Robotics Project

Collaborators

1. Γιαννουσάκης Κωστής 6975
2. Κοντουράς Ευστάθιος 7017

Subject

More advanced control of a robotic manipulator.

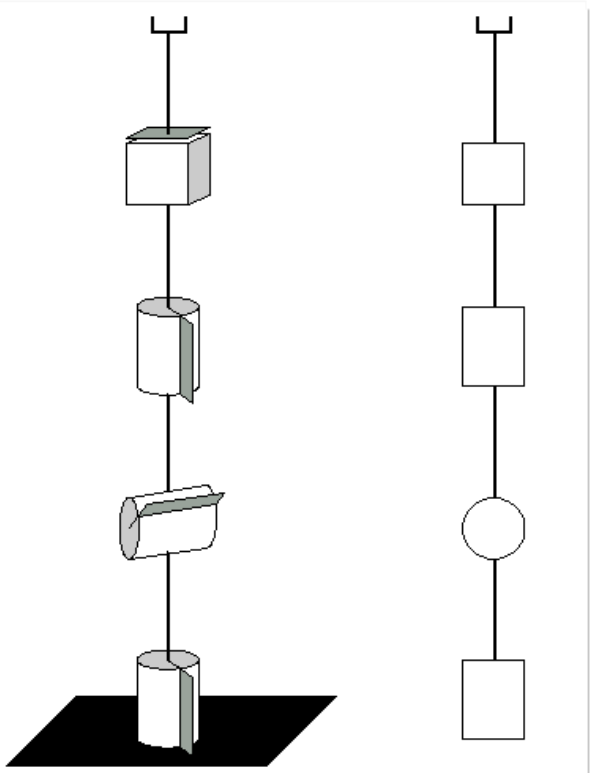

Figure 1: Project Manipulator

Table Of Contents

1. Forward Kinematics

• Analysis

By applying the Denavid - Hartenberg method we get:

LINK	a_i	α_i	d_i	θ_i
1	0	-90°	0	θ_1*
2	0	90°	0	θ_2*
3	0	0	0	θ_3*
4	0	0	d_4*	0

• Code

```
% FKMatrices.m

syms th1 th2 th3 d4

% Create the Denavid - Harterberg Values Matrix
DHV = [
        [ 0, -pi/2,  0, th1]
        [ 0,  pi/2,  0, th2]
        [ 0,     0,  0, th3]
        [ 0,     0, d4,   0]
      ];

[T, linksTrans] = FK(DHV);

T = simplify(T);
```

• Results

Using the above code we get all the matrices:

$$A_1^0 = \begin{bmatrix} \cos\theta_1 & 0 & -\sin\theta_1 & 0 \\ \sin\theta_1 & 0 & \cos\theta_1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad A_2^1 = \begin{bmatrix} \cos\theta_2 & 0 & \sin\theta_2 & 0 \\ \sin\theta_2 & 0 & -\cos\theta_2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$A_3^2 = \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 0 & 0 \\ \sin\theta_3 & \cos\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad A_4^3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_4^0 = \begin{bmatrix} \cos\theta_1\cos\theta_2\cos\theta_3 - \sin\theta_1\sin\theta_3 & -\cos\theta_3\sin\theta_1 - \cos\theta_1\cos\theta_2\sin\theta_3 & \cos\theta_1\sin\theta_2 & d_4\cos\theta_1\sin\theta_2 \\ \cos\theta_1\sin\theta_3 + \cos\theta_2\cos\theta_3\sin\theta_1 & \cos\theta_1\cos\theta_3 - \cos\theta_2\sin\theta_1\sin\theta_3 & \sin\theta_1\sin\theta_2 & d_4\sin\theta_1\sin\theta_2 \\ -\cos\theta_3\sin\theta_2 & \sin\theta_2\sin\theta_3 & \cos\theta_2 & d_4\cos\theta_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. Inverse Kinematics

- **Analysis**

We can notice that the robot is functioning in a spherical coordinate frame.

Let (x,y,z) be the desired position:

$$d_4 = \sqrt{x^2 + y^2 + z^2}$$

$$\theta_1 = \tan^{-1}\frac{y}{x}$$

$$\theta_2 = \tan^{-1}\frac{\sqrt{x^2 + y^2}}{z}$$

θ_3 does not affect the position of the end effector and so it can take any value.

If the rotation matrix R is given too, then:

$$\theta_3 = \tan^{-1}\left(-\frac{r_{3,2}}{r_{3,1}}\right)$$

- **Code**

The basic implementation is the `RIK` function in the 'Appendix' section.

3. Trajectory Planning

- **Analysis**

To create a circle in the 3D space we implemented the function `P3DCircle` which returns N points on the desired circle, with equal arch lengths.

By segmenting the desired time (8s) at N segments of equal interval and connecting the N points with the N time values, we achieve a constant mean linear velocity for all segments.

- **Code**

```
% TrajectoryPlanning.m

N = 20; % Original count of points on the circle
[X Y Z] = P3DCircle(2,3,3,8,1,1,N);
figure(1); plot3(X, Y, Z, 'mo')
xlabel('X axis'), ylabel('Y axis'), zlabel('Z axis')

Qp = zeros(N,3);
[Qp(:,1), Qp(:,2), Qp(:,4)] = RMIK(X,Y,Z);

Qp(:,3) = linspace(0, 2*pi, N);

% Trajectory creation
Tp  = linspace(0, 8, N);
Qdp = zeros(N, 4);

% Wanted velocity at every point
Qdp(2:(N-1),:) = 0.1*ones((N-2),4);

Ni = 0; % Points to create between circle points
[Q, Qd, Qdd, T] = jmtraj(Qp, Tp, Ni, Qdp);
```

And then we plot the results:

```
% Estimated Points
[Xe Ye Ze] = RMFK(Q(:,1), Q(:,2), Q(:,3), Q(:,4));
figure(1), hold on, plot3(Xe, Ye, Ze, 'r')
figure(1), legend('Trajectory Points', 'Path')

figure, plot(T, Xe, 'r')
title('X Trajecrory')
xlabel('Time (s)')
ylabel('X (cm)')

figure, plot(T, Ye, 'r')
title('Y Trajecrory')
xlabel('Time (s)')
ylabel('Y (cm)')

figure, plot(T, Ze, 'r')
title('Z Trajecrory')
xlabel('Time (s)')
ylabel('Z (cm)')
```

- **Results**

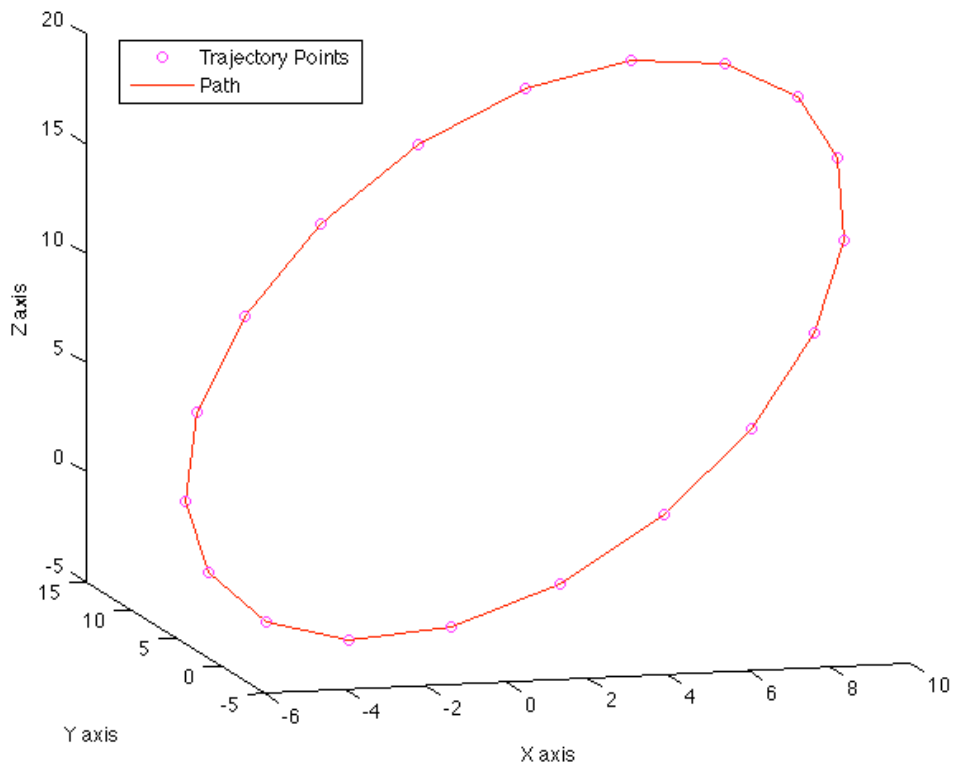

Figure 2: Desired Trajectory

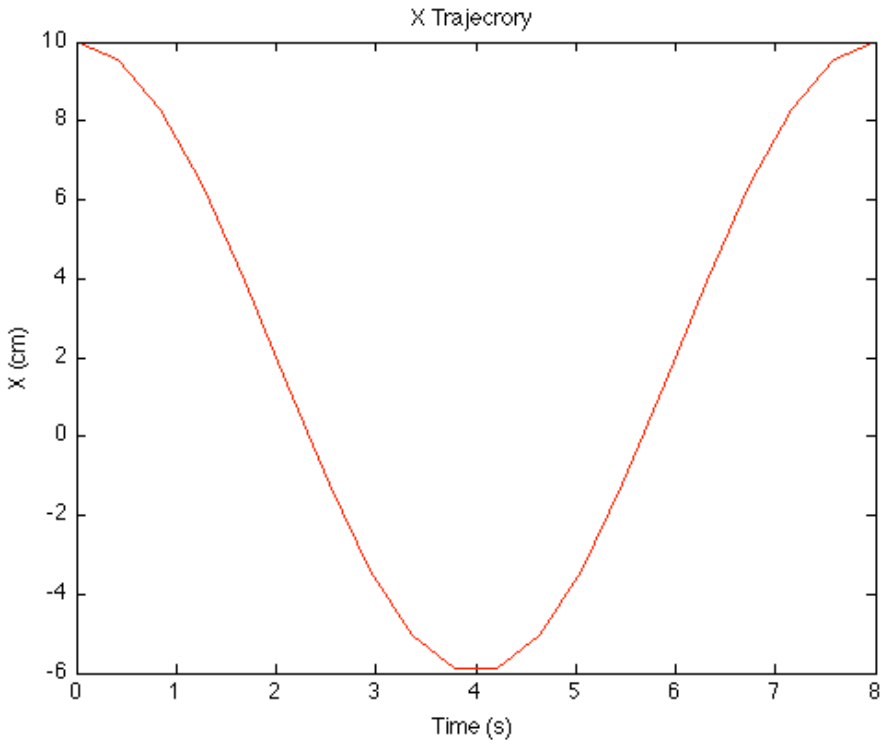

Figure 3: Desired X axis Trajectory

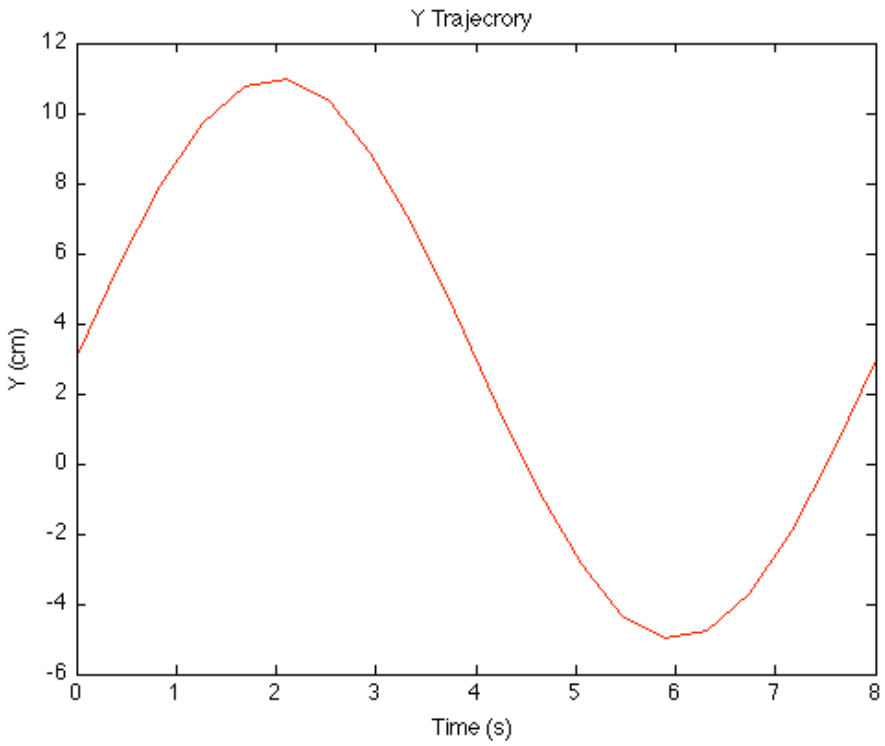

Figure 4: Desired Y axis Trajectory

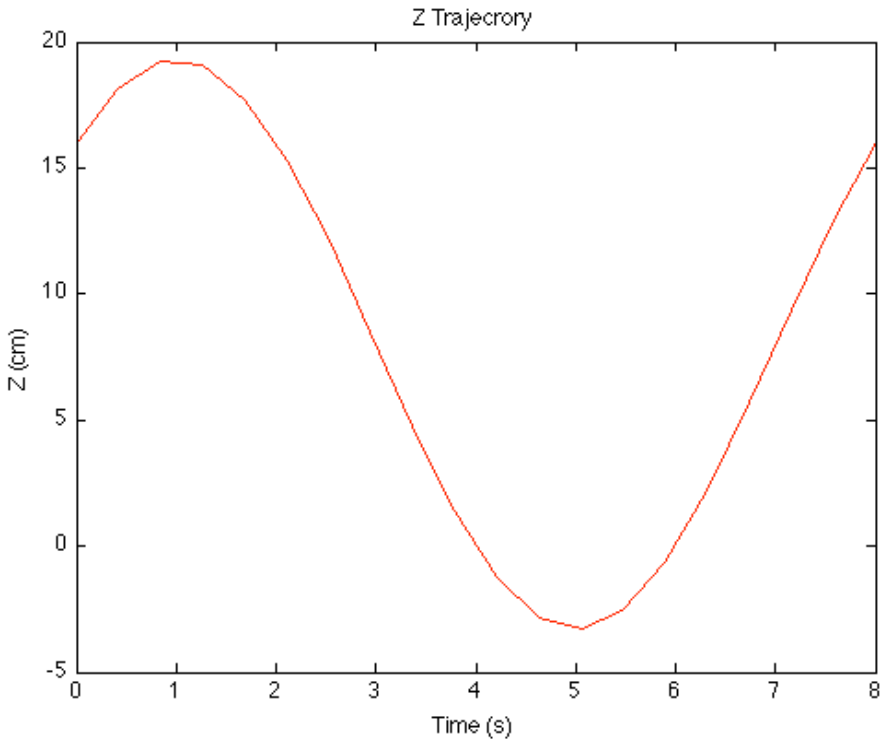

Figure 5: Desired Z axis Trajectory

4. Dynamic Equations

- **Analysis**

1. Jacobian

The Jacobian matrix has a number of columns equal to the number of the joints, where:

$$J_i = \begin{cases} \begin{bmatrix} z_{i-1} \times (o_n - o_{i-1}) \\ z_{i-1} \end{bmatrix} & \text{for revolute} \\ \begin{bmatrix} z_{i-1} \\ 0 \end{bmatrix} & \text{for prismatic} \end{cases}$$

2. Inertia Matrix

If n is the joints count the the inertia matrix is given by:

$$D = \sum_{i=0}^{n} \left(m_i \left(J_{v_i}^T \cdot J_{v_i} \right) + I_i \left(J_{\omega_i}^T \cdot J_{\omega_i} \right) \right)$$

3. Coriolis Matrix

If n is the joints count the the coriolis matrix elements are given by:

$$c_{kj} = \sum_{i=1}^{n} \frac{1}{2} \left\{ \frac{\partial d_{kj}}{\partial q_j} + \frac{\partial d_{ki}}{\partial q_j} - \frac{\partial d_{ij}}{\partial q_k} \right\} \dot{q}_i$$

where d_{ij} are inertia matrix elements and q_i are the joint variables.

4. Potential Matrix

The potential matrix elements are given by:

$$\phi_k = \frac{\partial P}{\partial q_k}$$

where P is the potential energy of the joint.

5. Euler-Lagrange dynamic equations

Finally we get the Euler-Lagrange dynamic equations by:

$$\sum_i d_{kj}(q)\ddot{q}_j + \sum_{i,j} c_{ikj}(q)\dot{q}_j + \phi_k(q) = \tau_k$$

The matrices are computed using the `Jacobian`, `inertiaMatrix`, `coriolisMatrix`, `potentialMatrix`, `ELMatrix` functions respectively.

- **Code**

```
% Dynamics.m

th1   = sym('th1'  , 'real');
th2   = sym('th2'  , 'real');
th3   = sym('th3'  , 'real');
d4    = sym('d4'   , 'real');
th1d  = sym('th1d' , 'real');
th2d  = sym('th2d' , 'real');
th3d  = sym('th3d' , 'real');
d4d   = sym('d4d'  , 'real');
th1dd = sym('th1dd', 'real');
th2dd = sym('th2dd', 'real');
th3dd = sym('th3dd', 'real');
d4dd  = sym('d4dd' , 'real');

Qs    = [  th1   th2   th3   d4];
Qds   = [ th1d  th2d  th3d  d4d]; % First  derivative
Qdds  = [th1dd th2dd th3dd d4dd]; % Second derivative

m     = 1;
M     = [    m     m     m    m];
len   = [    0     0     0   d4];
I     = M .* len;

g     = 9.81;
G     = [  0  0  g];

% Create the Denavid - Harterberg Values Matrix
DHV = [
        [ 0, -pi/2,  0, th1]
        [ 0,  pi/2,  0, th2]
        [ 0,     0,  0, th3]
        [ 0,     0, d4,   0]
      ];

[~, FKLM] = FK(DHV);

EL = ELMatrix(FKLM, Qs, Qds, Qdds, M, I, G);
```

- **Results**

$$\tau_1 = \ddot{\theta}_1\left(d_4^2\sin^2\theta_2 + d_4\right) + \ddot{\theta}_3 d_4\cos\theta_2 + \dot{\theta}_2\left(\dot{\theta}_1 d_4^2\cos\theta_2\sin\theta_2 - \frac{\dot{\theta}_3 d_4\sin\theta_2}{2}\right) + \dot{d}_4\left(\dot{\theta}_1\left(d_4\sin^2\theta_2 + \frac{1}{2}\right) + \frac{\dot{\theta}_3\cos\theta_2}{2}\right)$$

$$\tau_2 = d_4\ddot{\theta}_2 + \frac{\dot{d}_4\dot{\theta}_2}{2} + d_4^2\ddot{\theta}_2 - \frac{981\cdot d_4 sin\theta_2}{100} - \frac{d_4^2\dot{\theta}_2^2 sin(2\theta_2)}{2} + d_4\dot{d}_4\dot{\theta}_2 + d_4\dot{\theta}_1\dot{\theta}_3 sin\theta_2$$

$$\tau_3 = d_4\ddot{\theta}_3 + \dot{d}_4\left(\frac{\dot{\theta}_3}{2} + \frac{\dot{\theta}_1\cos\theta_2}{2}\right) + d_4\ddot{\theta}_1\cos\theta_2 - \frac{d_4\dot{\theta}_1\dot{\theta}_2\sin\theta_2}{2}$$

$$\tau_4 = \ddot{d}_4 + \frac{981\cos\theta_2}{100} - d_4\dot{\theta}_2^2 - \frac{\dot{\theta}_1^2 + \dot{\theta}_2^2 + \dot{\theta}_3^2}{2} - \dot{\theta}_1\dot{\theta}_3\cos\theta_2 + d_4\dot{\theta}_1^2\left(\cos^2\theta_2 - 1\right)$$

5. Feedforward Generalized Forces - Torques

- **Code**

```
% TrajectoryPlanningTorques.m

% Create the needed variables
TrajectoryPlanning
Dynamics

% Get Generalized Torques Values
Tor = GenTorques(EL, Q, Qs, Qd, Qds, Qdd, Qdds);

figure , plot(T, Tor(:,1), '-b')
hold on, plot(T, Tor(:,2), '-m')
hold on, plot(T, Tor(:,3), '-c')
hold on, plot(T, Tor(:,4), '-r')
legend('T1', 'T2', 'T3', 'T4')
title('Torques')
```

- **Results**

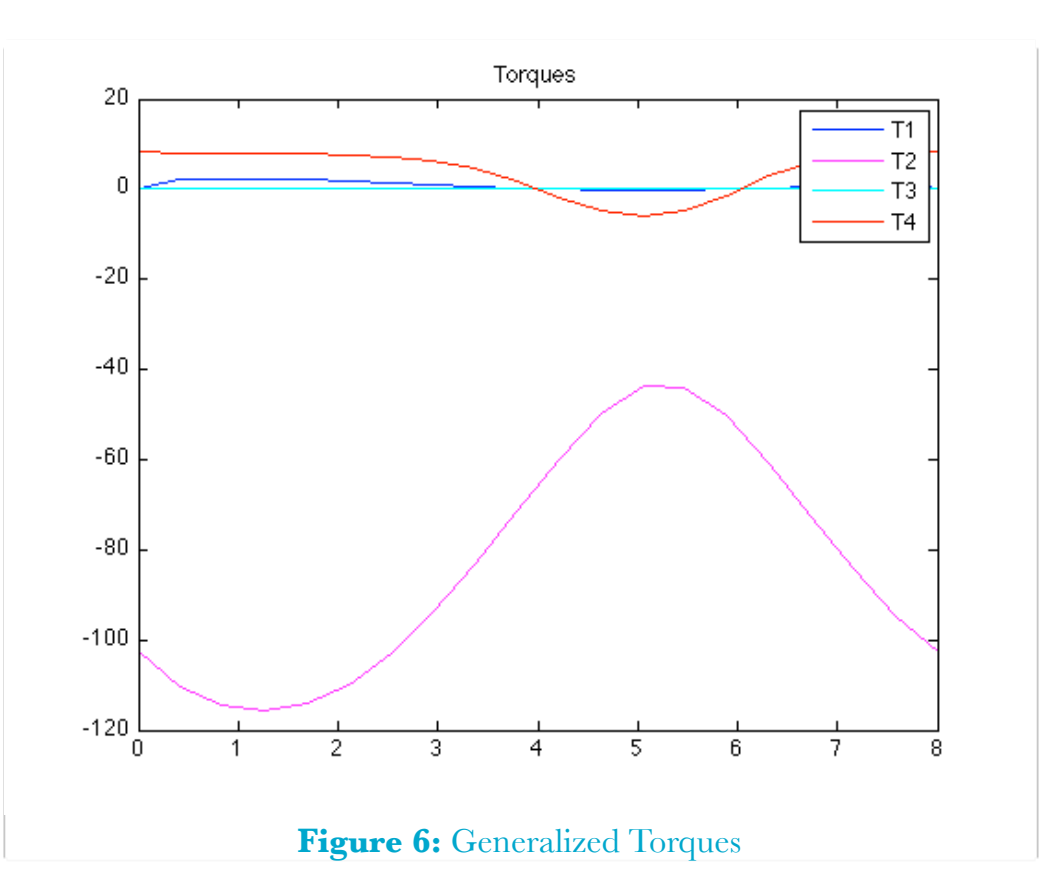

Figure 6: Generalized Torques

6. Trajectory with Torques into account

- **Analysis**

The mathematical analysis of the following system is generally known. In what follows, a brief proof of the transfer function model is presented.

- R_r = rotor electrical resistance of a DC - motor (Ω)
- L_r = rotor inductance DC - motor (H)
- U = inducted voltage (V)
- V = supply voltage (V)

$$V(t) = R_r i(t) + L_r \frac{di(t)}{dt} + U(t) \quad (1)$$

$$U(t) = C\Phi \cdot \dot{\theta}(t) \quad (2)$$

$$\tau(t) = C\Phi \cdot i(t) \quad (3)$$

Applying Laplace transform to (1), (2) and (3) and solving for the torque we obtain:

$$\tau(s) = C\Phi \frac{V(s) - s \cdot C\Phi \cdot \theta(s)}{R_r + s \cdot L_r} \quad (4)$$

Applying Newton's second law we obtain:

$$\tau(t) = J_{eff} \cdot \ddot{\theta}(t) + \beta_{eff} \cdot \dot{\theta}(t) \quad (5)$$

Applying Laplace transform to (5):

$$\tau(s) = s^2 \cdot J_{eff} \cdot \theta(s) + s \cdot \beta_{eff} \cdot \theta(s) \quad (6)$$

Finally from (4) and (6) we have:

$$\frac{\theta(s)}{V(s)} = \frac{C\Phi}{s\left[(R_r \cdot J_{eff})s + R_r \cdot \beta_{eff} + (C\Phi)^2\right]} \quad (7)$$

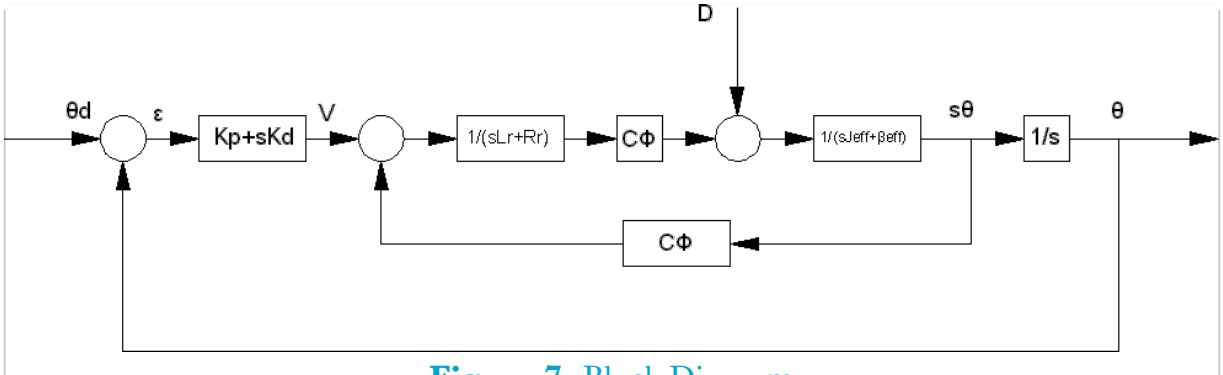

Figure 7: Block Diagram

We use a PD controller.

- The proportional term determines the response time.

- The derivative term determines the damping rate as well as the steady state error.

Using simple methods we easily obtain the transfer function for the closed loop system considering the existence of the PD controller:

$$H(s)=\frac{C\Phi\cdot K_d\cdot s+C\Phi\cdot K_p}{s^2\left(R_r\cdot J_{eff}\right)+s\left(R_r\cdot\beta_{eff}+(C\Phi)^2+C\Phi\cdot K_d\right)+C\Phi\cdot K_p}$$

The zero of $H(s)$ is responsible for the settling time and the overshoot of the response.

Considering the disturbance $D(s)$ (associated with gravity, Coriolis and centripetal terms) caused by the other movable links of the manipulator we have:

$$\tau(s)=s^2\cdot J_{eff}\cdot\theta(s)+s\cdot\beta_{eff}\cdot\theta(s)+D(s)$$

Finally using the superposition principle we obtain:

$$\theta(s)=\frac{C\Phi\left(K_p+sK_d\right)\theta_d(s)-R_r\cdot D(s)}{s^2\left(R_r\cdot J_{eff}\right)+s\left(R_r\cdot\beta_{eff}+(C\Phi)^2+C\Phi\cdot K_d\right)+C\Phi\cdot K_p} \quad (8)$$

Where $\theta_d(s)$ is the desired position and $D(s)$ in (Nm) is the generalized torque disturbance.

- **Code**

Turning equation (8) into a differential equation and by setting $K_p=0$, $K_d=0$ we can get the real actuator values.

```
% TrajectoryPlanningRealPath.m

% Create the needed variables
TrajectoryPlanningTorques

%% Get the 'real' Trajecory with no controllers

[Qc, Qdc, Qddc, Kp, Kd] = GetRealValues(T, Q, Qd, Qdd,
Tor, zeros(4,2), 'k');


[Xc Yc Zc] = RMFK(Qc(:,1), Qc(:,2), Qc(:,3), Qc(:,4));
figure(1),hold on, plot3(Xc, Yc, Zc, 'r')
figure(1), legend('Points', 'Wanted Path', 'Estimated
Path')

figure(2),hold on, plot(T, Xc, 'r')
legend('Wanted', 'Estimated')

figure(3),hold on, plot(T, Yc, 'r')
legend('Wanted', 'Estimated')

figure(4),hold on, plot(T, Zc, 'r')
legend('Wanted', 'Estimated')
```

- **Results**

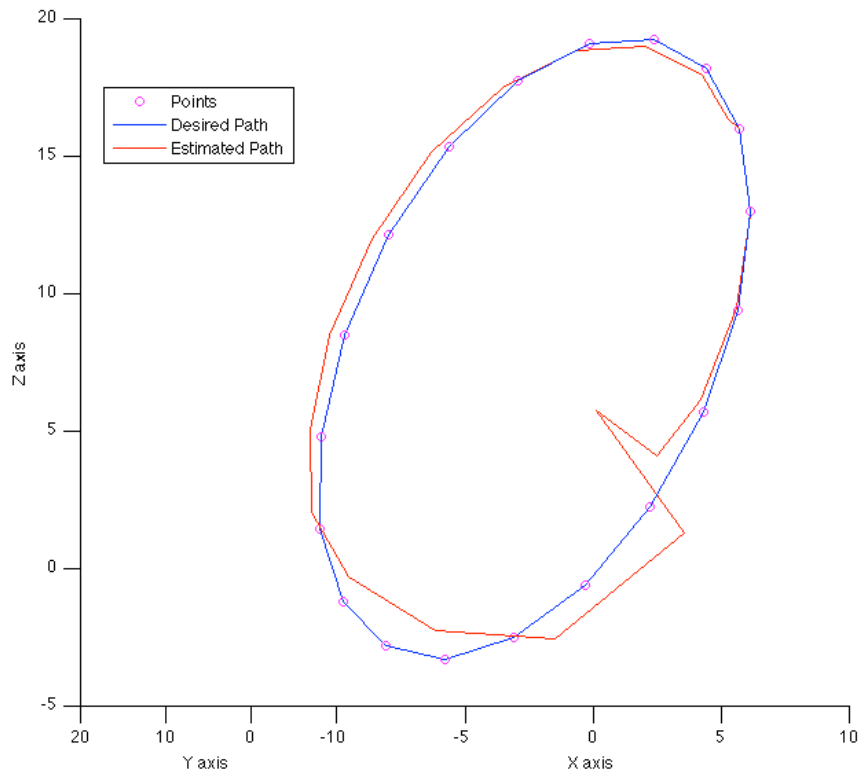

Figure 8: Estimated Trajectory

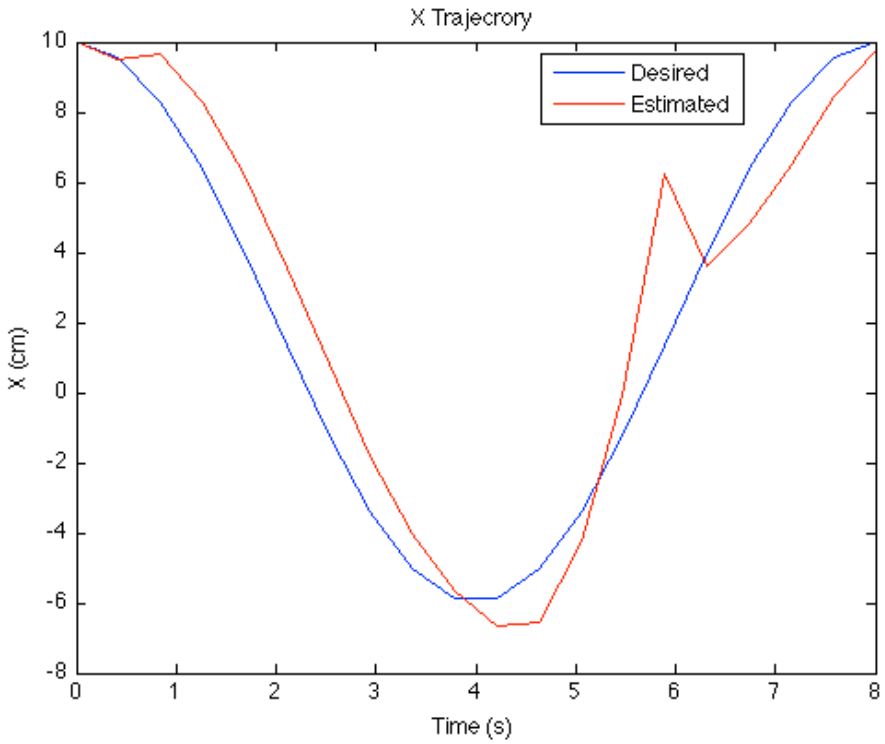

Figure 9: Estimated X Trajectory

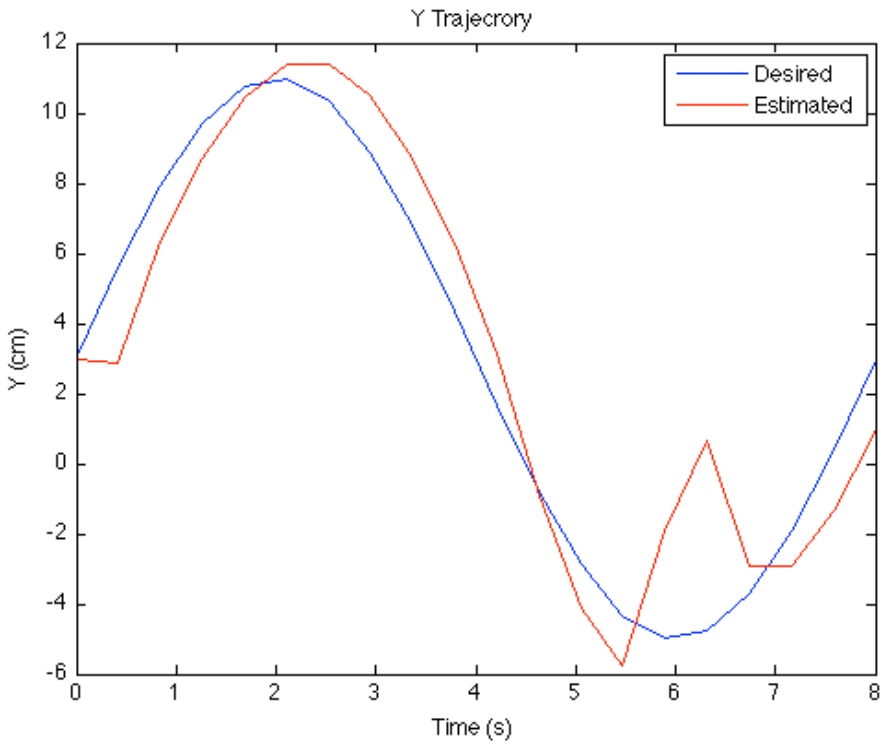

Figure 10: Estimated Y Trajectory

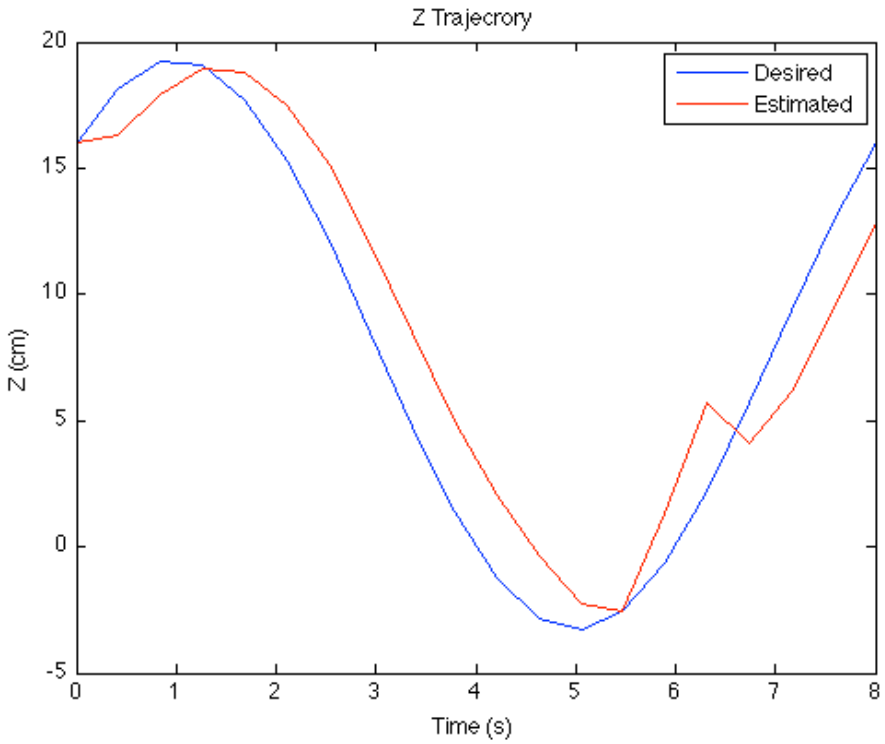

Figure 11: Estimated Z Trajectory

7. Pole Placement

- **Analysis**

  All the analysis is done at section 6 '*Trajectory with Torques into account*'.

  Here we have to determine the PD values of the controller of every joint.

- **Code**

```
% TrajectoryPlanningController.m

% Create the needed variables
TrajectoryPlanningTorques

A = [1 2 10 100];

[Qc, Qdc, Qddc, Kp, Kd] = GetRealValues(T, Q, Qd, Qdd,
Tor, [A' A'], 'p');

%% Get the estimated Trajectory
[Xc Yc Zc] = RMFK(Qc(:,1), Qc(:,2), Qc(:,3), Qc(:,4));
figure(1), plot3(Xc, Yc, Zc, 'r')
figure(1), figure(1), legend('Points', 'Desired Path',
'Estimated Path')

figure(2),hold on, plot(T, Xc, 'r')
legend('Desired', 'Estimated')

figure(3),hold on, plot(T, Yc, 'r')
legend('Desired', 'Estimated')

figure(4),hold on, plot(T, Zc, 'r')
legend('Desired', 'Estimated')


%% Get Generalized Torques Values
Torc = GenTorques(EL, Qc, Qs, Qdc, Qds, Qddc, Qdds);

figure , plot(T, Torc(:,1), '-b')
hold on, plot(T, Torc(:,2), '-m')
hold on, plot(T, Torc(:,3), '-c')
hold on, plot(T, Torc(:,4), '-r')
legend('T1', 'T2', 'T3', 'T4')
title('Torques')
```

- **Results**

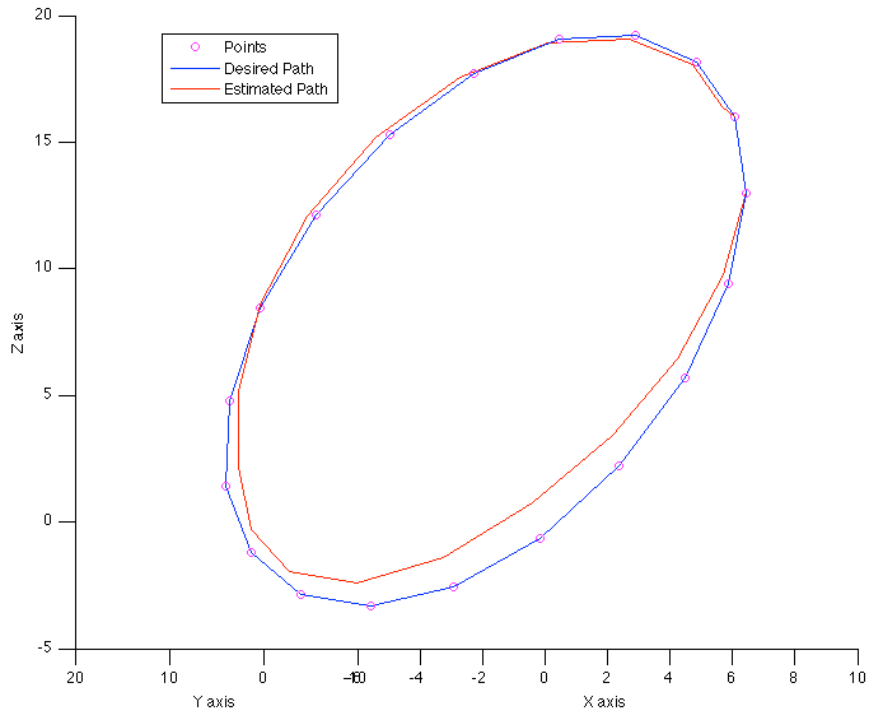

Figure 12: Estimated Trajectory

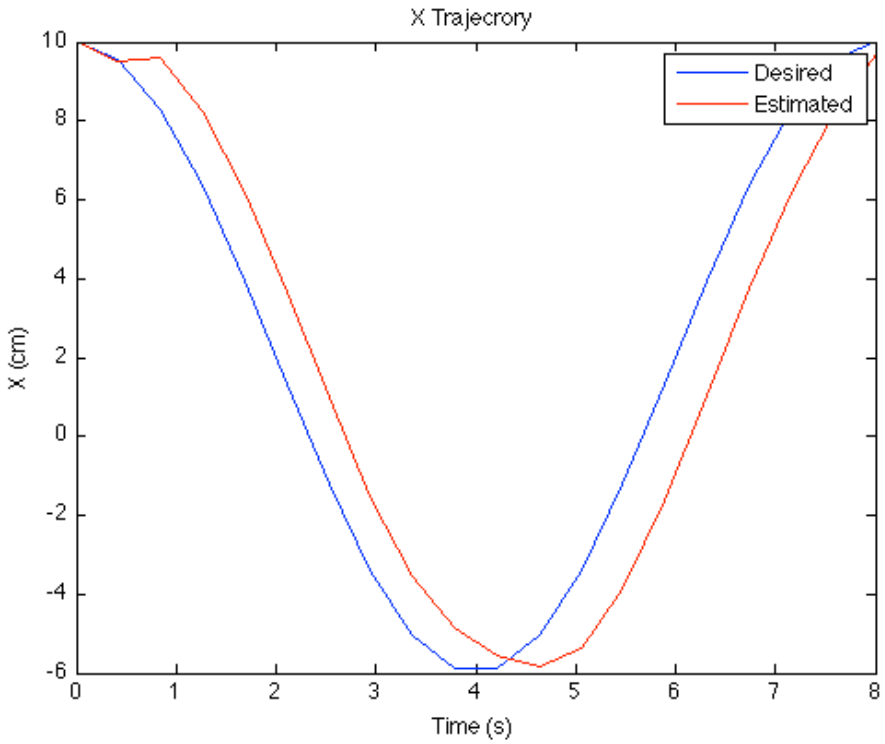

Figure 13: Estimated X Trajectory

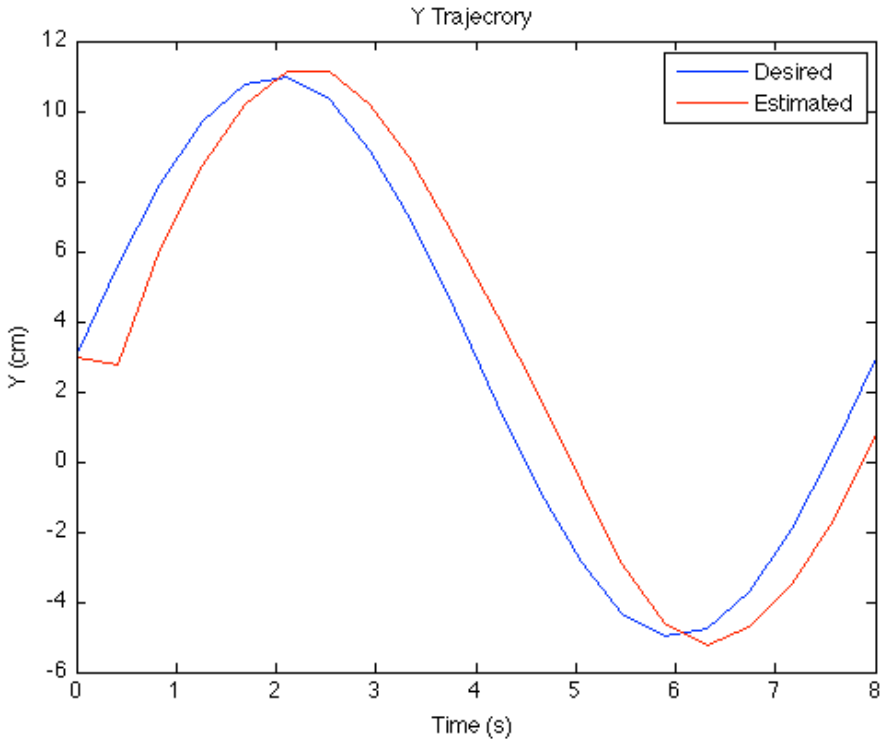

Figure 14: Estimated Y Trajectory

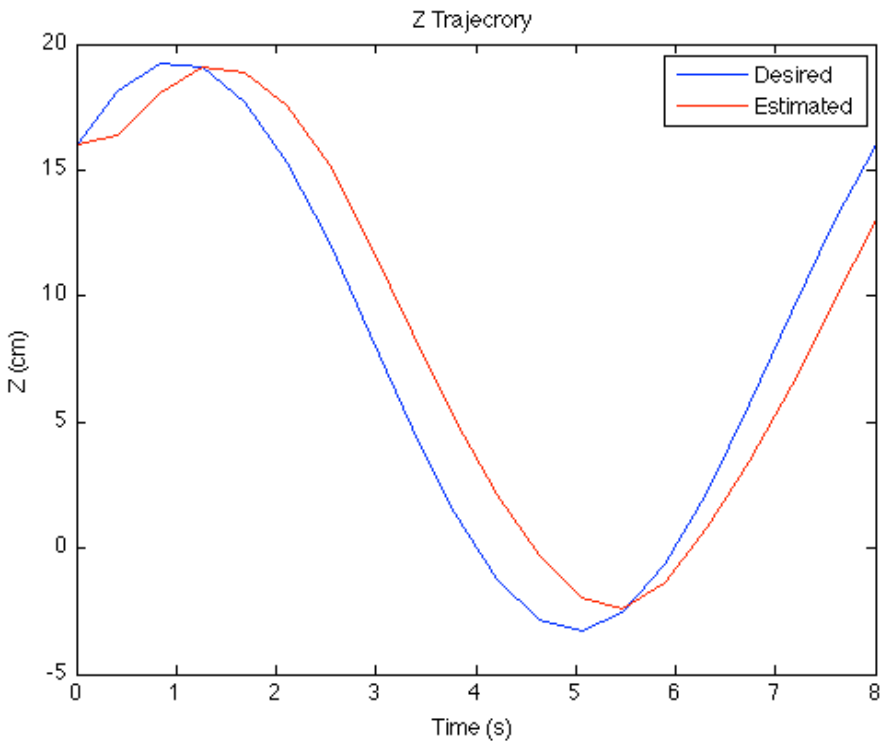

Figure 15: Estimated Z Trajectory

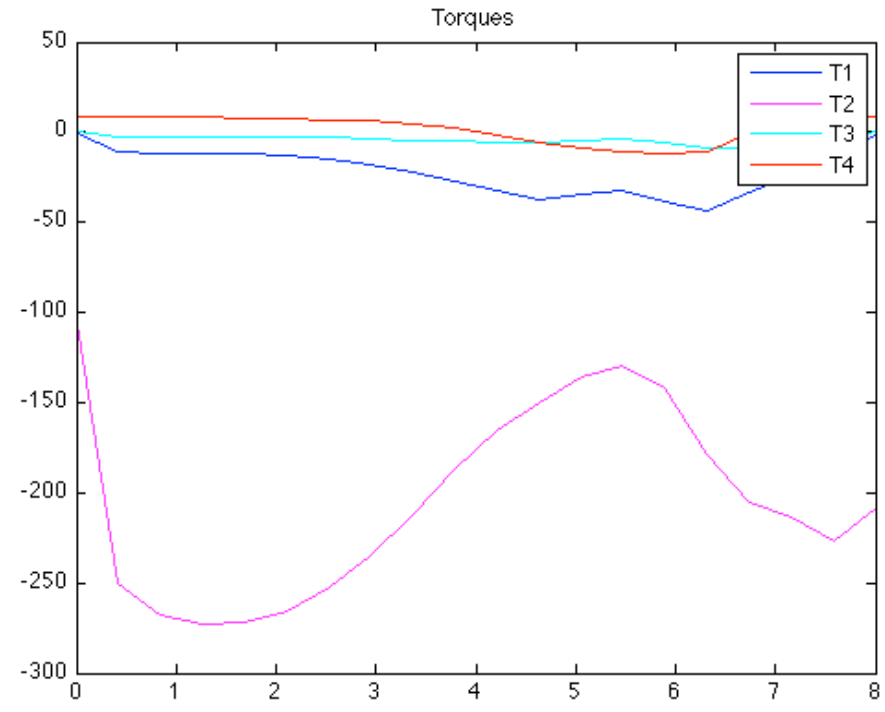

Figure 16: Generalized Torques

Appendix

- **Manipulator Specific Functions**

1. **RIK**

```
% Inverce Kinematics for the project robot

function [th1, th2, d] = RIK(x,y,z)
% spherical coordinate frame with respect to
% cartesian coordinate frame
th1 = atan2(y,x);
th2 = atan2(sqrt(x^2 + y^2),z);

d = sqrt(x^2 + y^2 + z^2);

if th1 < 0
    th1 = th1 +2*pi;
end

if th2 < 0
    th2 = th2 +2*pi;
end
```

2. **RFK**

```
% Forwarg Kinematics for the project robot
function [x, y, z] = RFK(th1, th2, th3, d4)

T = [
[(cos(th1)*cos(th2)*cos(th3) - sin(th1)*sin(th3)),...
(- cos(th3)*sin(th1) - cos(th1)*cos(th2)*sin(th3)),...
(cos(th1)*sin(th2)), (d4*cos(th1)*sin(th2))];
[(cos(th1)*sin(th3) + cos(th2)*cos(th3)*sin(th1)),...
(cos(th1)*cos(th3) - cos(th2)*sin(th1)*sin(th3)),...
(sin(th1)*sin(th2)), (d4*sin(th1)*sin(th2))];
[                            (-cos(th3)*sin(th2)),...
(sin(th2)*sin(th3)),          (cos(th2)),         ...
(d4*cos(th2))];
[                                                0,...
0,                   0,                      1];
    ];

P = T * [0 0 0 1]';

x = P(1);
y = P(2);
z = P(3);

end
```

3. RIKBest

```
% Compute Inverce Kinematics with known current state
% for the project robot
function [f, th, d] = RIKBest(x,y,z,f0,th0,d0)
% spherical coordinate frame with respect to
% cartesian coordinate frame
f  = atan2(y,x);
th = atan2(sqrt(x^2 + y^2),z);

d = sqrt(x^2 + y^2 + z^2);

if f < 0
    f = f +2*pi;
end

if abs(f - f0) > abs(2*pi + f - f0)
    f = f +2*pi;
end

if th < 0
    th = th +2*pi;
end

if abs(th - th0) > abs(2*pi + th - th0)
    th = th +2*pi;
end
```

4. RMFK

```
% Multiple values Forward Kinematics for the project
robot

function [X, Y, Z] = RMFK(TH1, TH2, TH3, D4)

N = length(TH1);

X = zeros(N,1);
Y = zeros(N,1);
Z = zeros(N,1);

for i = 1:N

   [X(i), Y(i), Z(i)] = RFK(TH1(i), TH2(i), TH3(i),
D4(i));

end
```

5. RMIK

```
function [f, th, d] = RMIK(X,Y,Z)

N  = length(X);

f  = zeros(N,1);
th = zeros(N,1);
d  = zeros(N,1);

[f(1), th(1), d(1)] = RIK(X(1), Y(1), Z(1));

for i = 2:N

    [f(i), th(i), d(i)] = RIKBest(X(i), Y(i),...
        Z(i), f(i-1), th(i-1), d(i-1));

end
```

6. GetRealValues_F

```
function xp = GetRealValues_F(t,x)
xp = zeros(2,1);
xp(1) = x(2);

load GetRealValues_el
load GetRealValues_J
load GetRealValues_Values

for i = 1:length(T)
    if T(i) >= t
        break;
    end
end


Th  = Q(i,el);
Thd = Qd(i,el);
D   = Tor(i,el);

xp(2) = -CF*Kp(el) /(R*J)*x(1)+(R*B+CF^2+CF*Kd(el))/
(R*J)*x(2)...
    +CF*Kp(el)*Th/(R*J) + CF*Kd(el)/(R*J)*Thd + D/J;
```

7. GetRealValues

```
% Get qi values and its derivatives based on the
desired values, the
% torques and the PD values at every joint of the
project manipulator
%
%    T: Time values
%    Q: Joint qi values list
%   Qd: Joint qi first  derivative values list
%  Qdd: Joint qi second derivative values list
%  Tor: Torques values list
%    p: 4x2 PD values or 4x2 poles values for every
joint
%  txt: Define p parameter. 'p' poles, 'k', PD

function [Qc, Qdc, Qddc, Kp, Kd] = GetRealValues(T, Q,
Qd, Qdd, Tor, p, txt)

    CF = 1;
    R  = 0.003;
    J  = 0.01;
    B  = 0;

    if not(exist('txt','var'))
        txt = 'k';
    end

    if txt ~= 'p'
        Kp = p(:,1);
        Kd = p(:,2);
    else

        for i = 1:4
            Kdp = sym('Kdp', 'real');
            Kpp = sym('Kpp', 'real');

            S = solve((R*B+CF^2+CF*Kdp)/(R*J)+CF*Kpp/
(R*J)...
                - (p(i,1) + p(i,2)),...
                Kpp - p(i,1)*p(i,2) * R*J / CF, 'Kpp,
Kdp');

            Kp(i) = double(S.Kpp);
            Kd(i) = double(S.Kdp);
        end

    end

% ...
```

```
% ...

Qc   = zeros(size(Q));
    Qdc  = zeros(size(Qd));
    Qddc = zeros(size(Qdd));

    Qc(1,:)   = Q(1,:);
    Qdc(1,:)  = Qd(1,:);
    Qddc(1,:) = Qdd(1,:);

    save GetRealValues_Values T Q Qd Qdd Tor CF R B Kp
Kd

    for el = 1:4

        save GetRealValues_el el

        for i = 2:length(T)

            J = Q(i-1,4)^2;

            save GetRealValues_J J

            [t,x] = ode45(@GetRealValues_F,
[T(i-1),T(i)],[Q(i-1,el),Qd(i-1,el)]);

            lastIdx = length(t);

            Qc(i,el)   = x(lastIdx,1);
            Qdc(i,el)  = x(lastIdx,2);
            Qddc(i,el) = (x(lastIdx,2) -
x(lastIdx-1,2))/(t(lastIdx)-t(lastIdx-1));

        end

    end

end
```

- **General Robotics Functions**

  1. **DHTrans**

```
% Computes the Homogeneous Transformation using the
% Denavid-Hartenberg convention
%
% A = DHTrans(a, alpha, d, theta)
%
%
% EXAMPLE
%
%   syms theta
%   A = DHTrans(0, 0, 0, theta)

function A = DHTrans(a, alpha, d, theta)

cTh = cos(theta);
sTh = sin(theta);
cAl = cos(alpha);
sAl = sin(alpha);

A = [
        [cTh, -sTh*cAl,  sTh*sAl, a*cTh];
        [sTh,  cTh*cAl, -cTh*sAl, a*sTh];
        [ 0 ,    sAl  ,    cAl  ,   d  ];
        [ 0 ,     0   ,     0   ,   1  ]
    ];
```

2. FK

```
% Computes the Homogeneous Transformations using the
Denavid-Hartenberg convention
%
% [Trans, linkTrans] = FK(DHV);
%
%
% EXAMPLE
%
% [A02, linkTrans] = FK([a1, alpha1, d1, theta1;
%                        a2, alpha2, d2, theta2]);

function [T, linkTrans] = FK(DHP)

    rows      = length(DHP(:,1));
    linkTrans = sym(zeros(4,4,rows));
    T         = eye(4);

    for i = 1:rows
        A = DHTrans(DHP(i,1), DHP(i,2), DHP(i,3), DHP(i,4));
        T = T * A;

        linkTrans(:,:,i) = A;
    end

end
```

3. ELMatrix

```
% Compute the Euler-Lagrange dynamic equations asuming
that the frame
% assigned to every link is located at its center of
mass.
%
% FKLM: Link matrices with qi as sym
%    Q: syms of qi
%   Qd: syms of qi first  derivative
%  Qdd: syms of qi second derivative
%    M: links masses
%    I: inertia tensors
%    G: gravitational acceleration as column vector

function EL = ELMatrix(FKLM, Q, Qd, Qdd, M, I, G)

    D  = inertiaMatrix(FKLM, M, I);
    C  = coriolisMatrix(D, Q, Qd);
    F  = potentialMatrix(FKLM, M, G, Q);

    EL = simplify(D * Qdd' + C * Qd' + F);

end
```

4. inertiaMatrix

```
% Computes the inertia matrix given link matrices from
forward
% kinematcs...
%
% FKLM: Link matrices with qi as sym
%    M: links masses
%    I: inertia tensors

function D = inertiaMatrix(FKLM, M, I)

    linksCount = size(FKLM);
    if numel(linksCount) == 3
        linksCount = linksCount(3);
    else
        linksCount = 1;
    end

    D  = sym(zeros(linksCount, linksCount));
    LJ = LinksJacobian(FKLM);

    for i = 1:linksCount

        LVJ = LJ(1:3,:,i);
        AVJ = LJ(4:6,:,i);

        D = D + M(i) * (LVJ' * LVJ) + I(i) * (AVJ' *
AVJ);

    end

    D = simplify(D);

end
```

5. coriolisMatrix

```
% Computes the coriolis matrix
% assigned to every link is located at its center of
mass.
%
%    Q: syms of qi
%   Qd: syms of qi first  derivative
%    D: inetria matrix

function C = coriolisMatrix(D, Q, Qd)

    s = size(D);

    C = sym(zeros(s));

    for k = 1:s(1)
       for j = 1:s(1)
          for i = 1:s(1)

              C(k,j) = C(k,j) +
1/2*(diff(D(k,j),Q(j))...
                  + diff(D(k,i),Q(j)) -
diff(D(i,j),Q(k))) * Qd(i);

          end
       end
    end

    simplify(C);

end
```

6. potentialMatrix

```
% Compute the potential matrix
%
% FKLM: Link matrices with qi as sym
%    Q: syms of qi
%    M: links masses
%    G: gravitational acceleration as column vector

function F = potentialMatrix(FKLM, M, G, Q)

    linksCount = size(FKLM);
    if numel(linksCount) == 3
        linksCount = linksCount(3);
    else
        linksCount = 1;
    end

    F = sym(zeros(linksCount, 1));
    T = eye(4);
    P = 0;

    for i =1:linksCount

        T = T * FKLM(:,:,i);

        P = P + M(i) * G * T(1:3,4);

    end

    for i =1:linksCount

        F(i) = diff(P, Q(i));

    end

    F = simplify(F);

end
```

7. Jacobian

```
% Computes the Jacobian matrix given the link matrices
from forward kinematcs
%
% Trans (optional) is the transformation matrix of the
end effector.
% If omited it is auto computed.

function J = Jacobian(FKLM, Trans)
    linksCount = size(FKLM);
    if numel(linksCount) == 3
        linksCount = linksCount(3);
    else
        linksCount = 1;
    end

    % Compute Trans if it does not exist
    if not(exist('Trans','var'))
        Trans = eye(4);
        for i = 1:linksCount
           Trans = Trans * FKLM(:,:,i);
        end
    end
    LVJ = sym(zeros(3, linksCount)); % Linear  Velocity
    AVJ = sym(zeros(3, linksCount)); % Angular Velocity
    T  = eye(4);
    O0 = T(1:3,4);
    On = Trans(1:3,4);      % End effector's position
    for i = 1:linksCount

        Z0 = T(1:3,3);      % Current Z(i-1) Vector

        A  = FKLM(:,:,i);
        T  = T * A;         % With Respect to base frame
        if TransformationIsRevolute(A)
            LVJ(:,i) = cross(Z0, On - O0);
            AVJ(:,i) = Z0;
        else
            LVJ(:,i) = Z0;
            AVJ(:,i) = [0;0;0];
        end

        O0 = T(1:3,4);      % Next O(i-1) Position
    end

    J = simplify([LVJ;AVJ]);
end
```

8. LinksJacobian

```
% Computes the Jacobian matrices given the link
matrices from forward
% kinematcs. Designed for dynamics equations creation.

function LJ = LinksJacobian(FKLM)

    linksCount = size(FKLM);
    if numel(linksCount) == 3
        linksCount = linksCount(3);
    else
        linksCount = 1;
    end

    LJ = sym(zeros(6,linksCount,linksCount));

    for i = 1:linksCount

        LJ(:,1:i,i) = Jacobian(FKLM(:,:,1:i));

    end


end
```

9. TransformationIsRevolute

```
function revolute = TransformationIsRevolute(T)

I = eye(3);

if (T(1:3,1:3) == I)
    revolute = 0;
else
    revolute = 1;

end
```

10. GenTorques

```
% Computes the generalized torques values
%
%   EL: Euler-Lagrange dynamic equations
%    Q: Joint qi values list
%   Qd: Joint qi first  derivative values list
%  Qdd: Joint qi second derivative values list
%   Qs: syms of qi
%  Qds: syms of qi first  derivative
% Qdds: syms of qi second derivative

function T = GenTorques(EL, Q, Qs, Qd, Qds, Qdd, Qdds)

    s = size(Q);
    pointsCount = s(1);
    jointsCount = s(2);

    T = zeros(pointsCount, jointsCount);

    for i = 1:pointsCount

        T(i,:) = subs(EL, [Qs Qds Qdds], [Q(i,:)
Qd(i,:) Qdd(i,:)])';

    end

end
```

11. jmtraj

```
% Qp: positions for every joint
%  T: Time for every position
%  N: Segments count for path from point to point
% Qd: Initial velocities of joints for every point.
Final velocity should be at an extra row.

function [Q, Qd, Qdd, T] = jmtraj(Qp, Tp, N, Qdp)

    N = N + 2; % Total

    pointsSize = size(Qp);

    jointsCount = pointsSize(2);
    pointsCount = pointsSize(1);

    newPointsCount = (pointsCount-1)*N -
(pointsCount-2);

    Q   = zeros(newPointsCount, jointsCount);
    Qd  = zeros(newPointsCount, jointsCount);
    Qdd = zeros(newPointsCount, jointsCount);
    T   = zeros(newPointsCount,1);

    newRowEIdx = 1;

    for i = 1:(pointsCount-1)

       Tn = linspace(Tp(i), Tp(i+1), N) - Tp(i);

       newRowSIdx = newRowEIdx;
       newRowEIdx = newRowSIdx+N-1;
       newRowIdxs = newRowSIdx:newRowEIdx;

       [Q(newRowIdxs,:), Qd(newRowIdxs,:),
Qdd(newRowIdxs,:)] =...
           jtraj(Qp(i,:), Qp(i+1,:), Tn, Qdp(i,:),
Qdp(i+1,:));

       T(newRowIdxs) = Tp(i) + Tn;

    end

end
```

- **Other Functions**

  1. **P3DCircle**

```
% returns N points around a circle
% C: (x0,y0,r) lying on the plane
% defined by Z = f(X,Y) = a * X + b * Y + z0

function  [X Y Z] = P3DCircle(x0, y0, z0, r, a, b, N)

    Ncalc = N;
    theta = linspace(0, 2*pi, Ncalc);

    X = [];
    Y = [];
    Z = [];

    for i = 1:1:Ncalc

        x = x0 + r * cos(theta(i));
        y = y0 + r * sin(theta(i));
        z = a * x + b * y + z0;
        X = [X;x];
        Y = [Y;y];
        Z = [Z;z];

    end

end
```