Mechatronics and robotics

INTRO TO BIOTRONICS ASSIGNMENT

GUI Assignment

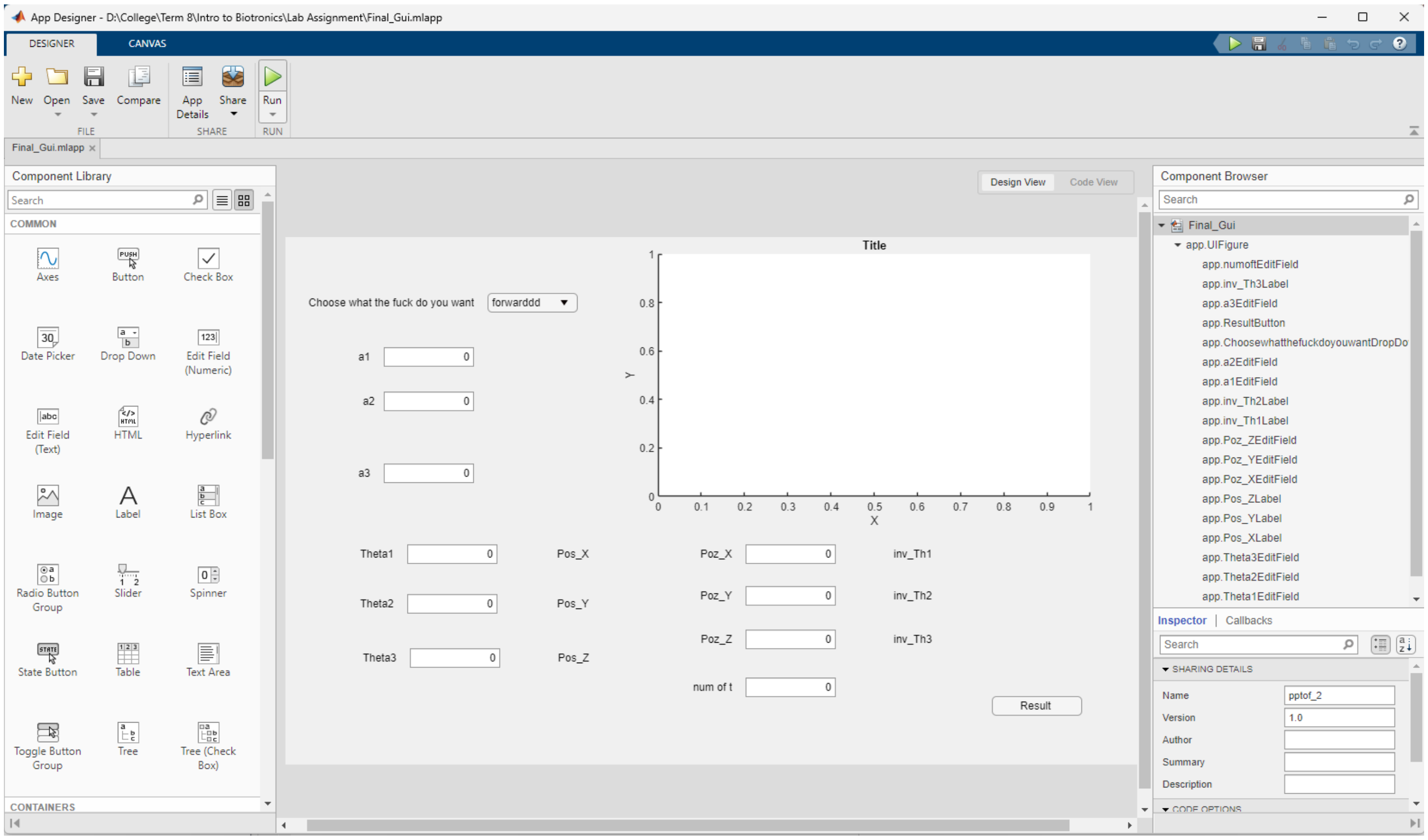
Ahmed Mohamed Kotp	7543
Omneya Haytham Mohamed	7717
Faress Ahmed Mohamed Amin	7925

GUI-Based Serial Link Kinematics Solver

Objective:

The objective of this assignment is to develop a graphical user interface (GUI) application in MATLAB for solving serial link kinematics problems. Students will design a user-friendly interface that allows users to input serial link configurations and select the type of problem (forward or inverse kinematics). They will then implement algorithms to solve the selected problem type and display the final pose of the serial link robot.

1. GUI Design:


App Designer - D:\College\Term 8\Intro to Biotronics\Lab Assignment\Final_Gui.mlapp

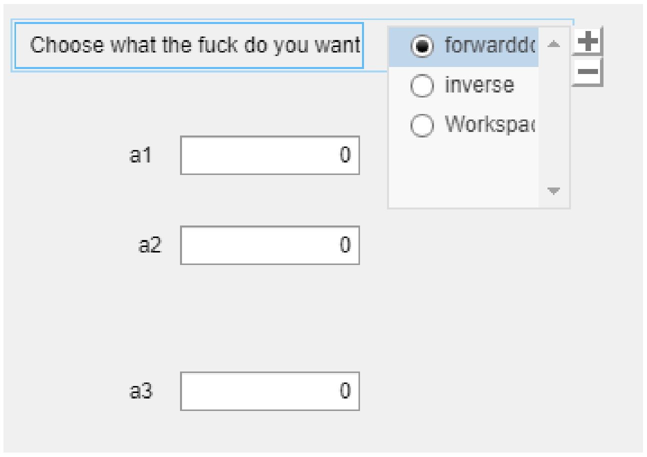
Choose what the fuck do you want: forwarddd

a1	0
a2	0
a3	0
Theta1	0
Theta2	0
Theta3	0
Poz_X	0
Poz_Y	0
Poz_Z	0
num of t	0

Component Browser: Final_Gui, app.UIFigure, app.numoftEditField, app.inv_Th3Label, app.a3EditField, app.ResultButton, app.ChoosewhatthefuckdoyouwantDropDo, app.a2EditField, app.a1EditField, app.inv_Th2Label, app.inv_Th1Label, app.Poz_ZEditField, app.Poz_YEditField, app.Poz_XEditField, app.Pos_ZLabel, app.Pos_YLabel, app.Pos_XLabel, app.Theta3EditField, app.Theta2EditField, app.Theta1EditField

SHARING DETAILS	
Name	pptof_2
Version	1.0
Author	
Summary	
Description	

- First Here we have a drop-down menu to choose between Forward kinematics, Inverse kinematics, and Workspace.



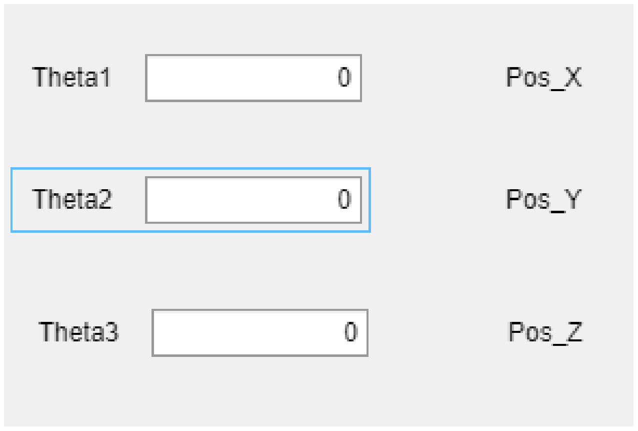
- Second, we have Edit fields for entering the three link lengths of the robot.

  - a1= length of first link
  - a2= length of second link
  - a3=length of third link

- For forward Kinematics:

we have the three joint angles input fields:

  - Theta1
  - Theta2
  - Theta3

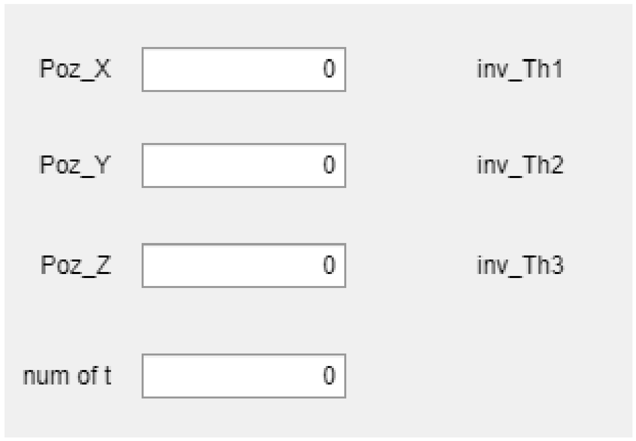


And corresponding X, Y, Z Positions for the end effector.

- For Inverse Kinematics:

We have the input fields for X, Y, Z of end effector (end effector position)

And the corresponding output thetas from the inverse kinematics:

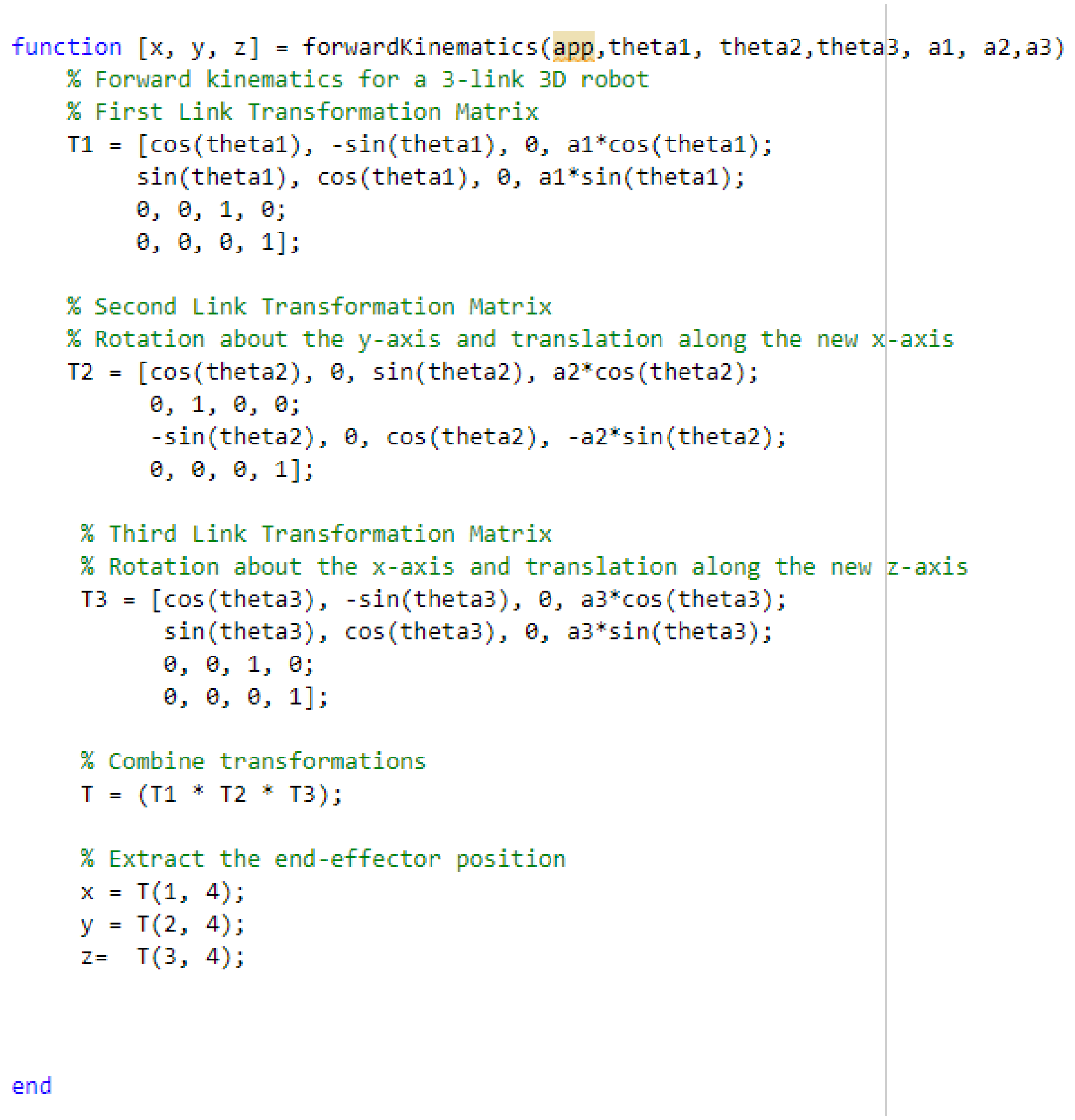


  - Inv_Th1
  - Inv_Th2
  - Inv_Th3

And we have number of iterations input field to minimize the error

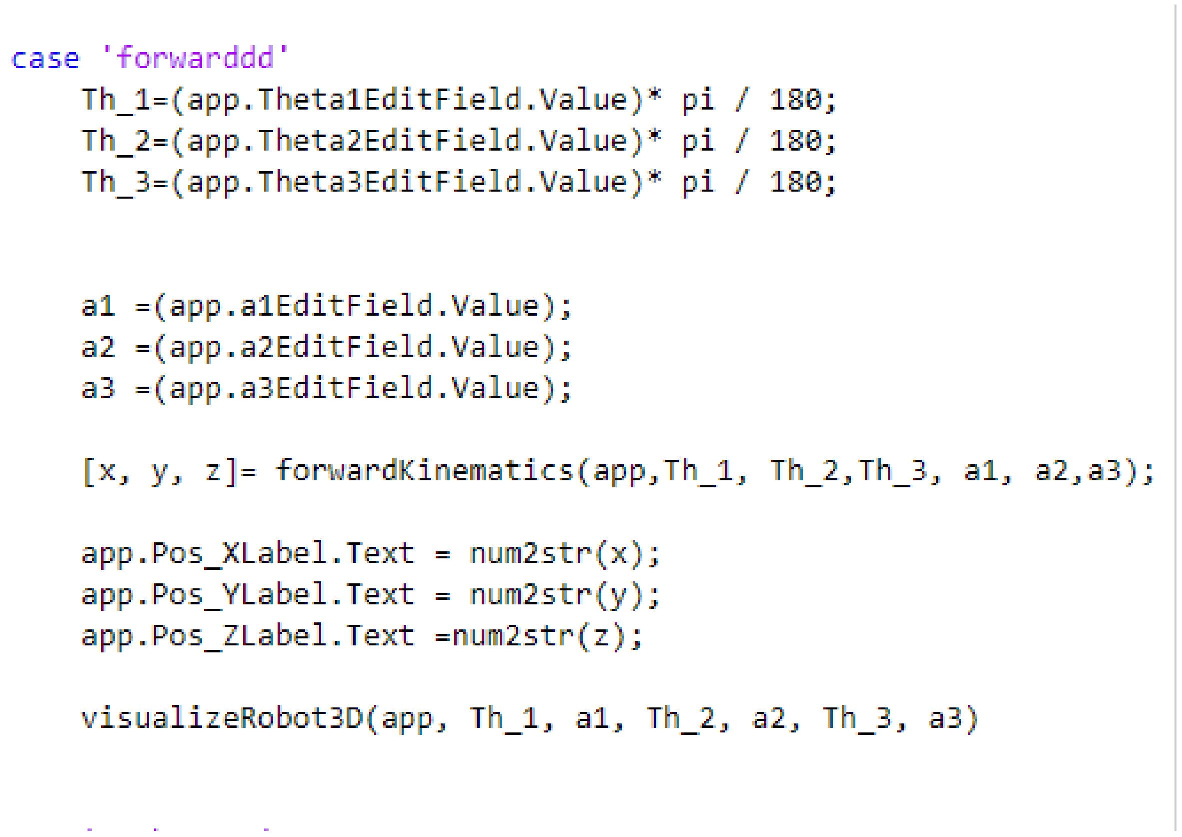
2. Forward Kinematics Solver:

Code Logic:



```
function [x, y, z] = forwardKinematics(app,theta1, theta2,theta3, a1, a2,a3)
    % Forward kinematics for a 3-link 3D robot
    % First Link Transformation Matrix
    T1 = [cos(theta1), -sin(theta1), 0, a1*cos(theta1);
         sin(theta1), cos(theta1), 0, a1*sin(theta1);
         0, 0, 1, 0;
         0, 0, 0, 1];

    % Second Link Transformation Matrix
    % Rotation about the y-axis and translation along the new x-axis
    T2 = [cos(theta2), 0, sin(theta2), a2*cos(theta2);
          0, 1, 0, 0;
          -sin(theta2), 0, cos(theta2), -a2*sin(theta2);
          0, 0, 0, 1];

     % Third Link Transformation Matrix
     % Rotation about the x-axis and translation along the new z-axis
     T3 = [cos(theta3), -sin(theta3), 0, a3*cos(theta3);
           sin(theta3), cos(theta3), 0, a3*sin(theta3);
           0, 0, 1, 0;
           0, 0, 0, 1];

     % Combine transformations
     T = (T1 * T2 * T3);

     % Extract the end-effector position
     x = T(1, 4);
     y = T(2, 4);
     z=  T(3, 4);



end
```

Here we calculate the transformation matrix of each link with respect to the previous Link, then getting the end effector transformation matrix with respect to the base by multiplying T1*T2*T3

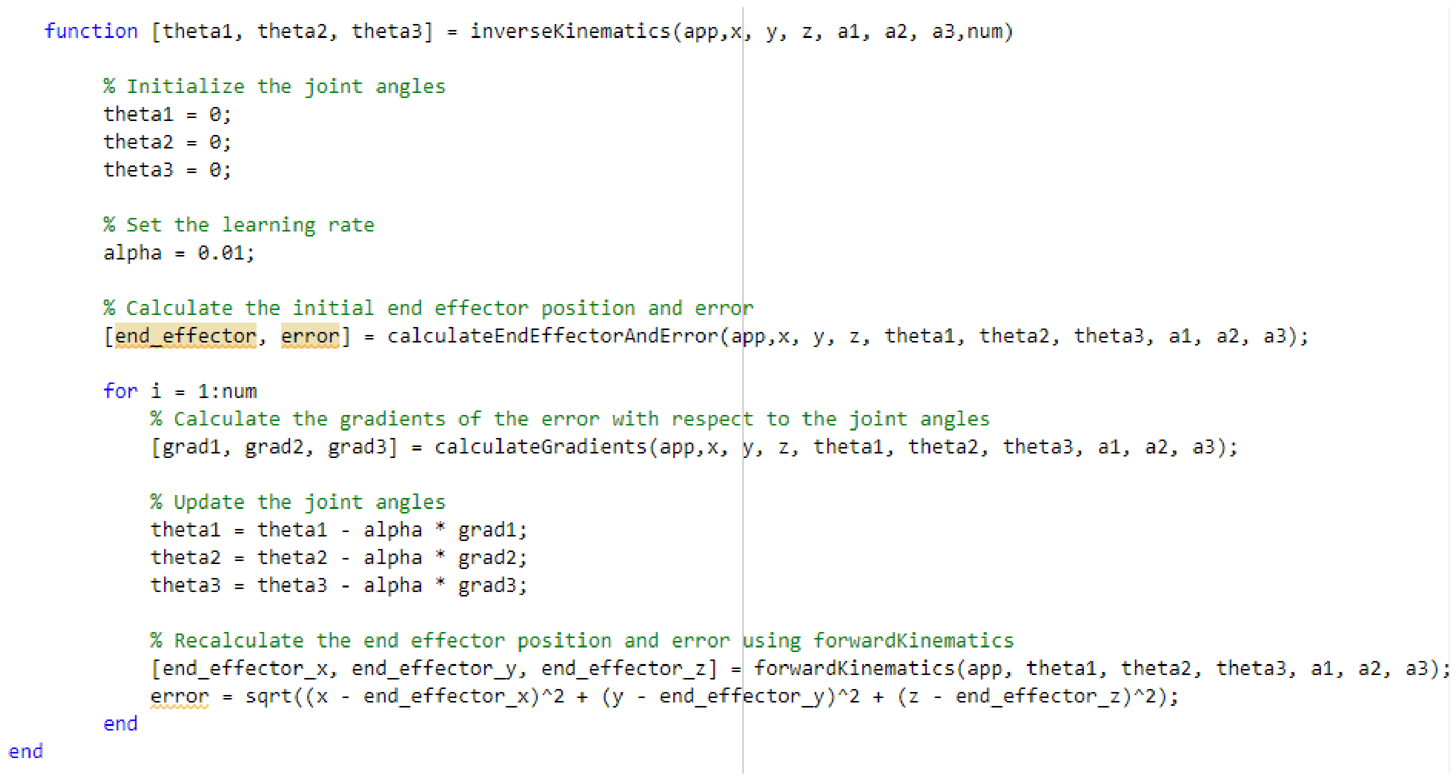


```
case 'forwarddd'
    Th_1=(app.Theta1EditField.Value)* pi / 180;
    Th_2=(app.Theta2EditField.Value)* pi / 180;
    Th_3=(app.Theta3EditField.Value)* pi / 180;


    a1 =(app.a1EditField.Value);
    a2 =(app.a2EditField.Value);
    a3 =(app.a3EditField.Value);

    [x, y, z]= forwardKinematics(app,Th_1, Th_2,Th_3, a1, a2,a3);

    app.Pos_XLabel.Text = num2str(x);
    app.Pos_YLabel.Text = num2str(y);
    app.Pos_ZLabel.Text =num2str(z);

    visualizeRobot3D(app, Th_1, a1, Th_2, a2, Th_3, a3)
```

In case of choosing Forward kinematics from drop-down menu:

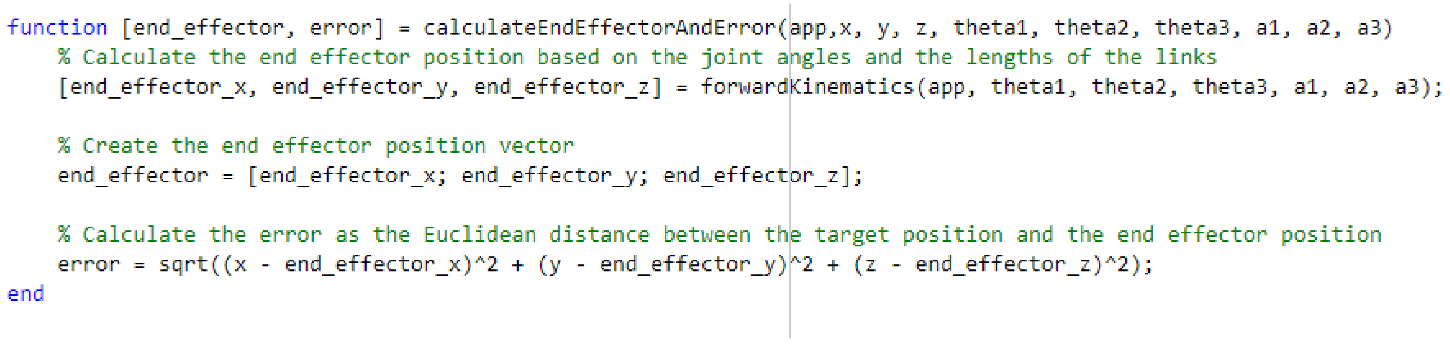
We will get the value entered of joint angles and links lengths in the input fields and use them in the forward kinematics function to get the position of end effector and plotting the visualization of the robot by a function named visualizationRobot3D.

3. Inverse Kinematics Solver:



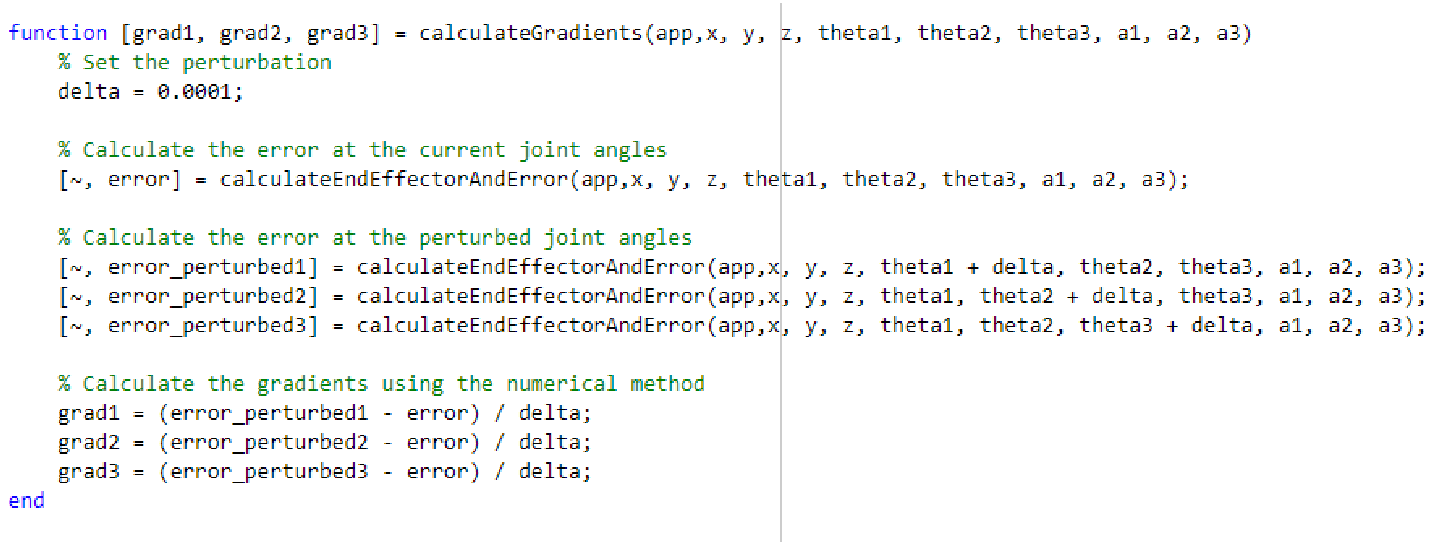
```
function [theta1, theta2, theta3] = inverseKinematics(app,x, y, z, a1, a2, a3,num)

    % Initialize the joint angles
    theta1 = 0;
    theta2 = 0;
    theta3 = 0;

    % Set the learning rate
    alpha = 0.01;

    % Calculate the initial end effector position and error
    [end_effector, error] = calculateEndEffectorAndError(app,x, y, z, theta1, theta2, theta3, a1, a2, a3);

    for i = 1:num
        % Calculate the gradients of the error with respect to the joint angles
        [grad1, grad2, grad3] = calculateGradients(app,x, y, z, theta1, theta2, theta3, a1, a2, a3);

        % Update the joint angles
        theta1 = theta1 - alpha * grad1;
        theta2 = theta2 - alpha * grad2;
        theta3 = theta3 - alpha * grad3;

        % Recalculate the end effector position and error using forwardKinematics
        [end_effector_x, end_effector_y, end_effector_z] = forwardKinematics(app, theta1, theta2, theta3, a1, a2, a3);
        error = sqrt((x - end_effector_x)^2 + (y - end_effector_y)^2 + (z - end_effector_z)^2);
    end
end
```

The main function that calculates the joint angles (theta1, theta2, theta3) for a given target position (x, y, z) using inverse kinematics.



```
function [end_effector, error] = calculateEndEffectorAndError(app,x, y, z, theta1, theta2, theta3, a1, a2, a3)
    % Calculate the end effector position based on the joint angles and the lengths of the links
    [end_effector_x, end_effector_y, end_effector_z] = forwardKinematics(app, theta1, theta2, theta3, a1, a2, a3);

    % Create the end effector position vector
    end_effector = [end_effector_x; end_effector_y; end_effector_z];

    % Calculate the error as the Euclidean distance between the target position and the end effector position
    error = sqrt((x - end_effector_x)^2 + (y - end_effector_y)^2 + (z - end_effector_z)^2);
end
```

This function calculates the end effector position and the error between the target position and the end effector position.



```
function [grad1, grad2, grad3] = calculateGradients(app,x, y, z, theta1, theta2, theta3, a1, a2, a3)
    % Set the perturbation
    delta = 0.0001;

    % Calculate the error at the current joint angles
    [~, error] = calculateEndEffectorAndError(app,x, y, z, theta1, theta2, theta3, a1, a2, a3);

    % Calculate the error at the perturbed joint angles
    [~, error_perturbed1] = calculateEndEffectorAndError(app,x, y, z, theta1 + delta, theta2, theta3, a1, a2, a3);
    [~, error_perturbed2] = calculateEndEffectorAndError(app,x, y, z, theta1, theta2 + delta, theta3, a1, a2, a3);
    [~, error_perturbed3] = calculateEndEffectorAndError(app,x, y, z, theta1, theta2, theta3 + delta, a1, a2, a3);

    % Calculate the gradients using the numerical method
    grad1 = (error_perturbed1 - error) / delta;
    grad2 = (error_perturbed2 - error) / delta;
    grad3 = (error_perturbed3 - error) / delta;
end
```

This function calculates the gradients of the error with respect to the joint angles using the numerical method.

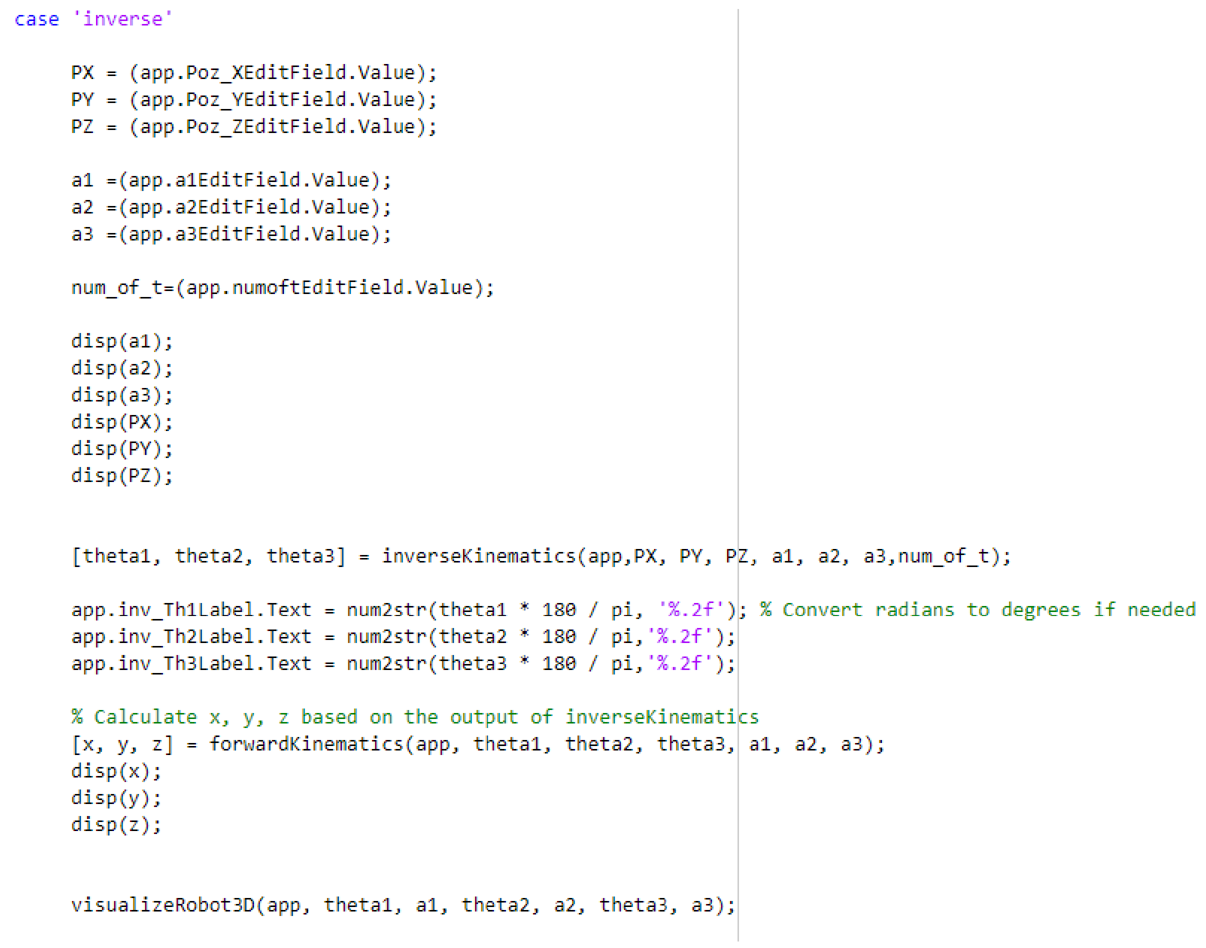
the Gradient refers to the slope or inclination of a line. However, in the context of optimization and machine learning, a gradient is a vector that points in the direction of the greatest rate of increase of a function. It’s calculated by taking the derivative of the function with respect to each of its variables. The gradient is a crucial concept in optimization algorithms because it provides guidance on how to adjust parameters to minimize or maximize a particular function.

the gradient descent method is a first-order optimization algorithm used to find the minimum of a function. It’s widely used in machine learning and deep learning for training models, where the function to be minimized is often a loss function that calculates the difference between the actual data and the model's predictions.

The gradient method works by iteratively adjusting the input of a function in the opposite direction of the gradient (for minimization).

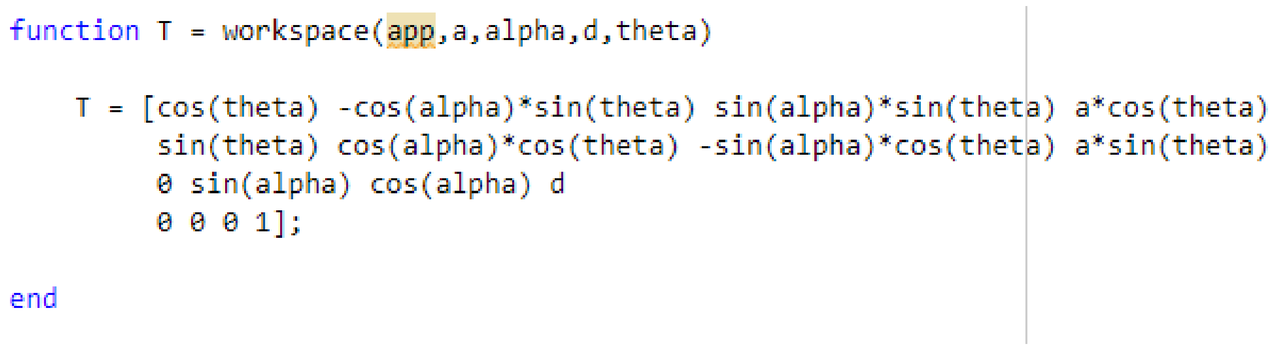
This is done until the algorithm converges to a local minimum. Here's a step-by-step process of how the gradient method works for minimization:

1. Initialize the input parameters.
2. Compute the gradient of the function at the current input.
3. Update the input by subtracting a fraction (known as the learning rate) of the gradient from the current input:
4. xn+1=xn−α∇f(xn)
5. where xn is the current input,
6. α is the learning rate,
7. and ∇f(xn) is the gradient of the function at xn
8. Repeat the above steps until the change in the function output is below a certain threshold or a maximum number of iterations is reached.



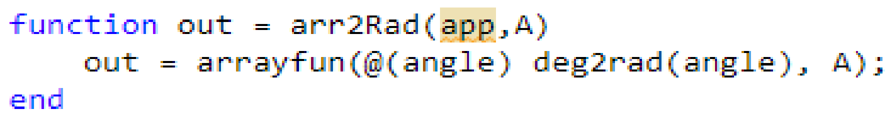
```
case 'inverse'

    PX = (app.Poz_XEditField.Value);
    PY = (app.Poz_YEditField.Value);
    PZ = (app.Poz_ZEditField.Value);

    a1 =(app.a1EditField.Value);
    a2 =(app.a2EditField.Value);
    a3 =(app.a3EditField.Value);

    num_of_t=(app.numoftEditField.Value);

    disp(a1);
    disp(a2);
    disp(a3);
    disp(PX);
    disp(PY);
    disp(PZ);


    [theta1, theta2, theta3] = inverseKinematics(app,PX, PY, PZ, a1, a2, a3,num_of_t);

    app.inv_Th1Label.Text = num2str(theta1 * 180 / pi, '%.2f'); % Convert radians to degrees if needed
    app.inv_Th2Label.Text = num2str(theta2 * 180 / pi,'%.2f');
    app.inv_Th3Label.Text = num2str(theta3 * 180 / pi,'%.2f');

    % Calculate x, y, z based on the output of inverseKinematics
    [x, y, z] = forwardKinematics(app, theta1, theta2, theta3, a1, a2, a3);
    disp(x);
    disp(y);
    disp(z);


    visualizeRobot3D(app, theta1, a1, theta2, a2, theta3, a3);
```

4. Robot workspace drawing:



```
function T = workspace(app,a,alpha,d,theta)

    T = [cos(theta) -cos(alpha)*sin(theta) sin(alpha)*sin(theta) a*cos(theta)
         sin(theta) cos(alpha)*cos(theta) -sin(alpha)*cos(theta) a*sin(theta)
         0 sin(alpha) cos(alpha) d
         0 0 0 1];

end
```

This is a function that have the transformation matrix we substitute in it with each theta and a



```
function out = arr2Rad(app,A)
    out = arrayfun(@(angle) deg2rad(angle), A);
end
```

- function out = arr2Rad(app,A):
  - This line defines the function arr2Rad with two input arguments: app and A.
  - A is the array of angles that we want to convert from degrees to radians.
  - The function will return the output as out.

- out = arrayfun(@(angle) deg2rad(angle), A);:
  - This line uses the arrayfun function to apply the conversion operation to each element of the array A.
  - arrayfun applies the function @(angle) deg2rad(angle) to each element of A.
  - @(angle) is an anonymous function that takes an input argument angle.
  - deg2rad(angle) is a built-in MATLAB function that converts an angle from degrees to radians.
  - The result of arrayfun is assigned to the variable out, which becomes the output of the arr2Rad function.

```
case 'Workspace'

    % Clear previous plot
    cla(app.UIAxes);

    a1 =(app.a1EditField.Value);
    a2 =(app.a2EditField.Value);
    a3 =(app.a3EditField.Value);

    % DH parameters inserted manually
    a_1=0;  alpha1=-pi/2;  d1=a1;
    a_2=a2; alpha2=0;      d2=0;
    a_3=a3; alpha3=0;      d3=0;

    t1_min=0;                t1_max=170*(pi/180);
    t2_min=0;                t2_max=30*(pi/180);
    t3_min=-20*(pi/180);     t3_max=170*(pi/180);

    theta1=t1_min + (t1_max - t1_min)*rand(10000,1);
    theta2=t2_min + (t2_max - t2_min)*rand(10000,1);
    theta3=t3_min + (t3_max - t3_min)*rand(10000,1);

    %rand: This is a built-in MATLAB function that generates random numbers from a uniform distribution between 0 and 1.
    % N: It specifies the number of random numbers to generate.
    %1: It indicates that the generated random numbers should be arranged as a column vector.

   for i=1:10000

        A1=workspace(app,a_1,alpha1,d1,theta1(i));
        A2=workspace(app,a_2,alpha2,d2,theta2(i));
        A3=workspace(app,a_3,alpha3,d3,theta3(i));

        T=A1*A2*A3;

        X=T(1,4);
        Y=T(2,4);
        Z=T(3,4);

        plot3(app.UIAxes,X,Y,Z,'.')
        hold(app.UIAxes, 'on')

   end
```

- cla(app.UIAxes):

This line clears the previous plot from the UIAxes object, which is likely a MATLAB GUI component used for displaying plots.

- The following lines assign values from GUI input fields.
  - a1 = app. a1EditField.Value
  - a2 = app. a2EditField.Value
  - a3 = app. a3EditField.Valuet

- Then manually set the Denavit-Hartenberg (DH) parameters for the robotic arm. These parameters describe the kinematic properties of each joint in the arm. Specifically:

  - a_1, alpha1, d1 represent the parameters for the first joint.
  - a_2, alpha2, d2 represent the parameters for the second joint.
  - a_3, alpha3, d3 represent the parameters for the third joint.

- define the minimum and maximum joint angles:
  - t1_min, t1_max
  - t2_min, t2_max
  - t3_min, t3_max

for each joint of the robotic arm.

- The theta1, theta2, and theta3 variables are generated using the rand function to create 10,000 random joint angle configurations within the specified ranges.

- The subsequent for loop iterates over each random joint angle configuration.

- Within the loop, the workspace function is called three times with different joint parameters and the current random joint angles:
  - theta1(i)
  - theta2(i)
  - theta3(i))

to calculate the transformation matrices A1, A2, and A3 for each joint.

- The transformation matrices A1, A2, and A3 are multiplied together to obtain the overall transformation matrix T representing the end-effector pose.

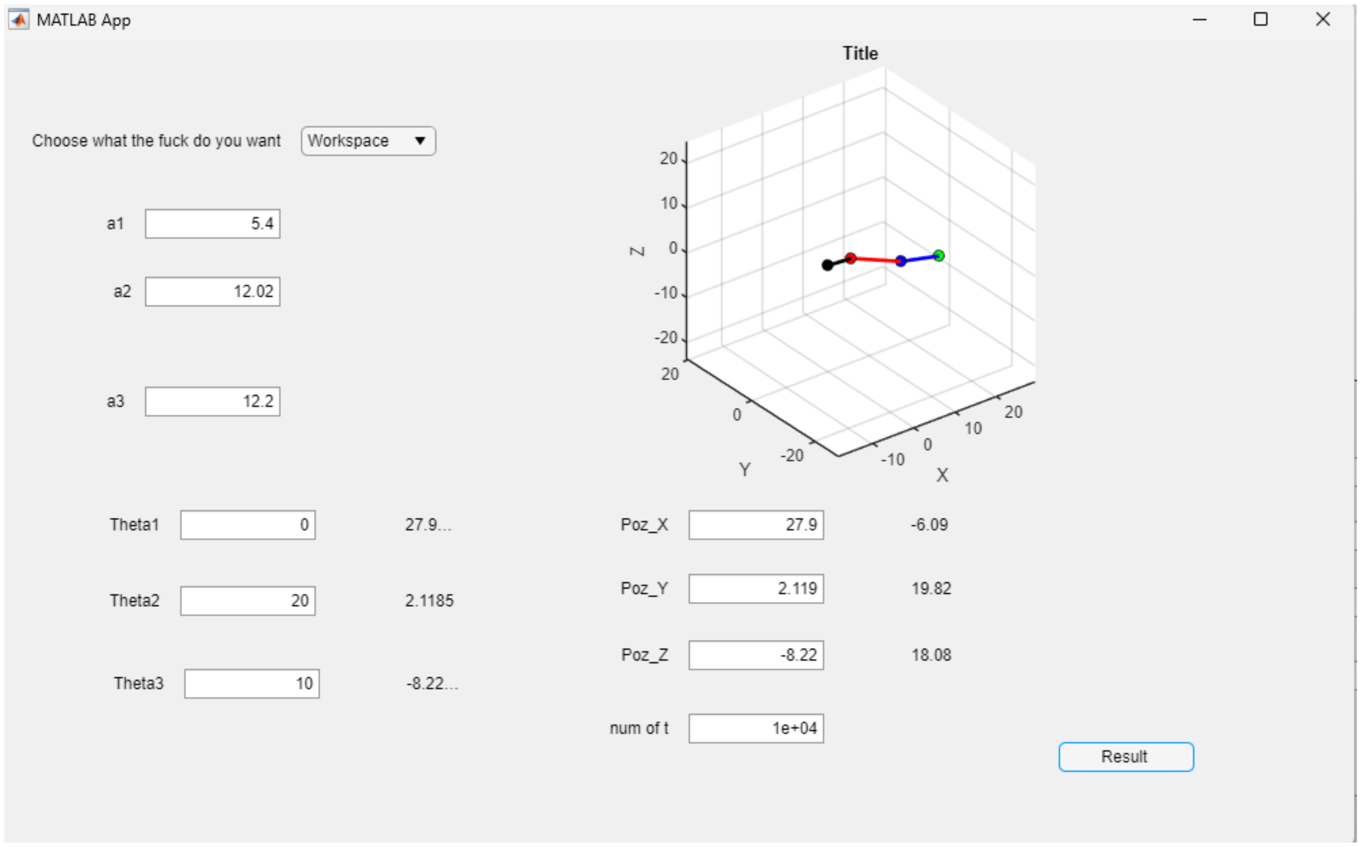
- The X, Y, and Z coordinates of the end-effector position are extracted from the transformation matrix T.

- The plot3 function is used to plot the X, Y, and Z coordinates on the UIAxes object as individual data points in a three-dimensional plot.

- The hold function is used to ensure that subsequent plots are overlaid on the same figure.

- The loop continues until all 10,000 random joint angle configurations have been processed.

5. Testing and Validation:



Choose what the fuck do you want: Workspace

a1: 5.4
a2: 12.02
a3: 12.2

Theta1: 0 27.9...
Theta2: 20 2.1185
Theta3: 10 -8.22...

Poz_X: 27.9 -6.09
Poz_Y: 2.119 19.82
Poz_Z: -8.22 18.08
num of t: 1e+04

A. GUI

Here is and example using the GUI we created with links lengths:

- ✓ a1 = 5.4
- ✓ a2 = 12.02
- ✓ a3 = 12.2

1. In forward Kinematics:

Joint revolute angles:

- ✓ Theta1 = 0 degrees
- ✓ Theta2 = 20 degrees
- ✓ Theta3 = 10 degrees

The output of end effector position:

- ✓ X = 27.9
- ✓ Y = 2.1185
- ✓ Z = -8.22

2. Inverse Kinematics:

The input is the same as the output from forward kinematics to check the error.

Input end effector position:

- ✓ X = 27.9
- ✓ Y = 2.1185
- ✓ Z = -8.22

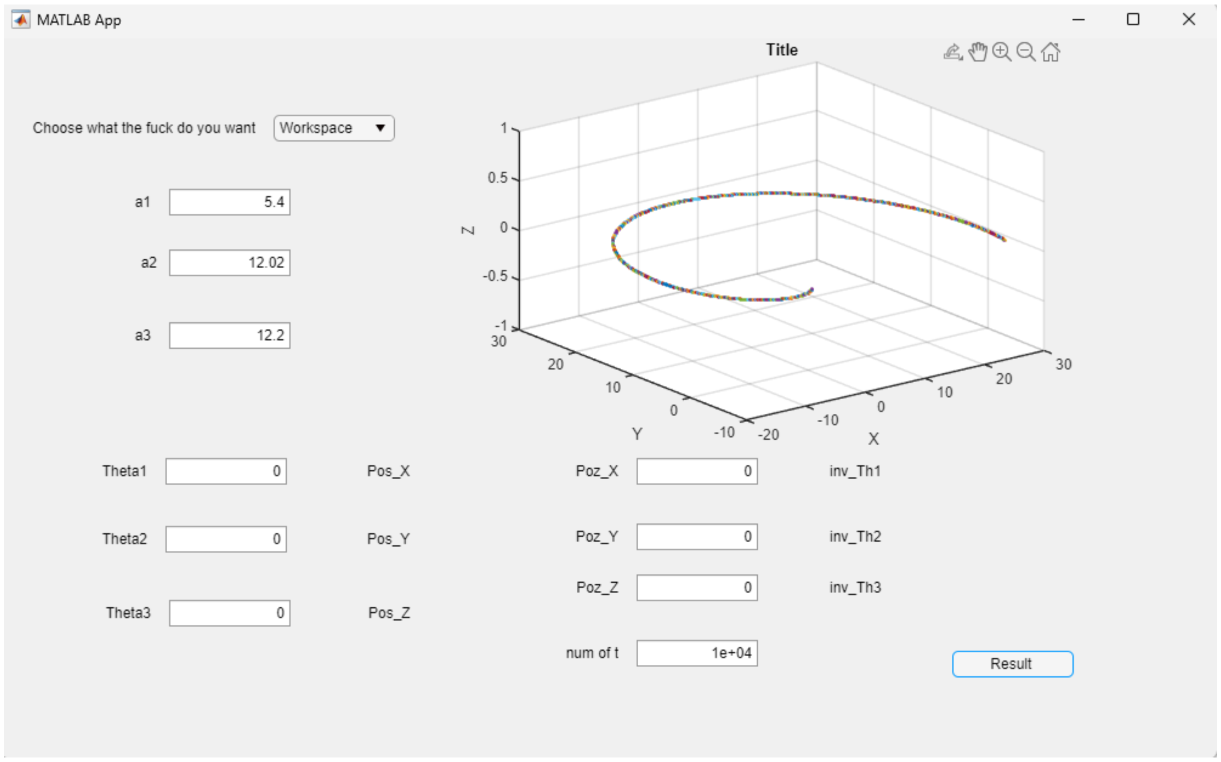
Output revolute angles of each joint:

- ✓ Inv_Theta1 = -6.09
- ✓ Inv_Theta2 = 19.82
- ✓ Inv_Theta3 = 18.08

As observed, there is error, but the error is small.

3. Workspace:

If we worked only planner (no rotation about z-axis):



Choose what the fuck do you want: Workspace

a1: 5.4
a2: 12.02
a3: 12.2

Theta1: 0
Theta2: 0
Theta3: 0

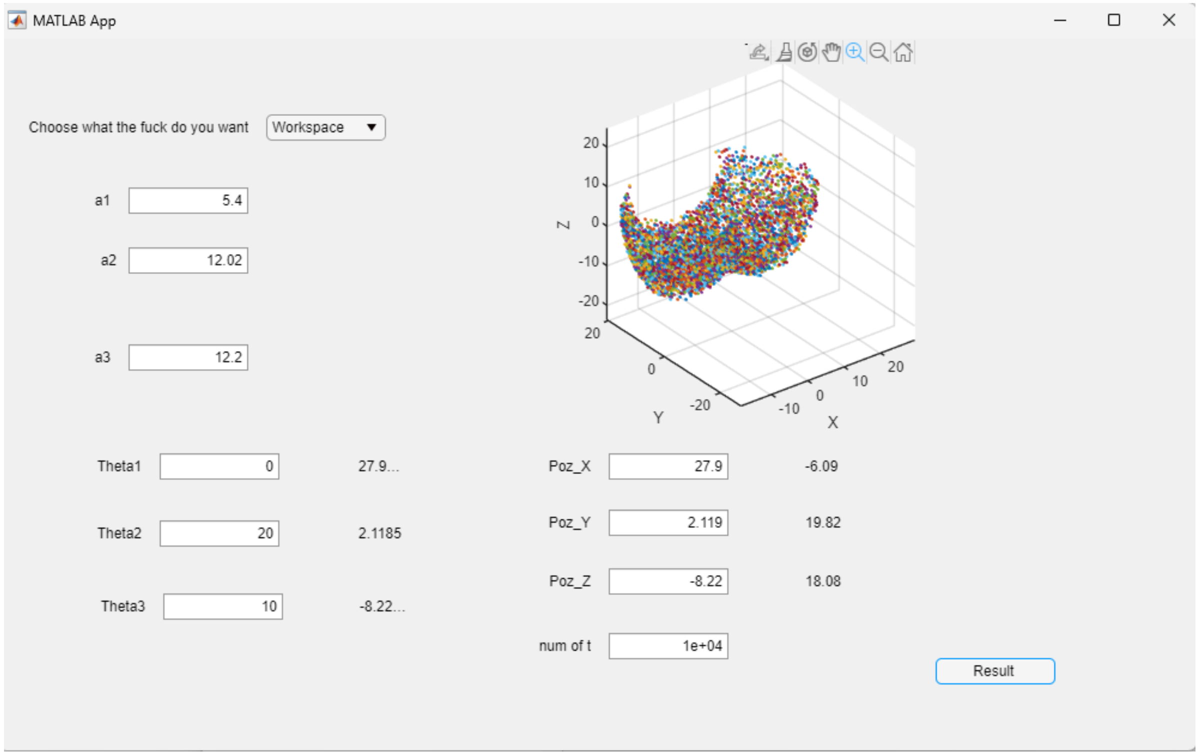
Pos_X
Pos_Y
Pos_Z

Poz_X: 0
Poz_Y: 0
Poz_Z: 0

num of t: 1e+04

inv_Th1
inv_Th2
inv_Th3

If we took into consideration rotation about Z:



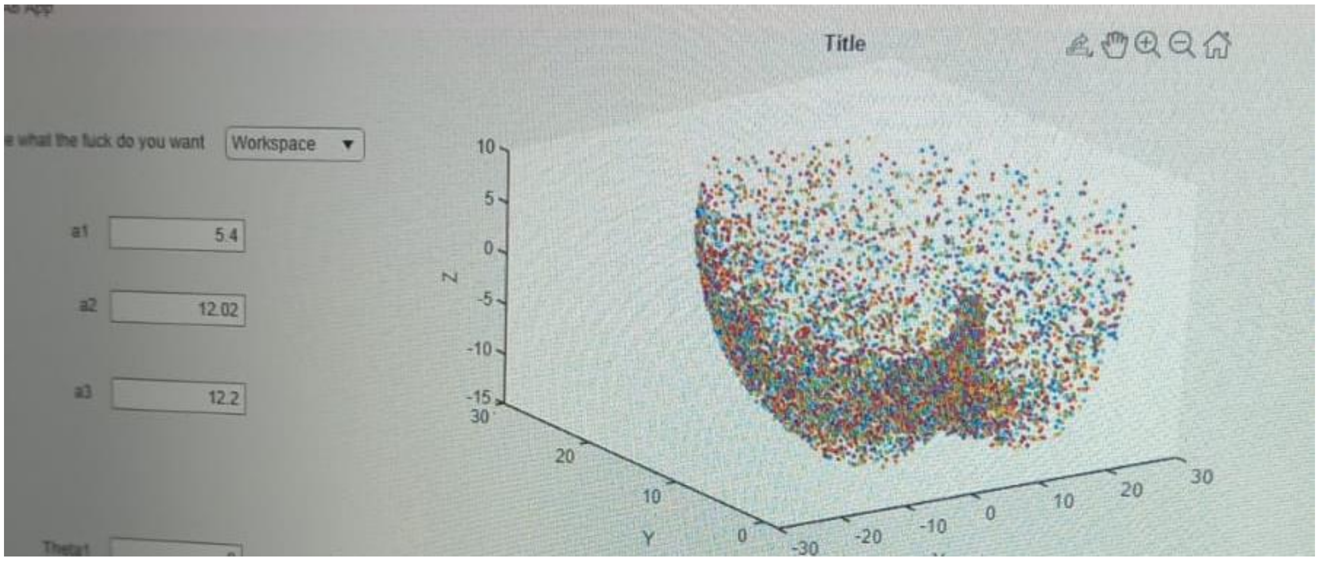
Choose what the fuck do you want: Workspace

a1: 5.4
a2: 12.02
a3: 12.2

Theta1: 0 — 27.9...
Theta2: 20 — 2.1185
Theta3: 10 — -8.22...

Poz_X: 27.9 — -6.09
Poz_Y: 2.119 — 19.82
Poz_Z: -8.22 — 18.08

num of t: 1e+04



e what the fuck do you want Workspace

a1 5.4

a2 12.02

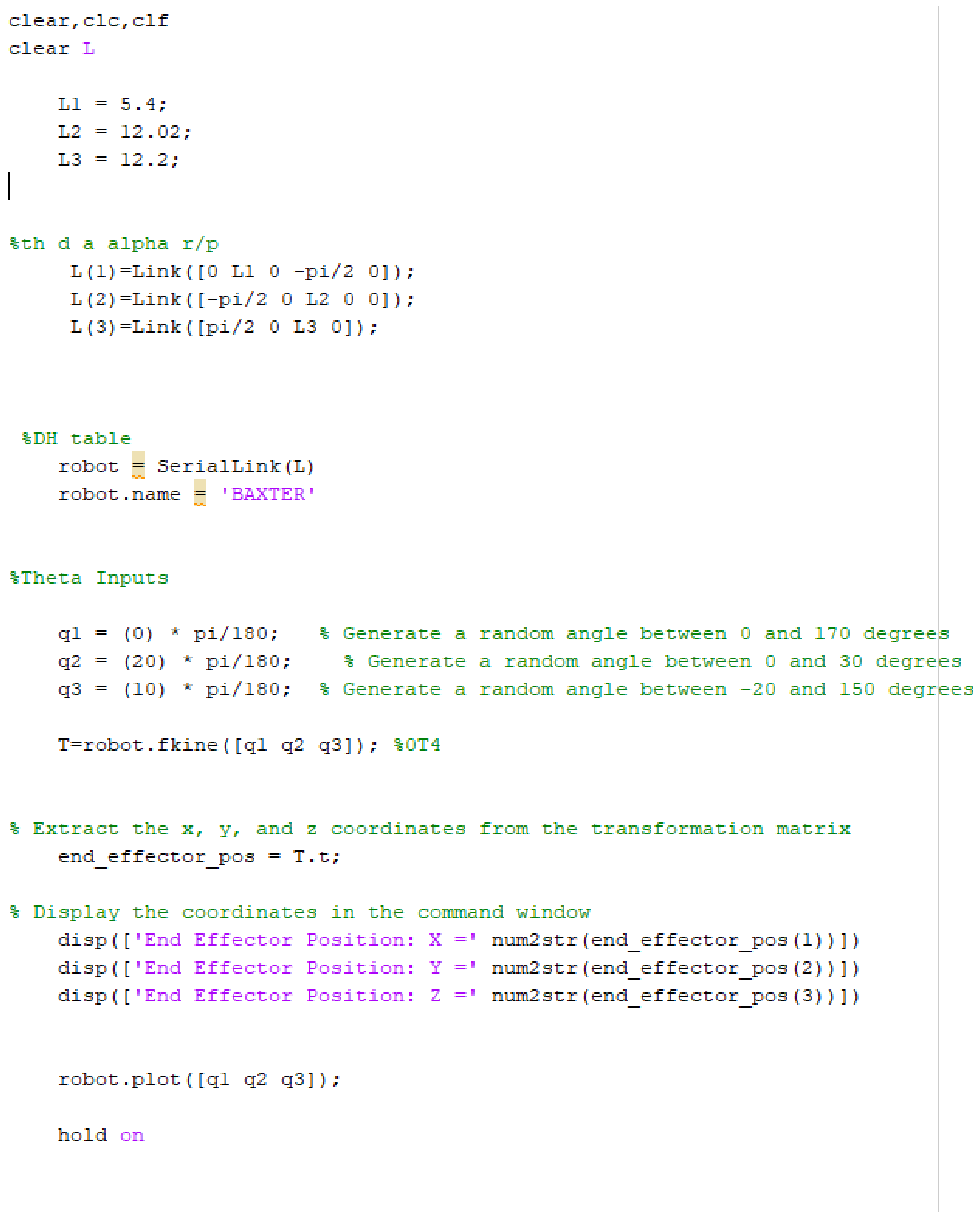
a3 12.2

Theta1

B. Peter Cork:

1. In forward Kinematics:

Code:



```
clear,clc,clf
clear L

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


%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]);



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


%Theta Inputs

    q1 = (0) * pi/180;   % Generate a random angle between 0 and 170 degrees
    q2 = (20) * pi/180;    % Generate a random angle between 0 and 30 degrees
    q3 = (10) * pi/180;  % Generate a random angle between -20 and 150 degrees

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


% Extract the x, y, and z coordinates from the transformation matrix
    end_effector_pos = T.t;

% Display the coordinates in the command window
    disp(['End Effector Position: X =' num2str(end_effector_pos(1))])
    disp(['End Effector Position: Y =' num2str(end_effector_pos(2))])
    disp(['End Effector Position: Z =' num2str(end_effector_pos(3))])


    robot.plot([q1 q2 q3]);

    hold on
```

Figure output:

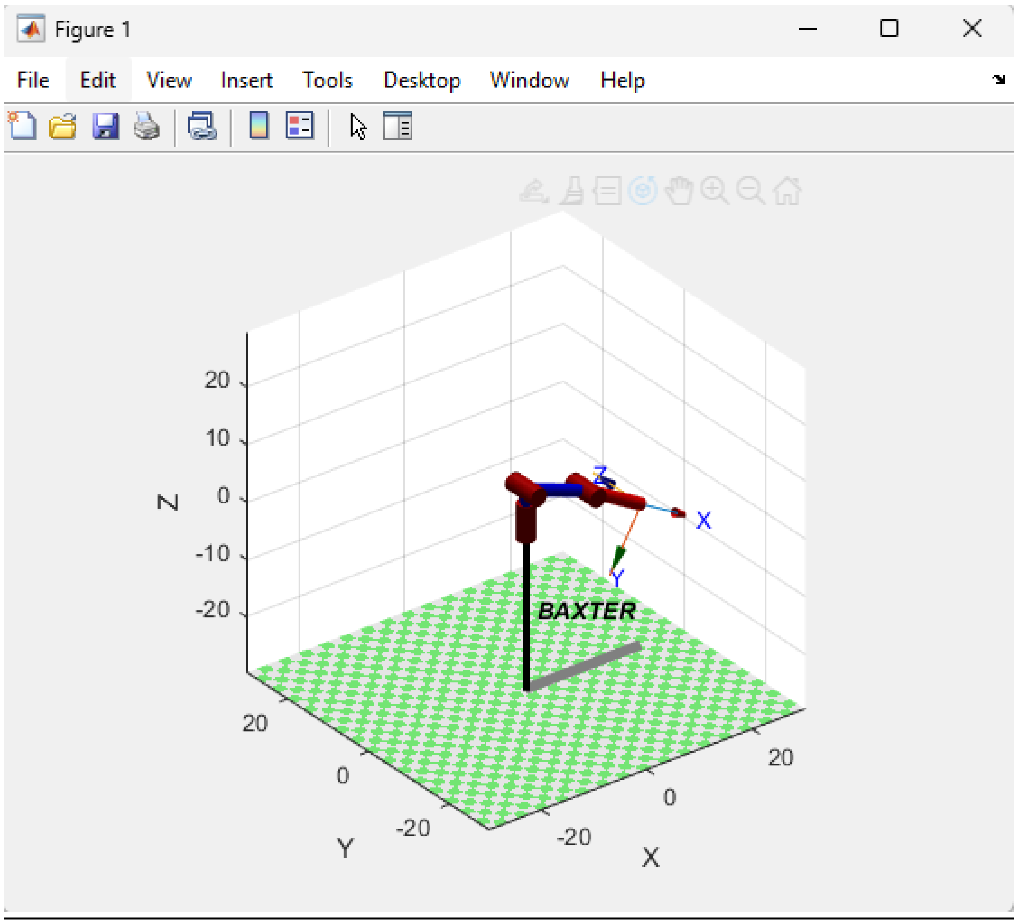
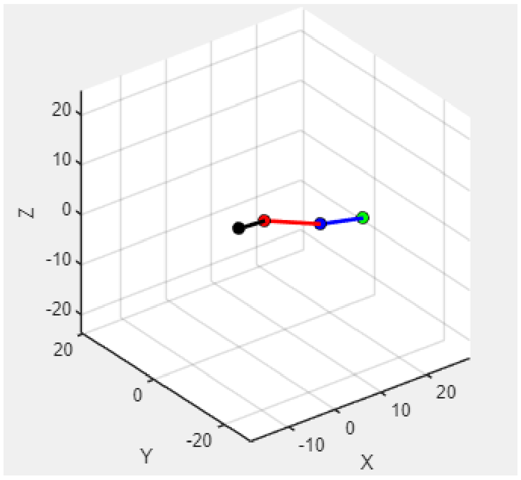


Figure from GUI



DH_Table and end effector position:

```
robot =

BAXTER:: 3 axis, RRR, 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|
+---+-----------+-----------+-----------+-----------+-----------+

End Effector Position: X =21.8606
End Effector Position: Y =6.2525e-16
End Effector Position: Z =-4.8111
```

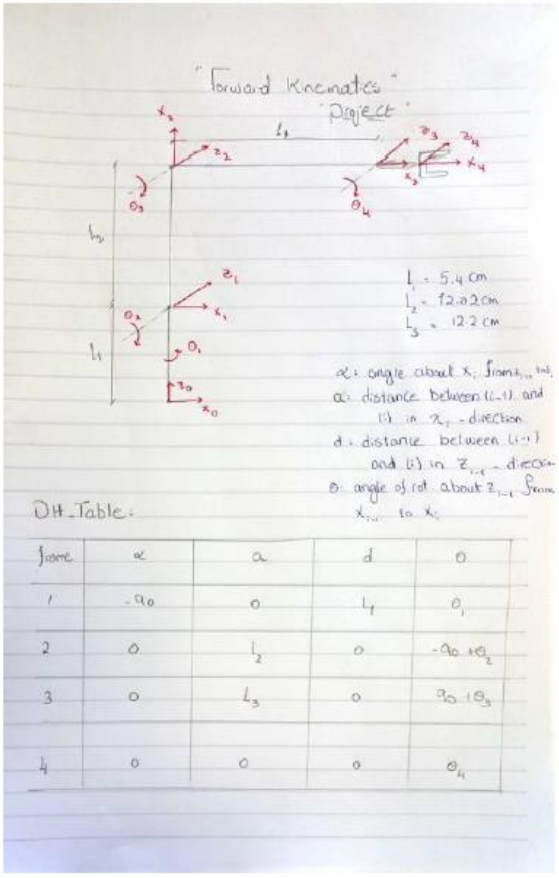
It is almost equal to the output of solver from GUI with a small error.

End effector position from Gui :

- ✓ X = 21.8606
- ✓ Y = 6.25
- ✓ Z = -4.8111

➢ **Forward Kinematics (DH Table):**

Handwritten problem:



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:

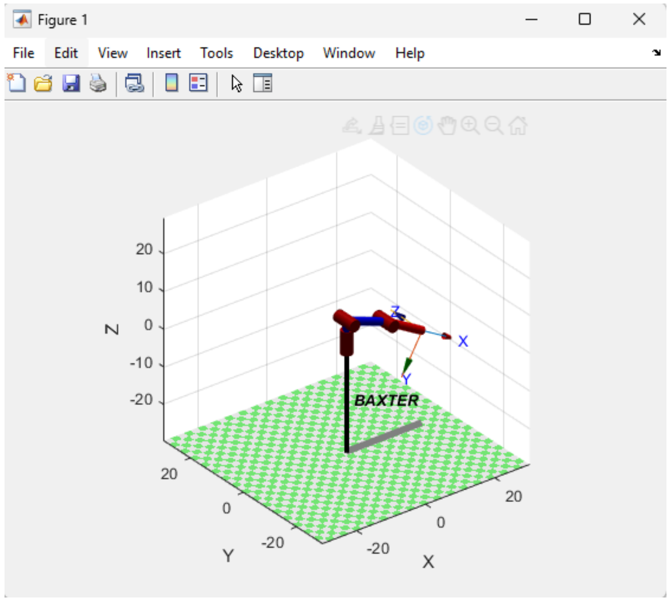
frame	α	a	d	θ
1	-90	0	L_1	θ_1
2	0	L_2	0	$-90+\theta_2$
3	0	L_3	0	$90+\theta_3$
4	0	0	0	θ_4

2. Inverse Kinematics:

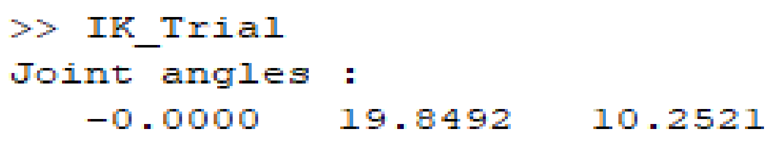
Code:

```
% Define the link lengths
L1 = 5.4;
L2 = 12.02;
L3 = 12.2;

% Create a 3-link planar manipulator
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, 0]);

robot = SerialLink(L, 'name', '3-Link Planar Manipulator');

% Define the end effector position
X = 27.9;
Y = 2.119;
Z = -8.22;

Tep = transl(X, Y, Z);

% Solve the inverse kinematics problem
q = robot.ikine(Tep, 'mask', [1 1 1 0 0 0]);

% Display the joint angles
disp('Joint angles :')
disp(q*180/pi)

% Plot the robot
robot.plot(q);
```

Figure output:



<u>Joint angles:</u>



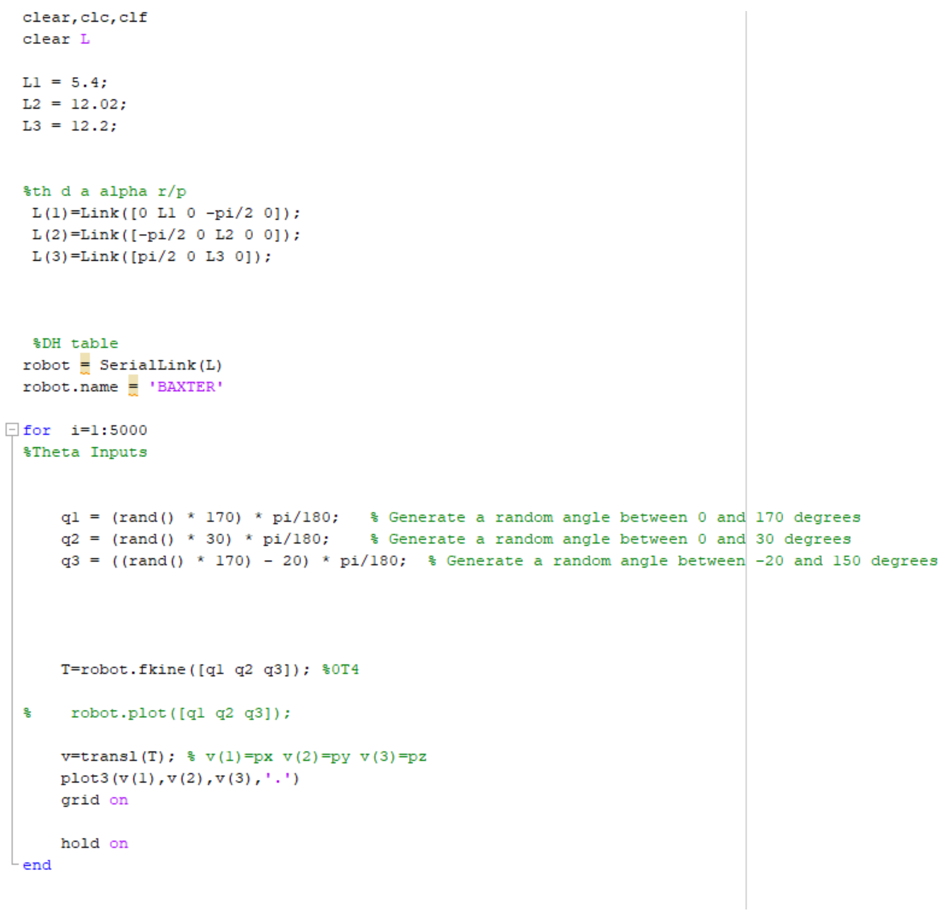
```
>> IK_Trial
Joint angles :
   -0.0000    19.8492    10.2521
```

From GUI:

- ✓ Inv_Theta1 = -6.09
- ✓ Inv_Theta2 = 19.82
- ✓ Inv_Theta3 = 18.08
- ✓ As observed, there is error, but the error is small.

3. Workspace:

<u>Code</u>



```
clear,clc,clf
clear L

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


%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]);



 %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




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

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

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

    hold on
end
```

Figure output:

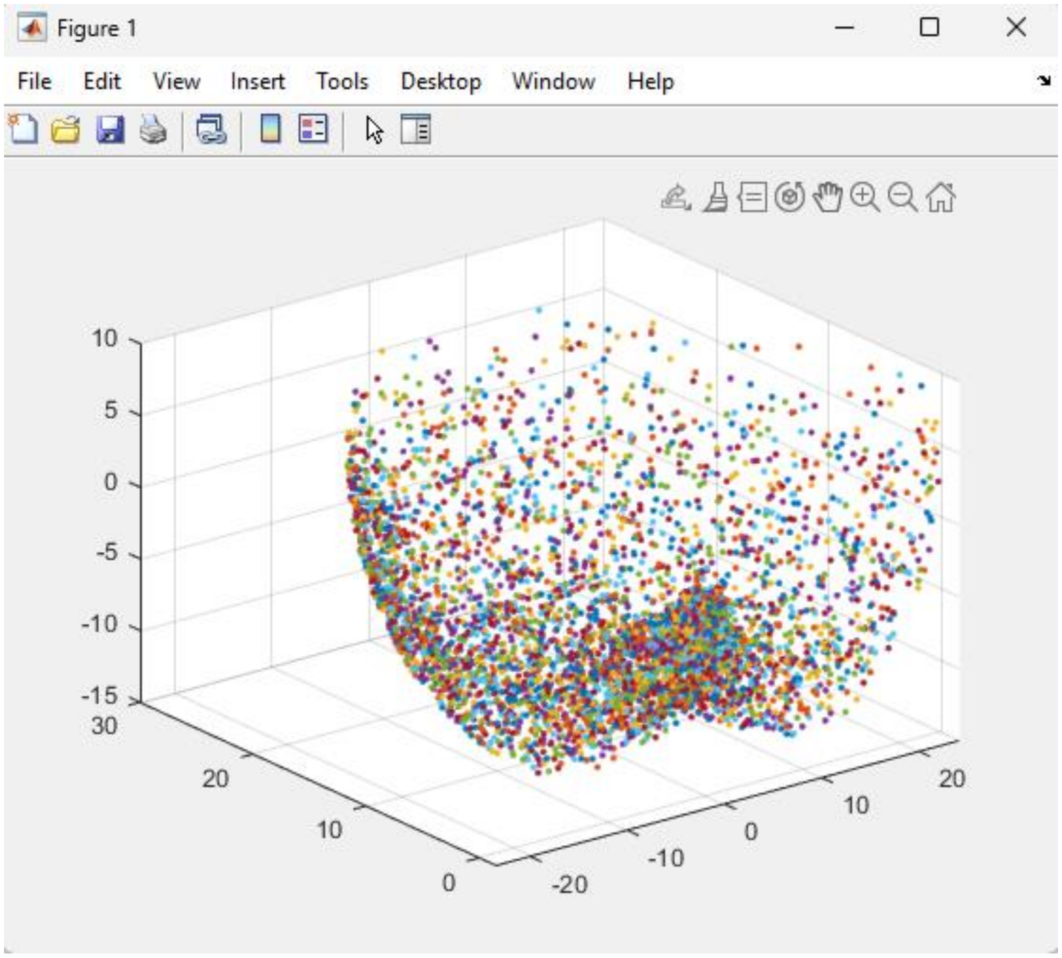
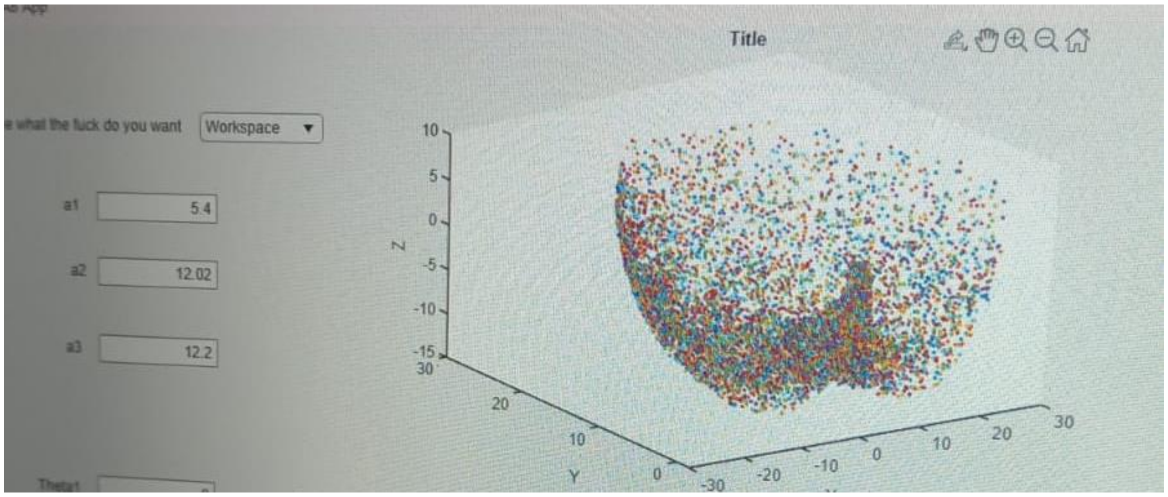


Figure from GUI



e what the fuck do you want: Workspace

a1: 5.4

a2: 12.02

a3: 12.2

Theta1