KINEMATICS AND DYNAMIICS

MET 5800

FINAL PROJECT REPORT

MODELING AND SIMULATION OF A 6 DEGREE OF FREEDOM ROBOT (RRRRRP)

SUBMITTED BY:- BALAJI NAMMALVAR

GUIDED BY: Mr. DAVID M. LABYAK PhD.

Michigan Technological University

INTRODUCTION: -

PROBLEM STATEMENT:

Productive and rigorous assembly execution using 6 degrees of freedom (RRRRRP) Robotic manipulator by performing kinematics approach in MATLAB, Simulink and simscape. We will be using MATHLAB, Simulink and simscape for generating trajectories of the output after compilation and will be comparing the inputs and outputs trajectories to get the error trajectory. These trajectories generated can be used for analyzing and finding how accurate the manipulator's movement is, can it get to certain point we specify or if there is any singularity present.

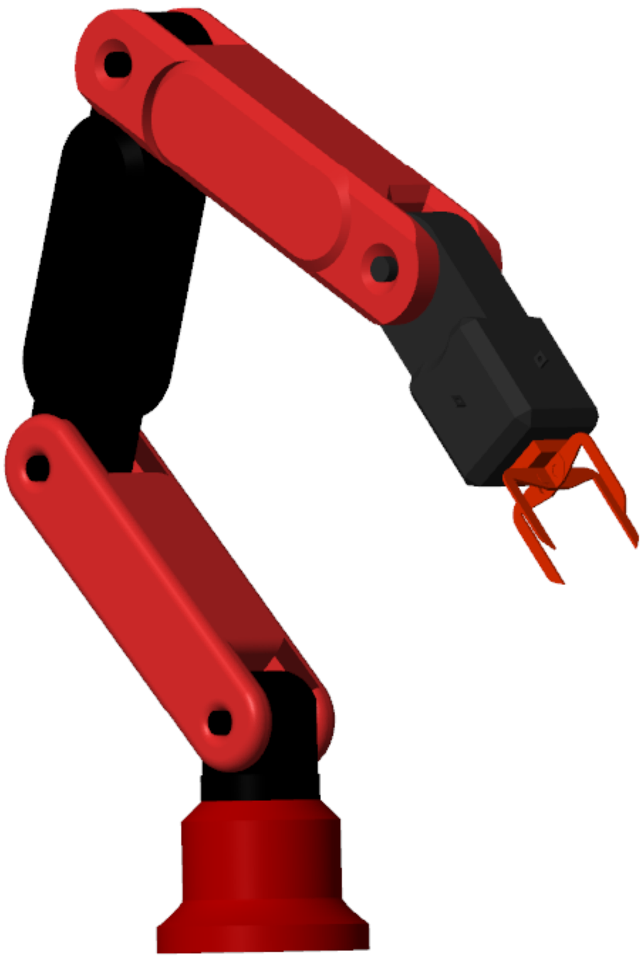


Figure 1: 6 DOF Robotic Manipulator

<u>METHODS: -</u>

1) Design a 6 degree of freedom /robotic /manipulator using any modeling software. I have used SolidWorks software for modeling.

<u>Step 1:</u>

Start with designing the base of your manipulator, make some diagrams. For better understanding and easy modelling sketch your idea using engineering graphics (Top view, Side view and front view).

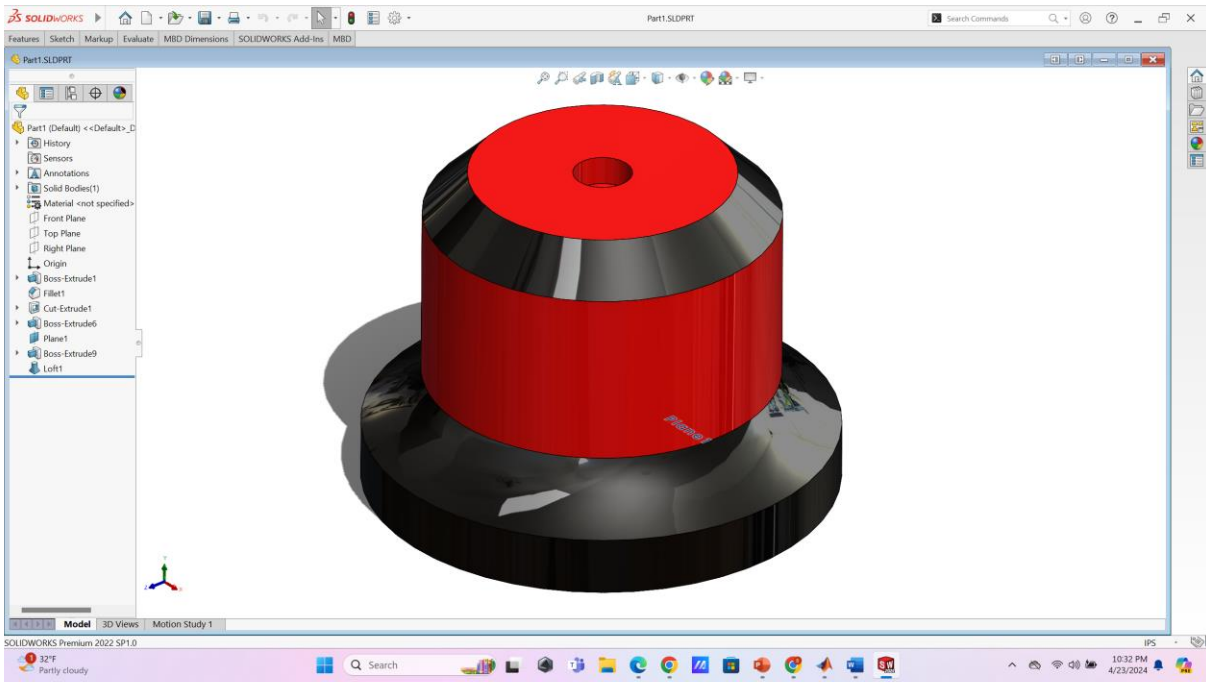


Figure 2: Fixed Base

The above image depicts the base which is designed in solid works make sure you extrude everything in correct axis. I made a mistake by extruding everything in y axis without my knowledge. If you extrude in wrong axis either it will be difficult to assemble in simscape or might need to set different axis after importing step file in file solid block while creating your Simulink rigid body tree.

Step 2: -

Start to create your robotic manipulator's joint links one by one like revolute or prismatic depending on the motion you need like translation or rotation.

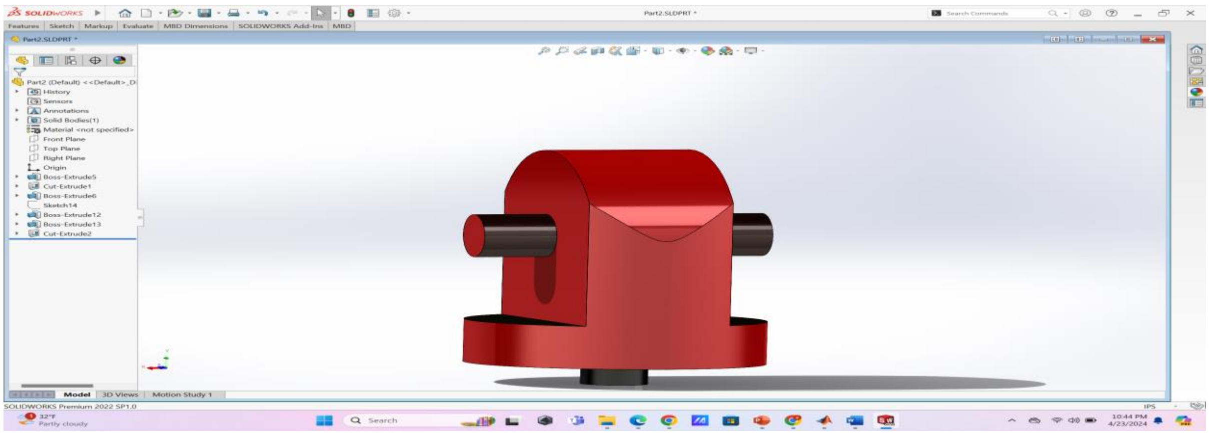


Figure 3: link 1

This is the first revolute joint made.

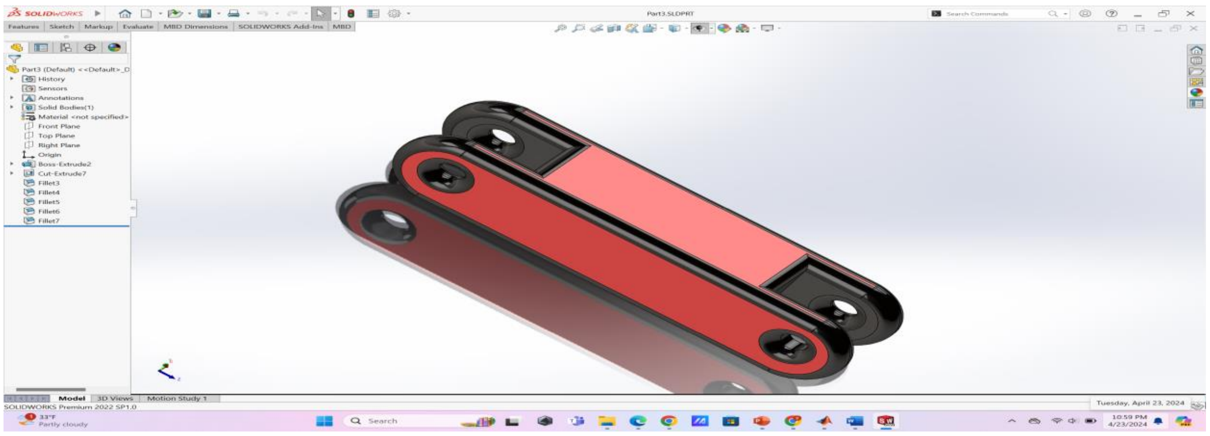


Figure 4: link2

Second revolute joint

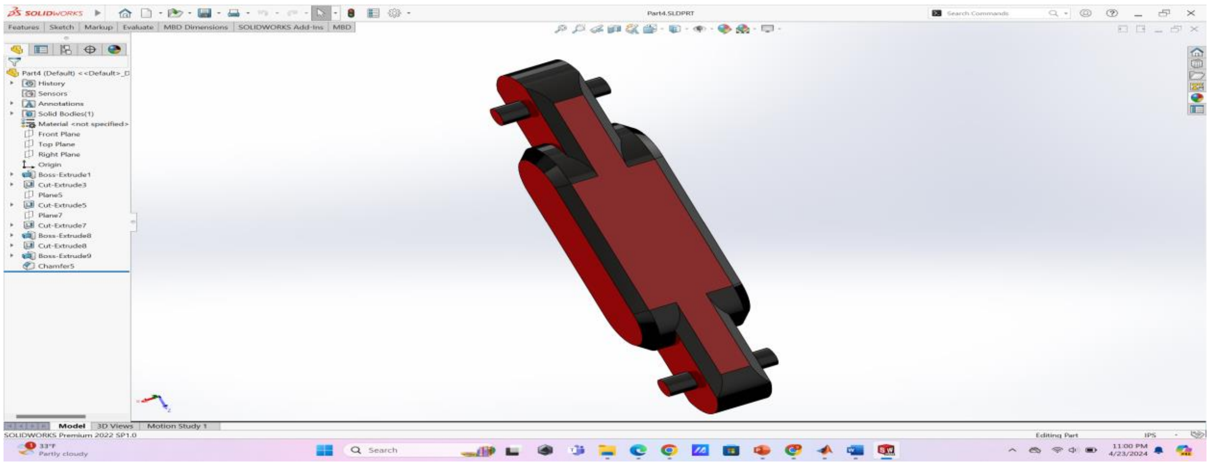


Figure 5:link 3

Third revolute joint

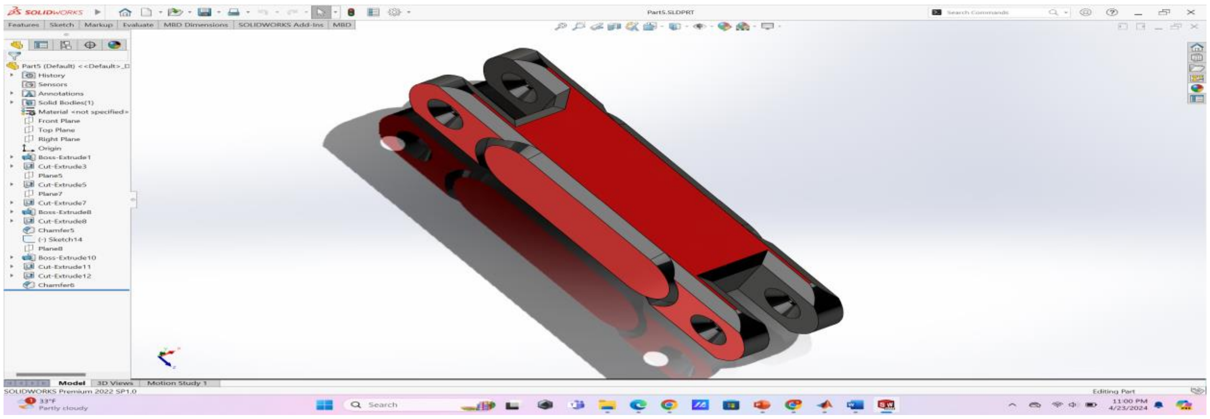


Figure 6: link 4

Fourth revolute joint

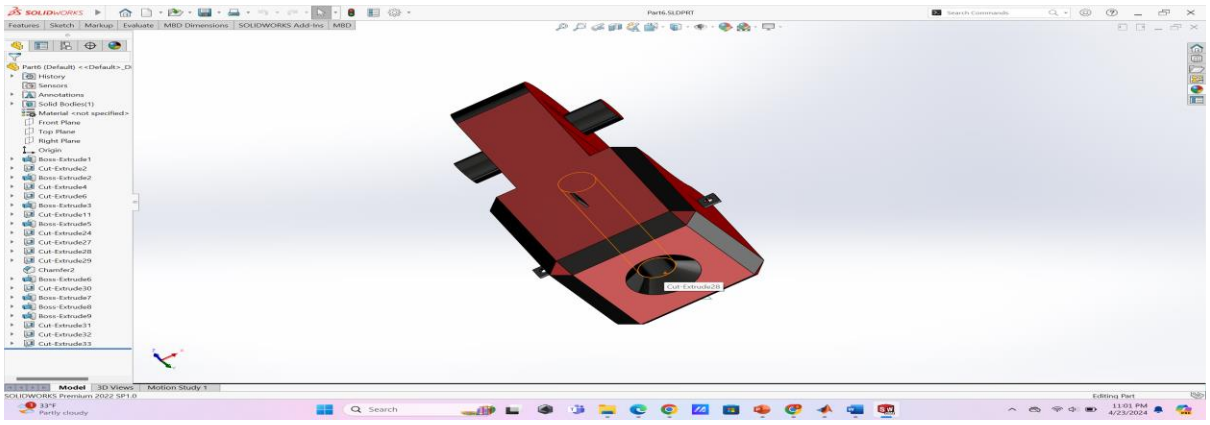


Figure 7: link 5

Fifth revolute joint

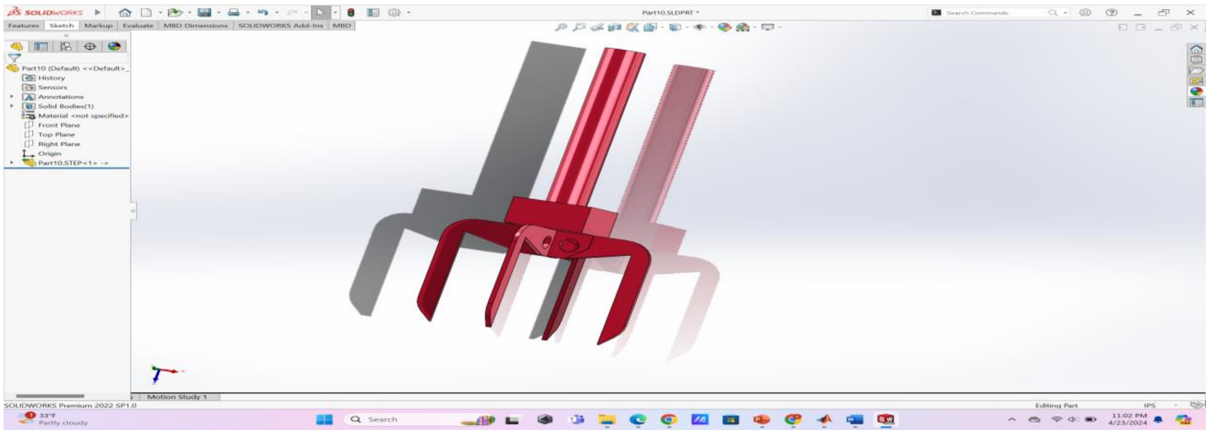


Figure 8: Link 6

Rigid End Effector with sixth joint prismatic

Step 3: -

After designing the required parts, you can start importing these step files into Simulink using file solid block directly and make your rigid body tree as shown below.

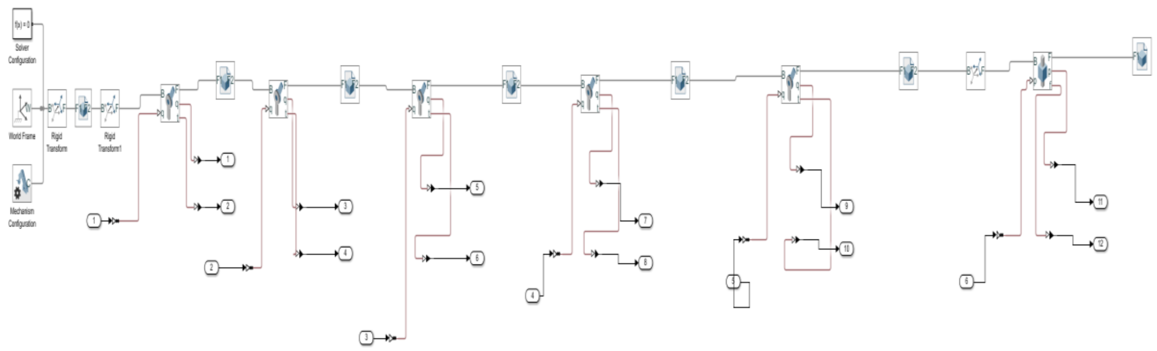


Figure 9: Rigidbodytree/Subsystem

Each link designed has been imported into simscape using file solid block in Simulink. The position vectors are provided as input to each of the joints from the inverse kinematics block for the motion. Joint torques, and velocity are taken as Outputs from each joint. The above rigid body tree has 5 revolute joints and a prismatic joint (RRRRRP).

Step 4: -

Start creating your main model like in the image shown below.

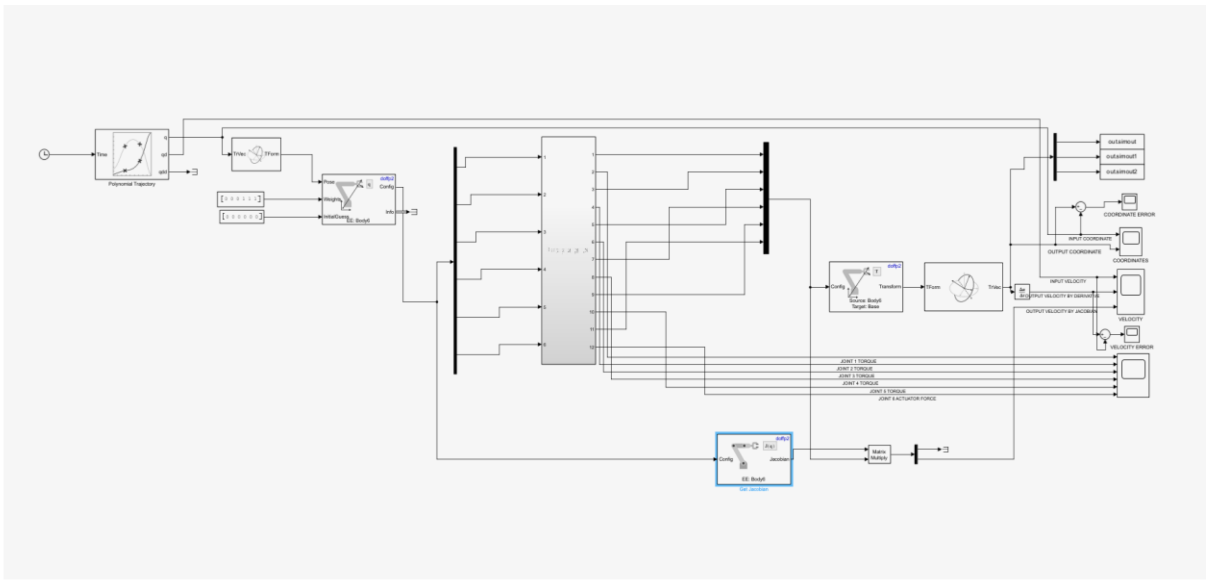
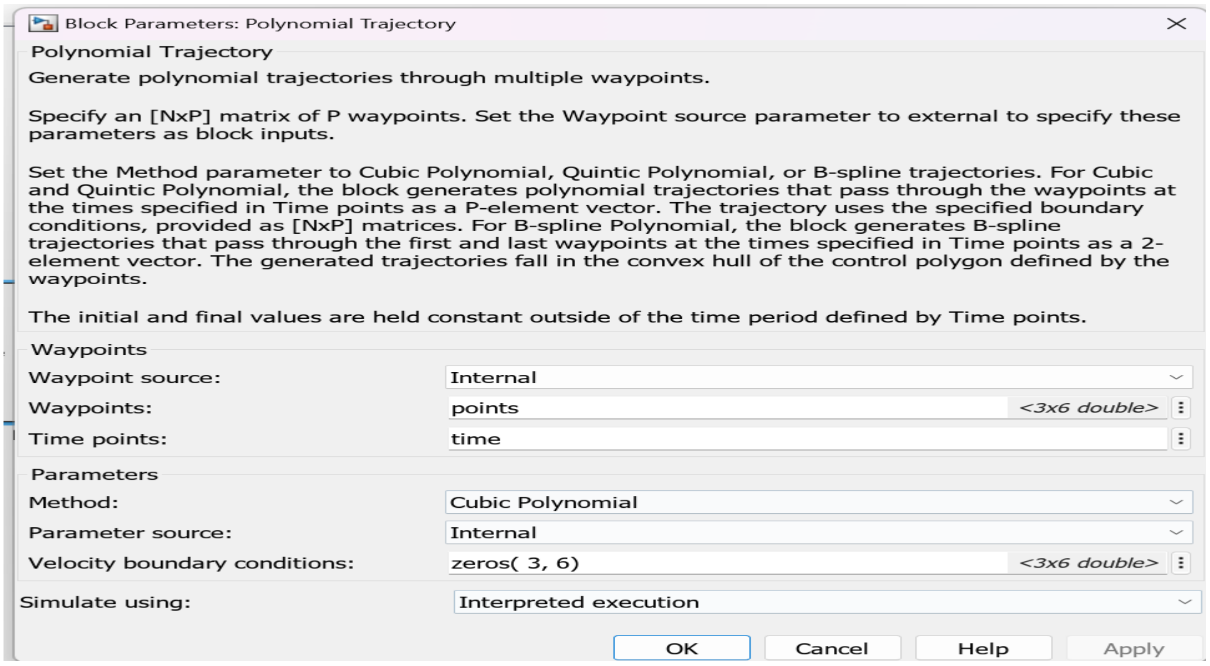


Figure 10: Main diagram

Have used polynomial trajectory for getting smooth curve output. The waypoints and time parameters are given as input to the polynomial trajectory block. Then the output of polynomial trajectory block will be position, velocity and acceleration, and these outputs can be used further to get desired results. The position output is fed as input to coordinate transformation conversion block which converts the position vector into homogeneous transformation matrix. Now for inverse kinematics block homogeneous transformation, appropriate weight and initial guess values are given as input based on number of joints we have. The output from the inverse kinematics block is joint positions vector and is given as input for the subsystem. Now based on the input joint positions given the joints can generate some torque, force and velocities also. The torque values are directly fed to scope, and it can be visualized in this scope clearly. Since I have taken 6 joints which is 5 revolute joints and a prismatic joint. So, from these I would get 5 torque graphs and a graph for actuator force of prismatic joint. The velocities which we get from the subsystem are combined using a mux block and directly fed to get transform block which generates or converts the position vectors to homogeneous transformations matrix, and this output is given as input to another “get Transform block” which converts this homogeneous transformation block to translation or position vectors. These position vectors can be directly used to plot the trajectory and visualize it in the scope block. I have used a derivative block after the coordinate transformation block to get the output velocity. I have also used a Jacobian block to get the linear velocities by using the position vector from the inverse kinematic block and plotted the trajectory and visualized it using scope. I have plotted the error between input and output coordinates or positions got. Have also plotted the error between the input and output velocities got. And finally, I have used demux block to get individual x,y and z coordinates and have and used “to workspace "plots for visualizing the path travelled by the manipulator.

Step 5: - Setting correct parameters for each block



Polynomial Trajectory

Generate polynomial trajectories through multiple waypoints.

Specify an [NxP] matrix of P waypoints. Set the Waypoint source parameter to external to specify these parameters as block inputs.

Set the Method parameter to Cubic Polynomial, Quintic Polynomial, or B-spline trajectories. For Cubic and Quintic Polynomial, the block generates polynomial trajectories that pass through the waypoints at the times specified in Time points as a P-element vector. The trajectory uses the specified boundary conditions, provided as [NxP] matrices. For B-spline Polynomial, the block generates B-spline trajectories that pass through the first and last waypoints at the times specified in Time points as a 2-element vector. The generated trajectories fall in the convex hull of the control polygon defined by the waypoints.

The initial and final values are held constant outside of the time period defined by Time points.

Waypoints

Waypoint source: Internal

Waypoints: points <3x6 double>

Time points: time

Parameters

Method: Cubic Polynomial

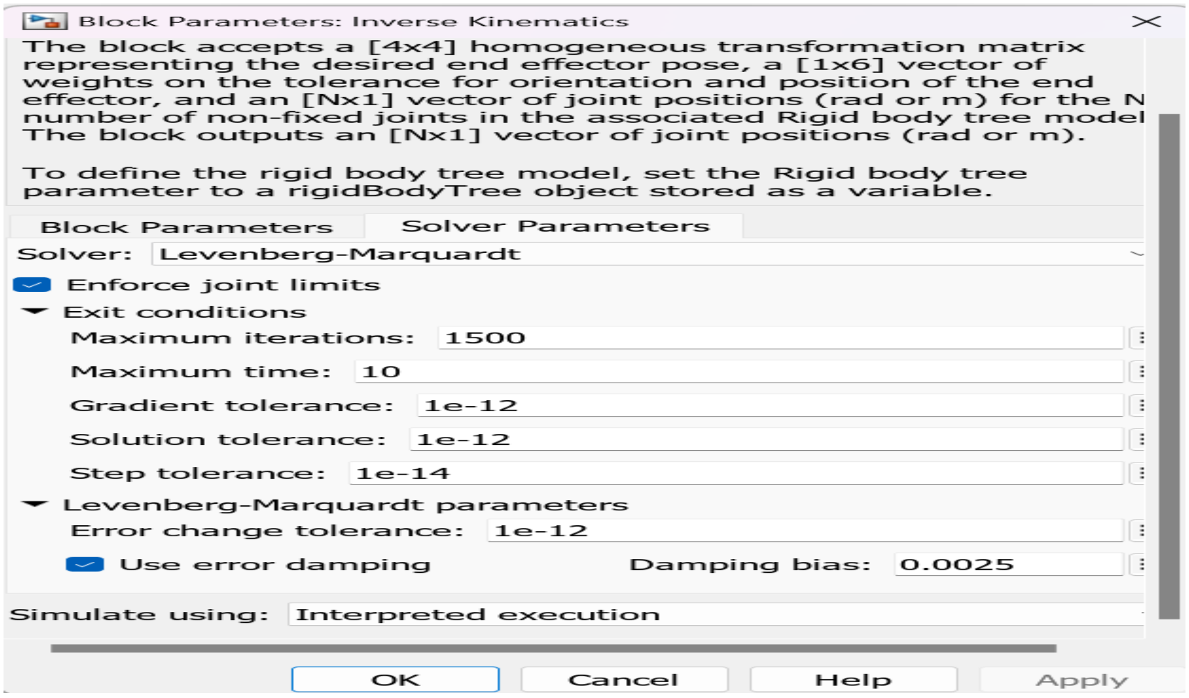
Parameter source: Internal

Velocity boundary conditions: zeros(3, 6) <3x6 double>

Simulate using: Interpreted execution

Figure 11: Polynomial trajectory parameters

As you can see, the above waypoints are given as input. Time is also given as input for the polynomial trajectory block. Velocity boundary conditions are also given and have 3 rows and 6 columns.



The block accepts a [4x4] homogeneous transformation matrix representing the desired end effector pose, a [1x6] vector of weights on the tolerance for orientation and position of the end effector, and an [Nx1] vector of joint positions (rad or m) for the N number of non-fixed joints in the associated Rigid body tree model The block outputs an [Nx1] vector of joint positions (rad or m).

To define the rigid body tree model, set the Rigid body tree parameter to a rigidBodyTree object stored as a variable.

Solver: Levenberg-Marquardt

✓ Enforce joint limits

Exit conditions

Maximum iterations: 1500

Maximum time: 10

Gradient tolerance: 1e-12

Solution tolerance: 1e-12

Step tolerance: 1e-14

Levenberg-Marquardt parameters

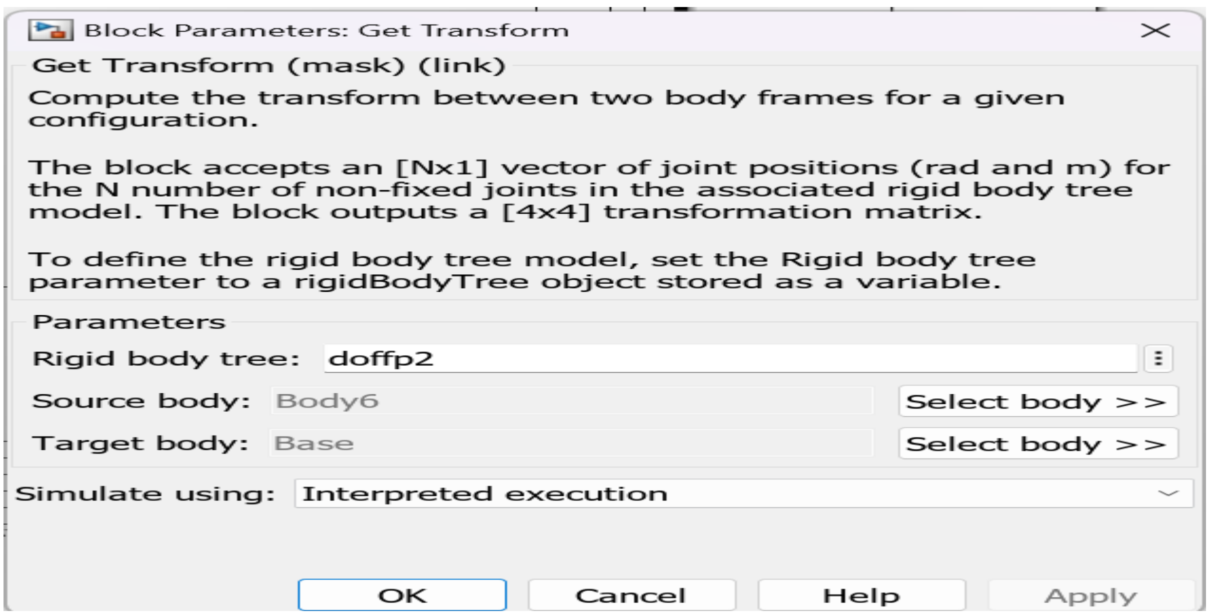
Error change tolerance: 1e-12

✓ Use error damping   Damping bias: 0.0025

Simulate using: Interpreted execution

Figure 12: Inverse kinematics Block solver parameters

As you can see above, I have maximized tolerance to the fullest to achieve smooth and correct error plot and all other parameters too like velocity and position vectors.


Block Parameters: Get Transform

Get Transform (mask) (link)

Compute the transform between two body frames for a given configuration.

The block accepts an [Nx1] vector of joint positions (rad and m) for the N number of non-fixed joints in the associated rigid body tree model. The block outputs a [4x4] transformation matrix.

To define the rigid body tree model, set the Rigid body tree parameter to a rigidBodyTree object stored as a variable.

Parameters

Rigid body tree: doffp2

Source body: Body6 — Select body >>

Target body: Base — Select body >>

Simulate using: Interpreted execution

Figure 13: Get Transform Block parameters

These are the Get Transform block parameters above. It requires source body and target body to be selected. In my case the source body is base which is fixed, and target body is the end effector which is body 6.

The tolerance value plays a major role in determining the trajectory and reducing the error values in a big scale even though it takes a lot of time to complete depending on the configuration of the device you are using to run the MATLAB, Simulink and simscape.

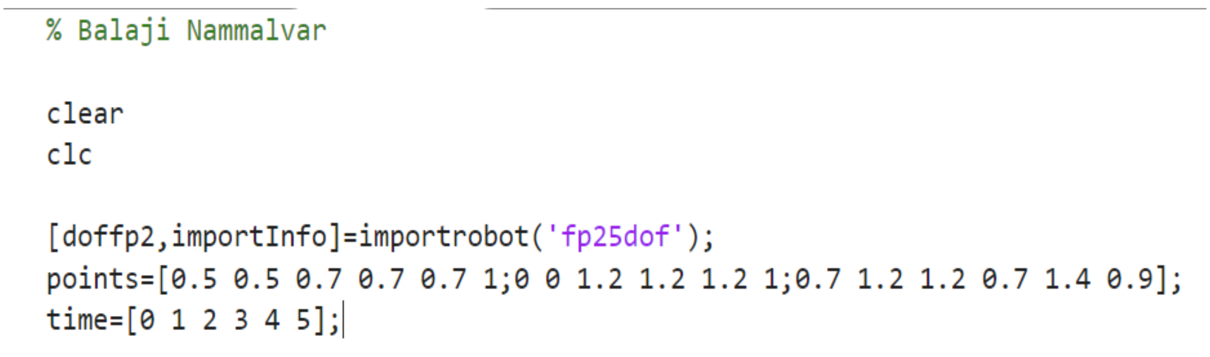
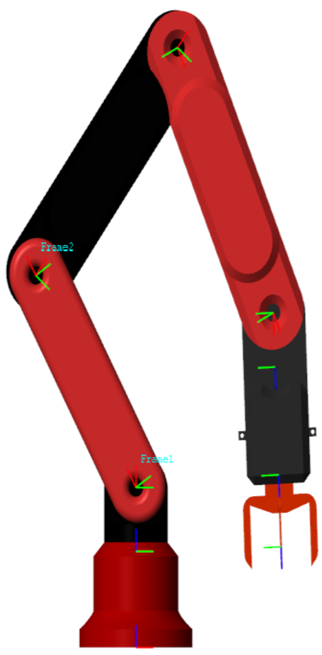

```
% Balaji Nammalvar

clear
clc

[doffp2,importInfo]=importrobot('fp25dof');
points=[0.5 0.5 0.7 0.7 0.7 1;0 0 1.2 1.2 1.2 1;0.7 1.2 1.2 0.7 1.4 0.9];
time=[0 1 2 3 4 5];
```

Figure 14: Waypoints for the polynomial trajectory

These are the waypoints given for my robotic manipulator (note that all the points are in meters).

If you follow the above steps correctly you will have a robot like the below in simscape.



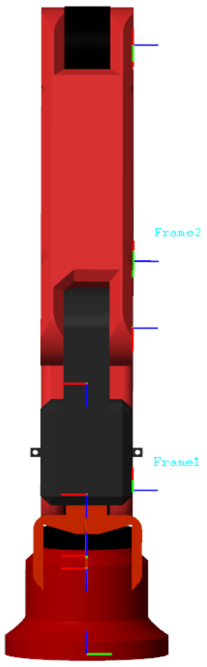


Figure 15: side view with axis of rotation

Figure 16: front view with axis of rotation

SIDE VIEW

FRONT VIEW

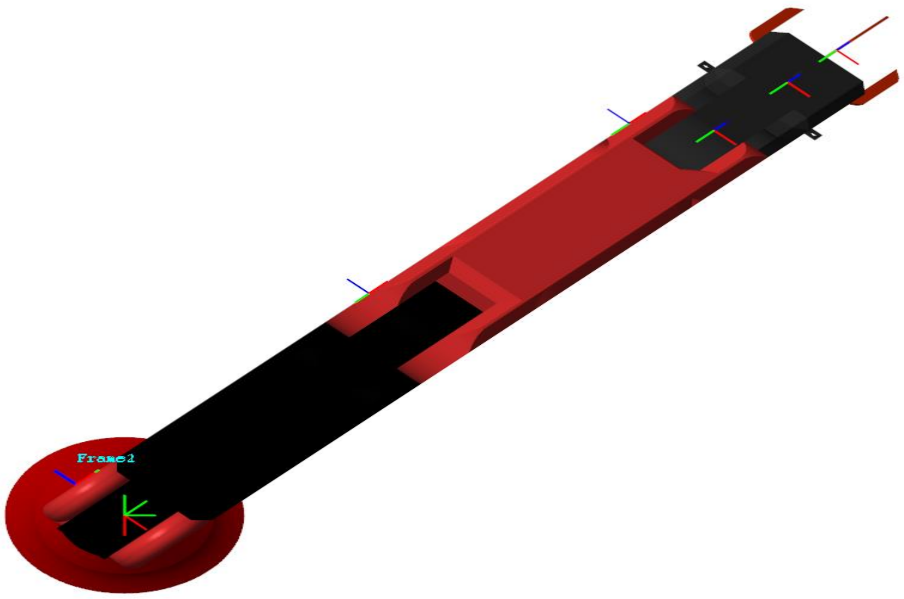


Figure 17: Top view with axis of rotation

TOP VIEW

Depending on the angle of each joint the views may vary as shown in the above diagrams.

RESULT: -

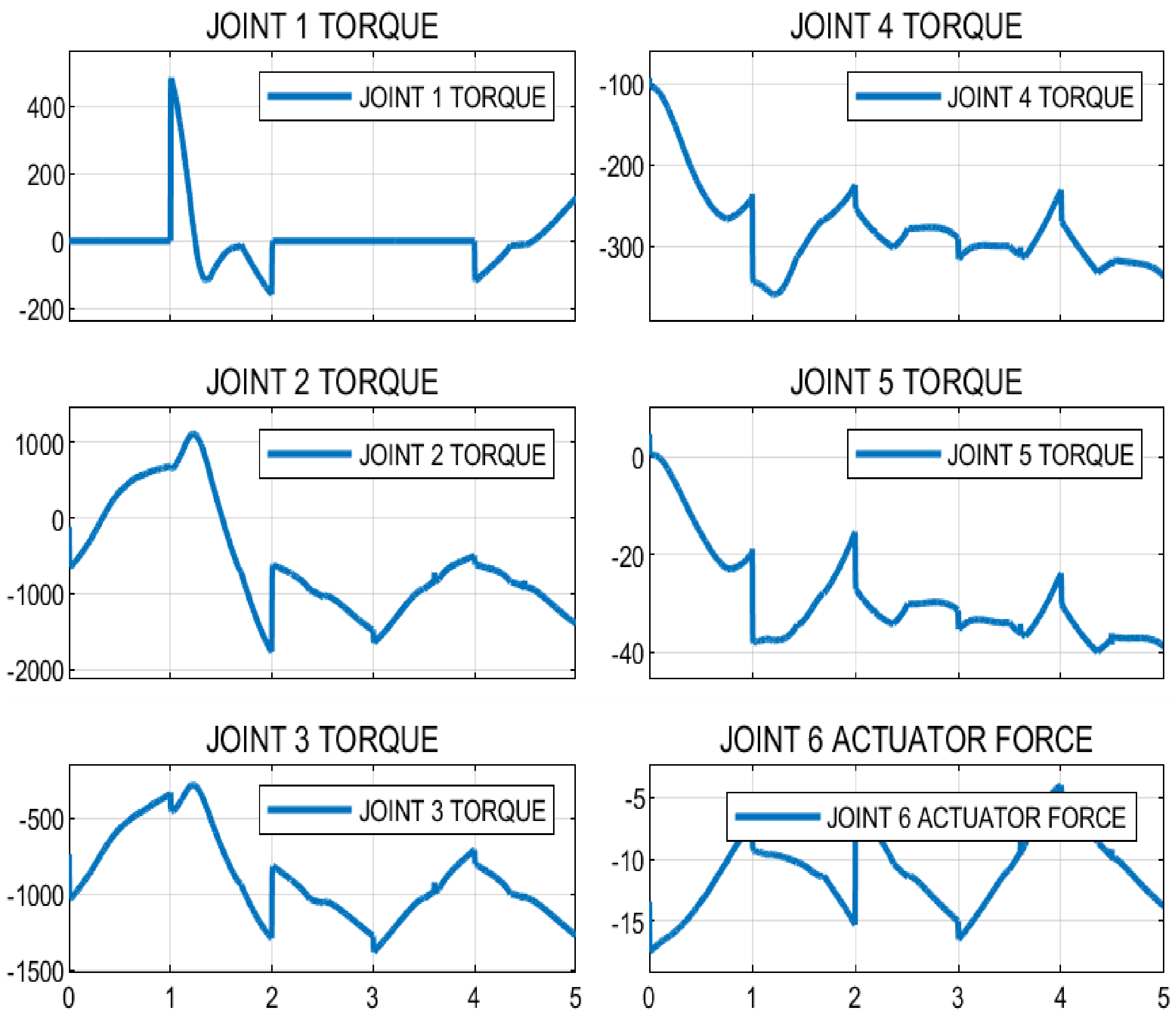


Figure 18: Output Torques

We can see the joint torque trajectories plotted for each joint. We have 6 joints. The first four graphs represent the joint torques of 5 revolute joints. The last graph, that's the 6th one, represents the actuator force of the prismatic joint.

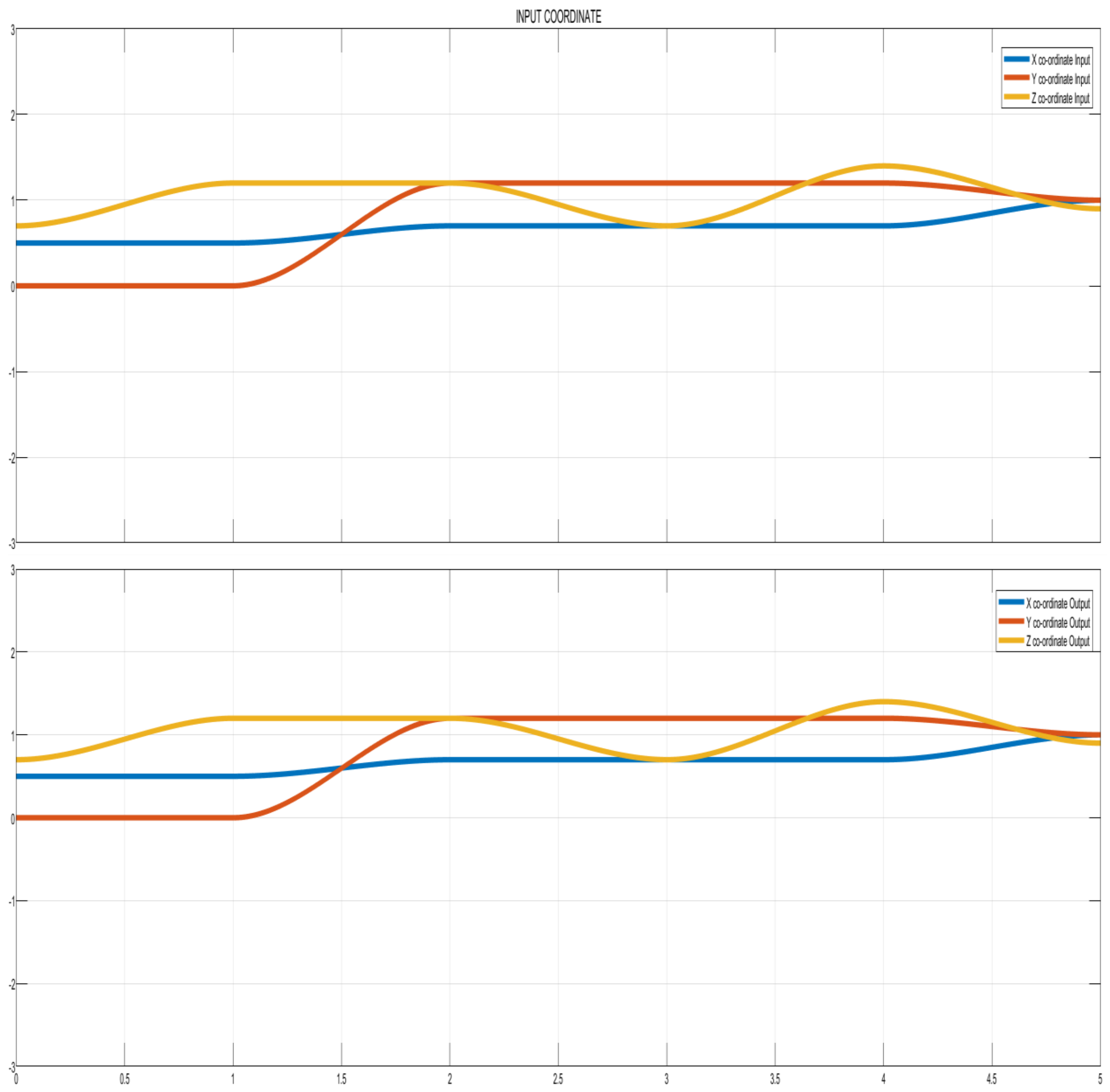


Figure 19: Input and Output Coordinates plot

The above graph depicts the input and output coordinates trajectory. All the values/units are in meters. The first graph is the input coordinate graph. The second graph is the output coordinate graph.

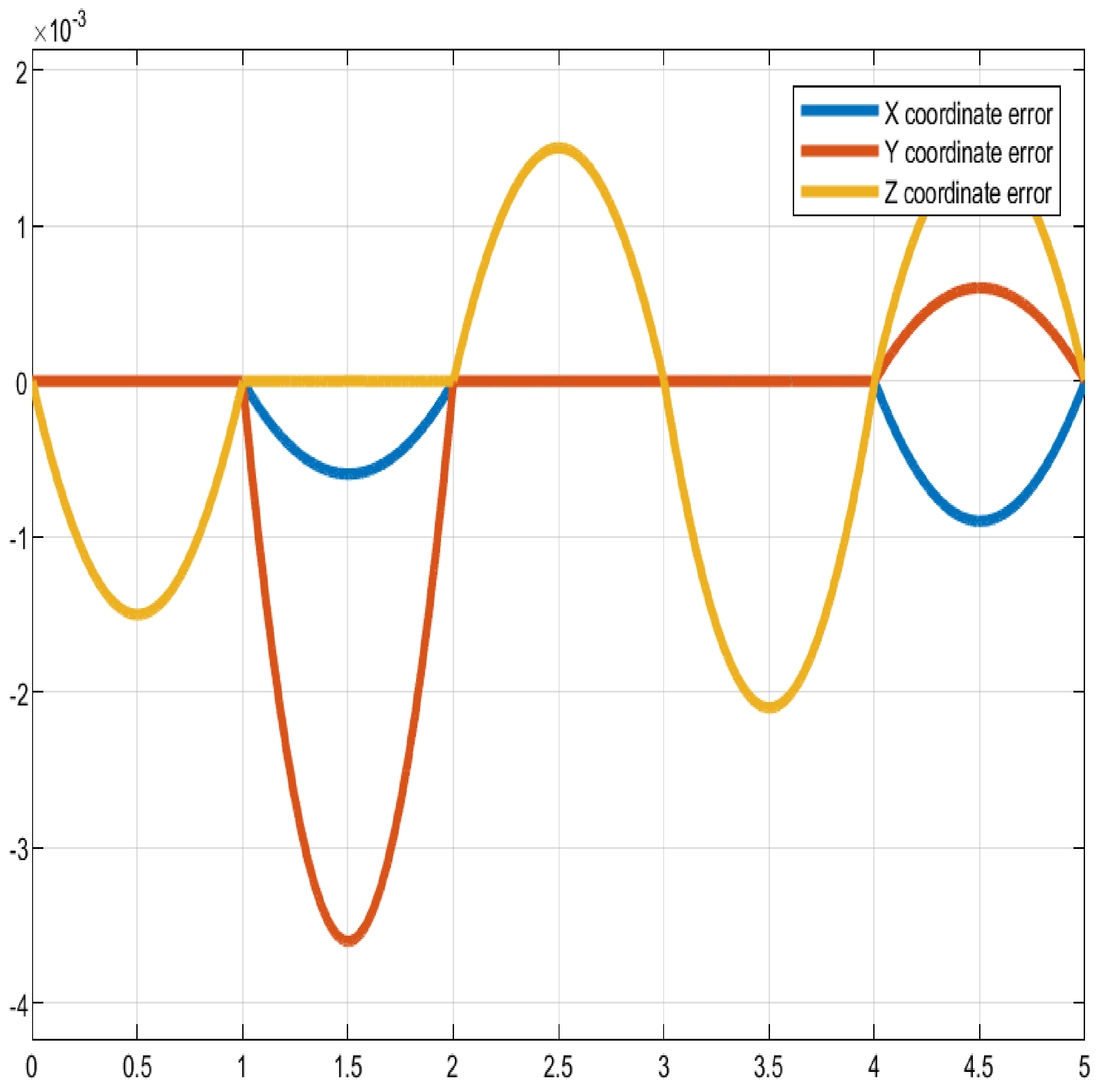


Figure 20: Input and Output Coordinates Error plot

As we can see the above generated plot is for coordinate error. The values are very low, which is about 10^-3 units. Since the error value is low it is trustable and can be used for many applications.

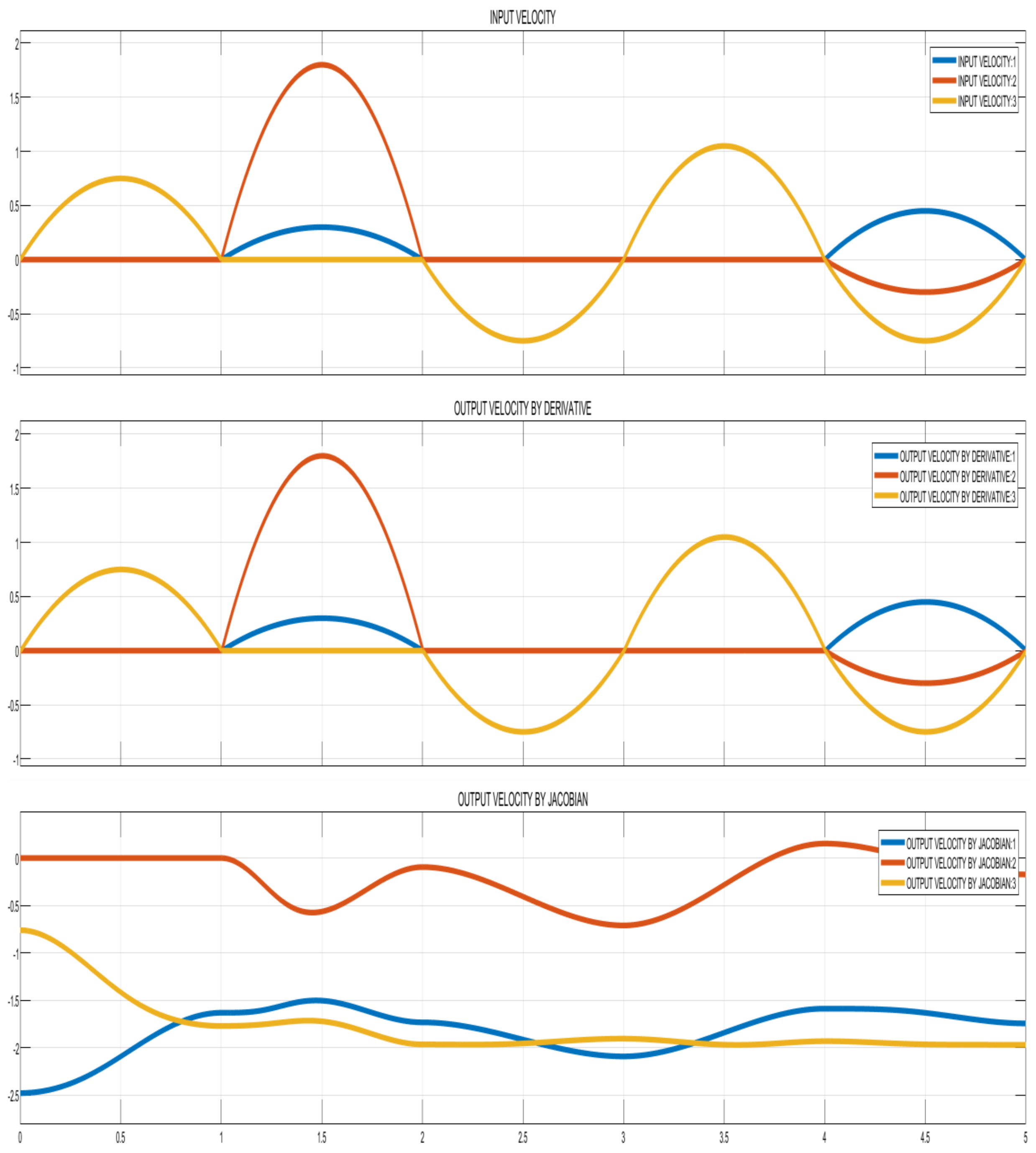


Figure 21: Input and Output Velocity plot

The above graph depicts the trajectories of the input, output and Jacobian velocities. Since polynomial trajectory the curves are smooth. We couldn't see any big difference between the input and output velocity trajectories since they look kind of similar.

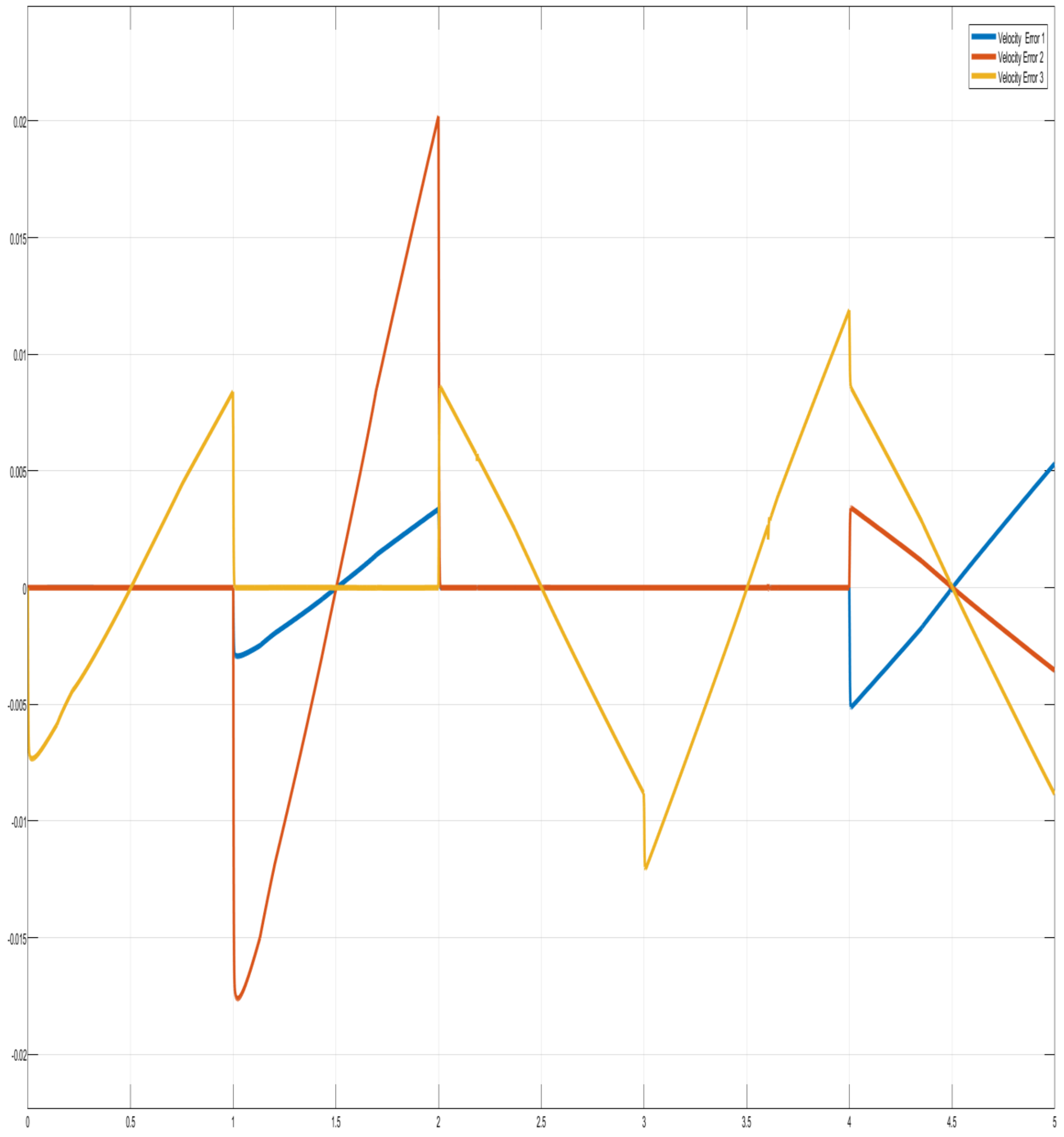


Figure 22: Input and Output Velocity Error plot

This is the error plot obtained for velocity trajectories. As we can see the error is very minimal, which is less than 0.02 as peak. These indicate that the robotic system is performing the task good.

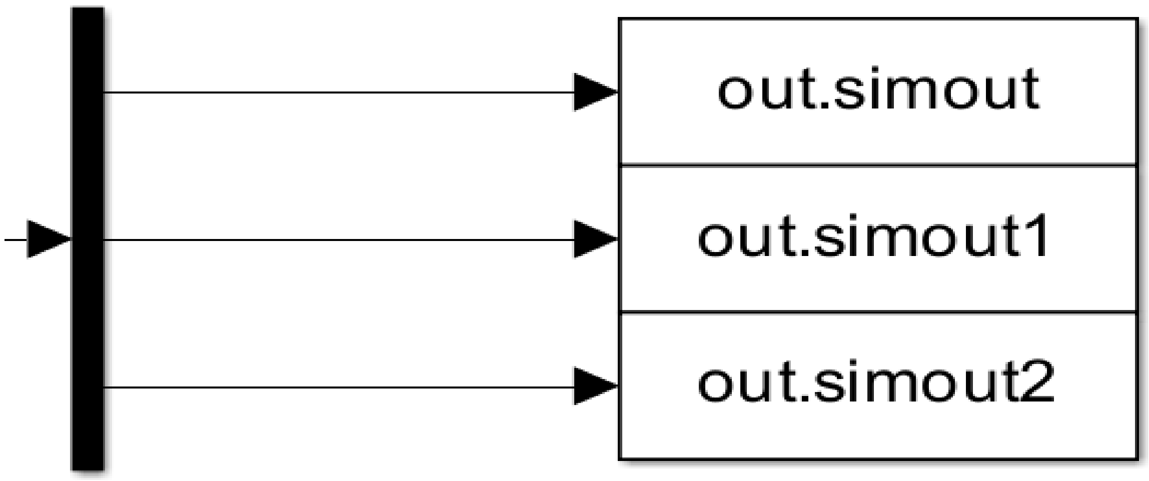
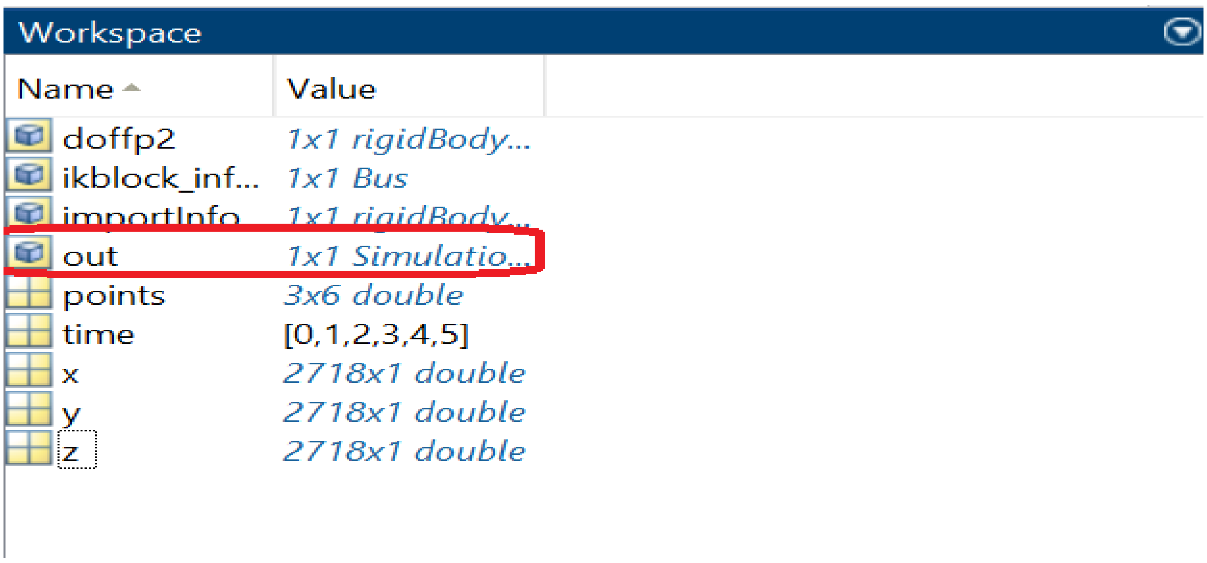


Figure 23: To workspace block

The above is known as the “To Workspace” block. Initially I used demux block to get the individual x,y and z coordinates. This coordinates as given as input to the “to Workspace block. This block extracts all the coordinate data from Simulink to Matlab Workspace. After extracted to workspace you can use that information to plot the path of the robotic manipulator travel.


Workspace

Name	Value
doffp2	1x1 rigidBody...
ikblock_inf...	1x1 Bus
importInfo	1x1 rigidBody...
out	1x1 Simulatio...
points	3x6 double
time	[0,1,2,3,4,5]
x	2718x1 double
y	2718x1 double
z	2718x1 double

Figure 24: T0 workspace extracted data plot

The above is the path travelled by the robotic manipulator in 3d. After the data is extracted to workspace you can use the below command to achieve the above plot.

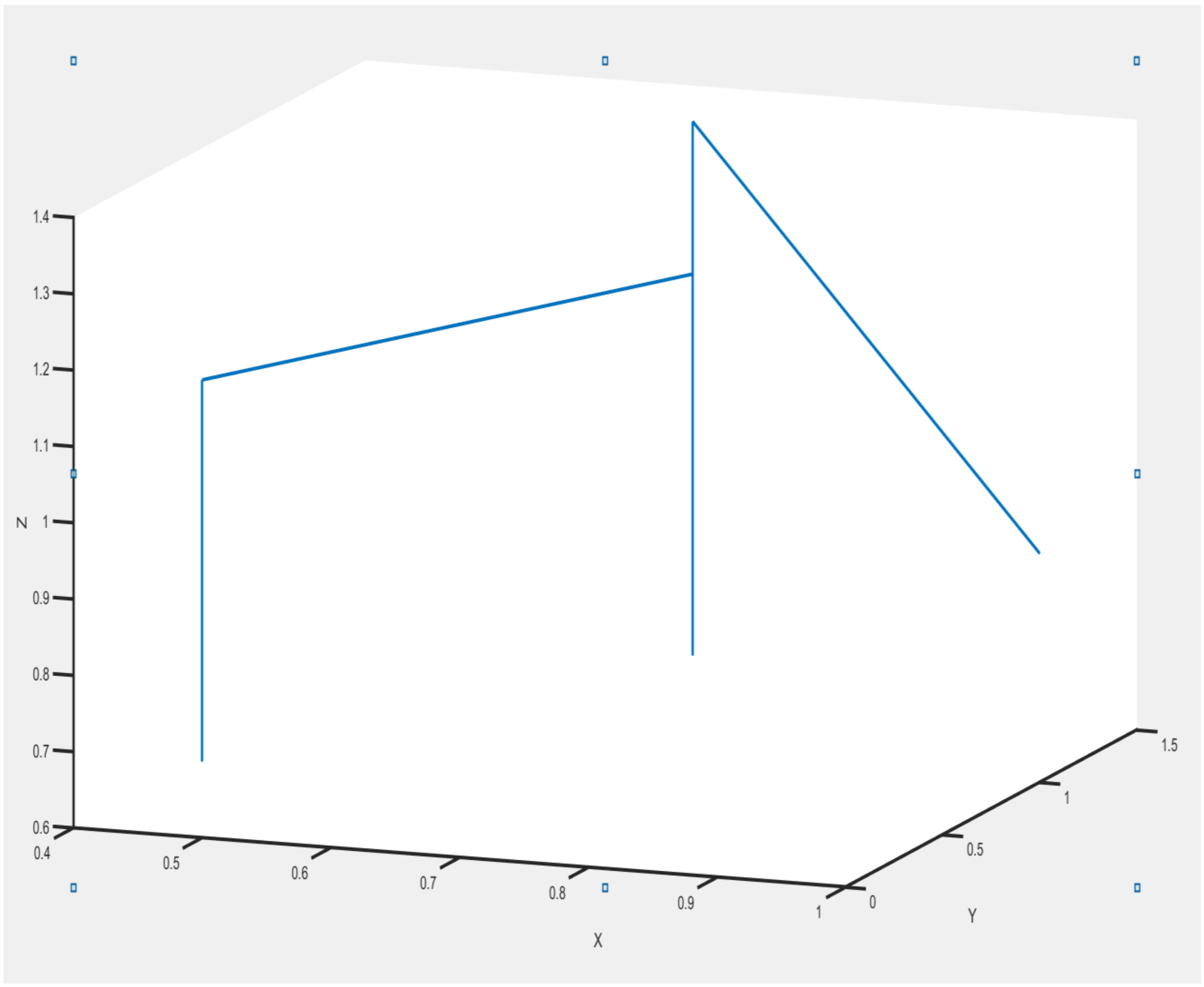


Figure 25: Robotic manipulator path plot

COMMAND WINDOW CODE:

```
x=squeeze(out.simout.Data);

y=squeeze(out.simout1.Data);

z=squeeze(out.simout2.Data);

plot3(x,y,z);

xlabel("X");

ylabel("Y");

zlabel("Z");
```

CONCLUSION: -

From the above result we can observe that the model or the subsystem which is fed with the position config is able to complete the assembly task with just very small error in the position and Velocity. This results in the precise and accurate motion of the robotic manipulator where a certain point can be reached without any big error.

In summary, the velocity graph, coordinate/position graph, error graph helps in analyzing the nature of the robotic manipulator which will be helpful in monitoring the system whole time and maintaining it accordingly if any fault or malfunction occurs.

MET 5800 was very useful for making this project, understanding the core concepts kinematics and dynamics of robot made it easier to do this. After all the lab sessions gave better understanding and were helpful without any doubt.

REFERENCES: -

- MET 5800 Class and Lab material.
- https://youtu.be/vtJafCOVk_s?si=-Ym91ytdm9kCDGEj (for designing the end effector).