Ryan Yamamoto

Summary

For my final solution, I followed the procedure outlined in the Final Project document, but added a few tweaks — joint limits, chassis and floor collision avoidance, and singularity avoidance. All maneuvers and functions are called in the "Runner.m" file. With this file, users are provided the option to change the simulation parameters such as the cube positions, robot configuration, and controller gains from the default setup. The default system parameters can be viewed by using the command "help Runner".

Once the system is defined, the script runs the "TrajectoryGenerator.m" function to generate N configurations of the robot arm based on the reference initial configuration of the robot. This maneuver defines each robot orientation to pick up and place the block. As the project description outlines, the velocity twists are then calculated, via feedforward control, between each configuration using the "FeedbackControl.m" function. In this method, proportional and integral controls are implemented to minimize error. Using the Jacobian, the twist is converted into individual joint velocities and is applied to the true robot configuration in time steps of 10 milliseconds. Function "NextState.m" applies these velocities using Euler's method to determine the next robot configuration. This process is repeated N-1 times.

My solution implements a few new elements beyond the basic project outline that is covered above. All these changes can be visualized in the "testJointLimits.m" function. In this function, the predicted next configuration is assessed for any physical violations in joint angles. Violated angles are kept unmoved by setting the corresponding Jacobian column to all zeros. This forces the joint velocity calculation to only be dependent on joints that do not violate any physical properties. There are three main ways my software addresses a violation. The first violation avoids any singularities. The system is specifically in a kinematic singular state when joints 3 and 4 are at or near zero. To minimize any drastic velocities caused by singularities, I limit the use of these joints if they are predicted to be within 0.1 radians of zero. The second violation is due to physical individual joint limits. Based on the physical robot and using scene 3, I limited each joint to its individual physical range of motion. Finally, I limited joint use when the robot arm would physically collide with the chassis or the floor. Based on the joint 1 position and the sum of joints 2 to 4, I set hard limits on the arm joints. The maneuvers created by these additions were interesting to visualize and made my maneuver solution much more suitable for real-world application.

One area of the project that was a bit unclear was the implementation of the integral control. The outline of the project fails to mention that it is necessary to add up the total integral error after each loop iteration. I needed to pass an integral error parameter to the FeedbackControl function for this to be possible, but this was never specified and was an extra input to the method. Possibly there is a way to do so without, adding an extra input parameter and I overlooked a solution. Regardless, I think this was the only major point neglected from the project description that I had some trouble with.

Results

Best Case

With a well-tuned controller, the robot was successfully able to start at an initial configuration with at least 30 degrees of orientation error and 0.2 meters of positional error. The following vectors show the initial default actual and reference configurations.

$$[\phi_{chassis} \quad x_{chassis} \quad y_{chassis} \quad \theta_1 \quad \theta_2 \quad \theta_3 \quad \theta_4 \quad \theta_5 \quad \theta_{whl,1} \quad \theta_{whl,2} \quad \theta_{whl,3} \quad \theta_{whl,4} \quad gripper]$$

Actual: $[0.5 \quad -0.5 \quad 0.5 \quad 0 \quad 0 \quad -0.5 \quad -0.3 \quad 0.6 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]$

Reference: $[0 \quad 0 \quad 0 \quad 0 \quad 0 \quad -1 \quad -0.4 \quad 0.6 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]$

For the best case, a feedforward-plus-P controller with a proportional gain of about 1.8 was implemented to correct the error without overshoot. The control reaches near zero error at about the 3 second mark before it reaches the end of the first trajectory, which takes 4 seconds. Integral controller gain was also explored but wasn't used since the system already was able to reach zero velocity error in a decent time with only proportional gain. The error plots of the robot velocity twist can be seen in Figure 1 below.

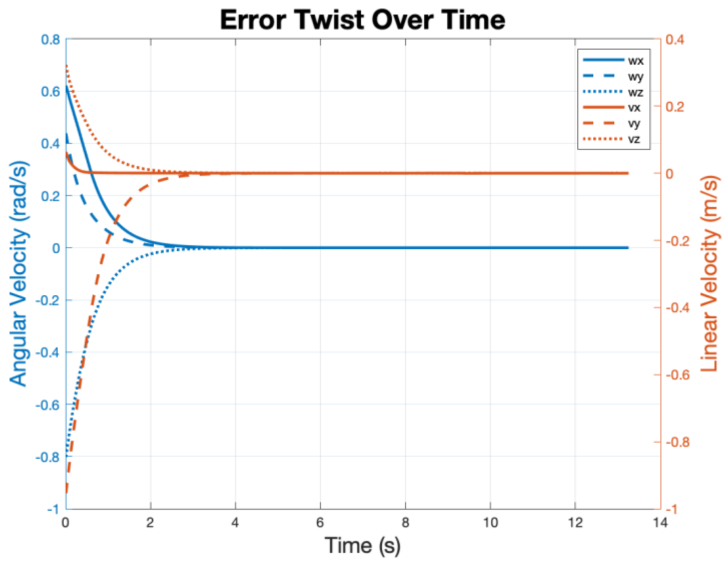

Figure 1: Error twist over time for the Best case.

A video of the best-case robot maneuver can be found at the following link. The robot is able to exhibit smooth motion and a successful pick-and-place of the cube from the default initial configuration to the end goal cube orientation.

https://youtu.be/rt6Kjc9IEaI

<u>Overshoot Case</u>

For the overshoot case, a less-well-tuned feedforward-plus-PI controller was implemented. This resulted in the velocity error showing overshoot and a bit of oscillation. The gains were set to 2.5 proportional gain and 3 integral gain. Overall, the robot over corrected itself and was smooth, but made a lot of unnecessary motion. By the end of the first trajectory, which was set to be 4 seconds long, the controller was still able to correct itself close enough to zero error. In simulation, it was still successful in completing the pick-and-place task. In Figure 2, overshoot and oscillation caused by an untuned controller can be seen in the error twist. However, error still manages to get to and remain at zero around when the first trajectory is complete.

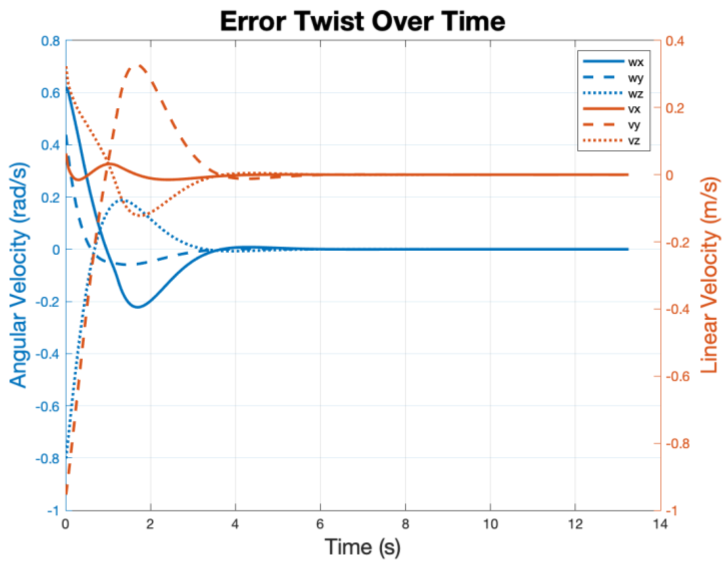

Figure 2: Error twist over time for the Overshoot case.

The video of the overshoot control and maneuvering of the youBot in CoppeliaSim can be seen in the following link. The overshoot and oscillation can be visualized in the first few seconds of the simulation.

https://youtu.be/QvUX08fSx7A

New Task Case

For the new task case, the initial and final block configurations were set to new configurations using the user interface of the Runner script which implements the "input" function in MATLAB. In this particular test case, the following cube parameters were set.

$$[x \quad y \quad \theta]$$
Initial: $[0.5 \quad -0.5 \quad \pi/4]$
Final: $[0.25 \quad 1 \quad 2\pi/3]$

For this maneuver a feedforward-P controller was implemented with the same proportional gain as the best-case scenario. This gain is set to 1.8. In the error twist plot, shown in Figure 3, the error shows no overshoot. It also shows similar attributes as the best case with the error converging to zero before the end of the first trajectory maneuver. At about the 9 second mark, the error starts to increase slightly due to the robot's natural limitations and gripping configuration. However, the feedback control is able to quickly minimize the error. The error is plotted over time and can be seen in Figure 3.

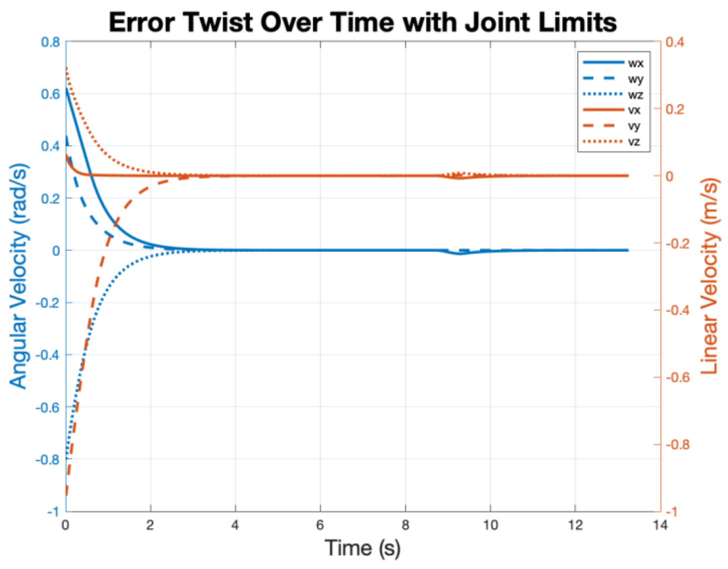

Figure 3: Error twist over time for the New Task case.

The video of this new task maneuver can be seen from the link below. In the video, the use of the input script is also shown.

https://youtu.be/PSxG4qfIoyY

<u>Joint Limitation Case</u>

To demonstrate the effectiveness of my joint limit tweaks to the final project, the robot was set to do the following maneuver with a different final cube position. Holding all other parameters the same as in the best-case scenario, the cube's final position was changed to the following.

$$[x_{final} \quad y_{final} \quad \theta_{final}] = [0.25 \quad -0.25 \quad 5\pi/6]$$

This configuration forces the robot to place the cube in a spot that it initially drives over. Without the joint limits, the robot folds over itself and has the arm passing directly through its chassis. In the real-world, this would potentially lead to damages to the robot and the cube. In the following video link, the maneuver is performed with and without joint limits for comparison. When the joint limit is in place, the robot relies more on its wheels rather than the arm to move the block to the desired final position.

https://youtu.be/SQhI7GgKeU0

In both situations, the controller gain is kept the same as the best-case scenario, where a feedforward-P controller is implemented with a gain of 1.8. Just as before, the robot is able to reach the zero error before the end of the first maneuver. In Figure 4, the two plots of the error twists for the robot with and without joint limits are shown. The results are exactly the same in terms of error.

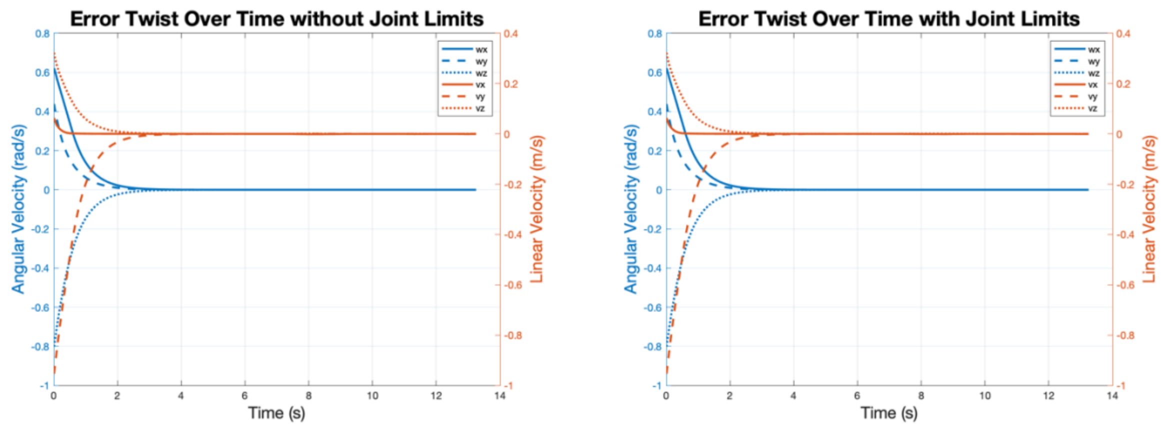

Figure 4: Error twist over time for the joint limit case.

Appendix

A1: Runner.m

```
% Ryan Yamamoto - A14478430
% Script Wrapper for MAE 204 Final Project
%
% This script utilizes all methods need to control a 5R mobile youBot.
% Running this script utilizes the NextState, TrajectoryGenerator, and
% FeedbackControl functions to move youBot to interact, pick up, and put
% down a cube in simulation through CoppeliaSim.
%
% ===============================  Default  ===============================
%
% % Set block and robot configuration parameters (x,y,theta)
% cbi    = [1  0     0];                      % cube initial configuration
% cbf    = [0 -1 -pi/2];                      % desired cube final position
% % actual youBot initial configuration
% bot    = [0.5 -0.5  0.5 ...                 % chassis (phi,x,y)
%              0    0 -0.5 -0.3  0.6 ...      % joints (J1,J2,J3,J4,J5)
%              0    0    0    0    0];      % Wheels and gripper
(W1,W2,W3,W4,gripper)
% % reference youBot initial trajectory
% botRef = [0  0  0 ...                       % chassis (phi,x,y)
%            0  0 -1 -0.4 0 ...               % joints (J1,J2,J3,J4,J5)
%            0  0  0    0 0];                 % Wheels and gripper
(W1,W2,W3,W4,gripper)
%
% % Controller gains
% Kp = 1.8; Ki = 0;
%

clear; close all; clc

disp('=================  MAE 204 Final Project Script  ==================')

%% Initialize system parameters

% Set block and robot configuration parameters (x,y,theta)
cbi    = [1  0     0];                      % cube initial configuration
cbf    = [0 -1 -pi/2];                      % desired cube final position
% actual youBot initial configuration
bot    = [0.5 -0.5  0.5 ...                 % chassis (phi,x,y)
            0    0 -0.5 -0.3  0.6 ...       % joints (J1,J2,J3,J4,J5)
            0    0    0    0    0];         % Wheels and gripper
(W1,W2,W3,W4,gripper)
% reference youBot initial trajectory
botRef = [0  0  0 ...                       % chassis (phi,x,y)
          0  0 -1 -0.4 0 ...                % joints (J1,J2,J3,J4,J5)
          0  0  0    0 0];                  % Wheels and gripper
(W1,W2,W3,W4,gripper)

% Controller gains
Kp = 1.8; Ki = 0;
```

```
% Pick-up and drop-down variables
k = 1;
a = 3/4*pi; % Grip angle when gripping cube - about y_e (radians)
standoff = 10e-2; % Standoff position above cube center (meters)
maxU = 20;


%% Get user input to change default parameters
changeDef = 1;
while changeDef
    defaultRes = upper(input('Change the default values? Y/N [N]:','s'));
    if defaultRes == 'Y'

        fprintf('\nBlank reponses will set as default. Default setup can be
seen \n')
        fprintf('through the "help Runner" command.\n\n')

        % Change initial cube configuration
        changeCbi = 1;
        while changeCbi
            cbiRes = input('Change cube initial configuration? (x,y,theta
vector):');
            if isempty(cbiRes)
                changeCbi = 0;
            elseif length(cbiRes) == 3
                cbi = cbiRes;
                changeCbi = 0;
            else
                disp('Invalid response.');
            end
        end

        % Change cube final configuration
        changeCbf = 1;
        while changeCbf
            cbfRes = input('Change cube final configuration? (x,y,theta
vector):');
            if isempty(cbfRes)
                changeCbf = 0;
            elseif length(cbfRes) == 3
                cbf = cbfRes;
                changeCbf = 0;
            else
                disp('Invalid response.');
            end
        end

        % Change youBot actual initial configuration
        changeBot = 1;
        while changeBot
            botRes = input('Change youBot actual initial configuration? (13
var vector):');
            if isempty(botRes)
                changeBot = 0;
            elseif length(botRes) == 13
                bot = botRes;
                changeBot = 0;
            else
                disp('Invalid response.');
            end
        end

        % Change youBot reference initial configuration
        changeBotRef = 1;
        while changeBotRef
            botRefRes = input('Change youBot reference initial 
configuration? (13 var vector):');
            if isempty(botRefRes)
                changeBotRef = 0;
            elseif length(botRefRes) == 13
                botRef = botRefRes;
                changeBotRef = 0;
            else
                disp('Invalid response.');
            end
        end

        % Change controller gains
        changeGain = 1;
        while changeGain
            GainRes = input('Change controller gain? (Kp,Ki) [(1.8,0)]:');
            if isempty(GainRes)
                changeGain = 0;
            elseif length(GainRes) == 2
                Kp = GainRes(1);
                Ki = GainRes(2);
                changeGain = 0;
            else
                disp('Invalid response.');
            end
        end

        changeDef = 0;
    elseif defaultRes=='N'
        changeDef = 0;
    elseif isempty(defaultRes)
        changeDef = 0;
    else
        disp('Invalid response.');
    end
end
```

```matlab
%% Define system matrices
fprintf('\n========================  Running Script
=========================')
fprintf('\nInitializing configuration space matrices...')

% Define body Jacobian from joint angles
Blist = [[0  0 1       0 0.033 0]', ...
         [0 -1 0 -0.5076     0 0]', ...
         [0 -1 0 -0.3526     0 0]', ...
         [0 -1 0 -0.2176     0 0]', ...
         [0  0 1       0     0 0]' ];

% Define robot matrices to determine end-effector initial configuration
Tb0  = [1 0 0 0.1662; 0 1 0 0; 0 0 1 0.0026; 0 0 0 1];
M0e  = [1 0 0 0.0330; 0 1 0 0; 0 0 1 0.6546; 0 0 0 1];

% Define initial arm orientation -> e-e in terms of {0}
T0ei = M0e;
for i = 1:5
    T0ei = T0ei*MatrixExp6(VecTose3(Blist(:,i)*botRef(i+3)));
end

wref = [0 0 1] * botRef(3);
pref = [botRef(1) botRef(2) 0.0963]';
Tsb  = RpToTrans(MatrixExp3(VecToso3(wref)), pref);
Tsei = Tsb*Tb0*T0ei; % Reference initial configuration Tse

% Define cube configurations
wcbi = [0 0 1] * cbi(3);
pcbi = [cbi(1) cbi(2) 2.5e-2]';
Tsci = RpToTrans(MatrixExp3(VecToso3(wcbi)),pcbi);
wcbf = [0 0 1] * cbf(3);
pcbf = [cbf(1) cbf(2) 2.5e-2]';
Tscf = RpToTrans(MatrixExp3(VecToso3(wcbf)),pcbf);

% Define end-effector configurations in {c}
Rg = MatrixExp3(VecToso3([0 1 0]'*a)); % Grip orientation rotation
Tceg = RpToTrans(Rg,[(0.043-0.025) 0 0]');
Tces = RpToTrans(Rg,[0 0 standoff]');

% Variables for base Jacobian
r = 0.0475;      % radius of all wheels
l = 0.235;       % distance from center of cart to wheel axles
w = 0.15;        % distance along axles from center to wheel
F = r/4*[-1/(l+w) 1/(l+w) 1/(l+w) -1/(l+w); 1 1 1 1; -1 1 -1 1];
F6 = [zeros(2,4); F; zeros(1,4)];

fprintf('.........done\n')

%% Generate reference trajectory
fprintf('Generating reference trajectories...')
N = TrajectoryGenerator(Tsei,Tsci,Tscf,Tceg,Tces,k);
fprintf('................done\n')
```

```
%% Loop through configurations N
fprintf('Looping through configurations...')
dt = 0.01;
count = 0;
X = bot; Xerr = []; Xint = 0;
for k=1:length(N)-1

    % Get velocity twist X,Xd,Xdn
    Xsb = RpToTrans(MatrixExp3(VecToso3([0 0
1]*X(k,1))),[X(k,2);X(k,3);0.0963]);
    T0e = M0e;
    for i = 1:5
        T0e = T0e*MatrixExp6(VecTose3(Blist(:,i)*X(k,i+3)));
    end
    Xk  = Xsb*Tb0*T0e;

    Xd  = RpToTrans(reshape(N(k,1:9),3,3)',N(k,10:12)');
    Xdn = RpToTrans(reshape(N(k+1,1:9),3,3)',N(k+1,10:12)');
    [V, Xerr(k,:), Xint] =
FeedbackControl(Xk,Xd,Xdn,Xint,Kp*eye(6),Ki*eye(6),dt);

    % Get angular wheel and joint velocities
    Je = [Adjoint(TransInv(T0e)*TransInv(Tb0))*F6
JacobianBody(Blist,X(k,4:8)')];
    uth = pinv(Je,1e-2)*V;
    U   = [uth(5:end)' uth(1:4)'];

    % Calculate next state configuration
    X(k+1,:) = [NextState(X(k,1:12),U,dt,maxU) N(k+1,13)];

    % Test joint limits and adjust
    viol = testJointLimits(X(k+1,:));
    if ~isempty(viol)
        for j = viol
            Je(:,j) = zeros(6,1);
        end
        uth = pinv(Je,1e-2)*V;
        U   = [uth(5:end)' uth(1:4)'];
        X(k+1,:) = [NextState(X(k,1:12),U,dt,maxU) N(k+1,13)];
    end

    if(k/(length(N)-1)*20)>count
        count = count+1;
        fprintf('.')
    end

end
fprintf('done\n')

%% Plot error twist Xerr
fprintf('Plotting error twist...')
fig = figure(1);
hold on; box on; grid on;
time = (1:length(Xerr)) * 0.01;
count = 1;
```

```matlab
% Plot angular velocities
yyaxis left
for i = 1:3
    plot(time,Xerr(:,i),'Linewidth',2)

    if((i/6)*30)>count
        count = count +1;
        fprintf('.....')
    end
end
ylabel('Angular Velocity (rad/s)','FontSize',16)

% Plot linear velocities
yyaxis right
for i = 4:6
    plot(time,Xerr(:,i),'Linewidth',2)

    if((i/6)*30)>count
        count = count +1;
        fprintf('.....')
    end
end
ylabel('Linear Velocity (m/s)','FontSize',16)

legend('wx','wy','wz','vx','vy','vz')
title('Error Twist Over Time with Joint Limits', 'FontSize', 20)
xlabel('Time (s)','FontSize',16)
fprintf('done\n')

%% Write Xerr and configuration X values to csv
fprintf('Writing to csv files...')
folderName = 'MAE204_Project_Results/Trial';
folderNum = 0;
while isfolder([folderName num2str(folderNum)])
    folderNum = folderNum+1;
end
folderNum = folderNum-1;
if ~isfolder([folderName num2str(folderNum)])
    mkdir([folderName num2str(folderNum)]);
end

% Write resulting configurations X
csvwrite([folderName num2str(folderNum) '/config.csv'],X);

% Write error twist file Xerr
csvwrite([folderName num2str(folderNum) '/error_twist.csv'],Xerr);

% Save figure
saveas(fig,[folderName num2str(folderNum) '/ErrorPlot.png'])

fprintf('..............................done\n')
fprintf(['\nFiles can be found in Trial ' num2str(folderNum) '!\n\n'])
fprintf('==========================  End Script 
===========================\n')
```

A2: NextState.m

```matlab
function Xf = NextState(Xi,U,dt,maxU)
%
% ===================  Next State Kinematics Simulator  ===================
%
% Param:  Xi   = current state of robots
%                -> format: 12 vars (3 chassis, 5 arms, 4 wheel angles),
%         U    = joint and wheel velocities (in radians)
%                -> format: 9 vars (5 arm, 4 wheels),
%         dt   = timestep size (in seconds),
%         maxU = maximum joint and wheel velocity magnitudes (in rad/s)
% Return: Xf   = next configuration state of robot - one time step later
%                -> format: 12 vars (3 chassis, 5 arms, 4 wheel angles)
%
% This function uses the kinematics of youBot to determine the next
% configuration - one time step later. Function assumes all input
% parameters are given in terms of meters, radians, and seconds.
%
% ===============================  Example  ===============================
%
% Input:
%
% clear all; close all; clc;
% Xi = [0 1 0 0 0 0 0 0 0 0 0 0];
% dt = 0.01;
% U = [1 1 0.5 1 0.25 10 -10 10 10];
% maxU = 12;
% Xf = NextState(Xi,U,dt,maxU)
%
% Output:
%
% Xf =
%   Columns 1 through 7
%    -0.0062    1.0024   -0.0024    0.0100    0.0100    0.0050    0.0100
%   Columns 8 through 12
%     0.0025    0.1000   -0.1000    0.1000    0.1000
%

%% Check joint and wheel velocities compared to limits
maxU = abs(maxU);
for i = 1:length(U)
    if (abs(U(i)) > maxU)
        U(i) = sign(U(i))*maxU;
    end
end

%% Seperate input parmeters by componets
q_chas = reshape(Xi(1:3),[],1);       % Chassis variables (phi,x,y)
th_arm = reshape(Xi(4:8),[],1);       % Arm position angles (radians)
th_whl = reshape(Xi(9:end),[],1);     % Wheel position angles (radians)
th_arm_dot = reshape(U(1:5),[],1);    % Arm joint velocities (rad/s)
th_whl_dot = reshape(U(6:end),[],1);  % Wheel angular velocities (rad/s)
```

```
%% Determine next step configuration via a first-order Euler step
th_arm_new = th_arm + th_arm_dot*dt;
th_whl_new = th_whl + th_whl_dot*dt;

%% Use odometry to determine new chassis configuration

% Initialize odometry variables
r = 0.0475;      % radius of all wheels
l = 0.235;       % distance from center of cart to wheel axles
w = 0.15;        % distance along axles from center to wheel

% get twist Vb = (wz,vx,vy) based on wheel velocities - Eq.13.33 in MR
F = r/4*[-1/(l+w) 1/(l+w) 1/(l+w) -1/(l+w); 1 1 1 1; -1 1 -1 1];
Vb = F*th_whl_dot*dt;
wz = Vb(1); vx = Vb(2); vy = Vb(3);

% Determine dq = (dphi_b,dx_b,dy_b) - Eq.13.35 in MR
if (Vb(1) == 0)
    dqb = Vb;
else
    dqb = [wz;(vx*sin(wz)+vy*(cos(wz)-1))/wz;(vy*sin(wz)+vx*(1-
cos(wz)))/wz];
end

% Convert dqb into {s} frame
phi = q_chas(1);
dq = [1 0 0; 0 cos(phi) -sin(phi); 0 sin(phi) cos(phi)]*dqb;

% Increment position
q_chas_new = q_chas + dq;

%% Concatenate new configuration variables
Xf = [q_chas_new' th_arm_new' th_whl_new'];

end
```

A3: TrajectoryGenerator.m

```
function N = TrajectoryGenerator(Tsei,Tsci,Tscf,Tceg,Tces,k)
%
% ========================  Trajectory Generator  =========================
%
% Param:  Tsei = initial configuration of end-effector,
%         Tsci = initial configuration of cube,
%         Tscf = Desired final configuration of cube,
%         Tceg = configuration of end-effector relative to cube while
%                grasping,
%         Tces = standoff configuration of the end-effector above the cube
%                (before and after grasping) relative to cube,
%         k    = number of trajectory reference configurations per
%                0.01 seconds (integer with a value >1)
% Return: N    = representation of configurations in time t with each
%                reference point being a transformation matrix Tse and
%                gripper state (0 open or 1 close),
%         => rep_config_ver#.csv file representing the N configurations
above
%         -> format: r11,r12,r13,r21,r22,r23,r31,r32,r33,px,py,pz,grip
%
% This function generates the trajectory motion of the youBot end-effector
% based on initial, gripping, and standoff positions of the robot-cube
% system. The maneuvers generated has the robot pick up a cube in its
% initial position and place it down at its final position. Function
% assumes all configuration values are given in meters and radians.
%
% ===============================  Example  ===============================
%
% Input:
%
% clear all; close all; clc;
% Tsei = [1 0 0 0.1992; 0 1 0 0; 0 0 1 0.7535; 0 0 0 1];
% Tsci = [1 0 0 1; 0 1 0 0; 0 0 1 0.025; 0 0 0 1];
% Tscf = [0 1 0 0; -1 0 0 -1; 0 0 1 0.025; 0 0 0 1];
% Tceg = [-0.7071 0 0.7071 0.018; 0 1 0 0; -0.7071 0 -0.7071 0; 0 0 0 1];
% Tces = [-0.7071 0 0.7071 0; 0 1 0 0; -0.7071 0 -0.7071 0.1; 0 0 0 1];
% k = 1;
% N = TrajectoryGenerator(Tsei,Tsci,Tscf,Tceg,Tces,k)
%
% Output:
%
% N = (926x13 double)
%   => with row values (r11,r12,r13,r21,r22,r23,r31,r32,r33,px,py,pz,grip)
%   => 'rep_config_ver#.csv' file also created with values of N, placed
%      in folder 'MAE204_Project_Trajectories'
%
```

```matlab
%% Initialize function variables
grip = 0;           % Gripper state (0-open / 1-closed)
poly = 5;           % Always use 5th order polynomial for smooth motion
t    = 4;           % Trajectory time in seconds
ts   = 1;           % Lift/Drop time in seconds
n    = t*k/0.01;    % Number of steps for each maneuver
ns   = ts*k/0.01;   % Number of steps for each lift/drop maneuver
N    = [];

%% 1. Move end-effector from initial to standoff (gripper open 0)
traj = CartesianTrajectory(Tsei,Tsci*Tces,t,n,poly);
for i=1:n
    [R,p] = TransToRp(traj{i});
    N = [N; [reshape(R',1,[]) p' grip]];
end

%% 2. Move gripper down to grasping position
traj = CartesianTrajectory(Tsci*Tces,Tsci*Tceg,ts,ns,poly);
for i=1:ns
    [R,p] = TransToRp(traj{i});
    N = [N; [reshape(R',1,[]) p' grip]];
end

%% 3. Close the gripper - takes 0.63s (#steps = 0.63*k/0.01)
grip = 1;
for i=1:0.63*k/0.01
    [R,p] = TransToRp(Tsci*Tceg);
    N = [N; [reshape(R',1,[]) p' grip]];
end

%% 4. Move gripper back to standoff configuration (with gripper closed 1)
traj = CartesianTrajectory(Tsci*Tceg,Tsci*Tces,ts,ns,poly);
for i=1:ns
    [R,p] = TransToRp(traj{i});
    N = [N; [reshape(R',1,[]) p' grip]];
end

%% 5. Move end-effector and cube to standoff above final position
traj = CartesianTrajectory(Tsci*Tces,Tscf*Tces,t,n,poly);
for i=1:n
    [R,p] = TransToRp(traj{i});
    N = [N; [reshape(R',1,[]) p' grip]];
end

%% 6. Move gripper down to dropping position
traj = CartesianTrajectory(Tscf*Tces,Tscf*Tceg,ts,ns,poly);
for i=1:ns
    [R,p] = TransToRp(traj{i});
    N = [N; [reshape(R',1,[]) p' grip]];
end
```

```matlab
%% 7. Open the gripper - takes 0.63s (#steps = 0.63*k/0.01)
grip = 0;
for i=1:0.63*k/0.01
    [R,p] = TransToRp(Tscf*Tceg);
    N = [N; [reshape(R',1,[]) p' grip]];
end

%% 8. Return to standoff position above final cube position
traj = CartesianTrajectory(Tscf*Tceg,Tscf*Tces,ts,ns,poly);
for i=1:ns
    [R,p] = TransToRp(traj{i});
    N = [N; [reshape(R',1,[]) p' grip]];
end

%% Create csv with values of N
folderName = 'MAE204_Project_Results/Trial';
folderNum = 0;
while isfolder([folderName num2str(folderNum)])
    folderNum = folderNum+1;
end
if ~isfolder([folderName num2str(folderNum)])
    mkdir([folderName num2str(folderNum)]);
end
csvwrite([folderName num2str(folderNum) '/rep_config.csv'],N);

end
```

A4: FeedbackControl.m

```
function varargout = FeedbackControl(X,Xd,Xdn,Xint,Kp,Ki,dt)
%
% ============================  Feedback Control  ============================
%
% Param:  X    = current actual end-effector configuration,
%         Xd   = current reference end-effector configuration,
%         Xdn  = end-effector configuration at the next step,
%         Xint = integral error over time before step dt,
%         Kp   = proportional gain matrix,
%         Ki   = integral gain matrix,
%         dt   = timestep size between reference trajectories (in seconds)
% Return: V    = commanded end-effector twist expressed in the end-effector
%                frame {e},
%         Xerr = error twist for the configuration after time step dt,
%         Xi   = total estimated integral error being summed up over time
%
% This function calculates the task-space feedforward plus feedback control
% law for a given maneuver. Function assumes all input parameters are given
% in terms of meters, radians, and seconds.
%
% ================================  Example  ================================
%
% Input:
%
% clear all; close all; clc;
% X   = [0.17 0 0.985 0.387; 0 1 0 0; -0.985 0 0.170 0.57; 0 0 0 1];
% Xd  = [0 0 1 0.5; 0 1 0 0; -1 0 0 0.5; 0 0 0 1];
% Xdn = [0 0 1 0.6; 0 1 0 0; -1 0 0 0.3; 0 0 0 1];
% [Kp,Ki] = deal(zeros(6));
% dt = 0.01;
% V = FeedbackControl(X,Xd,Xdn,Kp,Ki,dt);
%
% Output:
%
% V = [0 0 0 21.4 0 6.45]'
%

%% Calculate error twist Xerr
Xerr = se3ToVec(MatrixLog6(TransInv(X)*Xd));
%% Calculate an estimate of the integral of the error
Xi = Xint + Xerr*dt;
%% Calculate feedforward reference twist Vd
Vd = se3ToVec((1/dt)*MatrixLog6(TransInv(Xd)*Xdn));
%% Calculate end-effector twist Ve
V = Adjoint(TransInv(X)*Xd)*Vd+Kp*Xerr+Ki*Xi;

varargout{1} = V;
varargout{2} = Xerr';
varargout{3} = Xi;
end
```

A5: testJointLimits.m

```
function viol = testJointLimits(X)
%
% ========  Test Joint Limits - Avoid Collision and Singularities  ========
%
% Param:  X      = joint and wheel positions after time step dt
% Return: viol   = Vector of joints limits that are violated
%
% This function checks that each joint configuration does not result in a
% collision or singularity. Function assumes all input parameters are given
% in terms of meters, radians, and seconds.
%
% ===============================  Example  ===============================
%
% Input:
%
% X = [79 95 65 3 84 93 67 75 74 39 65 17 70];
% viol = testJointLimits(X)
%
% Output:
%
% viol = [5 6 7];
%

viol = [];

%% Define joint positions at step dt from current
j1 = X(4); j2 = X(5); j3 = X(6); j4 = X(7);

%% Define Joint limits
jlim   = 0.1;     % Singularity avoidance joint limits
j2lim  = pi*5/12; % j2 self-collision limit
j3lim  = pi*3/4;  % j3 self-collision limit
jlimcp = pi/4;    % Chassis angle max when j2 is positive
jlimcn = pi*3/4;  % Chassis angle min when j2 is negative
jc     = pi*9/16; % Chassis collision angle
jf     = pi*3/4;  % Floor collision angle

%% Avoid any possible singularities and self collision
if j2 > j2lim
    viol = [viol 6];
end

if abs(j3) < jlim || abs(j3) > j3lim
    viol = [viol 7];
end

if abs(j4) < jlim
    viol = [viol 8];
end
```

```
%% Avoid chassis and floor collision
j = [j2 j3 j4];
% Above chassis
if ((sign(j2)>0)&&(abs(j1)<jlimcp))||((sign(j2)<0)&&(abs(j1)>jlimcn))
    if (abs(sum(j)) > jc)
        indMax = find(max(j)==j,1);
        viol = [viol 5 5+indMax];
    end
else % Above floor
    if (abs(sum(j)) > jf)
        indMax = find(max(j)==j,1);
        viol = [viol 5 5+indMax];
    end
end

viol = sort(viol);
viol = unique(viol);

end
```