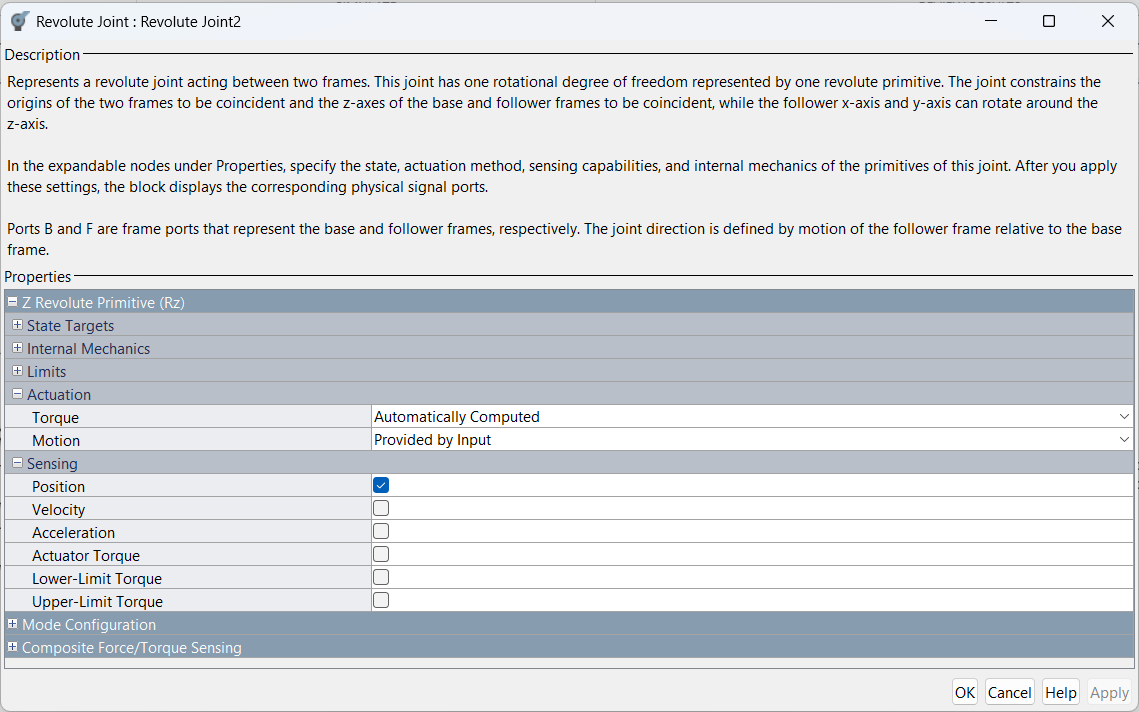
1. Connector 2: Create the second connecting block.



1. Link 3: Create the third arm link (Link 3).



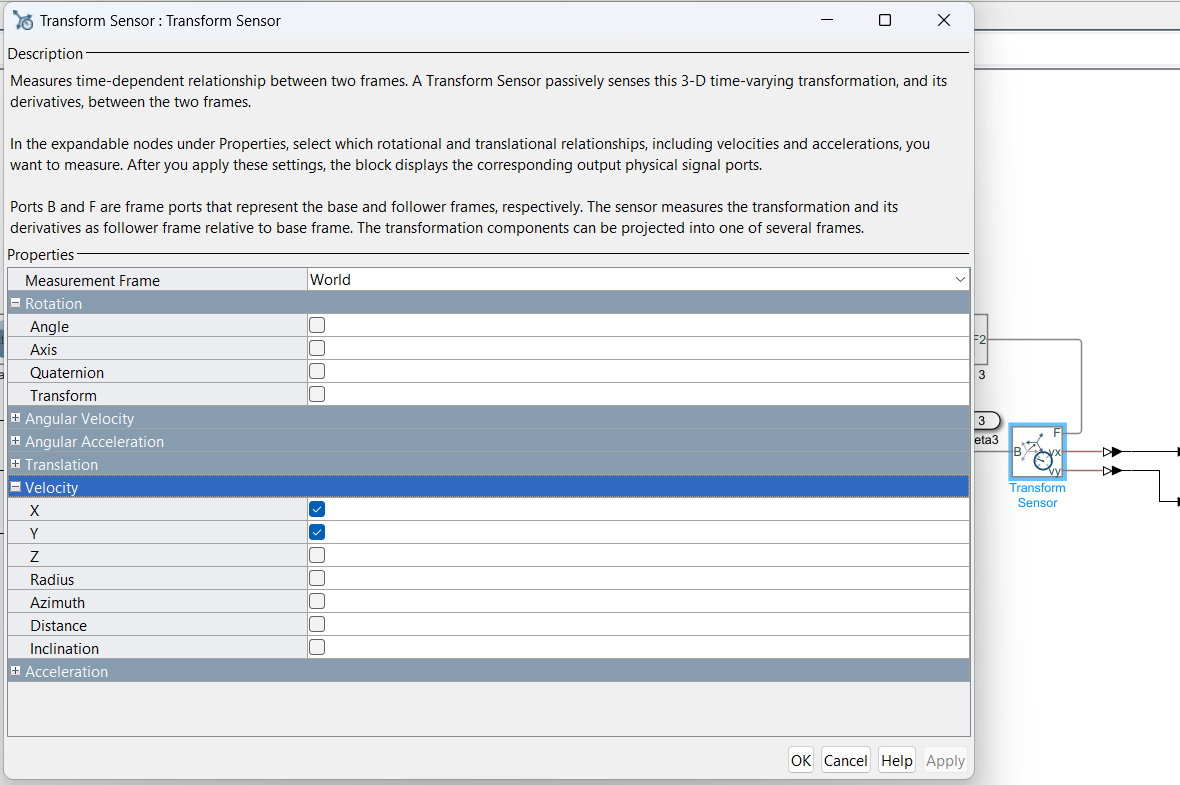
1. Joint 3: Create the third motion joint rotating around the Z-axis, connecting Link 3 and Connector 2.



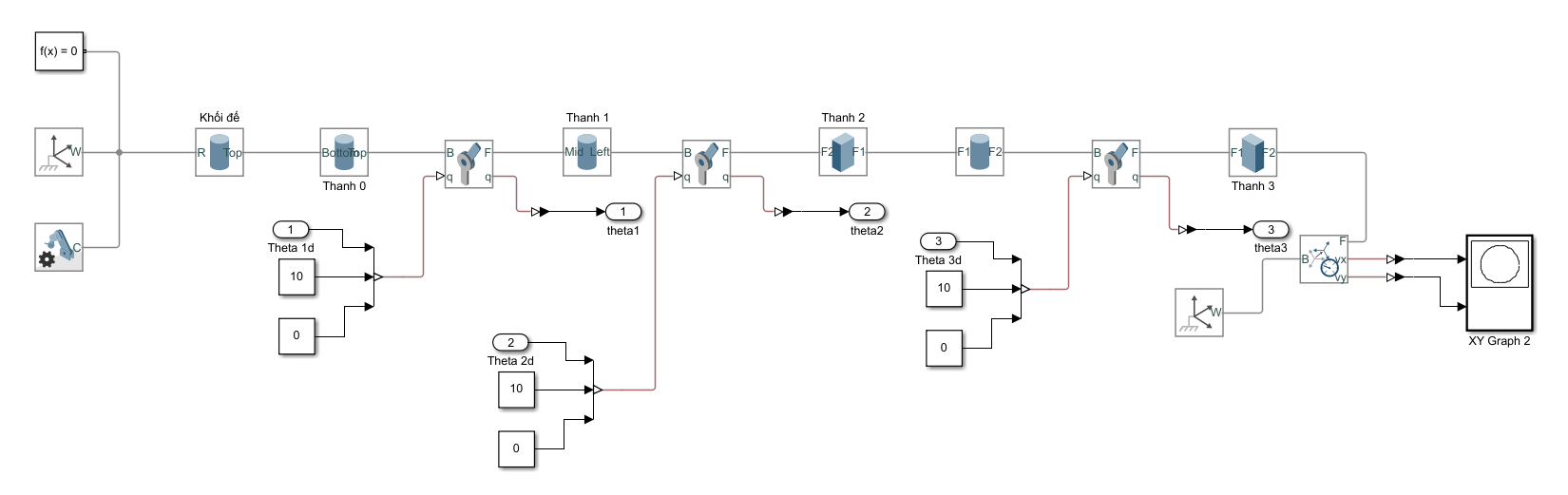
1. Input Configuration: Configure the necessary input ports for Position, Velocity, and Acceleration to drive the motion joints.



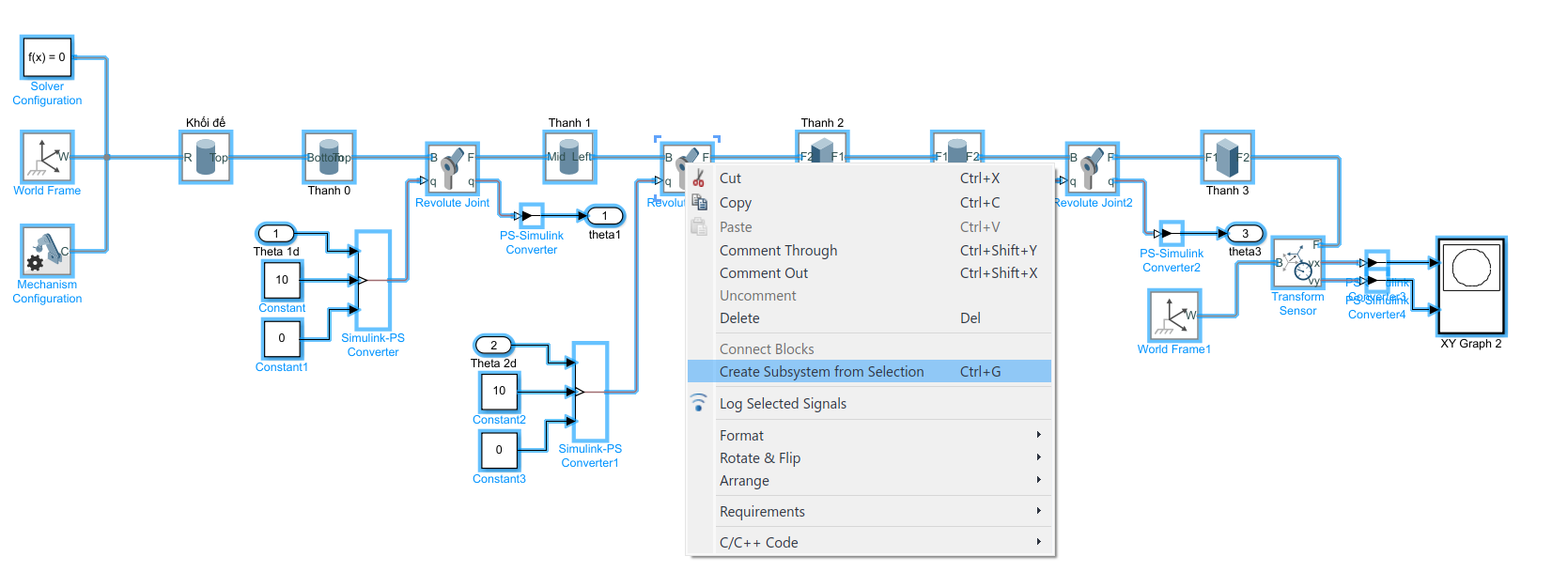
1. Sensing: Add a Transform Sensor block to measure the end-effector's position and visualize the trajectory using an XY Graph.

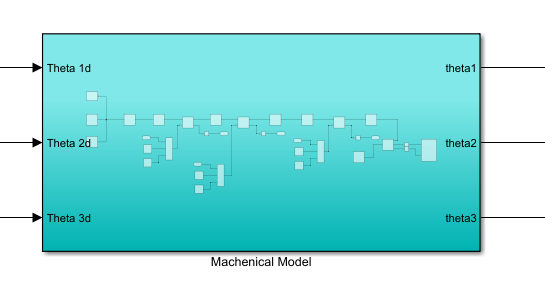


1. Connection: Finalize and verify all wiring connections within the block diagram.

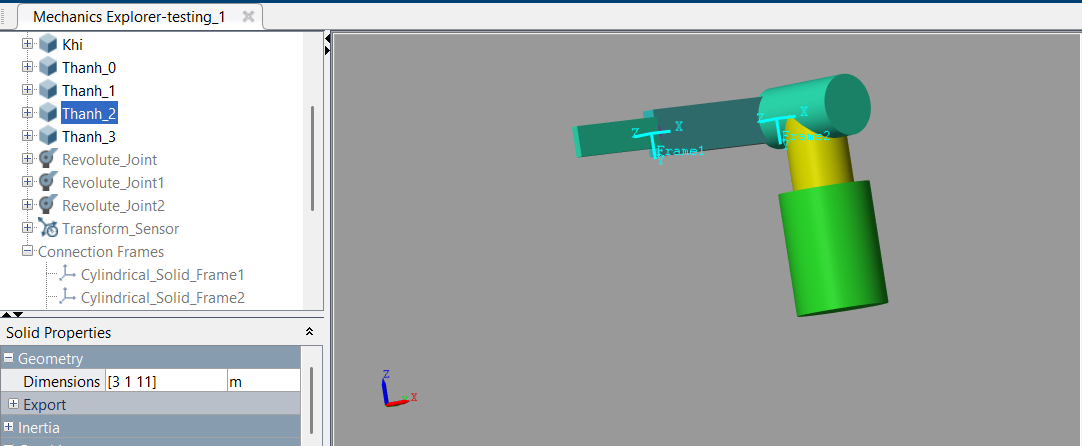


1. Subsystem: Encapsulate the entire mechanical model into a Subsystem for modularity.

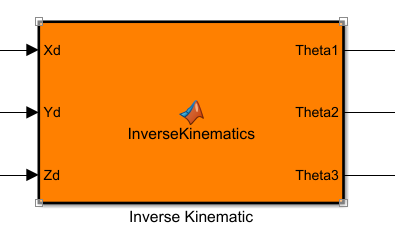




1. Mechanical Test: Perform an initial test run of the mechanical assembly to ensure proper constraints.



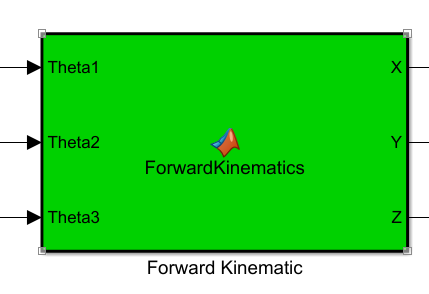
**Step 2**: Implementation of the Inverse Kinematics Block



Code:

| function [Theta1, Theta2, Theta3] = InverseKinematics(Xd, Yd, Zd)  % Chiều dài link  L1 = 7;  L2 = 11;  L3 = 7;  L0 = 13;  % --- Bước 1: Tính Theta2 và Theta3 (ràng buộc Theta3 = -Theta2)  sinTheta2 = -Yd / L2;  % Chặn giá trị trong [-1, 1] để tránh lỗi do sai số  sinTheta2 = min(max(sinTheta2, -1), 1);  Theta2 = asin(sinTheta2);  Theta3 = -Theta2; % ràng buộc  % --- Bước 2: Tính hệ số A, B, N, F, H để tìm Theta1  A = L2\*cos(Theta2) + L3; % do cos(Theta2+Theta3) = 1  B = L1;  N = B / A;  d = Zd - L0; % gọn kí hiệu  F = (-d - Xd\*N);  H = (Xd - d\*N);  % --- Bước 3: Tính Theta1  Theta1 = atan2(F, H);  end |
| --- |

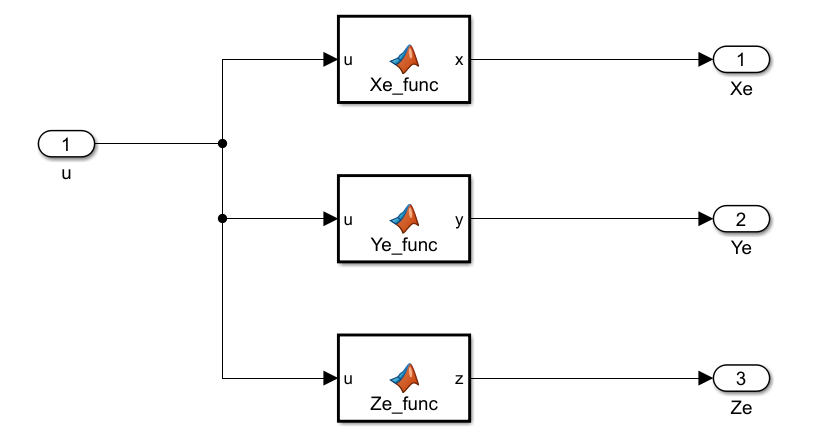
**Step 3:** Implementation of the Forward Kinematics Block



| %% -------- Forward Kinematics --------  function [X,Y,Z] = ForwardKinematics(Theta1,Theta2,Theta3)  % Tham số đã cho  L0 = 13; L1 = 7; L2 = 11; L3 = 7;  T10 = [cos(Theta1), -sin(Theta1), 0, 0;  sin(Theta1), cos(Theta1), 0, 0;  0, 0, 1, 0;  0, 0, 0, 1];  T21 = [cos(Theta2), -sin(Theta2), 0, 0;  0, 0, 1, L1;  -sin(Theta2), -cos(Theta2),0, 0;  0, 0, 0, 1];  T32 = [cos(Theta3), -sin(Theta3), 0, L2;  sin(Theta3), cos(Theta3), 0, 0;  0, 0, 1, 0;  0, 0, 0, 1];  Te3 = [1, 0, 0, L3;  0, 1, 0, 0;  0, 0, 1, 0;  0, 0, 0, 1];  T0W = [1,0,0,0;  0,0,1,0;  0,-1,0,L0;  0,0,0,1];  T20 = T10\*T21;  T30 = T20\*T32;  Te0 = T30\*Te3;  TeW = T0W\*Te0;  X = TeW(1,4);  Y = TeW(2,4);  Z = TeW(3,4);  end |
| --- |

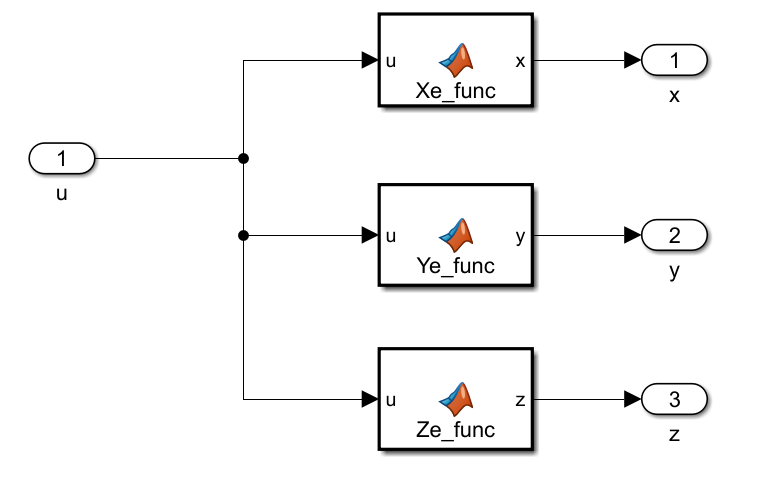
**Step 4**: Trajectory Generation for Simulation

1. Circular Trajectory: Define the equations for a circular path with a radius of R = 0.05m and a center point at (X\_e, Y\_e) = (0.05, 0.05).



| function x = Xe\_func(u)  % Xe\_func: Tính tọa độ X của quỹ đạo đường tròn  % Input:  % u - thời gian (hoặc biến độc lập từ Clock)  % Output:  % y - giá trị Xe  R = 3; % bán kính  x = R \* sin((2\*pi/5)\*u + pi/2);  end  function y = Ye\_func(u)  % Ye\_func: Tính tọa độ Y của quỹ đạo đường tròn  % Input:  % u - thời gian (hoặc biến độc lập từ Clock)  % Output:  % y - giá trị Ye  R = 3; % bán kính  y = R \* cos((2\*pi/5)\*u + pi/2);  end  function z = Ze\_func(u)  % Ze\_func: Tính tọa độ Z của quỹ đạo đường tròn  z = 13 - sqrt((11 + 7)^2 + 3.5^2); % ≈ -5.3371    end |
| --- |

1. Linear Trajectory:  
   Define the equations for a straight-line path constrained by two endpoints located at $(X, Y) = (0.95, 1)$ and $(1.05, 1)$



| function x = Xe\_func(u)  % Xe\_func: Quy dao duong thang tu x = -2 den x = 2  % Su dung quy dao Cosine de van toc bat dau va ket thuc bang 0  % Thoi gian du kien de di tu -2 den 2    RunTime = 10;    Start\_Point = -2;  End\_Point = 2;    if u <= RunTime  % u = 0 -> cos(0)=1 -> x = Start  % u = RunTime -> cos(pi)=-1 -> x = End    Center = (Start\_Point + End\_Point) / 2;  Amplitude = (Start\_Point - End\_Point) / 2;    x = Center + Amplitude \* cos(pi \* u / RunTime);  else  % Khi het thoi gian, giu nguyen vi tri tai dich  x = End\_Point;  end    end  function y = Ye\_func(u)  y = 0;  end  function z = Ze\_func(u)  %z = ;  %z = -14;  z = -6;  end |
| --- |

**Step 5**: Result system. → Click Run.



The simulation results and verification available in Chapter 2 (Section 2.7)

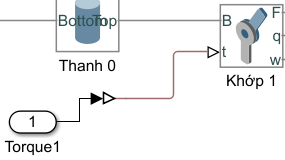
# Appendix B - Creating Multibody for Dynamic Analysis

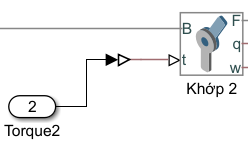
The system in this appendix is available at: [Dynamic\_Group4- Google Drive](https://drive.google.com/drive/folders/1vkbNLk7sZz-gEg9qe8Ytnm3x4jouOwCD?usp=drive_link)

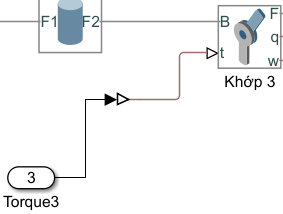
The steps below are implemented in Matlab version 2021b using Simscape Multibody.

**Step 1:** Construction of the Simscape Multibody Mechanical Model

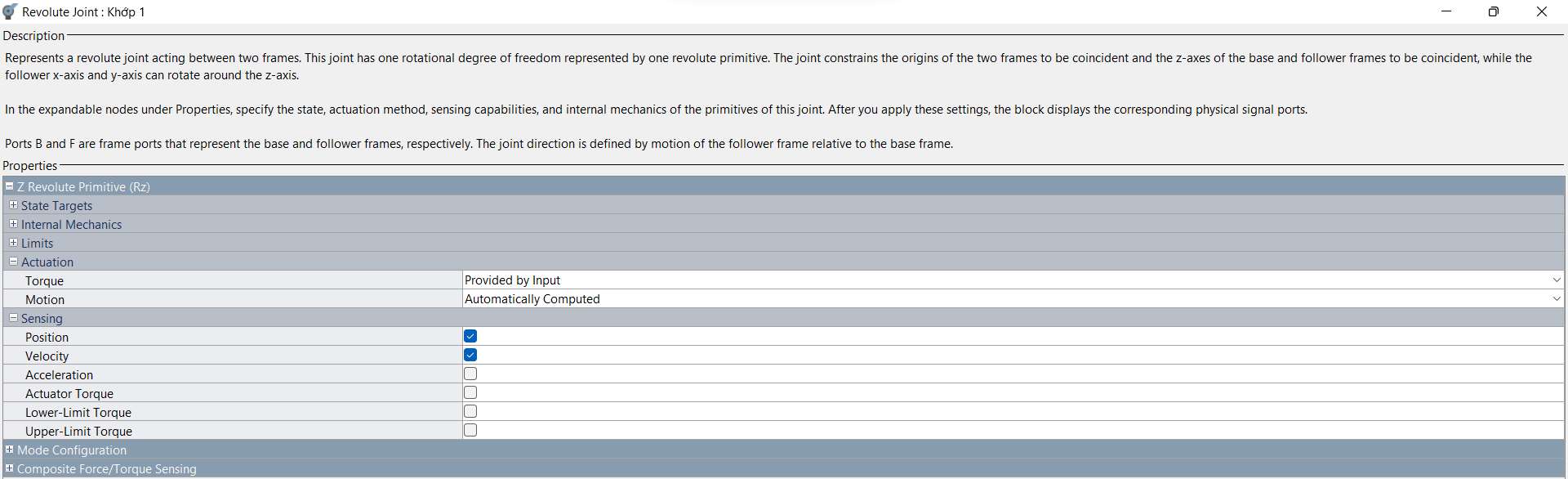
1. Remove the 3 constant values from the last model at all 3 joints by the input In1 and name them Torque1, Torque2, and Torque3.



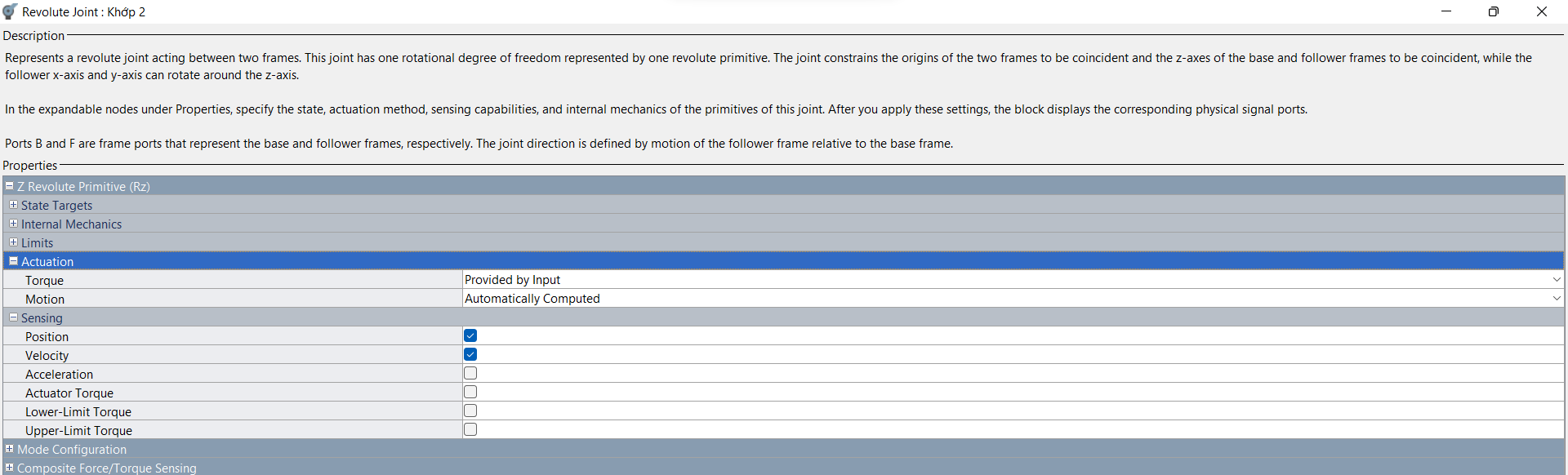




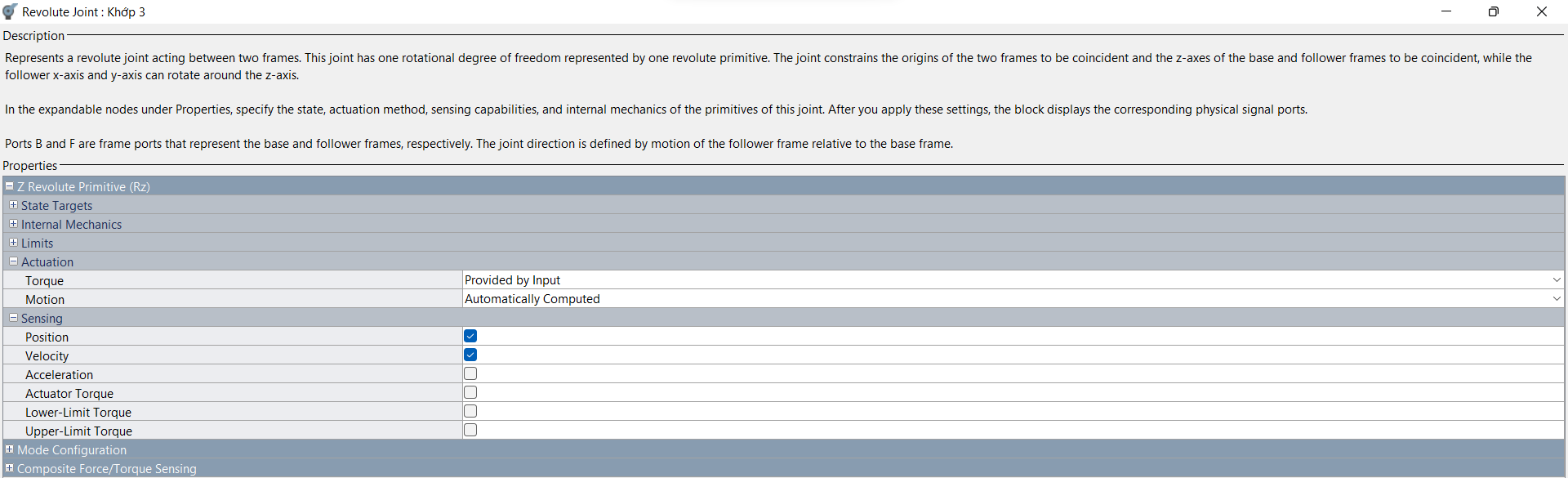
1. Joint 1: Turn on the Sensing Position and Velocity in Joint 1. Torque will be provided by input, while Motion will be automatically computed.



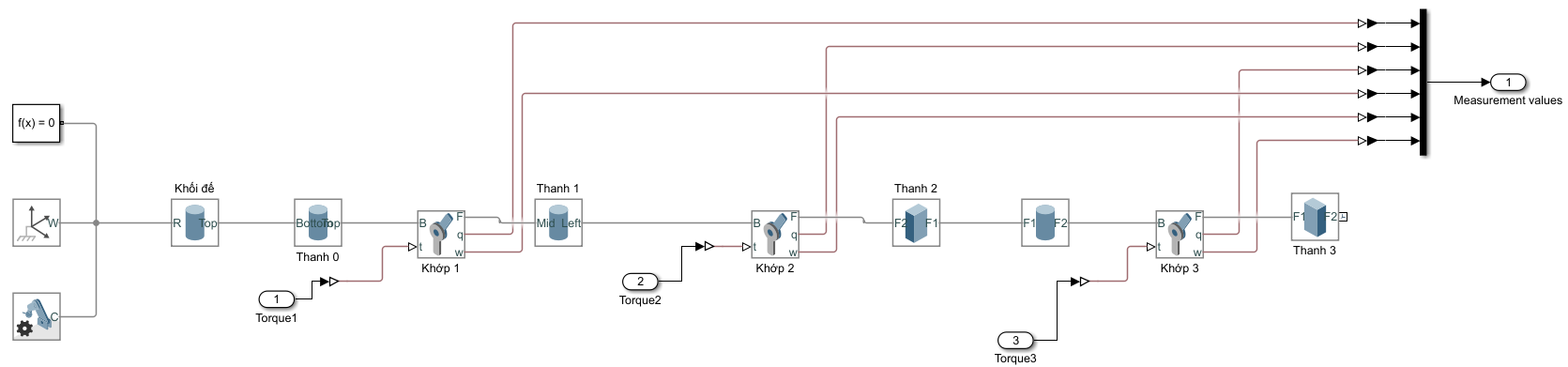
1. Joint 2: Turn on the Sensing Position and Velocity in Joint 2. Torque will be provided by input, while Motion will be automatically computed.



1. Joint 3: Turn on the Sensing Position and Velocity in Joint 3. Torque will be provided by input, while Motion will be automatically computed.

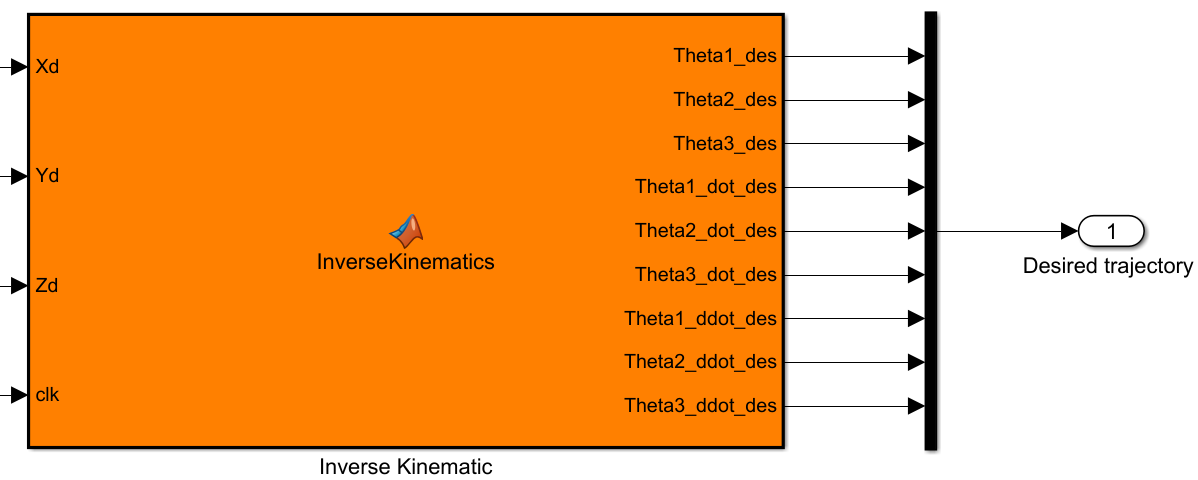


1. Connect all of the actual positions and velocities of the 3 joints and connect them to a Mux, name the output Measurement values.



**Step 2**: Implementation of the Inverse Kinematics Block

The output of this block is connected to a mux and named Desired trajectory for later use in the Torque Computed block.



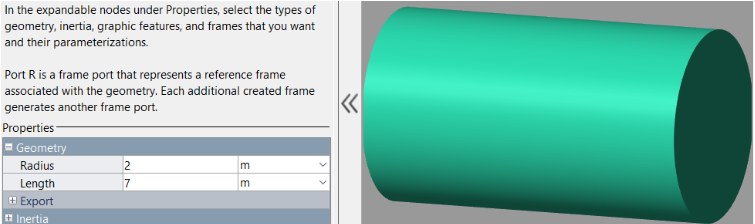
Code:

| function [Theta1\_des, Theta2\_des, Theta3\_des, ...  Theta1\_dot\_des, Theta2\_dot\_des, Theta3\_dot\_des, ...  Theta1\_ddot\_des, Theta2\_ddot\_des, Theta3\_ddot\_des] = InverseKinematics(Xd, Yd, Zd, Select\_Mode, clk)  % ============================================================  % UNIFIED INVERSE KINEMATICS (Straight Line & Circle)  % Input Select\_Mode:  % 1: Straight Line Trajectory  % 2: Circular Trajectory  % ============================================================  % --- 1. THONG SO HINH HOC (CHUNG CHO CA 2) ---  L1 = 7;  L2 = 11;  L3 = 7;  L0 = 13;  % --- 2. TINH TOAN DAO HAM (QUY HOACH QUY DAO) ---  % Khoi tao gia tri mac dinh  Xd\_dot = 0; Xd\_ddot = 0;  Yd\_dot = 0; Yd\_ddot = 0;  Zd\_dot = 0; Zd\_ddot = 0;  if Select\_Mode == 1  %% --- CHE DO 1: DUONG THANG (Straight Line) ---  Start\_Point = -2;  End\_Point = 2;  A = (End\_Point - Start\_Point) / 2; % Bien do = 2    TravelTime = 30; % Thoi gian di chuyen  w = pi / TravelTime; % Tan so goc  % Truc X (Cosine Trajectory)  Xd\_dot = A \* w \* sin(w \* clk);  Xd\_ddot = A \* w^2 \* cos(w \* clk);    % Truc Y va Z (Giữ nguyen = 0 va const)  Yd\_dot = 0; Yd\_ddot = 0;  Zd\_dot = 0; Zd\_ddot = 0;  else  %% --- CHE DO 2: HINH TRON (Circle) ---  T = 5; % Chu ky  w = 2\*pi/T; % Tan so goc  R = 2; % Ban kinh quy dao  % Dao ham cap 1  Xd\_dot = R \* w \* cos(w\*clk + pi/2);  Yd\_dot = -R \* w \* sin(w\*clk + pi/2);  Zd\_dot = 0;  % Dao ham cap 2  Xd\_ddot = -R \* w^2 \* sin(w\*clk + pi/2);  Yd\_ddot = -R \* w^2 \* cos(w\*clk + pi/2);  Zd\_ddot = 0;  end  % ============================================================  % --- 3. GIAI BAI TOAN DONG HOC NGUOC (CORE SOLVER) ---  % ============================================================  % --- Buoc 3.1: Tinh Theta2, Theta3 ---  sinTheta2 = -Yd / L2;    % Rang buoc gia tri trong khoang [-1, 1]  sinTheta2 = min(max(sinTheta2, -1), 1);    Theta2\_des = asin(sinTheta2);  Theta3\_des = -Theta2\_des; % Rang buoc Elbow-up/Parallel    cosTheta2 = cos(Theta2\_des); % Tinh cos(Theta2)  % --- Dao ham Theta2, Theta3 ---  % Check Singularity (tranh chia cho 0)  if abs(cosTheta2) < 1e-6  Theta2\_dot\_des = 0;  Theta2\_ddot\_des = 0;  else  Theta2\_dot\_des = -Yd\_dot / (L2 \* cosTheta2);  Theta2\_ddot\_des = (-Yd\_ddot / (L2 \* cosTheta2)) - (Theta2\_dot\_des^2 \* tan(Theta2\_des));  end  Theta3\_dot\_des = -Theta2\_dot\_des;  Theta3\_ddot\_des = -Theta2\_ddot\_des;  % --- Buoc 3.2: Tinh Theta1 ---  D = L2\*cosTheta2 + L3;  d = Zd - L0;    % Cong thuc hinh hoc  F = (-d - Xd\*(L1/D));  H = (Xd - d\*(L1/D));  Theta1\_des = atan2(F, H);  % --- Dao ham F, H ---  D\_dot = -L2\*sin(Theta2\_des)\*Theta2\_dot\_des;  D\_ddot = -L2\*(cos(Theta2\_des)\*Theta2\_dot\_des^2 + sin(Theta2\_des)\*Theta2\_ddot\_des);  % Bao ve mau so D  if abs(D) < 1e-6; D = 1e-6; end  F\_dot = -(Zd\_dot) - Xd\_dot\*(L1/D) + Xd\*(L1\*D\_dot)/(D^2);  H\_dot = Xd\_dot - Zd\_dot\*(L1/D) + d\*(L1\*D\_dot)/(D^2);  F\_ddot = -(Zd\_ddot) - Xd\_ddot\*(L1/D) ...  + 2\*Xd\_dot\*(L1\*D\_dot)/(D^2) ...  - Xd\*(L1\*D\_ddot)/(D^2) + 2\*Xd\*(L1\*D\_dot^2)/(D^3);  H\_ddot = Xd\_ddot - Zd\_ddot\*(L1/D) ...  + 2\*Zd\_dot\*(L1\*D\_dot)/(D^2) ...  - d\*(L1\*D\_ddot)/(D^2) + 2\*d\*(L1\*D\_dot^2)/(D^3);  % --- Dao ham Theta1 ---  denominator = F^2 + H^2;    % Check Singularity cho mau so cua Theta1  if denominator < 1e-6  Theta1\_dot\_des = 0;  Theta1\_ddot\_des = 0;  else  Theta1\_dot\_des = (H\*F\_dot - F\*H\_dot) / denominator;  Theta1\_ddot\_des = ((H\*F\_ddot - F\*H\_ddot)\*denominator ...  - 2\*(H\*F\_dot - F\*H\_dot)\*(F\*F\_dot + H\*H\_dot)) ...  / (denominator^2);  end  end |
| --- |

**Step 3: Calculate Momentum of Inertia**

Before getting started, we are choosing the material for the robot to be plastic, which has a density (p) of 1000 .

1. Link 1: Radius 2m, Length 7m



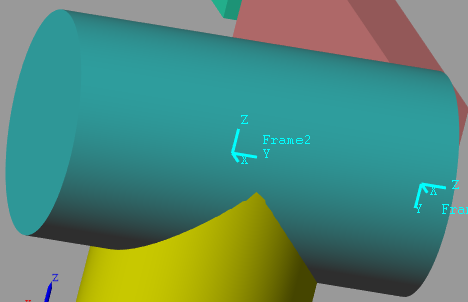
We have:

* Volume = 87,965 ()

Mass 79168.5 (kg)

Since the z-coordination of link 1 is pointing up, and the direction of the link is horizontal, we apply the formula below to calculate .

* = 402437.875



1. Link 2: Length 11m, Width 1m, Height 3m

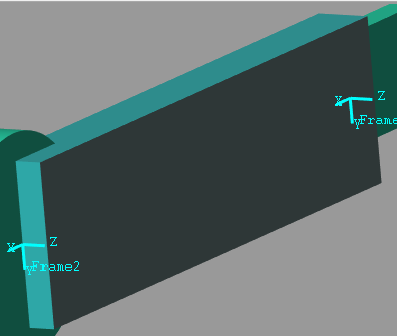
We have:

* Volume = 33 ()

Mass 29700 (kg)

Since the z-coordination of link 2 is pointing outside, the joint is at the middle beginning of the link, and the direction of the link is horizontal, we apply the formula below to calculate .

* = 1197900



1. Link 3: Length 7m, Width 2m, Height 0.5m

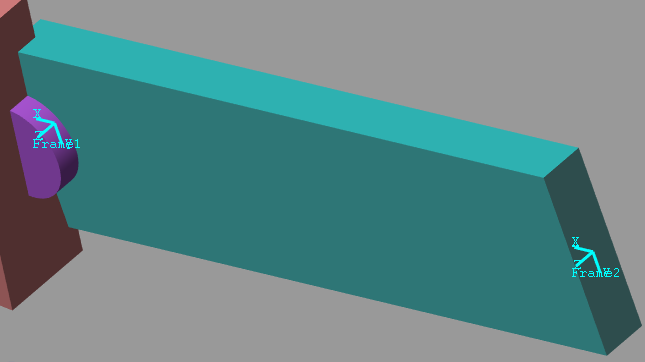
We have:

* Volume = 7 ()

Mass 6300 (kg)

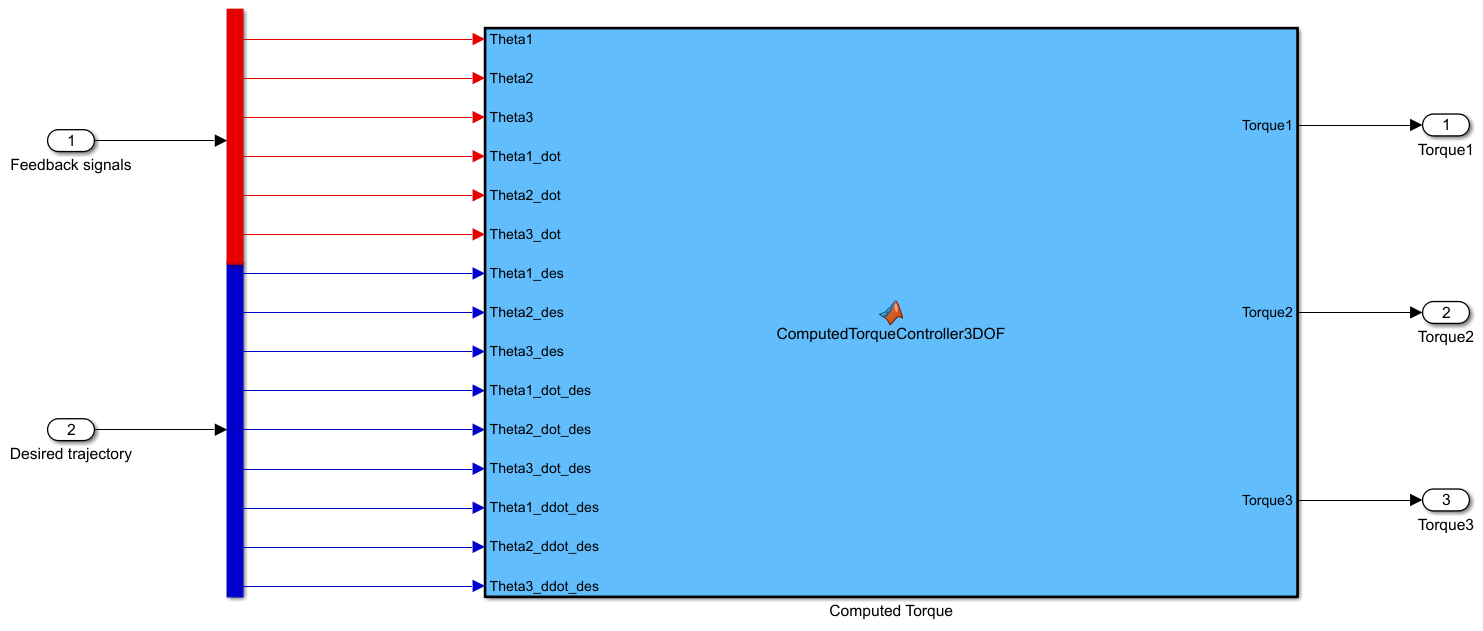
Since the z-coordination of link 3 is pointing outside, the joint is at the middle beginning of the link, and the direction of the link is horizontal, we apply the formula below to calculate .

* = 102900



**Step 4:** Implementation of the Computed Torque Controller block

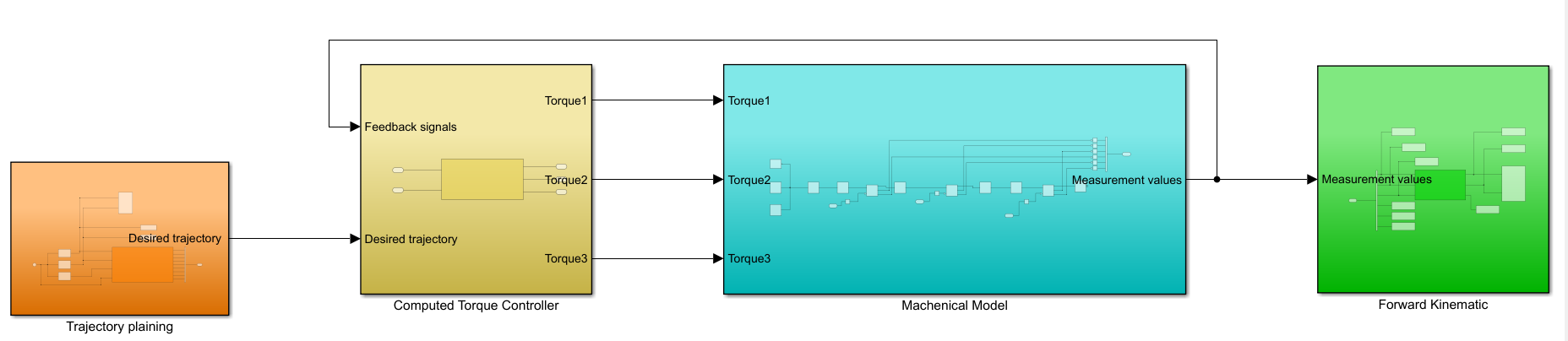
1. This block receives input from the Feedback signal (Mechanical model block) and the Desired trajectory (Inverse Kinematic block).
2. Add Demux to extract the value inside 2 inputs.



Code:

| function [Torque1, Torque2, Torque3] = ComputedTorqueController3DOF( ...  Theta1, Theta2, Theta3, ...  Theta1\_dot, Theta2\_dot, Theta3\_dot, ...  Theta1\_des, Theta2\_des, Theta3\_des, ...  Theta1\_dot\_des, Theta2\_dot\_des, Theta3\_dot\_des, ...  Theta1\_ddot\_des, Theta2\_ddot\_des, Theta3\_ddot\_des)  %% ---------------- Parameters ----------------  L1 = 7;  L2 = 11;  L3 = 7;  m1 = 79168.13487;  m2 = 29700;  m3 = 6300;  I1 = 402437.875;  I2 = 1197900;  I3 = 102900;  lc1 = L1 / 2;  lc2 = L2 / 2;  lc3 = L3 / 2;  g = 9.81;  %% ---------------- Desired and actual vectors ----------------  Theta = [Theta1; Theta2; Theta3];  Theta\_dot = [Theta1\_dot; Theta2\_dot; Theta3\_dot];  Theta\_des = [Theta1\_des; Theta2\_des; Theta3\_des];  Theta\_dot\_des = [Theta1\_dot\_des; Theta2\_dot\_des; Theta3\_dot\_des];  Theta\_ddot\_des = [Theta1\_ddot\_des; Theta2\_ddot\_des; Theta3\_ddot\_des];  %% ---------------- Error terms ----------------  E = Theta\_des - Theta;  E\_dot = Theta\_dot\_des - Theta\_dot;  %% ---------------- Dynamic model ----------------  % --- Inertia matrix M(Theta)  M11 = I1 + I2 + I3 ...  + m1\*lc1^2 ...  + m2\*(L1^2 + lc2^2 + 2\*L1\*lc2\*cos(Theta2)) ...  + m3\*(L1^2 + L2^2 + lc3^2 ...  + 2\*L1\*L2\*cos(Theta2) ...  + 2\*L1\*lc3\*cos(Theta2 + Theta3) ...  + 2\*L2\*lc3\*cos(Theta3));  M12 = I2 + I3 ...  + m2\*(lc2^2 + L1\*lc2\*cos(Theta2)) ...  + m3\*(L2^2 + lc3^2 + L1\*L2\*cos(Theta2) ...  + L1\*lc3\*cos(Theta2 + Theta3) ...  + L2\*lc3\*cos(Theta3));  M13 = I3 + m3\*(lc3^2 + L1\*lc3\*cos(Theta2 + Theta3) + L2\*lc3\*cos(Theta3));  M22 = I2 + I3 ...  + m2\*lc2^2 ...  + m3\*(L2^2 + lc3^2 + 2\*L2\*lc3\*cos(Theta3));  M23 = I3 + m3\*(lc3^2 + L2\*lc3\*cos(Theta3));  M33 = I3 + m3\*lc3^2;  M = [M11, M12, M13;  M12, M22, M23;  M13, M23, M33];  % --- Coriolis/Centrifugal matrix C(Theta, Theta\_dot)  h1 = -m2\*L1\*lc2\*sin(Theta2) ...  - m3\*L1\*L2\*sin(Theta2) ...  - m3\*L1\*lc3\*sin(Theta2 + Theta3);  h2 = -m3\*L2\*lc3\*sin(Theta3);  C = [ h1\*(Theta2\_dot + Theta3\_dot), h1\*(Theta2\_dot + Theta3\_dot), h1\*Theta3\_dot;  -h1\*Theta1\_dot, h2\*Theta3\_dot, h2\*(Theta2\_dot + Theta3\_dot);  -h1\*Theta1\_dot, -h2\*(Theta1\_dot + Theta2\_dot), 0];  % --- Gravity vector G(Theta)  G1 = (m1\*lc1 + m2\*L1 + m3\*L1)\*g\*cos(Theta1) ...  + (m2\*lc2 + m3\*L2)\*g\*cos(Theta1 + Theta2) ...  + m3\*lc3\*g\*cos(Theta1 + Theta2 + Theta3);  G2 = (m2\*lc2 + m3\*L2)\*g\*cos(Theta1 + Theta2) ...  + m3\*lc3\*g\*cos(Theta1 + Theta2 + Theta3);  G3 = m3\*lc3\*g\*cos(Theta1 + Theta2 + Theta3);  G = [G1; G2; G3];  %% ---------------- Controller gains ----------------  %Kp = diag([1800, 5400, 100]); % bộ Kp, Kd với hình tròn  %Kd = diag([180, 540, 100]);  Kp = diag([3200, 9600, 800]); % thử nghiệm với đường thẳng  Kd = diag([320, 960, 80]);  %% ---------------- Computed Torque ----------------  Torque = M\*(Theta\_ddot\_des + Kd\*E\_dot + Kp\*E) + C\*Theta\_dot + G;  %% ---------------- Outputs ----------------  Torque1 = Torque(1);  Torque2 = Torque(2);  Torque3 = Torque(3);  end |
| --- |

**Step 5:** Run the model



# 

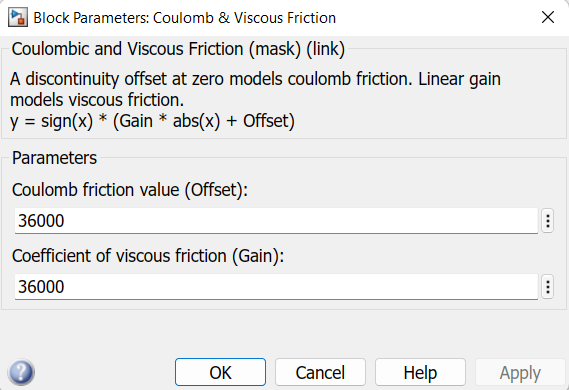
# Appendix C - Creating Multibody for Neural Network

The system in this appendix is available at: [Neural\_Network\_Group4- Google Drive](https://drive.google.com/drive/folders/1vkbNLk7sZz-gEg9qe8Ytnm3x4jouOwCD?usp=drive_link)

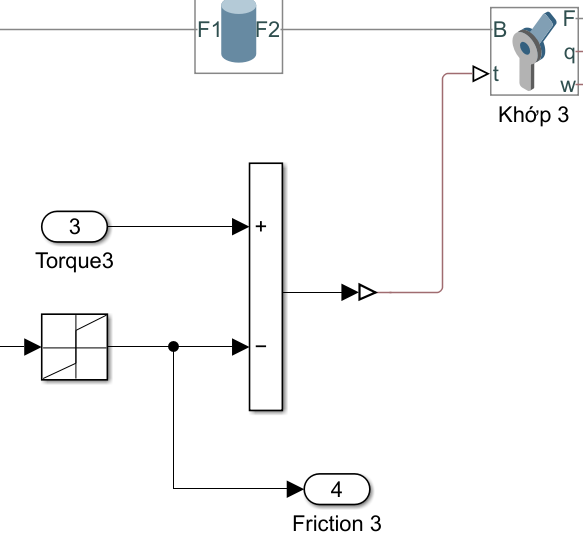
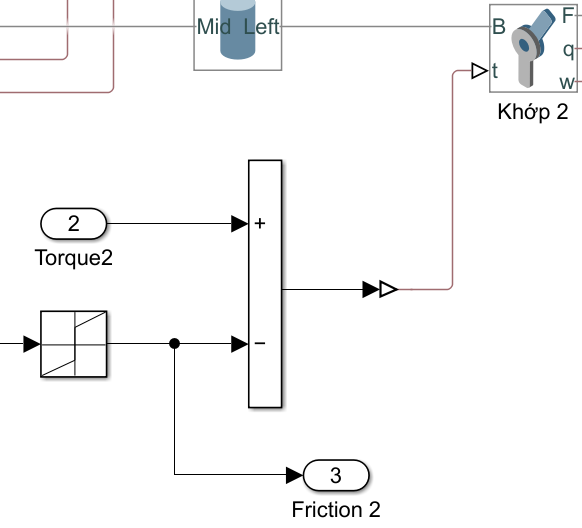
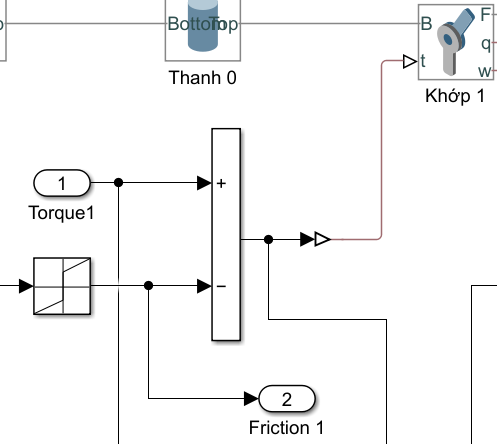
The steps below are implemented in Matlab version 2021b using Simscape Multibody.

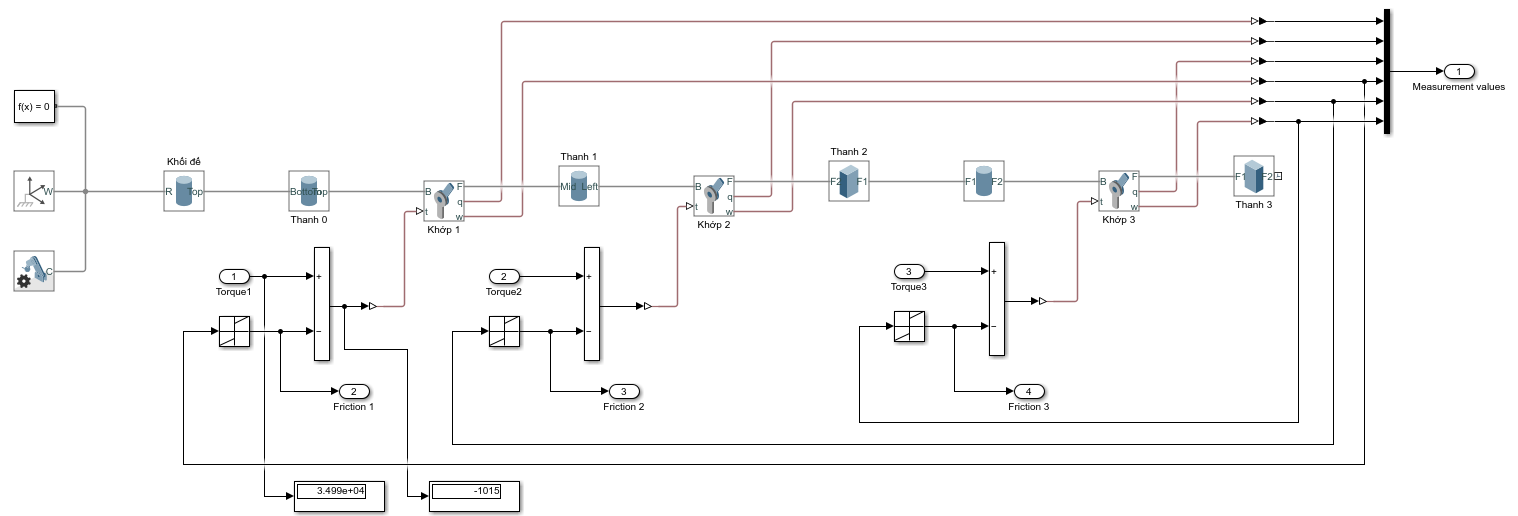
**Step 1:** Construction of the Simscape Multibody Mechanical Model

1. Set up the Coulomb and Vicious friction for Joints 1, 2, and 3

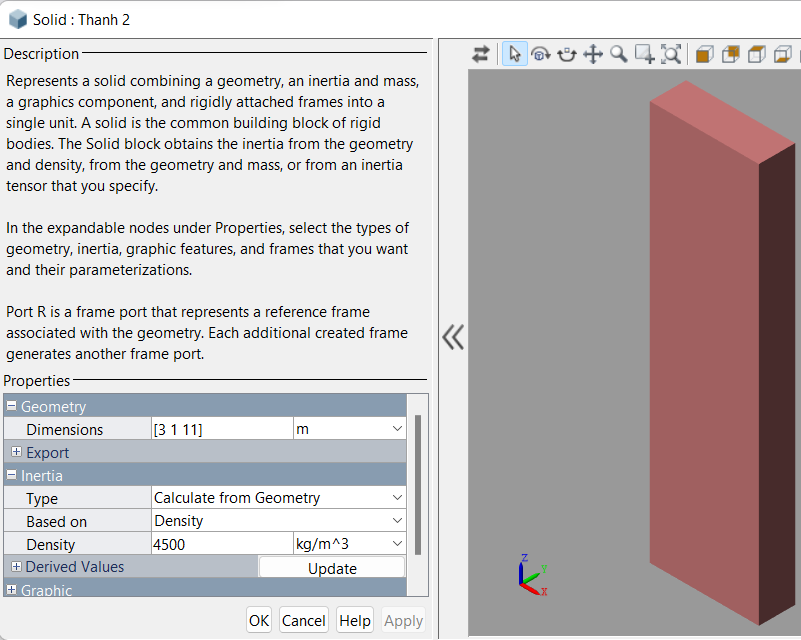


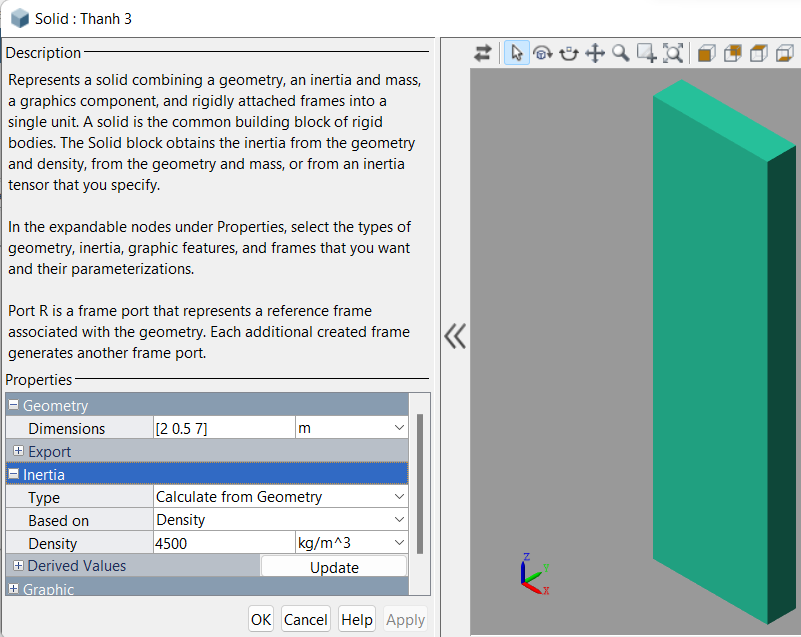
1. Match the input of the Friction block with the actual velocity of the joint accordingly, and the output connects to the Add block with a minus, while the input Torque is a plus.





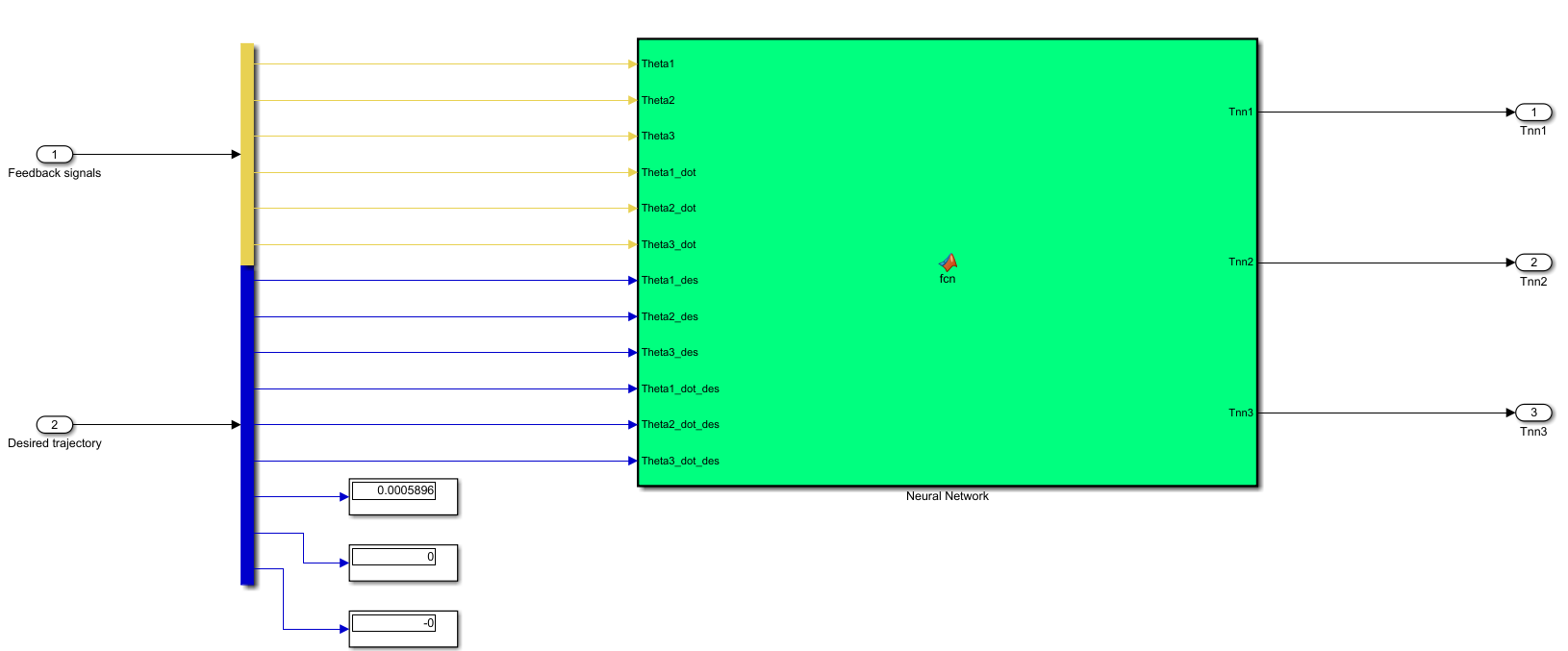
1. Increase the density (which also means increase weight) of the link 2 and link 3 5 times to make a strongly deviated orbit.





**Step 2:** Construction of the Neural Network block:

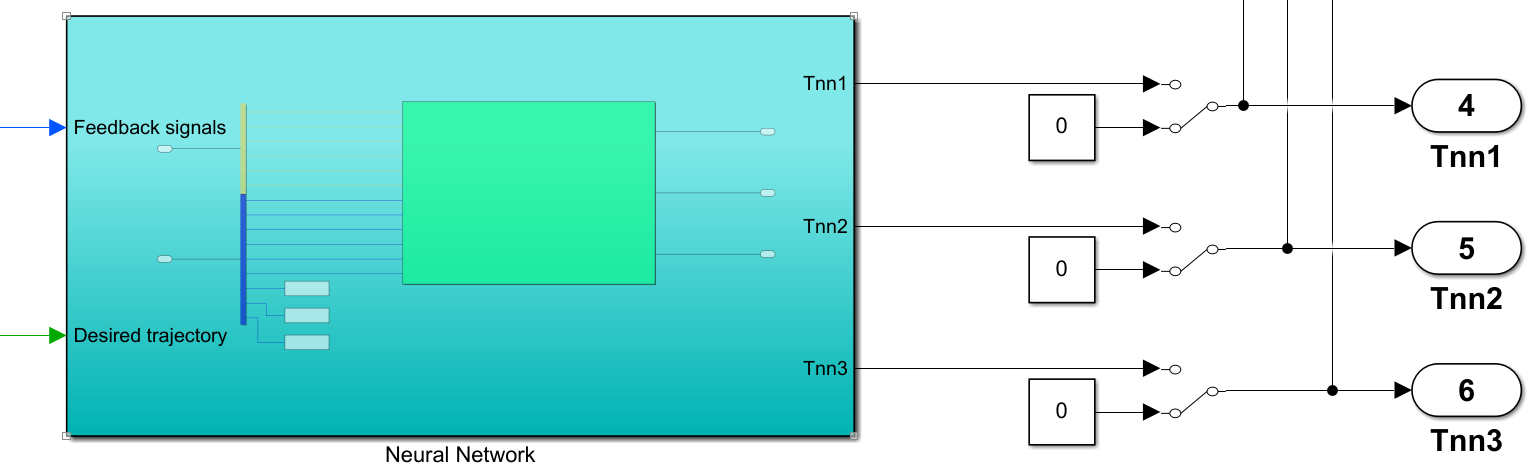
1. This block receives input from the Feedback signal (Mechanical model block) and the Desired trajectory (Inverse Kinematic block).
2. Add Demux to extract the value inside 2 inputs.



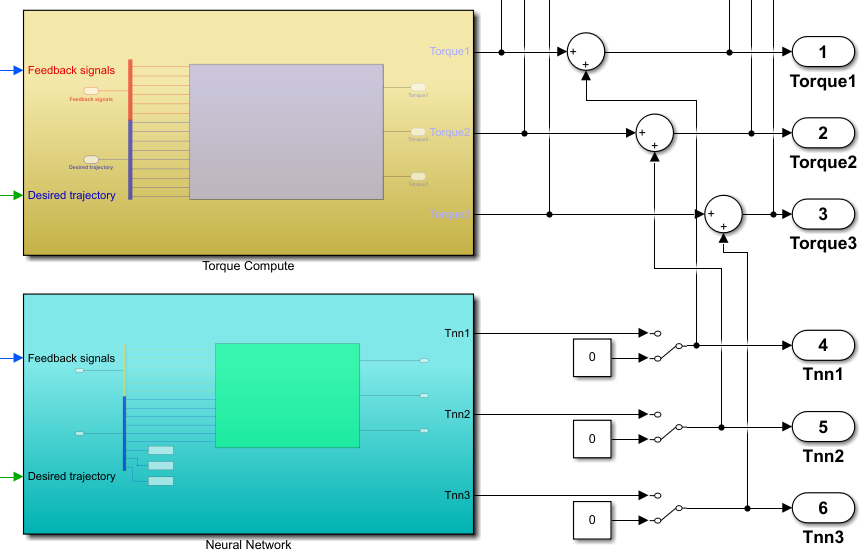
Code:

| function [Tnn1, Tnn2, Tnn3] = fcn(Theta1, Theta2, Theta3, Theta1\_dot, Theta2\_dot, Theta3\_dot, Theta1\_des, Theta2\_des, Theta3\_des, Theta1\_dot\_des, Theta2\_dot\_des, Theta3\_dot\_des)  % Neural Network Controller for RRR Robot (3 DOF)  % Input: 12 inputs (Position and Velocity: Actual vs Desired for 3 joints)  % Output: 3 Torques (Tnn1, Tnn2, Tnn3)  persistent W C X S  % Khoi tao ma tran trong so ban dau  if isempty(W)  % Tang khoi tao len mot chut de mang co tin hieu ban dau  W = 0.1 \* (2\*rand(6, 10) - 1);  C = 0.0 \* zeros(10, 3);  X = ones(6, 1);  S = ones(3, 1);  end  %% 1. Input of neural network (Sai so vi tri va van toc)  % Error position  e1 = Theta1 - Theta1\_des;  e2 = Theta2 - Theta2\_des;  e3 = Theta3 - Theta3\_des;  % Error velocity  ed1 = Theta1\_dot - Theta1\_dot\_des;  ed2 = Theta2\_dot - Theta2\_dot\_des;  ed3 = Theta3\_dot - Theta3\_dot\_des;  % --- KHUECH DAI DAU VAO ---  Input\_Gain = 50;  X(1) = e1 \* Input\_Gain;  X(2) = e2 \* Input\_Gain;  X(3) = e3 \* Input\_Gain;  X(4) = ed1 \* Input\_Gain;  X(5) = ed2 \* Input\_Gain;  X(6) = ed3 \* Input\_Gain;  %X(1) = e1;  %X(2) = e2;  %X(3) = e3;  %X(4) = ed1;  %X(5) = ed2;  %X(6) = ed3;  %% 2. Sliding surface (Mat truot)  lamda1 = 6; lamda2 = 6; lamda3 = 6;  s1 = e1 + lamda1 \* ed1;  s2 = e2 + lamda2 \* ed2;  s3 = e3 + lamda3 \* ed3;  S(1) = s1;  S(2) = s2;  S(3) = s3;  S = S .\* (abs(S) > 0.002);  %% 3. Hidden Layer Computation  H\_net = zeros(10, 1);  H\_out = zeros(10, 1);  % Compute H\_net(j)  for j = 1:10  for i = 1:6  H\_net(j) = H\_net(j) + W(i, j) \* X(i);  end  end  % Compute H\_out(j)  for j = 1:10  Net1 = H\_net(j);  H\_out(j) = (1 - exp(-Net1)) / (1 + exp(-Net1));  end  %% 4. OUTPUT LAYER (Compute Torques)  % --- KHUECH DAI DAU RA ---  % Robot nang hang chuc tan, can Torque hang tram nghin Nm  Scale\_Factor = 500000; % Nhan 500,000 lan  T\_net = zeros(3, 1);  for k = 1:3  for j = 1:10  % Tinh toan thuan tuy  val = C(j, k) \* H\_out(j);  % Nhan voi Scale Factor de ra luc thuc te  T\_net(k) = T\_net(k) + val \* Scale\_Factor;  end  end  %% 5. UPDATE WEIGHTS  % Learning rate (Toc do hoc)  % Vi dau ra da nhan Scale\_Factor lon, nen toc do hoc phai NHO LAI  m = 0.02;  n = 0.02;  % Update for Output layer (C)  for j = 1:10  for k = 1:3  C(j, k) = C(j, k) - m \* S(k) \* H\_out(j);  end  end  % Update for Hidden layer (W)  for i = 1:6  for j = 1:10  Delta\_Wij = 0;  for k = 1:3  Delta\_Wij = Delta\_Wij + C(j, k) \* S(k);  end  W(i, j) = W(i, j) - n \* Delta\_Wij \* (1 - H\_out(j)^2) \* X(i) / 2;  end  end  %% 6. Final Output with Saturation  % Gioi han bao hoa de tranh robot bi văng (Safety)  T\_limit = 3000000; % 3 Trieu Nm  Tnn1 = max(min(T\_net(1), T\_limit), -T\_limit);  Tnn2 = max(min(T\_net(2), T\_limit), -T\_limit);  Tnn3 = max(min(T\_net(3), T\_limit), -T\_limit);  end |
| --- |

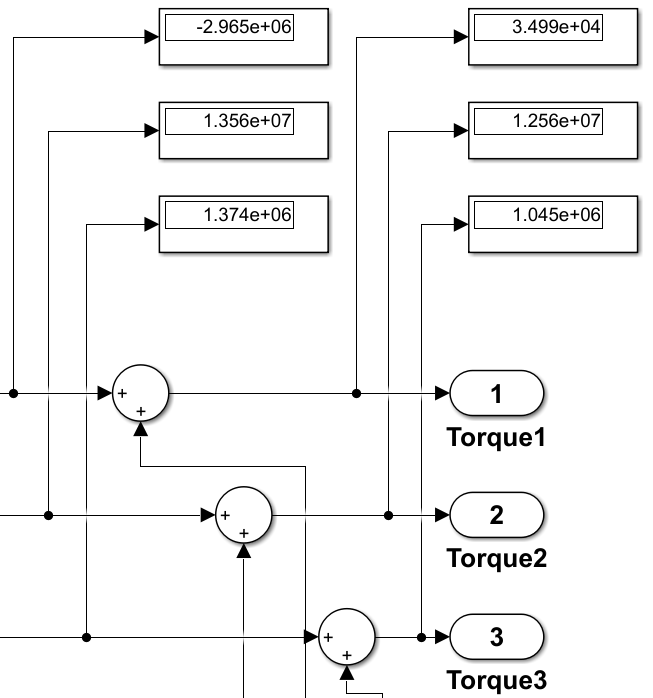
1. Add a manual switch block for easy testing with/ without a Neural network. Turn off the switch to test what will happen if you add more weight to the robot, and turn it on after that to figure out how the Neural Network improves the performance.



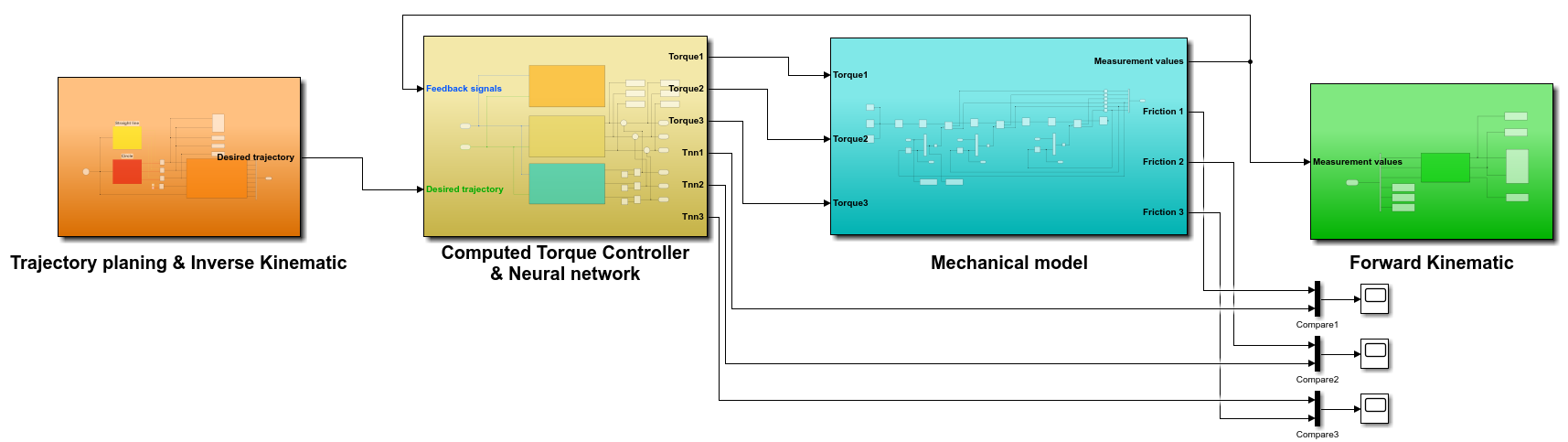
1. Connect the Neural Network output to the Add block with the Torque controller.



1. Add the display block to measure and compare the torque force before and after adding the Neural Network



**Step 3:** Run the model in both cases with and without a Neural Network for 60 seconds.



1. Detail blocks setup can be found in Appendix B [↑](#footnote-ref-0)
2. Detail blocks setup can be found in Appendix C [↑](#footnote-ref-1)
3. Detail blocks setup can be found in Appendix A [↑](#footnote-ref-2)
4. Detail calculation is at APPENDDIX B [↑](#footnote-ref-3)
5. Detail blocks setup can be found in Appendix B [↑](#footnote-ref-4)
6. Detail blocks setup can be found in Appendix C [↑](#footnote-ref-5)