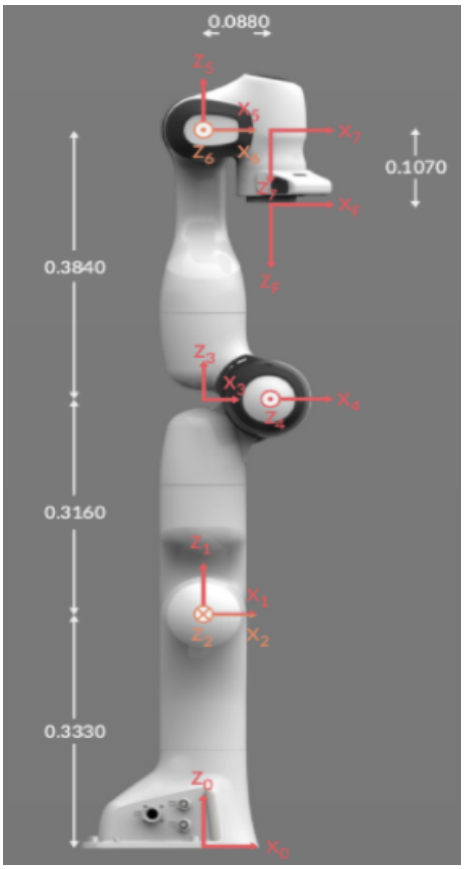
ME 397: Algorithm for Sensor-Based Robotics. THA_2 Programming Assignment

Bryant Zhou / Jonathan Salfity

April 8, 2022



In this assignment, we chose to work with the Franka-Emika Panda robot. Luckily, the Frank-Emika Panda robot is implemented in Matlab's Robotics Toolbox.

The zero or home position for the robot is chosen as in the figure above. Based on the configuration and manufacturer data sheet, the configuration of the end effector frame relative to the fixed base frame is represented as:

```
M = [1 0 0 0.088; ...
     0 -1 0 0; ...
     0 0 -1 0.926; ...
     0 0 0 1];
```

For most test cases, we compared the output values of our functions with the builtin Matlab Robotics Systems Toolbox functions. We picked specific values for the inputs and outputs that reflect corner cases and singularities.

Our code is available via https://github.com/jsalfity/Sp22_ASBR/tree/main/THA/THA2/PA.

Below is a description of our library of functions, test cases, and output we used to answer THA2 PA. The input params, output variables, and references for each function are within the function comments. Links to full functions are in the associated .zip file and/or in the github link above.

CONTENTS

Helper Functions

To account for numerical precision, we used a global epsilon value:

getGlobalEps.m

```
function eps = getGlobaleps
   eps = 1e-3;
return
```

Adjoint.m

```
function [Adjoint_T] = Adjoint(T)
%Adjoint: Calculates the Adjoint matrix
%   param:  T matrix (4x4)
%   return: Adjoint matrix (6x6)
   R = T(1:3,1:3);
   p = [T(1,4), T(2,4), T(3,4)];
   p_hat = [  0      -p(3)   p(2);...
             p(3)     0      -p(1); ...
            -p(2)     p(1)     0];
   Adjoint_T = [R, zeros(3,3);
                p_hat*R, R];
end
```

RinSO3.m

```
function output = RinSO3(R)
%RINSO3 Check R is in SO(3)
   % 3 checks
   % norm(R) = 1 (at least by some epsilon)
   % R * R' = I  (at least by some epsilon)
   % det(R) = 1  (at least by some epsilon)
   output = (norm(R) - 1)          < getGlobaleps && ...
            norm(R*R' - eye(3))    < getGlobaleps && ...
            (det(R) - 1)           < getGlobaleps ;

end
```

rotation_2_rpy.m

```
% PA 1c
function [phi, theta, psi] = rotation_2_rpy(R, theta_in)
   % param: R (3x3 rotation matrix)
   % param: theta_in (designates the input angle quadrant
   % return: phi
   % return: theta
```

```
    % return: psi
    % reference: ASBR W3L1 Pg7
    % check R is in SO(3)
    if ~RinSO3(R)
        phi = nan;
        theta = nan;
        psi = nan;
        return
    end
    if theta_in > -pi/2 && theta_in < pi/2
        phi = atan2(R(2,1), R(1,1));
        theta = atan2(-R(3,1), sqrt(R(3,2)^2 + R(3,3)^2));
        psi = atan2(R(3,2), R(3,3));
    elseif theta_in > pi/2 && theta_in < 3*pi/2
        phi = atan2(-R(2,1), -R(1,1));
        theta = atan2(-R(3,1), -sqrt(R(3,2)^2 + R(3,3)^2));
        psi = atan2(-R(3,2), -R(3,3));
    end
end
```

vector_from_skew.m

```
function [V] = vector_from_skew(M)
%vector_from_skew Summary of this function goes here
%   param: M (4x4 matrix)
%   return: V (6x1)
    V =  [M(3, 2);
          M(1, 3);
          M(2, 1);
          M(1, 4);
          M(2, 4);
          M(3, 4)];
end
```

screw_axis_2_se3.m

```
function [S] = screw_axis_2_se3(s)
%screw_axis_2_skew_symmetric Convers screw axis vector to skew symmetric
%matrix
%   param:  1x6 screw axis vector
%   return: 4x4 skew symmetrix matrix
%   reference: MR Ch4 Intro paragraph
    if length(s) ~= 6
        assert
    end
    omega = s(1:3);
    v = s(4:6);
    S = [   0          -omega(3)   omega(2)    v(1);
```

```
        omega(3)        0         -omega(1)   v(2);
        -omega(2)     omega(1)      0         v(3);
           0            0           0          0   ];
end
```

Panda Matlab Struct

We implemented a simple Matlab struct used to hold all the important kinematic information for the Panda. The data struct follows an ad-hoc convention and includes the n_joints, home position of the end effector, M, screw axes for both the space and body, and home position for each joint.

The home position of the end-effector was calculated by taking the link lengths and sequential joint frame transformations, provided by the manufacturer, and calculating the individual screw axis for use in the space/body form of exponential products.

By abstracting out these important kinematic values in a data struct, we were able to write our functions in a modular manner.

make_panda.m

```
% Panda Robot struct containing relevant kinematic info
panda.n_joints = 7;
panda.M = [1 0 0 0.088; ...
           0 -1 0 0; ...
           0 0 -1 0.926; ...
           0 0 0 1];

panda.space.screw_axes(:,:,1) = [0; 0; 1;     0;      0;      0];
panda.space.screw_axes(:,:,2) = [0; 1; 0;     -0.333; 0;      0];
panda.space.screw_axes(:,:,3) = [0; 0; 1;     0;      0;      0];
panda.space.screw_axes(:,:,4) = [0; -1; 0;    0.649; 0;      -0.088];
panda.space.screw_axes(:,:,5) = [0; 0; 1;     0;      0;      0 ];
panda.space.screw_axes(:,:,6) = [0; -1; 0;    1.033;  0;      0];
panda.space.screw_axes(:,:,7) = [0; 0; -1;    0;    0.088;   0];
% M for each joint (treat each end joint as an end effector)
panda.space.t = zeros(28,4);
panda.space.t(1:4,:) = [1 0 0 0;0 1 0 0;0 0 1 0.333;0 0 0 1];
panda.space.t(5:8,:) = [1 0 0 0;0 0 1 0;0 -1 0 0.333;0 0 0 1];
panda.space.t(9:12,:) = [1 0 0 0;0 1 0 0;0 0 1 0.649;0 0 0 1];
panda.space.t(13:16,:) = [1 0 0 0.088;0 0 -1 0;0 1 0 0.649;0 0 0 1];
panda.space.t(17:20,:) = [1 0 0 0;0 1 0 0;0 0 1 1.033;0 0 0 1];
panda.space.t(21:24,:) = [1 0 0 0;0 0 -1 0;0 1 0 1.033;0 0 0 1];
panda.space.t(25:28,:) = panda.M;
% panda.space.screw_axes_q(:,:,1) = [0; 0; 0.333];
% panda.space.screw_axes_q(:,:,2) = [0; 0; 0.333];
% panda.space.screw_axes_q(:,:,3) = [0; 0; 0.649];
% panda.space.screw_axes_q(:,:,4) = [0.088; 0; 0.649];
% panda.space.screw_axes_q(:,:,5) = [0; 0; 1.033];
% panda.space.screw_axes_q(:,:,6) = [0; 0; 1.033];
% panda.space.screw_axes_q(:,:,7) = [0.088; 0; 1.033];
panda.body.screw_axes(:,:,1) = [0; 0; -1;     0;      -0.088;      0];
panda.body.screw_axes(:,:,2) = [0; -1; 0;     0.593;    0;         0.088];
```

```
panda.body.screw_axes(:,:,3) = [0; 0; -1;     0;      -0.088;      0];
panda.body.screw_axes(:,:,4) = [0; 1; 0;    -0.277;     0;          0];
panda.body.screw_axes(:,:,5) = [0; 0; -1;     0;      -0.088;      0 ];
panda.body.screw_axes(:,:,6) = [0; 1; 0;    0.107;       0;      -0.088];
panda.body.screw_axes(:,:,7) = [0; 0;  1;    0;          0;   0];
% M for each joint (treat each end joint as an end effector)
panda.body.t = zeros(28,4);
panda.body.t(1:4,:) = [1 0 0 0;0 1 0 0;0 0 1 0.333;0 0 0 1];
panda.body.t(5:8,:) = [1 0 0 0;0 0 1 0;0 -1 0 0.333;0 0 0 1];
panda.body.t(9:12,:) = [1 0 0 0;0 1 0 0;0 0 1 0.649;0 0 0 1];
panda.body.t(13:16,:) = [1 0 0 0.088;0 0 -1 0;0 1 0 0.649;0 0 0 1];
panda.body.t(17:20,:) = [1 0 0 0;0 1 0 0;0 0 1 1.033;0 0 0 1];
panda.body.t(21:24,:) = [1 0 0 0;0 0 -1 0;0 1 0 1.033;0 0 0 1];
panda.body.t(25:28,:) = panda.M;
% panda.body.screw_axes_q(:,:,1) = [-0.088; 0; 0.593];
% panda.body.screw_axes_q(:,:,2) = [-0.088; 0; 0.593];
% panda.body.screw_axes_q(:,:,3) = [-0.088; 0; 0.277];
% panda.body.screw_axes_q(:,:,4) = [0.088; 0; 0.649];
% panda.body.screw_axes_q(:,:,5) = [-0.088; 0; 0.277];
% panda.body.screw_axes_q(:,:,6) = [-0.088; 0; 0.277];
% panda.body.screw_axes_q(:,:,7) = [nan];
```

Forward Kinematics

The **space form** for the product of exponentials formula is given as:

$$T(\theta) = e^{[S_1]\theta_1} \dots e^{[S_{n-1}]\theta_{n-1}} e^{[S_n]\theta_n} M$$

Where $M \in SE(3)$ is the end effector configuration, $S_1 .. S_n$ are the screw axes se(3) matrices, and $\theta_1 .. \theta_n$ are the joint angles.

The **body form** for the product of exponentials formula is given as:

$$T(\theta) = M e^{[B_1]\theta_1} \dots e^{[B_{n-1}]\theta_{n-1}} e^{[B_n]\theta_n}$$

Where $M \in SE(3)$ is the end effector configuration, B_i is given by $M^{-1}[S_i]M$, and $\theta_1 .. \theta_n$ are the joint angles.

To test, we passed in a few initial joint angles and compared the resulting 4x4 matrix output using Matlab's Robotics Toolbox getTransform() function. In the images below, the Matlab Robotics Toolbox visualization is on the left and our implementation is on the right.

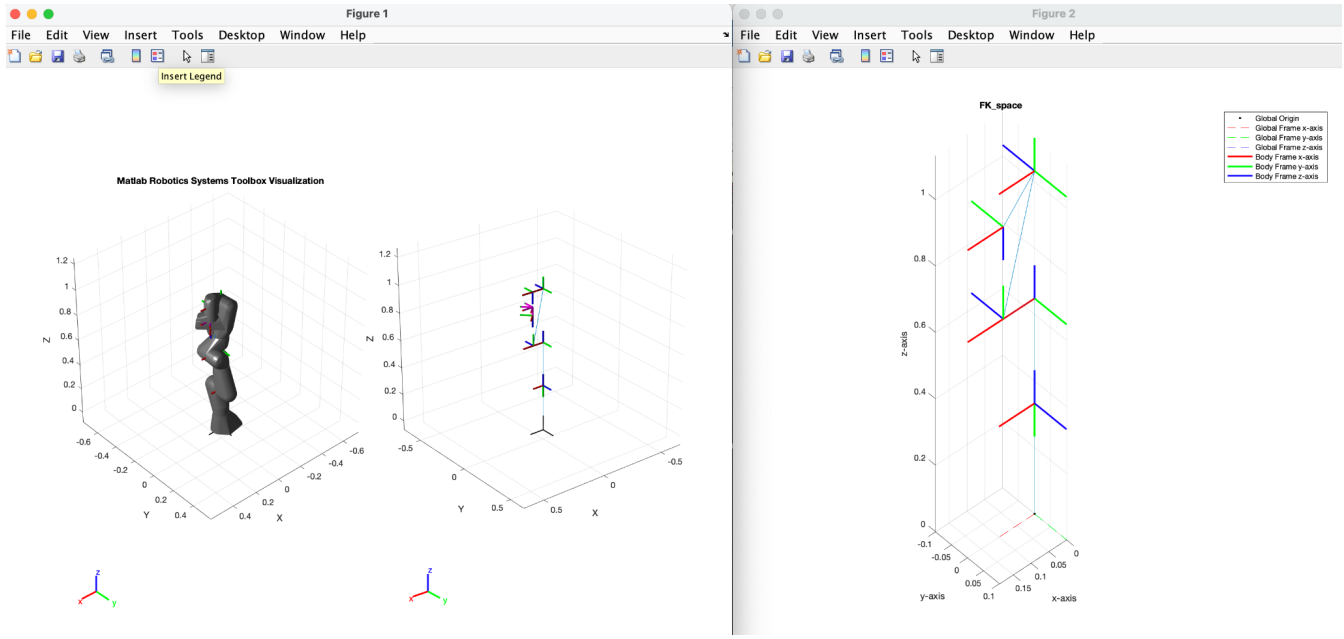
FK_space.m

```
function [T] = FK_space(robot, q, init_pose, viz)
%FK_space calculates the forward kinematics with matrix exponential
%   param: robot (struct: n_joints, M, screw_axes, screw_axes_q)
%   param: q (1xn) joints array
%   param: init_pose (initial position of the end effector)
%   param: viz (bool) visualization flag
%   return: T_final
%   reference: MR 4.1.1
```

(Code omitted in report for brevity. Code Located in associated .zip file)

Example 1:

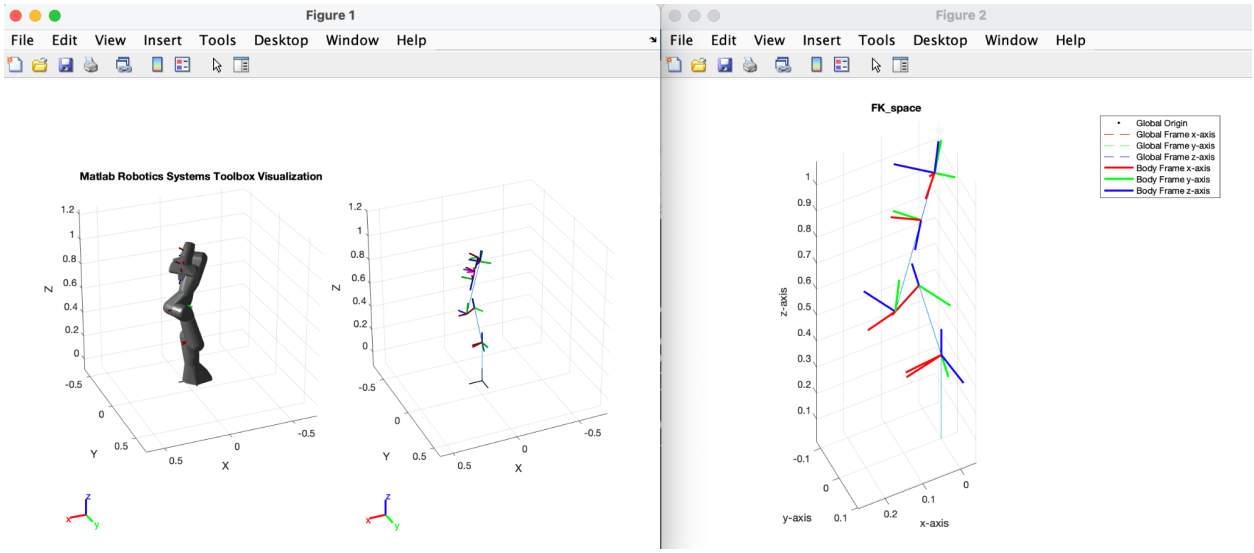
```
q = [0 0 0 0 0 0]';
```




```
T_final_s =

    1.0000         0         0    0.0880
         0   -1.0000         0         0
         0         0   -1.0000    0.9260
         0         0         0    1.0000


T_final_matlab =

    1.0000         0         0    0.0880
         0   -1.0000   -0.0000   -0.0000
         0    0.0000   -1.0000    0.9260
         0         0         0    1.0000
```

Example 2:

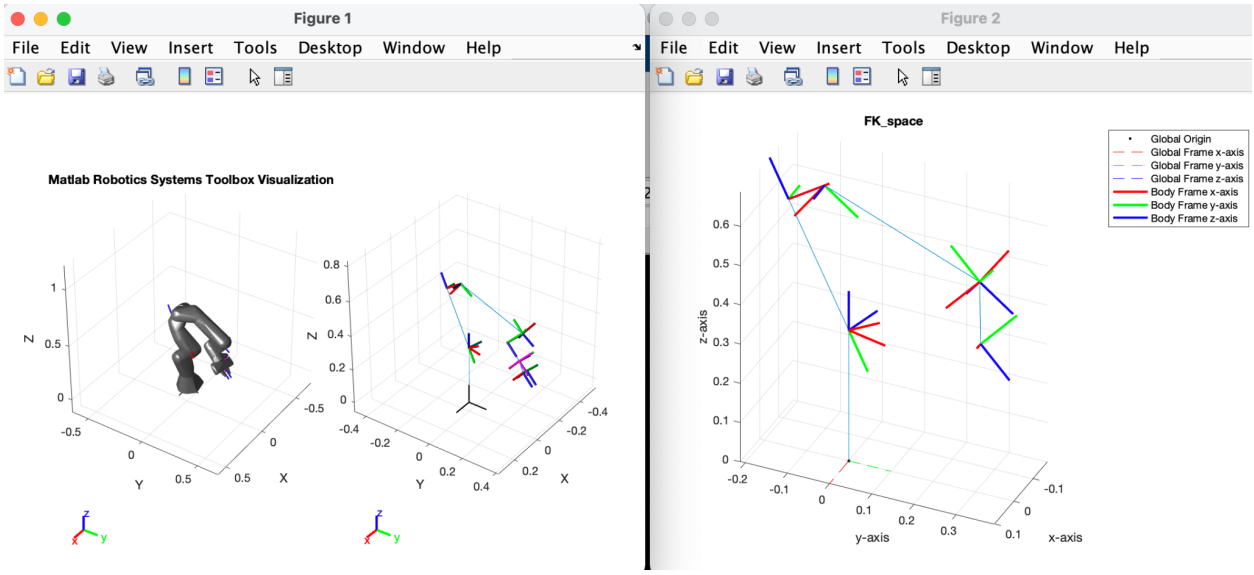
```
q = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]';
```




```
T_final_s =

    0.8110    0.3261    0.4857    0.0850
    0.0152   -0.8417    0.5397    0.0638
    0.5848   -0.4303   -0.6876    0.9730
         0         0         0    1.0000


T_final_matlab =

    0.8110    0.3261    0.4857    0.0851
    0.0152   -0.8417    0.5397    0.0637
    0.5848   -0.4303   -0.6876    0.9752
         0         0         0    1.0000
```

Example 3:

```
q = [1.33, -0.55, 0.48, -2.74, 2.35, 3.29, 1.84]';
```




```
T_final_s =

   -0.6774   -0.7219    0.1410    0.0070
   -0.4063    0.5271    0.7464    0.3157
   -0.6132    0.4483   -0.6504    0.3851
         0         0         0    1.0000


T_final_matlab =

   -0.6774   -0.7219    0.1410    0.0101
   -0.4063    0.5271    0.7464    0.3079
   -0.6132    0.4483   -0.6504    0.3783
         0         0         0    1.0000
```

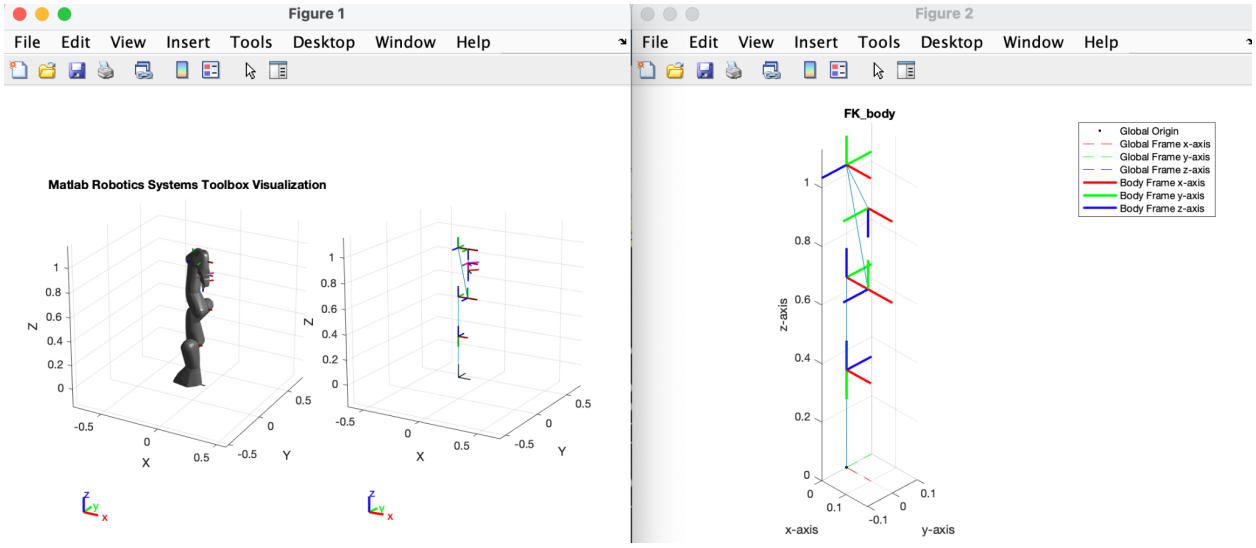
FK_body.m

```
function [T] = FK_body(robot, q, init_pose, viz)
%FK_body calculates the forward kinematics with matrix exponential in body
% frame
%   param: robot (struct with n_joints, M, screw_axes, qs)
%   param: q (1xn) joint angle array
%   param: init_pose (initial position of the end effector)
%   param: viz (bool) visualization flag
%   return: T_final
```

(Code omitted in report for brevity. Code Located in associated .zip file)

Example 1:

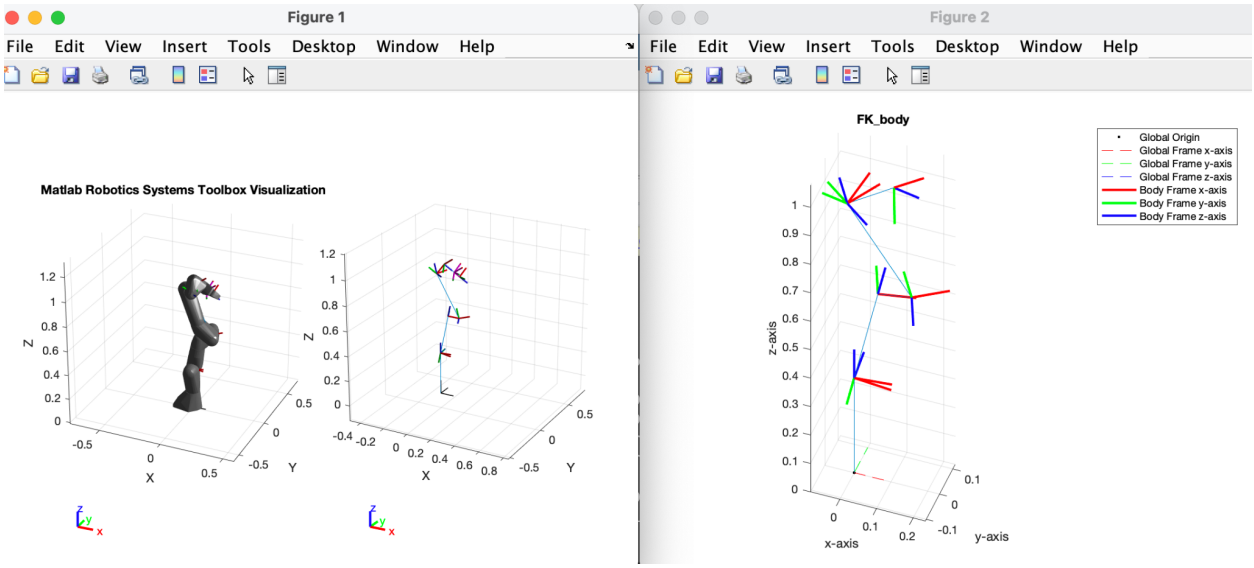
```
q = [0 0 0 0 0 0 0]';
```




```
T_final_b =

    1.0000         0         0    0.0880
         0   -1.0000         0         0
         0         0   -1.0000    0.9260
         0         0         0    1.0000


T_final_matlab =

    1.0000         0         0    0.0880
         0   -1.0000   -0.0000   -0.0000
         0    0.0000   -1.0000    0.9260
         0         0         0    1.0000
```

Example 2:

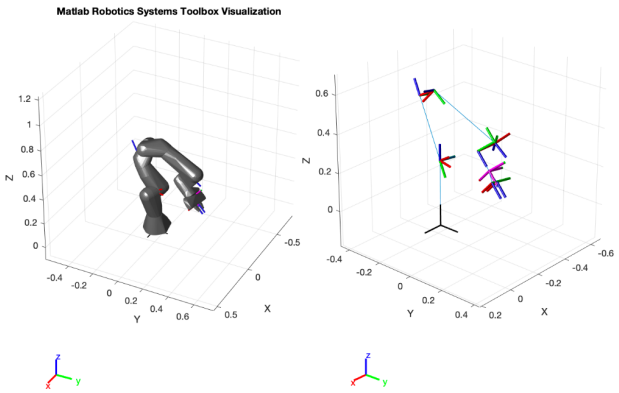
```
q = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]';
```

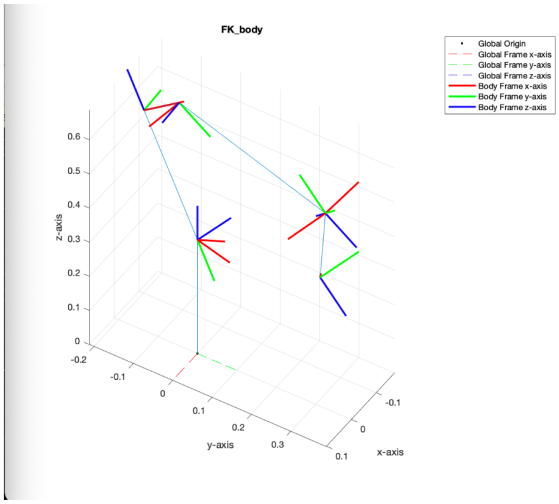



```
T_final_b =

    0.8110    0.3261    0.4857    0.0850
    0.0152   -0.8417    0.5397    0.0638
    0.5848   -0.4303   -0.6876    0.9730
         0         0         0    1.0000


T_final_matlab =

    0.8110    0.3261    0.4857    0.0851
    0.0152   -0.8417    0.5397    0.0637
    0.5848   -0.4303   -0.6876    0.9752
         0         0         0    1.0000
```

Example 3:

```
q = [1.33, -0.55, 0.48, -2.74, 2.35, 3.29, 1.84]'
```






```
T_final_b =

   -0.6774   -0.7219    0.1410    0.0070
   -0.4063    0.5271    0.7464    0.3157
   -0.6132    0.4483   -0.6504    0.3851
         0         0         0    1.0000
```

```
T_final_matlab =

   -0.6774   -0.7219    0.1410    0.0101
   -0.4063    0.5271    0.7464    0.3079
   -0.6132    0.4483   -0.6504    0.3783
         0         0         0    1.0000
```

Jacobian Analysis

The Jacobian derivation and analysis starts with relating the position of the any end-effector or frame, $x(t)$, to the joint angles, $\theta(t)$.

$$x(t) = f(\theta(t))$$

Differentiating with respect to time, the end-effector velocity, $V_{end-effector}$, can be written:

$$V_{end-effector} = \dot{x} = \frac{\partial f(\theta)}{\partial \theta}\dot{\theta} = J(\theta)\,\dot{\theta}$$

Therefore, the **Space Jacobian**, $J_s(\theta) \in R^{6xn}$**,** relates the joint rate vector, $\theta \in R^n$, to the twist V_s in the spatial frame via:

$$V_s = J_s(\theta)\,\dot{\theta}$$

The ith column of $J_s(\theta)$ is:

$$J_{si}(\theta) = Ad_{e^{[S_1]\theta_1}\ldots e^{[S_{i-1}]\theta_{i-1}}}(S_i)$$

For $i = 2, \ldots, n$ with the first column $J_{s1} = S_1$.

The **Body Jacobian,** $J_B(\theta) \in R^{6xn}$**,** relates the joint rate vector, $\theta \in R^n$, to the twist V_b in the body frame via:

$$V_b = J_b(\theta)\,\dot{\theta}$$

The ith column of $J_b(\theta)$ is:

$$J_{bi}(\theta) = Ad_{e^{-[B_n]\theta_n}\ldots e^{-[B_{i+1}]\theta_{i+1}}}(B_i)$$

For $i = n-1, \ldots 1$ with the first column $J_{bn} = S_n$.

To test our Space Jacobian and Body Jacobian calculations we used Modern Robotics Eq 5.22:

$$J_b(\theta) = [Ad_{T_{bs}}]\,J_s(\theta)$$

J_space.m

```
function [Js] = J_space(robot, q)
%J_space Summary of this function goes here
%   param: robot (struct with n_joints, M, screw_axes, qs)
%   param: q (nx1 input thetas)
%   return: Js in space form
%   reference: MR 5.1.1
   Js(:, 1) = robot.space.screw_axes(:,:,1);
   T = eye(4);
   for idx = 2: robot.n_joints
       s = robot.space.screw_axes(:,:,idx-1);
       S = screw_axis_2_se3(s);
       T = T * expm(S * q(idx-1));
       Js(:, idx) = Adjoint(T) * robot.space.screw_axes(:,:,idx);
   end
end
```

In the following examples, we compare our Jacobian calculation with the output from Matlab's geometricJacobian().

It's noted that only the first three rows of the Jacobian math with Matlab's. After consulting with Dr. Alambeigi, we concluded this makes sense, as Matlab uses the Denavit-Hartenberg (D-H) convention for link to link frame transformations and because this is a subjective calculation the velocity portion of the Jacobian, the last three rows, will be different.

Example 1:

```
q = [0 0 0 0 0 0 0]';
Js =

         0         0         0         0         0         0         0
         0    1.0000         0   -1.0000         0   -1.0000         0
    1.0000         0    1.0000         0    1.0000         0   -1.0000
         0   -0.3330         0    0.6490         0    1.0330         0
         0         0         0         0         0         0    0.0880
         0         0         0   -0.0880         0         0         0

Js_matlab =

         0         0         0         0         0         0         0
         0    1.0000         0   -1.0000         0   -1.0000   -0.0000
    1.0000    0.0000    1.0000    0.0000    1.0000    0.0000   -1.0000
    0.0000    0.5930   -0.0000   -0.2770    0.0000    0.1070   -0.0000
    0.0880    0.0000    0.0880    0.0000    0.0880    0.0000         0
         0   -0.0880         0    0.0055         0    0.0880         0
```

Here we note that `rank(Js) = 5`, which makes sense because our robot is singular at its home position.

Example 2:

```
q = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]';

Js =

         0   -0.0998    0.1977    0.3836   -0.1692    0.7719    0.4857
         0    0.9950    0.0198   -0.9216   -0.1326   -0.6340    0.5397
    1.0000         0    0.9801   -0.0587    0.9766    0.0476   -0.6876
         0   -0.3313   -0.0066    0.5746    0.0887    0.6211   -0.5690
         0   -0.0332    0.0658    0.2484   -0.1634    0.7587    0.5311
         0         0    0.0000   -0.1462   -0.0068    0.0348    0.0149


Js_matlab =

    0.0000   -0.0998    0.1977    0.3836   -0.1692    0.7719    0.4857
    0.0000    0.9950    0.0198   -0.9216   -0.1326   -0.6340    0.5397
    1.0000    0.0000    0.9801   -0.0587    0.9766    0.0476   -0.6876
   -0.0637    0.6390   -0.0497   -0.3194   -0.1027    0.0012    0.0000
    0.0851    0.0641   -0.0436   -0.1305    0.0844    0.0118   -0.0000
         0   -0.0910    0.0109   -0.0380   -0.0063    0.1380    0.0000
```

Example 3:

```
q = [1.33, -0.55, 0.48, -2.74, 2.35, 3.29, 1.84]';

Js =

         0   -0.9711   -0.1246    0.9553    0.0099   -0.4610    0.1410
         0    0.2385   -0.5076    0.1708    0.7973   -0.5319    0.7464
    1.0000         0    0.8525    0.2414   -0.6035   -0.7103   -0.6504
         0   -0.0794    0.1690   -0.1306   -0.5449    0.0619   -0.4927
         0   -0.3234   -0.0415    0.6296   -0.0467   -0.2810    0.0589
         0         0    0.0000    0.0715   -0.0706    0.1703   -0.0393


Js_matlab =

   -0.0000   -0.9711   -0.1246    0.9553    0.0099   -0.4610    0.1410
   -0.0000    0.2385   -0.5076    0.1708    0.7973   -0.5319    0.7464
    1.0000    0.0000    0.8525    0.2414   -0.6035   -0.7103   -0.6504
   -0.3079    0.0108   -0.2855   -0.1410   -0.0474    0.0813   -0.0000
    0.0101    0.0440    0.0143    0.2679   -0.0547   -0.1085   -0.0000
         0   -0.3014   -0.0332    0.3685   -0.0731    0.0284   -0.0000
```

J_body.m

```
function [Jb] = J_body(robot, q)
%J_space Summary of this function goes here
%   param: robot (struct with n_joints, M, screw_axes, qs)
%   param: q (nx1 input thetas)
%   return: Js in body form
%   reference: MR 5.1.2
   Jb(:, robot.n_joints) = robot.body.screw_axes(:,:,robot.n_joints);
   T = eye(4);
   % go in reverse
   for idx = robot.n_joints - 1: -1: 1
       b = robot.body.screw_axes(:,:,idx+1);
       B = screw_axis_2_se3(b);
       T = T * expm(-1 * B * q(idx+1));
       Jb(:, idx) = Adjoint(T) * robot.body.screw_axes(:,:,idx);
   end
end
```

We cannot use the aforementioned geometricJacobian() function to compare the Body Jacobian, because the geometric Jacobian from Matlab computes the Jacobian relative to the base, whereas the Body Jacobian computes the Jacobian relative to the end-effector frame.

To test, we calculate $[Ad_{T_{bs}}]J_s(\theta)$ and compare to $J_b(\theta)$. For each of the examples below, they indeed match up. (See `test.m` for implementation details)

Example 1:

```
q = [0 0 0 0 0 0 0]';
Jb =

         0         0         0         0         0         0         0
         0   -1.0000         0    1.0000         0    1.0000         0
   -1.0000         0   -1.0000         0   -1.0000         0    1.0000
         0    0.5930         0   -0.2770         0    0.1070         0
   -0.0880         0   -0.0880         0   -0.0880         0         0
         0    0.0880         0         0         0   -0.0880         0
```

Here we note that `rank(Jb) = 5`, which is correct because our robot is singular at its home position.

Example 2:

```
q = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]';

Jb =

    0.5848   -0.0658    0.7338    0.2627    0.4319    0.6442         0
   -0.4303   -0.8701   -0.3739    0.9261   -0.3638    0.7648         0
```

```
   -0.6876    0.4885   -0.5672   -0.2707   -0.8253         0    1.0000
   -0.0505    0.4642   -0.0347   -0.2856   -0.0857    0.0818         0
   -0.0924    0.1930    0.0154    0.0240   -0.1018   -0.0689         0
    0.0149    0.4064   -0.0550   -0.1949    0.0000   -0.0880         0
```

Example 3:

```
q = [1.33, -0.55, 0.48, -2.74, 2.35, 3.29, 1.84]';

Jb =

   -0.6132    0.5610   -0.2320   -0.8645    0.0393    0.9640         0
    0.4483    0.8268    0.2046   -0.4914    0.1425   -0.2660         0
   -0.6504    0.0410   -0.9509    0.1052    0.9890         0    1.0000
    0.2110    0.1600    0.2171   -0.2395    0.0991   -0.0285         0
    0.2316   -0.1205    0.2039    0.4074   -0.0274   -0.1031         0
   -0.0393    0.2400   -0.0091   -0.0651         0   -0.0880         0
```

Singularity Analysis

at_singularity.m

Singularity occurs when $rank(J_s) < 6$. The physical meaning implies the robot is at postures which the robot's end-effector loses the ability to move instantaneously in one or more directions.

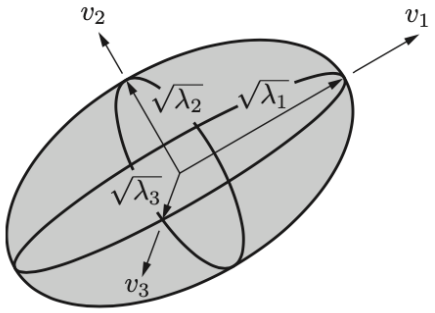
```
function [output] = at_singularity(robot, input_thetas)
%at_singularity Determine if robot at given configuration is at singularity
%   param: robot (struct with n_joints, M, screw_axes, qs)
%   param: thetas (1xn) joints array
%   return: output (bool)
   Js = J_space(robot, input_thetas);
   output = (rank(Js) < 6);
end
```

Manipulability Analysis

Manipulability is a measure of how close the robot joints are to kinematic singularity. To measure the angular and linear manipulability we use the Jacobian, partitioned in it's angular, J_ω, and linear velocity, J_v , components:

$$J(\theta) = \left[J_\omega(\theta)\ J_v(\theta)\right]^T$$

The **manipulability ellipsoid** for the **angular** and **linear** component is defined as the length of each of the three axes being an eigenvalue pointed in the associated eigenvector direction.



(Figure taken from MR)

The manipulability ellipsoid volume is exactly this measure, where the volume is proportional to

$$\sqrt{det(J\,J^T)} = \sqrt{det(A)} = \sqrt{\lambda_1\lambda_2 \ldots \lambda_m}$$

We measure the **angular manipulability ellipsoid volume** using $\sqrt{det(J_\omega(\theta)\ * J_\omega(\theta)^T)}$

We measure the **angular manipulability ellipsoid volume** using $\sqrt{det(J_v(\theta)\ * J_v(\theta)^T)}$

The **isotropy** is a measure of how easy the robot can move at a given posture, which becomes the ratio of the longest and shortest semi-axes of the manipulability ellipsoid.

$$\mu_1 = \frac{\sqrt{\lambda_{max}(A)}}{\sqrt{\lambda_{min}(A)}}\text{, where } A = J\,J^T$$

Similarly, the **condition number** is calculated as:

$$\mu_2 = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}\text{, where } A = J\,J^T$$

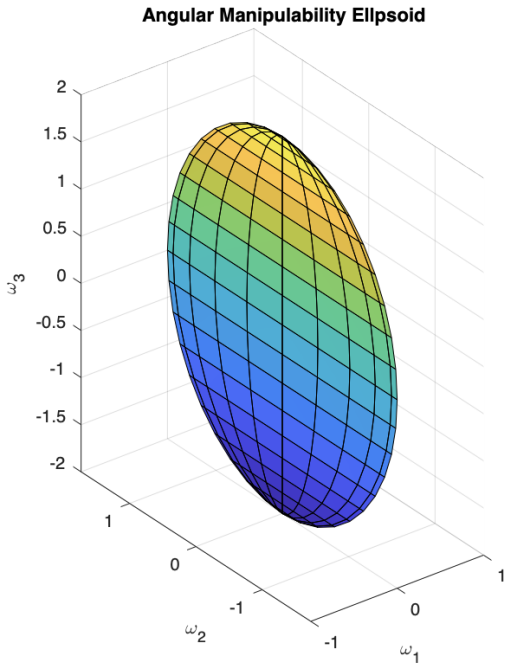
ellipsoid_plot_angular.m

```
function [isotropy, condition, volume] = ellipsoid_plot_angular(robot,
input_thetas)
%ellipsoid_plot_angular
%   param: robot (struct with n_joints, M, screw_axes, qs)
%   param: thetas (1xn) joints array
%   return: isotropy, condition, volume (floats)
%   reference: MR 5.4
end
```

(Code omitted in report for brevity. Code Located in associated .zip file)

Example 1:

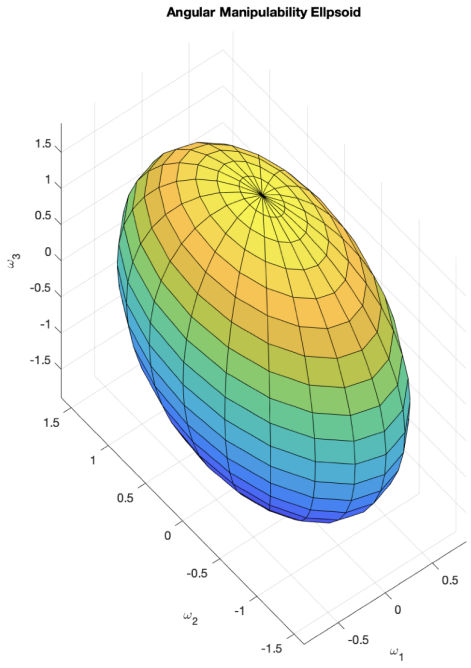
```
q = [0 0 0 0 0 0 0]';
volume = 0
isotropy = Inf
condition = Inf
```



This example is correct as it has **zero volume** given the robot is at an angular velocity singularity.

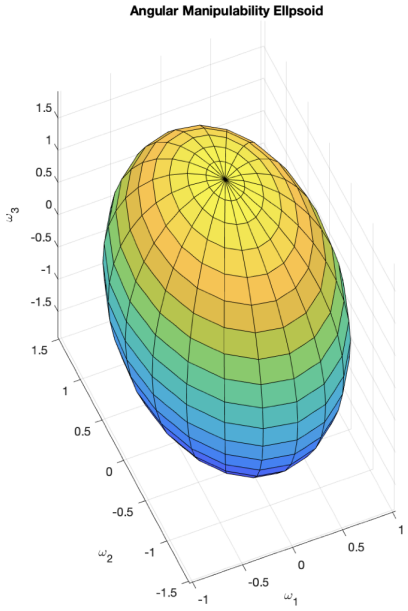
Example 2:

```
q = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]';
volume = 2.6611
isotropy = 2.2056
condition = 4.8648
```



Example 3:

```
q = [1.33, -0.55, 0.48, -2.74, 2.35, 3.29, 1.84]';
volume = 2.9829
isotropy = 1.8488
condition = 3.4180
```



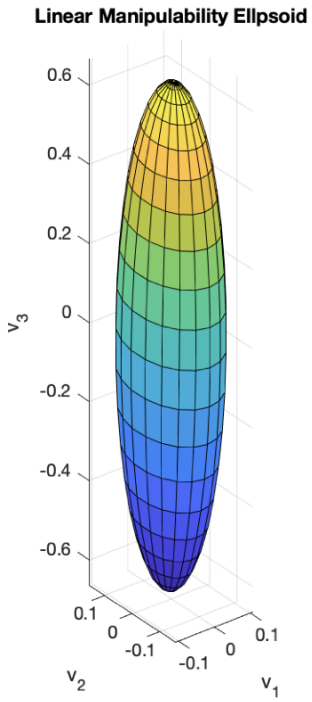
ellipsoid_plot_linear.m

```
function [isotropy, condition, volume] = ellipsoid_plot_linear(robot, q,
verbose)
%ellipsoid_plot_linear
%   param: robot (struct with n_joints, M, screw_axes, qs)
%   param: q (1xn) joints array
%   param: verbose (bool) to report the ellipsoid metrics
%   return: isotropy, condition, volume (floats)
%   reference: MR 5.4
```

(Code omitted in report for brevity. Code Located in associated .zip file)

Example 1:

```
q = [0 0 0 0 0 0 0]';
volume = 0.0108
isotropy = 6.292
condition = 39.579
```

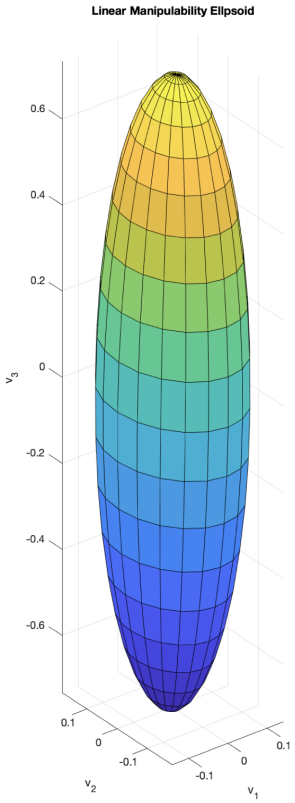


This example is especially interesting, because while the angular velocity component is at a singularity, the linear velocity component is not. However, it is close, as the volume is quite small.

Example 2:

```
q = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]';
volume = 0.018
isotropy = 5.193
```

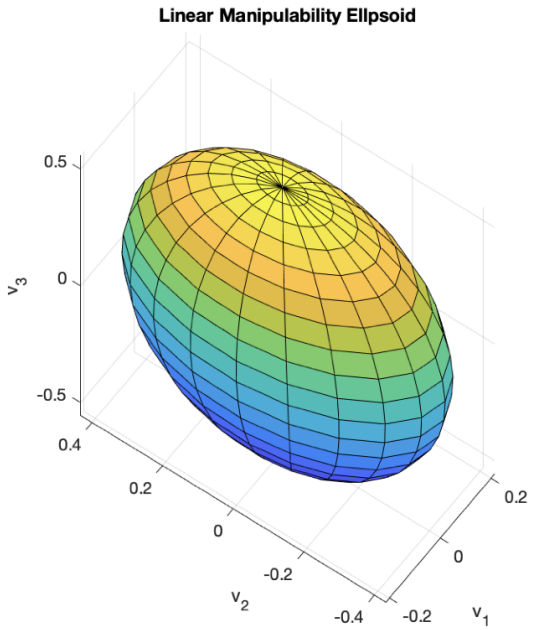
```
condition = 26.967
```



Similarly, this example is especially interesting, because the volume is quite small.

Example 3:

```
q = [1.33, -0.55, 0.48, -2.74, 2.35, 3.29, 1.84]';
volume = 0.0515
isotropy = 2.6127
condition = 6.686
```



Similarly, this example is especially interesting, because the volume is quite small.

J_ellipsoid_volume.m

```
function [volume] = J_ellipsoid_volume(A)
%J_ellispoid_volume Calculates the condition number, a single scalar measure,
% defining the sensitivity of the matrix A to scaling errors
%   param: A (3x3 matrix, usually calculated like:
%                         Js = J_space(robot, input_thetas);
%                         A = Js(1:3,:)*Js(1:3,:)';
%   return: volume (float) volume
%   reference: MR 5.4
   volume = sqrt(det(A));
end
```

J_isotropy.m

```
function [mu1] = J_isotropy(A)
%J_isotropy Calculates the isotropy, a single scalar measure, defining how
%easily the robot can move at a given posture
%   param: A (3x3 matrix, usually calculated like:
%                         Js = J_space(robot, input_thetas);
%                         A = Js(1:3,:)*Js(1:3,:)';
%   return: mu1 (float) isotropy measure
%   reference: MR 5.4
   % calculate e-vecs and e-vals
   evals = eig(A);
   eval_min = min(evals);
   eval_max = max(evals);
   mu1 = sqrt(eval_max / eval_min);
end
```

J_condition.m

```
function [mu2] = J_condition(A)
%J_condition Calculates the condition number, a single scalar measure,
% defining the sensitivity of the matrix A to scaling errors
%   param: A (3x3 matrix, usually calculated like:
%                         Js = J_space(robot, input_thetas);
%                         A = Js(1:3,:)*Js(1:3,:)';
%   return: mu2 (float) condition measure
%   reference: MR 5.4
   % calculate e-vecs and e-vals
   evals = eig(A);
   eval_min = min(evals);
   eval_max = max(evals);
   mu2 = eval_max / eval_min;
end
```

Inverse Kinematics

Inverse kinematics returns the joint angles that map the robot to the desired end-effector configuration.

To **iteratively solve** for the correct inverse kinematics, suppose we want a desired configuration, x_d, and the robot is currently at a configuration $f(x_d)$.

We can define the difference between the two as $g(x_d)$ and of course we want the difference to be zero.

$$g(x_d) = x_d - f(\theta_d) = 0$$
$$\Rightarrow x_d = f(\theta_d)$$

Performing a Taylor Expansion around θ_d:

$$f(\theta_d) = f(\theta^0) + \frac{\partial f}{\partial \theta}|_{\theta^0} (\theta_d - \theta^0)$$

We recognize the Jacobian, $\frac{\partial f}{\partial \theta}|_{\theta^0} = J(\theta^0)$ and $(\theta_d - \theta^0) = \Delta\theta$

$$\Rightarrow f(\theta_d) = f(\theta^0) + J(\theta^0)\Delta\theta = x_d$$

Therefore the update rule in any iterative inverse kinematics algorithm becomes:

$$\Delta\theta = J(\theta^i)^{-1}(x_d - f(\theta^i))$$

There are quite a few different methods to calculate the inverse of the Jacobian, which will be addressed in the three different inverse kinematics functions below.

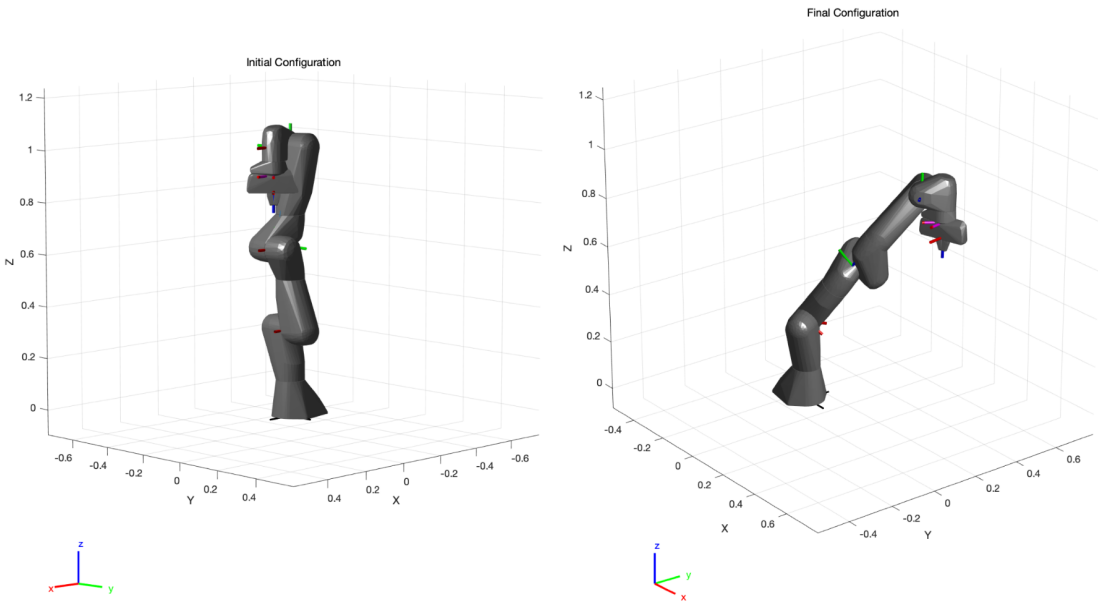
J_inverse_kinematics.m

This first algorithm is perhaps the simplest. To calculate the inverse of the Jacobian, we use the `pinv()` function from matlab and continue updating until the difference between the

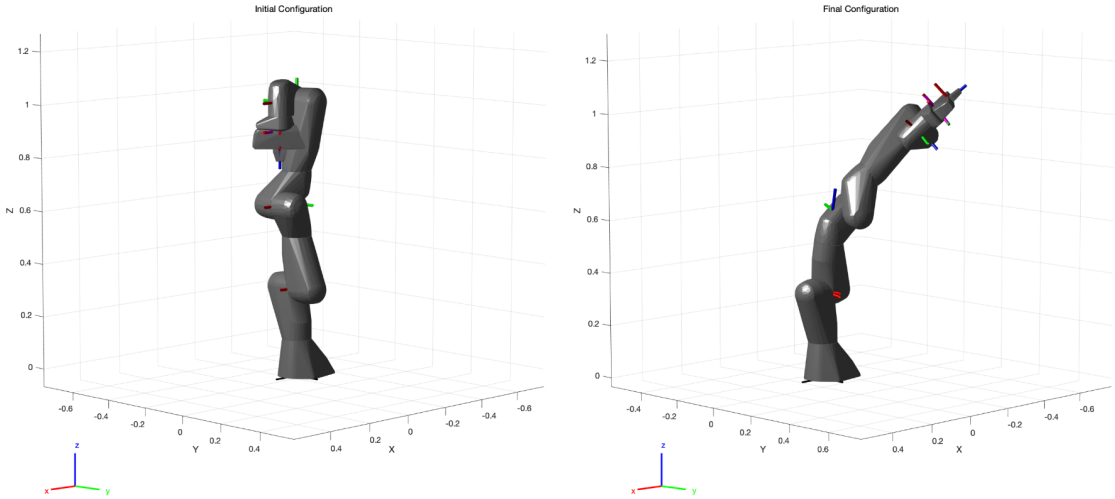
```
function [q, idx, e] = J_inverse_kinematics(robot, Ti, Tf, q0, max_iterations)
%J_inverse_kinematics Summary of this function goes here
%   param: robot (struct with n_joints, M, screw_axes, qs)
%   param: Ti    (Initial Transformation)
%   param: Tf    (Final Transformation)
%   param: q0    (Initial guess of the joint angle)
%   reference: MR 6.2.2
(Code omitted in report for brevity. Code Located in associated .zip file)
```

To test the inverse kinematics, we relied on Matlab's Robotics Systems Toolbox. We generate a random configuration using `randomConfiguration`, then recover the associated final configuration transformation with `getTransform` to know the correct final configuration. We use the randomly generated desired final configuration, **`Tf_matlab`**, as the desired final configuration. The solved joint angles are returned, **`q_inverse`**, then passed through our **`FK_space`** function to find the associated **`Tf`**. Finally, we **`assert`** the difference between **`Tf_matlab`** and **`Tf`**, as they should be equivalent.

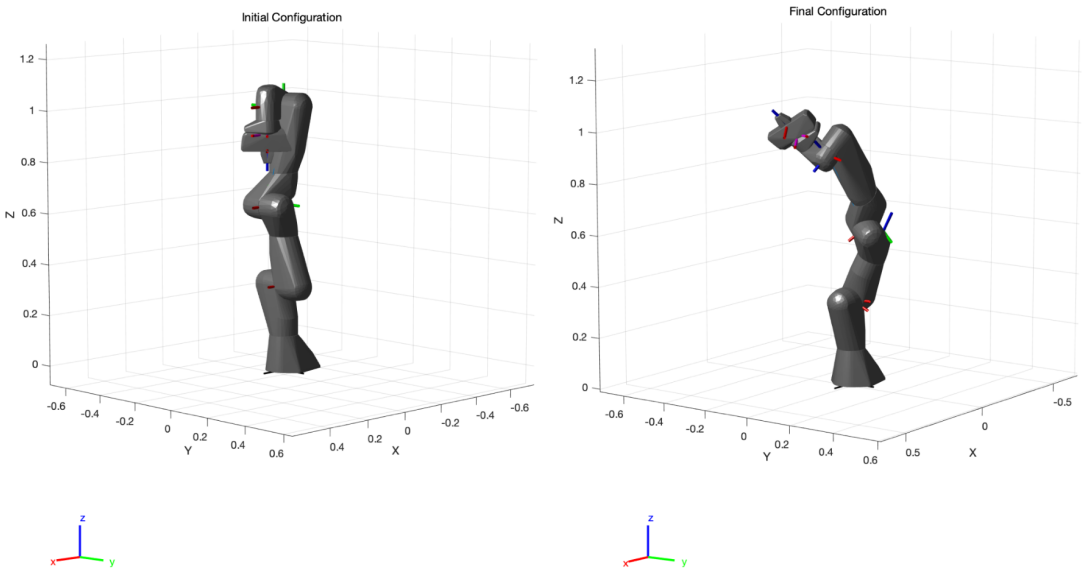
```
for n_test = 1:5
    % always at home position
    q0 = [0 0 0 0 0 0 0]';
    Ti = [1 0 0 0.088;
      0 -1 0 0;
      0 0 -1 0.926;
      0 0 0 1];
    qf = randomConfiguration(panda_matlab);
    Tf_matlab = getTransform(panda_matlab, qf, 'panda_link8');
    [q_inverse, idx_inverse, e_inverse] = J_inverse_kinematics(panda, Ti, ...
                                                               Tf_matlab, ...
                                                               q0, 200);
    % q_inverse will not always be qf
    % but Tf == Tf_matlab
    Tf_inverse = FK_space(panda, q_inverse, Ti, 0);
    assert(norm(Tf_matlab - Tf_inerse) < eps)

    figure
    axis equal
    subplot(121)
    show(panda_matlab, [q0;0;0])
    title('Initial Configuration')
    subplot(122)
    show(panda_matlab, [q_inverse;0;0])
    title('Final Configuration')
end
```




```
Tf =

   -0.5031   -0.8639    0.0226    0.3963
   -0.8635    0.5014   -0.0541    0.3547
    0.0354   -0.0467   -0.9983    0.7805
         0         0         0    1.0000


Tf_matlab =

   -0.5031   -0.8639    0.0228    0.3964
   -0.8635    0.5014   -0.0543    0.3548
    0.0355   -0.0470   -0.9983    0.7805
         0         0         0    1.0000
```




```
Tf =

    0.8651    0.1272   -0.4852   -0.2857
    0.1928    0.8086    0.5558    0.3262
    0.4631   -0.5744    0.6750    1.0241
         0         0         0    1.0000

Tf_matlab =

    0.8651    0.1272   -0.4852   -0.2857
    0.1928    0.8086    0.5558    0.3262
    0.4631   -0.5744    0.6750    1.0241
         0         0         0    1.0000
```




```
Tf =

    0.4673   -0.6111    0.6390    0.3274
    0.8838    0.3046   -0.3550   -0.0096
    0.0223    0.7306    0.6824    1.0220
         0         0         0    1.0000


Tf_matlab =

    0.4672   -0.6112    0.6389    0.3273
    0.8839    0.3046   -0.3550   -0.0095
    0.0223    0.7306    0.6825    1.0220
         0         0         0    1.0000
```

J_transpose_kinematics.m

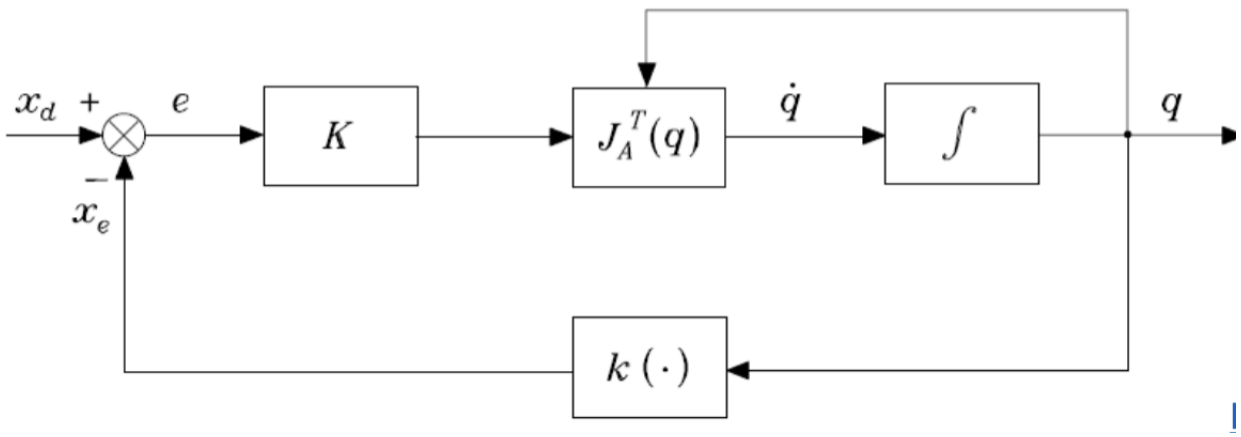
This is the second way of calculating the inverse kinematics that is based on the concept of Lyapunov stability and the transpose of the Jacobian matrix. A Lyapunov function *V* requires that $V(e) > 0, \forall e \neq 0$ and $V(0) = 0$. If the first-order derivative of the Lyapunov function is negative definite, the theorem guarantees an asymptotic stability of the system. Note that there are multiple Lyapunov function candidates that satisfy the above requirements. Here, we chose a possible candidate as follows:

$V(e) = \frac{1}{2} e^T K e,$

where *e* is the error between the current pose and the target pose, and *K* is a positive definite matrix to guarantee that *V* is positive definite. After carefully choosing the joint velocity, the derivative of the Lyapunov function is negative definite as shown below:

$\dot{V}(e) = e^T K \dot{x}_d - e^T K J_A(q) J_A(q)^T K e.$

We assumed that the target velocity is zero to guarantee that the above equation is negative definite. Together with the Lyapunov stability theory, the pose error is guaranteed to asymptotically converge to zero. The block diagram below (obtained from the lecture slides) shows the algorithm formulation.



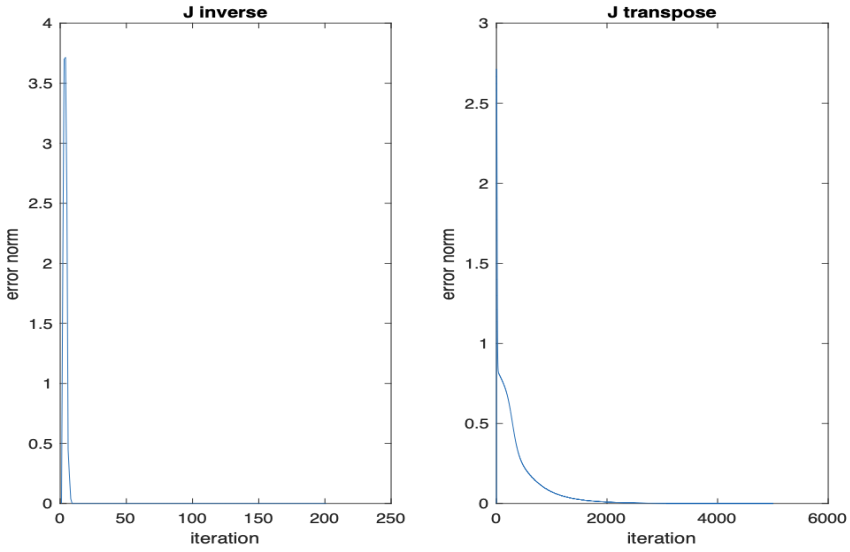
Note that the logic of finding the target joint angle is similar to the previous function that uses the inverse of Jacobian. The only difference here is that the transpose of the Jacobian matrix is used (refer to the block diagram above). Descriptions for the function input and output are shown here:

```
function [q, idx, e] = J_transpose_kinematics(robot, Ti, Tf, q0, alpha,
max_iterations)
%J_inverse_kinematics Summary of this function goes here
%   param: robot (struct with n_joints, M, screw_axes, qs)
%   param: Ti    (Initial Transformation)
%   param: Tf    (Final Transformation)
%   param: q0    (Initial guess of the joint angle)
%   alpha: gain
%   reference: Lecture notes 10-L1
```

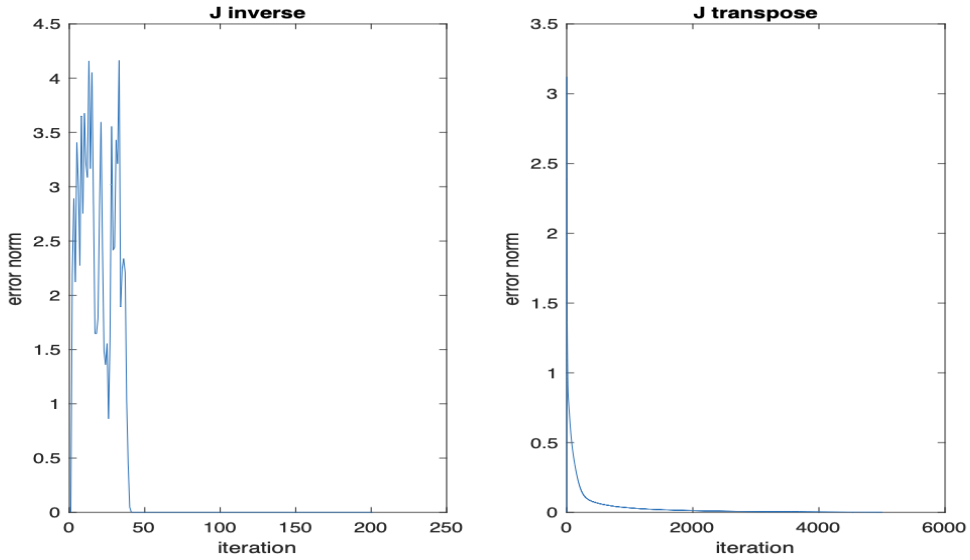
(Code omitted in report for brevity. Code Located in associated .zip file)

Note that we use the exact same test as described in `J_inverse_kinematics`, and we placed an “assert” function in our test file to “error check” the targeted pose and the pose obtained from the `FK_space` function using the joint angle from this “Jacobian transpose” method. Thus, instead of graphically showing robot poses as above, we compare the Jacobian inverse and Jacobian transpose methods in terms of their convergence speed. The convergence thresholds for both methods are 0.001. The results are shown below for each of the test cases.

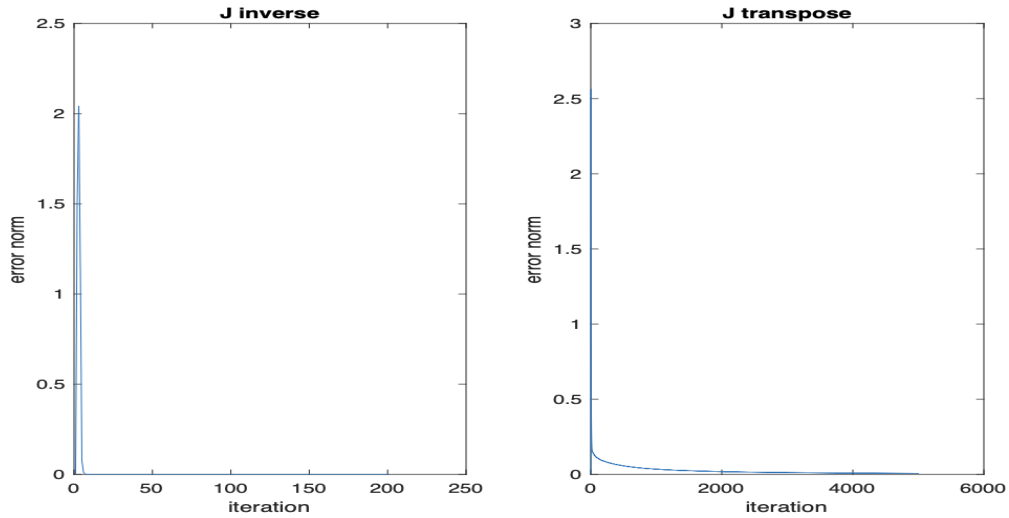
Test 1:



Test 2:



Test 3:



It is obvious to see that the convergence speed of the “Jacobian transpose” method is significantly slower than the “Jacobian inverse” method. This is expected, since the Lyapunov method only guarantees an asymptotic convergence.

redudancy_resolution.m

The final inverse kinematics algorithm we wrote incorporates an optimization problem to solve for the set of joint angles which minimizes an objective function. In this case, we set our objective function to **increase the manipulability measure.**

The optimization problem:

$$min\ g(\dot{q}) = \frac{1}{2}\dot{q}\,W\,\dot{q}$$
$$s.t \quad v_e = J\,\dot{q}$$

The matrix **W** is the weighting matrix for different joint angles.

Solving this problem through the method of Lagrange multipliers, the first order conditions require the requested solution

$$\dot{q} = W^{-1}J^T\lambda$$

For redundant manipulators and when W is the identity matrix, the general solution becomes:

$$\dot{q} = \dot{q}\,^* + P\dot{q}_0 = J^{\dagger}v_e + (I_n - J^{\dagger}J)\dot{q}_0$$

Where

$$\dot{q} = \kappa_0\left(\frac{\partial w(q)}{\partial q}\right)^T$$

To move away from kinematic singularities, **our objective function is the manipulability measure**:

$$w(q) = \sqrt{(det(J(q)\,J(q)^T}$$

To numerically take the derivative, $\frac{\partial w(q)}{\partial q}$, we computed the difference between $w(q)$ and q in the current iteration and the previous iteration.

```
function [q, idx, e] = redundancy_resolution(robot, Ti, Tf, q0,
max_iterations, K)
%redundancy_resolution Summary of this function goes here
%   param: robot (struct with n_joints, M, screw_axes, qs)
%   param: Ti    (Initial Transformation)
%   param: Tf    (Final Transformation)
%   param: q0    (Initial joint angle guess)
%   param: K     (Weighting matrix for manipulabilty)
%   return: q (final joint angles)
%   return: idx (n_iterations completed)
%   return: e (error array)
```

```
%   reference: ASBR W10L1
```
(Code omitted in report for brevity. Code Located in associated .zip file)

Note that we use the exact same test as described in `J_inverse_kinematics` and `J_transpose_kinematics`, and we placed an `assert` function in our test file to “error check” the targeted pose and the pose obtained from the `FK_space` function using the joint angle from this `redudancy_resolution` method.

References

Prof Farshid Alambeigi, **UT Austin ME 397: Algorithms for Sensor Based Robotics.** Spring 2022

Lynch and Park, “**Modern Robotics**”, Cambridge U. Press, 2017

Matlab Robotics Toolbox

Appendix

Test.m

```
clc; clear all; close all;
set(0, 'DefaultFigureColor', 'w')
eps = 0.1; % I have to make this larger, otherwise the FK test will fail
% % Panda Robot struct containing relevant kinematic info
run('make_panda.m') % panda robot struct
panda_matlab = loadrobot('frankaEmikaPanda', 'DataFormat','column');
% make 1x9 to make compatible with panda_matlab.
% note that our functions only use 1x7
% q = [0 0 0 0 0 0 0 0 0]';
% q = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0, 0]';
q = [1.33, -0.55, 0.48, -2.74, 2.35, 3.29, 1.84, 0.01, 0.02]';
% % singularity test
% display(at_singularity(panda, q))
% plot panda_matlab robot
figure
axis equal
subplot(121)
show(panda_matlab, q)
title('Matlab Robotics Systems Toolbox Visualization')
subplot(122)
show(panda_matlab, q, 'Visuals','off')
% FK_space.m
T_final_s = FK_space(panda, q,  panda.M, 1)
T_final_matlab = getTransform(panda_matlab, q, 'panda_link8')
assert(norm(T_final_matlab - T_final_s) < eps)
% FK_body.m
T_final_b = FK_body(panda, q,  panda.M, 1)
T_final_matlab = getTransform(panda_matlab, q, 'panda_link8')
assert(norm(T_final_matlab - T_final_b) < eps)
% J_space.m
Js = J_space(panda, q)
Js_matlab = geometricJacobian(panda_matlab, q, 'panda_link8')
% J_body.m
Jb = J_body(panda, q);
% Check the correctness of Jacobian calculation using Adjoint
assert(norm(Adjoint(T_final_s)*Jb - Js) < getGlobaleps);
% plot angular manipulability ellipsoid
ellipsoid_plot_angular(panda, q, 1)
% plot linear manipulabilty ellipsoid
ellipsoid_plot_linear(panda, q, 1)
% J_inverse_kinematics
for n_test = 1:5
   % always at home position
   q0 = [0 0 0 0 0 0 0]';
   Ti = [1 0 0 0.088;
```

```
    0 -1 0 0;
    0 0 -1 0.926;
    0 0 0 1];
  qf = randomConfiguration(panda_matlab);
  Tf_matlab = getTransform(panda_matlab, qf, 'panda_link8');
  % J_inverse_kinematics
  [q_inverse, idx_inverse, e_inverse] = J_inverse_kinematics(panda, Ti, ...
                                                             Tf_matlab, ...
                                                             q0, 200);
  % q_inverse will not always be qf
  % but Tf should be Tf_matlab
  Tf_inverse = FK_space(panda, q_inverse, Ti, 0);
  assert(norm(Tf_matlab - Tf_inverse) < eps)
  figure
  axis equal
  subplot(121)
  show(panda_matlab, [q0;0;0])
  subtitle('Initial Configuration')
  subplot(122)
  show(panda_matlab, [q_inverse;0;0])
  subtitle('Final Configuration')
  % J_transpose_kinematics
  % NOTE!! The Lyapunov method using J_transpose takes significantly long
  % iterations to converge than the J_inverse method under this initial and
  % final confiugrations. It makes sense since the Lyapunov stability
  % condition only guarantees an asymptotic stability. In order to decrease
  % the number of iterations, we can also compare the difference of two
  % consecutive errors and set a threshold. However, for a fair comparison,
  % we might want to keep this consistent with the J_inverse method, and we
  % can mention this in our report.
  [q_transpose, idx_transpose, e_transpose] = J_transpose_kinematics(panda,
...
                                                                     Ti,
Tf_matlab, ...
                                                                     q0, 0.1,
5000);
  % q_inverse will not always be qf
  % but Tf_transpose should be Tf_matlab
  Tf_transpose = FK_space(panda, q_transpose, Ti, 0);
  assert(norm(Tf_matlab - Tf_transpose) < eps)
  % Convergence plot for a more explicit comparison between J transpose and J
  % inverse
  figure
  subplot(1,2,1)
  plot(e_inverse)
  xlabel('iteration')
  ylabel('error norm')
  title('J inverse')
  subplot(1,2,2)
```

```
    plot(e_transpose)
    xlabel('iteration')
    ylabel('error norm')
    title('J transpose')

    % REDUDANCY RESOLUTION
    [q_rr, idx_rr, e_rr] = redundancy_resolution(panda, Ti, Tf_matlab, ...
                                                 q0, 200, eye(7));
    % q_inverse will not always be qf
    % but Tf should be Tf_matlab
    Tf_rr = FK_space(panda, q_rr, Ti, 0);
    assert(norm(Tf_matlab - Tf_rr) < eps)
end
```

Screw Axis Calculations

Based on the configuration shown in Fig. 1, the configuration of the end effector frame relative to the fixed base frame is represented as:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0.088 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0.333 + 0.316 + 0.384 - 0.107 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0.088 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0.926 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The angular and linear velocities of each joint is shown in the following Table.

Table I. Angular and linear velocities of each joint in the space frame

i	w_i	q_i	v_i
1	(0,0, 1)	(0,0, 0.333)	(0,0, 0)
2	(0,1, 0)	(0,0, 0.333)	(− 0.333, 0,0)
3	(0,0, 1)	(0, 0,0 . 649)	(0,0, 0)
4	(0, − 1,0)	(0.088, 0,0 . 649)	(0.649, 0, − 0.088)
5	(0,0, 1)	(0,0, 1.033)	(0,0, 0)
6	(0, − 1,0)	(0,0, 1.033)	(1.033, 0,0)
7	(0,0, − 1)	(0.088, 0,1 . 033)	(0,0 . 088,0)

Screw axis of each joint is shown below.

$$s_1 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$s_2 = \begin{bmatrix} 0 & 0 & 1 & -0.333 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$s_3 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$s_4 = \begin{bmatrix} 0 & 0 & -1 & 0.649 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -0.088 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$s_5 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$s_6 = \begin{bmatrix} 0 & 0 & -1 & 1.033 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$s_7 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0.088 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The space form Jacobian has the following form

$$J_s(\theta) = \begin{bmatrix} J_{s1} & J_{s2}(\theta) & \dots & J_{sn}(\theta) \end{bmatrix},$$

where

$$J_{s1} = S_1,$$

$$J_{s2}(\theta) = Ad_{e^{[S_1]\theta_1}}(S_2),$$

$$J_{s3}(\theta) = Ad_{e^{[S_1]\theta_1 e^{[S_2]\theta_2}}}(S_3).$$

From Table I, we have the following twist vectors:

$$S_1 = [0,0,1,0,0,0]^T,$$

$$S_2 = [0,1,0,-0.333,0,0]^T,$$

$$S_3 = [0,0,1,0,0,0]^T,$$

$$S_4 = [0,-1,0,0.649,0,-0.088]^T,$$

$$S_5 = [0,0,1,0,0,0]^T,$$

$$S_6 = [0,-1,0,1.033,0,0]^T,$$

$$S_7 = [0,0,-1,0,0.088,0]^T.$$

Table II. Angular and linear velocities of each joint in the end effector frame			
i	w_i	q_i	v_i
1	$(0,0,-1)$	$(-0.088,0,0.593)$	$(0,-0.088,0)$
2	$(0,-1,0)$	$(-0.088,0,0.593)$	$(0.593,0,0.088)$
3	$(0,0,-1)$	$(-0.088,0,0.277)$	$(0,-0.088,0)$
4	$(0,1,0)$	$(0,0,0.277)$	$(-0.277,0,0)$
5	$(0,0,-1)$	$(-0.088,0,-0.107)$	$(0,-0.088,0)$
6	$(0,1,0)$	$(-0.088,0,-0.107)$	$(0.107,0,-0.088)$
7	$(0,0,1)$		$(0,0,0)$

$$b_1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & -0.088 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$b_2 = \begin{bmatrix} 0 & 0 & -1 & -0.593 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0.088 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$b_3 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & -0.088 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$b_4 = \begin{bmatrix} 0 & 0 & 1 & -0.277 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$b_5 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & -0.088 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$b_6 = \begin{bmatrix} 0 & 0 & 1 & 0.107 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & -0.088 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$b_7 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Body form Jacobian:

The space form Jacobian has the following form

$$J_b(\theta) = \begin{bmatrix} J_{b1}(\theta) & J_{b2}(\theta) & \dots & J_{bn} \end{bmatrix},$$

where

$$J_{b1}(\theta) = Ad_{e^{-[B_n]\theta_n \cdots e^{-[B_2]\theta_2}}}(B_1),$$

$$J_{b2}(\theta) = Ad_{e^{-[B_n]\theta_n \cdots e^{-[B_3]\theta_3}}}(B_2),$$

$$J_{b_{n-1}}(\theta) = Ad_{e^{-[B_n]\theta_n}}(B_{n-1}),$$

$$J_{bn} = B_n.$$

From Table I, we have the following twist vectors:

$$B_1 = [0,0,-1,0,-0.088,0]^T,$$

$$B_2 = [0,-1,0,0.593,0,0.088]^T,$$

$$B_3 = [0,0,-1,0,-0.088,0]^T,$$

$$B_4 = [0,1,0,-0.277,0,0]^T,$$

$$B_5 = [0,0,-1,0,-0.088,0]^T,$$

$$B_6 = [0,1,0,0.107,0,-0.088]^T,$$

$$B_7 = [0,0,1,0,0,0]^T.$$