HW5 ME 739 Introduction to robotics

Spring 2015
Department of Mechanical Engineering
University of Wisconsin, Madison

Instructor: Professor Michael Zinn

By

Nasser M. Abbasi

May 3, 2022

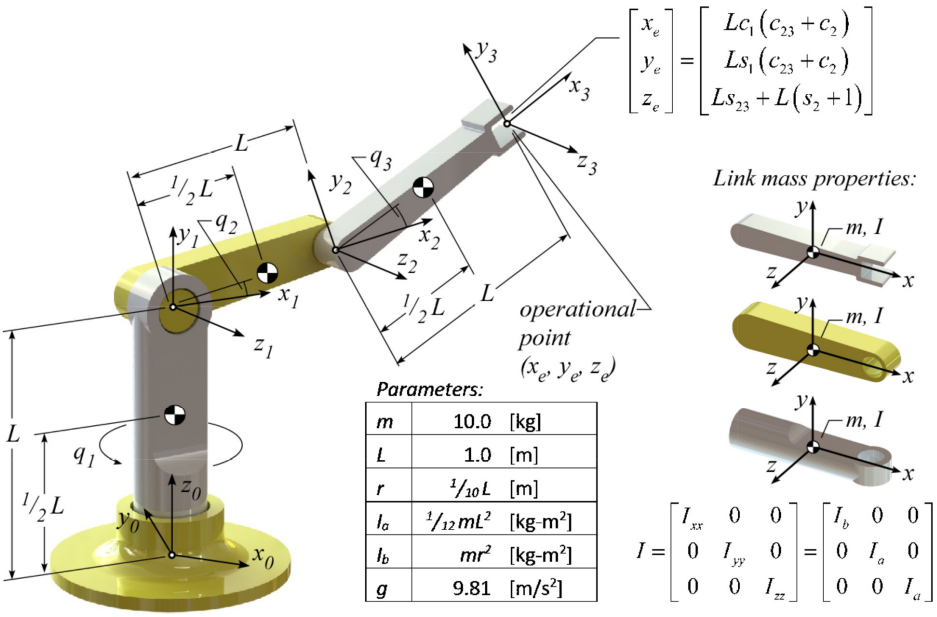
Contents

List of Tables

0.1 Problem description

Problem 1. [100 points]

You are to design a series of position controllers for the three degree of freedom manipulator shown in the figure below. The equations of motion and corresponding Matlab simulation code are given in the lecture notes (4-9 – Dynamics – EOM Simulation: slides 4-136 to 4-147). The simulation code has been posted on the course Learn@UW page (Simulation Example #2: Three DOF RRR Manipulator). For completeness, the equations of motion are given on the next page.



m	10.0	[kg]
L	1.0	[m]
r	$1/10\,L$	[m]
I_a	$1/12\,mL^2$	[kg-m^2]
I_b	mr^2	[kg-m^2]
g	9.81	[m/s^2]

Equations of motion: $D\ddot{q} + B[\dot{q}\dot{q}] + C[\dot{q}^2] + G = \tau$ where the terms are given as:

Mass matrix:

$$D = \begin{bmatrix} D_{11} & 0 & 0 \\ 0 & D_{22} & D_{23} \\ 0 & D_{32} & D_{33} \end{bmatrix} \quad \text{where} \quad \begin{aligned} D_{11} &= \tfrac{1}{4}mL^2c_2{}^2 + \tfrac{1}{4}mL^2\left(c_{23} + 2c_2\right)^2 + \cdots \\ &\quad I_b + I_a\left(c_2{}^2 + c_{23}{}^2\right) + I_b\left(s_2{}^2 + s_{23}{}^2\right) \\ D_{22} &= \tfrac{3}{2}mL^2 + mL^2c_3 + 2I_a \\ D_{23} &= \tfrac{1}{4}mL^2 + \tfrac{1}{2}mL^2c_3 + I_a \\ D_{32} &= d_{23} \\ D_{33} &= \tfrac{1}{4}mL^2 + I_a \end{aligned}$$

Coriolis matrix:

$$B(q)[\dot{q}\dot{q}] = \begin{bmatrix} B_{11} & B_{12} & 0 \\ 0 & 0 & B_{23} \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{q}_1\dot{q}_2 \\ \dot{q}_1\dot{q}_3 \\ \dot{q}_2\dot{q}_3 \end{bmatrix} \quad \text{where} \quad \begin{aligned} B_{11} &= \left(I_b - I_a - \tfrac{1}{4}mL^2\right)\sin\left(2q_2 + 2q_3\right) + \cdots \\ &\quad \left(I_b - I_a - \tfrac{5}{4}mL^2\right)\sin\left(2q_2\right) - mL^2\sin\left(2q_2 + q_3\right) \\ B_{12} &= -\tfrac{1}{2}\Big(\sin\left(q_2 + q_3\right)\Big(\left(4I_a - 4I_b + mL^2\right)\cdots \\ &\quad \cos\left(q_2 + q_3\right) + 2mL^2\cos\left(q_2\right)\Big)\Big) \\ B_{23} &= -mL^2\sin\left(q_3\right) \end{aligned}$$

Centrifugal matrix:

$$\mathrm{C}(q)[\dot{q}^2] = \begin{bmatrix} 0 & 0 & 0 \\ C_{21} & 0 & C_{23} \\ C_{31} & C_{32} & 0 \end{bmatrix} \begin{bmatrix} \dot{q}_1{}^2 \\ \dot{q}_2{}^2 \\ \dot{q}_3{}^2 \end{bmatrix} \quad \text{where} \quad \begin{aligned} C_{21} &= \tfrac{1}{2}\left(I_a - I_b + \tfrac{1}{4}mL^2\right)\sin\left(2q_2 + 2q_3\right) + \cdots \\ &\quad \tfrac{1}{2}\left(I_a - I_b + \tfrac{5}{4}mL^2\right)\sin\left(2q_2\right) + \tfrac{1}{2}mL^2\sin\left(2q_2 + q_3\right) \\ C_{23} &= -\tfrac{1}{2}mL^2\sin\left(q_3\right) \\ C_{32} &= \tfrac{1}{2}mL^2\sin\left(q_3\right) \\ C_{31} &= \tfrac{1}{4}\Big(\sin\left(q_2 + q_3\right)\Big(\left(4I_a - 4I_b + \tfrac{1}{4}mL^2\right)\cdots \\ &\quad \cos\left(q_2 + q_3\right) + 2mL^2\cos\left(q_2\right)\Big)\Big) \end{aligned}$$

Gravity vector:

$$G = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \end{bmatrix} \quad \text{where} \quad \begin{aligned} g_1 &= 0 \\ g_2 &= \tfrac{1}{2}mgL\left(3\mathrm{c}_2 + c_{23}\right) \\ g_3 &= \tfrac{1}{2}mgLc_{23} \end{aligned}$$

► ***Joint-space Control***

In parts (a) – (c) design and implement a joint-space inverse dynamics controller. Use a simple proportional-derivative (PD) controller to control the decoupled system. Set the controller gains such that the closed-loop position controller has an undamped natural frequency, ω_n, equal to 2 Hz and a damping ratio, ζ, equal to 1.0. Simulate the response of the system to a step input position command. The initial joint positions of the manipulator are given as $q_0 = [0 \quad \pi/4 \quad -\pi/2]^T$. The final (or desired) joint positions of the manipulator are given as $q_f = [\pi \quad \pi \quad \pi/2]^T$. Note, when simulating your system, you may need to adjust the integration step size of the Runge-Kutta algorithm to ensure that the system does not become ill-conditioned.

For parts (a) – (c) below, plot/animate the following:

- Plot the joint space displacements and velocities as a function of time
- Plot the operational-point (task frame origin) displacements and velocities as a function of time
- Plot the operational-point displacements in three-dimensions [i.e. `plot3(x, y, z)`]
- Animate the motion of the manipulator

(a) Design and implement a joint-space inverse dynamics controller, where the nonlinear Coriolis, centrifugal, and gravity terms are compensated and the equations are completely decoupled using the joint-space mass matrix.

(b) Design and implement a modified joint-space inverse dynamics controller. In this case, compensate for the gravitational terms but do not compensate for the nonlinear Coriolis or centrifugal terms.

(c) Design and implement a decentralized joint-space controller. In this case, compensate for the nonlinear Coriolis, centrifugal, and gravity terms but decouple the system using an *average* mass matrix. Derive an average mass matrix using the true mass matrix. Explain and justify your choice of an average (constant) mass matrix.

(d) Compare the response of the three controllers from parts (a) – (c) and comment on the differences, advantages, and disadvantages.

► ***Operational-space Control***

In parts (a) – (c), design and implement an operational-space inverse dynamics controller. Use a simple proportional-derivative (PD) controller to control the decoupled system. Set the controller gains such that the closed-loop position controller has an undamped natural frequency, ω_n, equal to 2 Hz and a damping ratio, ζ, equal to 1.0.

For parts (a) – (c) below, plot/animate the following:

- Plot the joint space displacements and velocities as a function of time
- Plot the operational-point (task frame origin) displacements and velocities as a function of time
- Plot the operational-point displacements in three-dimensions [i.e. `plot3(x, y, z)`]
- Animate the motion of the manipulator

(a) Design and implement an operational-space inverse dynamics controller, where the nonlinear Coriolis, centrifugal, and gravity terms are compensated and the equations are completely decoupled using the operational-space mass matrix[1]. Simulate the response of the system to a step input position command. The initial joint positions of the manipulator are given as $q_0 = [0 \quad \pi/4 \quad -\pi/2]^T$. The final operational-space position of the manipulator is given as $x_f = [-L \quad -L \quad 0]^T$.

(b) Using the controller designed in part (a), simulate the response of the system to a step input position command where the final operational-space position of the manipulator is given as $x_f = [-L \quad -L/10 \quad 0]^T$ (the initial joint positions of the manipulator are still given as $q_0 = [0 \quad \pi/4 \quad -\pi/2]^T$.

(c) Using the controller designed in part (a), implement the operational-space linear velocity limiting heuristic described in the lecture notes. Set the maximum linear velocity equal to 5 m/s. Simulate the response of the system to a step input position command where the initial joint positions are given as $q_0 = [0 \quad \pi/4 \quad -\pi/2]^T$ and the final operational-space positions is given as $x_f = [-L \quad -L \quad 0]^T$.

(d) Compare the response of the controllers from parts (a) and (b) comment on the differences, advantages, and disadvantages.

0.2 Joint space control

The main goal is to decouple the nonlinear equation of motion of the robotic arm which is given by the equation below, where [ ] indicates a matrix and { } indicates a column vector. Notice that $[D(q)]$ means that that matrix $[D]$ is a function of q. It does not mean the matrix $[D]$ is multiplied by q.

$$[D(q)]\{\ddot{q}\} + [B(q)][\dot{q}\dot{q}] + [C(q)]\{\dot{q}^2\} + [G(q)] = \{\tau\}$$

The sizes of each of above quantities in terms of n which is the number of generalized coordinates, or the degrees of freedom, or the number of joints, which is 3 in this example, are given by

$$\overbrace{[D(q)]}^{n\times n}\overbrace{\{\ddot{q}\}}^{n\times 1} + \overbrace{[B(q)]}^{n\times \frac{(n-1)n}{2}}\overbrace{[\dot{q}\dot{q}]}^{\frac{(n-1)n}{2}\times 1} + \overbrace{[C(q)]}^{n\times n}\overbrace{\{\dot{q}^2\}}^{n\times 1} + \overbrace{[G(q)]}^{n\times 1} = \overbrace{\{\tau\}}^{n\times 1}$$

The velocity terms and the gravity terms and the mass terms are indicated below

$$\overbrace{[D(q)]\{\ddot{q}\}}^{\text{Mass or inertia term}} + \overbrace{[B(q)][\dot{q}\dot{q}] + [C(q)]\dot{q}^2\}}^{\text{velocity nonlinear terms}} + \overbrace{[G(q)]}^{\text{gravity nonlinear term}} = \overbrace{\{\tau\}}^{\text{forces and torques}}$$

The above equation of motion can be written in simpler form for analysis by letting $V = [B(q)][\dot{q}\dot{q}] + [C(q)]\{\dot{q}^2\}$. Therefore the above becomes

$$[D]\{\ddot{q}\} + V + G = \tau$$

$$[D]\{\ddot{q}\} = \tau - V - G \tag{1}$$

Setting $\tau = \tilde{D}\tau' + \tilde{V} + \tilde{G}$ where $\tilde{D}, \tilde{V}, \tilde{G}$ are estimates of the the actual D, V, G. The estimates are computed in real time. τ' is the actuating signal generated by the propertional derivative (P.D.) controller which is given by

$$\tau' = k_d(\dot{q}_{\text{desired}} - \dot{q}_{\text{actual}}) + k_p(q_{\text{desired}} - q_{\text{actual}})$$

Using the above equation (1) becomes

$$\begin{aligned} D[\ddot{q}] &= \tilde{D}\tau' + \tilde{V} + \tilde{G} - V - G \\ &= \tilde{D}\tau' + (\tilde{V} - V) + (\tilde{G} - G) \end{aligned} \tag{2}$$

Assuming the estimates are perfect with no noise and no delay, then $\tilde{V} = V$, $\tilde{G} = G$ and $\tilde{D} = D$ and (2) reduces to

$$\begin{aligned} [I]\{\ddot{q}\} &= D^{-1}\tilde{D}\tau' \\ &= \{\tau'\} \end{aligned} \tag{3}$$

Where $[I]$ is the identity matrix. Therefore the equations have been decoupled.

τ was determined based on the control being implemented as follows

part(a)	$\tau = \tilde{D}\tau' + \tilde{G} + \tilde{V}$	full compensation
part(b)	$\tau = \tilde{D}\tau' + \tilde{G}$	no velocity terms compensation, only gravity is compensated for. This means the V term is not estimated at run time hence the equations of motion will not be fully decoupled and some cross joints motion effects will result.
part (c)	$\tau = D_{\text{average}}\tau' + \tilde{G} + \tilde{V}$	decentralized control with full compensation. Mass matrix is constant which represents the average mass matrix.

For part (c), the average mass matrix D_{average} was found by setting the joint angles to zero $q_i = 0$ for the three joints in the original $[D]$ mass matrix. Therefore all the $\cos(q)$ terms were replaced by one and all $\sin(q)$ terms were replaced by zero.

Damping was not used as the problem did not specify one but this can be easily added in the Matlab code. In addition, the gear ratio N was not used as the problem did not specify a value for N.

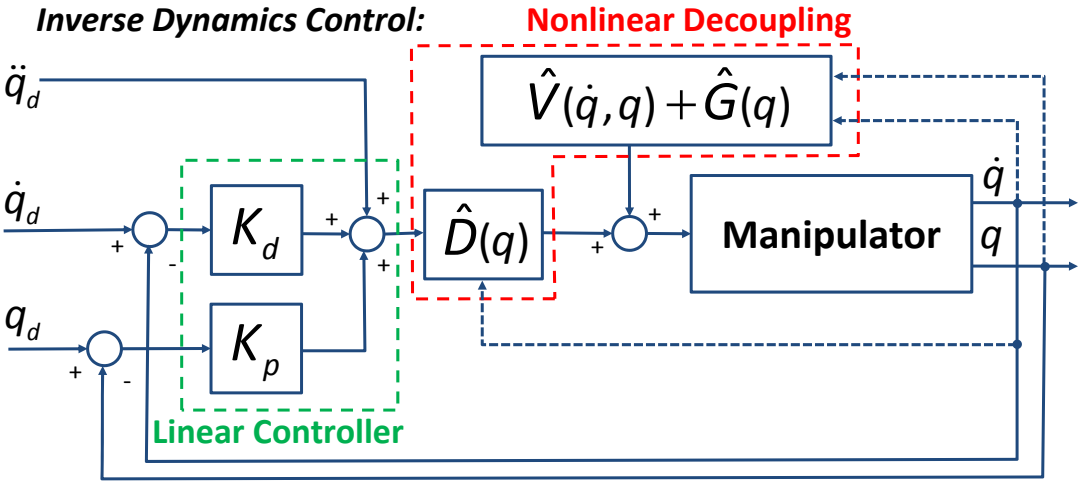
The end-effector position in task space was found to be

$$\begin{aligned} X_{\text{end}} &= T_3^0 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} L\cos(q_1(t))\cos(q_2(t)) + L\cos(q_1(t))\cos(q_2(t))\cos(q_3(t)) - L\cos(q_1(t))\sin(q_2(t))\sin(q_3(t)) \\ L\cos(q_2(t))\sin(q_1(t)) + L\cos(q_2(t))\cos(q_3(t))\sin(q_1(t)) - L\sin(q_1(t))\sin(q_2(t))\sin(q_3(t)) \\ L\left(\sin(q_2(t)) + 1\right) + L\cos(q_2(t))\sin(q_3(t)) + L\cos(q_3(t))\sin(q_2(t)) \end{bmatrix} \end{aligned}$$

The above was evaluated at each time instance and used to plot the path of the end-effector in 3D.

The following diagram shows the controller layout taken from the class handout, page 6-120 which shows the controller with full compensation.

Nonlinear Dynamic Decoupling



- decoupling torques based on *measured* manipulator states
- Requires real-time computation of nonlinear decoupling terms
- Nonlinear terms affected by sensor noise and delay

6-120

Figure 1: Diagram of controller, inverse dynamics control, full compensation. From class lecture notes

In the Matlab simulation, it was assumed that the measured manipulator states are exact and no noise was present as mentioned above. In practice this will not be the case and there will be an error in the estimates.

The following diagrams gives a high level overview of the Matlab software design for the implementation of the joint space control and the M files used.

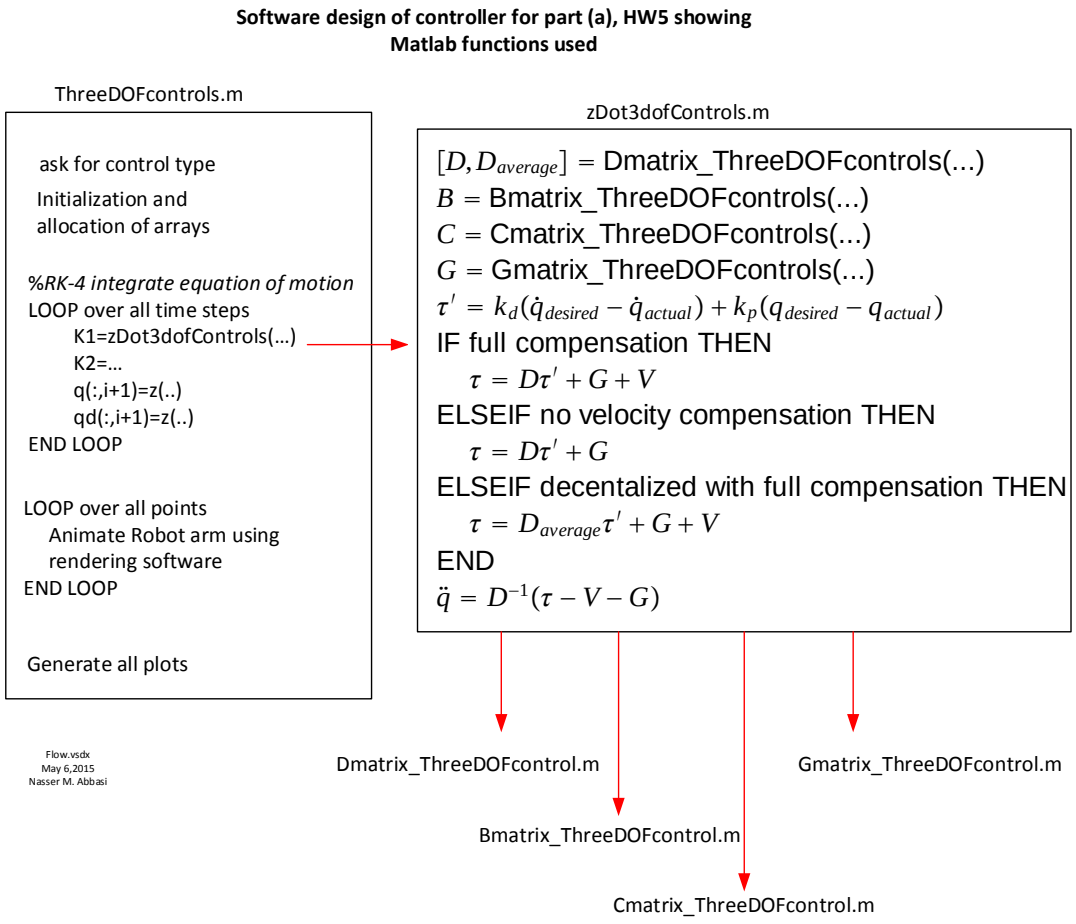


Figure 2: Matlab program design for joint space controller

The output for each part is now given followed by discussion of the results.

0.2.1 Part (a) Coriolis, centrifugal, and gravity terms are compensated

In this part, full compensation was made for the nonlinear Coriolis, centrifugal and gravity terms. This gave complete decoupling between the joints. Becuase of this one expects no oscillation in the plots of the joints displacements over the time of the simulation. This was verified from the plots generated by the simulation.

In addition to the required plots, an additional plot was made showing the end-effector speed over time. This was found by finding the end-effector linear speed using $\dot{X} = J\dot{q}$ where J is the end-effector Jacobian. The Jacobian was found using symbolic matlab in the file `ThreeDOFcontrols_symbolic.m` and the output was used in the the file `ThreeDOFcontrol.m` in order to calculate $|\dot{X}|$ over each time step.

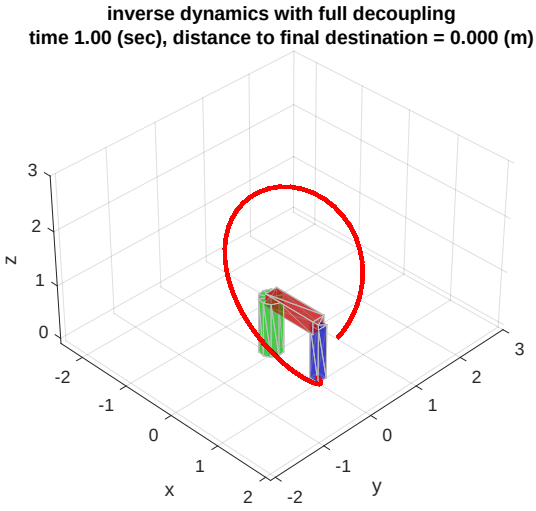


Figure 3: Final position of robot arm, Joint space control, part(a)

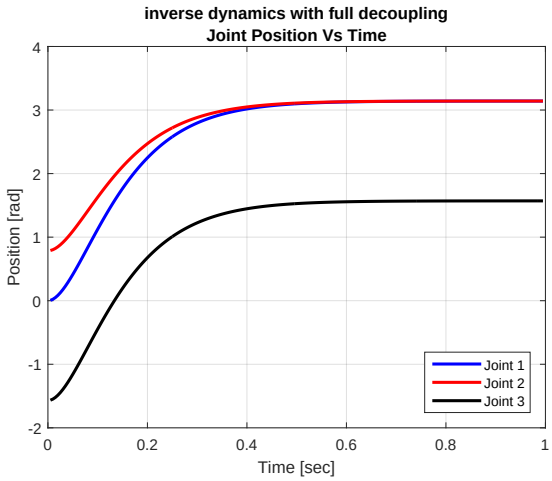


Figure 4: Joint position vs. time, Joint space control, part(a)

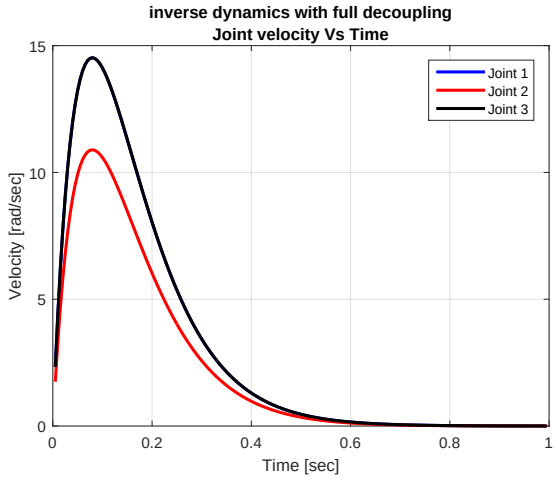


Figure 5: Joint velocity vs. time, Joint space control, part(a)

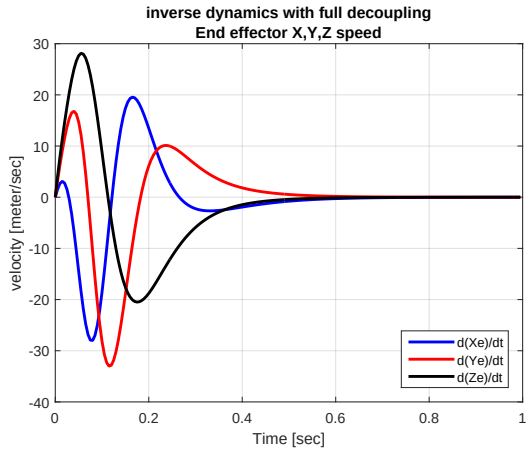


Figure 6: End effector $\frac{dX_e}{dt}$, $\frac{dY_e}{dt}$, $\frac{dZ_e}{dt}$ Joint space control, part(a)

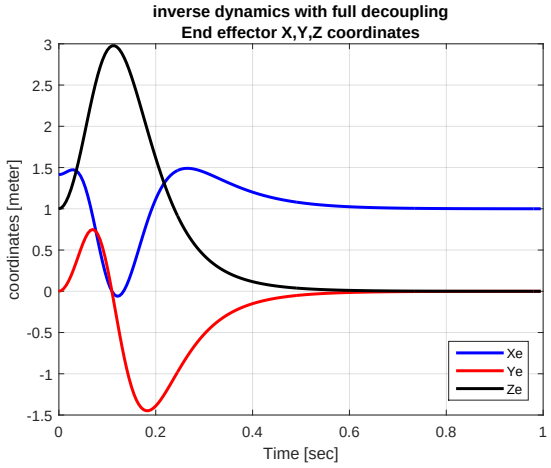


Figure 7: End effector X_e, Y_e, Z_e vs. time, Joint space control, part(a)

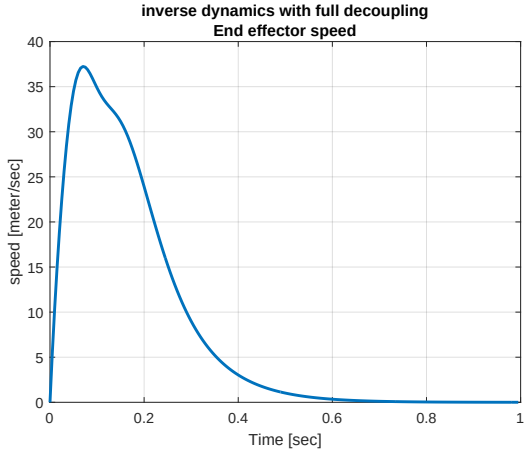


Figure 8: End effector linear speed vs. time, Joint space control, part(a)

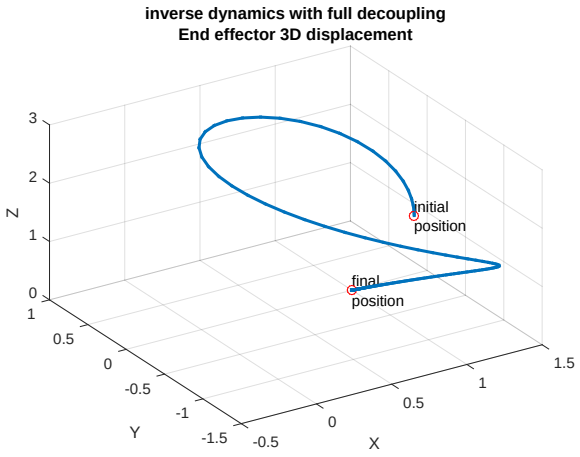


Figure 9: End effector plot3 displacement, Joint space control, part(a)

0.2.2 Part (b) No velocity compensation. Only gravity compensation

In this part, only the gravity terms were compensated for. Since there is coupling that remains between the joints, one expects to see some oscillation in the joints speeds as they are no longer independent from each other as with part (a).

This was verified by the plots generated from the simulation.

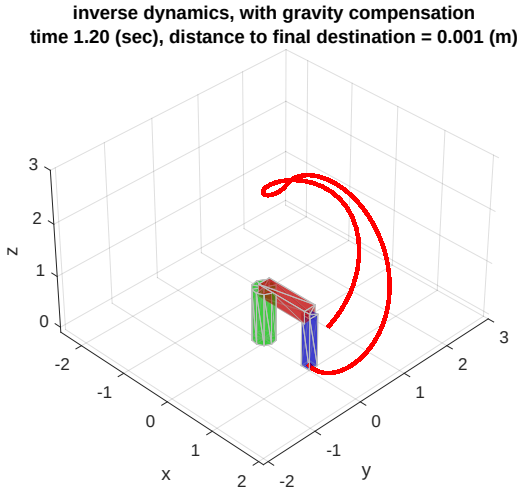


Figure 10: Final position of robot arm, Joint space control, part(b)

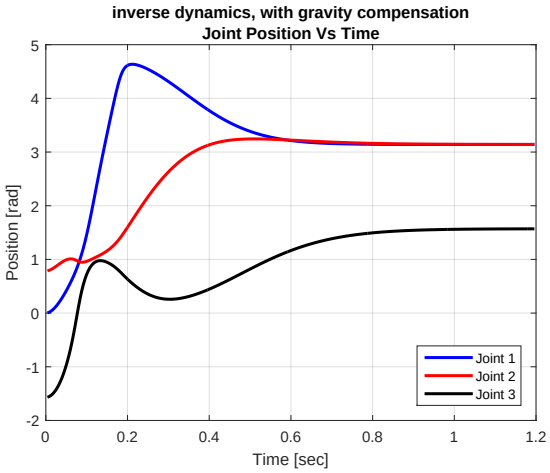


Figure 11: Joint position vs. time, Joint space control, part(b)

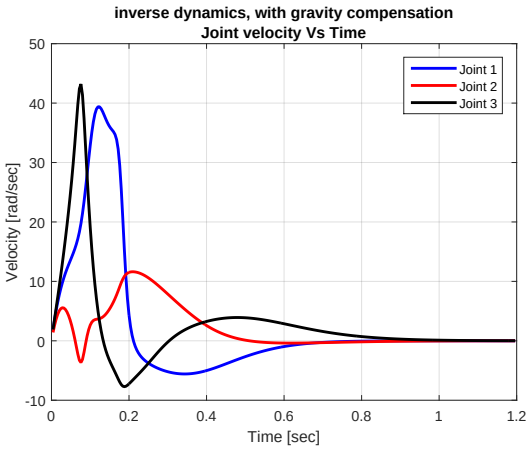


Figure 12: Joint velocity vs. time, Joint space control, part(b)

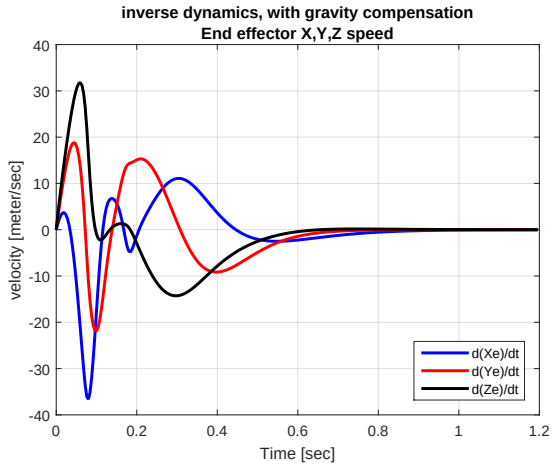


Figure 13: End effector $\frac{dX_e}{dt}, \frac{dY_e}{dt}, \frac{dZ_e}{dt}$ Joint space control, part(b)

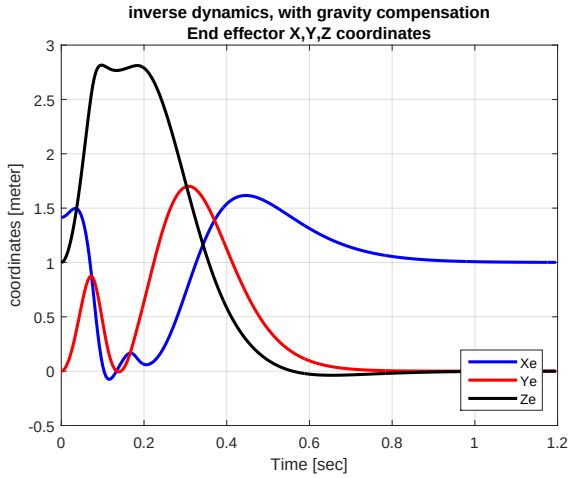


Figure 14: End effector X_e, Y_e, Z_e vs. time, Joint space control, part(b)

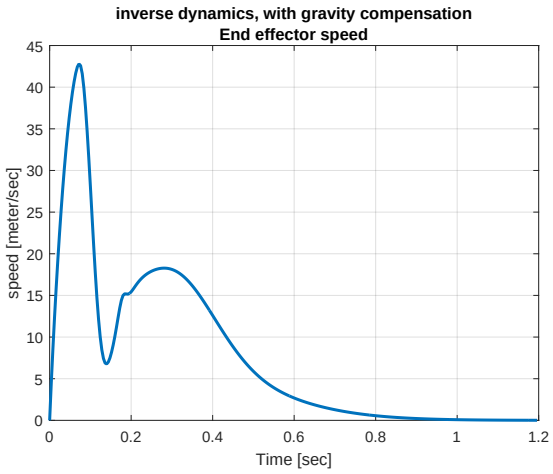


Figure 15: End effector linear speed vs. time, Joint space control, part(b)

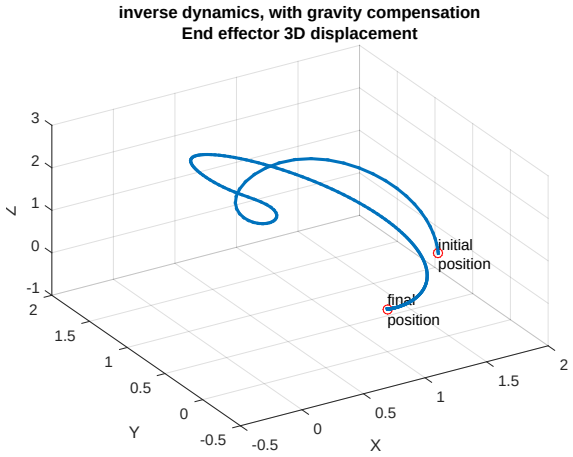


Figure 16: End effector plot3 displacement, Joint space control, part(b)

0.2.3 Part (c) Decentralized joint-space controller, full compensation using average D matrix

In this part, full compensation were made for the nonlinear Coriolis, centrifugal, and gravity terms, however the mass matrix D used was a constant matrix which represented the average of the original mass matrix.

The approximate mass matrix is given by

$$\tilde{D} = D_{\text{average}} + J_m^T I_m J_m$$

Where I_m is the actuator moment of inertia and J_m is the actuator Jacobian. In this problem these were not used as they were not specified, and only the average mass matrix needed to be determined.

The average mass matrix D_{average} was obtained from D by setting each joint position given by q to zero and by also setting the off diagonal elements to zero. Setting the off diagonal elements to zero was needed as the average mass matrix needs to be diagonal to produce the decoupling effect.

Therefore all the cosine terms were set to one and all the sine terms were set to zero. The original D matrix is

$$[D] = \begin{bmatrix} \frac{1}{4}mL^2c_2^2 + \frac{1}{4}mL^2(c_{23}+2c_2)^2 + I_b + I_a(c_2^2+c_{23}^2) + I_b(s_2^2+s_{23}^2) & 0 & 0 \\ 0 & \frac{3}{2}mL^2 + mL^2c_3 + 2I_a & \frac{1}{4}mL^2 + \frac{1}{2}mL^2c_3 + I_a \\ 0 & \frac{1}{4}mL^2 + \frac{1}{2}mL^2c_3 + I_a & \frac{1}{4}mL^2 + I_a \end{bmatrix}$$

Hence the average D matrix becomes

$$[D_{\text{average}}] = \begin{bmatrix} \frac{1}{4}mL^2 + \frac{1}{4}mL^2(1+2)^2 + I_b + I_a(1+1) + I_b(0+0) & 0 & 0 \\ 0 & \frac{3}{2}mL^2 + mL^2 + 2I_a & 0 \\ 0 & 0 & \frac{1}{4}mL^2 + I_a \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{4}mL^2 + \frac{9}{4}mL^2 + I_b + 2I_a & 0 & 0 \\ 0 & \frac{3}{2}mL^2 + mL^2 + 2I_a & 0 \\ 0 & 0 & \frac{1}{4}mL^2 + I_a \end{bmatrix}$$

Using the above $[D_{\text{average}}]$ the following plots shows the output obtained.

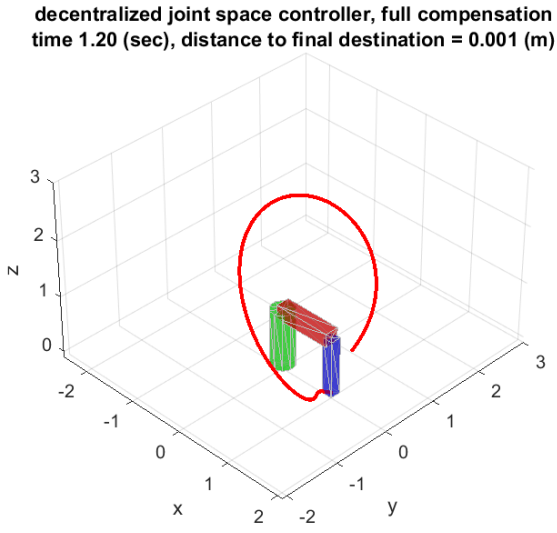


Figure 17: Final position of robot arm, Joint space control, part(c)

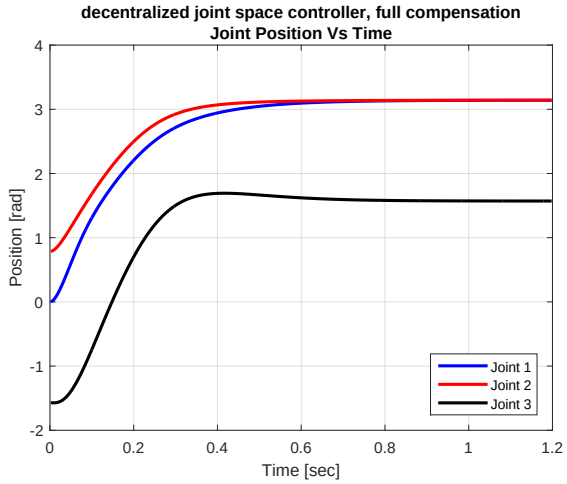


Figure 18: Joint position vs. time, Joint space control, part(c)

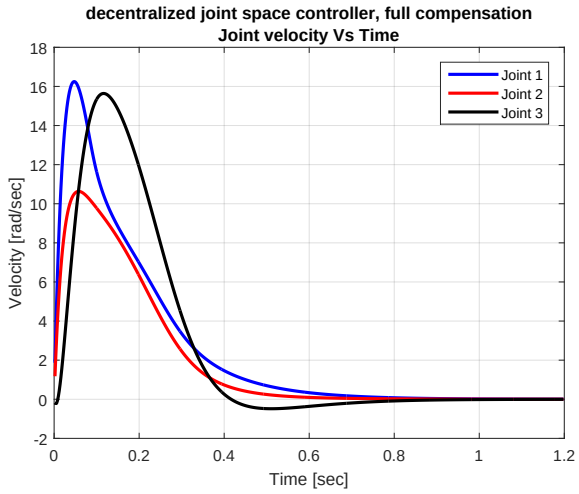


Figure 19: Joint velocity vs. time, Joint space control, part(c)

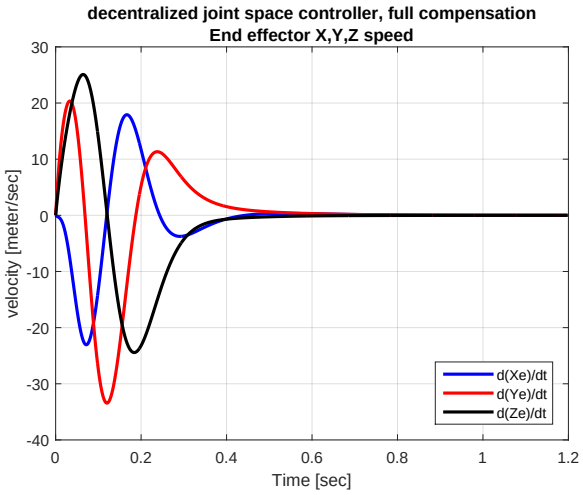


Figure 20: End effector $\frac{dX_e}{dt}, \frac{dY_e}{dt}, \frac{dZ_e}{dt}$ Joint space control, part(c)

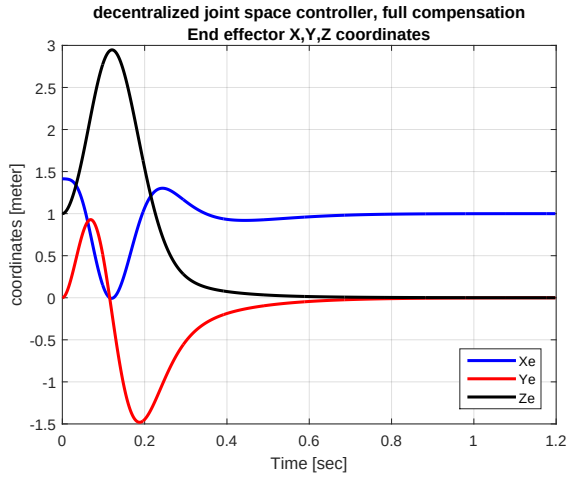


Figure 21: End effector X_e, Y_e, Z_e vs. time, Joint space control, part(c)

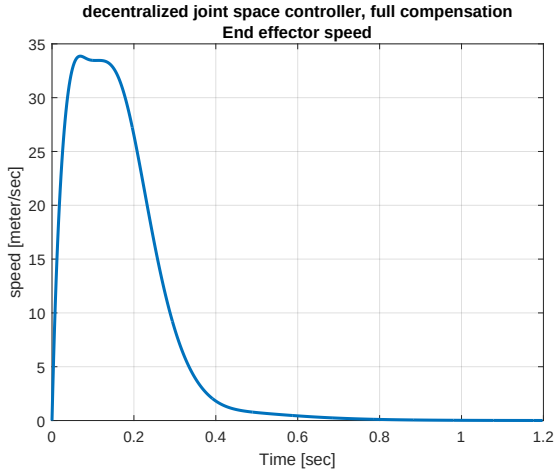


Figure 22: End effector linear speed vs. time, Joint space control, part(c)

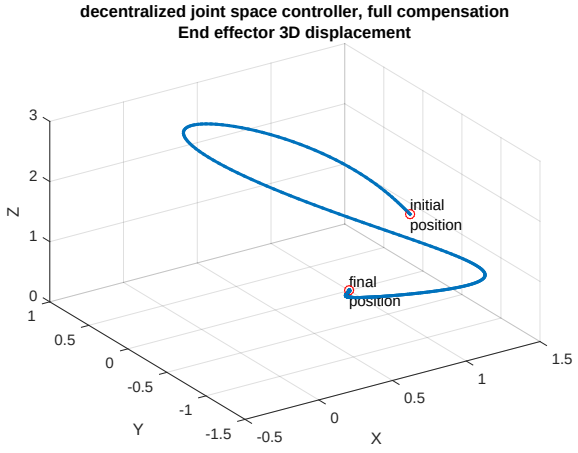


Figure 23: End effector plot3 displacement, Joint space control, part(c)

0.2.4 Part (d) discussion of result, compare control methods

There are two main methods for nonlinear dynamics decoupling. These are the computed torque method and the inverse dynamics control. In this problem the inverse dynamics control method was used. In this method, decoupling is based on measured or estimated manipulator states as described above.

The advantages and disadvantages of each controller are given below followed by a discussion on the output.

Full compensation

advantages Becuase all the non-linear terms were compensated for, this produced a smooth

motion with no overshoot. In addition, the time step for the integration during simulation was not required to be too small.

disadvantages In practice, this requires measurements in real time of all the states in order to compute and estimate the current Mass, Coriolis, centrifugal and gravity matrices at each sample time. This can be expensive and require a fast CPU. Also compensating for noise and delay in measurements makes this more complicated and there will always be some error in the estimates made.

No velocity terms compensation, only gravity compensation

advantages The main advantage is that in practice this controller will be less complicated as the Coriolis and centrifugal terms do not need to be measured and computed at each time step. This will reduce the cost and make it faster to operate.

disadvantages By not compensating for velocity terms, coupling between the joints motion remains. This can be seen by the overshoot in joints motion from the desired value and the oscillation in motion, even with the use of critical damping which should produce no overshoot. This can be severe disadvantage for an end-effector which is required not to overshoot and possibly hit the target as it approaches it.

Decentralized controller with full compensation

advantages Since the mass matrix is constant, this reduces the computation in real time, as the mass matrix do not have to evaluated at each time sample as with the other controllers. This makes the controller simpler to implement.

disadvantages Since the mass matrix is the average, it is an approximation of the real mass matrix. This can produce errors. In the simulation it was found that a smaller time step was needed for the numerical integration to reduce the overshoot. Even with a much smaller time step, one joint had very small amount of overshoot. In practice this might require a small sampling time and faster CPU to implement.

The following discussion gives a review of the output of each controller.

In part(a), full decoupling was made, which means each joint motion was independent of the other joints. Comparing the joint position vs. time plot generated in part (a) shows that the joints motion was smooth and had no oscillation since it was not affected by other joints motions. There was no overshoot in the joint positions since the damping ratio is set to be critical.

While in part(b), where no velocity compensation was made (these are the centrifugal and Coriolis terms) but only gravity was compensated for, the joint motion that resulted had oscillation in it which showed as well in the speed profile which was not as smooth compared to speed profile of the full decoupling case in part (a).

In addition, there was an overshoot in the joint position which was most apparent in joint 1 motion. This can cause problems in applications where the end-effector must approach the target without hitting it.

Part(c) initially showed some small oscillation in the motion of the joints when compared to part(a). However, this turned out to be due to using a large time step for the Runge Kutta integration. By reducing the time step to smaller value than that used in part (a) and part (b), the result improved and showed no oscillation in the joint positions nor in the joint velocities.

The time step used for part(c) was 0.002 seconds, while for part(a) it was 0.005 seconds. The simulation time to converge to the final destination did not changed compared to part (a). The only change needed was to reduce the time step.

However in part (c), there was a very small overshoot in the motion of joint 3 around 0.35 second as can be seen in the plot.

The mass matrix for part (c), which is the average mass matrix D_{average}, is a constant matrix as described above, found by eliminating all the variable terms in the entries of the original mass matrix D by setting q to zero and updating the corresponding cosine and sine terms accordingly and by also setting the off-diagonal elements to zero to insure decoupling of the equations.

Of the above three controllers, the first one (part(a)) produced the best result (fast convergence to target and no oscillation in joints motion). Part(c) was similar to part(a), except for need to use much smaller integration time step. However, part (c) is the simplest controller and can make the implementation faster since the mass matrix used is not as complicated as the other two methods as it is a constant and hence no need to estimate it at run time. Therefore it is simpler method to execute and can be faster in practice.

The following diagram shows the joint position vs. time for the three controllers next to each others to make it easier to compare and contrast. Part (a) is similar to part(c), except for the small overshoot in joint 3 using part (c). Part(b) clearly did not produce good joint motion with large overshoot and large oscillations.

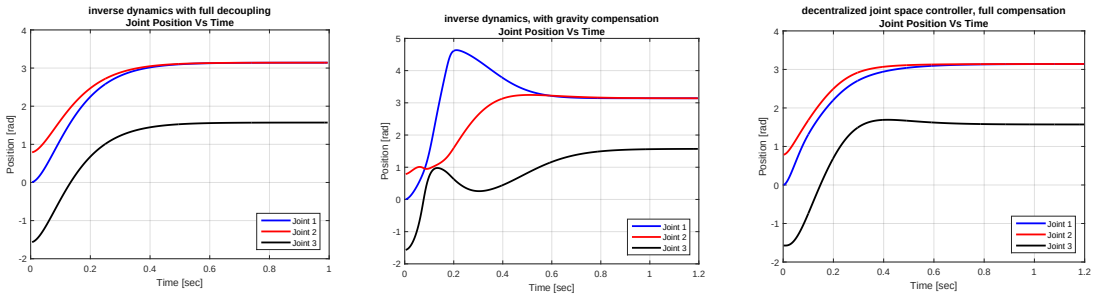


Figure 24: Joints positions vs. time, part(a),(b),(c) side by side

The following diagram shows the joint velocity vs. time for the three controller next to each others to make it easier to compare. Part(b) had the most oscillations. In Part(a), joint 1 had the same joint velocity as joint 2, and that is why the blue line in the plot did not show as it is below the black line. For part(c) this was not the case.

Even though each joint had smooth velocity profile, joint 1 and joint 2 did not have the same velocity profile as with the full decoupling case of part(a).

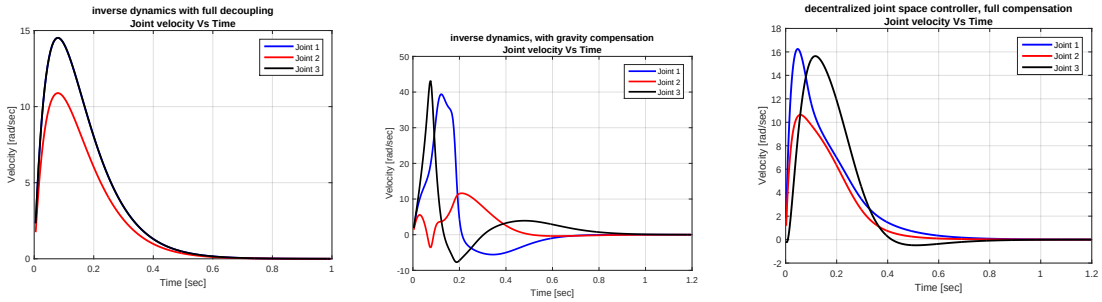


Figure 25: Joints velocity vs. time, part(a),(b),(c) side by side

The following diagram shows the end effector speed for the three control methods side by side. As discussed above, this was generated using $\dot{X} = J\dot{q}$ where J is the basic Jacobian for the end effector. This shows the end-effector speed profile in part (c) was similar to part (a), while Part(b) end-effector speed profile showed the effect of the coupling that remained between the joints where there was a number of places where an accelerations and deceleration showed up over the full time of the simulation. The motion was not as smooth as the other two methods.

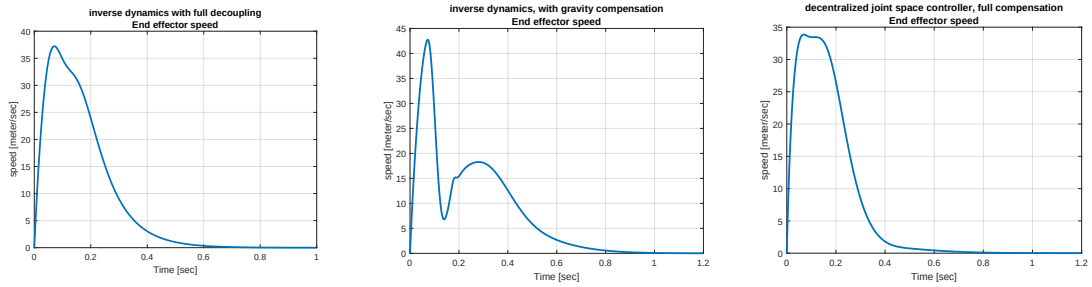


Figure 26: end effector speed, part(a),(b),(c) side by side

0.2.5 source code listing for joint-space control

The following is the Matlab source code listing which implements the joint based control part of the HW.

To run the script for this part, the command is

```
ThreeDOFcontrols
```

ThreeDOFcontrols.m

```
%file ThreeDOFcontrols.m
%This the main script used to implement HW5, part a.
%Modified original code from ME739 UW learn@UW, Madison by Professor Zinn
%
%Nasser M. Abbasi  5/10/2015

%--------------------------------------------------------------------------
%  NUMERICAL INTEGRATION OF DYNAMIC EQUATIONS
%--------------------------------------------------------------------------

%clear all;
close all; clc;

% set the model parameters per the HW problem.
modelParameters = InitializeThreeDOFmodel();

% Ask use for which part to run. There are 3 types of controllers
disp('Specify control method:')
disp('   1 = option(a), inverse dynamics with full decoupling')
disp('   2 = option(b), inverse dynamics - with gravity compensation')
disp('   3 = option(c), decentralized joint space controller')
modelParameters.controlMethod = input(' ');

%depending on the control type, set different values. The decentralized
%was found to require a smaller step size for RK-4 to behave well.
if modelParameters.controlMethod ==1
    tend    = 1;          %simulation run time
    dT      = .005;       %integration step size
    title_add_on = 'inverse dynamics with full decoupling';
elseif modelParameters.controlMethod ==2
    tend    = 1.3;
    dT      = .005;
    title_add_on = 'inverse dynamics, with gravity compensation';
else
    tend    = 1.25;
    dT      = .002;
    title_add_on = 'decentralized joint space controller, full compensation';
end;

numPts  = floor(tend/dT);

%--------------------------------------------------------------------------
%  RENDERING INITIALIZATION
%--------------------------------------------------------------------------
L           = modelParameters.L;
L1          = L;
L2          = L;
L3          = L;
f_handle    = 1;
axis_limits = L*[-2.5 2.1 -2.1 3 -.1 3];
render_view = [1 -1 1]; view_up = [0 0 1];
SetRenderingViewParameters(axis_limits,render_view,view_up,f_handle);
```

```
camproj perspective

%----initialize rendering

% link 1 rendering initialization
r1         = L1/5;
sides1     = 10;
axis1      = 2;
norm_L1    = 1.0;
linkColor1 = [0 0.75 0];
plotFrame1 = 0;
d1         =  CreateLinkRendering(L1,r1,sides1,axis1,norm_L1,linkColor1,...
    plotFrame1,f_handle);

% link 2 rendering initialization
r2         = L2/6;
sides2     = 4;
axis2      = 1;
norm_L2    = 1.0;
linkColor2 = [0.75 0 0];
plotFrame2 = 0;
d2         =  CreateLinkRendering(L2,r2,sides2,axis2,norm_L2,linkColor2,...
    plotFrame2,f_handle);

% link 3 rendering initialization
r3         = L2/8;
sides3     = 4;
axis3      = 1;
norm_L3    = 1.0;
linkColor3 = [0 0 0.75];
plotFrame3 = 0;
d3         =  CreateLinkRendering(L3,r3,sides3,axis3,norm_L3,linkColor3,...
    plotFrame3,f_handle);

q       = zeros(3,numPts);
dq      = zeros(3,numPts);
t       = zeros(1,numPts);
q(:,1)  = [0; pi/4; -pi/2];   %initial position
%q(:,1) = [0; 0.01; 0];   %initial position
qd(:,1) = [0; 0; 0];          %initial velocity
z       = [q(:,1); qd(:,1)];  %initialize the state variables
qDes    = [pi;pi;pi/2];       %desired final joint space position
%qDes   = [pi/3;pi/2;pi/4];       %desired final joint space position
qdDes   = [0; 0; 0];          %desired final joint velocity
zDes    = [qDes; qdDes];
Xend    = zeros(3,numPts);    %end effector coordinates
Xvend   = zeros(3,numPts);    %end effector linear velocity

% integrate equations of motion,     % Runge-Kutta 4th order
for i = 1:numPts-1
    k1 = zDot3dofControls(z,zDes,modelParameters);
    k2 = zDot3dofControls(z + 0.5*k1*dT,zDes,modelParameters);
    k3 = zDot3dofControls(z + 0.5*k2*dT,zDes,modelParameters);
    k4 = zDot3dofControls(z + k3*dT,zDes,modelParameters);
```

```
    z = z + (1/6)*(k1 + 2*k2 + 2*k3 + k4)*dT;

    % store joint position and velocity for post processing
    q(:,i+1)  = z(1:3);
    qd(:,i+1) = z(4:6);
    t(1,i+1)  = t(1,i) + dT;
end
%-------------------------------------------------------------------
%  DISPLAY INTERATION RESULTS
%-------------------------------------------------------------------

for i = 1:numPts
    q1 = q(1,i);
    q2 = q(2,i);
    q3 = q(3,i);

    % Update frame {1}
    c   = cos(q1);
    s   = sin(q1);
    L   = L1;
    T10 = [c  0   s  0
        s  0  -c  0
        0  1   0  L
        0  0   0  1];

    % Update frame {2}
    c   = cos(q2);
    s   = sin(q2);
    L   = L2;
    T21 = [c  -s  0  L*c
        s   c  0  L*s
        0   0  1   0
        0   0  0   1];

    % Update frame {3}
    c   = cos(q3);
    s   = sin(q3);
    L   = L3;
    T32 = [c  -s  0  L*c
        s   c  0  L*s
        0   0  1   0
        0   0  0   1];
    T20 = T10*T21;
    T30 = T20*T32;

    % end-effector position
    Xend(:,i) = T30(1:3,4);

    %to find end effector velocity, we use X' = J* q' where the Jacobian
    %for the end effector is found in the Three_DOF_symbolic.m script
    %as part of this HW, in the same folder as this file.

    J=zeros(3,3);
    J(1,1)=L*sin(q1)*sin(q2)*sin(q3) - L*cos(q2)*cos(q3)*sin(q1) - L*cos(q2)*sin(q1);
```

```
    J(1,2)=-cos(q1)*(L*(sin(q2) + 1) - L + L*cos(q2)*sin(q3) + L*cos(q3)*sin(q2));
    J(1,3)=-cos(q1)*(L*cos(q2)*sin(q3) + L*cos(q3)*sin(q2));
    J(2,1)=L*cos(q1)*cos(q2) + L*cos(q1)*cos(q2)*cos(q3) - L*cos(q1)*sin(q2)*sin(q3);
    J(2,2)=-sin(q1)*(L*(sin(q2) + 1) - L + L*cos(q2)*sin(q3) + L*cos(q3)*sin(q2));
    J(2,3)=-sin(q1)*(L*cos(q2)*sin(q3) + L*cos(q3)*sin(q2));
    J(3,1)=0;
    J(3,2)=cos(q1)*(L*cos(q1)*cos(q2) + L*cos(q1)*cos(q2)*cos(q3) - L*cos(q1)*sin(q2)*sin(q3)) + sin(q
    J(3,3)=cos(q1)*(L*cos(q1)*cos(q2)*cos(q3) - L*cos(q1)*sin(q2)*sin(q3)) + sin(q1)*(L*cos(q2)*cos(q3

    Xvend(:,i) = J*qd(:,i);

    % update rendering
    figure(f_handle);
    UpdateLink(d1,T10);
    UpdateLink(d2,T20);
    UpdateLink(d3,T30);
    title(sprintf('%s\ntime %3.2f (sec), distance to final destination = %4.3f (m)',...
        title_add_on,i*dT,norm(Xend(:,i)-[1;0;0])));
    hold on;
    plot3(Xend(1,1:i),Xend(2,1:i),Xend(3,1:i),'r','LineWidth',2);

    if i == 1;  %pause at start of simulation rendering
        pause;
    end
    drawnow;
end

%-----------------------------------------------------------------------
%  PLOT JOINT POSITIONS
%-----------------------------------------------------------------------

figure(2);
plot(t(2:end),q(1,2:end),'b','LineWidth',2); hold on
plot(t(2:end),q(2,2:end),'r','LineWidth',2); hold on
plot(t(2:end),q(3,2:end),'k','LineWidth',2); hold off
title(sprintf('%s\n%s',title_add_on,'Joint Position Vs Time'));
xlabel('Time [sec]'); ylabel('Position [rad]');
grid on
legend('Joint 1','Joint 2','Joint 3','Location','southeast');

%-----------------------------------------------------------------------
%  PLOT JOINT Velocities
%-----------------------------------------------------------------------

figure(3);
plot(t(2:end),qd(1,2:end),'b','LineWidth',2); hold on
plot(t(2:end),qd(2,2:end),'r','LineWidth',2); hold on
plot(t(2:end),qd(3,2:end),'k','LineWidth',2); hold off
title(sprintf('%s\n%s',title_add_on,'Joint velocity Vs Time'));
xlabel('Time [sec]'); ylabel('Velocity [rad/sec]');
grid on
legend('Joint 1','Joint 2','Joint 3','Location','northeast');

%-----------------------------------------------------------------------
```

```
%  PLOT end effector X,Y,Z velocites
%-------------------------------------------------------------------------

figure(4);
plot(t,Xvend(1,:),'b','LineWidth',2); hold on
plot(t,Xvend(2,:),'r','LineWidth',2); hold on
plot(t,Xvend(3,:),'k','LineWidth',2); hold off
title(sprintf('%s\n%s',title_add_on,'End effector X,Y,Z speed'));
xlabel('Time [sec]'); ylabel('velocity [meter/sec]');
grid on
legend('d(Xe)/dt','d(Ye)/dt','d(Ze)/dt','Location','southeast');

%-------------------------------------------------------------------------
%  PLOT end effector X,Y,Z positions
%-------------------------------------------------------------------------

figure(5);
plot(t,Xend(1,:),'b','LineWidth',2); hold on
plot(t,Xend(2,:),'r','LineWidth',2); hold on
plot(t,Xend(3,:),'k','LineWidth',2); hold off
title(sprintf('%s\n%s',title_add_on,'End effector X,Y,Z coordinates'));
xlabel('Time [sec]'); ylabel('coordinates [meter]');
grid on
legend('Xe','Ye','Ze','Location','southeast');

%-------------------------------------------------------------------------
%  PLOT end effector speed (magnitude)
%-------------------------------------------------------------------------

figure(6);
plot(t, sqrt(sum(Xvend.^2,1)),'LineWidth',2);
title(sprintf('%s\n%s',title_add_on,'End effector speed'));
xlabel('Time [sec]'); ylabel('speed [meter/sec]');
grid on

%-------------------------------------------------------------------------
%  PLOT end effector displacement in 3D
%-------------------------------------------------------------------------

figure(7);
plot3(Xend(1,:),Xend(2,:),Xend(3,:),'LineWidth',2); hold on;
plot3(Xend(1,1),Xend(2,1),Xend(3,1),'ro');
text(Xend(1,1),Xend(2,1),Xend(3,1),{'initial','position'});
plot3(Xend(1,end),Xend(2,end),Xend(3,end),'ro');
text(Xend(1,end),Xend(2,end),Xend(3,end),{'final','position'});
title(sprintf('%s\n%s',title_add_on,'End effector 3D displacement'));
xlabel('X'); ylabel('Y'); zlabel('Z');
grid on
```

InitializeThreeDOFmodel.m

```
function modelParameters = InitializeThreeDOFmodel

    % set model parameters

    % gravitational constant [m/s^2]
    g = 9.81;

    % link mass [kg]
    m = 10;

    % link length [m]
    L = 1;

    % link COM location [m]
    Lc = L/2;

    % link radius [m]
    r = 0.1*L;

    % link inertia (_|_ to  link's CL) [kg/m^2]
    Ia = (1/12)*m*L^2;
    Ib = m*r^2;

    % assign values of model parameter structure
    modelParameters.g = 9.81; % gravitational constant [m/s^2]
    modelParameters.m = m;   % link mass [kg]
    modelParameters.L = L;    % link length [m]
    modelParameters.Lc = Lc;  % link COM location [m]
    modelParameters.Ia = Ia;   % inertia (_|_ to  link's CL) [kg/m^2]
    modelParameters.Ib = Ib;   % inertia (colinear to link's CL) [kg/m^2]
    modelParameters.controlMethod = 1;
end
```

zDot3dofControls.m

```
function [zDot] = zDot3dofControls(z,zDes,modelParameters)

    % assign joint displacements / velocities from state variables
    q      = z(1:3);
    qd     = z(4:end);
    qDes   = zDes(1:3);
    qdDes  = zDes(4:end);

    % mass and "average" mass matrix calculation
    [D,Davg] = Dmatrix_ThreeDOFcontrols(q,modelParameters);

    % calculate D, B, D, and G matrices
    B = Bmatrix_ThreeDOFcontrols(q,modelParameters);
    C = Cmatrix_ThreeDOFcontrols(q,modelParameters);
    V = B*[qd(1)*qd(2);qd(1)*qd(3);qd(2)*qd(3)]...
```

```
        +C*[qd(1)^2;  qd(2)^2; qd(3)^2];

    % gravity vector
    G = Gvector_ThreeDOFcontrols(q,modelParameters);

    % decoupled system PD-controller torques
    wn       = 2*2*pi; %using 2Hz per HW problem specs
    zeta     = 1;  %critical damping, per HW problem specs
    Kp       = wn^2;
    Kd       = 2*zeta*wn;
    tauPrime = Kd*(qdDes - qd) + Kp*(qDes - q);  %PD controller


    % calculate total control torques (PD control + nonlinear decoupling)
    % make up perfect estimate for simulation only
    G_estimate = G;
    V_estimate = V;
    D_estimate = D;
    if modelParameters.controlMethod == 1
         % inverse dynamics with full decoupling
         tau = D_estimate*tauPrime + G_estimate + V_estimate;
    elseif modelParameters.controlMethod == 2
        %inverse dynamics - with gravity compensation
        tau = D_estimate*tauPrime + G_estimate;
    elseif modelParameters.controlMethod == 3
        %decentralized joint space controller, full compensation but
        %use average D matrix, a constant matrix
        tau = Davg*tauPrime + G_estimate + V_estimate;
    end

    %form joint acceleration vector
    qdd = D\(tau -V - G);
    % assign state variable derivatives
    zDot = [qd; qdd];

end
```

Three_DOF_symbolic.m

```
%file Three_DOF_symbolic.m
%used for solving HW5, ME 739.
%finds the Jacobian also finds the derivative of the Jacobian,
%needed for part(b) of the HW5 problem
%
%Modifield slightly from original code from class web site.
%I changed the notation to T_0_1 instead of T_1_0, since this makes
%it more clear to me.

%Nasser M. Abbasi

clear all; clc;
```

```
MAKE_FUNCTION = 1;

syms q1 q2 q3 L1 L2 L3 Lc1 Lc2 Lc3 m1 m2 m3 g dq1 dq2 dq3
syms c1 c2 c3 s1 s2 s3
syms Ixx1 Ixx2 Ixx3 Iyy1 Iyy2 Iyy3 Izz1 Izz2 Izz3
syms Ia Ib L m

% simplifying assumptions
Ixx1 = Ia;  Iyy1 = Ib;  Izz1 = Ia;
Ixx2 = Ib;  Iyy2 = Ia;  Izz2 = Ia;
Ixx3 = Ib;  Iyy3 = Ia;  Izz3 = Ia;
L1 = L;  L2 = L;  L3 = L;
Lc1 = L/2;  Lc2 = L/2;  Lc3 = L/2;
m1 = m;  m2 = m;  m3 = m;

gVector = [0; 0; -g];
Ic1 = [Ixx1   0    0
        0   Iyy1   0
        0     0  Izz1];
Ic2 = [Ixx2   0    0
        0   Iyy2   0
        0     0  Izz2];
Ic3 = [Ixx3   0    0
        0   Iyy3   0
        0     0  Izz3];


disp('Evaluating kinematics')
% calculate homogeneous transformation matrices to the link center of mass
%   => Link 1
c = cos(q1); s = sin(q1);
T_0_1 = [c  0  s  0
         s  0 -c  0
         0  1  0  L1
         0  0  0  1];
T_0_c1 = [c  0  s   0
          s  0 -c   0
          0  1  0  Lc1
          0  0  0   1];
%   => Link 2
c = cos(q2); s = sin(q2);
T_1_2 = [c  -s  0  L2*c
         s   c  0  L2*s
         0   0  1   0
         0   0  0   1];
T_1_c2 = [c  -s  0  Lc2*c
          s   c  0  Lc2*s
          0   0  1    0
          0   0  0    1];
%   => Link 3
c = cos(q3); s = sin(q3);
T_2_3 = [c  -s  0  L3*c
         s   c  0  L3*s
         0   0  1   0
```

```
         0   0  0   1];
T_2_c3 = [c  -s  0  Lc3*c
          s   c  0  Lc3*s
          0   0  1    0
          0   0  0    1];
T_0_2 = T_0_1*T_1_2;        T_0_2 = simplify(T_0_2);
T_0_c2 = T_0_1*T_1_c2;      T_0_c2 = simplify(T_0_c2);
T_0_c3 = T_0_2*T_2_c3;      T_0_c3 = simplify(T_0_c3);
T_0_3 = T_0_2*T_2_3;

% calculate the linear and angular velocity Jacobian of each link (COM)
z0 = [0 0 1]';  z1 = T_0_1(1:3,3);  z2 = T_0_2(1:3,3);
o0 = [0 0 0]';  o1 = T_0_1(1:3,4);  o2 = T_0_2(1:3,4);
o3 = T_0_3(1:3,4);
oc1 = T_0_c1(1:3,4);  oc2 = T_0_c2(1:3,4);  oc3 = T_0_c3(1:3,4);

%find the Jacobian for end effector first. This is needed to find
%x' = J * q' for solving part (a)

Jv3 = [cross(z0,o3)  cross(z1,(o3 - o1))  cross(z2,(o3 - o2))];
Jw3 = [z0 z1 z2];
Jacobian = [Jv3;Jw3];

%now we need to find time derivative of the above Jacobian

tmp = subs(Jacobian,{q1,q2,q3},{'q1(t)','q2(t)','q3(t)'});
syms t;
der_Jacobian = diff(tmp,t);
der_Jacobian = subs(der_Jacobian,{'diff(q1(t),t)','diff(q2(t),t)','diff(q3(t),t)'},...
    {'qd(1)','qd(2)','qd(3)'});
der_Jacobian = subs(der_Jacobian,{'q1(t)','q2(t)','q3(t)'},...
    {q1,q2,q3});

der_Jacobian = subs(der_Jacobian,{'sin(q1)','sin(q2)',...
    'sin(q3)','cos(q1)','cos(q2)','cos(q3)'},...
    {s1,s2,s3,c1,c2,c3});


%find the end effector position vector using forward kinematics

Xend = T_0_3 * [0;0;0;1];
Xend = subs(Xend,{'cos(q1)','cos(q2)','cos(q3)','sin(q1)',...
    'sin(q2)','sin(q3)'},{c1,c2,c3,s1,s2,s3});

%   => Jvc1
Jvc1 = [cross(z0,(oc1 - o0))  [0; 0; 0]  [0; 0; 0]];
Jvc2 = [cross(z0,oc2)  cross(z1,(oc2 - o1))  [0; 0; 0]];
Jvc3 = [cross(z0,oc3)  cross(z1,(oc3 - o1))  cross(z2,(oc3 - o2))];
Jw1 = [z0 [0; 0; 0] [0; 0; 0]];
Jw2 = [z0 z1 [0; 0; 0]];
Jw3 = [z0 z1 z2];

% extract rotation matrices
R10 = T_0_c1(1:3,1:3);
```

```
R20 = T_0_c2(1:3,1:3);
R30 = T_0_c3(1:3,1:3);


% calculate mass matrix
disp('Evaluating mass matrix');

Dv1 =  m1*transpose(Jvc1)*Jvc1;
Dv1 = simplify(Dv1);

Dv2 =  m2*transpose(Jvc2)*Jvc2;
Dv2 = simplify(Dv2);

Dv3 =  m3*transpose(Jvc3)*Jvc3;
Dv3 = simplify(Dv3);

Dw1 = transpose(Jw1)*R10*Ic1*transpose(R10)*Jw1;
Dw1 = simplify(Dw1);

Dw2 = transpose(Jw2)*R20*Ic2*transpose(R20)*Jw2;
Dw2 = simplify(Dw2);

Dw3 = transpose(Jw3)*R30*Ic3*transpose(R30)*Jw3;
Dw3 = simplify(Dw3);

D = Dv1 + Dv2 + Dv3 + Dw1 + Dw2 + Dw3;
D = simplify(D);

% after examining solution - try to get further simplification
D = subs(D,{cos(q2)^2 - 1},{-sin(q2)^2});
D = subs(D,{cos(q2 + q3)^2 - 1},{-sin(q2 + q3)^2});

% calculate B and C maatrices
disp('Evaluating Coriolis and centrifual terms')

% form partial derivatives
for i = 1:3
    for j = 1:3
        for k = 1:3
            if k == 1
                d(i,j,k) = diff(D(i,j),q1);
            elseif k == 2
                d(i,j,k) = diff(D(i,j),q2);
            elseif k == 3
                d(i,j,k) = diff(D(i,j),q3);
            end
        end
    end
end
d = simplify(d);

% form Christofel symbols
for i = 1:3
    for j = 1:3
```

```
        for k = 1:3
            b(i,j,k) = 0.5*(d(i,j,k) + d(i,k,j) - d(j,k,i));
        end
    end
end
b = simplify(b);

% assemble C and B matrices
B = [2*b(1,1,2)  2*b(1,1,3)  2*b(1,2,3)
     2*b(2,1,2)  2*b(2,1,3)  2*b(2,2,3)
     2*b(3,1,2)  2*b(3,1,3)  2*b(3,2,3)];
C = [b(1,1,1)  b(1,2,2)  b(1,3,3)
     b(2,1,1)  b(2,2,2)  b(2,3,3)
     b(3,1,1)  b(3,2,2)  b(3,3,3)];

% form G vector
disp('Evaluating gravity vector')
G1 = -(transpose(Jvc1)*m1*gVector);
G2 = -(transpose(Jvc2)*m3*gVector);
G3 = -(transpose(Jvc3)*m3*gVector);
G = G1 + G2 + G3;
G = simplify(G);

% Auto-generate Matlab functions to evaluate D, B, D, and G matrices
if MAKE_FUNCTION == 1
    disp('Auto-generating Matlab functions');
    disp('    => generating D matrix function');
    matlabFunction(D,'file','EvaluateDmatrix');
    disp('    => generating B matrix function');
    matlabFunction(B,'file','EvaluateBmatrix');
    disp('    => generating C matrix function');
    matlabFunction(C,'file','EvaluateCmatrix');
    disp('    => generating G vector function');
    matlabFunction(G,'file','EvaluateGvector');
    disp('    => generating derivative of analytical Jacobian function');
    matlabFunction(der_Jacobian,'file','EvaluateJd');


end
```

Bmatrix_ThreeDOFcontrols.m

```
function B = Bmatrix_ThreeDOFcontrols(q,modelParameters)

    % assign model parameters to local variables
    g = modelParameters.g;
    m = modelParameters.m;
    Ia = modelParameters.Ia;
    L = modelParameters.L;
    Ib = modelParameters.Ib;

    q1 = q(1);  q2 = q(2);  q3 = q(3);
```

```
    B(1,1) = (Ib-Ia-(1/4)*m*L^2)*sin(2*q2+2*q3)+...
             (Ib-Ia- (5/4)*m*L^2)*sin(2*q2)-m*L^2*sin(2*q2+q3);
    B(1,2) = -(1/2)*(sin(q2+q3)*((4*Ia-4*Ib+m*L^2)*cos(q2+q3)+...
             2*m*L^2*cos(q2)));

    B(1,3) = 0;
    B(2,1) = 0;
    B(2,2) = 0;
    B(2,3) = -m*L^2*sin(q3);
    B(3,1) = 0;
    B(3,2) = 0;
    B(3,3) = 0;

end
```

Cmatrix_ThreeDOFcontrols.m

```
function C = Cmatrix_ThreeDOFcontrols(q,modelParameters)

    % assign model parameters to local variables
    g = modelParameters.g;
    m = modelParameters.m;
    Ia = modelParameters.Ia;
    L = modelParameters.L;
    Lc = modelParameters.Lc;
    Ib = modelParameters.Ib;

    q1 = q(1);  q2 = q(2);  q3 = q(3);
    C(1,1) = 0;
    C(1,2) = 0;
    C(1,3) = 0;
    C(2,1) = (1/2)*(Ia-Ib+(1/4)*m*L^2)*sin(2*q2+2*q3)+...
             (1/2)*(Ia-Ib+(5/4)*m*L^2)*sin(2*q2)+(1/2)*m*L^2*sin(2*q2+q3);
    C(2,2) = 0;
    C(2,3) = -(1/2)*m*L^2*sin(q3);
    C(3,1) = (1/4)*(sin(q2+q3)*((4*Ia-4*Ib+(1/4)*m*L^2)*...
              cos(q2+q3)+2*m*L^2*cos(q2)));
    C(3,2) = (1/2)*m*L^2*sin(q3);
    C(3,3) = 0; %verify with my derivation, I got something not zero

end
```

Dmatrix_ThreeDOFcontrols.m

```
function [D,Davg] = Dmatrix_ThreeDOFcontrols(q,modelParameters)

    % assign model parameters to local variables
    % model parameters
    m = modelParameters.m;
    L = modelParameters.L;
    Ia = modelParameters.Ia;
    Ib = modelParameters.Ib;

    s2   = sin(q(2));
    c2   = cos(q(2));
    c3   = cos(q(3));
    s23  = sin(q(2) + q(3));
    c23  = cos(q(2) + q(3));

    D    = zeros(3,3);
    Davg = D;

    %see ThreeDOFcontrols_symbolic.m for derivation of these
    D(1,1)= 1/4*m*L^2*c2^2+1/4*m*L^2*(c23+2*c2)^2+...
        Ib+Ia*(c2^2+c23^2)+Ib*(s2^2+s23^2);
    D(1,2) = 0;
    D(1,3) = 0;

    D(2,1) = 0;
    D(2,2) = 2*Ia + (3/2)*m*L^2 + L^2*m*c3;
    D(2,3) = 1/4*m*L^2 + 1/2*m*L^2*c3 + Ia;

    D(3,1) = 0;
    D(3,2) = D(2,3);
    D(3,3) = 1/4*m*L^2 + Ia;

    % "average" mass matrix calculation,
    % trigometric identities - assume q equals zero and set off diagonal
    % to zero
    c2 = 1.0; c3 = 1.0; c23=1; s2=0;s23=0;

    Davg(1,1)= 1/4*m*L^2*c2^2+1/4*m*L^2*(c23+2*c2)^2+...
        Ib+Ia*(c2^2+c23^2)+Ib*(s2^2+s23^2);
    Davg(1,2) = 0;
    Davg(1,3) = 0;

    Davg(2,1) = 0;
    Davg(2,2) = 2*Ia + (3/2)*m*L^2 + L^2*m*c3;
    Davg(2,3) = 0;  %  1/4*m*L^2 + 1/2*m*L^2*c3 + Ia;  Notice, set to zero

    Davg(3,1) = 0;
    Davg(3,2) = 0; %notice, set to zero
    Davg(3,3) = 1/4*m*L^2 + Ia;

end
```

Gvector_ThreeDOFcontrols.m

```
function G = Gvector_ThreeDOFcontrols(q,modelParameters)

    % assign model parameters to local variables
    g = modelParameters.g;
    m = modelParameters.m;
    L = modelParameters.L;

    q1 = q(1);  q2 = q(2);  q3 = q(3);


    G(1,1) = 0;
    G(2,1) = (1/2)*m*g*L*(3*cos(q2)+cos(q2+q3));
    G(3,1) = (1/2)*m*g*L*cos(q2+q3);


end
```

0.3 Operational space control

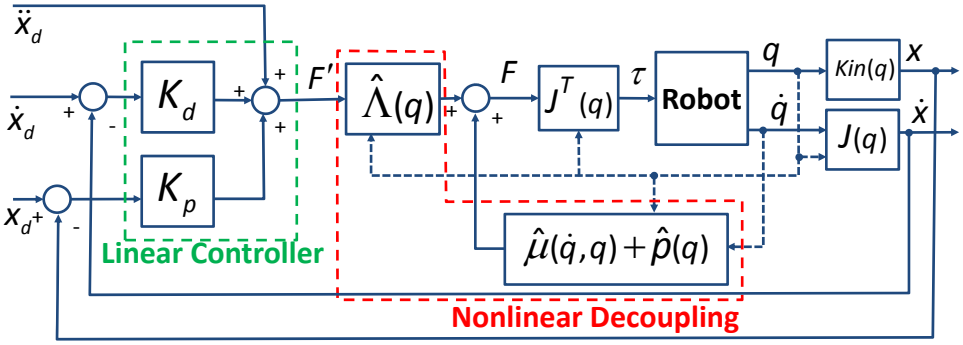
The difficult part of this implementation was finding the analytical Jacobian and its derivative with respect to time. Matlab symbolic was used for this and the file `Three_DOF_symbolic.m` contains the implementation of this.

The operational space controller using proportional derivative was implemented in the files `ThreeDOFcontrols_OP.m` and `zDot3dofControls_OP.m`. The following shows the plots generated for parts (a,b,c) followed by discussion comparing the results.

The following diagram shows the operational space controller layout taken from the class handout, page 6-163 which shows the controller with full compensation.

Operational Space Dynamic Decoupling

Operational (or Task) Space Inverse Dynamics Control:



- decoupling forces based on measured manipulator states
- Requires real-time computation of nonlinear decoupling terms
- Requires real-time computation of Λ and μ, both of which require inverse matrix computations
- Nonlinear terms affected by sensor noise and delay

6-163

Figure 27: Diagram of controller, task space inverse dynamics control, full compensation. From class lecture notes

The following diagrams gives a high level overview of the Matlab software design for the implementation of the task space control. The Jacobian use in this controller is much more critical than with joint space controller due to the mapping between joint space and task space needed in the implementation.

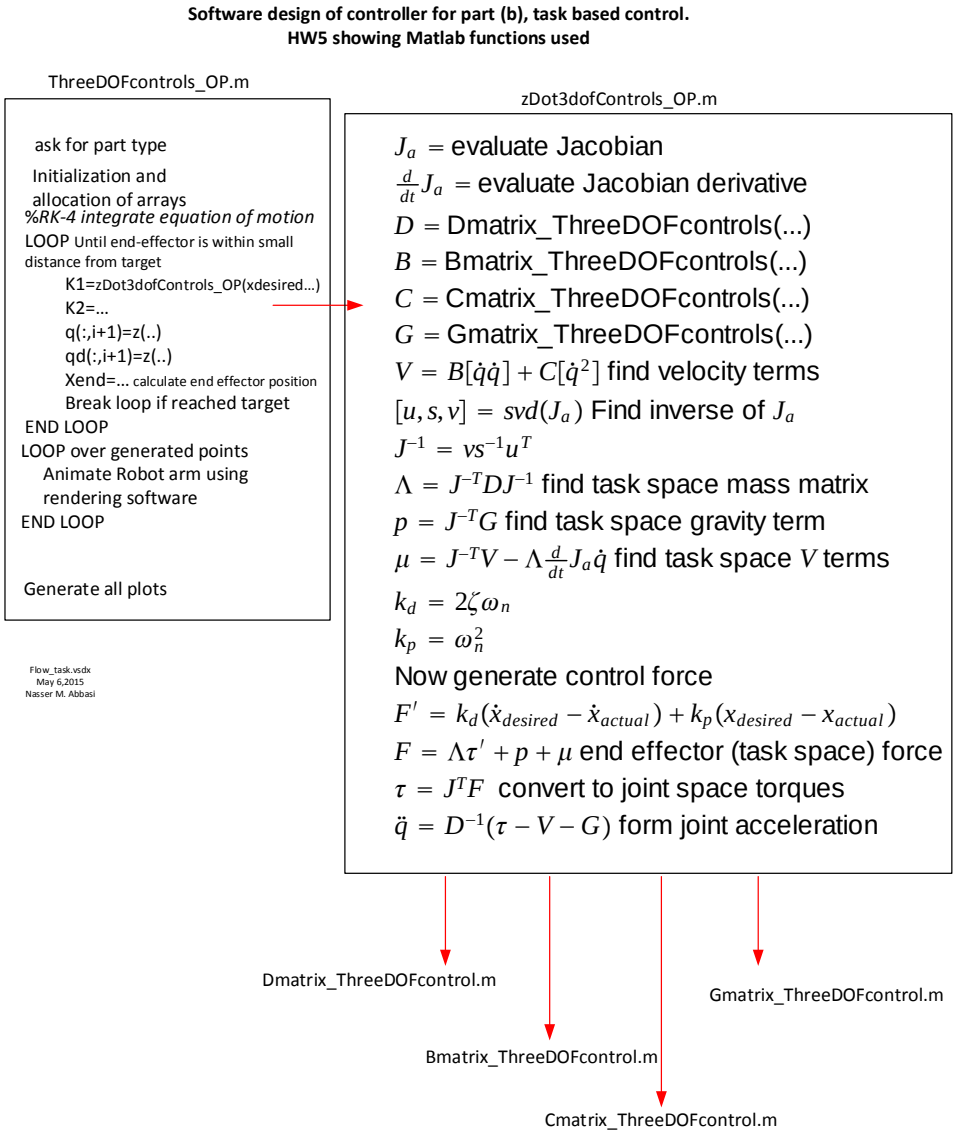


Figure 28: Matlab program design for task space controller

0.3.1 Part (a) $x_f = [-L, -L, 0]^T$

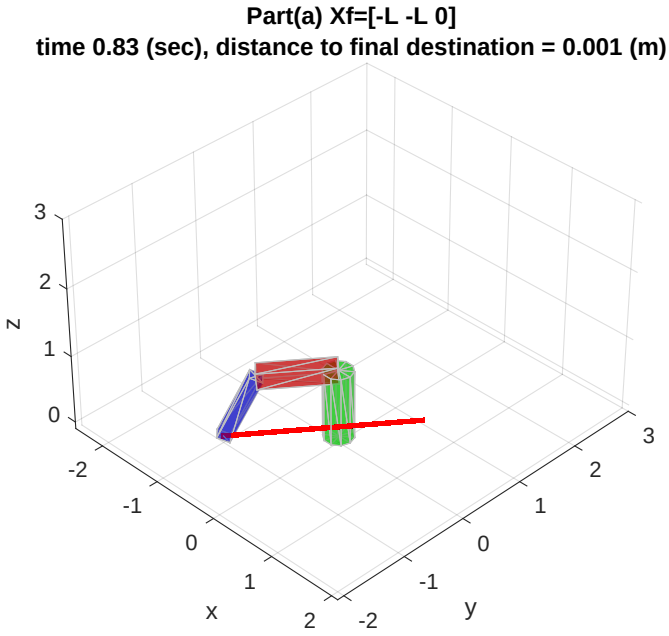


Figure 29: Final position of robot arm, Operational space control, part(a)

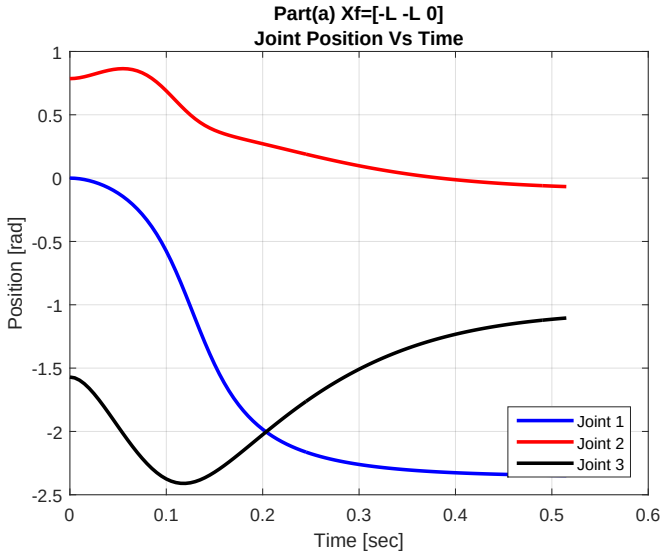


Figure 30: Joint position vs. time, Operational space control, part(a)

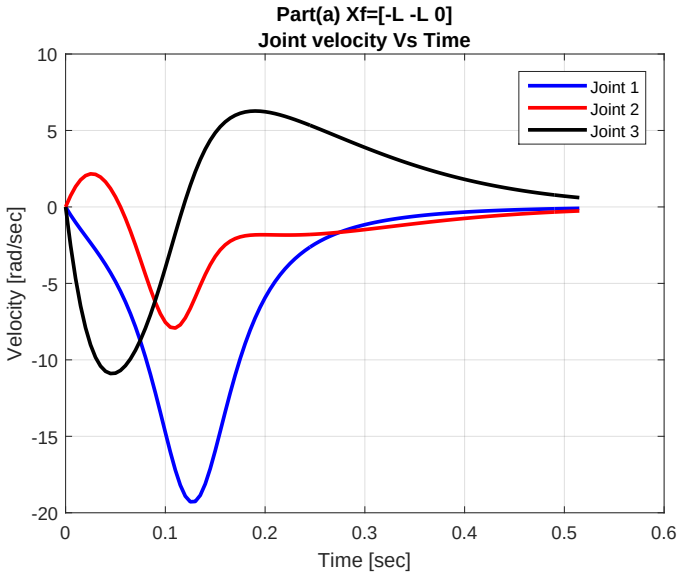


Figure 31: Joint velocity vs. time, Operational space control, part(a)

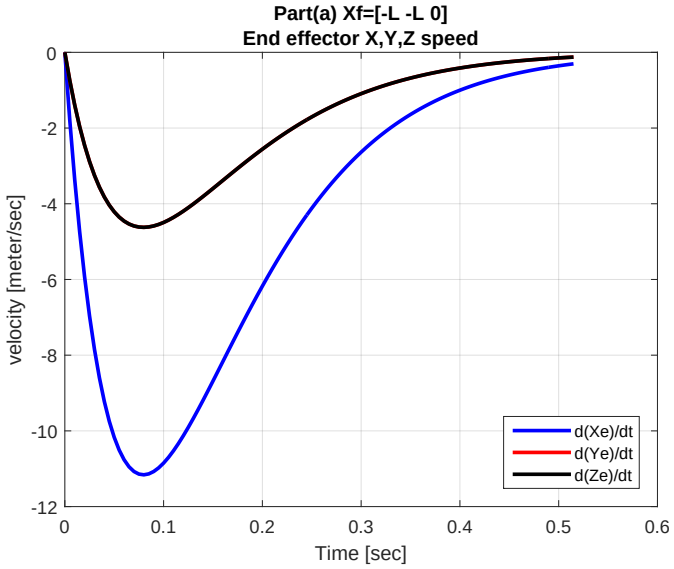


Figure 32: End effector $\frac{dX_e}{dt}$, $\frac{dY_e}{dt}$, $\frac{dZ_e}{dt}$ Operational space control, part(a)

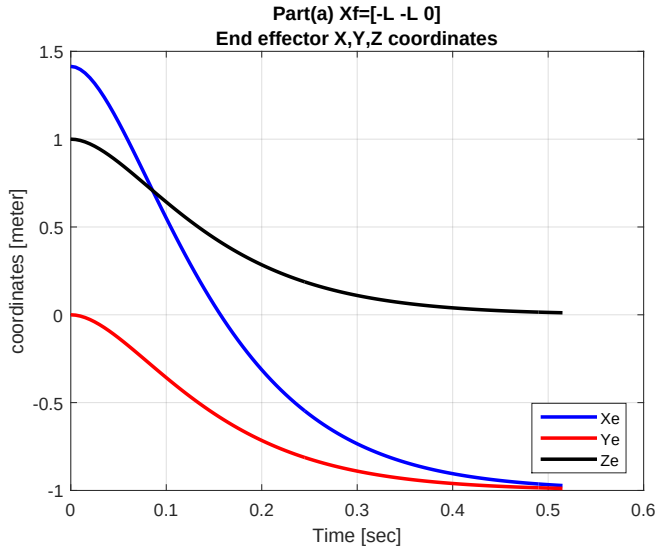


Figure 33: End effector X_e, Y_e, Z_e vs. time, Operational space control, part(a)

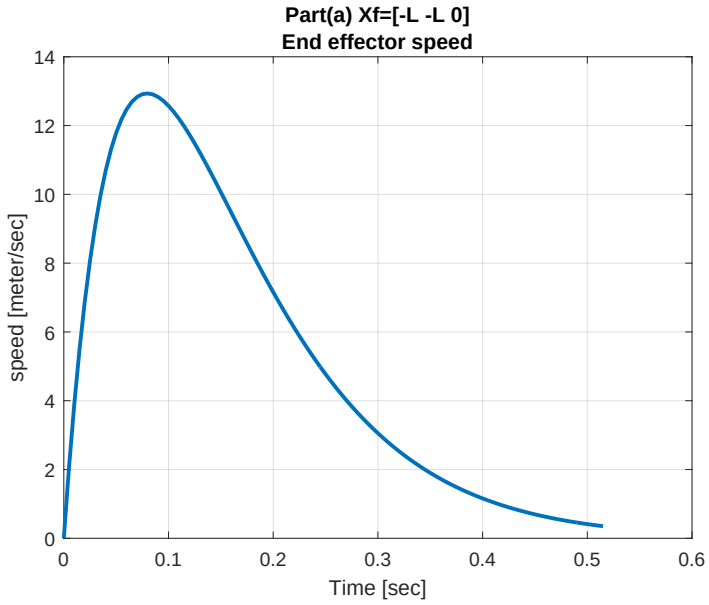


Figure 34: End effector linear speed vs. time, Operational space control, part(a)

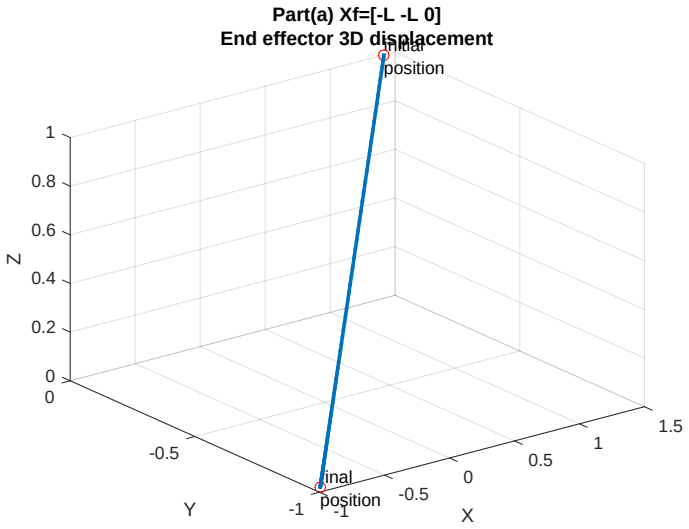


Figure 35: End effector plot3 displacement, Operational space control, part(a)

0.3.2 Part (b) $x_f = [-L, -\frac{L}{10}, 0]^T$

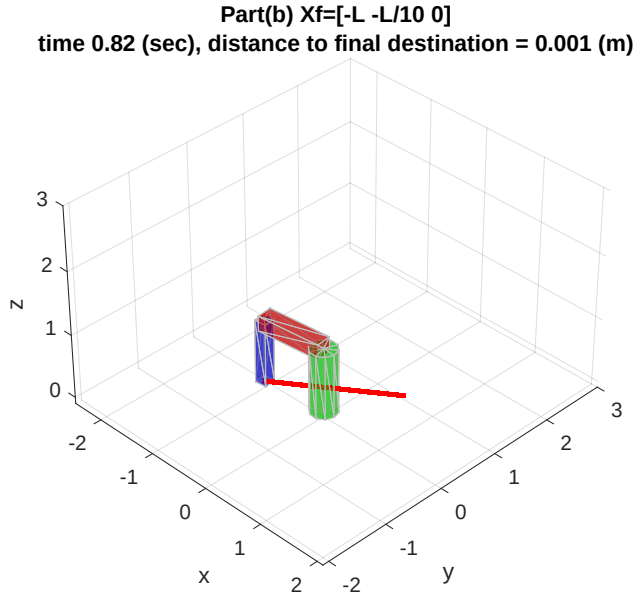


Figure 36: Final position of robot arm, Operational space control, part(b)

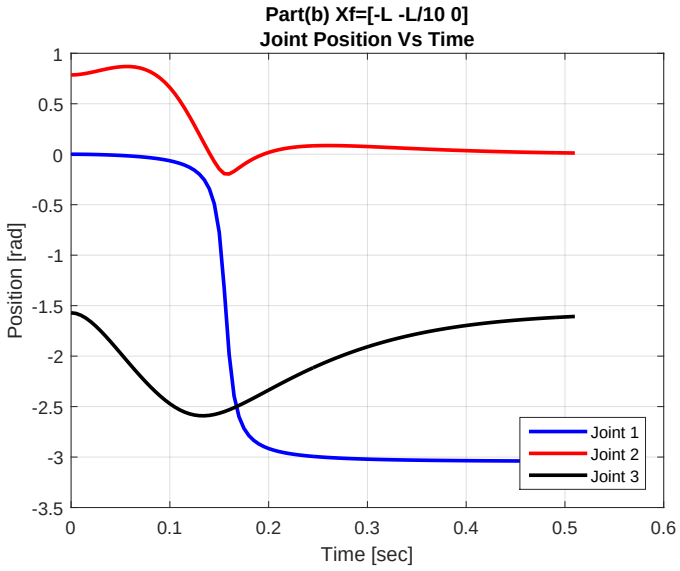


Figure 37: Joint position vs. time, Operational space control, part(b)

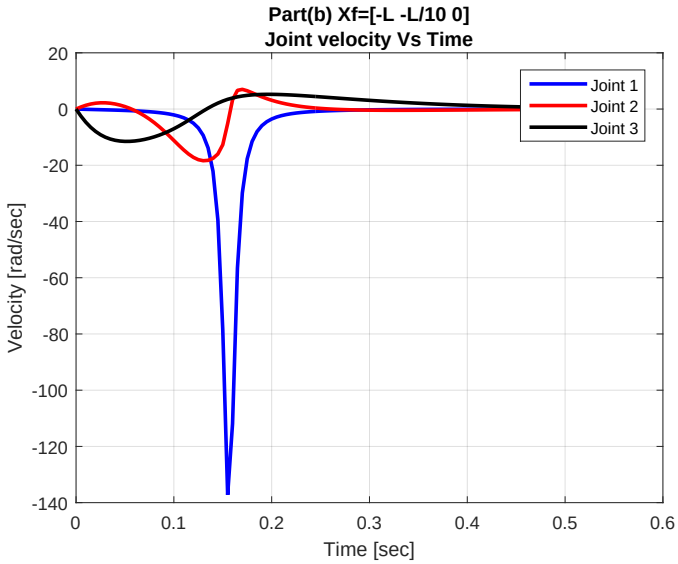


Figure 38: Joint velocity vs. time, Operational space control, part(b)

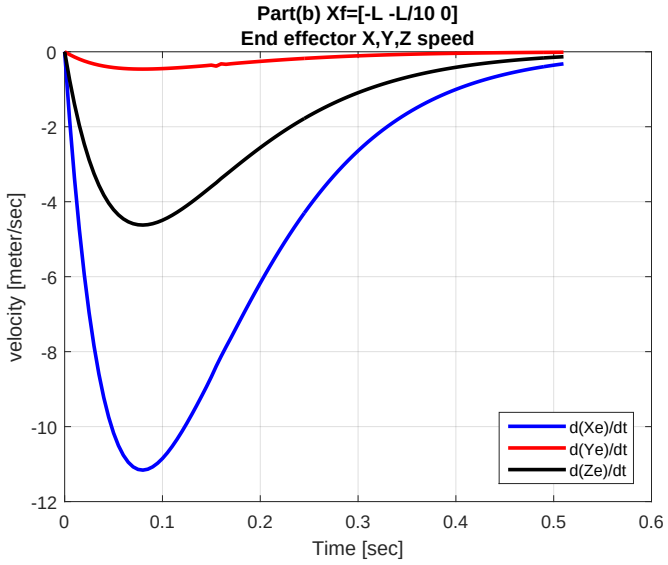


Figure 39: End effector $\frac{dX_e}{dt}, \frac{dY_e}{dt}, \frac{dZ_e}{dt}$ Operational space control, part(b)

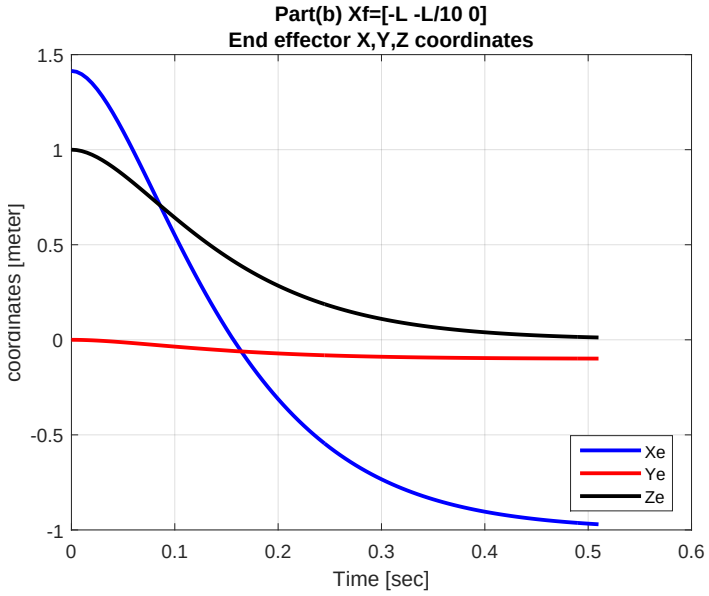


Figure 40: End effector X_e, Y_e, Z_e vs. time, Operational space control, part(b)

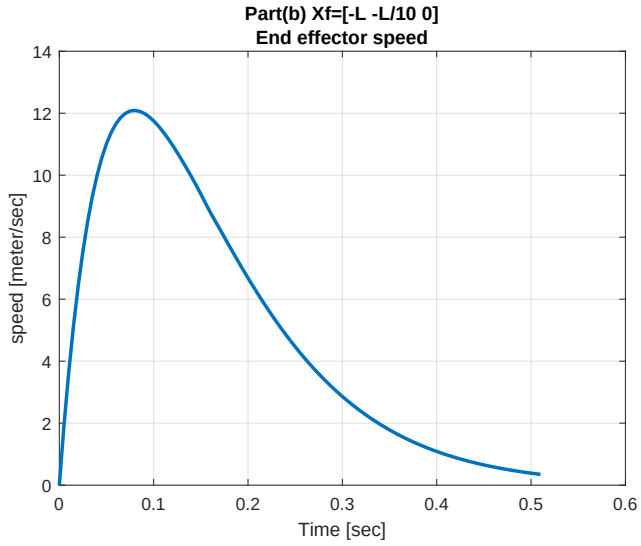


Figure 41: End effector linear speed vs. time, Operational space control, part(b)

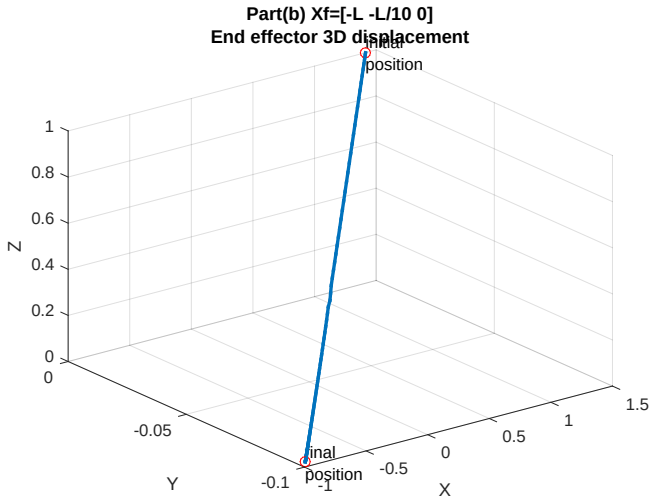


Figure 42: End effector plot3 displacement, Operational space control, part(b)

0.3.3 Part (c) $x_f = [-L, -L, 0]^T$ with velocity limiting heuristic

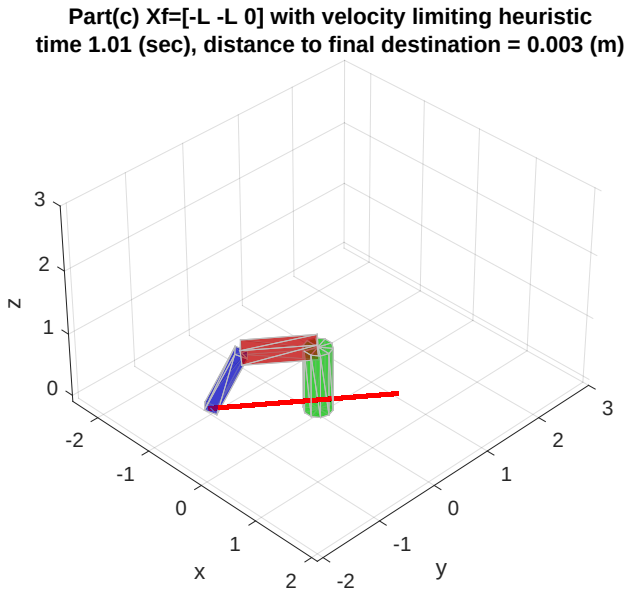


Figure 43: Final position of robot arm, Operational space control, part(c)

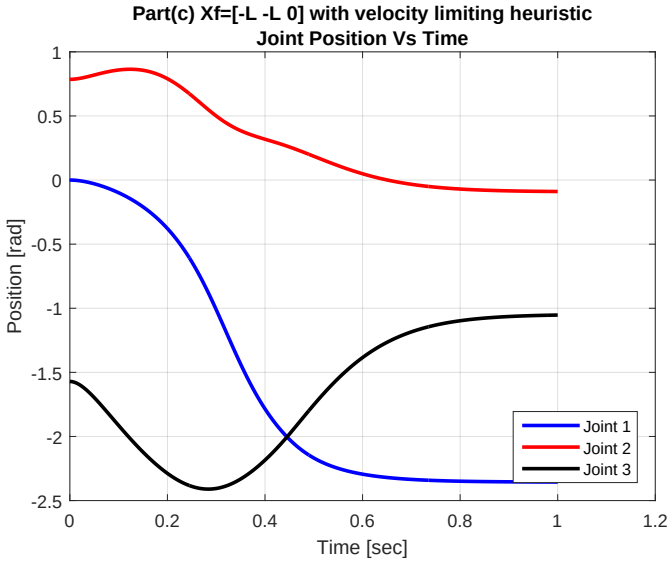


Figure 44: Joint position vs. time, Operational space control, part(c)

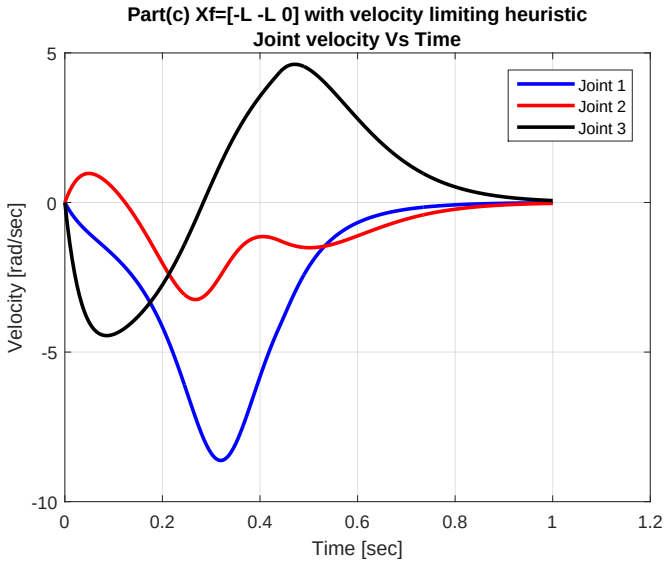


Figure 45: Joint velocity vs. time, Operational space control, part(c)

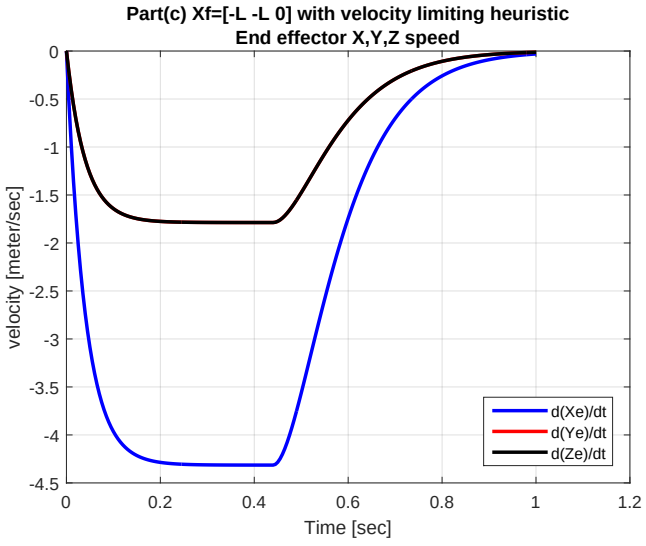


Figure 46: End effector $\frac{dX_e}{dt}$, $\frac{dY_e}{dt}$, $\frac{dZ_e}{dt}$ Operational space control, part(c)

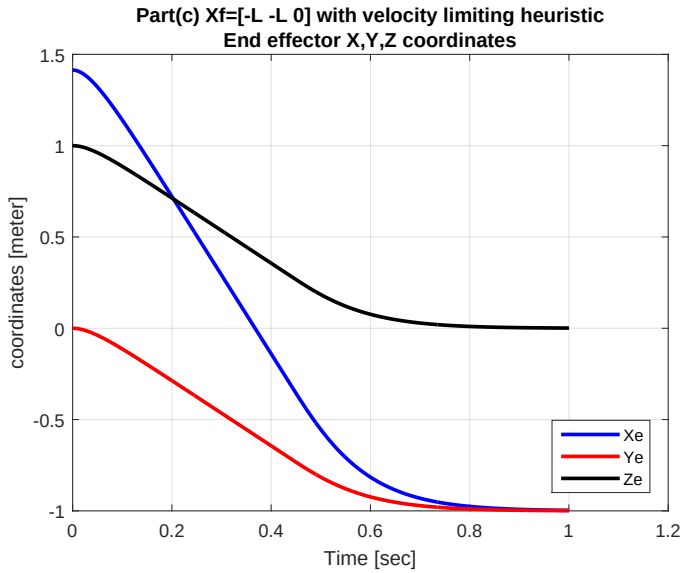


Figure 47: End effector X_e, Y_e, Z_e vs. time, Operational space control, part(c)

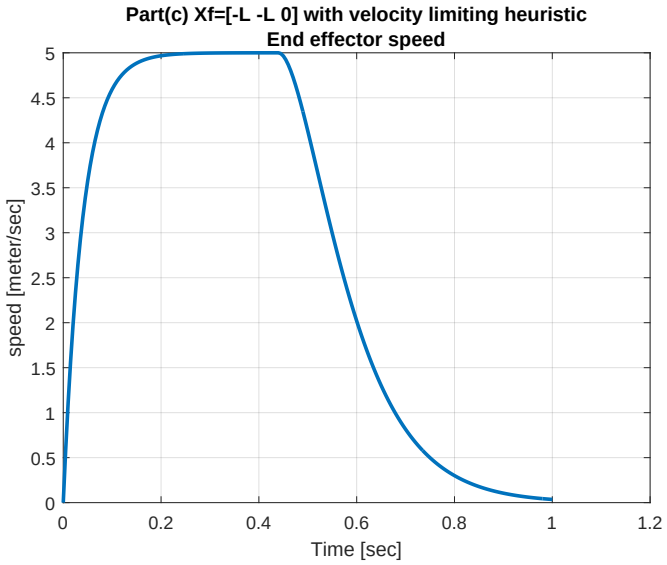


Figure 48: End effector linear speed vs. time, Operational space control, part(c)

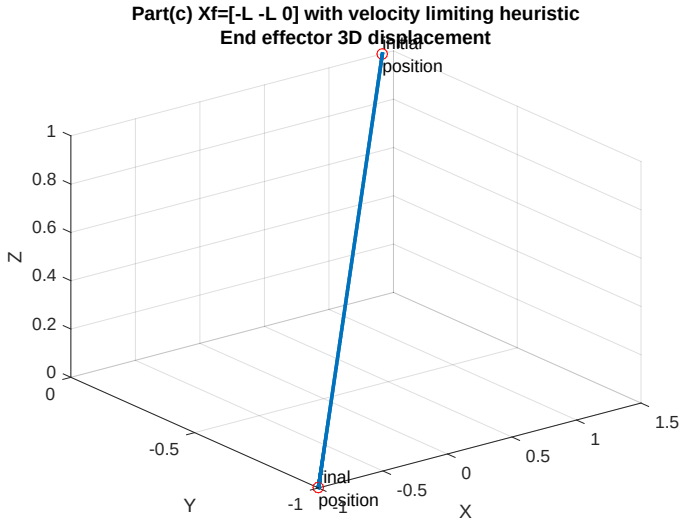


Figure 49: End effector plot3 displacement, Operational space control, part(c)

0.3.4 Part (d) discussion of result, compare control methods

In this part of the problem, the same controller was used for all parts, which is a P.D. controller with compensation for velocity terms (Coriolis and centrifugal terms) and the gravity terms.

Operational space based control is more intuitive since the target coordinates are given in operational space rather than in joint space and therefore it is easier to describe in terms of where the end-effector target should be.

One issue with Operational space is that the Jacobian can become ill-conditioned near singularities. Since the Jacobain and its derivatives are used in this control to map between joint space and operational space, this can cause a problem. This means the operational space control has to either avoid getting close to singularity region or be modified near singularities. In this simulation, no singularity was encountered.

In the three cases, the path traveled by the end effector showed a straight line from the initial position to the final position as desired. This was different from the joint space control, where the path traveled by the end effector was curved and had number of twists and turns.

The time used to reach the target for part (c) (with linear velocity limiting) was the longest, a little over one second. This is about 30% longer than the time taken by part(a) which did not have the velocity limiting heuristic and also had the same target coordinates in task space.

This is as expected, since the maximum speed the end effector can reach in part(c) was kept below 5 meter/sec. The end-effector speed profile for part(a) shows it reached maximum speed of 12 meter/sec.

The advantage of speed limiting heuristic is that it eliminated large acceleration and deceleration of the end effector which can be important in some applications where joints speed have to be kept below some value.

In all cases, there was no overshoot in the operational space motion. But looking at joint space, case (b) showed sudden change in the joint one speed around 0.15 second.

Since control is based on operational space and not joint space, the joints positions and speeds that result can become much larger and exhibit large oscillation compared with joint based control. This is seen in part(b) where joint 1 had sudden change in speed. This can be a problem depending on the application where the joints actuators can not provide the required joint speed and will saturate.

The following plot shows side by side the joints space displacements that resulted for each part.

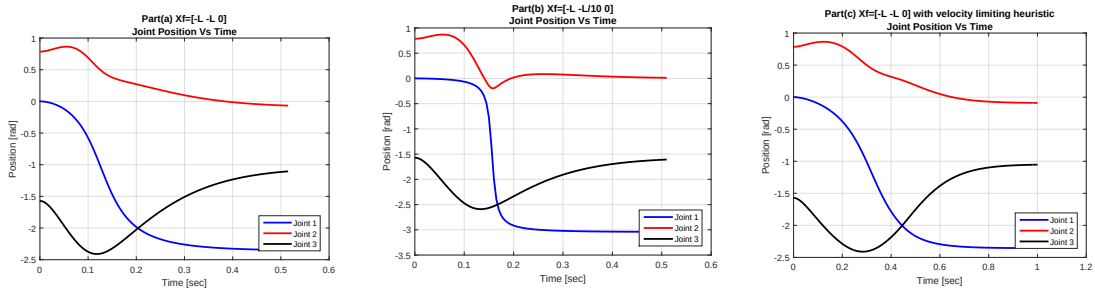


Figure 50: Joints positions vs. time. Operational space. part(a),(b),(c) side by side

In the above, joint 1 had sudden change in motion for part(b), this is due to the end target location. In the velocity profile below, one can see the corresponding sudden change in speed for this joint as well.

The following diagram shows the joint velocity vs. time for each case.

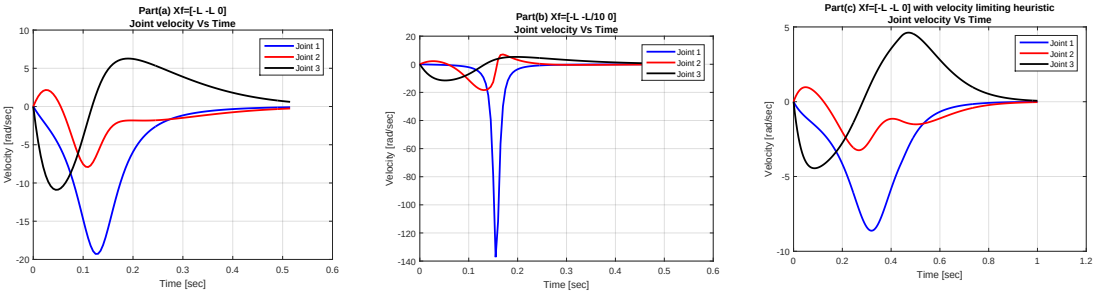


Figure 51: Joints velocity vs. time, operational space, part(a),(b),(c) side by side

The following diagram shows the end effector speed for the three cases side by side.

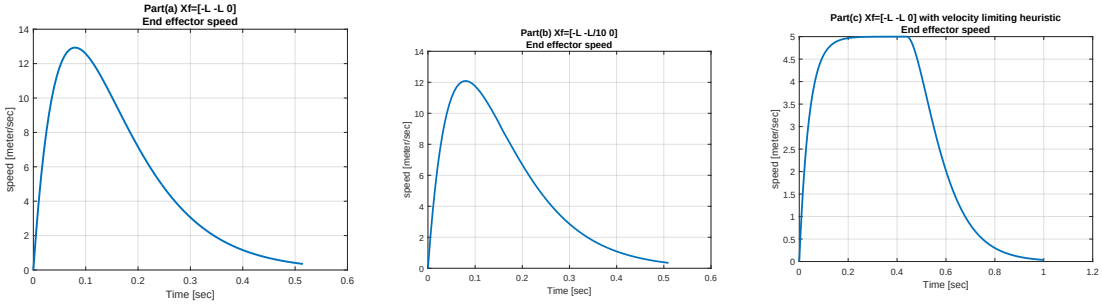


Figure 52: end effector speed. Operational space. part(a),(b),(c) side by side

0.3.5 Source code listing for operational space control

The following gives the Matlab source code listing that implements the operational space control part of the HW. Three new files were needed in addition of the files listed in part (a) above.

To run the script for this part, the command is

```
ThreeDOFcontrols_OP
```

ThreeDOFcontrols_OP.m

```
%file ThreeDOFcontrols_OP.m
%This the main script used to implement HW5, part b.
%Modified original code from ME739 UW learn@UW, Madison by Professor Zinn
%
%Nasser M. Abbasi  5/10/2015

%--------------------------------------------------------------------------
%  NUMERICAL INTEGRATION OF DYNAMIC EQUATIONS
%--------------------------------------------------------------------------

%clear all;
close all;
clc;

% set the model parameters per the HW problem.
modelParameters = InitializeThreeDOFmodel_OP();
L = modelParameters.L;

% Ask use for which part to run. There are 3 sections for this problem
disp('Specify part of problem to solve:')
disp('   1 = option(a)  Xf =[-L -L 0]')
disp('   2 = option(b)  Xf =[-L  -L/10 0]')
disp('   3 = option(c), Xf =[-L -L 0] with velocity limiting heuristic')
modelParameters.controlMethod = input('> ');

%depending on the section, set labels as needed
xdDes   = [0; 0; 0];   %desired final task space velocity
tend    = 1;     %max simulation run time, Actual time is determined below
```

```
dT      = .005;  %integration step size

switch modelParameters.controlMethod
  case 1
   title_add_on = 'Part(a) Xf=[-L -L 0]';
   xDes    = [-L;-L;0; xdDes];     %desired final task space states
  case 2
   title_add_on = 'Part(b) Xf=[-L -L/10 0]';
   xDes    = [-L;-L/10;0; xdDes];     %desired final task space states
  case 3
   title_add_on = 'Part(c) Xf=[-L -L 0] with velocity limiting heuristic';
   xDes    = [-L;-L;0; xdDes];     %desired final task space states
end;

%-----------------------------------------------------------------------
%  RENDERING INITIALIZATION
%-----------------------------------------------------------------------
L           = modelParameters.L;
L1          = L;
L2          = L;
L3          = L;
f_handle    = 1;
axis_limits = L*[-2.5 2.1 -2.1 3 -.1 3];
render_view = [1 -1 1]; view_up = [0 0 1];
SetRenderingViewParameters(axis_limits,render_view,view_up,f_handle);
camproj perspective

%----initialize rendering

% link 1 rendering initialization
r1         = L1/5;
sides1     = 10;
axis1      = 2;
norm_L1    = 1.0;
linkColor1 = [0 0.75 0];
plotFrame1 = 0;
d1         =  CreateLinkRendering(L1,r1,sides1,axis1,norm_L1,linkColor1,...
    plotFrame1,f_handle);

% link 2 rendering initialization
r2         = L2/6;
sides2     = 4;
axis2      = 1;
norm_L2    = 1.0;
linkColor2 = [0.75 0 0];
plotFrame2 = 0;
d2         =  CreateLinkRendering(L2,r2,sides2,axis2,norm_L2,linkColor2,...
    plotFrame2,f_handle);

% link 3 rendering initialization
r3         = L2/8;
sides3     = 4;
axis3      = 1;
norm_L3    = 1.0;
```

```
linkColor3 = [0 0 0.75];
plotFrame3 = 0;
d3         =  CreateLinkRendering(L3,r3,sides3,axis3,norm_L3,linkColor3,...
    plotFrame3,f_handle);

numPts  = floor(tend/dT);

q       = zeros(3,numPts);
dq      = zeros(3,numPts);
t       = zeros(1,numPts);
q(:,1)  = [0; pi/4; -pi/2];   %initial position
qd(:,1) = [0; 0; 0];          %initial velocity
z       = [q(:,1); qd(:,1)];  %initialize the state variables

% integrate equations of motion,     % Runge-Kutta 4th order
keep_running = true;
k0=1;  %counter. Loop updates this and terminates when arm reaches target

while keep_running
    k1 = zDot3dofControls_OP(z,xDes,modelParameters);
    k2 = zDot3dofControls_OP(z + 0.5*k1*dT,xDes,modelParameters);
    k3 = zDot3dofControls_OP(z + 0.5*k2*dT,xDes,modelParameters);
    k4 = zDot3dofControls_OP(z + k3*dT,xDes,modelParameters);
    z = z + (1/6)*(k1 + 2*k2 + 2*k3 + k4)*dT;

    % store joint position and velocity for post processing
    q(:,k0+1)  = z(1:3);
    qd(:,k0+1) = z(4:6);
    t(1,k0+1)  = t(1,k0) + dT;

    % check if goal position has been reached
    q1 = z(1); q2 = z(2);  q3=z(3);
    s1=sin(q1); c23=cos(q2+q3); s23= sin(q2+q3);
    c2=cos(q2); c1=cos(q1);
    s2=sin(q2); c3=cos(q3); s3=sin(q3);

    %This is end effector position vector found from forward kinematics
    %using the Three_DOF_symbolic.m symbolic computation script

    X  = [ L*c1*c2 + L*c1*c2*c3 - L*c1*s2*s3;
           L*c2*s1 - L*s1*s2*s3 + L*c2*c3*s1;
           L*(s2 + 1) + L*c2*s3 + L*c3*s2];
    xError = norm(X - xDes(1:3));
    if (xError < 1e-3 || k0 > numPts)
       keep_running = false;
    else
       k0=k0+1;
    end
end
%--------------------------------------------------------------------
%  DISPLAY INTERATION RESULTS
%--------------------------------------------------------------------

Xend    = zeros(3,k0);    %end effector coordinates
```

```
Xvend   = zeros(3,k0);    %end effector linear velocity

for i = 1:k0
    q1 = q(1,i);
    q2 = q(2,i);
    q3 = q(3,i);

    % Update frame {1}
    c   = cos(q1);
    s   = sin(q1);
    L   = L1;
    T10 = [c  0   s  0
        s  0  -c  0
        0  1   0  L
        0  0   0  1];

    % Update frame {2}
    c   = cos(q2);
    s   = sin(q2);
    L   = L2;
    T21 = [c  -s  0  L*c
        s   c  0  L*s
        0   0  1   0
        0   0  0   1];

    % Update frame {3}
    c   = cos(q3);
    s   = sin(q3);
    L   = L3;
    T32 = [c  -s  0  L*c
        s   c  0  L*s
        0   0  1   0
        0   0  0   1];
    T20 = T10*T21;
    T30 = T20*T32;

    % end-effector position
    Xend(:,i) = T30(1:3,4);

    %to find end effector velocity, we use X' = J* q' where the Jacobian
    %for the end effector is found in the Three_DOF_symbolic.m script
    %as part of this HW, in the same folder as this file.

    J=zeros(3,3);
    J(1,1)=L*sin(q1)*sin(q2)*sin(q3) - L*cos(q2)*cos(q3)*sin(q1) - L*cos(q2)*sin(q1);
    J(1,2)=-cos(q1)*(L*(sin(q2) + 1) - L + L*cos(q2)*sin(q3) + L*cos(q3)*sin(q2));
    J(1,3)=-cos(q1)*(L*cos(q2)*sin(q3) + L*cos(q3)*sin(q2));
    J(2,1)=L*cos(q1)*cos(q2) + L*cos(q1)*cos(q2)*cos(q3) - L*cos(q1)*sin(q2)*sin(q3);
    J(2,2)=-sin(q1)*(L*(sin(q2) + 1) - L + L*cos(q2)*sin(q3) + L*cos(q3)*sin(q2));
    J(2,3)=-sin(q1)*(L*cos(q2)*sin(q3) + L*cos(q3)*sin(q2));
    J(3,1)=0;
    J(3,2)=cos(q1)*(L*cos(q1)*cos(q2) + L*cos(q1)*cos(q2)*cos(q3) - ...
        L*cos(q1)*sin(q2)*sin(q3)) + sin(q1)*(L*cos(q2)*sin(q1) + ...
        L*cos(q2)*cos(q3)*sin(q1) - L*sin(q1)*sin(q2)*sin(q3));
```

```
    J(3,3)=cos(q1)*(L*cos(q1)*cos(q2)*cos(q3) - L*cos(q1)*sin(q2)*...
        sin(q3)) + sin(q1)*(L*cos(q2)*cos(q3)*sin(q1) - ...
        L*sin(q1)*sin(q2)*sin(q3));

    Xvend(:,i) = J*qd(:,i);

    % update rendering
    figure(f_handle);
    UpdateLink(d1,T10);
    UpdateLink(d2,T20);
    UpdateLink(d3,T30);
    title(sprintf('%s\ntime %3.2f (sec), distance to final destination = %4.3f (m)',...
        title_add_on,i*dT,norm(Xend(:,i)-xDes(1:3))));
    hold on;
    plot3(Xend(1,1:i),Xend(2,1:i),Xend(3,1:i),'r','LineWidth',2);

    if i == 1;  %pause at start of simulation rendering
        pause;
    end
    drawnow;
end

%-------------------------------------------------------------------------
%  PLOT JOINT POSITIONS
%-------------------------------------------------------------------------

figure(2);
plot(t(1:k0),q(1,1:k0),'b','LineWidth',2); hold on
plot(t(1:k0),q(2,1:k0),'r','LineWidth',2); hold on
plot(t(1:k0),q(3,1:k0),'k','LineWidth',2); hold off
title(sprintf('%s\n%s',title_add_on,'Joint Position Vs Time'));
xlabel('Time [sec]'); ylabel('Position [rad]');
grid on
legend('Joint 1','Joint 2','Joint 3','Location','southeast');

%-------------------------------------------------------------------------
%  PLOT JOINT Velocities
%-------------------------------------------------------------------------

figure(3);
plot(t(1:k0),qd(1,1:k0),'b','LineWidth',2); hold on
plot(t(1:k0),qd(2,1:k0),'r','LineWidth',2); hold on
plot(t(1:k0),qd(3,1:k0),'k','LineWidth',2); hold off
title(sprintf('%s\n%s',title_add_on,'Joint velocity Vs Time'));
xlabel('Time [sec]'); ylabel('Velocity [rad/sec]');
grid on
legend('Joint 1','Joint 2','Joint 3','Location','northeast');

%-------------------------------------------------------------------------
%  PLOT end effector X,Y,Z velocites
%-------------------------------------------------------------------------

figure(4);
plot(t(1:k0),Xvend(1,:),'b','LineWidth',2); hold on
```

```
plot(t(1:k0),Xvend(2,:),'r','LineWidth',2); hold on
plot(t(1:k0),Xvend(3,:),'k','LineWidth',2); hold off
title(sprintf('%s\n%s',title_add_on,'End effector X,Y,Z speed'));
xlabel('Time [sec]'); ylabel('velocity [meter/sec]');
grid on
legend('d(Xe)/dt','d(Ye)/dt','d(Ze)/dt','Location','southeast');

%--------------------------------------------------------------------
%  PLOT end effector X,Y,Z positions
%--------------------------------------------------------------------

figure(5);
plot(t(1:k0),Xend(1,:),'b','LineWidth',2); hold on
plot(t(1:k0),Xend(2,:),'r','LineWidth',2); hold on
plot(t(1:k0),Xend(3,:),'k','LineWidth',2); hold off
title(sprintf('%s\n%s',title_add_on,'End effector X,Y,Z coordinates'));
xlabel('Time [sec]'); ylabel('coordinates [meter]');
grid on
legend('Xe','Ye','Ze','Location','southeast');

%--------------------------------------------------------------------
%  PLOT end effector speed (magnitude)
%--------------------------------------------------------------------

figure(6);
plot(t(1:k0), sqrt(sum(Xvend.^2,1)),'LineWidth',2);
title(sprintf('%s\n%s',title_add_on,'End effector speed'));
xlabel('Time [sec]'); ylabel('speed [meter/sec]');
grid on

%--------------------------------------------------------------------
%  PLOT end effector displacement in 3D
%--------------------------------------------------------------------

figure(7);
plot3(Xend(1,:),Xend(2,:),Xend(3,:),'LineWidth',2); hold on;
plot3(Xend(1,1),Xend(2,1),Xend(3,1),'ro');
text(Xend(1,1),Xend(2,1),Xend(3,1),{'initial','position'});
plot3(Xend(1,end),Xend(2,end),Xend(3,end),'ro');
text(Xend(1,end),Xend(2,end),Xend(3,end),{'final','position'});
title(sprintf('%s\n%s',title_add_on,'End effector 3D displacement'));
xlabel('X'); ylabel('Y'); zlabel('Z');
grid on
```

InitializeThreeDOFmodel_OP.m

```
function modelParameters = InitializeThreeDOFmodel_OP

    % set model parameters

    % gravitational constant [m/s^2]
    g = 9.81;

    % link mass [kg]
    m = 10;

    % link length [m]
    L = 1;

    % link COM location [m]
    Lc = L/2;

    % link radius [m]
    r = 0.1*L;

    % link inertia (_|_ to  link's CL) [kg/m^2]
    Ia = (1/12)*m*L^2;
    Ib = m*r^2;

    % assign values of model parameter structure
    modelParameters.g = 9.81; % gravitational constant [m/s^2]
    modelParameters.m = m;   % link mass [kg]
    modelParameters.L = L;    % link length [m]
    modelParameters.Lc = Lc;  % link COM location [m]
    modelParameters.Ia = Ia;   % inertia (_|_ to  link's CL) [kg/m^2]
    modelParameters.Ib = Ib;   % inertia (colinear to link's CL) [kg/m^2]
    modelParameters.controlMethod = 1;
    modelParameters.vMax=5; %meter/sec
end
```

zDot3dofControls_OP.m

```
function [zDot] = zDot3dofControls_OP(z,xDesired,modelParameters)

    % assign joint displacements / velocities from state variables
    q      = z(1:3);
    qd     = z(4:end);
    xDes   = xDesired(1:3);
    xdDes  = xDesired(4:end);
    L = modelParameters.L;
    % precalculate sin and cos terms
    s1=sin(q(1)); c23=cos(q(2)+q(3)); s23= sin(q(2)+q(3));
    c2=cos(q(2)); c1=cos(q(1));
    s2=sin(q(2)); c3=cos(q(3)); s3=sin(q(3));
    q1=q(1);q2=q(2);q3=q(3);
    J=zeros(3,3);
```

```
J(1,1)=L*sin(q1)*sin(q2)*sin(q3) - L*cos(q2)*cos(q3)*sin(q1) - L*cos(q2)*sin(q1);
J(1,2)=-cos(q1)*(L*(sin(q2) + 1) - L + L*cos(q2)*sin(q3) + L*cos(q3)*sin(q2));
J(1,3)=-cos(q1)*(L*cos(q2)*sin(q3) + L*cos(q3)*sin(q2));
J(2,1)=L*cos(q1)*cos(q2) + L*cos(q1)*cos(q2)*cos(q3) - L*cos(q1)*sin(q2)*sin(q3);
J(2,2)=-sin(q1)*(L*(sin(q2) + 1) - L + L*cos(q2)*sin(q3) + L*cos(q3)*sin(q2));
J(2,3)=-sin(q1)*(L*cos(q2)*sin(q3) + L*cos(q3)*sin(q2));
J(3,1)=0;
J(3,2)=cos(q1)*(L*cos(q1)*cos(q2) + L*cos(q1)*cos(q2)*cos(q3) - L*cos(q1)*sin(q2)*sin(q3)) + sin(q
J(3,3)=cos(q1)*(L*cos(q1)*cos(q2)*cos(q3) - L*cos(q1)*sin(q2)*sin(q3)) + sin(q1)*(L*cos(q2)*cos(q3


%the derivative of the Jacobian was derived in the Three_DOF_symbolic.m
%file
Jd = zeros(3,3);
Jd(1,1)=L*s1*s2*qd(2) - L*c1*c2*qd(1) - L*c1*c2*c3*qd(1) + ...
    L*c1*s2*s3*qd(1) + L*c2*s1*s3*qd(2) + L*c3*s1*s2*qd(2) + ...
    L*c2*s1*s3*qd(3) + L*c3*s1*s2*qd(3);
Jd(1,2)=s1*qd(1)*(L*(s2 + 1) - L + L*c2*s3 + L*c3*s2) - ...
    c1*(L*c2*qd(2) - L*s2*s3*qd(2) - L*s2*s3*qd(3) + ...
    L*c2*c3*qd(2) + L*c2*c3*qd(3));
Jd(1,3)=c1*(L*s2*s3*qd(2) + L*s2*s3*qd(3) - L*c2*c3*qd(2) - ...
    L*c2*c3*qd(3)) + s1*qd(1)*(L*c2*s3 + L*c3*s2);
Jd(2,1)=L*s1*s2*s3*qd(1) - L*c1*s2*qd(2) - L*c2*c3*s1*qd(1) -...
    L*c1*c2*s3*qd(2) - L*c1*c3*s2*qd(2) - L*c1*c2*s3*qd(3) -...
    L*c1*c3*s2*qd(3) - L*c2*s1*qd(1);
Jd(2,2)=- s1*(L*c2*qd(2) - L*s2*s3*qd(2) - L*s2*s3*qd(3) + ...
    L*c2*c3*qd(2) + L*c2*c3*qd(3)) - c1*qd(1)*(L*(s2 + 1) -...
    L + L*c2*s3 + L*c3*s2);
Jd(2,3)=s1*(L*s2*s3*qd(2) + L*s2*s3*qd(3) - L*c2*c3*qd(2) - ...
    L*c2*c3*qd(3)) - c1*qd(1)*(L*c2*s3 + L*c3*s2);
Jd(3,1)=0;
Jd(3,2)=c1*qd(1)*(L*c2*s1 - L*s1*s2*s3 + L*c2*c3*s1) - ...
    s1*(L*s1*s2*qd(2) - L*c1*c2*qd(1) - L*c1*c2*c3*qd(1) +...
    L*c1*s2*s3*qd(1) + L*c2*s1*s3*qd(2) + L*c3*s1*s2*qd(2) +...
    L*c2*s1*s3*qd(3) + L*c3*s1*s2*qd(3)) - s1*qd(1)*(L*c1*c2 +...
    L*c1*c2*c3 - L*c1*s2*s3) - c1*(L*c2*s1*qd(1) + L*c1*s2*qd(2) +...
    L*c2*c3*s1*qd(1) + L*c1*c2*s3*qd(2) + L*c1*c3*s2*qd(2) +...
    L*c1*c2*s3*qd(3) + L*c1*c3*s2*qd(3) - L*s1*s2*s3*qd(1));
Jd(3,3)=- c1*(L*c2*c3*s1*qd(1) + L*c1*c2*s3*qd(2) + L*c1*c3*s2*qd(2)...
    + L*c1*c2*s3*qd(3) + L*c1*c3*s2*qd(3) - L*s1*s2*s3*qd(1)) - ...
    s1*(L*c1*s2*s3*qd(1) - L*c1*c2*c3*qd(1) + L*c2*s1*s3*qd(2) +...
    L*c3*s1*s2*qd(2) + L*c2*s1*s3*qd(3) + L*c3*s1*s2*qd(3)) -...
    s1*qd(1)*(L*c1*c2*c3 - L*c1*s2*s3) - c1*qd(1)*(L*s1*s2*s3 -...
    L*c2*c3*s1);

% mass matrix calculation
D = Dmatrix_ThreeDOFcontrols_OP(q,modelParameters);

% calculate D, B, D, and G matrices
B = Bmatrix_ThreeDOFcontrols(q,modelParameters);
C = Cmatrix_ThreeDOFcontrols(q,modelParameters);
V = B*[qd(1)*qd(2);qd(1)*qd(3);qd(2)*qd(3)]...
    +C*[qd(1)^2;  qd(2)^2; qd(3)^2];
```

```
    % gravity vector
    G = Gvector_ThreeDOFcontrols(q,modelParameters);

    % evaluate inverese of the Jacobian
    [u,s,v] = svd(J);
    sInv = eye(size(J));
    for i = 1: size(J,1)
         if s(i,i) < .01
             sInv(i,i) = 0;
         else
             sInv(i,i) = 1/s(i,i);
         end
     end
    Jinv = v*sInv*u';

    % evaluate operational space terms
    L0  = Jinv'*D*Jinv;
    p   = Jinv'*G;
    mu = Jinv'*V - L0*Jd*qd;

    % op space position and velocity. THis was found in Three_DOF_symbolic.m
    x  = [ L*c1*c2 + L*c1*c2*c3 - L*c1*s2*s3;
           L*c2*s1 - L*s1*s2*s3 + L*c2*c3*s1;
           L*(s2 + 1) + L*c2*s3 + L*c3*s2];
    xd = J*qd;

    % decoupled system PD-controller torques
    wn       = 2*2*pi; %using 2Hz per HW problem specs
    zeta     = 1;  %critical damping, per HW problem specs
    Kp       = wn^2;
    Kd       = 2*zeta*wn;


    % task-space velocity limiting heuristic
    if modelParameters.controlMethod ==3
       vMax   = modelParameters.vMax;
       xError = norm(xDes - x);
       %xErrorMag = sqrt(xError(1)^2 + xError(2)^2);
       KpMax     = Kd*vMax/xError; %xErrorMag;
       if Kp > KpMax
          Kp = KpMax;
       end
    end

    Fprime = -Kd*(xd - xdDes) - Kp*(x - xDes);  %PD controller
    F = L0*Fprime + p + mu;
    tau = J'*F;

    %form joint acceleration vector
    qdd = D\(tau -V - G);
    % assign state variable derivatives
    zDot = [qd; qdd];

end
```