Robotic Arm Project

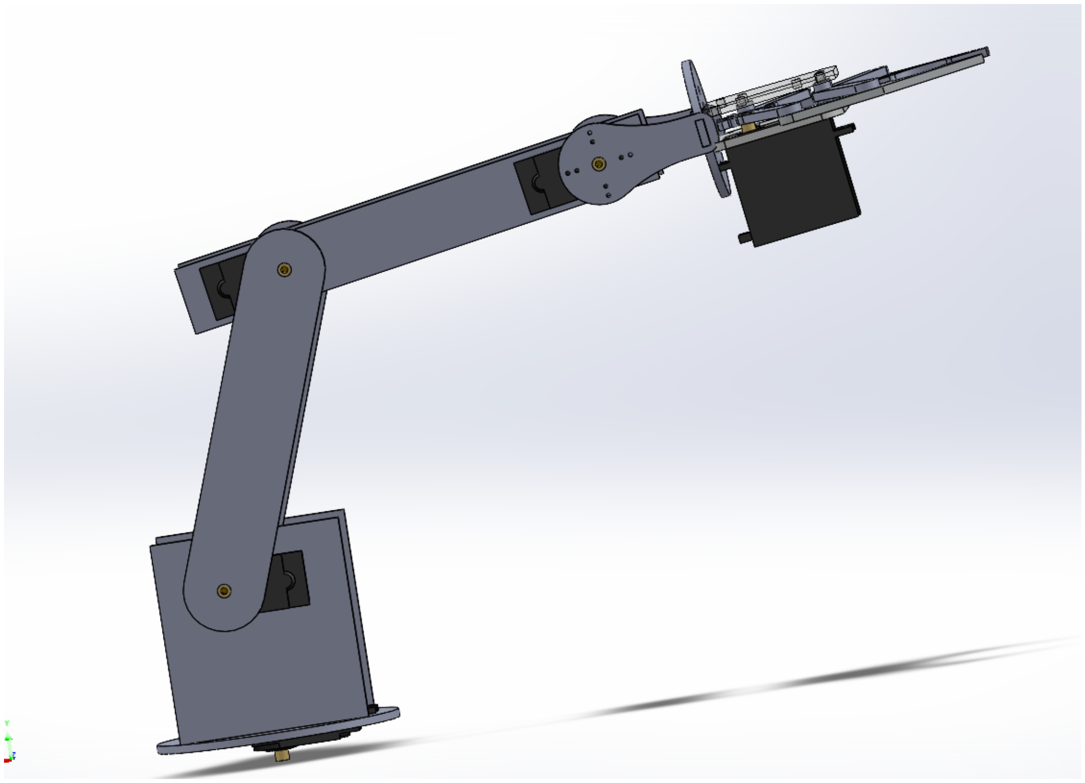
Due Tuesday 12 December 2023

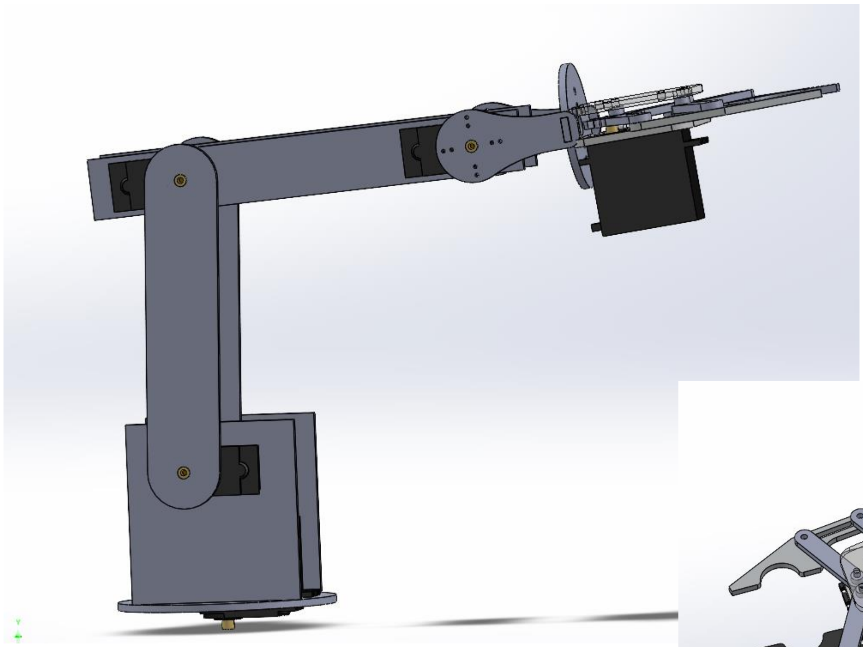
Represented by:

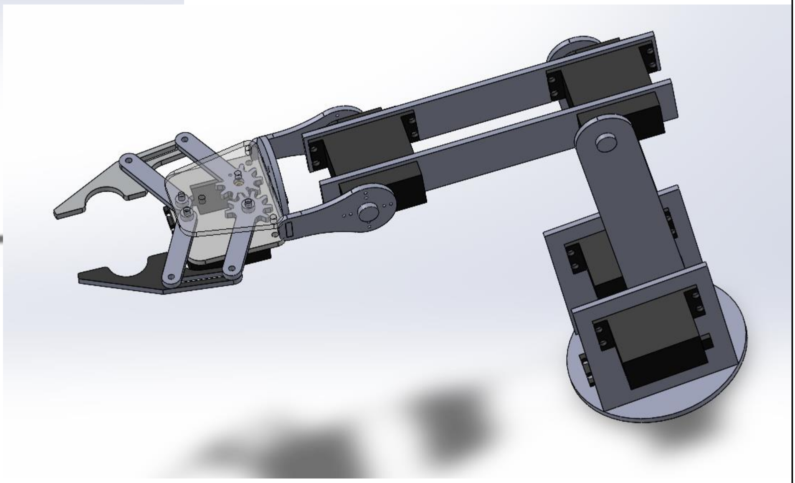
Omneya Haytham Mohamed Ezz Eldin	**7717**
Mohamed Hussein Mohamed Elzoheiry	**7818**
Faress Ahmed Mohamed Amin	**7925**
Ahmed Mohamed Kotp	**7543**
Ahmed Reda	**7417**

Mechatronics

➢ SolidWorks Design:

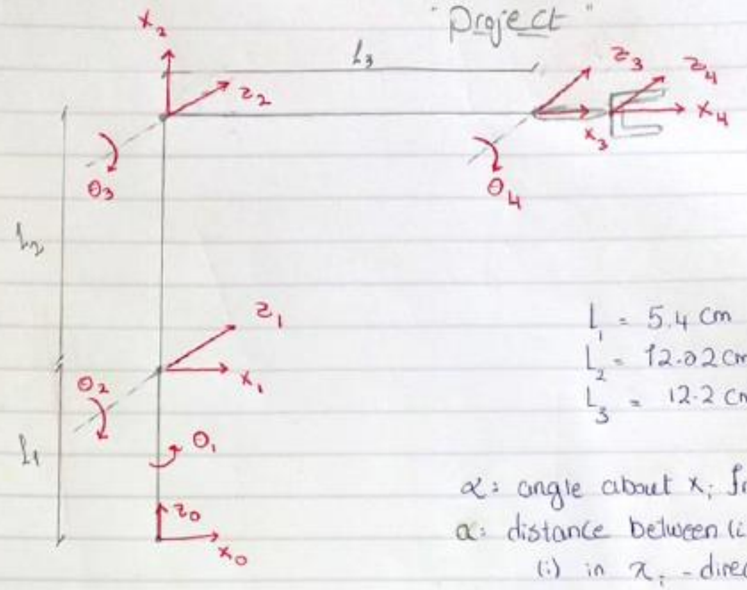






➢ Forward Kinematics (DH Table):

"Forward Kinematics"
"Project"



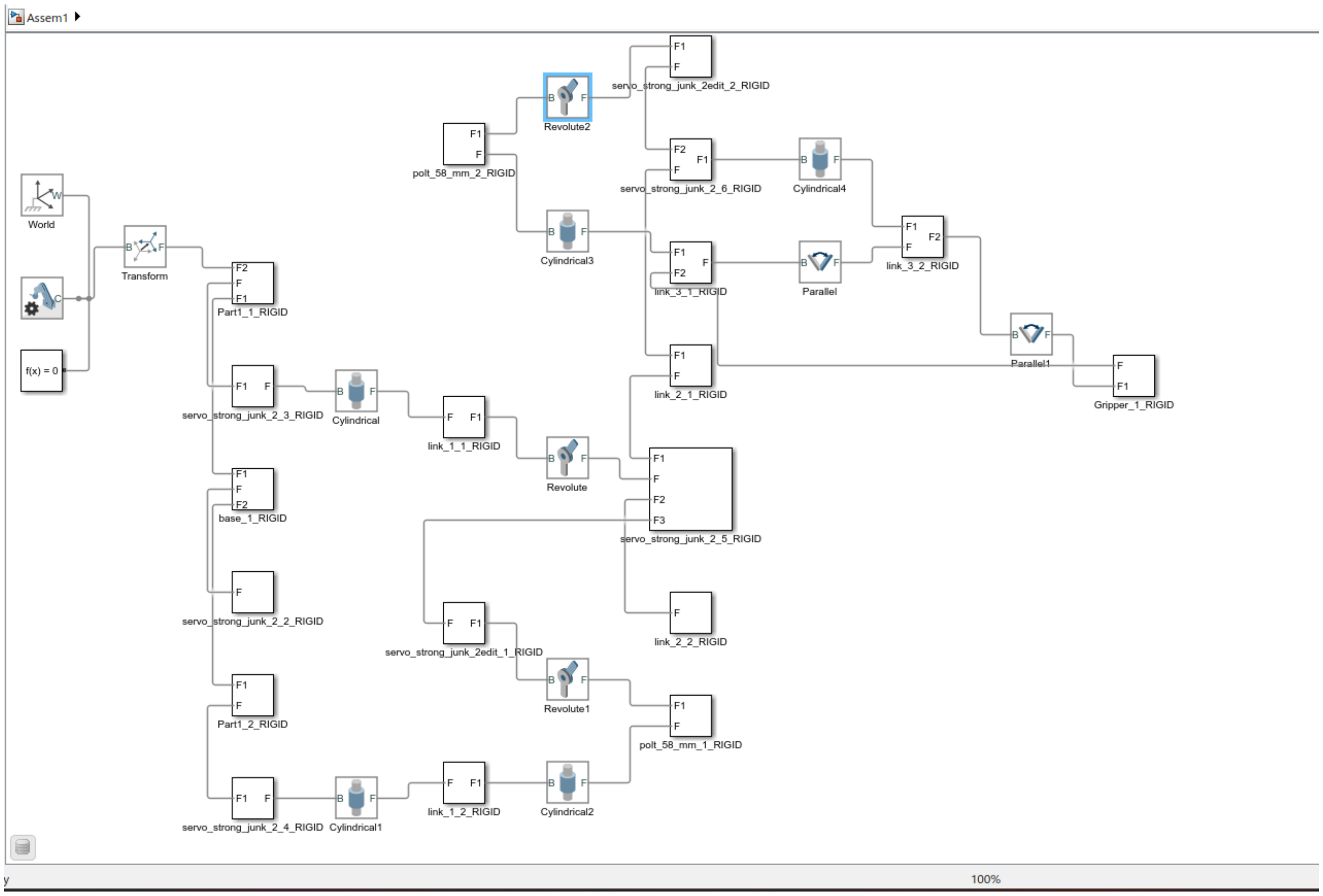
$L_1 = 5.4$ cm
$L_2 = 12.02$ cm
$L_3 = 12.2$ cm

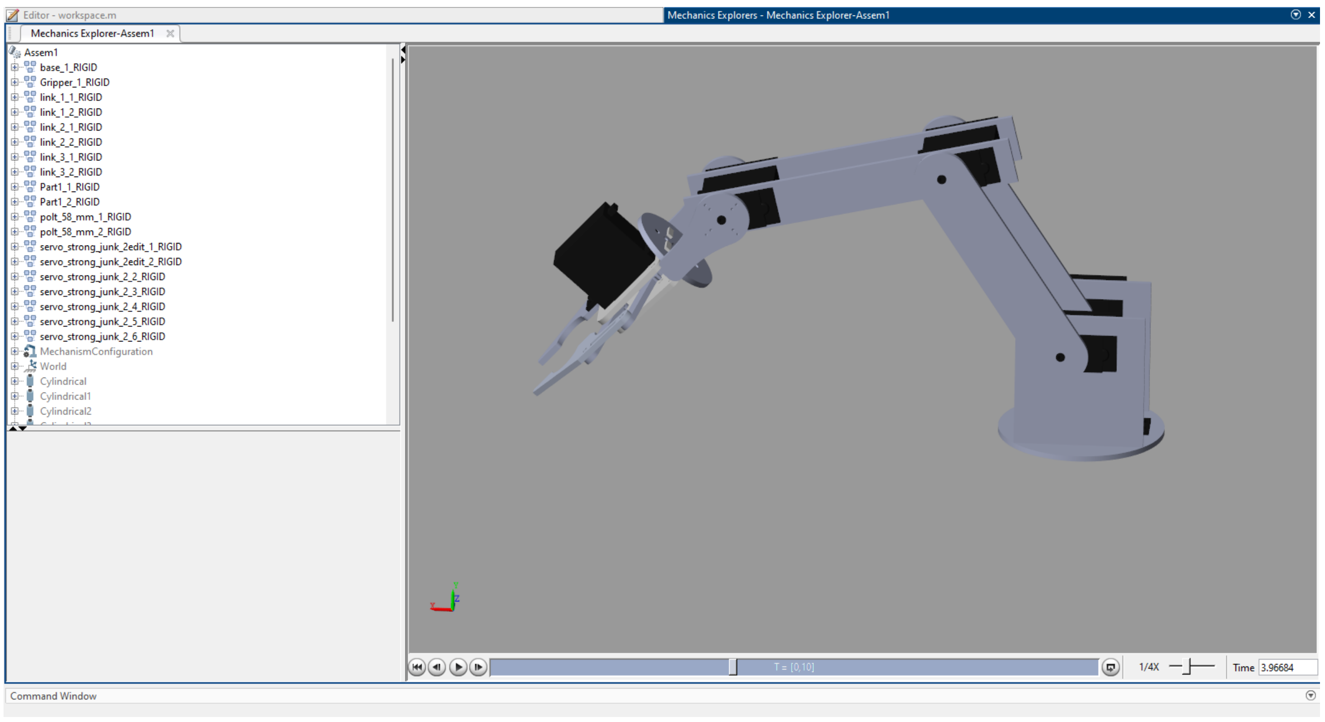
α: angle about x_i from z_{i-1} to z_i
a: distance between $(i-1)$ and (i) in x_i - direction
d: distance between $(i-1)$ and (i) in Z_{i-1} - direction
θ: angle of rot. about z_{i-1} from x_{i-1} to x_i

DH_Table:

frame	α	a	d	θ
1	-90	0	L_1	θ_1
2	0	L_2	0	$-90 + \theta_2$
3	0	L_3	0	$90 + \theta_3$
4	0	0	0	θ_4

➢ Simscape Model:





➢ Workspace:

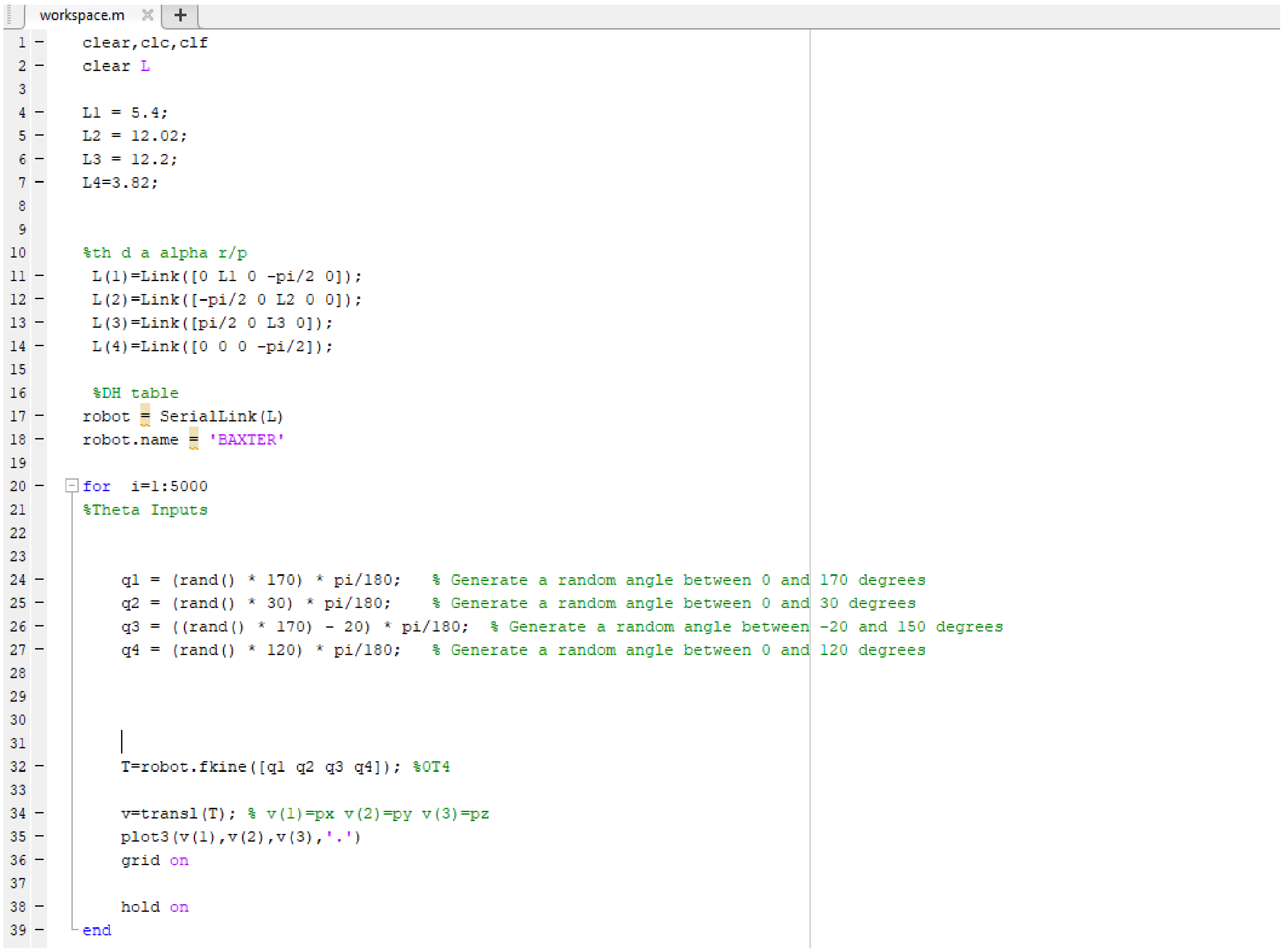
- Angle of rotation of each link Approximately.

  Link1: 170 degrees.

  Link2: 30 degrees.

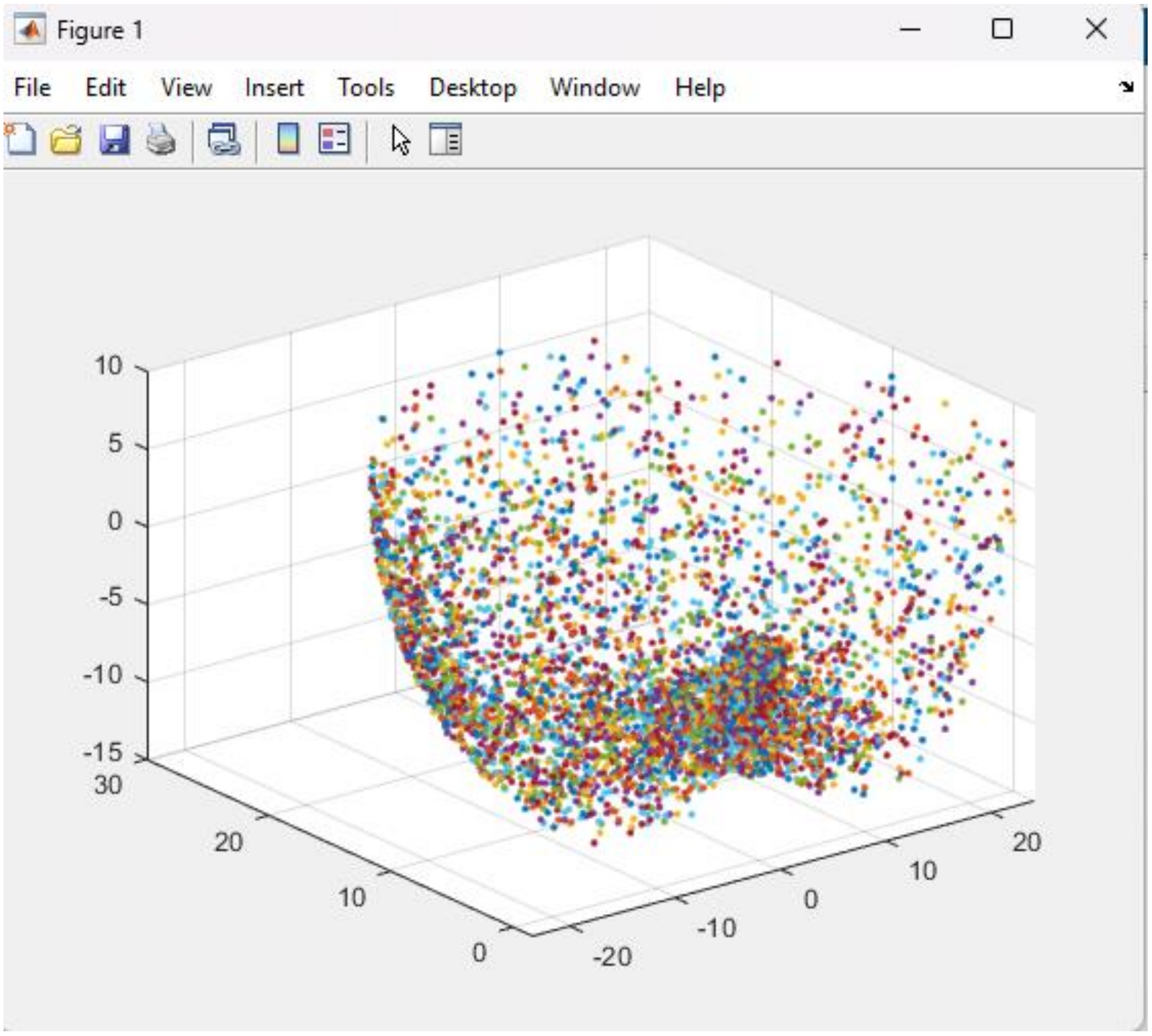
  Link3: 170 degrees.

  Link4: 120 degrees.



workspace.m

```matlab
clear,clc,clf
clear L

L1 = 5.4;
L2 = 12.02;
L3 = 12.2;
L4=3.82;


%th d a alpha r/p
 L(1)=Link([0 L1 0 -pi/2 0]);
 L(2)=Link([-pi/2 0 L2 0 0]);
 L(3)=Link([pi/2 0 L3 0]);
 L(4)=Link([0 0 0 -pi/2]);

 %DH table
robot = SerialLink(L)
robot.name = 'BAXTER'

for  i=1:5000
%Theta Inputs


    q1 = (rand() * 170) * pi/180;   % Generate a random angle between 0 and 170 degrees
    q2 = (rand() * 30) * pi/180;    % Generate a random angle between 0 and 30 degrees
    q3 = ((rand() * 170) - 20) * pi/180;  % Generate a random angle between -20 and 150 degrees
    q4 = (rand() * 120) * pi/180;   % Generate a random angle between 0 and 120 degrees




    T=robot.fkine([q1 q2 q3 q4]); %0T4

    v=transl(T); % v(1)=px v(2)=py v(3)=pz
    plot3(v(1),v(2),v(3),'.')
    grid on

    hold on
end
```



If we want to draw a specific position, we can add this line:

```
%robot.plot([q1 q2 q3 q4]);
```

And substitute with the desired q value.

For example:

If we want:

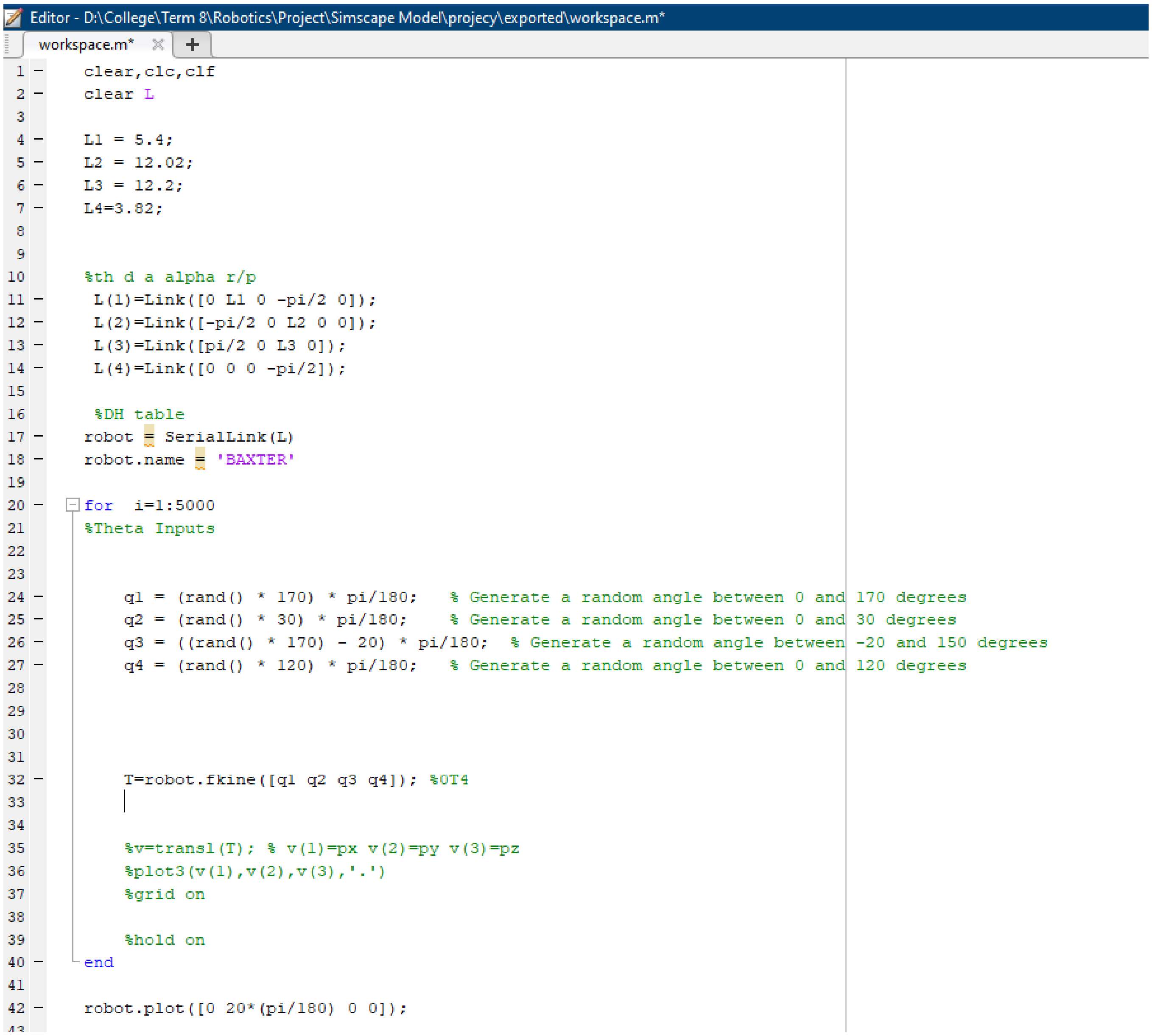
$q1 = 0,$

$q2 = 20*(pi/180),$

$q3 = 0,$

$q4 = 0$

- The Code:

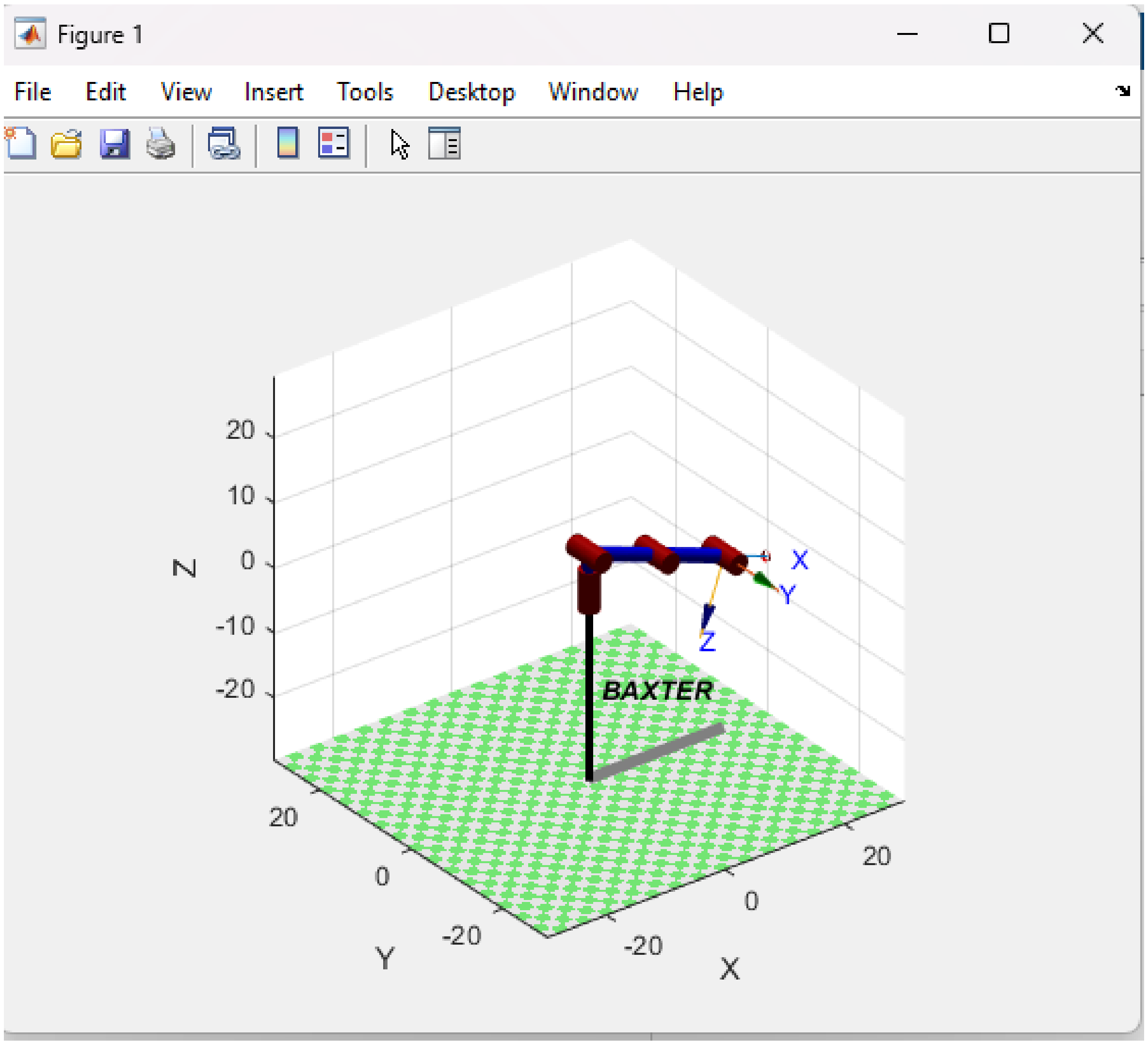


Editor - D:\College\Term 8\Robotics\Project\Simscape Model\projecy\exported\workspace.m*

workspace.m*

```matlab
clear,clc,clf
clear L

L1 = 5.4;
L2 = 12.02;
L3 = 12.2;
L4=3.82;


%th d a alpha r/p
 L(1)=Link([0 L1 0 -pi/2 0]);
 L(2)=Link([-pi/2 0 L2 0 0]);
 L(3)=Link([pi/2 0 L3 0]);
 L(4)=Link([0 0 0 -pi/2]);

 %DH table
robot = SerialLink(L)
robot.name = 'BAXTER'

for  i=1:5000
%Theta Inputs


    q1 = (rand() * 170) * pi/180;    % Generate a random angle between 0 and 170 degrees
    q2 = (rand() * 30) * pi/180;     % Generate a random angle between 0 and 30 degrees
    q3 = ((rand() * 170) - 20) * pi/180;  % Generate a random angle between -20 and 150 degrees
    q4 = (rand() * 120) * pi/180;    % Generate a random angle between 0 and 120 degrees




    T=robot.fkine([q1 q2 q3 q4]); %0T4


    %v=transl(T); % v(1)=px v(2)=py v(3)=pz
    %plot3(v(1),v(2),v(3),'.')
    %grid on

    %hold on
end

robot.plot([0 20*(pi/180) 0 0]);
```

- Robot position:



- The output DH_Table from the code:

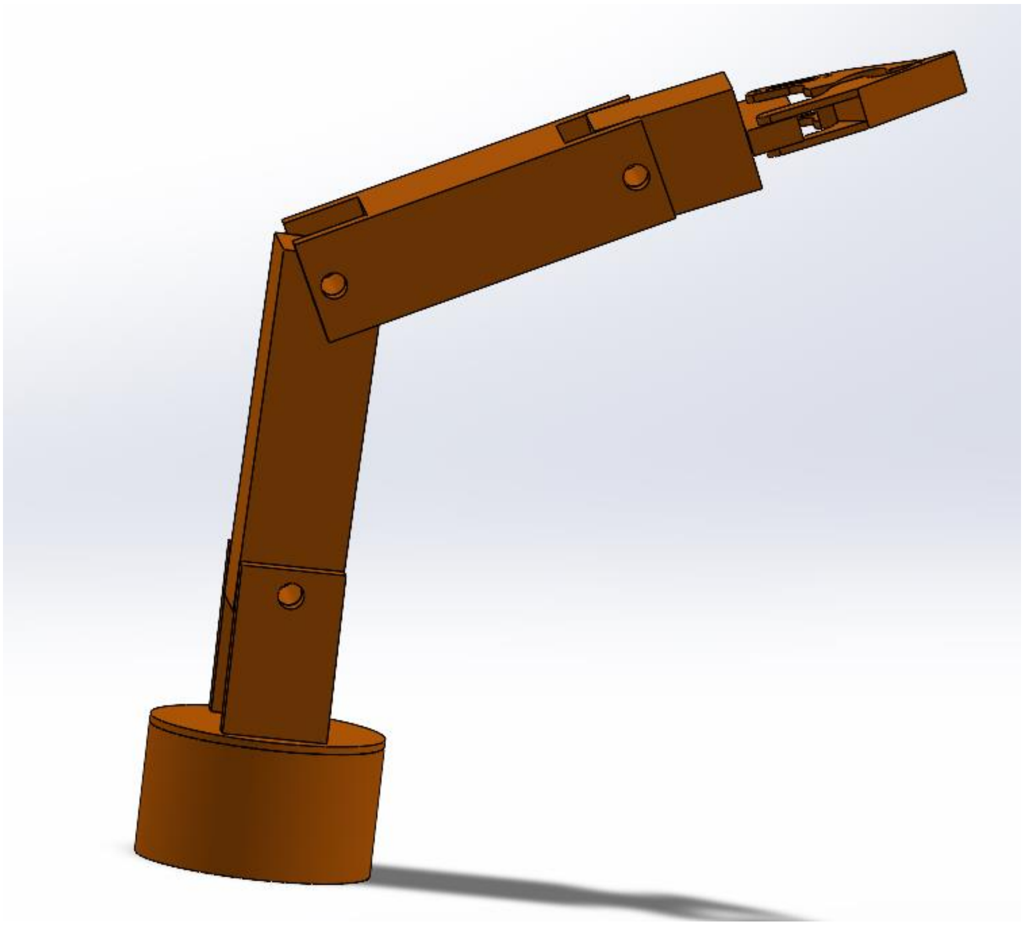
```
robot =

BAXTER:: 4 axis, RRRR, stdDH, slowRNE
+---+-----------+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |    offset |
+---+-----------+-----------+-----------+-----------+-----------+
|  1|         q1|        5.4|          0|    -1.5708|          0|
|  2|         q2|          0|      12.02|          0|          0|
|  3|         q3|          0|       12.2|          0|          0|
|  4|         q4|          0|          0|    -1.5708|          0|
+---+-----------+-----------+-----------+-----------+-----------+
```

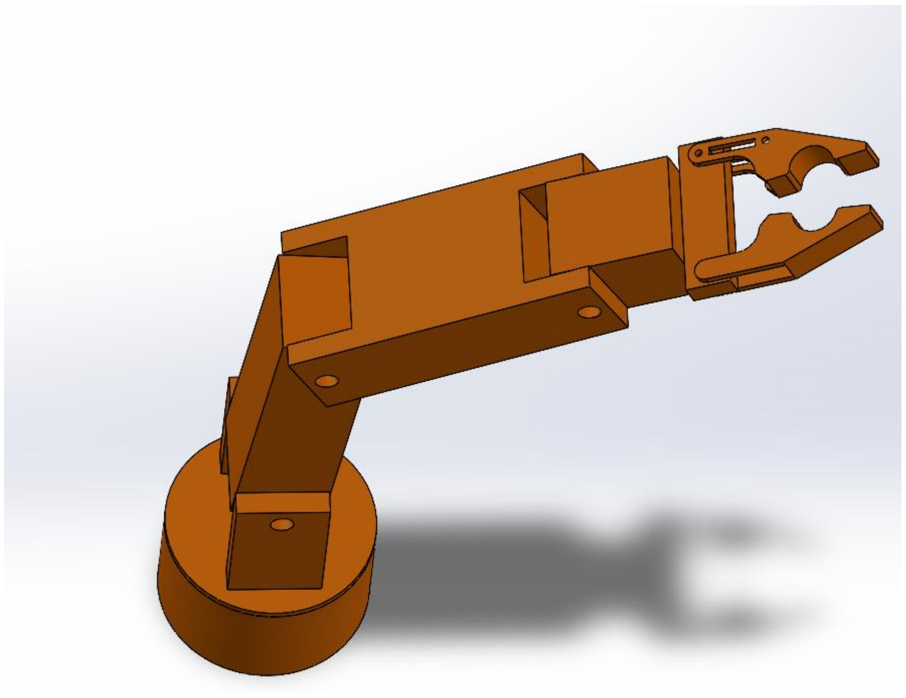
The previous robot design was complex and have a lot of errors when trying to work on MATLAB due to servo motors.

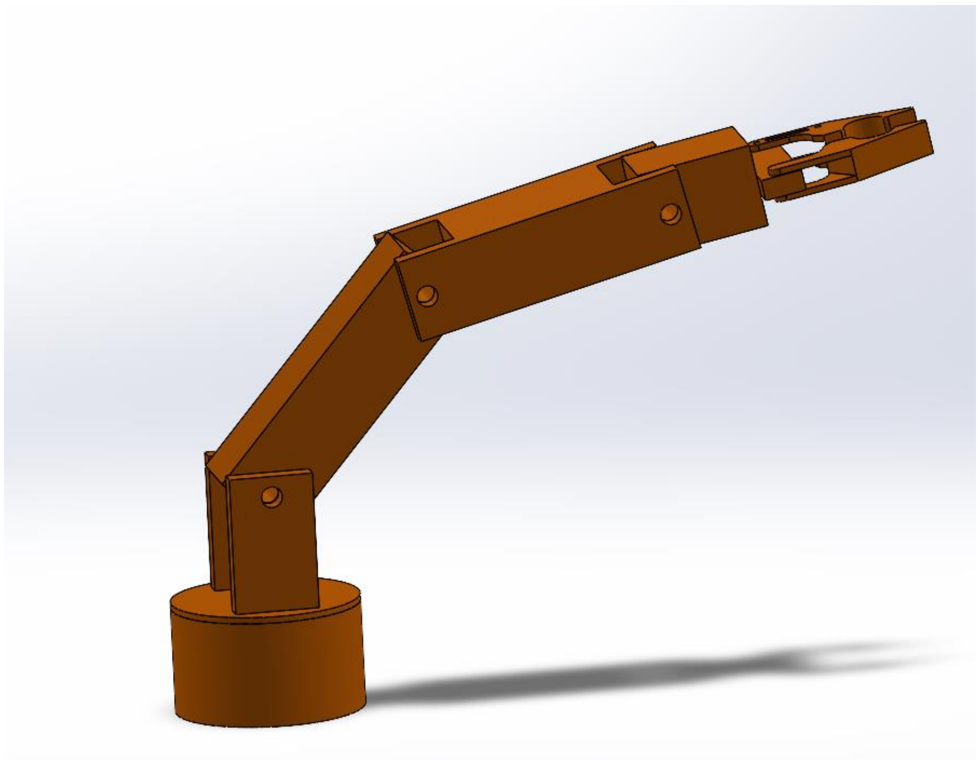
And another problem is that MATLAB see the parallel links as more than one link, so it is not considered as one link.

Therefore, we had to modify the SolidWorks design to a simple model with rigid links.

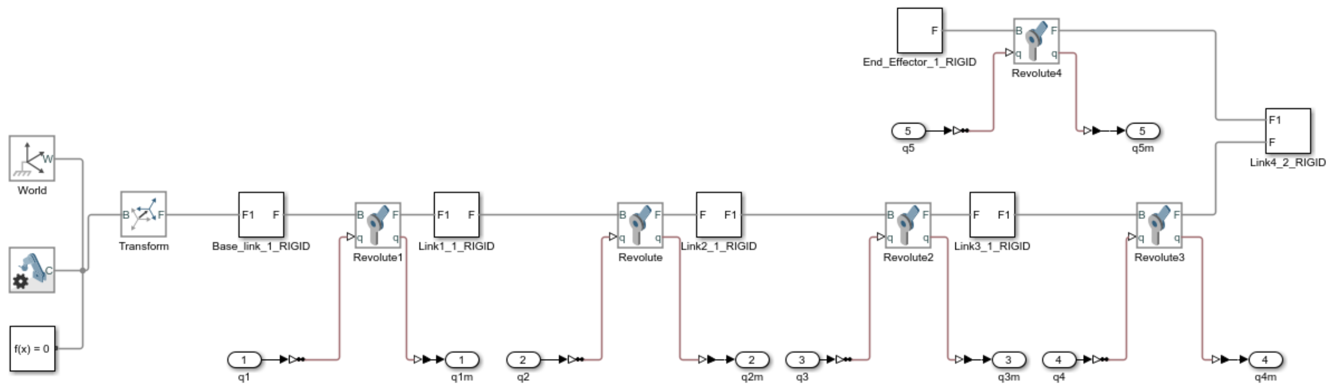
➢ New SolidWorks Design:

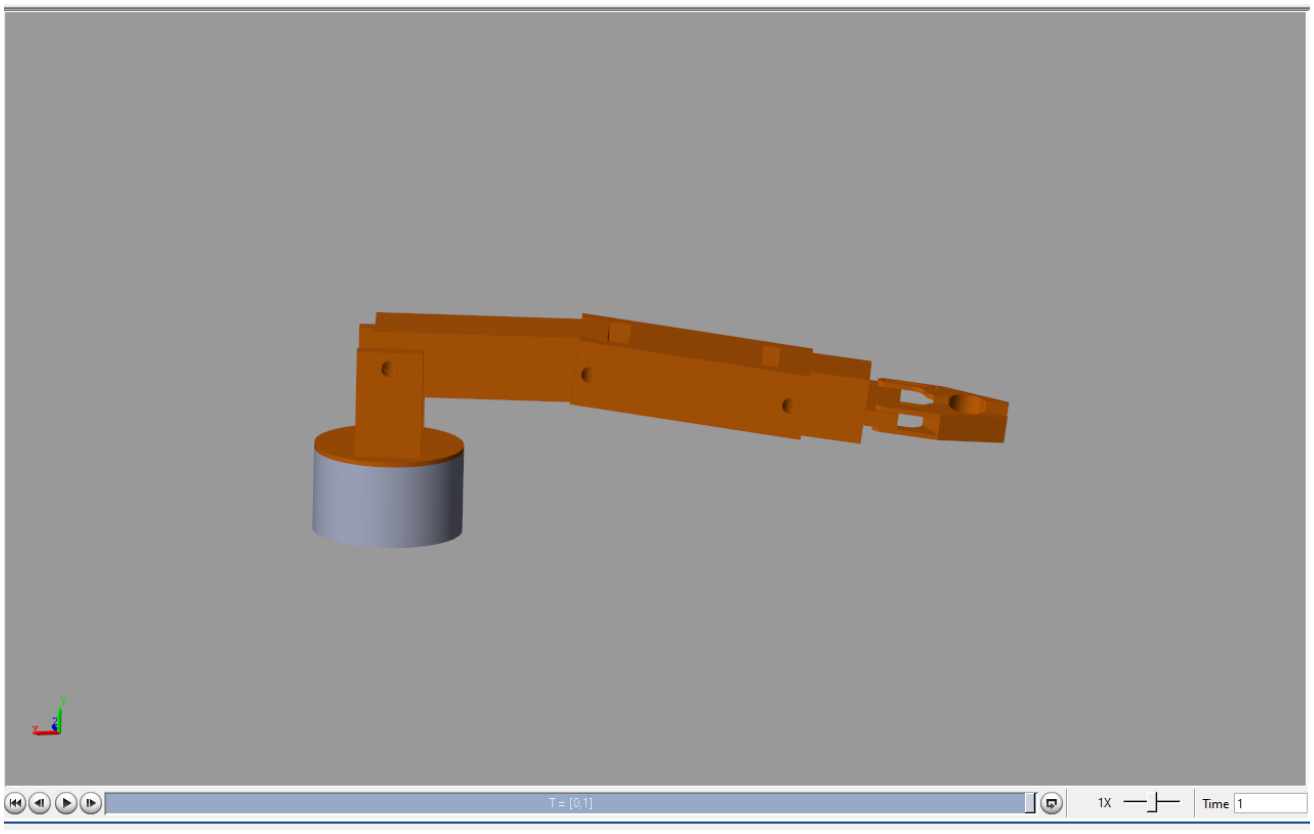






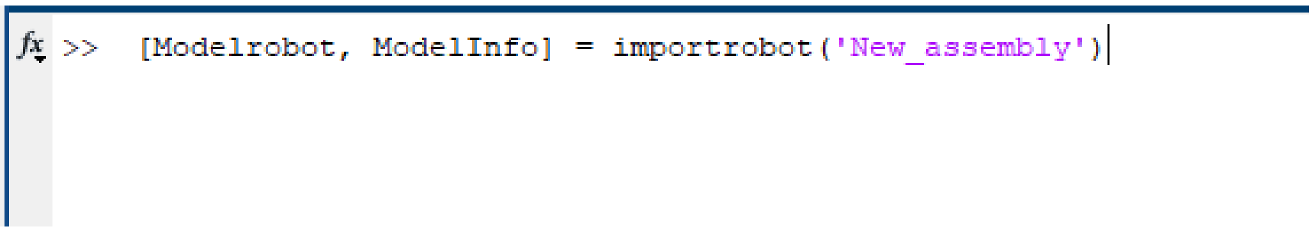
➢ New Simscape Model:



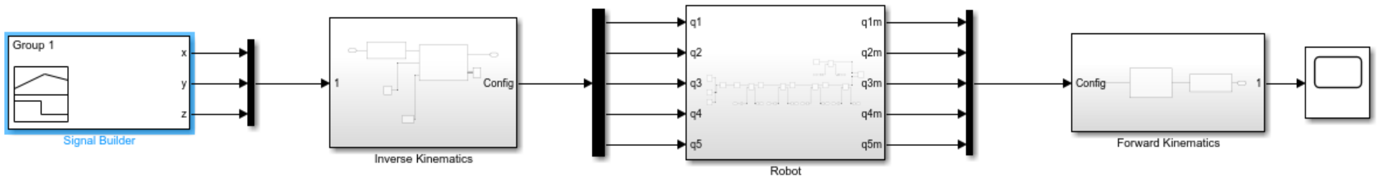


➢ For the Robot to follow a specific trajectory:

First, we must make the rigid bodytree variable for the robot using the following command:



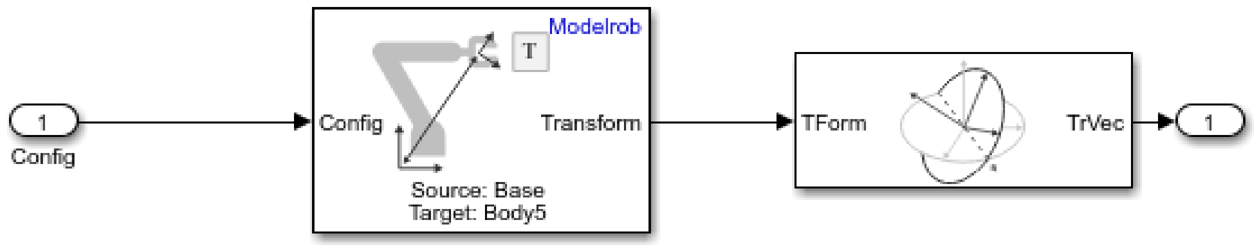
```
>>   [Modelrobot, ModelInfo] = importrobot('New_assembly')
```



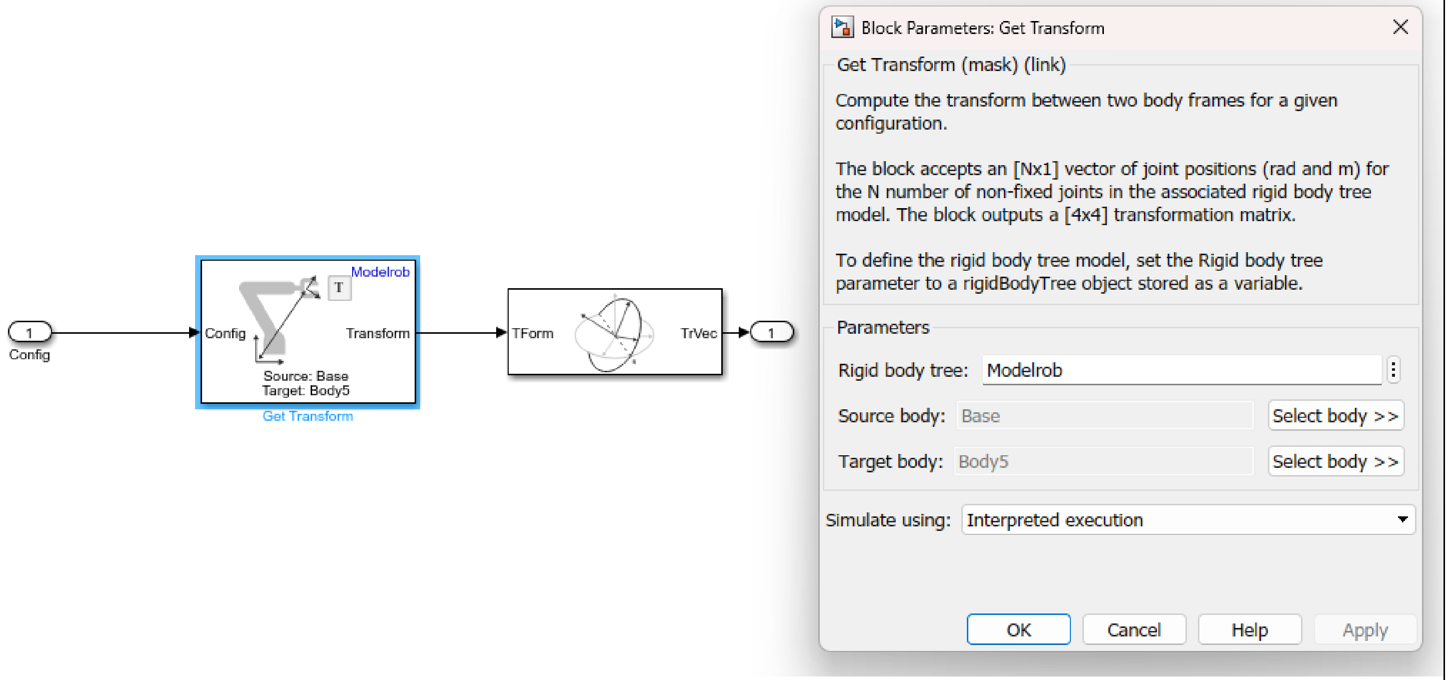
We generated a signal (X, Y, Z) using signal builder, the positions X, Y, Z output of signal builder is the input to the inverse kinematics block. This block gives us the angles of the joints passed to the robot.

The sensed position from robot passed to the forward kinematics block to give back the position of end effector and compare it with the signal we made.

Forward Kinematics block:



We must specify the rigid body tree, Base, and End effector.



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

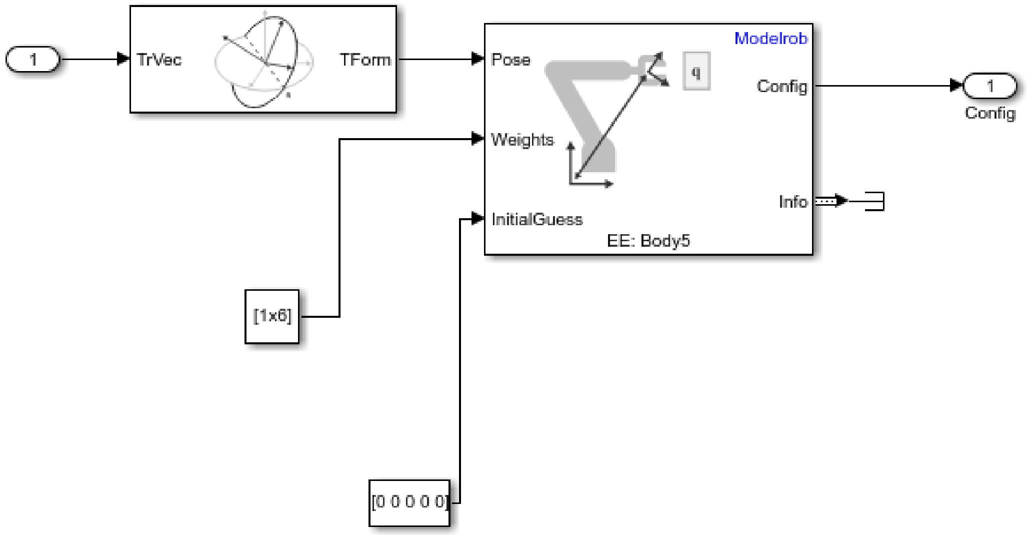
Rigid body tree: Modelrob

Source body: Base

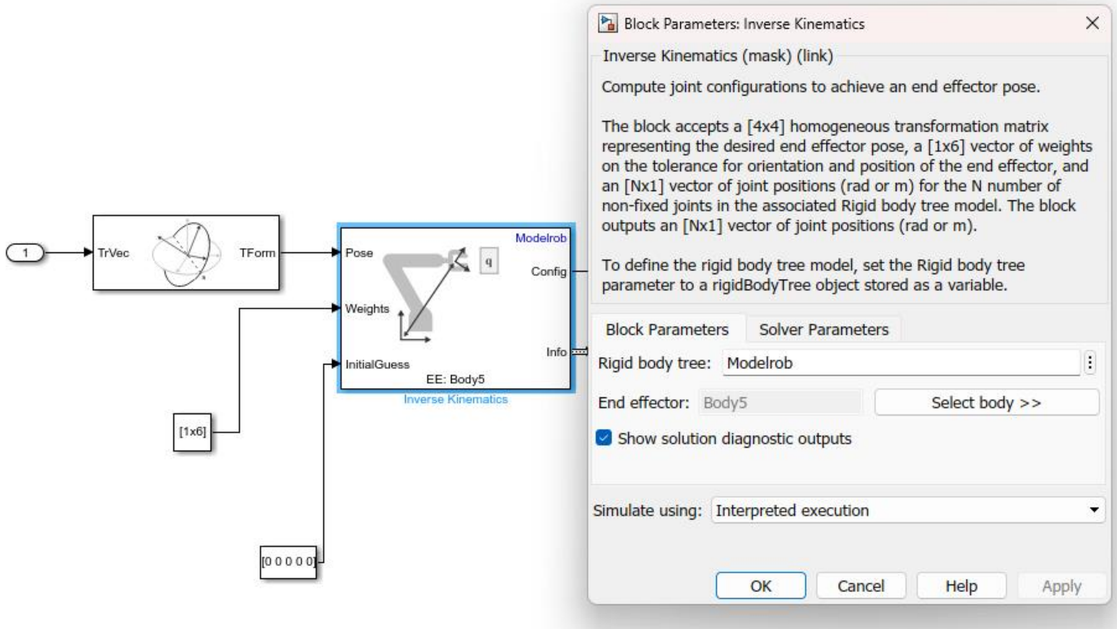
Target body: Body5

Simulate using: Interpreted execution

Inverse Kinematics Block:



Here we also must specify the rigid body tree and the end effector only.



Inverse Kinematics (mask) (link)

Compute joint configurations to achieve an end effector pose.

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.

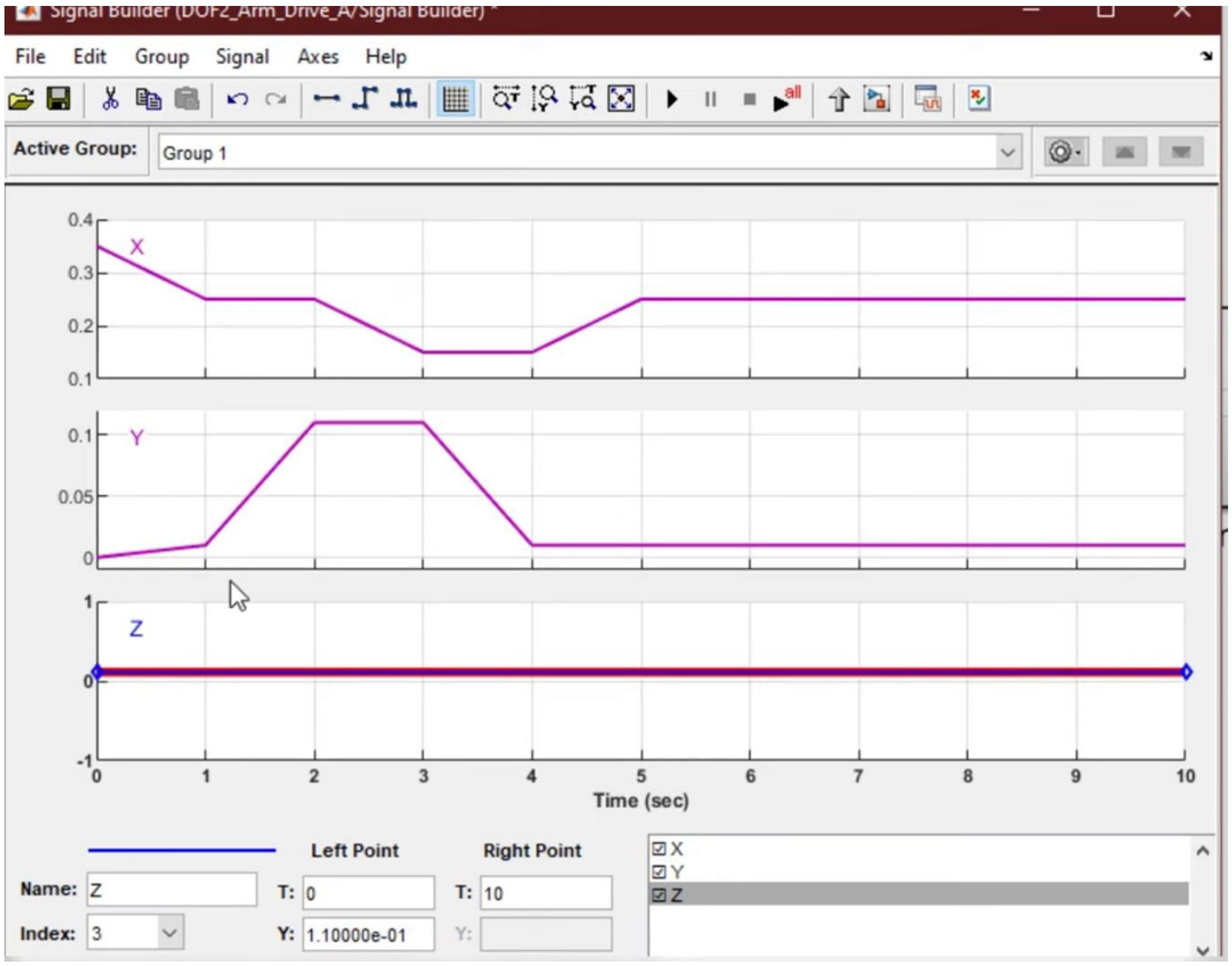
Rigid body tree: Modelrob

End effector: Body5

[✓] Show solution diagnostic outputs

Simulate using: Interpreted execution

Signal Builder:


Signal Builder (DOF2_Arm_Drive_A/Signal Builder) *
File Edit Group Signal Axes Help
X
Y
Z
Time (sec)
Left Point
Right Point
☑ X
☑ Y
☑ Z

Active Group: Group 1

Name: Z

Index: 3

Left Point T: 0

Left Point Y: 1.10000e-01

Right Point T: 10

Instead of making a random signal, there is several trajectory blocks in MATLAB robotic toolbox:

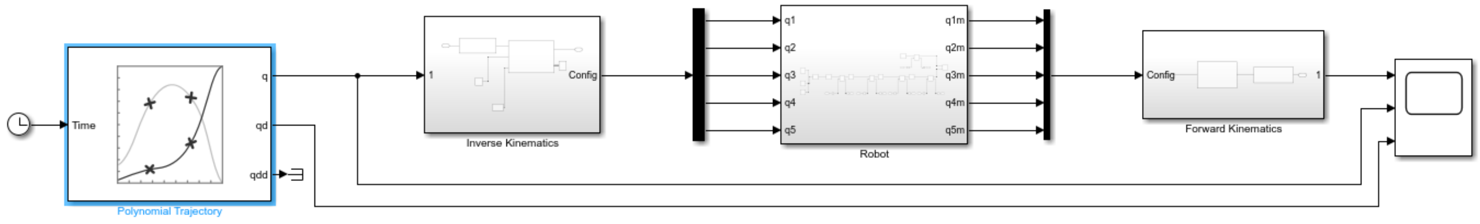
1. Polynomial trajectory:

At start we generated trajectory,providing that trajectory to the inverse kinematics block.

Inverse kinematic block converts that trajectory into joint angles which is provided to the robot.

Then measuring the joint angles of the robot giving those joint angles to the forward kinematics block, the forward kinematics block convert these angles back to the end effector positions.

Therefore, we are comparing the trajectory on which the robot was moving with the trajectory that we were providing using the scope.

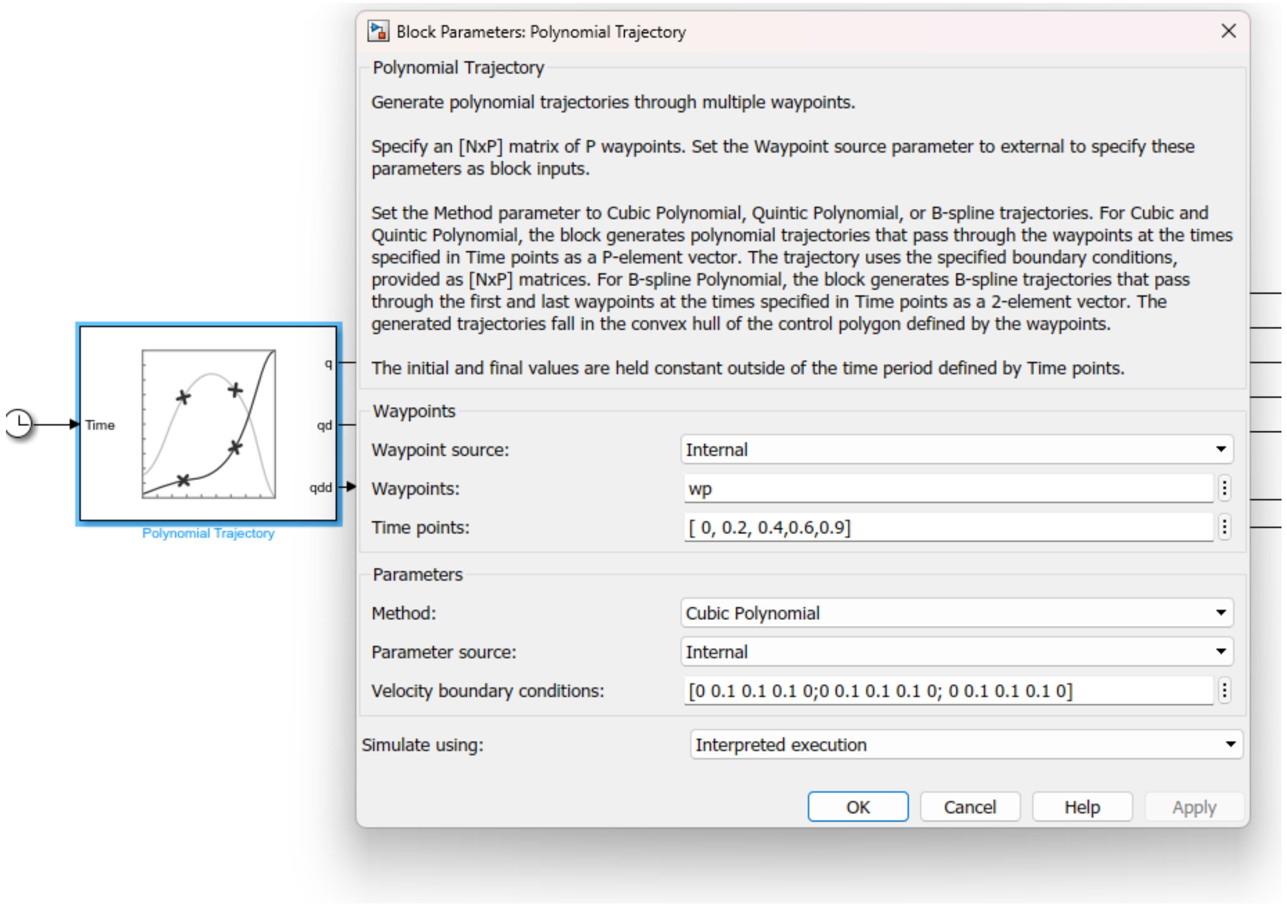


We used cubic polynomial.

For this type we had to give it waypoints that the robot must follow, and time taken for the robot to reach each waypoint.

Here are the waypoints:

```
>> wp=[0.351,0.4,0,-0.4,-0.351; 0.109,0.2,0.351,0.2,0.109; 0,0.2,0.1,0.2,0]

wp =

    0.3510    0.4000         0   -0.4000   -0.3510
    0.1090    0.2000    0.3510    0.2000    0.1090
         0    0.2000    0.1000    0.2000         0

>> save('wp.mat')
```



Block Parameters: Polynomial Trajectory

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: wp

Time points: [0, 0.2, 0.4,0.6,0.9]

Parameters

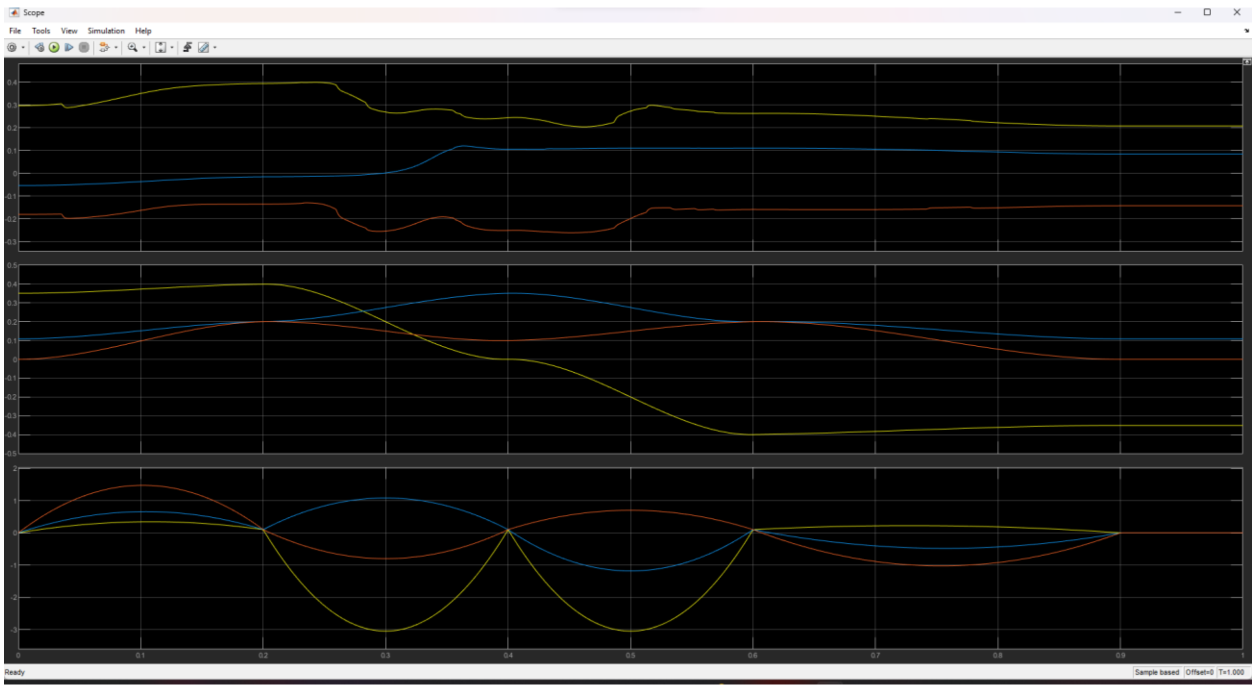
Method: Cubic Polynomial

Parameter source: Internal

Velocity boundary conditions: [0 0.1 0.1 0.1 0;0 0.1 0.1 0.1 0; 0 0.1 0.1 0.1 0]

Simulate using: Interpreted execution

Scope output:



The first graph is the output X, Y, Z from the forward kinematics.

The second graph is the output from our trajectory.

And the third graph shows the velocities in X, Y, Z.

As we can see here there is error between the position from trajectory and forward kinematics, this error is handled by a controller (PID) to eliminate the error.

2. Trapezoidal velocity profile:

Used if robot moves in linear motion, want to transverse in straight line.

- Move with constant acceleration.
- Achieve a certain velocity.
- Move with that constant velocity for some time.
- Then decelerates again.

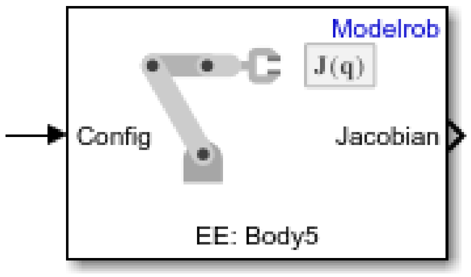
We need to do the same comparison of the position to the velocities; therefore, we will use Jacobian.

Jacobian:

Using the get Jacobian block from robotic toolbox:

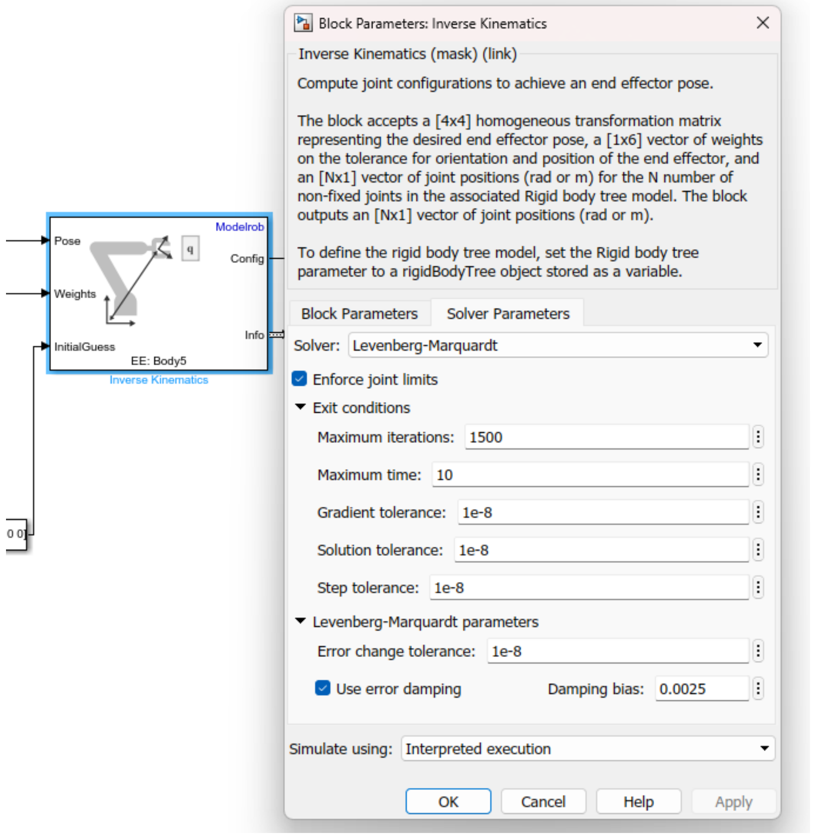
This Jacobian matrix will be 6*n matrix.

6 is for the 6 velocities we are going to get; the first three velocities are the angular velocities, and the last three velocities will be the linear velocities.



The columns of this Jacobian matrix will correspond to the number of joints we have.

There is a huge error in the values due to the error coming from the inverse kinematics block, therefore, to reduce this error by decrease the tolerance.



Block Parameters: Inverse Kinematics

Inverse Kinematics (mask) (link)

Compute joint configurations to achieve an end effector pose.

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.

Block Parameters | Solver Parameters

Solver: Levenberg-Marquardt

☑ Enforce joint limits

Exit conditions

Maximum iterations: 1500

Maximum time: 10

Gradient tolerance: 1e-8

Solution tolerance: 1e-8

Step tolerance: 1e-8

Levenberg-Marquardt parameters

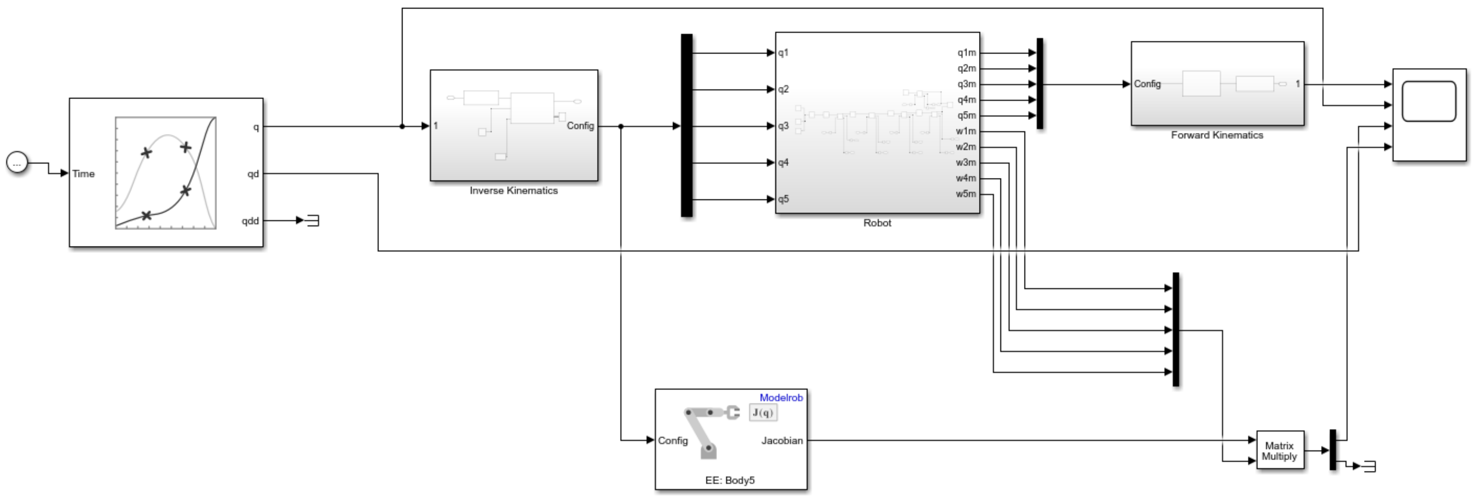
Error change tolerance: 1e-8

☑ Use error damping Damping bias: 0.0025

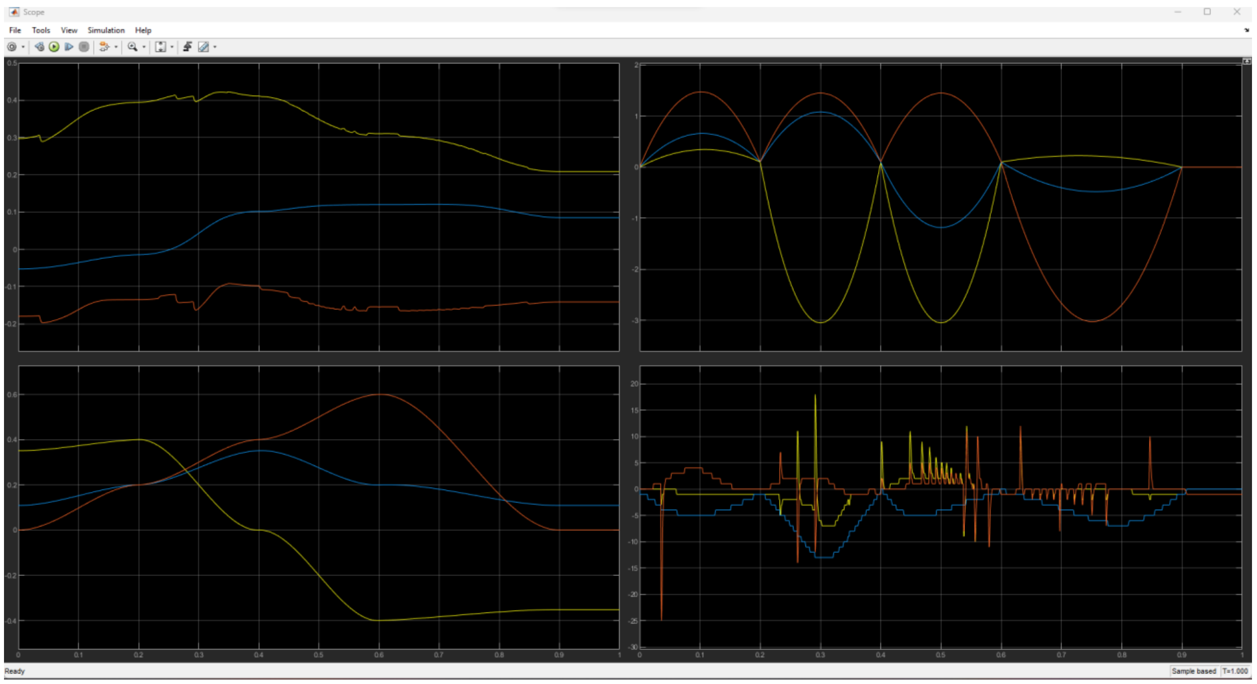
Simulate using: Interpreted execution

OK Cancel Help Apply

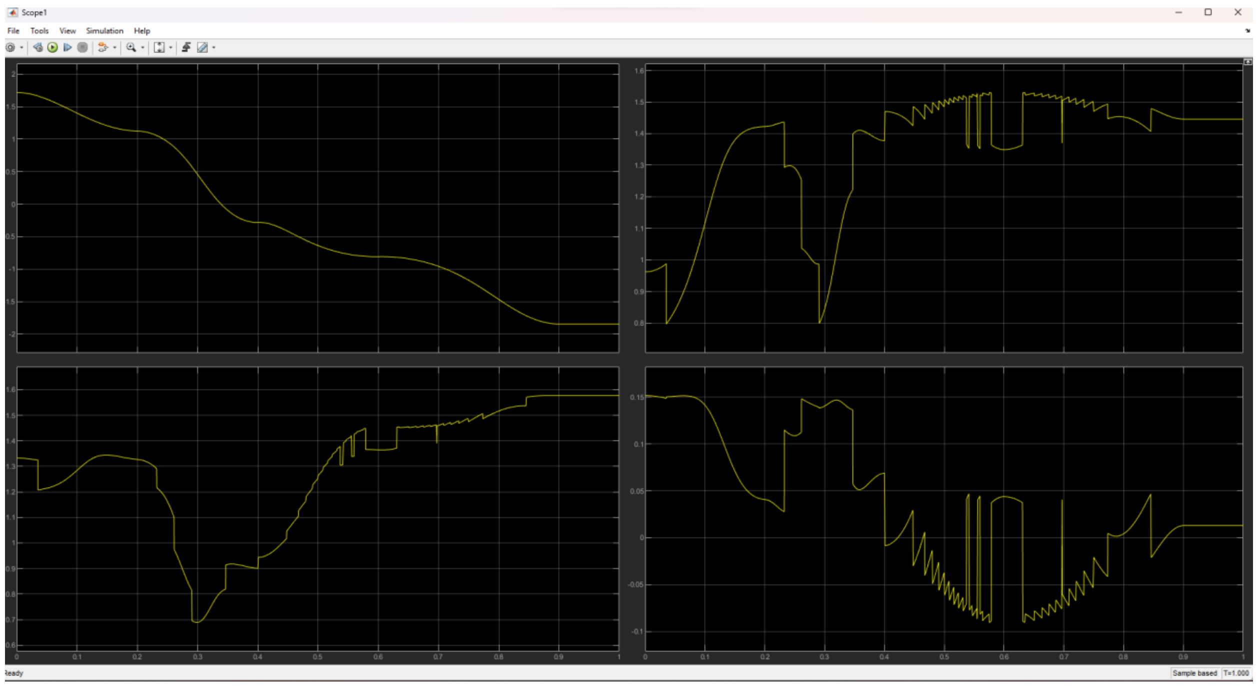
For velocity comparison:



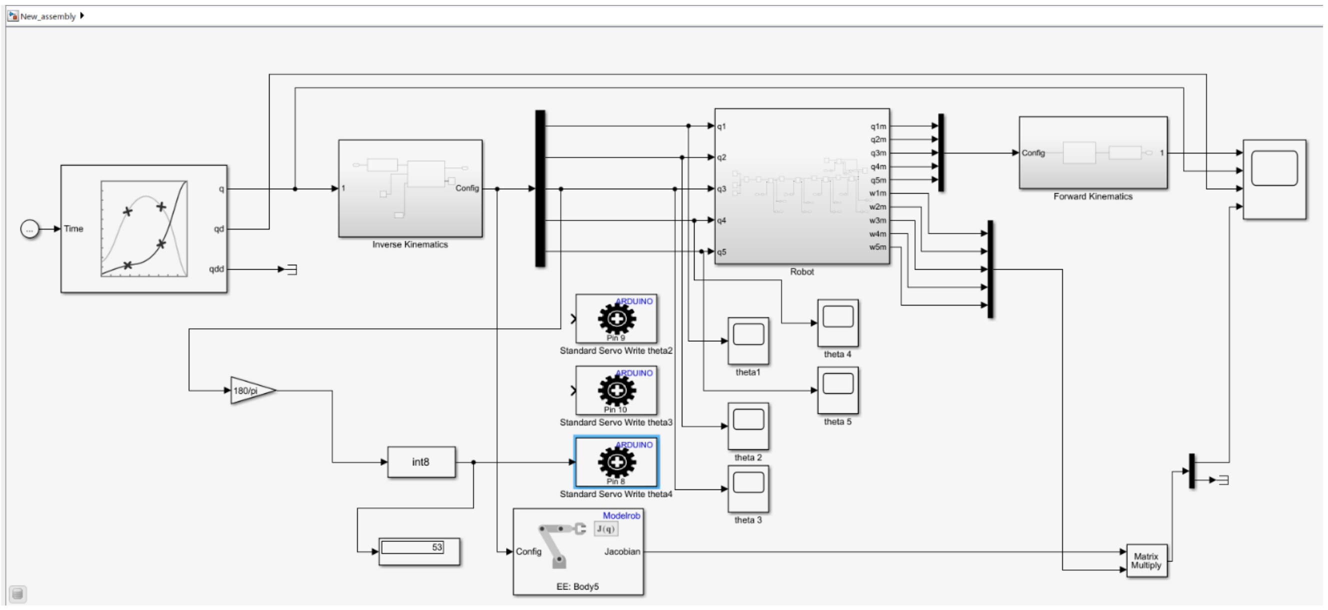
Scope output:



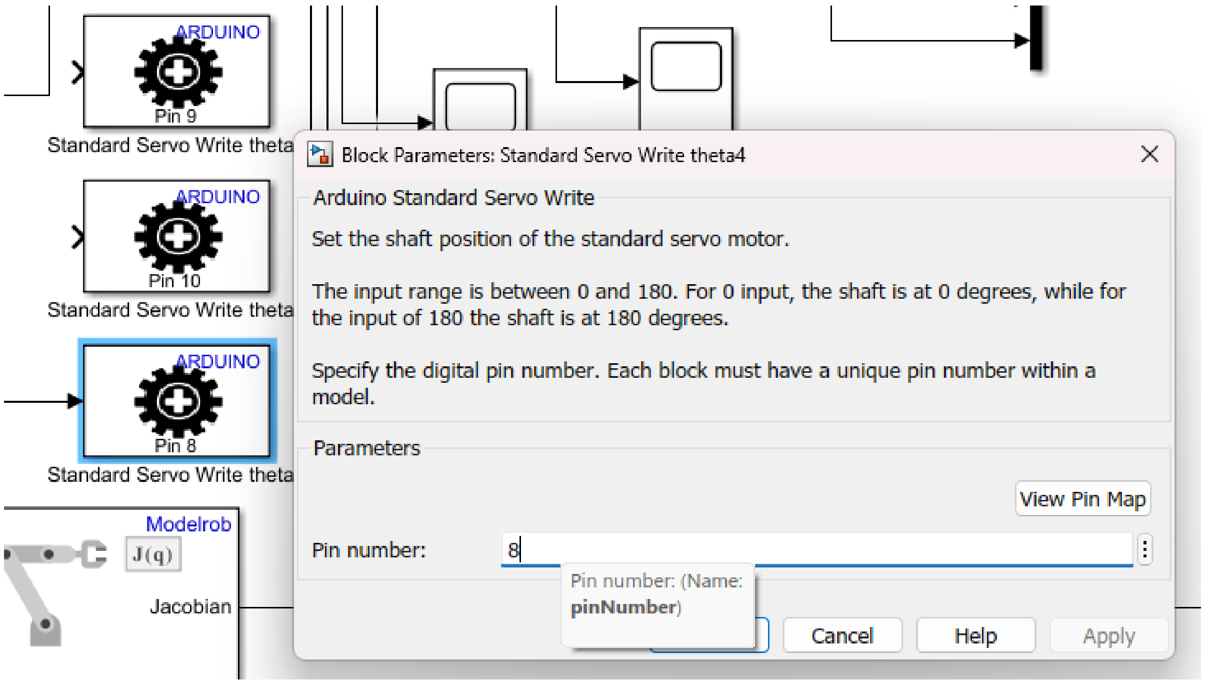
Visualizing the angles from inverse kinematics:



To take the output angles from the inverse kinematics block and send the angles to the hardware Arduino package is used in MATLAB Simulink.



Servo write is used to send the output on the servo motor in the hardware by defining the pin of the signal.



Block Parameters: Standard Servo Write theta4

Arduino Standard Servo Write

Set the shaft position of the standard servo motor.

The input range is between 0 and 180. For 0 input, the shaft is at 0 degrees, while for the input of 180 the shaft is at 180 degrees.

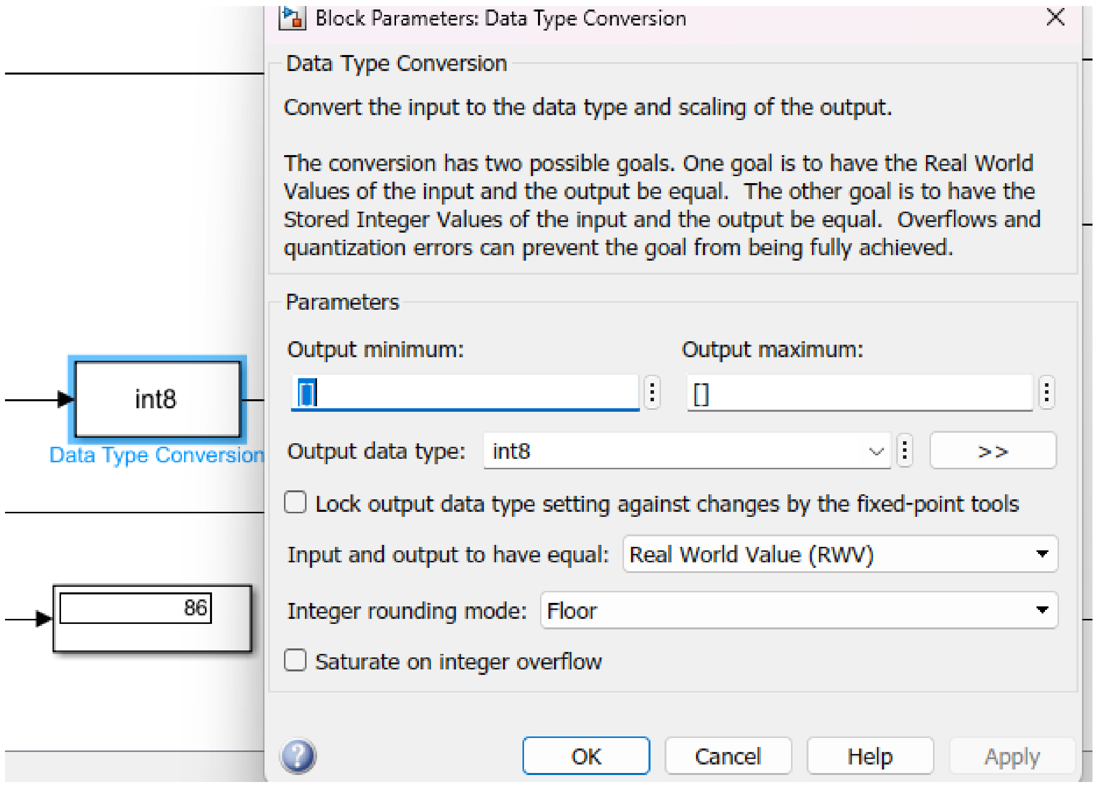
Specify the digital pin number. Each block must have a unique pin number within a model.

Parameters

Pin number: 8

Pin number: (Name: **pinNumber**)

We used Data Type Conversion Block to round the output angle an send it as an integer, as the servo write block only take integer values.



Block Parameters: Data Type Conversion

Data Type Conversion

Convert the input to the data type and scaling of the output.

The conversion has two possible goals. One goal is to have the Real World Values of the input and the output be equal. The other goal is to have the Stored Integer Values of the input and the output be equal. Overflows and quantization errors can prevent the goal from being fully achieved.

Parameters

Output minimum: []

Output maximum: []

Output data type: int8

☐ Lock output data type setting against changes by the fixed-point tools

Input and output to have equal: Real World Value (RWV)

Integer rounding mode: Floor

☐ Saturate on integer overflow

Using ROS

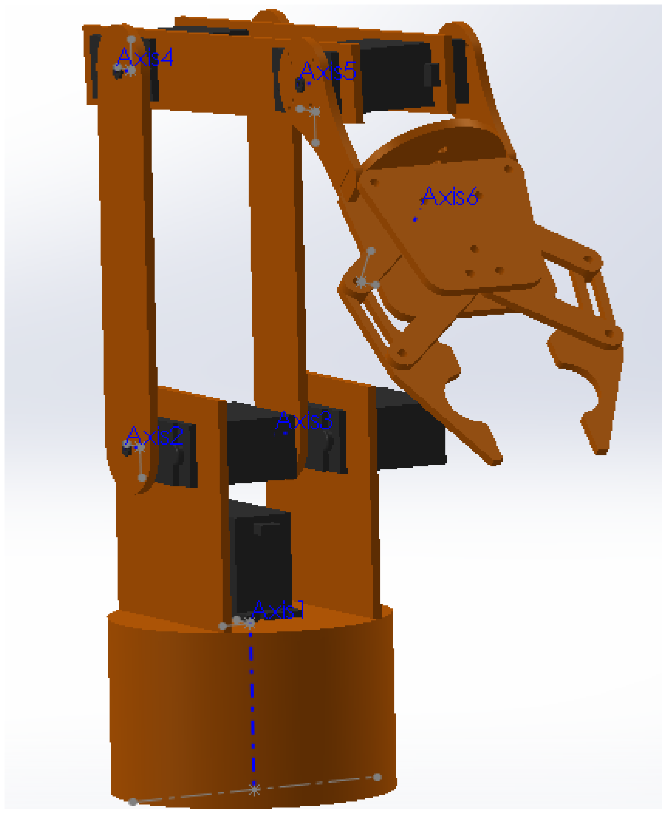
For the controlling of the robot arm we have used Moveit.

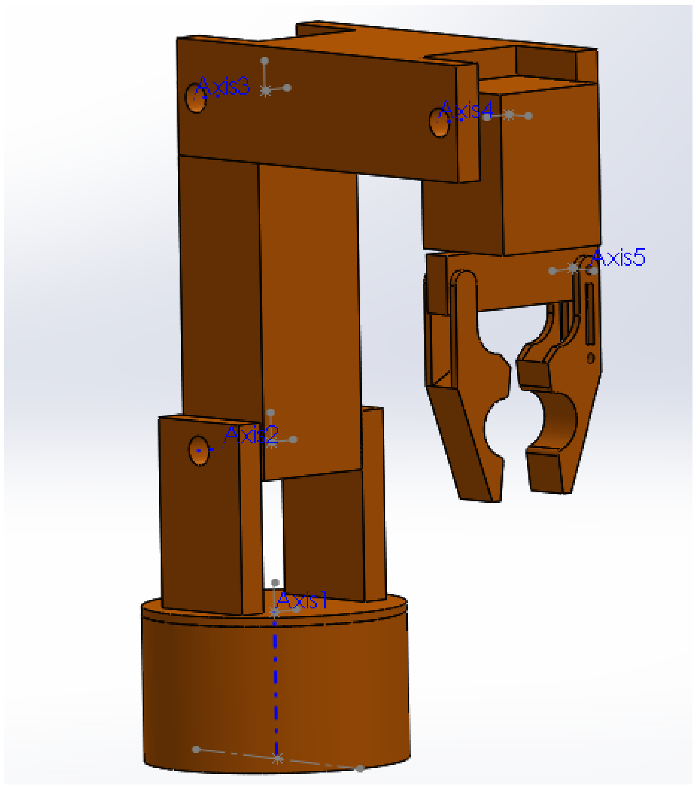
MoveIt builds on the ROS messaging and build systems and utilizes some of the common tools in ROS like the ROS Visualizer (Rviz) and the ROS robot format (URDF).

Moveit runs on the top of ROS (robot operating system).

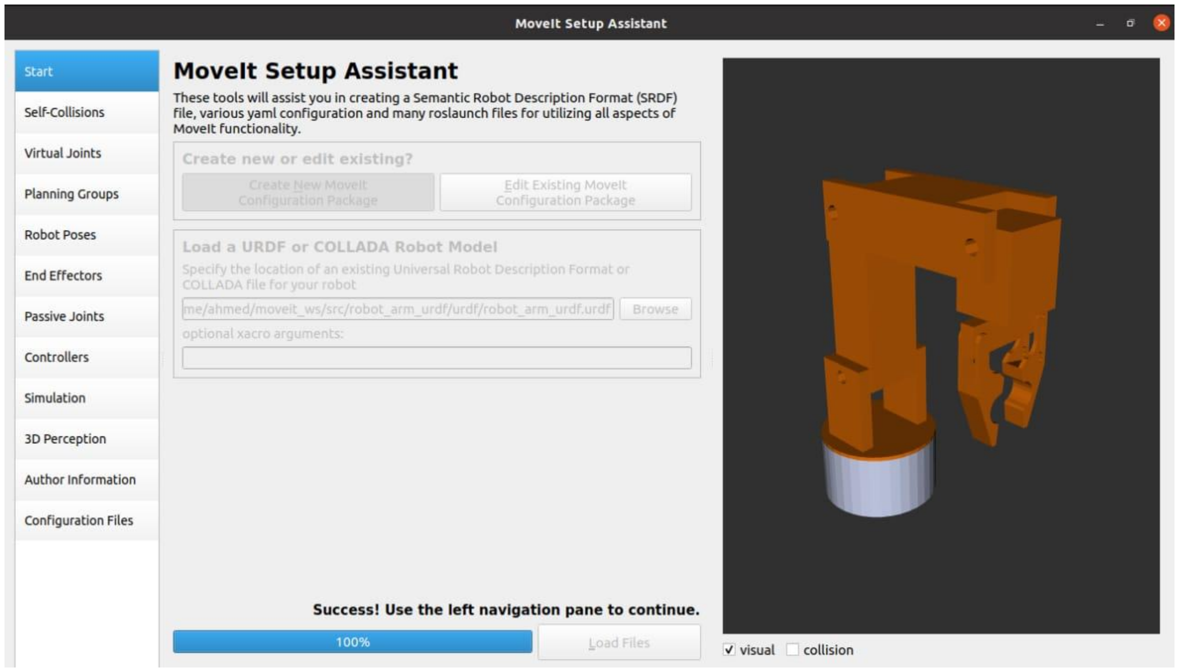
ROS is an open-source meta-operating system for robots that provides low-level functionality like a build system, message passing, device drivers and some integrated capabilities like navigation.

First we simplify the design of the real robotic arm to make the URDF exportation easier.





Then after configuring the URDF file , we imported it in Moveit setup assistant and we defined specific planning groups and poses for the robotic arm to follow.



MoveIt Setup Assistant

These tools will assist you in creating a Semantic Robot Description Format (SRDF) file, various yaml configuration and many roslaunch files for utilizing all aspects of MoveIt functionality.

Create new or edit existing?

Load a URDF or COLLADA Robot Model

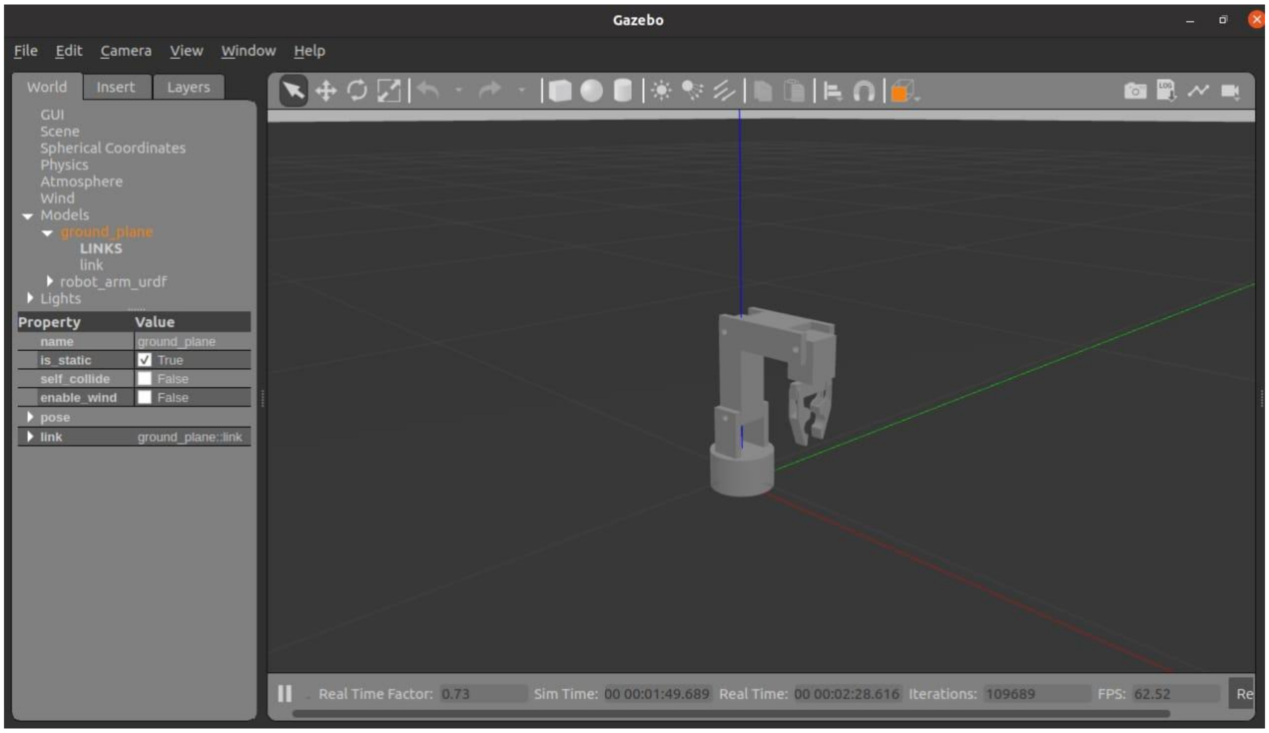
Specify the location of an existing Universal Robot Description Format or COLLADA file for your robot

me/ahmed/moveit_ws/src/robot_arm_urdf/urdf/robot_arm_urdf.urdf

optional xacro arguments:

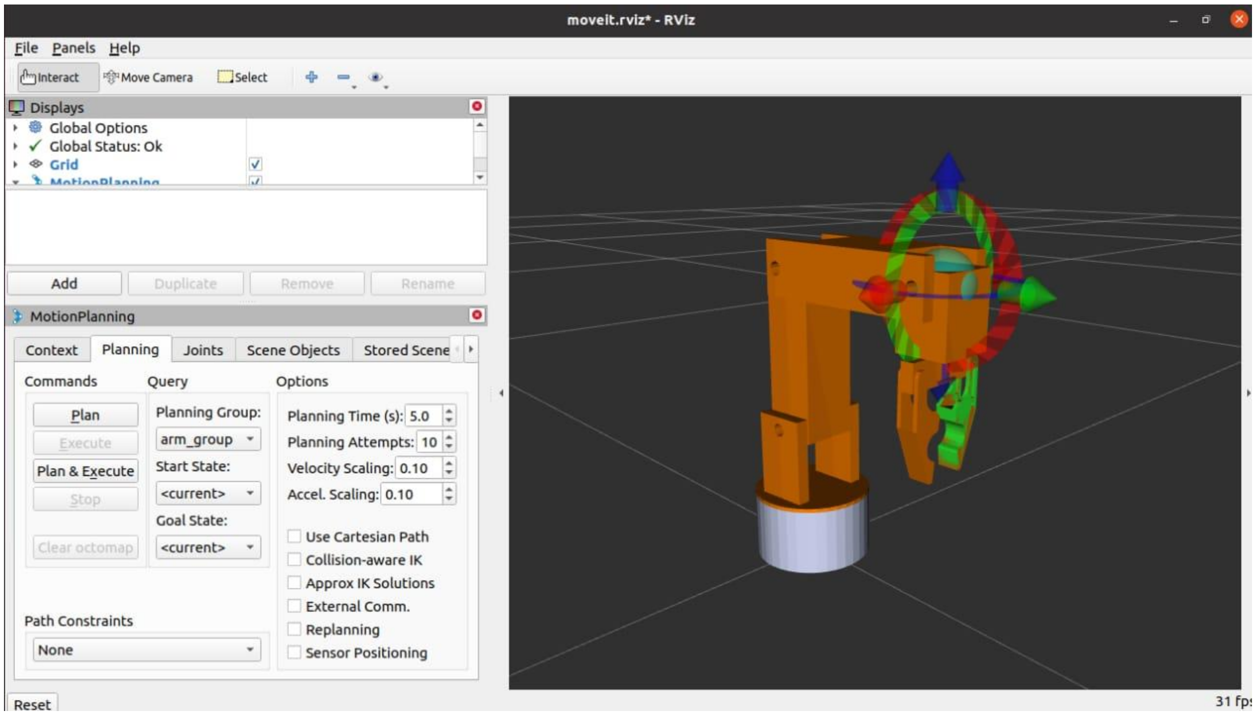
Success! Use the left navigation pane to continue.

Then we generated our package which contain all the needed launch files which run Gazebo, Rviz, RosControl tools which helps us in our simulation and control of the robotic arm.



Property	Value
name	ground_plane
is_static	✓ True
self_collide	False
enable_wind	False
pose	
link	ground_plane::link

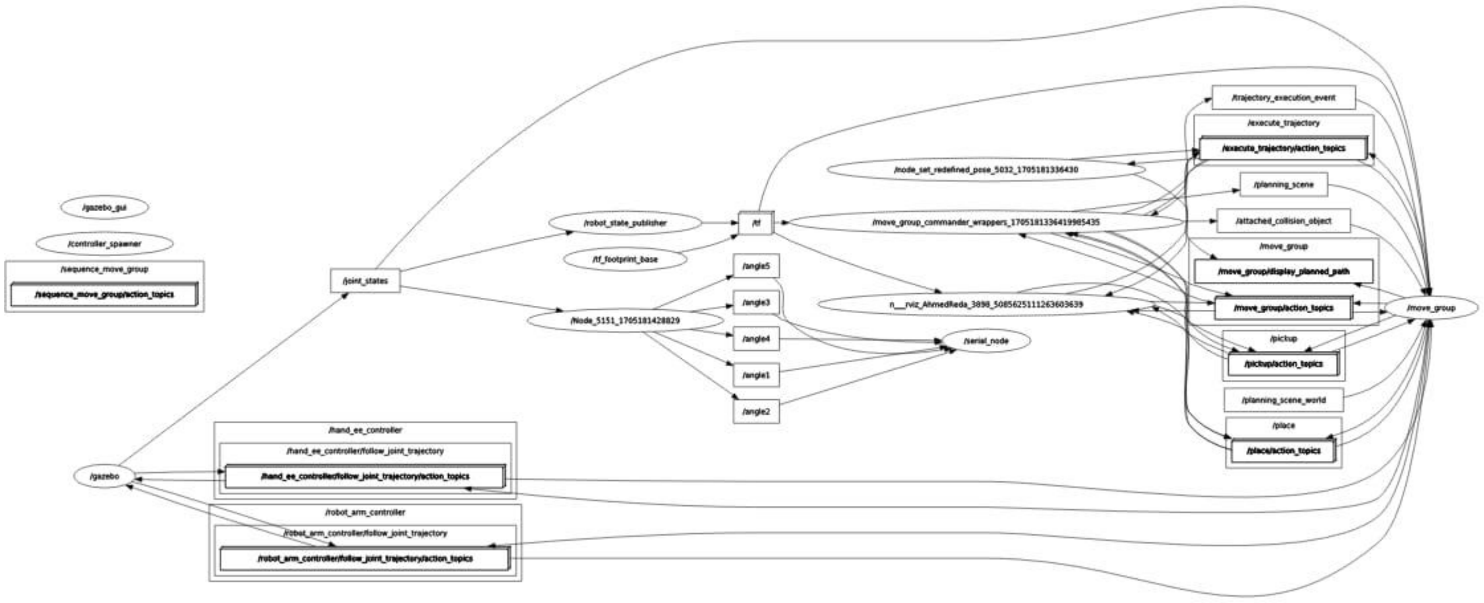
Real Time Factor: 0.73 Sim Time: 00 00:01:49.689 Real Time: 00 00:02:28.616 Iterations: 109689 FPS: 62.52



Planning Group: arm_group
Start State: <current>
Goal State: <current>
Planning Time (s): 5.0
Planning Attempts: 10
Velocity Scaling: 0.10
Accel. Scaling: 0.10
Use Cartesian Path
Collision-aware IK
Approx IK Solutions
External Comm.
Replanning
Sensor Positioning
Path Constraints: None

For the interfacing with the hardware of the robotic arm and send the angles to the servo motors according to the motion planning done by Moveit we have used ROSSERIAL with Arduino mega and we make the real robotic arm move like the simulation in Gazebo and Rviz.

Finally, this is the rqt graph of our system.


/gazebo_gui
/controller_spawner
/sequence_move_group
/sequence_move_group/action_topics
/joint_states
/gazebo
/hand_ee_controller
/hand_ee_controller/follow_joint_trajectory
/hand_ee_controller/follow_joint_trajectory/action_topics
/robot_arm_controller
/robot_arm_controller/follow_joint_trajectory
/robot_arm_controller/follow_joint_trajectory/action_topics
/robot_state_publisher
/tf_footprint_base
/tf
/Node_5151_1705181428829
/angle5
/angle3
/angle4
/angle1
/angle2
/node_set_redefined_pose_5032_1705181336430
/move_group_commander_wrappers_1705181336419985435
n__rviz_AhmedReda_3898_5085625111263603639
/serial_node
/trajectory_execution_event
/execute_trajectory
/execute_trajectory/action_topics
/planning_scene
/attached_collision_object
/move_group
/move_group/display_planned_path
/move_group/action_topics
/pickup
/pickup/action_topics
/planning_scene_world
/place
/place/action_topics
/move_group