Manipulator Kinematics and Robotics Project

Peiyuan Liao 廖培元
Institute of Automation, Chinese Academy of Sciences
July 31, 2017

Topics:

1. Modify the model of Puma560 by adding a prismatic link extending on the Z-axis as the {1}. Link 2-7 are adapted from the original model. A revolute link on frame 8 needs to be added with a displacement on the Z-axis to represent the end-effector. Achieve this manipulator in MATLAB with modified DH parameters and:

    (a) Create the animation in which eight links move respectively from angle 0 to 2π (prismatic link from -5 to 0) with a speed of 0.1.

    (b) Create the animation in which eight links move collectively.

    (c) Perform an analysis of inverse kinematics of the manipulator. Evaluate a solution for the inverse kinematics equation and give its respective transformation matrix.

2.Redo 1. (b) in Simulink. Achieve:

    (a) A trapezoidal velocity profile for the manipulator.

    (b) A polynomial velocity profile for the manipulator.

    Explain the Simulink model created and plot the movement of the end-effector in 3-D space along with the animation of the manipulator's movement.

3.Generate either a circular or a linear trajectory in Cartesian space for Puma560 and animate it in Simulink. Use a subsystem in the Simulink model and prove that the trajectory is linear/circular.

4.Model a walking robot in MATLAB by the function 'SerialLink.' Animate the walking movement of the robot.

5.Calculate the Jacobian matrix relative to frame {0} for Puma560 at the joint angle [0,0,0,0,0,0] and compare it to the result calculated from MATLAB.

1.

The original kinematics model of the Puma560 by Craig involves six rotational joints and no prismatic joints (2005, p79). Since two more links are added onto the original Puma560, now the manipulator has eight degrees of freedom. The first is a prismatic joint while the rest seven are rotational joints. A new model can be created by partially adapting the original code. Figure 1.1 shows the link parameters of the new manipulator. (P560")

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	d_1	0
2	0	0	0	θ_2
3	$-90°$	a_2	d_3	θ_3
4	0	a_2	d_4	θ_4
5	$-90°$	0	0	θ_5
6	$90°$	0	0	θ_6
7	$-90°$	0	0	θ_7
8	0	0	$d_{8\_}$	θ_8

Figure 1.1

The following is the code of the kinematics model for Puma560 in the Robotics Toolbox (Corke, n.d.).

```
clear L
deg = pi/180;

% joint angle limits from
% A combined optimization method for solving
the inverse kinematics problem...
% Wang & Chen
% IEEE Trans. RA 7(4) 1991 pp 489-
L(1) = Revolute('d', 0, 'a', 0, 'alpha',
pi/2, ...
    'I', [0, 0.35, 0, 0, 0, 0], ...
    'r', [0, 0, 0], ...
    'm', 0, ...
    'Jm', 200e-6, ...
    'G', -62.6111, ...
    'B', 1.48e-3, ...
    'Tc', [0.395 -0.435], ...
    'qlim', [-160 160]*deg );

L(2) = Revolute('d', 0, 'a', 0.4318,
'alpha', 0, ...
    'I', [0.13, 0.524, 0.539, 0, 0, 0], ...
    'r', [-0.3638, 0.006, 0.2275], ...
    'm', 17.4, ...
    'Jm', 200e-6, ...
    'G', 107.815, ...
    'B', .817e-3, ...
    'Tc', [0.126 -0.071], ...
    'qlim', [-45 225]*deg );

L(3) = Revolute('d', 0.15005, 'a', 0.0203,
'alpha', -pi/2,  ...
    'I', [0.066, 0.086, 0.0125, 0, 0,
0], ...
    'r', [-0.0203, -0.0141, 0.070], ...
    'm', 4.8, ...
    'Jm', 200e-6, ...
    'G', -53.7063, ...
    'B', 1.38e-3, ...
    'Tc', [0.132, -0.105], ...
    'qlim', [-225 45]*deg );

L(4) = Revolute('d', 0.4318, 'a', 0,
'alpha', pi/2,  ...
    'I', [1.8e-3, 1.3e-3, 1.8e-3, 0, 0,
0], ...
    'r', [0, 0.019, 0], ...
    'm', 0.82, ...
    'Jm', 33e-6, ...
    'G', 76.0364, ...
    'B', 71.2e-6, ...
    'Tc', [11.2e-3, -16.9e-3], ...
    'qlim', [-110 170]*deg);

L(5) = Revolute('d', 0, 'a', 0, 'alpha', -
pi/2,  ...
```

```
    'I', [0.3e-3, 0.4e-3, 0.3e-3, 0, 0,
0], ...
    'r', [0, 0, 0], ...
    'm', 0.34, ...
    'Jm', 33e-6, ...
    'G', 71.923, ...
    'B', 82.6e-6, ...
    'Tc', [9.26e-3, -14.5e-3], ...
    'qlim', [-100 100]*deg );


L(6) = Revolute('d', 0, 'a', 0, 'alpha',
0,  ...
    'I', [0.15e-3, 0.15e-3, 0.04e-3, 0, 0,
0], ...
    'r', [0, 0, 0.032], ...
    'm', 0.09, ...
    'Jm', 33e-6, ...
    'G', 76.686, ...
    'B', 36.7e-6, ...
    'Tc', [3.96e-3, -10.5e-3], ...
    'qlim', [-266 266]*deg );

p560 = SerialLink(L, 'name', 'Puma 560', ...
    'manufacturer', 'Unimation', 'comment',
'viscous friction; params of 8/95');
```

From the code, we can infer that the original kinematics model is using the standard Denavit–Hartenberg parameters. Hence, we need to convert it to modified DH parameters and adapt it to the new model. Furthermore, in order to control the movement of each joint, a new matrix (t) needs to be declared and its elements are needed to be embedded into the respective parameters of the links so that changing these elements can result in a different transformation matrix. The following is the source code:

```
clear L
deg = pi/180;
t=[0 0 0 0 0 0 0 0];

L(1) =
Link([0,0,0,0,1],'modified');
L(1).qlim=[-.5 .5]
L(1).d=t(1)

L(2) = Revolute('d', 0, 'a', 0,
'alpha', -pi/2, ...
    'I', [0, 0.35, 0, 0, 0, 0], ...
    'r', [0, 0, 0], ...
    'm', 0, ...
    'Jm', 200e-6, ...
    'G', -62.6111, ...
    'B', 1.48e-3, ...
    'Tc', [0.395 -0.435], ...
    'qlim', [-160
160]*deg,'modified' );

L(3) = Revolute('d', 0, 'a',
0.4318, 'alpha', 0, ...
    'I', [0.13, 0.524, 0.539, 0, 0,
0], ...
    'r', [-0.3638, 0.006,
0.2275], ...
    'm', 17.4, ...
    'Jm', 200e-6, ...
    'G', 107.815, ...
    'B', .817e-3, ...
    'Tc', [0.126 -0.071], ...
    'qlim', [-45
225]*deg,'modified' );

L(4) = Revolute('d', 0.15005, 'a',
0.0203, 'alpha', -pi/2,  ...
    'I', [0.066, 0.086, 0.0125, 0,
0, 0], ...
    'r', [-0.0203, -0.0141,
0.070], ...
    'm', 4.8, ...
    'Jm', 200e-6, ...
    'G', -53.7063, ...
    'B', 1.38e-3, ...
    'Tc', [0.132, -0.105], ...
    'qlim', [-225
45]*deg,'modified' );

L(5) = Revolute('d', 0.4318, 'a',
0, 'alpha', pi/2,  ...
    'I', [1.8e-3, 1.3e-3, 1.8e-3,
0, 0, 0], ...
    'r', [0, 0.019, 0], ...
    'm', 0.82, ...
    'Jm', 33e-6, ...
    'G', 76.0364, ...
    'B', 71.2e-6, ...
    'Tc', [11.2e-3, -16.9e-3], ...
    'qlim', [-110
170]*deg,'modified');

L(6) = Revolute('d', 0, 'a', 0,
'alpha', -pi/2,  ...
    'I', [0.3e-3, 0.4e-3, 0.3e-3,
0, 0, 0], ...
    'r', [0, 0, 0], ...
    'm', 0.34, ...
    'Jm', 33e-6, ...
    'G', 71.923, ...
    'B', 82.6e-6, ...
    'Tc', [9.26e-3, -14.5e-3], ...
```

```
    'qlim', [-100
100]*deg,'modified' );


L(7) = Revolute('d', 0, 'a', 0,
'alpha', 0,  ...
    'I', [0.15e-3, 0.15e-3, 0.04e-
3, 0, 0, 0], ...
    'r', [0, 0, 0.032], ...
    'm', 0.09, ...
    'Jm', 33e-6, ...
    'G', 76.686, ...
    'B', 36.7e-6, ...
    'Tc', [3.96e-3, -10.5e-3], ...
```

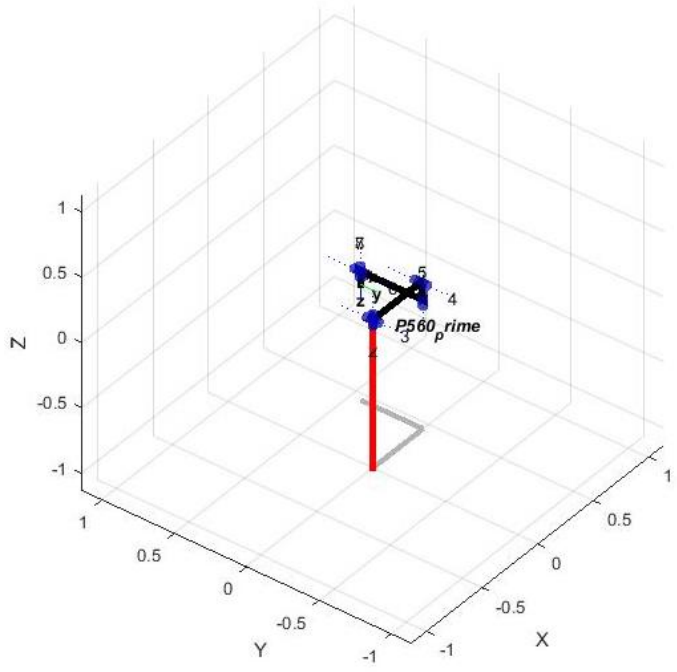
```
    'qlim', [-266
266]*deg ,'modified');

L(8) =
Link([0,0,0,0,0],'modified');
L(8).d=0.1
for i=2:7
    L(i).theta=t(i)
end
clear i

p560 = SerialLink(L, 'name',
'P560"');
```

Figure 1.2 is a graphical representation of the modified manipulator in its starting position.

Figure 1.2



(a) The animation of links for a manipulator in MATLAB relies on two functions for the class 'SerialLink' in the Robotics Toolbox: 'fkine' and 'plot.' 'fkine' is the abbreviation for 'Forward Kinematics.' The function takes in joint angles/prismatic joint movements and gives out a transformation matrix to which the manipulator will move if the angles and displacements are applied to the joints. The 'plot' function then represents this transformation through a 3-D figure. So, if all other elements in the t matrix, aka the matrix used to control the movement/rotation of the joints, remains constant while only one is constantly changing, and a 'plot' function is applied each time the element in t changes, the rotation/movement of a link has just been achieved. If all eight joint angles are altered one at a time with the other held constant, the links will appear to be moving respectively. The following is the source code:

```
t1=t(1)
```

```
t2=t(2)
```

```
t3=t(3)
t4=t(4)
t5=t(5)
t6=t(6)
t7=t(7)
t8=t(8)

for t1=-0.5:0.1:0.5
    t(1)=t1
    p560.fkine(t);
    p560.plot(t);
end

for t2=0:0.1:pi
    t(2)=t2
    p560.fkine(t);
    p560.plot(t);
end

for t3=0:0.1:pi
    t(3)=t3
    p560.fkine(t);
    p560.plot(t);
end

for t4=0:0.1:pi
    t(4)=t4
    p560.fkine(t);
    p560.plot(t);
end

for t5=0:0.1:pi
    t(5)=t5
    p560.fkine(t);
    p560.plot(t);
end

for t6=0:0.1:pi
    t(6)=t6
    p560.fkine(t);
    p560.plot(t);
end

for t7=0:0.1:pi
    t(7)=t7
    p560.fkine(t);
    p560.plot(t);
end

for t8=0:0.1:pi
    t(8)=t8
    p560.fkine(t);
    p560.plot(t);
end
```

(b) Similarly, the synchronized movement of all eight links can be achieved by altering all elements in the t matrix at the same time.

```
t1=t(1)
t2=t(2)
t3=t(3)
t4=t(4)
t5=t(5)
t6=t(6)
t7=t(7)
t8=t(8)

for t2=0:0.1:pi
    for he=2:8
        t(he)=t2
    end
    clear he
    t(1)=t2/(2*pi)
    p560.fkine(t);
    p560.plot(t);
end
```

(c) The inverse manipulator kinematics deals with the opposite problem that direct kinematics tackles. Instead of turning the transformation matrix into joint angles, the inverse kinematics solve for possible joint angles base on a given transformation matrix (Craig, 2005, p. 101). This means that for some matrices the inverse kinematics equation is solvable while others have more than one solutions. The

basic idea behind what is solvable and what's not is that each manipulator has a specific workspace. If the transformation matrix is located in the span of the workspace of the manipulator, the inverse kinematic equation will have at least one solution.

Hence the transformation matrix the forward kinematics yields must have at least one corresponding inverse kinematics solution. Utilizing this feature, we can animate the movement of the manipulator in MATLAB. The following is the source code:

```
T1=p560.fkine([.4,-pi/4,-pi/2,0,0,0,0,0])
inv_kin=p560.ikine(T1);
a=mtraj(@lspb,t,inv_kin,25);

for k=1:8
    for i=1:size(a)
        t(k)=a(i,k)
        p560.fkine(t)
        p560.plot(t)
    end
end
```

In this example, the transformation matrix is $\begin{bmatrix} \frac{-\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} & 0.4678 \\ 0 & 0 & 0 & 0.4318 \\ \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} & 0.8965 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, and the corresponding solution is $\begin{bmatrix} 0.4 \\ -\frac{\pi}{4} \\ -\frac{\pi}{2} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$.

2.

Both the trapezoidal speed profile and the polynomial speed profile refers to the graph of the speed/distance of each effector when plotting against time. In trapezoidal speed profile, the graph of effector speed against time takes the shape of a trapezoid. In polynomial speed profile, the graph of effector displacement against time is approached by a fifth-degree polynomial. The advantage of polynomial speed profile is that overall the derivatives of acceleration against time is smaller than that of the trapezoidal speed profile. But since at the middle of the graph the trapezoidal speed profile the acceleration of the effector is zero, it enjoys a greater degree of industrial application. Figure 2.1 is the graphical representation of polynomial speed profile while Figure 2.2 is the trapezoidal speed profile.

The Robotics Toolbox only includes the subsystems for single-link trajectory generation and multi-link linear trajectories generation, so a new subsystem must be created from existing templates. The initialization code for the trapezoidal speed profile is as follows:

```
n = length(q0);
if length(q0) ~= length(qf) ,
 error('q0 and qf must be same length')
end
t=[0:100]'/100*tmax;
[q,qd,qdd] = mtraj(@lspb,q0,qf,t)
```

If "@lspb" is replaced by "@tpoly," polynomial speed profile can also be achieved.

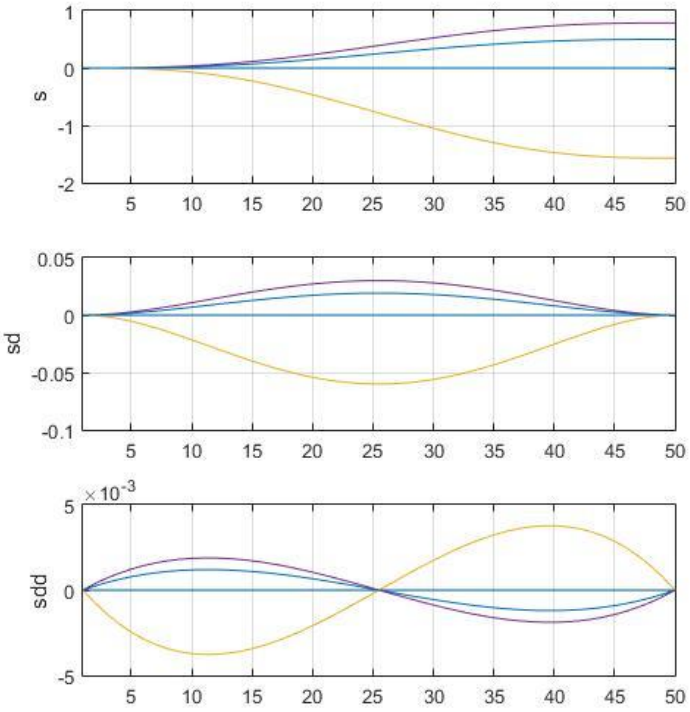


Figure 2.1

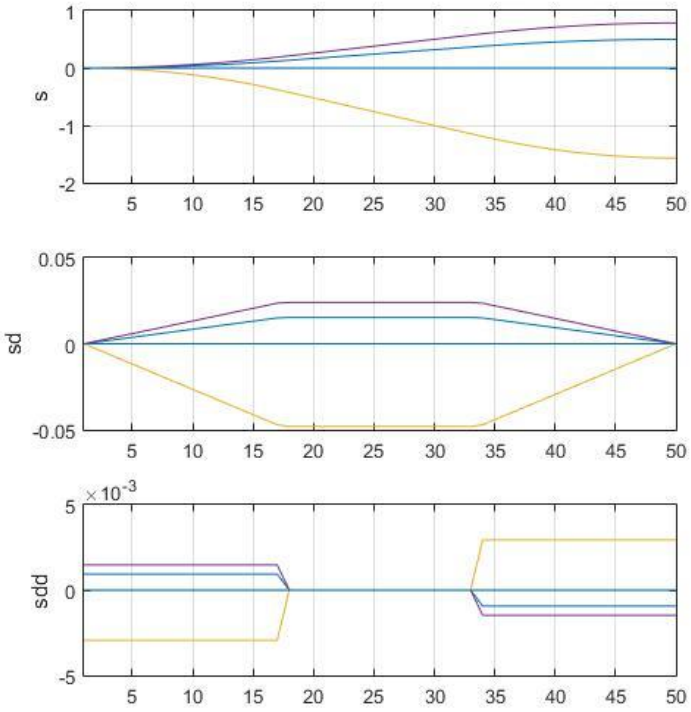


Figure 2.2

Figure 2.3 and 2.4 is the model simulation for the two profiles, 2.3 for polynomial, 2.4 for trapezoidal.

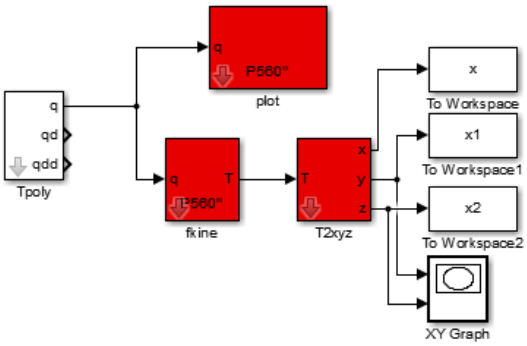


Figure 2.3

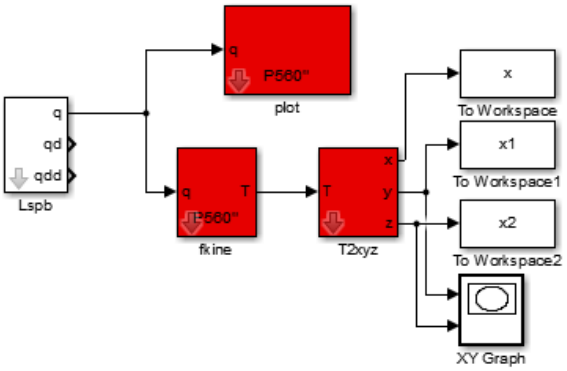


Figure 2.4

In the model, the "Tpoly"/"Lspb" subsystem generates the trajectories for all eight links and transform them into signals in *Timeseries* form. "plot" subsystem generates the animation in which the motion of effectors could be illustrated. The "fkine" and "T2xyz" subsystems then transform the signal back into three arrays that represent the movement of the end-effector. Execute the "plot3" function back in the MATLAB workspace, and the movement of the end-effector will be plotted in a blue line. Figure 2.5 shows the P560" model moving in polynomial speed profile at its final position with the blue line representing the path end-effector has traveled.

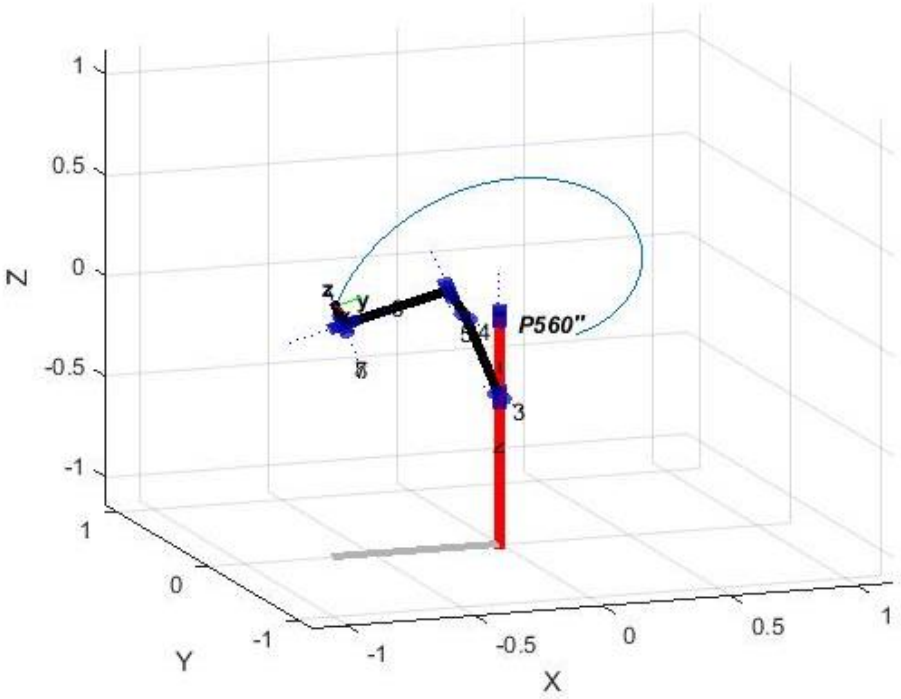


Figure 2.5

In the animations, the parameters for the manipulator are respectively $\begin{bmatrix} -0.4 \\ -\frac{3\pi}{5} \\ -\frac{\pi}{2} \\ \frac{\pi}{2} \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ for polynomial and $\begin{bmatrix} -0.4 \\ -\frac{\pi}{2} \\ -\frac{\pi}{2} \\ \frac{\pi}{2} \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ for trapezoidal.

3.

The difficulties at generating circular/linear trajectories in Cartesian space is that the Jacobian transformation from joint velocity to Cartesian velocity is only locally linear. This means that in a larger scale, a linear increase in the joint angles often does not yield linear changes in the transformation matrix. So, in order to utilize the inverse kinematics function in MATLAB to generate a circular trajectory in Cartesian space, the trajectory needs to be divided into smaller, equal-sized fractions to approach the infinitely-small shift ($\mathrm{d}y_i, i = 1,2,3,4,5,6$) the Jacobian Matrix requires.

```
transf=0:0.2:3
theta1=[pi/8,0,0,0,0,0]
T1=p560.fkine([pi/8,0,0,0,0,0])
T2=p560.fkine([pi/16,0,0,0,0,0])
TD=ctraj(T1,T2,transf)
TS=[]
solv=[]
TD(:,:,1)
solve1=p560.ikine(TD(:,:,1),theta1)
TS=[TS;solve1]
for i=2:15
    TD(:,:,i)
    solv=p560.ikine(TD(:,:,i),solv,'pinv')
    TS =[TS;solv]
end
```

In the code, the trajectory starts from the position $\begin{bmatrix} 0.9239 & 0 & -0.3827 & 0.3603 \\ 0 & -1 & 0 & 0.4318 \\ -0.3827 & 0 & -0.9239 & -0.3116 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, represented in transformation matrix, and moves to the position $\begin{bmatrix} 0.9808 & 0 & -0.1951 & 0.4141 \\ 0 & -1 & 0 & 0.4318 \\ -0.1951 & 0 & -0.9808 & 0.2354 \\ 0 & 0 & 0 & 1 \end{bmatrix}$. Now, if the trajectory is generated directly from the two transformation matrices, the path end-effector moves will hardly be linear due to the nonlinear nature of the Jacobian matrix. So, the two matrices are interpolated by 13 more, making the total number of movements 15. Since now the distances between two frames represented by transformation matrices are reduced to 1/15 of its original size, the resulting movement of the end-effector will appear to be linear.

```
inte=0.1:0.1:1.5
inte=inte'
FE=horzcat(inte,TS)
```

The next step is to transform the joint-angle matrix into Simulink signals. A Source block needs to be created to import the variable FE from the workspace. Then, since the new transformation is not strictly linear and extrapolation might exceed the workspace of the manipulator, the "Holding final value" option must be selected instead of "Extrapolation." The rest of the Simulink model is comparable to those of the model in Topic 2. Figure 3.1 illustrates the similarities.

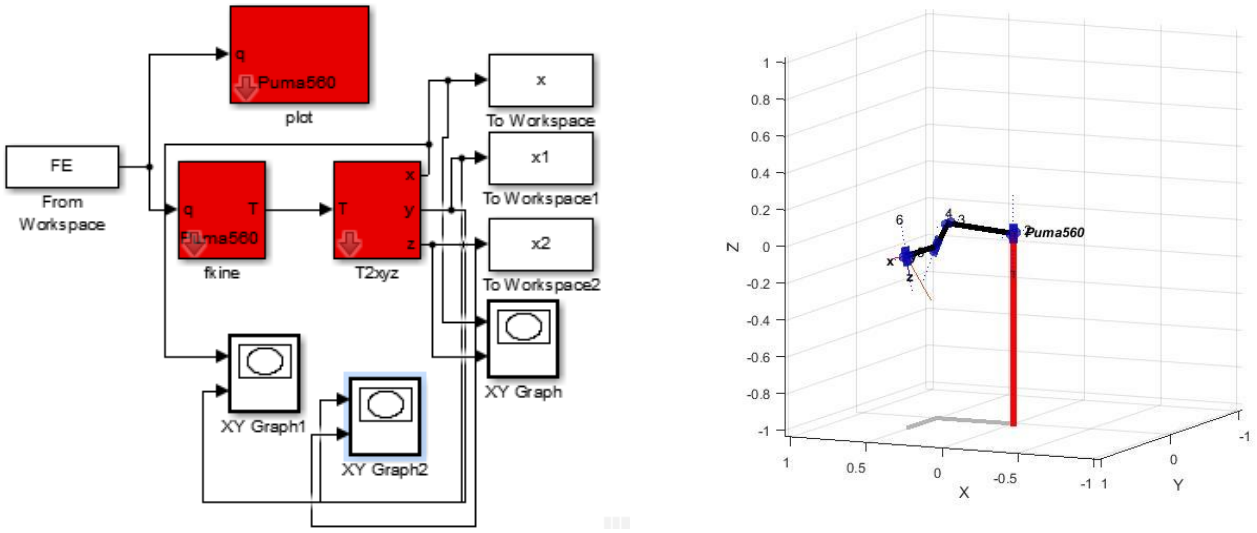


Figure 3.2

The red line in Figure 3.2 shows the movement of the end-effector in the manipulator. We can loosely see that it is linear.

Since now the data of x, y and z value are stored in variables in the workspace in MATLAB, they could be relocated for linear regression study. From Figure 3.3, we can see that the path is truly linear in 3-dimensional space.

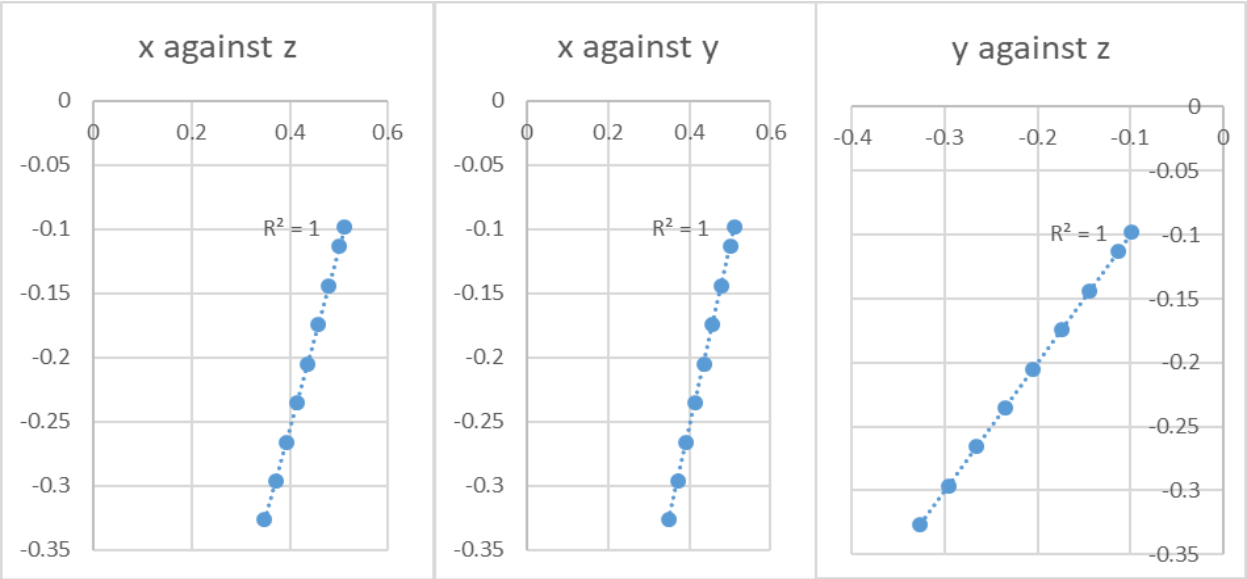


Figure 3.3

4.

According to Corke, a walking robot can be modeled by four manipulators each having revolute links that can simulate the movement of a leg(n.d.). When considered by both modified and standard D-H conventions, there exist two ways of summarizing the parameters of each leg. Figure 4.1 is the link parameters of a single leg created by standard D-H convention while Figure 4.2 belongs to modified D-H convention.

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	0	θ_1
2	0	0	d_2	θ_2
3	0	0	$-d_3$	θ_3

$\theta_1 = 90°, d_2 > 0, d_3 > 0, \theta_2 = 0°, \theta_3 = 90°$

Figure 4.1

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	90°	0	0	θ_1
2	0	0	0	θ_2
3	0	0	$-d_2$	θ_3
4	0	0	d_3	θ_4

Figure 4.2

Although an extra link is used in modified D-H convention, the result of altering θ_{1-3} remains the same for both conventions. This means that the solutions for inverse kinematics computed from standard model could also be used in the modified model, with $\theta_4 = 0°$ in all occasions. This provides us an approach different from that of Corke, where signals are generated from the standard model and modified to be used in the modified model(n.d.). The following is the source code for the generation of signals.

```
clear all
% Author:Alexander Liao Kent'20
%-----------Signal Generation Based on
Standard DH parameters-----------%
Jt2=0.1;
Jt3=0.1;

SignalL(1) = Link([    0       0    0
pi/2   ], 'standard');
SignalL(2) = Link([    0       0    Jt2
0    ], 'standard');
SignalL(3) = Link([    0       0    -
Jt3   0    ], 'standard');

SignalLeg=SerialLink(SignalL,'name','Le
g','offset',[pi/2 0 pi/2]);

x_forward = -5; x_backward = -
x_forward;
y_axis = 5;
z_upward = -2; z_downward = -5;

displacement = [x_forward y_axis
z_downward; x_backward y_axis
z_downward; x_backward y_axis z_upward;
x_forward y_axis z_upward] * 0.01;

displacement = [displacement;
displacement];
t_traj = [3 0.25 0.5 0.25]';
t_traj = [1; t_traj; t_traj;];
trajectory=mstraj(displacement, [],
t_traj, displacement(1,:), 0.03, 0.01);


traj_cycle = trajectory(32:161,:);
walk_cycle =
SignalLeg.ikine(transl(traj_cycle), [],
[1 1 1 0 0 0],'pinv' );

width = 0.1;
length = 0.2;

k=1
leg1signal=[];
leg2signal=[];
leg3signal=[];
leg4signal=[];

while k<1001
     k=k+1;
    i1=gait(walk_cycle, k, 0,   0);
    i2=gait(walk_cycle, k, 100,   0);
    i3=gait(walk_cycle, k, 200,   0);
    i4=gait(walk_cycle, k, 300,   0);
    leg4signal=vertcat(leg4signal,i4);
    leg3signal=vertcat(leg3signal,i3);
    leg2signal=vertcat(leg2signal,i2);
    leg1signal=vertcat(leg1signal,i1);
end

t_indicator=0.01:0.01:10;
t_indicator=t_indicator'
leg4signal=horzcat(t_indicator,leg4sign
al);
leg3signal=horzcat(t_indicator,leg3sign
al);
leg2signal=horzcat(t_indicator,leg2sign
al);
leg1signal=horzcat(t_indicator,leg1sign
al);

zero_s=zeros(1000,1);
leg4signal=horzcat(leg4signal,zero_s);
leg3signal=horzcat(leg3signal,zero_s);
leg2signal=horzcat(leg2signal,zero_s);
leg1signal=horzcat(leg1signal,zero_s);
```

Implementing the methods of Corke from line 13-26, the trajectories of the three links can be planned by using the function “mstraj” (n.d.). The rest of the code is dedicated to convert the signal acceptable by the standard model by adding time indicators and signals for the fourth link (which is zero all the time).

```
%-----------Modeling Based on Modified
DH parameters-----------%

L(1) = Link([    pi/2    0    0
0   ], 'modified');
L(2) = Link([    0       0    0
pi/2  ], 'modified');
L(3) = Link([    0       0    Jt2
0   ], 'modified');
L(4) = Link([    0       0    Jt3
0   ], 'modified');

Leg=SerialLink(L,'name','Leg','offset',
[pi/2 0 -pi/2 0]);

options = Leg.plot({'noraise',
'nobase', 'noshadow'});

Legs(1) = SerialLink(Leg, 'name',
'Leg1');
Legs(2) = SerialLink(Leg, 'name',
'Leg2', 'base', transl(-length, 0, 0));
Legs(3) = SerialLink(Leg, 'name',
'Leg3', 'base', transl(-length, -width,
0)*trotz(pi));
Legs(4) = SerialLink(Leg, 'name',
'Leg4', 'base', transl(0, -width,
0)*trotz(pi));
```

```
hold on

patch([0 -length -length 0], [0 0 -
width -width], [0 0 0 0], ...
    'FaceColor', 'b', 'FaceAlpha', .2)
```

```
for i=1:4
    Legs(i).plot([0 0 0 0],options);
end

hold off
```

The code above creates the model created from modified D-H convention, which is the model to be used for animation. The next step involves creating the Simulink model needed to process the signals and turn them into animations. Figure 4.3 illustrates the model. Since extrapolation is not applicable to the plotting of the signals, the final value of the input should be held constant if the animation is not over. The creation of "walkingplot" is a variant of the "slplotbot" function by Corke, and below is the source code for the function used in this model(n.d.):

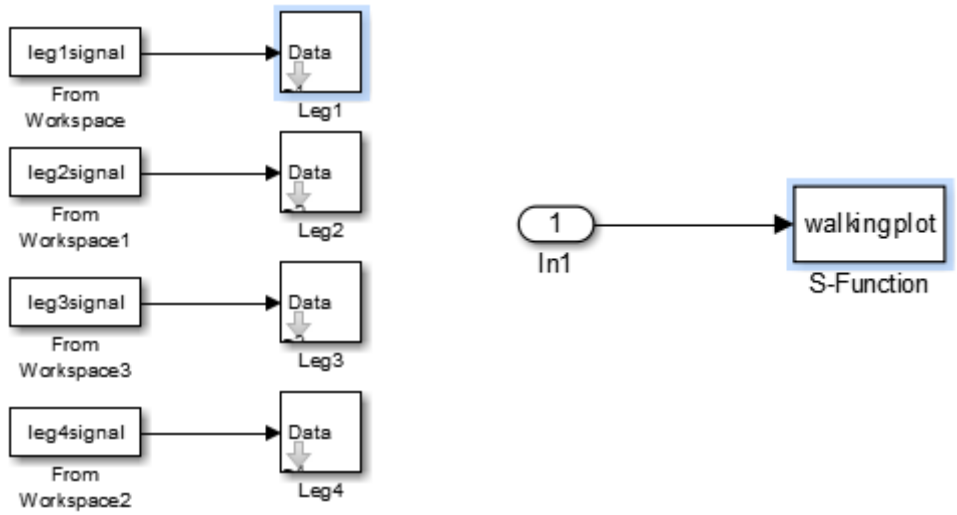


Figure 4.3

```
function [sys,x0,str,ts] =
walkingplot(t,x,u,flag, robot, fps,
holdplot)
    switch flag,

    case 0
        % initialize the robot graphics
        [sys,x0,str,ts] =
mdlInitializeSizes(fps);  % Init
        if ~isempty(robot)
            axis([-0.3 0.1 -0.2 0.1 -0.05
0.1]); set(gca)
```

```
            robot.plot(zeros(1, robot.n),
'delay', 0, 'noraise', 'nobase',
'noshadow', ...
    'nowrist', 'nojaxes')
        end
        if holdplot
            hold on
        end

    case 3
        % come here on update
        if ~isempty(u)
            %fprintf('--slplotbot: t=%f\n',
t);
            axis([-0.3 0.1 -0.2 0.1 -0.05
0.1]); set(gca)
```

```
            robot.animate(u');
            drawnow
        end
        ret = [];
    case {1, 2, 4, 9}
        ret = [];
    end
%
%=============================================
==================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and
sample times for the S-function.
%=============================================
==================================
%
function
[sys,x0,str,ts]=mdlInitializeSizes(fps)

%
% call simsizes for a sizes structure, fill
it in and convert it to a
% sizes array.
%
% Note that in this example, the values are
hard coded.  This is not a
% recommended practice as the
characteristics of the block are typically
% defined by the S-function parameters.
%
sizes = simsizes;

sizes.NumContStates  = 0;
sizes.NumDiscStates  = 0;
sizes.NumOutputs     = 0;
sizes.NumInputs      = -1;
sizes.DirFeedthrough = 1;
sizes.NumSampleTimes = 1;   % at least one
sample time is needed

sys = simsizes(sizes);

%
% initialize the initial conditions
%
x0  = [];

%
% str is always an empty matrix
%
str = [];

%
% initialize the array of sample times
%
ts  = [-1 0]; %[1.0/fps];

% end mdlInitializeSizes
```

Figure 4.4 is the robot at its stopping position.

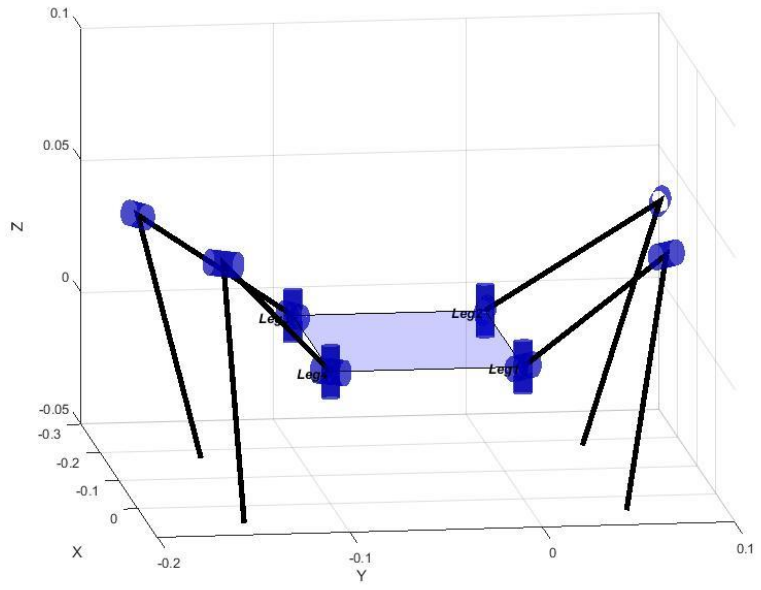


Figure 4.4

5.

The Jacobian matrix in a mathematical sense is a matrix containing all combinations of partial derivatives of the 6×1 Cartesian speed vector ${}_0^i v$ (i depends on the joint the matrix is describing), organized in a manner that when the matrix multiplies the differentiated vector containing joint angle velocities, the result is the Cartesian speed vector itself. In the case of Puma560, since there are six joint angles, the Jacobian matrix can be illustrated in Figure 5.1:

$$\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_6} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_6}{\partial x_1} & \cdots & \frac{\partial f_6}{\partial x_6} \end{bmatrix}$$

Figure 5.1

According to the method of Craig, the Cartesian speed vector could be calculated by iterating the linear velocity vector ν as well as the rotational velocity ω and stacking them together in the end. Below is the process of calculation as well as formulae for iteration (n.d.). The profile of Puma560 is taken from Croke's *mdl_puma560* file(n.d.). Ans since the initial joint angles are known, we can calculate every iteration of ω and ν.

$${}_{i+1}^{i+1}\omega = {}_i^{i+1}R\,{}_i^i\omega + \dot{\theta}_{i+1}\,{}_{i+1}^{i+1}\hat{Z}$$

$${}_{i+1}^{i+1}\nu = {}_i^{i+1}R({}_i^i\nu + {}_i^i\omega \times {}_{i+1}^{i}P)$$

$${}_1^1\omega = \begin{bmatrix} 0 \\ \dot{\theta}_1 \\ 0 \end{bmatrix}$$

$${}_1^1\nu = \begin{bmatrix} 0 \\ \dot{\theta}_1 \\ 0 \end{bmatrix}$$

$${}_2^2\omega = \begin{bmatrix} 0 \\ \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix}$$

$${}_2^2\nu = \begin{bmatrix} 0 \\ 0.4318\dot{\theta}_2 \\ -0.4318\dot{\theta}_1 \end{bmatrix}$$

$${}_3^3\omega = \begin{bmatrix} 0 \\ -\dot{\theta}_2 - \dot{\theta}_3 \\ \dot{\theta}_1 \end{bmatrix}$$

$${}_3^3\nu = \begin{bmatrix} 0.15005\dot{\theta}_1 \\ 0.4521\dot{\theta}_1 \\ 0.4521\dot{\theta}_2 + 0.0203\dot{\theta}_3 \end{bmatrix}$$

$${}_4^4\omega = \begin{bmatrix} 0 \\ \dot{\theta}_1 + \dot{\theta}_4 \\ \dot{\theta}_2 + \dot{\theta}_3 \end{bmatrix}$$

$${}_4^4\nu = \begin{bmatrix} 0.15005\dot{\theta}_1 - 0.4318\dot{\theta}_2 - 0.4318\dot{\theta}_3 \\ 0.4521\dot{\theta}_2 + 0.0203\dot{\theta}_3 \\ -0.4521\dot{\theta}_1 \end{bmatrix}$$

$${}_5^5\omega = \begin{bmatrix} 0 \\ -\dot{\theta}_2 - \dot{\theta}_3 - \dot{\theta}_5 \\ \dot{\theta}_1 + \dot{\theta}_4 \end{bmatrix}$$

$${}_5^5\nu = \begin{bmatrix} 0.15005\dot{\theta}_1 - 0.4318\dot{\theta}_2 - 0.4318\dot{\theta}_3 \\ 0.4521\dot{\theta}_1 \\ 0.4521\dot{\theta}_2 + 0.0203\dot{\theta}_3 \end{bmatrix}$$

$$ {}^{6}_{6}\omega = \begin{bmatrix} 0 \\ -\dot{\theta}_2 - \dot{\theta}_3 - \dot{\theta}_5 \\ \dot{\theta}_1 + \dot{\theta}_4 + \dot{\theta}_6 \end{bmatrix} $$

$$ {}^{6}_{6}\nu = \begin{bmatrix} 0.15005\dot{\theta}_1 - 0.4318\dot{\theta}_2 - 0.4318\dot{\theta}_3 \\ 0.4521\dot{\theta}_1 \\ 0.4521\dot{\theta}_2 + 0.0203\dot{\theta}_3 \end{bmatrix} $$

$$ {}^{6}_{0}R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{6}_{0}\nu = {}^{6}_{0}R\,{}^{6}_{6}\nu = \begin{bmatrix} 0.15005\dot{\theta}_1 - 0.4318\dot{\theta}_2 - 0.4318\dot{\theta}_3 \\ 0.4521\dot{\theta}_1 \\ 0.4521\dot{\theta}_2 + 0.0203\dot{\theta}_3 \end{bmatrix} $$

$$ {}^{6}_{0}\omega = {}^{6}_{0}R\,{}^{6}_{6}\omega = \begin{bmatrix} 0 \\ -\dot{\theta}_2 - \dot{\theta}_3 - \dot{\theta}_5 \\ \dot{\theta}_1 + \dot{\theta}_4 + \dot{\theta}_6 \end{bmatrix} $$

$$ {}^{6}_{0}v = \begin{bmatrix} {}^{6}_{0}\nu \\ {}^{6}_{0}\omega \end{bmatrix} = \begin{bmatrix} 0.15005\dot{\theta}_1 - 0.4318\dot{\theta}_2 - 0.4318\dot{\theta}_3 \\ 0.4521\dot{\theta}_1 \\ 0.4521\dot{\theta}_2 + 0.0203\dot{\theta}_3 \\ 0 \\ -\dot{\theta}_2 - \dot{\theta}_3 - \dot{\theta}_5 \\ \dot{\theta}_1 + \dot{\theta}_4 + \dot{\theta}_6 \end{bmatrix} $$

∴

$$ \begin{cases} f_1(\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \dot{\theta}_4, \dot{\theta}_5, \dot{\theta}_6) = 0.15005\dot{\theta}_1 - 0.4318\dot{\theta}_2 - 0.4318\dot{\theta}_3 \\ f_2(\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \dot{\theta}_4, \dot{\theta}_5, \dot{\theta}_6) = 0.4521\dot{\theta}_1 \\ f_3(\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \dot{\theta}_4, \dot{\theta}_5, \dot{\theta}_6) = 0.4521\dot{\theta}_2 + 0.0203\dot{\theta}_3 \\ f_4(\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \dot{\theta}_4, \dot{\theta}_5, \dot{\theta}_6) = 0 \\ f_5(\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \dot{\theta}_4, \dot{\theta}_5, \dot{\theta}_6) = -\dot{\theta}_2 - \dot{\theta}_3 - \dot{\theta}_5 \\ f_6(\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \dot{\theta}_4, \dot{\theta}_5, \dot{\theta}_6) = \dot{\theta}_1 + \dot{\theta}_4 + \dot{\theta}_6 \end{cases} $$

$$ {}^{6}_{0}J = \begin{bmatrix} \frac{\partial f_1}{\partial \dot{\theta}_1} & \frac{\partial f_1}{\partial \dot{\theta}_2} & \frac{\partial f_1}{\partial \dot{\theta}_3} & \frac{\partial f_1}{\partial \dot{\theta}_4} & \frac{\partial f_1}{\partial \dot{\theta}_5} & \frac{\partial f_1}{\partial \dot{\theta}_6} \\ \frac{\partial f_2}{\partial \dot{\theta}_1} & \frac{\partial f_2}{\partial \dot{\theta}_2} & \frac{\partial f_2}{\partial \dot{\theta}_3} & \frac{\partial f_2}{\partial \dot{\theta}_4} & \frac{\partial f_2}{\partial \dot{\theta}_5} & \frac{\partial f_2}{\partial \dot{\theta}_6} \\ \frac{\partial f_3}{\partial \dot{\theta}_1} & \frac{\partial f_3}{\partial \dot{\theta}_2} & \frac{\partial f_3}{\partial \dot{\theta}_3} & \frac{\partial f_3}{\partial \dot{\theta}_4} & \frac{\partial f_3}{\partial \dot{\theta}_5} & \frac{\partial f_3}{\partial \dot{\theta}_6} \\ \frac{\partial f_4}{\partial \dot{\theta}_1} & \frac{\partial f_4}{\partial \dot{\theta}_2} & \frac{\partial f_4}{\partial \dot{\theta}_3} & \frac{\partial f_4}{\partial \dot{\theta}_4} & \frac{\partial f_4}{\partial \dot{\theta}_5} & \frac{\partial f_4}{\partial \dot{\theta}_6} \\ \frac{\partial f_5}{\partial \dot{\theta}_1} & \frac{\partial f_5}{\partial \dot{\theta}_2} & \frac{\partial f_5}{\partial \dot{\theta}_3} & \frac{\partial f_5}{\partial \dot{\theta}_4} & \frac{\partial f_5}{\partial \dot{\theta}_5} & \frac{\partial f_5}{\partial \dot{\theta}_6} \\ \frac{\partial f_6}{\partial \dot{\theta}_1} & \frac{\partial f_6}{\partial \dot{\theta}_2} & \frac{\partial f_6}{\partial \dot{\theta}_3} & \frac{\partial f_6}{\partial \dot{\theta}_4} & \frac{\partial f_6}{\partial \dot{\theta}_5} & \frac{\partial f_6}{\partial \dot{\theta}_6} \end{bmatrix} $$

∴

$$ {}^{6}_{0}J = \begin{bmatrix} 0.15005 & -0.4318 & -0.4318 & 0.0000 & 0.0000 & 0.0000 \\ 0.4521 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.4521 & 0.0203 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & -1.0000 & -1.0000 & 0.0000 & -1.0000 & 0.0000 \\ 1.0000 & 0.0000 & 0.0000 & 1.0000 & 0.0000 & 1.0000 \end{bmatrix} $$

References

Corke, P. (n.d.). *mdl_puma560* [MATLAB Script file]. Retrieved from https://github.com/zchen24/rvctools/blob/master/robot/mdl_puma560.m

Corke, P. (n.d.). *sl_jspace.mdl* [Simulink Model File]. Retrieved from https://github.com/zchen24/rvctools/blob/master/simulink/sl_jspace.mdl

Corke, P. (n.d.). *slplotbot.m* [Simulink Model File]. Retrieved from https://github.com/zchen24/rvctools/blob/master/simulink/slplotbot.m

Corke, P. (n.d.). *walking.m* [MATLAB M-file]. Retrieved from https://github.com/zchen24/rvctools/blob/master/robot/examples/walking.m

Craig, J. J. (2005). *Introduction to robotics: Mechanics and control* (3rd ed.). Upper Saddle River: Pearson Education International.

Appendix

All animation videos regarding this project are uploaded to the website https://www.bilibili.com/video/av12668256/.

Acknowledgement

This project couldn't have been finished without the guidance and help from Dr. Wang Zhengsheng, who offered useful insights into the D-H conventions, Jacobian matrices and partial derivatives. Yimo Chong (Kent ′20) also aided in the process by proofreading my work.