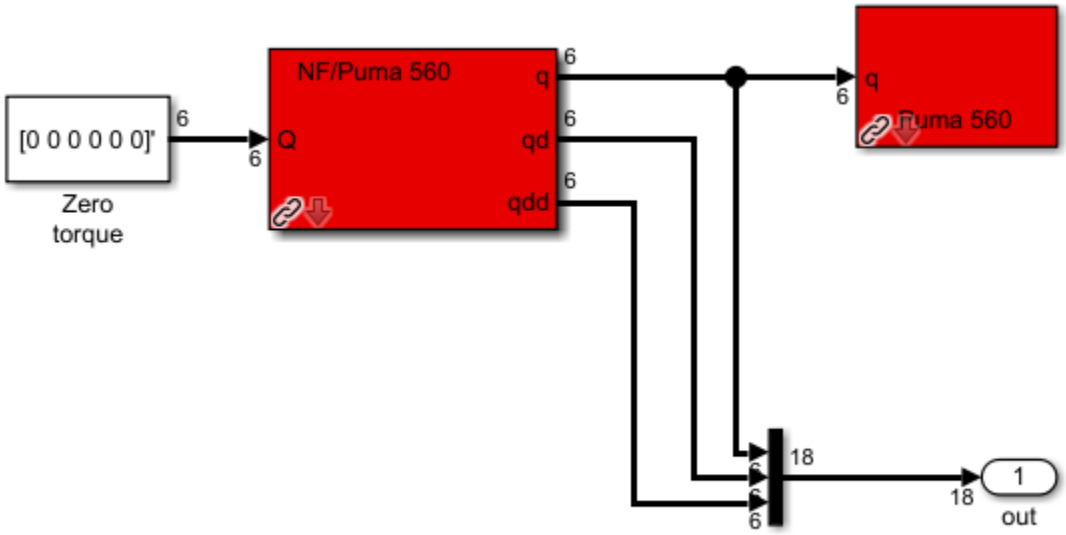
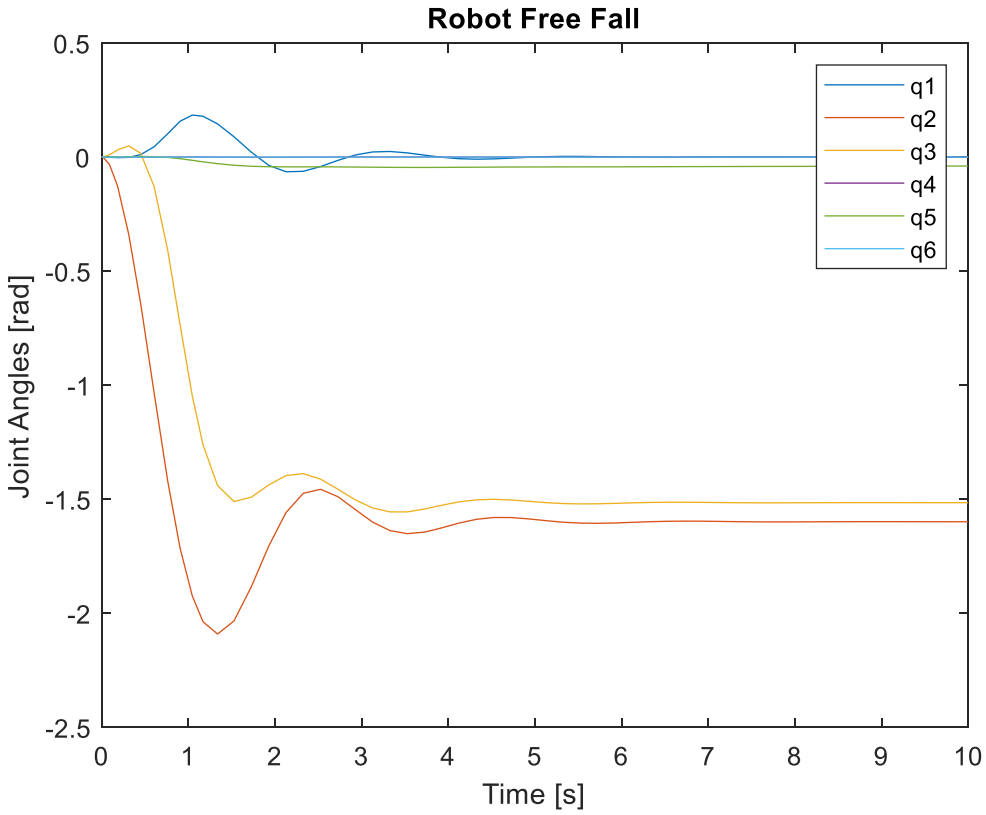
Daniel Sousa Schulman

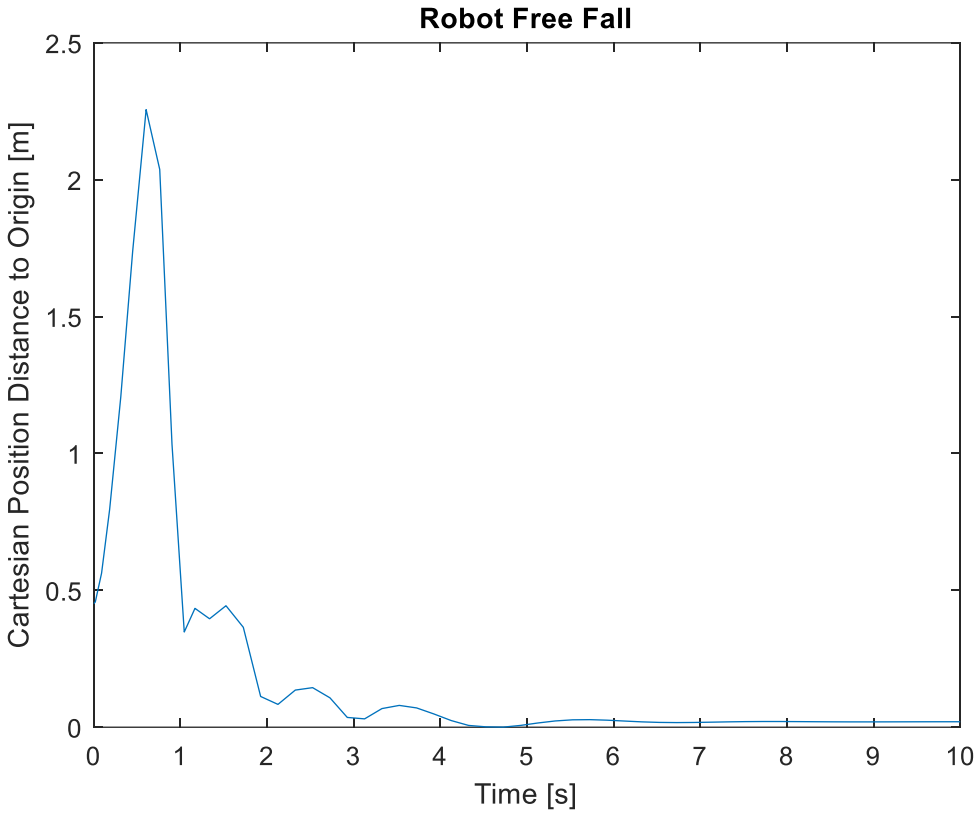
ME8287 IP2

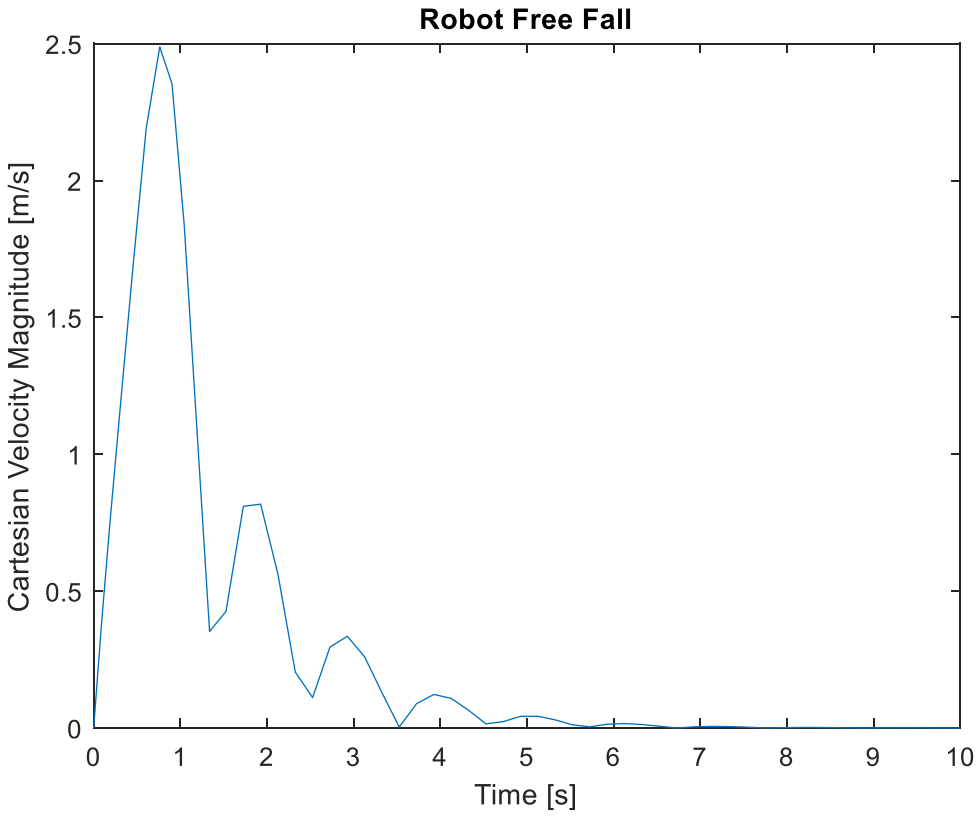
Part A
A.1

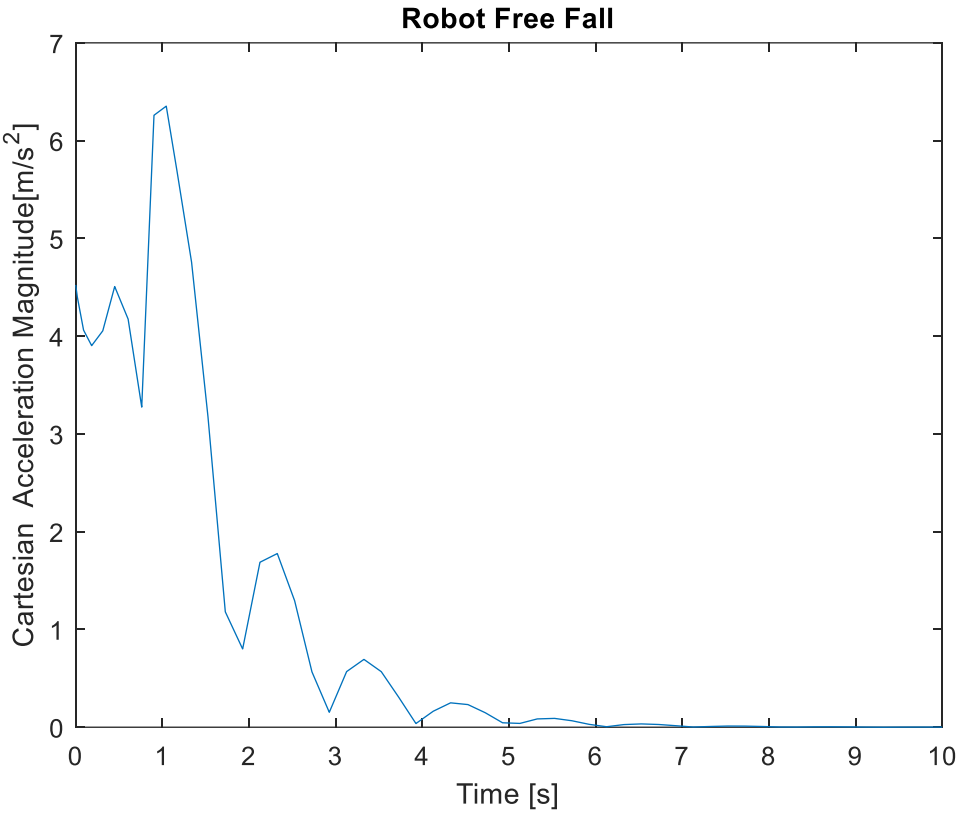

[0 0 0 0 0 0]'
Zero
torque
NF/Puma 560
Q
q
qd
qdd
Puma 560
18
1
out

A.2

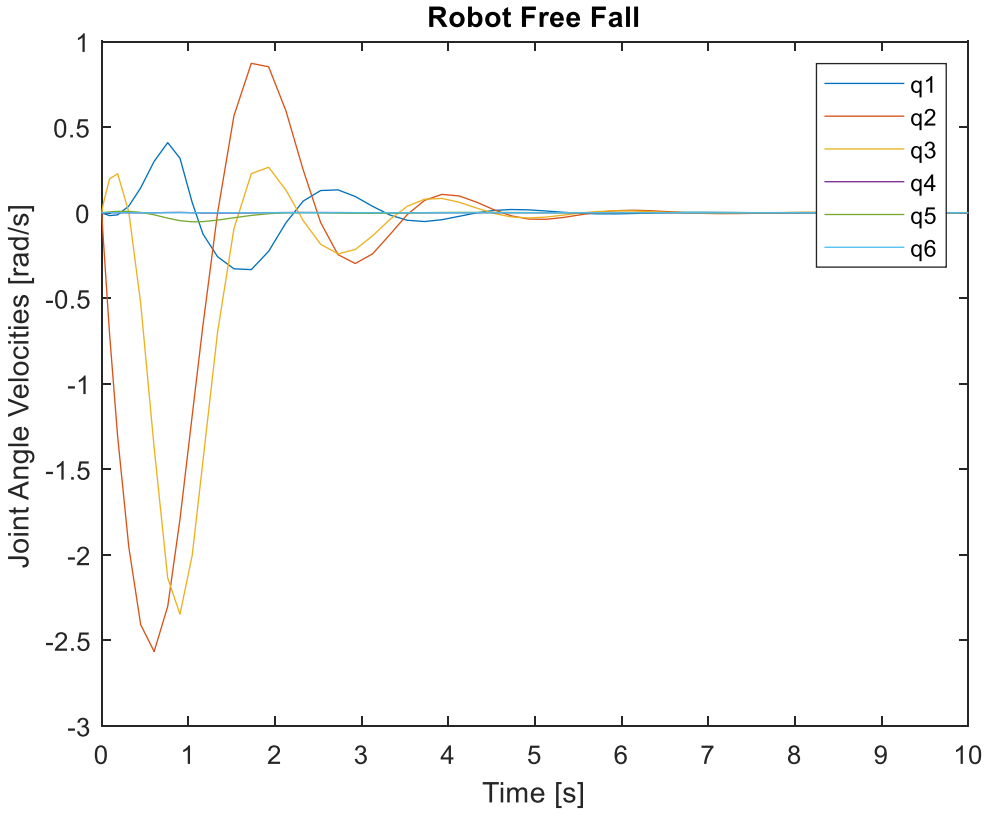

Robot Free Fall
Joint Angles [rad]
Time [s]
q1
q2
q3
q4
q5
q6


Robot Free Fall
Cartesian Position Distance to Origin [m]
Time [s]
0
0.5
1
1.5
2
2.5
0
1
2
3
4
5
6
7
8
9
10

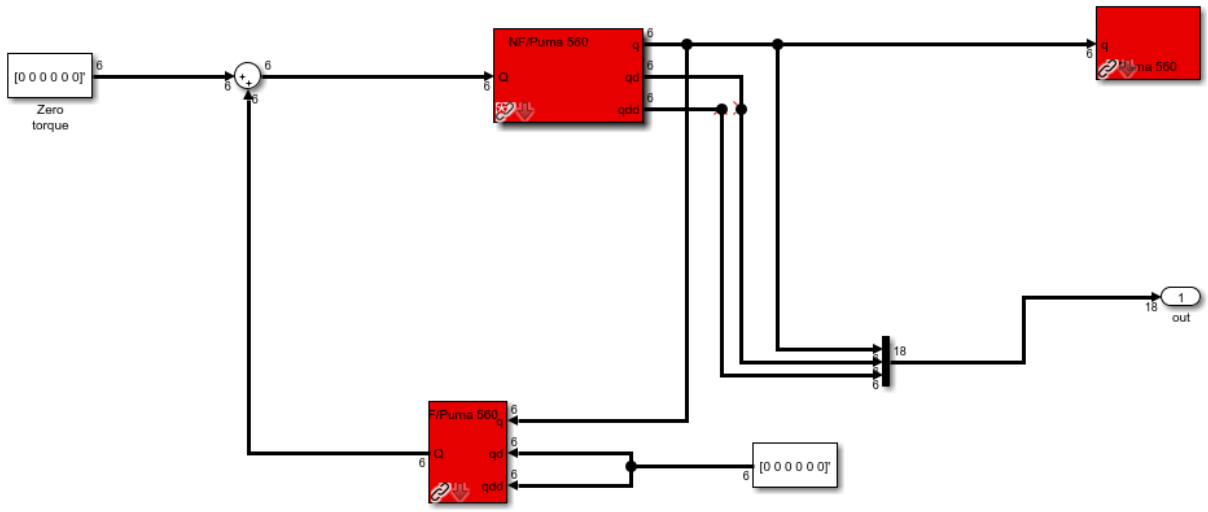

Robot Free Fall
Cartesian Velocity Magnitude [m/s]
Time [s]
0
0.5
1
1.5
2
2.5
0
1
2
3
4
5
6
7
8
9
10


Robot Free Fall
Cartesian Acceleration Magnitude[m/s^2]
Time [s]
0
1
2
3
4
5
6
7
8
9
10

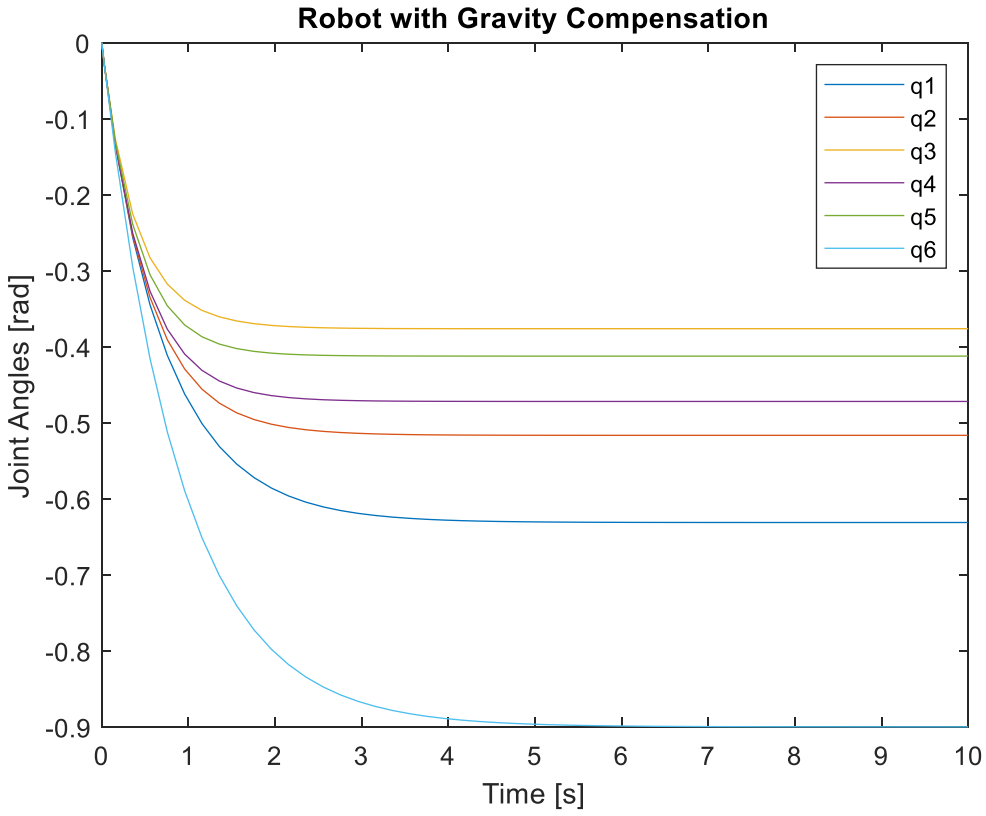
A.3

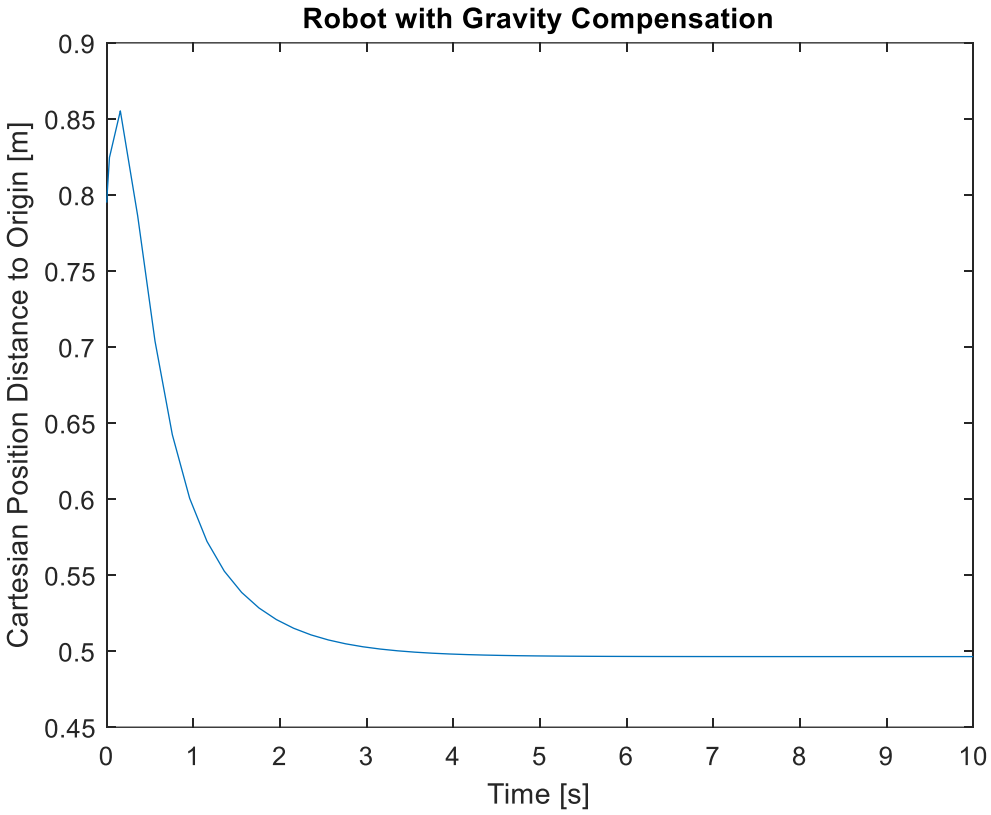

Robot Free Fall
Joint Angle Velocities [rad/s]
Time [s]
q1
q2
q3
q4
q5
q6
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-3
0
1
2
3
4
5
6
7
8
9
10

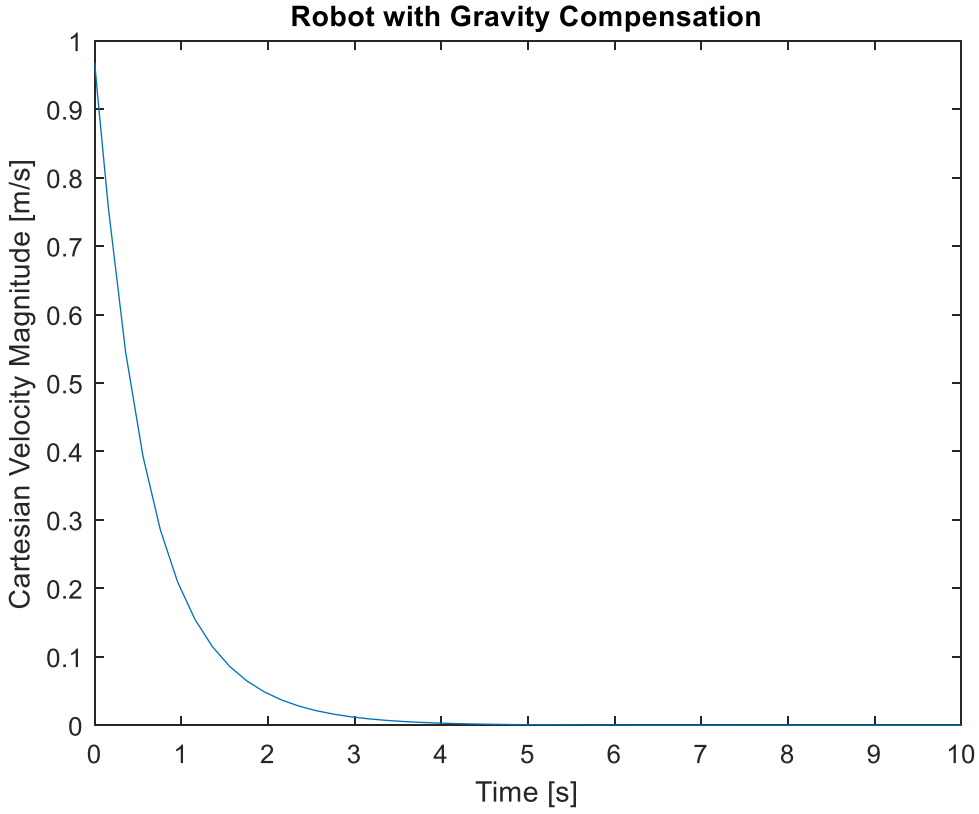
Part B
B.1

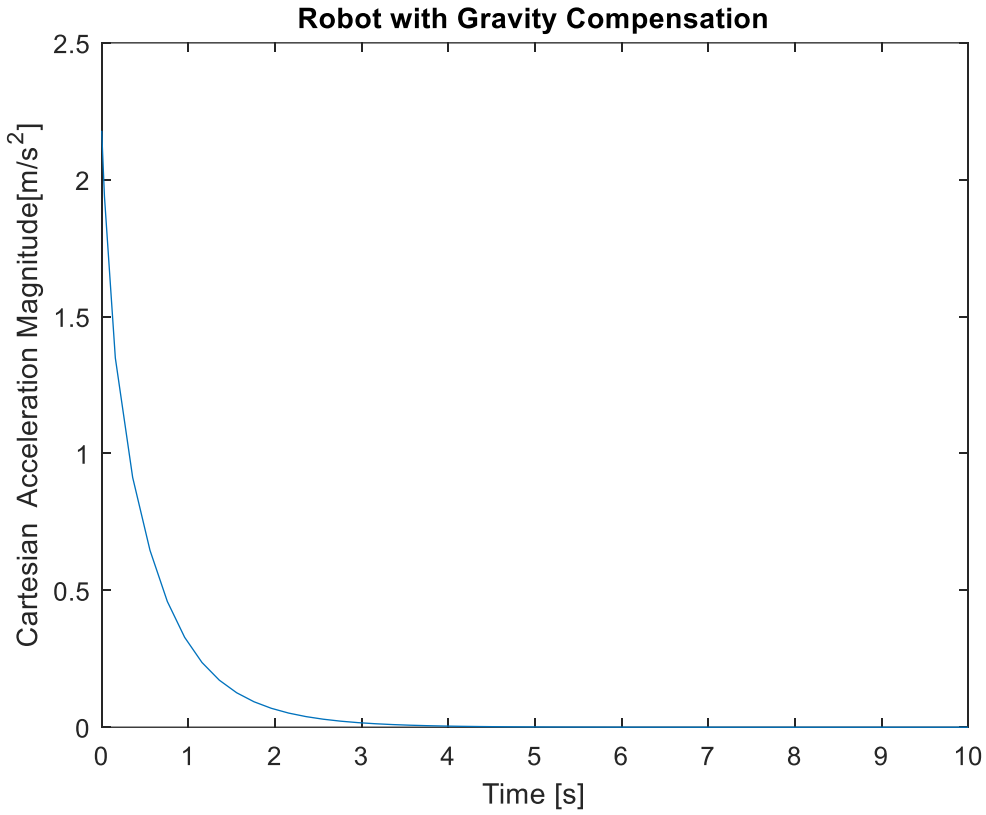

[0 0 0 0 0 0]'
Zero
torque
NF/Puma 560
Q
q
qd
qdd
F/Puma 560
18
1
out
[0 0 0 0 0 0]'

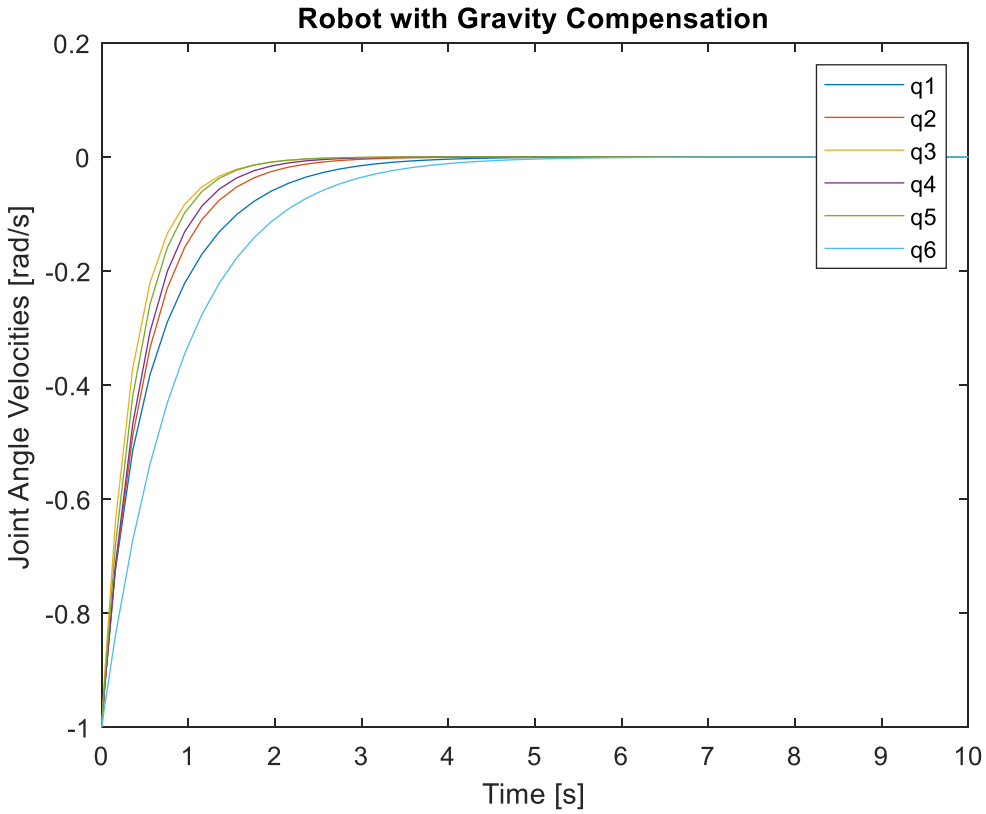
B.2


Robot with Gravity Compensation
Joint Angles [rad]
Time [s]
q1
q2
q3
q4
q5
q6
0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.6
-0.7
-0.8
-0.9
0
1
2
3
4
5
6
7
8
9
10

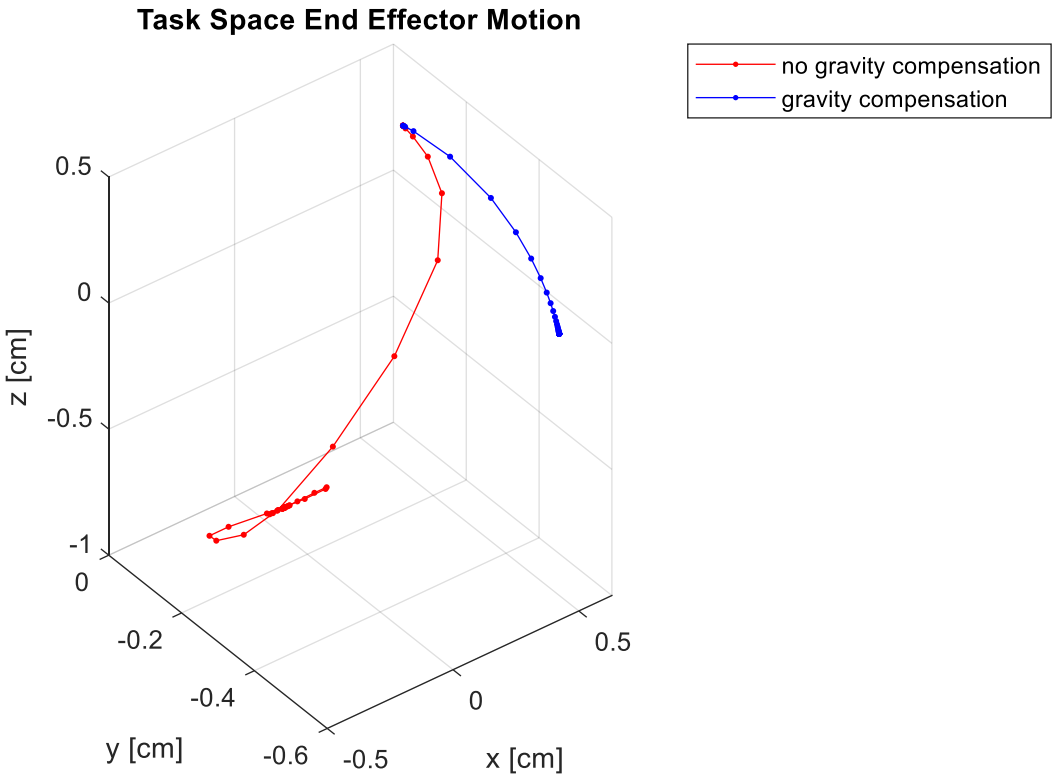

Robot with Gravity Compensation
Cartesian Position Distance to Origin [m]
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0
1
2
3
4
5
6
7
8
9
10
Time [s]


Robot with Gravity Compensation
Cartesian Velocity Magnitude [m/s]
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
1
2
3
4
5
6
7
8
9
10
Time [s]


Robot with Gravity Compensation
Cartesian Acceleration Magnitude[m/s^2]
Time [s]
2.5
2
1.5
1
0.5
0
0 1 2 3 4 5 6 7 8 9 10


Robot with Gravity Compensation
Joint Angle Velocities [rad/s]
Time [s]
q1
q2
q3
q4
q5
q6
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0 1 2 3 4 5 6 7 8 9 10

B.3



B.4

A Yes
B Each of the links can be modeled as a mass-spring system. Further, given viscous friction, there is damping associated with the system. A spring-mass-damper system is stable to an impulse in both translation and rotation. This occurs since the spring generates an opposing force to the direction of elongation and the damper reduces the energy of the system, causing the mass to eventually stabilize around the equilibrium point. Thus, if every robot link is held at a certain position through gravity compensation, a disturbance in the end effector will cause an oscillation about the equilibrium points for each link, but it will ultimately lead to asymptotic stability.

B.5

a) No, since the gravity compensation accounts only for the constant gravitational acceleration and not the robot joints' accelerations. An evidence of this is that the gravload function calls the RNE function with 'dqq' as a zero vector.
b) Yes, since gravity compensation uses the inertia matrix to create a model for torque feedforward control.
c) Yes, since the robot's kinematics define the orientation of the links based on a given pose in joint space, which influences the force distribution along the robot.

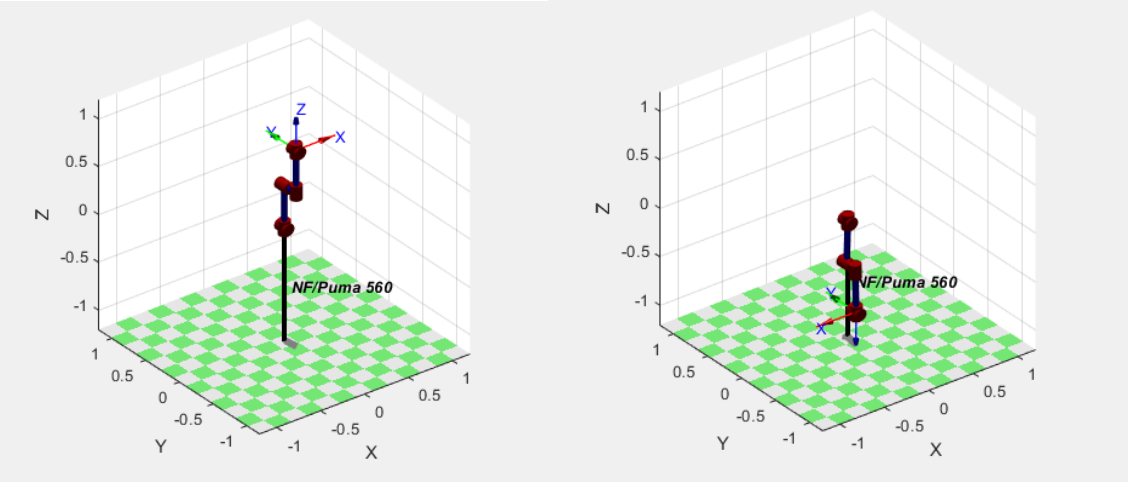
B.6

For all cases, qdd=[0 0 0 0 0 0]

Applying initial condition qd0= -[1 1 1 1 1 1]
Torque for straight up pose =[-5.8466, -10.2243, -3.7314, -0.4116, -0.4270, -0.2158]
Torque for straight down pose=[-5.8634 -8.6740 -4.2293 -0.4116 -0.4270 -0.2158]

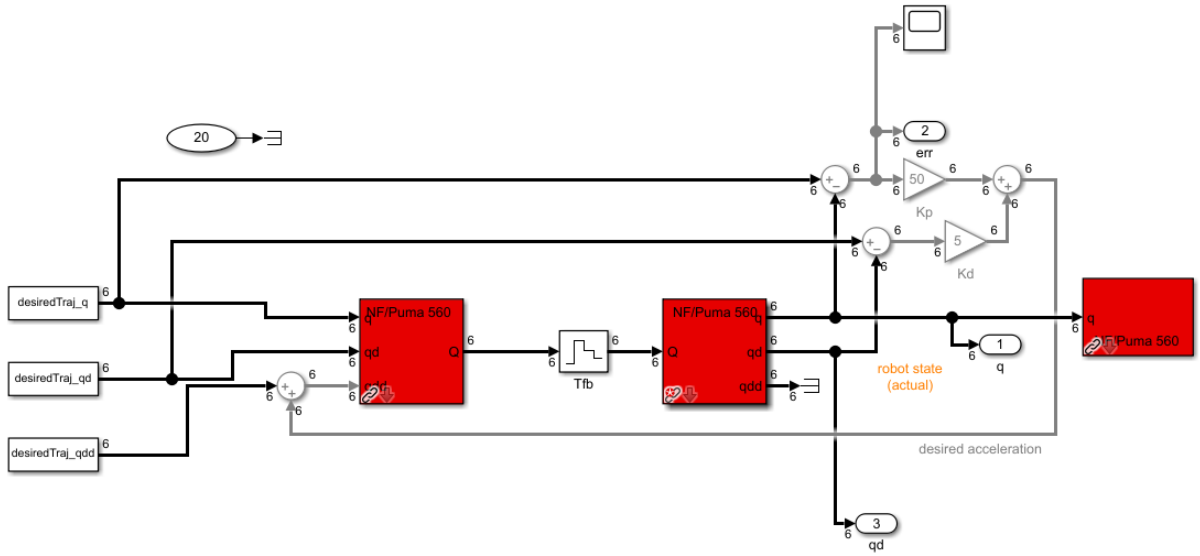
Applying initial condition qd0=-[0 0 0 0 0 0]
Torque for straight up pose =[0, -0.7752, 0.2489, 0, 0, 0]
Torque for straight down pose=[0.0000, 0.7751, -0.2490, 0.0000, 0.0000, 0]

These torques, even though close, are not zero. This occurs because there is an offset between the center of mass of some of the links and the base of the robot. This generates the need to compensate the moment generated by the offset with torques. However, as most of the links are aligned and most of the reaction forces act opposing gravity, the torques are overall much lower when compared to other positions. Both poses are shown below.



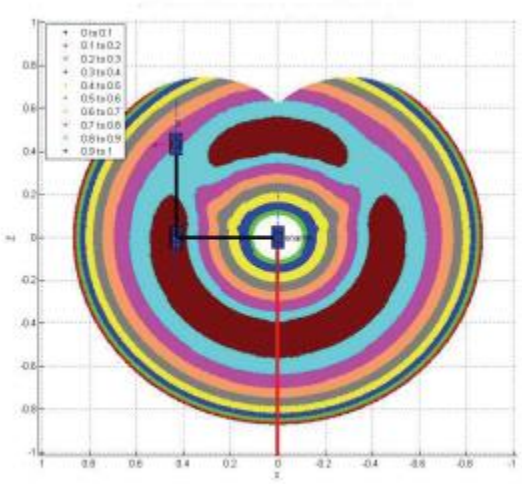
Part C

C.1



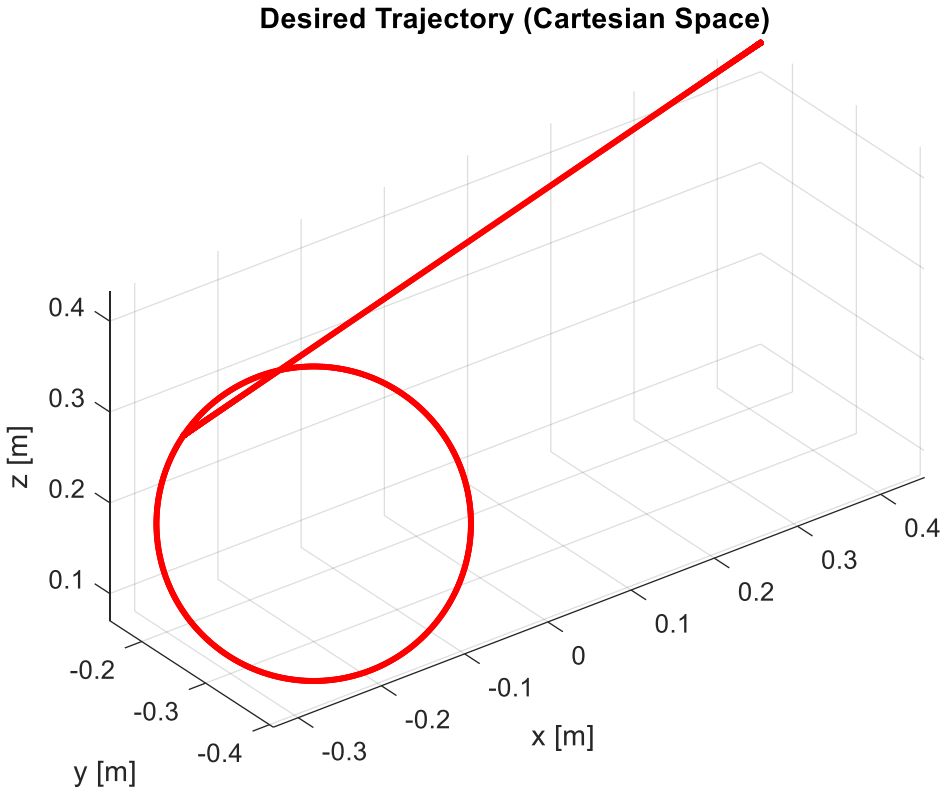
C.2

Elkady et al. studied the manipulability (through singular value decomposition) of the Puma 560 Robot in a single plane, which gives a good indication of where the dexterity of the robot is the highest. The Figure below was extracted from the manuscript and graphically shows the band regions for manipulability measure.



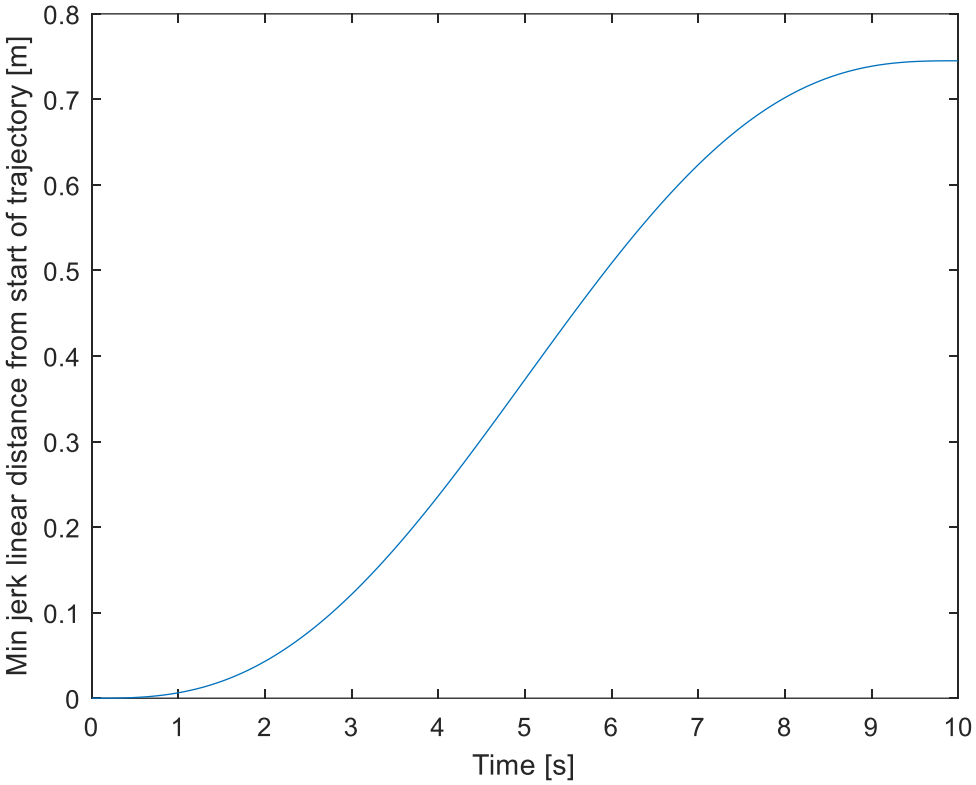
Based on that information, the circle was placed such that the center would fall in the middle of the highest manipulability band. While the manipulability in the y is not shown, it is assumed that the geometric pattern display of the manipulability bands is reasonably preserved when the plane

is rotated 360 degrees. The value for the y offset was such that it would be placed in the best 2D manipulability band when rotated. Thus, the position offset vector chosen was [-0.2, -0.3, 0.2]. The initial and final positions for the end effector in the circular trajectory are the same. They were chosen such that they would minimize the distance between the circular trajectory and the home pose initial position as shown in the Figure below.



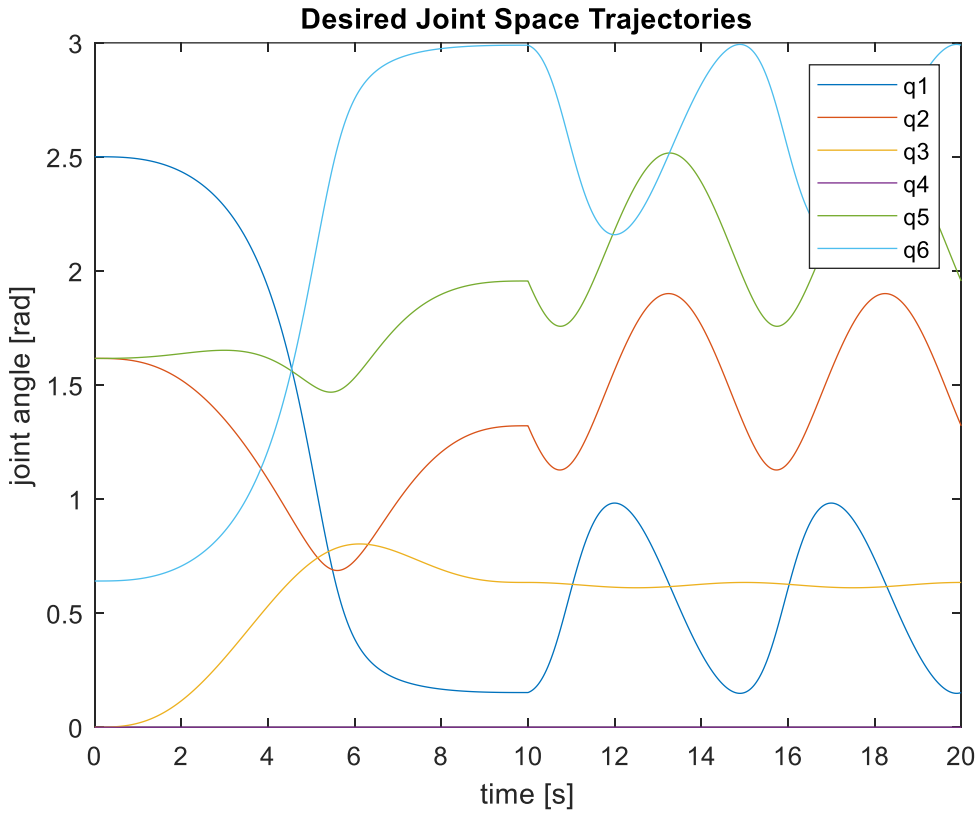
C.3

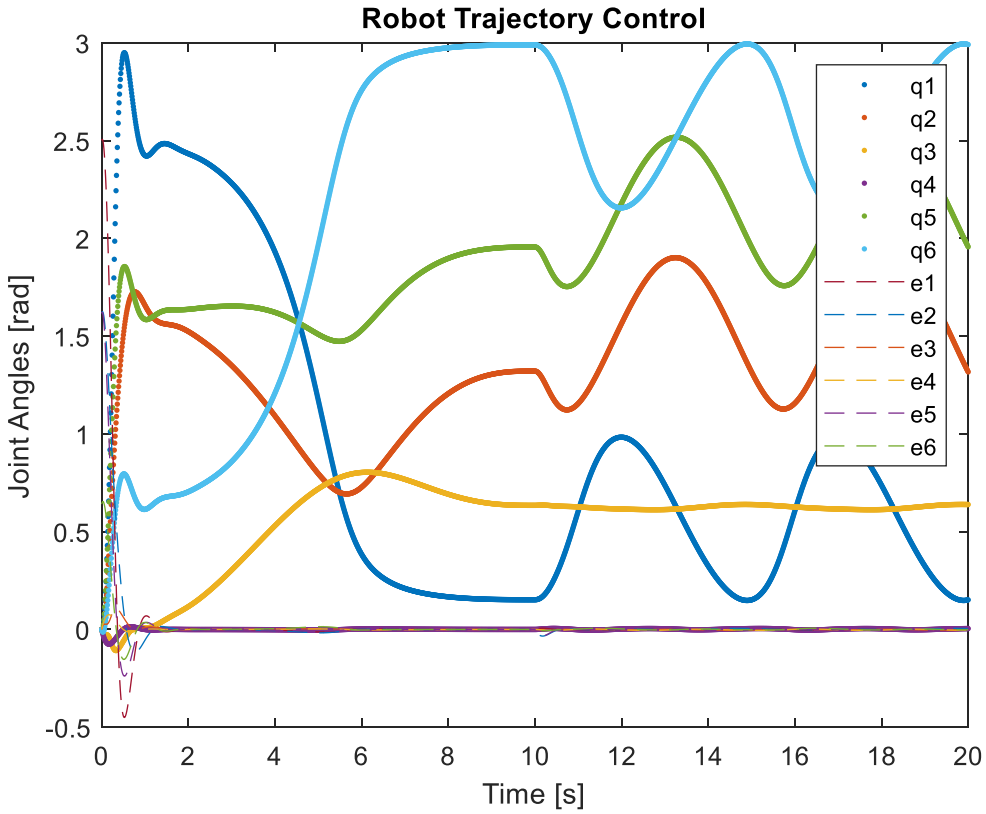
The trajectory between the home pose initial position and the circular trajectory was chosen in order to minimize jerk. In order to do so, the initial home pose Cartesian coordinate was found by calculating the forward kinematics of the robot and a line was traced between that point and the point chosen for the start of the circular trajectory. A line was used due to the possibility of using the 1D trajectory output from "minJerkTrajectory.m" in order to parametrize the line. The jerk linear trajectory output (shown in the Figure below) was multiplied by the unit vector that defined the direction from the home start point to the circular trajectory starting point and the start point offset was added to define the initial trajectory point. The chosen time duration for this linear trajectory was 10 seconds. Details for the execution of this algorithm are included in the comments in the Appendix code.



One consequence of this approach is that the first and second order derivatives for the trajectory had to be calculated numerically using forward difference instead of using a syms variable since that would not be supported by the minimum jerk trajectory code provided.

A problem that was observed with defining the home pose Cartesian position is that when trying to calculate the inverse kinematics for that and surrounding points, the existence of singularities in certain directions causes ikine6s to not include the solution for the joint home pose configuration as a possibility. All possible configurations of elbow up vs down, arm right vs left and wrist flipped vs not flipped were tested and none resulted in the home pose configuration. The second norm of the difference between the possible initial joint angle positions and the home pose joint angle positions was used in order to determine the solution that would fit best, which turned out to be the default one. This issue caused the robot to move outside of the linear trajectory for a small amount before initiating the desired trajectory. The desired values for the joint angles and the actual output with errors associated are shown in the two Figures below.

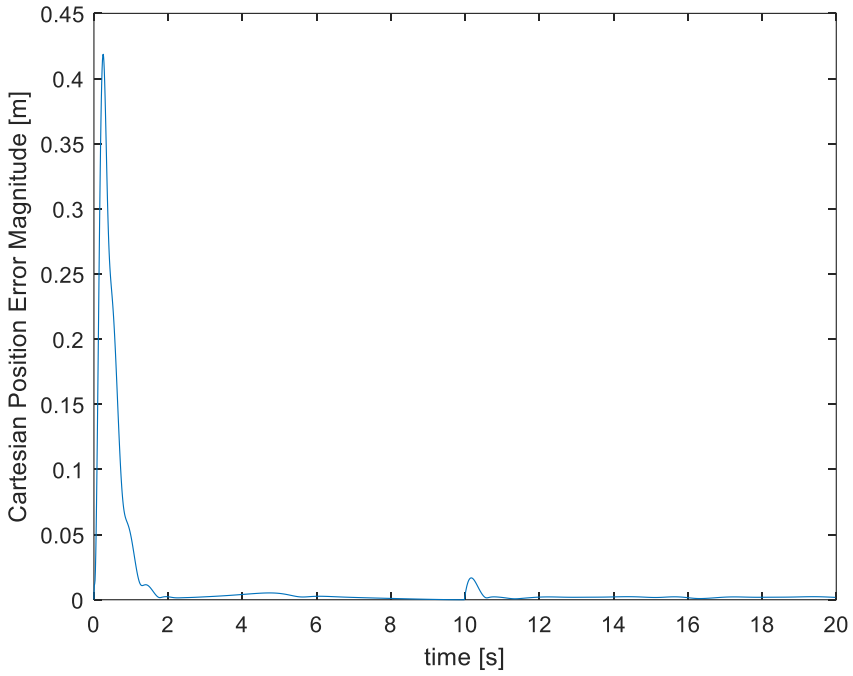

Desired Joint Space Trajectories
joint angle [rad]
time [s]
q1
q2
q3
q4
q5
q6
0
0.5
1
1.5
2
2.5
3
0
2
4
6
8
10
12
14
16
18
20

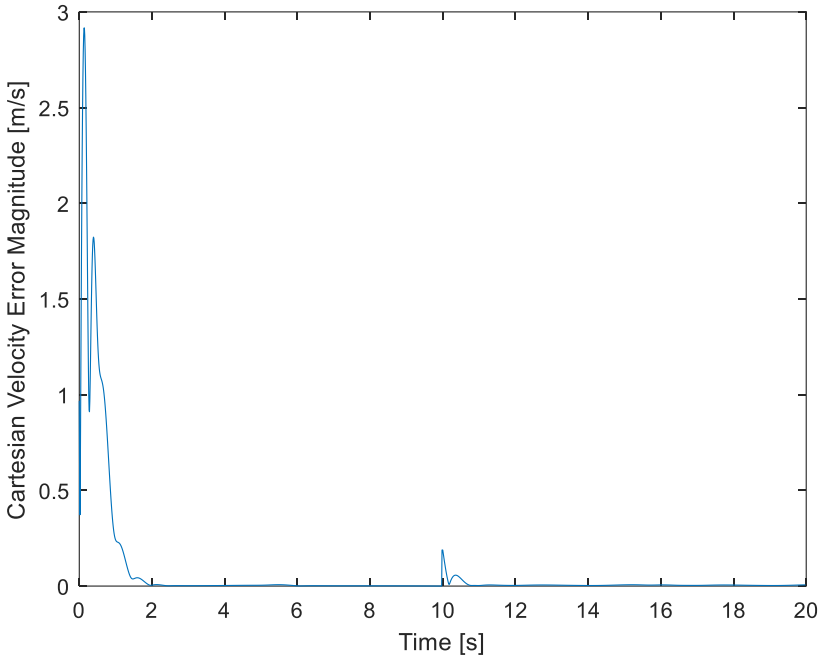

Robot Trajectory Control
Joint Angles [rad]
Time [s]
q1
q2
q3
q4
q5
q6
e1
e2
e3
e4
e5
e6
-0.5
0
0.5
1
1.5
2
2.5
3
0
2
4
6
8
10
12
14
16
18
20

C.4

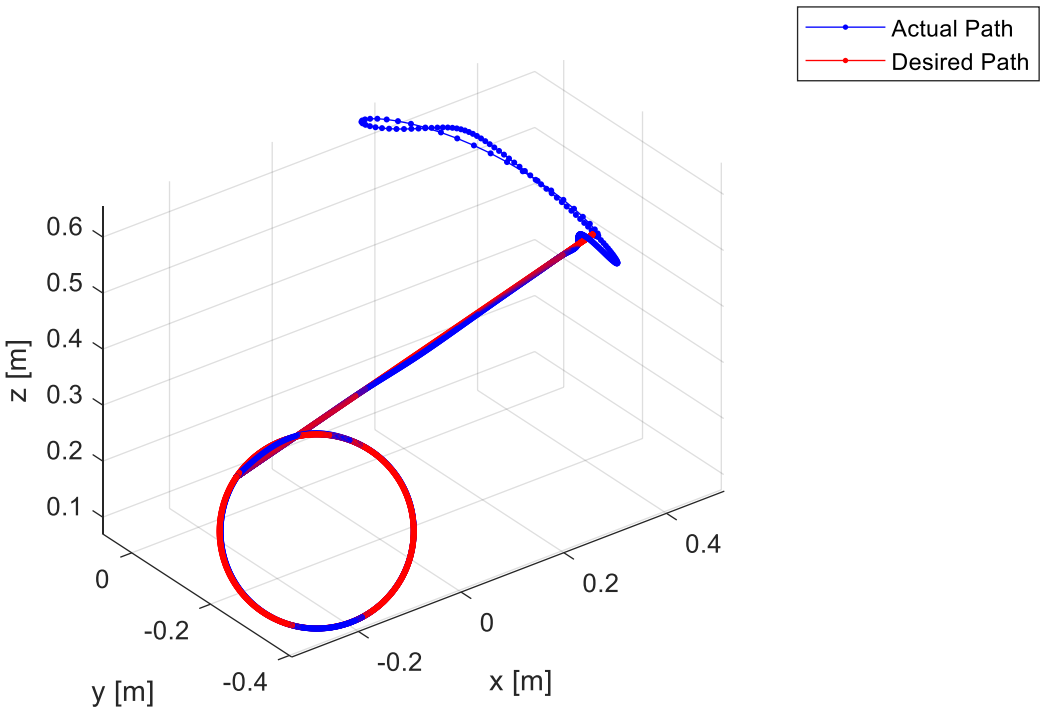
In the absence of an accurate dynamic model of the puma robot, an experimental approach similar to the one presented in Corke Chapter 9.4 was used. An increase in Kp causes a decrease in steady state error, which is the biggest concern for this problem. Thus, it was straightforward that a high Kp value had to be chosen. A lower value for Kd helped counteracting the increase in percent overshoot caused by an increase in Kp, Through experimentation, the values of Kp and Kd that met the position error spec and had a reasonable overshoot were Kp=70 and Kd=5.

C.5

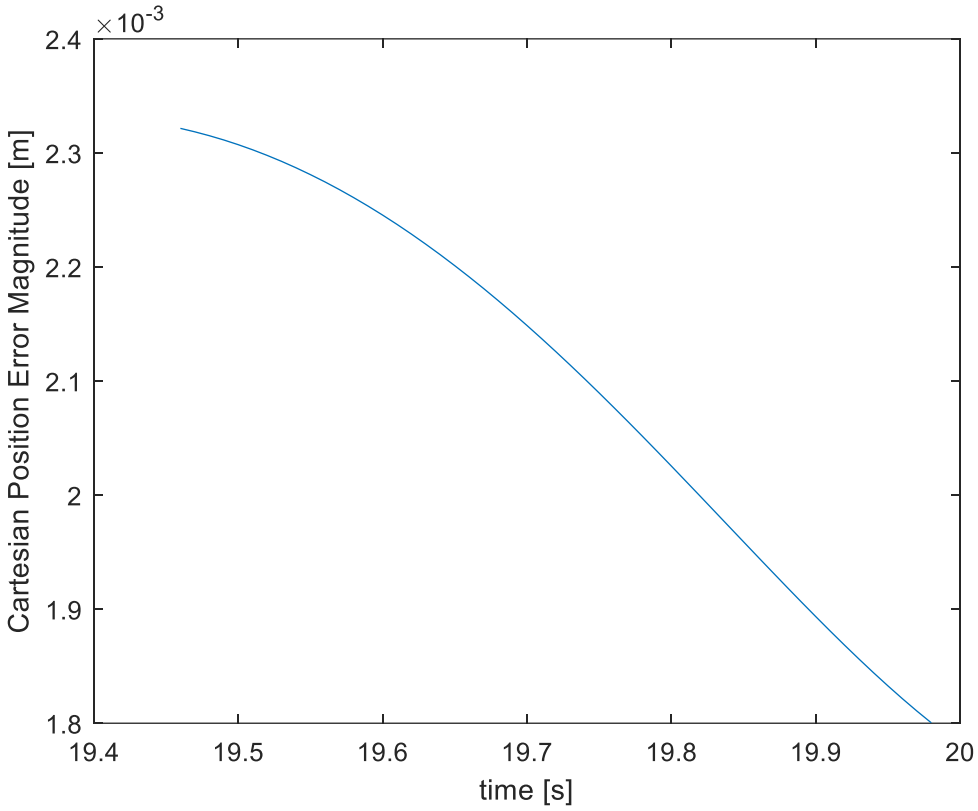


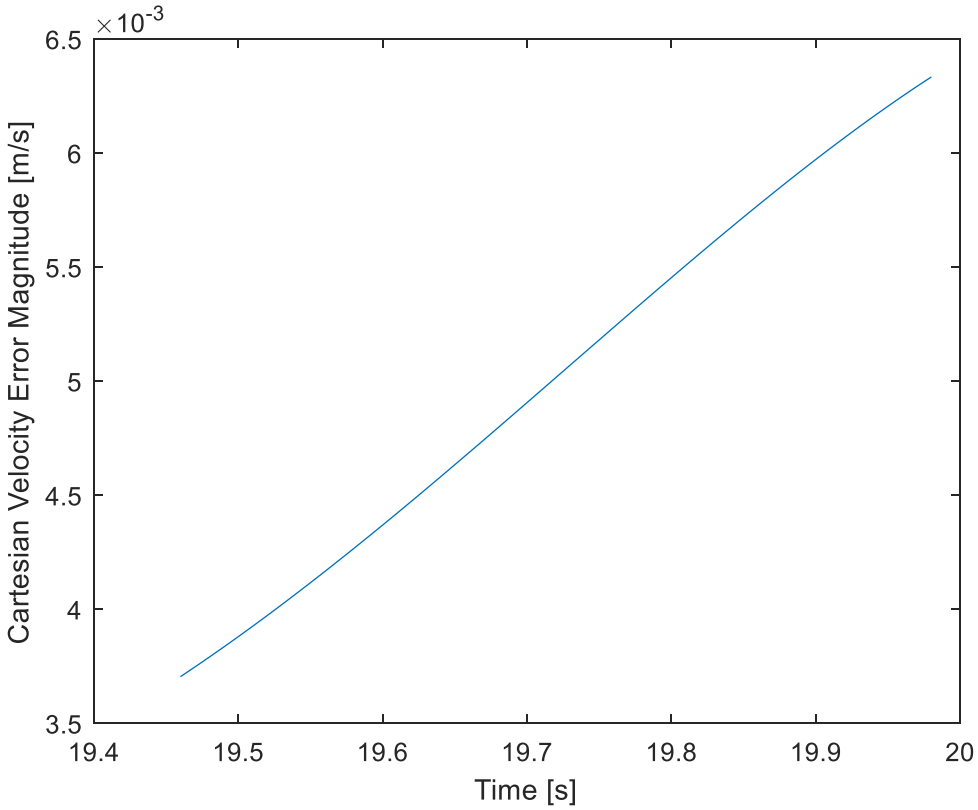

Cartesian Velocity Error Magnitude [m/s]
Time [s]
0
0.5
1
1.5
2
2.5
3
0
2
4
6
8
10
12
14
16
18
20

C.6


Actual Path
Desired Path
z [m]
y [m]
x [m]
0.6
0.5
0.4
0.3
0.2
0.1
0
-0.2
-0.4
0.4
0.2
0
-0.2

C.7


×10^-3
2.4
2.3
2.2
2.1
2
1.9
1.8
Cartesian Position Error Magnitude [m]
19.4
19.5
19.6
19.7
19.8
19.9
20
time [s]


×10^-3
6.5
6
5.5
5
4.5
4
3.5
Cartesian Velocity Error Magnitude [m/s]
19.4
19.5
19.6
19.7
19.8
19.9
20
Time [s]

The positional error was in the order of 2.5mm. The error spec defined in the problem for 0.5mm is really low for the puma 560 model. Even when the controller gains are increased to values that would be unfeasible in the real world (order of 10^3), the steady state error does not fall below 0.5mm. The existence of a steady state error is associated with the absence of an accurate feedforward term, making it impossible to achieve exact tracking for non-trivial trajectories. A solution to the problem of eliminating steady state error would be including an integral gain to the controller. However, special care needs to be taken in order to ensure stability and avoid integrator windup. A possibility is to use the Ziegler–Nichols method for PID control design.

References

Elkady, Ayssam Yehia, et al. "A New Algorithm for Measuring and Optimizing the Manipulability Index." *Journal of Intelligent and Robotic Systems*, vol. 59, no. 1, 2009, pp. 75–86., doi:10.1007/s10846-009-9388-9.

Corke, Peter I. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB®*. Springer, 2017.

```
%% Project 2: Dynamic Control of a Robot
% NOTE: this scirpt assumes you load Corke's original Simulink examples,
% save them to new filenames (e.g. load sl_ztorque and save as
% sl_ztorqueFreefallsl), then modify them to work with this script.
%}

%% Constants, for Simulink Files:

% TO DO:  edit your simulink models to use each of these ...
q0 = [0 0 0 0 0 0];     % initial robot pose (invoked within
                        % sl_ztroqueFreefall's Robot block
                        % (within it's second internal integrator)
qd0= -[1 1 1 1 1 1];    % initial robot joint velocities (invoked within
                        % s1_ztorqueFreefalls's Robot block
                        % (within it's 1st internal integrator)
Q_in = [ 0 0 0 0 0 0]'; % Torque input to motors; this is what you'll want
                        % to change in a computed torque controller
% Select a model
mdl_puma560;
myRobot = p560;  % myRobot is used internally by sl_ztroqueFreefall
myRobot = p560.nofriction;  % drop coloumb friction for faster sim times
% This still includes viscous friction

% % Select a different model
% mdl_stanford;
% myRobot = stanf.nofriction;  % myRobot is used internally by
sl_ztroqueFreefall

tf = 10;    % simulation stop time
% simulate the robot in free fall


figure(1); clf
open('sl_ztorqueFreefall.slx');
result = sim('sl_ztorqueFreefall', 'StopTime', num2str(tf) )  ; % run
simulation
% HINT: for faster execution, reduce frames per second (fps) in the plot
block.

% get info out of simulink
t =       result.get('tout)';
outputs = result.get('yout)';

q   = outputs(    1:6 , : ); % first 6 columns are joint angles
qd  = outputs( 6+[1:6] , : ); % next 6 columns are joint velocities
qdd = outputs(12+[1:6] , : ); % next 6 columns are joint accelerations


CartVel=zeros(6,length(q(1,:)));
CartPos=zeros(3,length(q(1,:)));
CartAccel=zeros(6,length(q(1,:)));

for i=1:1:length(q(1,:))
   CartVel(:,i)=(jacob0(myRobot,q(:,i)')*qd(:,i))';
   transform_myRob=myRobot.fkine(q(:,i)');
```

```
   CartPos(:,i)=transform_myRob(1:3,4);

CartAccel(:,i)=(myRobot.jacob0(q(:,i)')*qdd(:,i))'+(myRobot.jacob_dot(q(:,i)'
,qd(:,i)'))';
end

% plot stuff
figure(2); clf
plot(t,q);
xlabel('Time [s]')
ylabel('Joint Angles [rad]')
legend([['qqqqqq']' ['123456']' ] )
title('\bfRobot Free Fall')

figure(3); clf
plot(t,qd);
xlabel('Time [s]')
ylabel('Joint Angle Velocities [rad/s]')
legend([['qqqqqq']' ['123456']' ] )
title('\bfRobot Free Fall')

figure(4); clf
CartVel_norm=sqrt(CartVel(1,:).^2+CartVel(2,:).^2+CartVel(3,:).^2);
plot(t,CartVel_norm);
xlabel('Time [s]')
ylabel('Cartesian Velocity Magnitude [m/s]')
title('\bfRobot Free Fall')

figure(5); clf
CartPos_norm=sqrt(CartPos(1,:).^2+CartVel(2,:).^2+CartVel(3,:).^2);
plot(t,CartPos_norm);
xlabel('Time [s]')
ylabel('Cartesian Position Distance to Origin [m]')
title('\bfRobot Free Fall')

figure(6); clf
CartAccel_norm=sqrt(CartAccel(1,:).^2+CartAccel(2,:).^2+CartAccel(3,:).^2);
plot(t,CartAccel_norm);
xlabel('Time [s]')
ylabel('Cartesian  Acceleration Magnitude[m/s^2]')
title('\bfRobot Free Fall')

%return  % this stops execution here.  Get rid of this to keep going...

%% PART B Let's implement gravity compensation

% what does each joint torque have to be to counteract gravity at q = q0?
% % TauG = p560.gravload(q0)
% %
% % % look into this, i.e., type edit gravload and review the actual code,
% % % it's really just RNE with zero joint vel. and acceleration:
% % qd  = 0*q0;  % ignoring velocity effects
% % qdd = 0*q0;  % ignoring acceleration effects
% % Tau = myRobot.rne(q0, 0*q0, 0*q0)   % , GRAV, FEXT) % extras: Grav
redefines
```

```
% % % gravity vector, Fext defines forces
% % % applied to End Effector
% % % The lesson here:  You don't need simulink for this problem.

% simulate the robot in free fall WITH gravity compensation
% we'll use simulink anyway just to get familiar with it.
figure(1); clf; % Corke's toolkit can only plot single robot, use same fig
open('sl_ztorqueGravComp.slx');
result_g = sim('sl_ztorqueGravComp', 'StopTime', num2str(tf) )  ;
% HINT: for faster execution, reduce frames per second (fps) in the plot
block.

% get info out of simulink
t_g =       result_g.get('tout')';
outputs = result_g.get('yout')';

q_g   = outputs(    1:6 , : ); % first 6 columns are joint angles
qd_g  = outputs( 6+[1:6] , : ); % next 6 columns are joint velocities
qdd_g = outputs(12+[1:6] , : ); % next 6 columns are joint accelerations

CartVel_g=zeros(6,length(q_g(1,:)));
CartPos_g=zeros(3,length(q_g(1,:)));
CartAccel_g=zeros(6,length(q_g(1,:)));

for i=1:1:length(q_g(1,:))
   CartVel_g(:,i)=(jacob0(myRobot,q_g(:,i)')*qd_g(:,i))';
   transform_myRob_g=myRobot.fkine(q_g(:,i)');
   CartPos_g(:,i)=transform_myRob_g(1:3,4);

CartAccel_g(:,i)=(myRobot.jacob0(q_g(:,i)')*qdd_g(:,i))'+(myRobot.jacob_dot(q
_g(:,i)',qd_g(:,i)'))';
end

% plot stuff
figure(7); clf
plot(t_g, q_g);
xlabel('Time [s]')
ylabel('Joint Angles [rad]')
legend([['qqqqqq']' ['123456']' ] )
title('\bfRobot with Gravity Compensation')

figure(8); clf
plot(t_g,qd_g);
xlabel('Time [s]')
ylabel('Joint Angle Velocities [rad/s]')
legend([['qqqqqq']' ['123456']' ] )
title('\bfRobot with Gravity Compensation')

figure(9); clf
CartVel_norm_g=sqrt(CartVel_g(1,:).^2+CartVel_g(2,:).^2+CartVel_g(3,:).^2);
plot(t_g,CartVel_norm_g);
xlabel('Time [s]')
ylabel('Cartesian Velocity Magnitude [m/s]')
title('\bfRobot with Gravity Compensation')
```

```
figure(10); clf
CartPos_norm_g=sqrt(CartPos_g(1,:).^2+CartVel_g(2,:).^2+CartVel_g(3,:).^2);
plot(t_g,CartPos_norm_g);
xlabel('Time [s]')
ylabel('Cartesian Position Distance to Origin [m]')
title('\bfRobot with Gravity Compensation')

figure(11); clf
CartAccel_norm_g=sqrt(CartAccel_g(1,:).^2+CartAccel_g(2,:).^2+CartAccel_g(3,:
).^2);
plot(t_g,CartAccel_norm_g);
xlabel('Time [s]')
ylabel('Cartesian  Acceleration Magnitude[m/s^2]')
title('\bfRobot with Gravity Compensation')


figure(12); clf
p   = myRobot.fkine(q.'         );     % get Forward Kinematics (FK)
p_g = myRobot.fkine(q_g.'       );
p   = transl(p);                       % extract positions only
p_g = transl(p_g);

plot3( p(:,1)  , p(:,2)  , p(:,3)  , 'r.-'); hold on;
plot3( p_g(:,1), p_g(:,2), p_g(:,3), 'b.-'); grid on
legend('no gravity compensation', 'gravity compensation')
xlabel('x [cm]')
ylabel('y [cm]')
zlabel('z [cm]')
title('\bfTask Space End Effector Motion')

disp('Torques for straight up position');
qdd0=[0 0 0 0 0 0];
q_down=[0   -1.5708   -1.5708         0         0         0]; %extracted from
the last free fall position
%Assume same qd0 and 0 vector for acceleration
myRobot.rne(qr,qd0, qdd0) %straight up
disp('Torques for straight down position');
myRobot.gravload(q_down,qd0, qdd0) %straight down
%}

%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART C: Here is the Cartesian trajectory you want to the robot to follow:
% you CANNOT change this part...
syms t A k P Pd Pdd

% parametric equations in t for a circle in the x-y plane in symbolic form
x = A*sin(k*t);
y = A*cos(k*t);
z = 0;

% rotate by x an y  axes; THIS IS YOUR DESIRED MOTION:
P   = [  x y z 1 ]   *  trotx( -pi/4 )  * troty( pi/4 );
Pd  = diff(  P , t);
Pdd = diff(  Pd, t);
```

```
% substitute in real numbers
A = 0.15;    % circle radius in meters
k = 2*pi/5;  % angular freq. (speed around circle)
Ts = .01;    % sample period
t = 0:Ts:tf;% numerical time vector
% ...END OF TRAJECTORY DEFINITION

% You get to place this anywhere you like in the workspace, change the
% offset and provide rationale for doing so:
Poffset = [ -0.2 -.3 .2 0];  % here's an example, not the best.
P = P + Poffset;

%% create the desired trajectory  in Cartesian AND joint spaces...
% NOTE:  Can use the following relationships to get accelrations...
%   q =  IK( x )                        x = FK( q )
%  qd =  inv(J) * xd                   xd = J * qd
% qdd =  inv(J)*[xdd - Jdot*qd]      xdd = Jdot*qd + J*qdd
t = t.';

% Cartesian space trajectories
traj.P = eval(  [P(1) P(2) P(3)]  ); % x-y-z position columns (vs time)
traj.T = transl(traj.P);            % Transformation matrix for tranlations
traj.Pd  = eval( [Pd(1) Pd(2) Pd(3)] )  ; % x-y-z 1st derivative (of time)
traj.Pdd = eval( [Pdd(1) Pdd(2) Pdd(3)]); % x-y-z 2nd derivative (of time)

% joint space trajectories
traj.q  = myRobot.ikine6s( traj.T );
traj.qd = zeros(length(t), 6);  % preallocate for speed...
traj.qdd= zeros(length(t), 6);  % preallocate for speed...
tic
for i = 1:length(t)
    % compute jacobian, its inverse, and time derivative-qd product
    J    = myRobot.jacob0(    traj.q(i,:) );
    Jinv = inv( J );
    qd   = Jinv * [traj.Pd(i,:) 0 0 0].';
    JdotQd = myRobot.jacob_dot( traj.q(i,:), qd ) ;
    qdd=  Jinv * [ [traj.Pdd(i,:) 0 0 0]' - JdotQd ];
    % save for each time step
    traj.qd(i,:)  = qd';
    traj.qdd(i,:) = qdd';
end
toc
%% TO DO: design and append your own initial approach trajectory segment,
%  that is, motion fro from the home pose to the initial circle contact
%  point
homeTransform=myRobot.fkine([q0]);
startPt=homeTransform(1:3,4)'; %home pose cartesian position
finalPt=traj.P(1,:); %first point defined in the circle trajectory
x=0+xoffset and y=A+yoffset ie. the top of the circle
%this is a point in the circle to the home pose

timeStep=0.01;
timeVector=0:timeStep:10;
%find unidirectional distance between the points
dist=sqrt((finalPt(1)-startPt(1))^2+(finalPt(2)-startPt(2))^2+(finalPt(3)-
startPt(3))^2);
```

```
unitDirectionVector=(finalPt-startPt)/dist;
%find minimum jerk for that distance and t=4s
minJerk1D=minJerkTrajectory(timeVector',[0 0 0],[dist 0 0]);

%convert 1D jerk trajectoy back to our line of interest in 3D
initialOffset=repmat(startPt,length(minJerk1D),1);

myNewTrajSegment.P=initialOffset+unitDirectionVector.*minJerk1D;

numerDeriv=zeros(length(myNewTrajSegment.P),3);

%forward difference numerical approx
for i=1:1:(length(myNewTrajSegment.P)-1)
    numerDeriv(i,1)=(myNewTrajSegment.P(i+1,1)-
myNewTrajSegment.P(i,1))/timeStep;
    numerDeriv(i,2)=(myNewTrajSegment.P(i+1,2)-
myNewTrajSegment.P(i,2))/timeStep;
    numerDeriv(i,3)=(myNewTrajSegment.P(i+1,3)-
myNewTrajSegment.P(i,3))/timeStep;
end
%fix difference in array size assuming const derivative for last iteration
numerDeriv(length(myNewTrajSegment.P),1)=numerDeriv(length(myNewTrajSegment.P
)-1,1);
numerDeriv(length(myNewTrajSegment.P),2)=numerDeriv(length(myNewTrajSegment.P
)-1,2);
numerDeriv(length(myNewTrajSegment.P),3)=numerDeriv(length(myNewTrajSegment.P
)-1,3);
myNewTrajSegment.Pd=numerDeriv;

numerDeriv2=zeros(length(numerDeriv),3); %second order derivative
%it is bad practice to calculate second order derivatives with a numerical
%model like this, but Prof K's code for Jerk trajectory doesn't support
%syms variables, so this is the only way to work around that

%forward difference numerical approx
for i=1:1:(length(numerDeriv)-1)
    numerDeriv2(i,1)=(numerDeriv(i+1,1)-numerDeriv(i,1))/timeStep;
    numerDeriv2(i,2)=(numerDeriv(i+1,2)-numerDeriv(i,2))/timeStep;
    numerDeriv2(i,3)=(numerDeriv(i+1,3)-numerDeriv(i,3))/timeStep;
end
%fix difference in array size assuming const derivative for last iteration
numerDeriv2(length(numerDeriv),1)=numerDeriv2(length(numerDeriv)-1,1);
numerDeriv2(length(numerDeriv),2)=numerDeriv2(length(numerDeriv)-1,2);
numerDeriv2(length(numerDeriv),3)=numerDeriv2(length(numerDeriv)-1,3);
myNewTrajSegment.Pdd=numerDeriv2;



myNewTrajSegment.T = transl(myNewTrajSegment.P);
%myNewTrajSegment.q = myRobot.ikine6s( myNewTrajSegment.T );
traj.P=[myNewTrajSegment.P; traj.P];

%traj.q = [ myNewTrajSegment.q ; traj.q];
%trak.qd = ...
```

```
% plot the full target Cartesian trajectory (should include your initial
segment):
figure(45); clf
plot3( traj.P(:,1), traj.P(:,2), traj.P(:,3), 'r.-'); hold on; grid on
xlabel('x [m]')
ylabel('y [m]')
zlabel('z [m]')
title('\bfDesired Trajectory (Cartesian Space)')
axis equal % maintain aspect ratio

% plot with robot
%figure(1);
%myRobot.plot([0 0 0 0 0 0]); % home pose-- robot starts here!
%hold on;
%plot3( traj.P(:,1), traj.P(:,2), traj.P(:,3), 'r.-'); hold on; grid on
% Check to ensure that what are asking the robot to do in space makes
sense...
%title('Desired Motion'); myRobot.animate( traj.q ); % comment this out
later.

myNewTrajSegment.q  = myRobot.ikine6s( myNewTrajSegment.T);
myNewTrajSegment.qd = zeros(length(timeVector), 6);  % preallocate for
speed...
myNewTrajSegment.qdd= zeros(length(timeVector), 6);
tic
for i = 1:length(timeVector)
    % compute jacobian, its inverse, and time derivative-qd product
    J    = myRobot.jacob0( myNewTrajSegment.q(i,:) );
    Jinv = inv( J );
    qd   = Jinv * [myNewTrajSegment.Pd(i,:) 0 0 0].';
    JdotQd = myRobot.jacob_dot(myNewTrajSegment.q(i,:), qd ) ;
    qdd=  Jinv * [ [myNewTrajSegment.Pdd(i,:) 0 0 0]' - JdotQd ];
    % save for each time step
    myNewTrajSegment.qd(i,:)  = qd';
    myNewTrajSegment.qdd(i,:) = qdd';
end
toc
traj.q=[myNewTrajSegment.q; traj.q];
traj.q(:,4) = 0;  % override this angle since it keeps flipping at +/- pi
                  % ...just numerical rounding error
traj.qd=[myNewTrajSegment.qd; traj.qd];
traj.qdd=[myNewTrajSegment.qdd; traj.qdd];
t_before=timeVector;
t_after=timeVector(end):0.01:(timeVector(end)+10);
t_total=[t_before t_after];
figure(432);clf
plot(t_total, traj.q ); hold on; title('Desired Joint Space Trajectories')
xlabel('time [s]'); ylabel('joint angle [rad]')
legend([['qqqqqq']' ['123456']' ] )

tf=20; %update with additional simulation for first part
t=t_total';
% send this trajectory and its derivates to Simulink :
desiredTraj_q  =  [t traj.q];  % Append time for simulink format
desiredTraj_qd = [t traj.qd ];
desiredTraj_qdd= [t traj.qdd];
```

```
% use "from workspace" block in simulink to get access to this data by name

%% Now Control the robot (you need to fix this so it does the correct
trajectory):
% Corke examples (read the Corke & Sastry sections very closely!)
%sl_fforward; % does Feedforward Control 9.4.3.1 Corke (PD Control Sastry,
4.5.3)
%sl_ctorque;  % does computed torque control 9.4.3.1 Corke (PD Control
Sastry, 4.5.3)

%% ME8287 Simulink Models for trajectory following ...
% Corke's examples aren't great for arbitrary trajectories.
% Prof. K made a trajectory-generating function that is compatible with
% simulink and modified the simulink model to use it.
% You need to edit that simulink function to invoke your own trajectory

figure(1); clf; % Corke's toolkit can only plot single robot, use same fig
open('sl_ctorqueTraj.slx');  % Different model, uses 'myRobot' and custom
% trajectory generation:  desiredTrajectory()
% note that it's a simulink function so it'll
% compile to mex and you'll need to modify it
% within Simulink to alter it
r = sim('sl_ctorqueTraj', 'StopTime', num2str(tf) )  ;

% get info out of simulink
t = r.get('tout')';
y = r.get('yout')';
q = y(     [1:6] ,  :   ); % joint angles
e = y( 6 + [1:6] ,  :   ); % error
qd  = y( 12+[1:6] , : );

% plot stuff
figure(40); clf
plot(t,q, '.'); hold on
plot(t,e,'--');
xlabel('Time [s]')
ylabel('Joint Angles [rad]')
legend([['qqqqqq' 'eeeeee']' ['123456' '123456']' ] )
title('\bfRobot Trajectory Control')

%% Plot both desired traj and actual traj
figure(46); clf

% actual
myTraj2 = myRobot.fkine( q' );
[X,Y,Z] = transl(myTraj2);
plot3( X, Y, Z, 'b.-'); hold on; grid on

% desired
plot3( traj.P(:,1), traj.P(:,2), traj.P(:,3), 'r.-'); hold on; grid on
xlabel('x [m]')
ylabel('y [m]')
zlabel('z [m]')
axis equal % maintain aspect ratio
```

```
legend ('Actual Path', 'Desired Path')

%positional error
figure(71);
posError=sqrt((X(2:end-1)'-traj.P(:,1)).^2+(Y(2:end-1)'-
traj.P(:,2)).^2+(Z(2:end-1)'-traj.P(:,3)).^2);
plot(t(1:end-2),posError);
xlabel('time [s]')
ylabel('Cartesian Position Error Magnitude [m]')

figure(80);
%find actual cartesian velocity
CartVel_torque=zeros(6,length(q(1,1:end-2)));
%find desired cartesian velocity
CartVel_desired=zeros(6,length(q(1,1:end-2)));
traj.q=traj.q';
traj.qd=traj.qd';
for i=1:1:length(q(1,1:end-2))
   CartVel_torque(:,i)=(jacob0(myRobot,q(:,i)')*qd(:,i))';
   CartVel_desired(:,i)=(jacob0(myRobot,traj.q(:,i)')*traj.qd(:,i))';
end

CartVel_norm_torque=sqrt(CartVel_torque(1,:).^2+CartVel_torque(2,:).^2+CartVe
l_torque(3,:).^2);
CartVel_norm_desired=sqrt(CartVel_desired(1,:).^2+CartVel_desired(2,:).^2+Car
tVel_desired(3,:).^2);
errorVel=sqrt((CartVel_desired(1,:)-
CartVel_torque(1,:)).^2+(CartVel_desired(2,:)-
CartVel_torque(2,:)).^2+(CartVel_desired(3,:)-CartVel_torque(3,:)).^2);
plot(t(1:end-2),errorVel);
xlabel('Time [s]')
ylabel('Cartesian Velocity Error Magnitude [m/s]')

%lets zoom in the steady state error

figure(100);
plot(t(1950:end-2),posError(1950:end));
xlabel('time [s]')
ylabel('Cartesian Position Error Magnitude [m]')

figure(101);
plot(t(1950:end-2),errorVel(1950:end));
xlabel('Time [s]')
ylabel('Cartesian Velocity Error Magnitude [m/s]')
```