FOUNDATIONS OF ROBOTICS

TERM PROJECT

By

Shrey Shridhar Kulkarni
ss16015

Kiruthiga Chandra Shekar
kc5012

TABLE OF CONTENTS

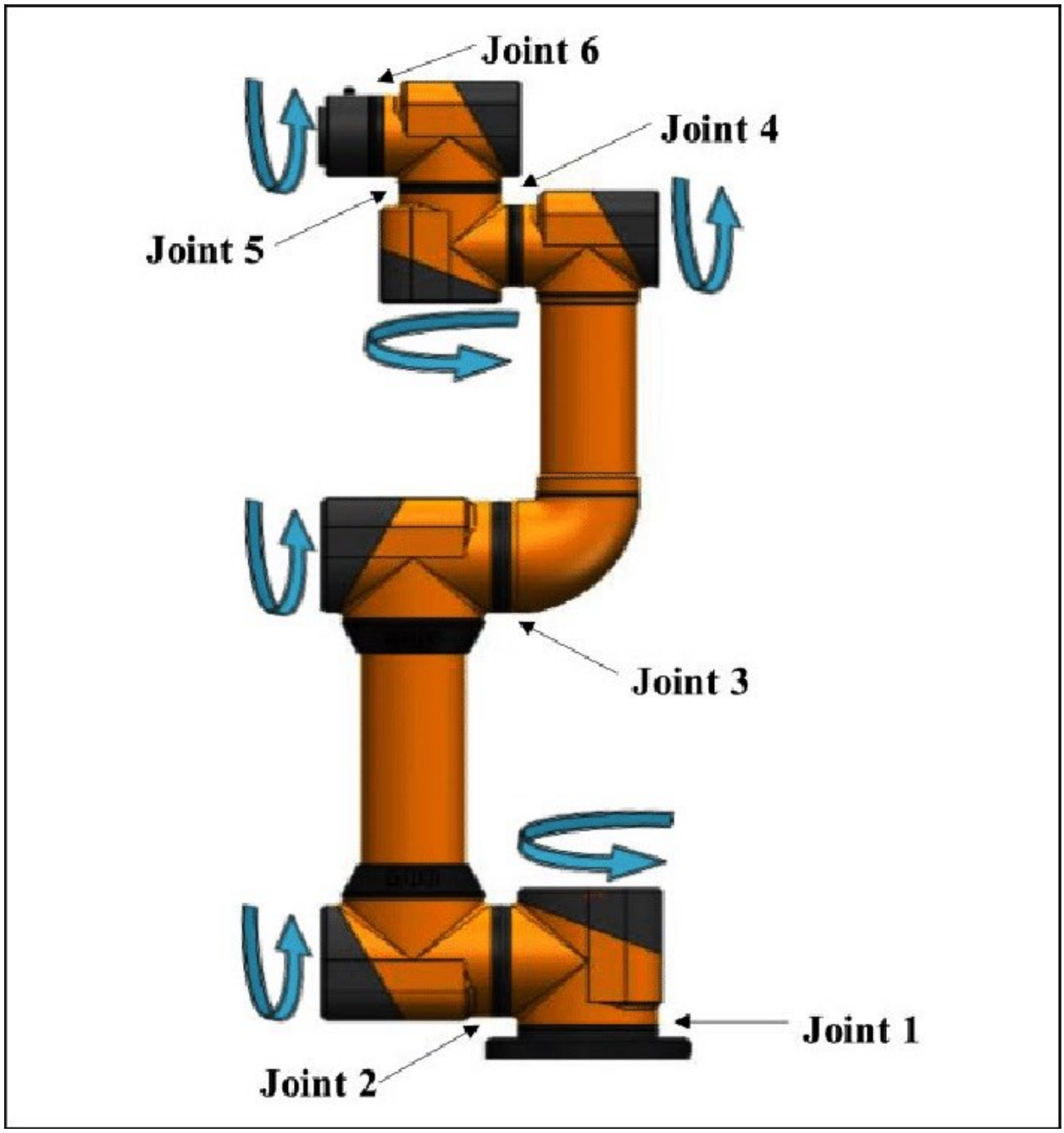

Figure 1

USER MANUAL

- The MATLAB version 2022B has been used.
- Every file is a **stand-alone file**. No dependency/requirement to run another file. **Figure 1** is the robot we have tried to emulate.
- Four major files, all ***.mlx*** type:
    - Forward Kinematics
    - Inverse Kinematics (Analytical)
    - Inverse Kinematics (Jacobian)
    - Workspace
- The visualization of a robot has been implemented in every file, therefore, we haven't dedicated a separate file for it.
- In addition, we have attached images of results in a separate file named "Results".
- Furthermore, this user-manual also contains snippets of the code and results.

RESULTS

Forward Kinematics

- For implementing forward kinematics, we have employed the Denavit-Hartenberg convention.
- All joint variables for the 6R robot, have been initialized as symbolic variables.
- **Figure 3** is an implementation of the transformation matrix associated with the dh convention.
- **Figure 4 & 5** are the results. The results display (x,y,z,roll,pitch,yaw)

$$
\begin{aligned}
A_i &= R_{z,\theta_i}\,\mathrm{Trans}_{z,d_i}\,\mathrm{Trans}_{x,a_i}\,R_{x,\alpha_i} \qquad (3.10)\\
&= \begin{bmatrix} c_{\theta_i} & -s_{\theta_i} & 0 & 0 \\ s_{\theta_i} & c_{\theta_i} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c_{\alpha_i} & -s_{\alpha_i} & 0 \\ 0 & s_{\alpha_i} & c_{\alpha_i} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\\
&= \begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

Figure 2

```
function T = T_matrix(theta,alpha,a,d)
    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
```

Figure 3

```
DH TABLE:
     Theta         d         a         alpha
        90    +10.00     +0.00         +0.00
         0     +0.00    +40.00        +45.00
        45     +0.00    +30.00         +0.00
         0    +10.00     +0.00         +0.00
        60    +10.00     +0.00        +90.00
        20    +50.00    +20.00         +0.00
```

```
end_eff_position =
```

$$\begin{pmatrix} -14.15 \\ 55.28 \\ 47.48 \end{pmatrix}$$

```
roll = 1.6456
pitch = 0.5518
yaw = 2.9414
```

Figure 4 & 5

Inverse Kinematics (Analytical)

- To quantify the orientation of the end effector, we have implemented the Euler angles. Subsequently we have calculated the Euler angles as a function of the elements of the T0_6 transformation matrix.
- For inverse kinematics, we have used the non-linear least squares solver. Reasoning behind this choice, is that we had 12 equations and 6 unknowns. Solvers like fsolve and solve only take n equations for n variables. Thus, to account for the redundancy (multiple solutions), we used **lsqnonlin**.
- **Figures 6 & 7** are results from two different runs of the code. Essentially, we have randomly initialized the joint variables (theta1 to theta6). Depending on

the initializations, the solver might or might not get stuck at a local minima. The goal of course is to reach the global minima.

- Among the the several parameters that the solver **lsqnonlin** quantifies, we will expound on **Norm of Step** and **f(x)**. f(x) is the cost function, and from figures 6 & 7, we can see that f(x) and Norm of Step keep exponentially reducing as the iterations proceed.
- An interesting inference from these results is that, in the context of the cost function f(x), **figure 7 represents the local minimum and figure 6 represents the global minimum (Since f(x) is much smaller in figure 6)**. They're two separate runs, with different guesses.
- Thus, for every guess, the solver returns a different answer. Understandably so because the system has redundancies. In addition, the guess also determines the solver converging on a global or a local minima, which in turn changes the result, i.e. the end effector's position and orientation.

```
A =

    []

                                          Norm of      First-order
 Iteration  Func-count     f(x)            step          optimality
     0          7         5206.65                       2.58e+03
     1         14         5206.65              10       2.58e+03
     2         21         1739.11             2.5            967
     3         28         857.489             2.5       1.84e+03
     4         35         53.6298         1.29844            218
     5         42         5.88113        0.580915           70.7
     6         49         3.00291       0.0893158           1.29
     7         56         3.00001       0.0039516         0.0737
     8         63               3     0.000240324          0.008
     9         70               3     1.36593e-05       1.96e-05
    10         77               3     1.74058e-07       2.97e-07

Local minimum found.

Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.

<stopping criteria details>

A = 6x1                    Output joint
                           vairables
      9.7768
     22.4320
     13.8327
     -3.3204
      8.1544
    -12.0326
```

Figure 6

```
   130        917         14.1092       0.078125          4.8
   131        924         14.1031       0.078125          3.1
   132        931         14.0842       0.078125         4.67
   133        938         14.0842       0.078125         4.67
   134        945         14.0768      0.0195312         3.11
   135        952         14.0659      0.0195312         5.32
   136        959         14.0555      0.0390625         3.09
   137        966         14.0462       0.078125         5.68
   138        973         14.0319        0.15625          4.8
   139        980         14.0183        0.15625         2.83
   140        987         13.9782        0.15625         10.8
   141        994         13.9782         0.3125         10.8
   142       1001         13.9419       0.078125         2.54

Solver stopped prematurely.

lsqnonlin stopped because it exceeded the function evaluation limit,
options.MaxFunctionEvaluations = 1.000000e+03.

A = 6×4
      9.7768   37.7168   18.5411   24.8155
     22.4320  -45.4820  -45.1923   24.2573
     13.8327  -42.7072   82.0780  -69.5051
     -3.3204  -26.4600 -103.2482   -8.4156
      8.1544 -115.0568   -6.2306  -18.8367
    -12.0326   57.6131  -16.2227   46.4814
```

Joint variable output for this particular run

Figure 7

Inverse Kinematics (Jacobian)

- To perform inverse kinematics using the jacobian matrix, we employed the analytical jacobian, which consists of partial derivatives of the end effector positions with respect to the joint variables.
- **Figure 8 and 9** illustrate the methodology behind our code. Each increment to the old joint variable is essentially made of:
    - **Update factor** (usually 0.05 - 0.5, depending on how low an error you want)
    - **Inverse of the Jacobian**
    - **Delta_x** (This is essentially the difference between the desired and actual end effector positions. We initially randomize the actual end effector positions, following which we use the gradients from the jacobian to converge to the desired positions)
- In the code, we have plotted the **norm of detla_x** to better visualize if the system is actually converging. **Figure 10 & 11** contains the result.
- As seen from figure 9, detla_x tends to zero as the iterations proceed. In addition, the graph converges. Essentially this means that the "actual joint variables" are converging to a specific value, i.e. the desired values.

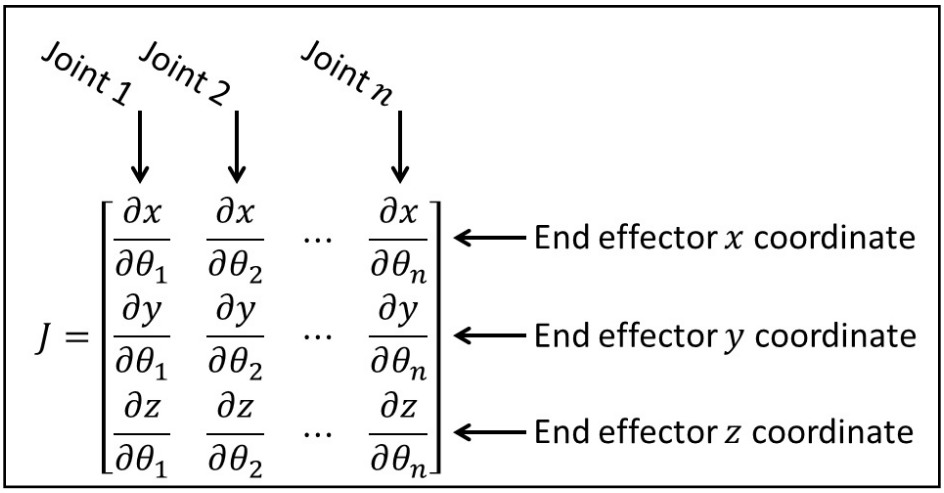

Figure 8

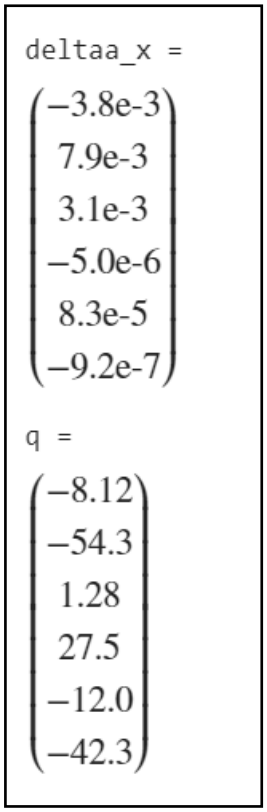

Figure 9

Start with a desired end-effector transform $\mathbf{T}_d$, and actual end-effector transform $\mathbf{T}_k$ calculated from forward kinematics at step k:

$$\mathbf{T}_d = \begin{bmatrix} \mathbf{R}_d & \mathbf{p}_d \\ \mathbf{0}_{1\times3} & 1 \end{bmatrix}, \qquad \mathbf{T}_k(\mathbf{q}) = \begin{bmatrix} \mathbf{R}_k & \mathbf{p}_k \\ \mathbf{0}_{1\times3} & 1 \end{bmatrix}$$

The difference in the desired and actual pose is:

$$\Delta\mathbf{x} = \begin{bmatrix} \mathbf{e}_p \\ \mathbf{e}_o \end{bmatrix}$$

Where:

$$\mathbf{e}_p = \mathbf{p}_d - \mathbf{p}_k$$
$$\mathbf{e}_o \leftarrow \mathbf{R}_e = \mathbf{R}_d\mathbf{R}_k^T$$

The orientation error $\mathbf{e}_o$ is expressed in RPY angles, extracted from rotation error $\mathbf{R}_e$.

From the forward kinematics:

$$\mathbf{x} = \mathbf{f}(\mathbf{q})$$
$$\Delta\mathbf{x} = (\partial\mathbf{f}/\partial\mathbf{q})\Delta\mathbf{q}$$
$$\Delta\mathbf{q} = (\partial\mathbf{f}/\partial\mathbf{q})^{-1}\Delta\mathbf{x}$$

Then increment the joint configuration $\mathbf{q}$ to cause the actual end-effector pose to converge on the desired pose:

$$\mathbf{q}_{k+1} = \mathbf{q}_k + \alpha\Delta\mathbf{q}_k$$
$$= \mathbf{q}_k + \alpha(\partial\mathbf{f}/\partial\mathbf{q})^{-1}\Delta\mathbf{x}_k$$

Where:

- $\alpha \in \mathbb{R}$ is a scalar
- $(\partial\mathbf{f}/\partial\mathbf{q})^{-1} \in \mathbb{R}^{n\times m}$ maps the pose error $\Delta\mathbf{x}$ from Cartesian space to joint space $\mathbf{q}$.

Figure 10

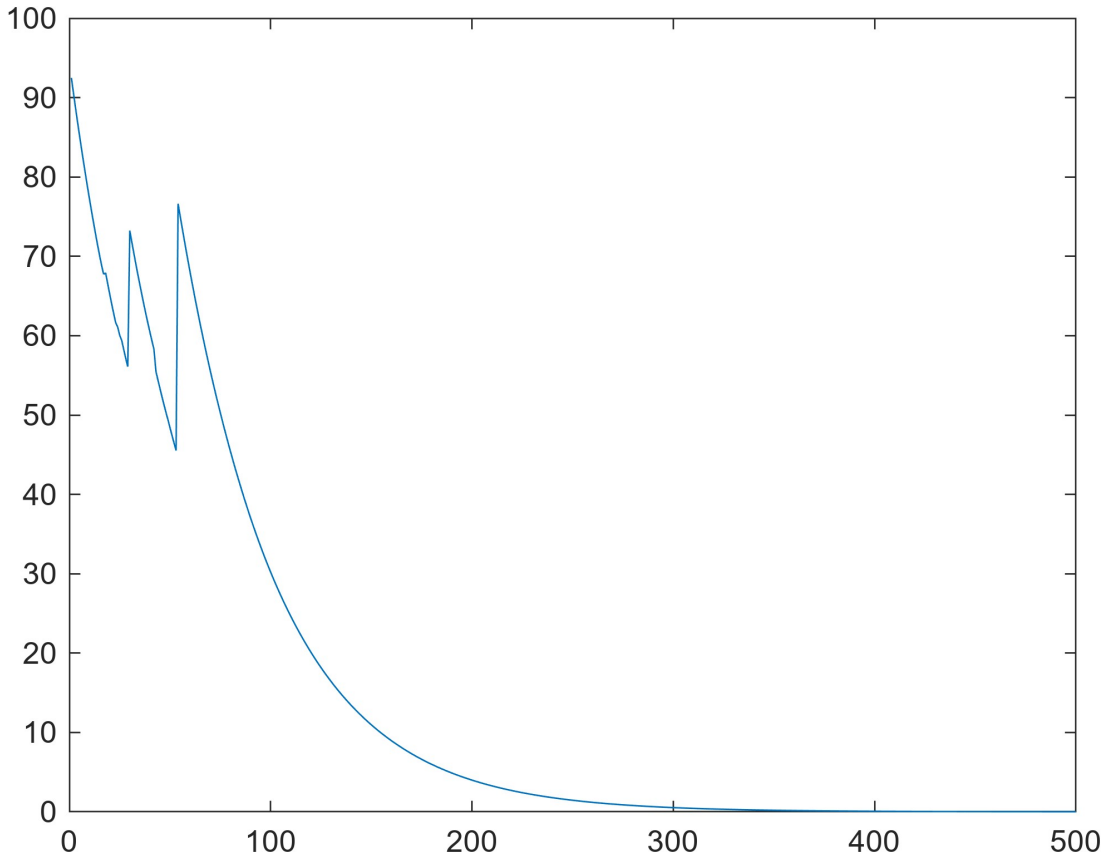

Figure 11

VISUALIZATION

- For the purpose of visualization, we have employed **rigidBodyTree.**
- This function helps us create rigid body links and joints. In addition, we needed a DH table, using which the function module would orient the subsequent links.
- A rigid body object called **body1** is initialized and a joint of **revolute** type is created which is attached to the rigid body. Similarly, six such revolute joints are created and attached successively.
- **Figure 12** is a snippet of how the joints have been initialized and **Figure 13** illustrates the links and joint axes.
- One thing to note is that, we had to divide the DH parameters **a** and **d** by 100 in order to clearly visualize the robot WITH the joint axes visible.

```
%VISUALSATION
robot = rigidBodyTree
body1 = rigidBody('body1');
jnt1 = rigidBodyJoint('jnt1','revolute');

setFixedTransform(jnt1,dhparams(1,:),'dh');
body1.Joint = jnt1;

addBody(robot,body1,'base')

body2 = rigidBody('body2');
jnt2 = rigidBodyJoint('jnt2','revolute');
body3 = rigidBody('body3');
jnt3 = rigidBodyJoint('jnt3','revolute');
body4 = rigidBody('body4');
jnt4 = rigidBodyJoint('jnt4','revolute');
body5 = rigidBody('body5');
jnt5 = rigidBodyJoint('jnt5','revolute');
body6 = rigidBody('body6');
jnt6 = rigidBodyJoint('jnt6','revolute');
```

Figure 12

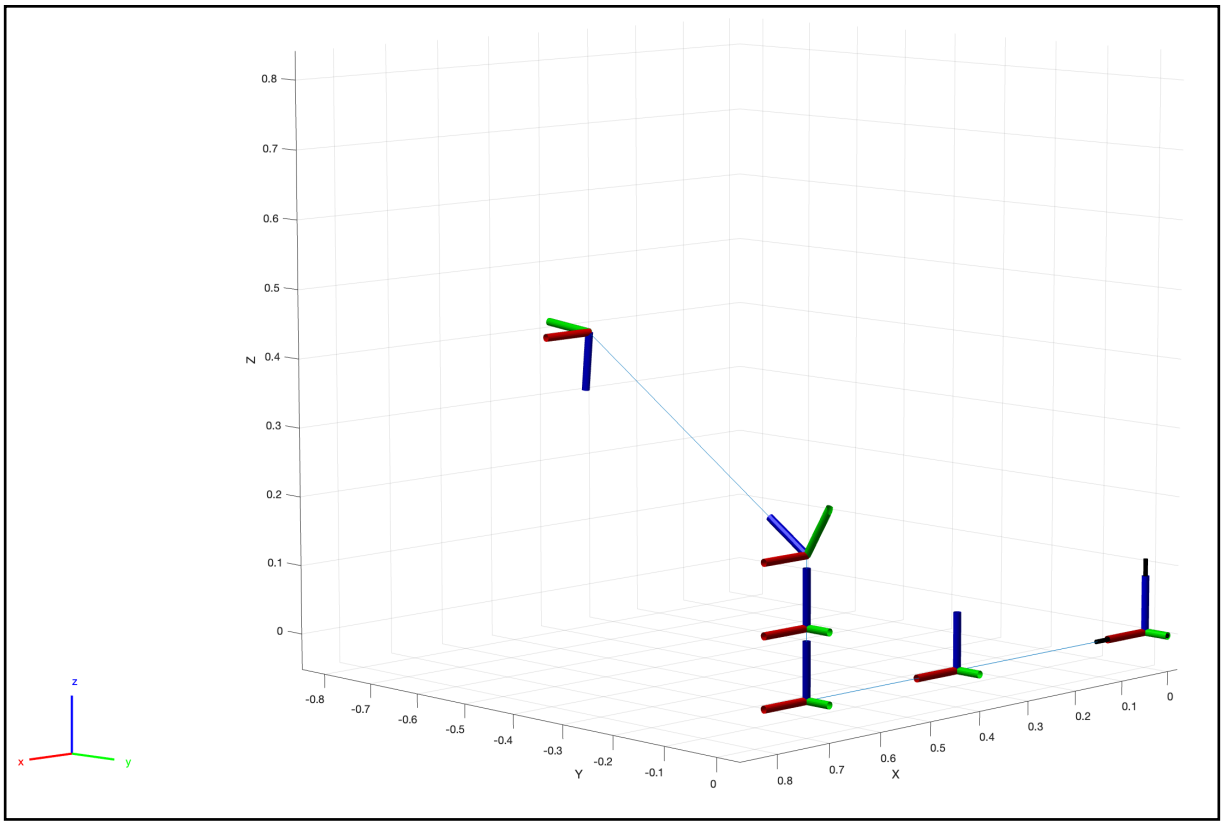

Figure 13

WORKSPACE

- In order to compute the workspace, we employed the monte-carlo method which uses randomness to solve problems that might be deterministic in principle.
- In our case, say our joint angle range is -180 to 180, and we compute angle sets every 10 degrees. This would mean we have 36 possible angles for each joint angle. Consequently, this would, for a 6R robot we have 36^6 combinations, i.e. approximately 2.1 Billion combinations.
- To obviate this issue, since we can make the state space (joint angle range) deterministic by choosing angles, every few degrees. Using monte-carlo we chose 5000 random combinations all across the angle set, and subsequently plot their tantamount robot orientation. **Figure 14** illustrates this.
- In this context, monte-carlo attempts to get good idea of the angle set ranges, without actually performing every single computation.

- In our code, we randomly initiate a theta value for a joint, between the joint range. We randomly pick 5000 different theta value for each joint and iterate through all of them. **Figure 15** illustrates the workspace.

```
clear all
syms theta1 theta2 theta3 theta4 theta5 theta6 ;
N = 5000 ;
theta1_min = -180 ; theta1_max = 180 ;
theta2_min = -180 ; theta2_max = 180 ;
theta3_min = -180 ; theta3_max = 180 ;
theta4_min = -180 ; theta4_max = 180 ;
theta5_min = -180 ; theta5_max = 180 ;
theta6_min = -180 ; theta6_max = 180 ;

t1 = theta1_min + (theta1_max-theta1_min)*rand(N,1);
t2 = theta2_min + (theta2_max-theta2_min)*rand(N,1);
t3 = theta3_min + (theta3_max-theta3_min)*rand(N,1);
t4 = theta4_min + (theta4_max-theta4_min)*rand(N,1);
t5 = theta5_min + (theta5_max-theta5_min)*rand(N,1);
t6 = theta6_min + (theta6_max-theta6_min)*rand(N,1);


alpha = [90 0 0 -90 90 0] ;
a = [0 40 30 0 0 0] ;
d = [0 0 0 10 10 50] ;
T0_1 = T_matrix(theta1,alpha(1),a(1),d(1)) ;
T1_2 = T_matrix(theta2,alpha(2),a(2),d(2)) ;
T2_3 = T_matrix(theta3,alpha(3),a(3),d(3)) ;
T3_4 = T_matrix(theta4,alpha(4),a(4),d(4)) ;
T4_5 = T_matrix(theta5,alpha(4),a(4),d(4)) ;
T5_6 = T_matrix(theta6,alpha(5),a(5),d(5)) ;
T0_6 = T0_1*T1_2*T2_3*T3_4*T4_5*T5_6 ;
```

Figure 14

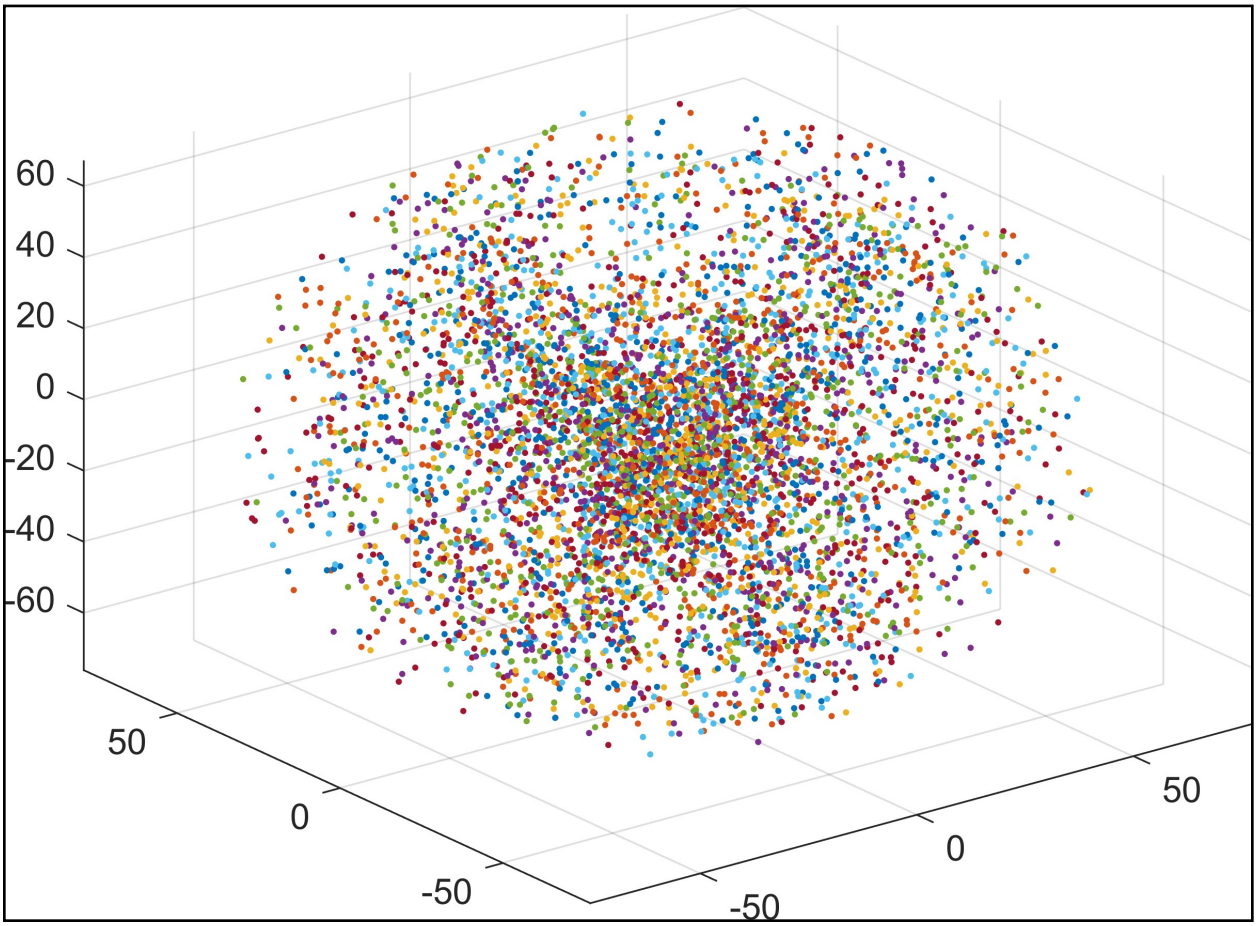

Figure 15

REFERENCES

[1] https://en.wikipedia.org/wiki/Forward_kinematics
[2] https://www.mathworks.com/help/robotics/ug/build-a-robot-step-by-step.html
[3]https://www.mathworks.com/help/robotics/ref/rigidbodytree.showdetails.html
[4]https://www.mathworks.com/help/matlab/ref/matlab.graphics.chart.primitive.line-properties.html
[5] https://github.com/Indushekhar/Modeling-and-Simulation-of-5-DOF-Robotic-manipulator/blob/master/ENPM667/ENPM667_Project.pdf
[6] https://www.mathworks.com/help/optim/ug/lsqnonlin.html
[7] https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjc2tCi74T8AhWIEmIAHQsKD9UQFnoECBIQAQ&url=https://www.mdpi.com/2076-3417/12/16/8330/pdf&usg=AOvVaw0HepUyFmcAQ1NhWwuEkK16
[8] https://homes.cs.washington.edu/~todorov/courses/cseP590/06_JacobianMethods.pdf
[9] https://modernrobotics.northwestern.edu/nu-gm-book-resource/6-2-numerical-inverse-kinematics-part-1-of-2/#department