




Homework/Programming Assignment #2

Homework/midterm Due: 04/06/2023- 5:00 PM

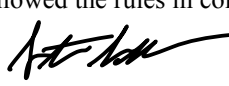
Name/EID: Jared Rosenbaum / jr73784

Email: jared.rosenbaum@utexas.edu

Signature (required) 
 I/We have followed the rules in completing this Assignment.

Name/EID: Steven Swanbeck sas9563

Email: stevenswanbeck@utexas.edu

Signature (required) 
 I/We have followed the rules in completing this Assignment.

Question	Points	Total
HA 1	25	
HA 2	25	
HA 3	25	
HA 4	25	
PA	100	
PA. k (Bonus)	15	
PA. m (Bonus)	30	
Presentation* (Bonus)	20	

Instruction:

- Remember that this is a graded assignment. It is the equivalent of a **midterm take-home exam**.
- * You should present the results of the PA in the class** and receive extra bonus depending on the quality of your presentation!
- For PA questions, you need to write a report showing how you derived your equations, describes your approach, test functions, and discusses the results.** You should show your test results for each function.
- You are to work **alone** or **in teams of two** and are **not to discuss the problems with anyone** other than the TAs or the instructor.
- It is open book, notes, and web. But you should cite any references you consult.
- Unless I say otherwise in class, it is due before the start of class on the due date mentioned in the Assignment.
- Sign and append** this score sheet as the first sheet of your assignment.
- Remember to submit your assignment in Canvas.

THA Programming Assignment #2

ME 397 ASBR, Sp 23, Dr. Farshid Alambeigi

Jared Rosenbaum and Steven Swanbeck

PA

General Description:

In this assignment, elements related to forward and inverse kinematics, Jacobians, and singularity situations are calculated and investigated for a Franka-Emika Panda robot. The dimensions and parameters of the robot that will be used throughout this assignment were found here [1]. These dimensions are also shown in Fig. 1, which is also taken from [1].

a) Space Form Inverse Kinematics

Problem Description:

This problem requests the derivation of the forward kinematics of the selected robot using the space form of the exponential products.

Method for Solution:

To find the forward kinematics of the Franka-Emika Panda, the Product of Exponentials method requires the screw axes of each robot joint as well as $M \in SE(3)$, the end-effector frame configuration with respect to the fixed base frame. Using the base configuration as found on the Franka-Emika github documentation, pictured below, M and the screw axes were defined as follows (Note: A screw axis is of the form $S = [q \ v]^T$).

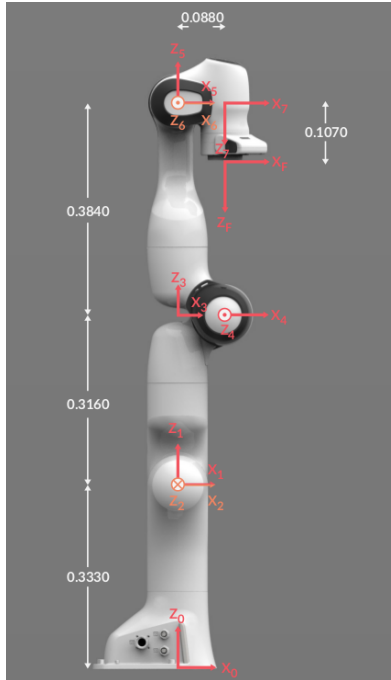


Fig. 1 shows what will be referred to as the “zero” configuration of the robot. The “ready” configuration of the robot will be referenced in many subsequent portions of this report, and is shown in Fig. 2 below.

Table 1. Symbolic screw axes

i	w	q	v
1	0 0 1	0 0 0	0 0 0
2	0 1 0	0 0 L1	L1 0 0
3	0 0 1	0 0 0	0 0 0
4	0 -1 0	a 0 L1+L2	L1+L2 0 -a
5	0 0 1	0 0 0	0 0 0
6	0 -1 0	0 0 L1+L2+L3	L1+L2+L3 0 0
7	0 0 -1	L4 0 L1+L2+L3	0 L4 0

$$M = \begin{bmatrix} 1 & 0 & 0 & L4 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & L1 + L2 + L3 - L5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

With the screw axes and M matrix defined, the space form of the Power of Exponentials forward kinematics is simply defined by the following equation:

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

[illegible]

b) *FK_space.m*

Problem Description:

This problem requests the creation of a function *FK_space.m* that calculates the space-form forward kinematics as done in problem 1, and represents the defined frames and screw axis graphically.

Method for Solution:

In order to numerically calculate the forward kinematics, the previously derived screw axes and end-effector position matrix *M* were populated with link lengths.

Table 2. Screw axes with numbers

<i>i</i>	<i>w</i>	<i>q</i>	<i>v</i>
1	0 0 1	0 0 0	0 0 0
2	0 1 0	0 0 0.333	-0.333 0 0
3	0 0 1	0 0 0	0 0 0
4	0 -1 0	0.0825 0 0.649	0.649 0 -0.0825
5	0 0 1	0 0 0	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

$$M = \begin{bmatrix} 1 & 0 & 0 & 0.088 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0.926 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Using these, the forward kinematics can be derived as described above, using the Product of Exponentials equation.

Explanation of Program:

The following function, *FK_space.m*, was developed to complete this problem.

```
function [FK_solution, T_bank, T_total_bank] = FK_space(M, S_mat, thetas,
is_symbolic, should_plot, M_intermediates, should_create_plot)
    %Accepts a base configuration transformation matrix M, a matrix of column
    vectors representing the screw axes, and a vector of theta values corresponding
    to the rotation about each screw axis
    % Should_plot will determine whether a plot is created for the frames or
    % not. If it and all following arguments are not set, this will default
    % to false. If set true, the user must also provide configurations
    % analogous to M corresponding to each joint location as a cell array of
    transformation matrices. If
    % should_create_plot is not provided, it will default true, and a new,
    % self-contained figure is created. To override this behavior, provide
    % false and the output will be plotted on the current figure in focus.

    % Pseudo-default arguments
    if nargin < 4
        is_symbolic = false;
    end
    if nargin < 5
        should_plot = false;
    elseif nargin < 7
        should_create_plot = true;
    end

    S_bank = cell(1, size(S_mat, 2));
    for i = 1:size(S_mat, 2)
        S_bank{i} = S_mat(:,i);
    end

    % To allow plotting base configuration frames
    T_bank_base = cell(1, size(S_mat, 2));
    for i = 1:size(S_mat, 2)
        T_bank_base{i} = S2T(S_bank{i}, 0, is_symbolic);
    end

    % Transformations using the input angles
    T_bank = cell(1, size(S_mat, 2));
    for i = 1:size(S_mat, 2)
        T_bank{i} = S2T(S_bank{i}, thetas(i), is_symbolic);
    end

    % Performing space forward kinematics
    T_total = eye(4);
    T_total_bank = cell(1, size(S_mat, 2));
    for i = 1:size(T_bank,2)
        T_total = T_total * T_bank{i};
        T_total_bank{i} = T_total;
    end
```

```

% Final solution calculation
FK_solution = T_total * M;

% Plot of new frames
if should_plot == true
if should_create_plot == true
    figure
    view(3)
    box on
    hold on
    grid on
    axis equal;
    xlabel('x'), ylabel('y'), zlabel('z');
end
plotFrame_T(eye(4), '$\lbrace s \rbrace$', 0.1)
stored_origin = [0, 0, 0];
for i = 1:length(thetas)
    T_total = T_total_bank{i} * M_intermediates{i};
    plotFrame_T(T_total, strcat('$\lbrace b_', string(i) , '\rbrace$'),
0.1);

    plotLink_p(stored_origin, T_total(1:3, 4));
    stored_origin = T_total(1:3, 4);
end
Tf = T_total_bank{end} * M;
plotFrame_T(Tf, '$\lbrace f \rbrace$', 0.1)
plotLink_p(stored_origin, Tf(1:3, 4));

    plotScrewAxis_T(Tf);
end
end

```

FK_space accepts the matrix M , an array of screw axes, and an array of joint angles. The function also has four optional inputs. The first input prompts the user to specify whether the inputs are symbolic (as seen in problem a) or real (as used in this problem). The next three inputs are related to plotting - the function inquires if it should create a plot of the robot in the specified position, if the plot should be a new figure or if it should be on an existing figure, and finally, if it is told that it should plot, it requests a set of M matrices for each individual joint, allowing each joint to be plotted using an intermediate Power of Exponentials forward kinematics calculation.

The *FK_space* function calls upon a few other functions. The first and foremost of these functions is the *S2T* function created for THA1 - this function turns a screw axis and an angle θ into a 4x4 transformation matrix. The other functions that are called are all related to plotting. These functions are *plotFrame_T*, *plotLink_p*, and *plotScrewAxis_T*. These aptly named functions plot the frames of each joint, the links of the Panda, and the screw axis between the base and end-effector as found by the forward kinematics. The functions are implemented as follows.


```

function plotFrame_T(T, label, delta)
%Plots coordinate frame corresponding to rigid body transformation matrix T
%in an existing 3D plot
% label is a string that will be attached to the frame and delta scales
% the lengths of the axes. Default for delta is 1, corresponding to one
% distance unit.

% Default argument sets delta if not provided
if nargin < 3
    delta = 1;
end
origin = T(1:3, 4)';

% x-axisdelta
line('XData', [origin(1) origin(1) + delta * T(1,1)], 'YData', [origin(2)
origin(2) + delta * T(2,1)],...
'ZData', [origin(3) origin(3) + delta * T(3,1)],
'Color','r','LineWidth',3);

% y-axis
line('XData', [origin(1) origin(1) + delta * T(1,2)], 'YData', [origin(2)
origin(2) + delta * T(2,2)],...
'ZData', [origin(3) origin(3) + delta * T(3,2)],
'Color','g','LineWidth',3);

% z-axis
line('XData', [origin(1) origin(1) + delta * T(1,3)], 'YData', [origin(2)
origin(2) + delta * T(2,3)],...
'ZData', [origin(3) origin(3) + delta * T(3,3)],
'Color','b','LineWidth',3);

% frame label
% text(origin(1) - delta * 0.2 * T(1,1), origin(2) - delta * 0.2 * T(2,2),
origin(3) - delta * 0.2 * T(3,3), horzcat('$\lbrace ', label, ' \rbrace$'));
text(origin(1) - delta * 0.3 * T(1,1), origin(2) - delta * 0.3 * T(2,2),
origin(3) - delta * 0.3 * T(3,3), label);

% axes labels
text(origin(1) + delta * T(1,1), origin(2) + delta * T(2,1), origin(3) +
delta * T(3,1), '$x$');
text(origin(1) + delta * T(1,2), origin(2) + delta * T(2,2), origin(3) +
delta * T(3,2), '$y$');
text(origin(1) + delta * T(1,3), origin(2) + delta * T(2,3), origin(3) +
delta * T(3,3), '$z$');
end

```

```

function plotLink_p(p0, p1, alpha)
%Plots a link between two points. Useful for visualizing manipulator
%links. alpha is link transparency and defaults to 0.5.
if nargin < 3

```

```

    alpha = 0.5;
    end
    line('XData', [p0(1) p1(1)], 'YData', [p0(2) p1(2)],...
        'ZData', [p0(3) p1(3)], 'Color',[0.4, 0.4, 0.4, alpha], 'LineWidth',10);
end

function plotScrewAxis_T(T, C)
    %Plots an equivalent screw axis in an existing figure given a transformation
    matrix
    % length of frame vectors
    if nargin < 2
        C = 1;
    end

    S = T2S(T);
    w = S(1:3);
    v = S(4:6);

    h = w' * v;
    s_hat = w;
    q = Axis2SkewSymmetricMatrix(s_hat) * (v - h * s_hat);

    plotScrewAxis_qsh(q, s_hat, h, C);
end

```

Answer / Test Cases:

The FK_space function was tested extensively with many robot configurations. Some of these test cases are shown below.

Test Case 1: The robot is in the zero configuration, with $\text{Thetas} = [0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0]$

The function returns:

1.00	0	0	0.0880
0	-1.00	0	0
0	0	-1.00	0.9260
0	0	0	1.00

Forward Kinematics for Test1

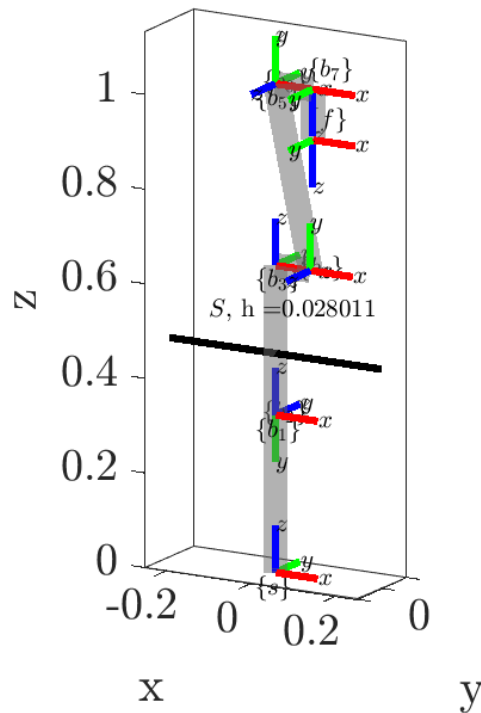


Figure 3. FK_space test case 1

The relative frames align with expectations and the dimensions are all correct from Figure 1 [1]. The screw axis that defines the rotation between the base and end-effector frames of the robot is shown in black and labeled with the corresponding pitch.

Test Case 2: The robot is in the “ready” configuration, and has $\Theta = [0 \ -\pi/4 \ 0 \ -3\pi/4 \ 0 \ \pi/2 \ \pi/4]$

The function returns:

```

0.7071 -0.7071 -0.0000 0.3069
-0.7071 -0.7071 0 0
-0.0000 0.0000 -1.0000 0.5903
0 0 0 1.0000

```

Forward Kinematics for Test2

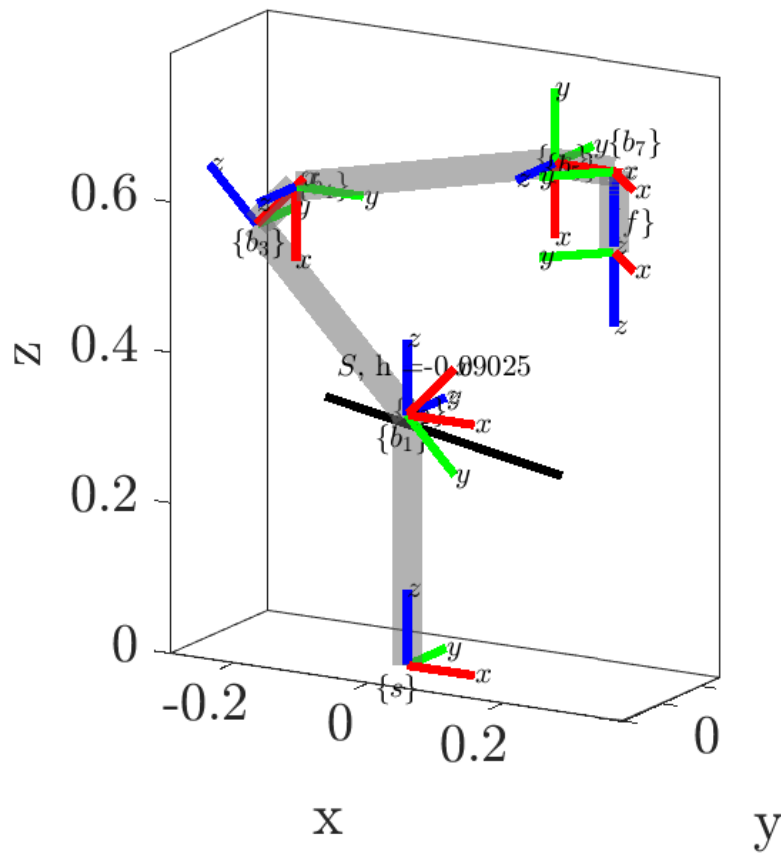


Figure 4. *FK_space* test case 2

The relative frames align with expectations and the dimensions are all correct from Figure 2 [1]. The screw axis that defines the rotation between the base and end-effector frames of the robot is shown in black and labeled with the corresponding pitch.

Test Case 3: The robot has randomly generated Thetas = [4.7418 1.7343 4.2707 4.1161 1.0217 0.7477 3.1313]

The function returns:

```

0.6102 -0.6304 0.4798 -0.3995
-0.7326 -0.6795 0.0390 -0.1400
0.3014 -0.3753 -0.8765 0.3739
0      0      0      1.0000

```

Forward Kinematics for Test3

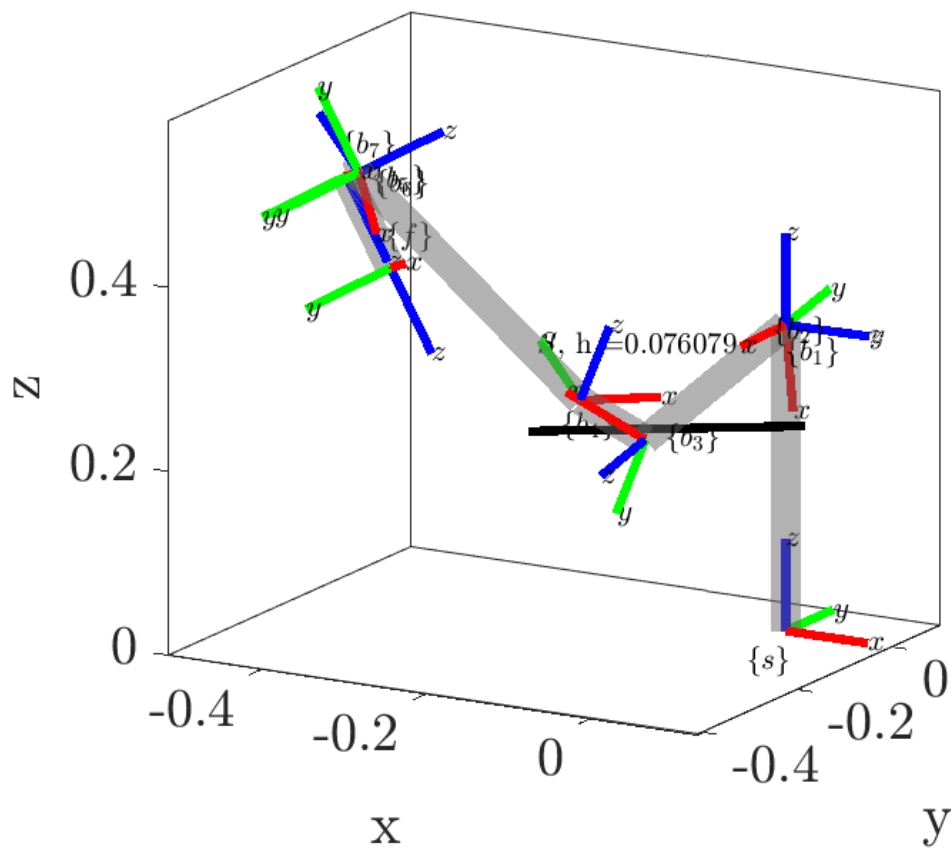


Figure 5. *FK_space* test case 3

c) Body Form Inverse Kinematics and *FK_body.m*

Problem Description:

This problem requests the body-form forward kinematic equivalents for problems A and B.

Method for Solution:

The approach for the body-form forward kinematics problem is very similar to the space-form. The only difference between the two is the derivation of the screw axes are with respect to the end-effector frame, and the final equation is instead of the form:

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

The corresponding screw axes are:

Table 2. Body-form symbolic screw axes

i	w	q	v
1	0 0 -1	-L4 0 L2+L3-L5	0 -L4 0
2	0 -1 0	-L4 0 L2+L3-L5	L2+L3-L5 0 L4
3	0 0 -1	-L4 0 L3-L5	0 -L4 0
4	0 1 0	-L4+a 0 L3-L5	L5-L3 0 -L4+a
5	0 0 -1	-L4 0 -L5	0 -L4 0
6	0 1 0	-L4 0 -L5	L5 0 -L4
7	0 0 1	0 0 0	0 0 0

Table 2. Body form screw axes with numbers

i	w	q	v
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	-0.0055 0 0.277	-0.277 0 -0.0055
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	0 0 0

With M the same as in parts A and B, the forward kinematics can now be computed.

Explanation of Program:

The following function, *FK_body.m*, was developed to complete this problem.

```
function [FK_solution, T_bank, T_total_bank] = FK_body(M, B_mat, thetas,
is_symbolic, should_plot, M_intermediates, should_create_plot)
    %Accepts a base configuration transformation matrix M, a matrix of column
    vectors representing the screw axes, and a vector of theta values corresponding
    to the rotation about each screw axis
    % Should_plot will determine whether a plot is created for the frames or
    % not. If it and all following arguments are not set, this will default
    % to false. If set true, the user must also provide configurations
    % analogous to M corresponding to each joint location as a cell array of
    transformation matrices. If
    % should_create_plot is not provided, it will default true, and a new,
    % self-contained figure is created. To override this behavior, provide
    % false and the output will be plotted on the current figure in focus.

    % Pseudo-default arguments
    if nargin < 4
        is_symbolic = false;
    end
    if nargin < 5
        should_plot = false;
    elseif nargin < 7
        should_create_plot = true;
    end

    B_bank = cell(1, size(B_mat, 2));
    for i = 1:size(B_mat, 2)
        B_bank{i} = B_mat(:,i);
    end

    % To allow plotting base configuration frames
    T_bank_base = cell(1, size(B_mat, 2));
    for i = 1:size(B_mat, 2)
        T_bank_base{i} = S2T(B_bank{i}, 0, is_symbolic);
    end

    % Transformations using the input angles
    T_bank = cell(1, size(B_mat, 2));
    for i = 1:size(B_mat, 2)
        T_bank{i} = S2T(B_bank{i}, thetas(i), is_symbolic);
    end

    % Performing body forward kinematics
    T_total = eye(4);
    T_total_bank = cell(1, size(B_mat, 2));
    for i = 1:size(T_bank,2)
        T_total = T_total * T_bank{i};
        T_total_bank{i} = T_total;
    end
end
```

```

% Final solution calculation
FK_solution = M * T_total;

% Plot of new frames
if should_plot == true
    if should_create_plot == true
        figure
        view(3)
        box on
        hold on
        grid on
        axis equal;
        xlabel('x'), ylabel('y'), zlabel('z');
        %         xlim([-0.75 0.75]);
        %         ylim([-0.75 0.75]);
        %         zlim([0 1.5]);
    end
    plotFrame_T(eye(4), '$\lbrace s \rbrace$', 0.1)
    stored_origin = [0, 0, 0];
    for i = 1:length(thetas)
        T_total = M_intermediates{i} * T_total_bank{i};
        plotFrame_T(T_total, strcat('$\lbrace b_{', string(i) , '\rbrace$'),
0.1);

        plotLink_p(stored_origin, T_total(1:3, 4));
        stored_origin = T_total(1:3, 4);
    end
    Tf = M * T_total_bank{end};
    plotFrame_T(Tf, '$\lbrace f \rbrace$', 0.1)
    plotLink_p(stored_origin, Tf(1:3, 4));

    plotScrewAxis_T(Tf);
end
end

```

FK_body functions nearly identically to *FK_space*, with the only significant difference being the pre-multiplication of the *M* matrix.

Answer / Test Cases:

The *FK_body* function was tested extensively with many robot configurations. Some of these test cases are shown below. Note that the intermediate frames of the robot are not shown visually here in these cases, as the screw axes were defined relative to the tip of the robot, and so would need to be redefined for each intermediate frame to plot. Alternatively, we could convert the space twists from the space FK into the body frame, but this is not done here. To still give the approximated semblance of the robot, the end-effector frame is connected to the base frame by a single link.

Test Case 1: The robot is in the zero configuration, with $\text{Thetas} = [0 \ 0 \ 0 \ 0 \ 0 \ 0]$

The function returns:

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

Forward Kinematics for Test1

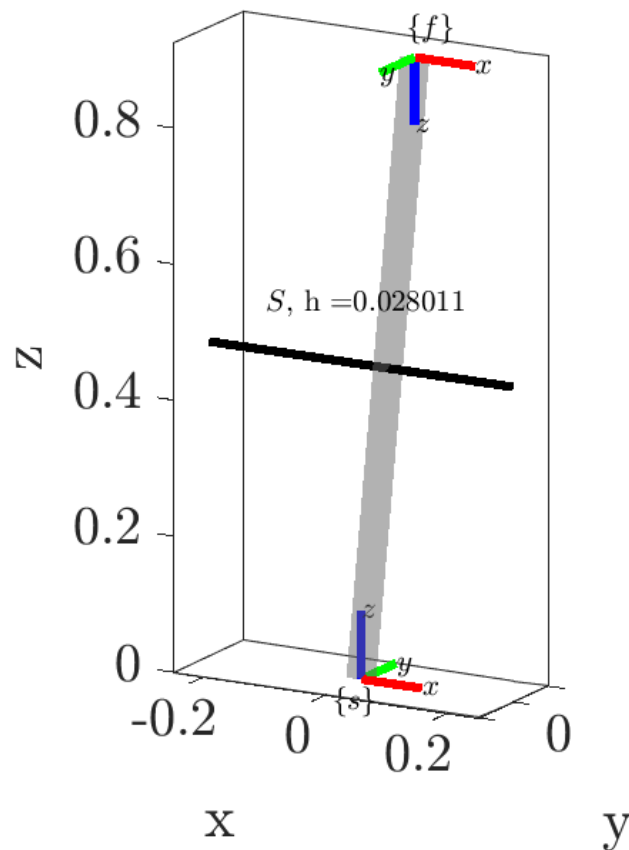


Figure 6. *FK_Body* test case 1

The poses of the base and end-effector frames agree with the space FK solution, as does the calculated screw axis between the two frames.

Test Case 2: The robot is in the “ready” configuration, and has $\Theta = [0 \ -\pi/4 \ 0 \ -3\pi/4 \ 0 \ \pi/2 \ \pi/4]$

The function returns:

```

0.7071 -0.7071 -0.0000 0.3069
-0.7071 -0.7071 0 0
-0.0000 0.0000 -1.0000 0.5903
0 0 0 1.0000

```

Forward Kinematics for Test2

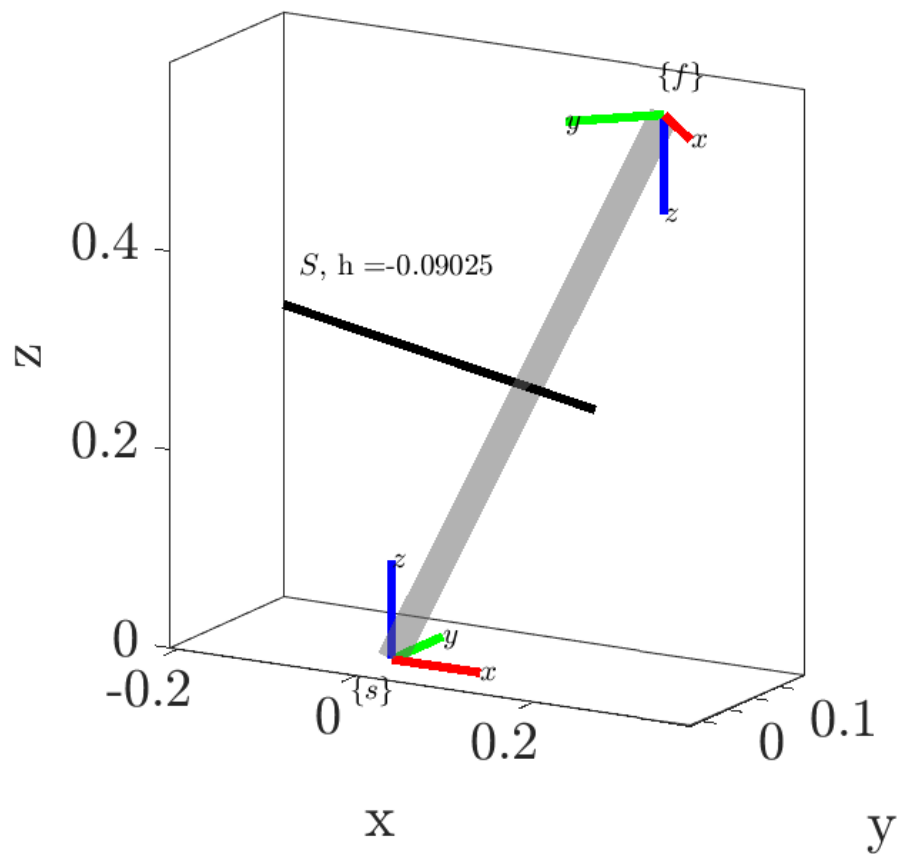


Figure 7. FK_Body test case 2

The poses of the base and end-effector frames agree with the space FK solution, as does the calculated screw axis between the two frames.

Test Case 3: The robot has randomly generated Thetas = [4.7418 1.7343 4.2707 4.1161 1.0217 0.7477 3.1313]

The function returns:

```
0.6102 -0.6304 0.4798 -0.3995
-0.7326 -0.6795 0.0390 -0.1400
0.3014 -0.3753 -0.8765 0.3739
0 0 0 1.0000
```

Forward Kinematics for Test3

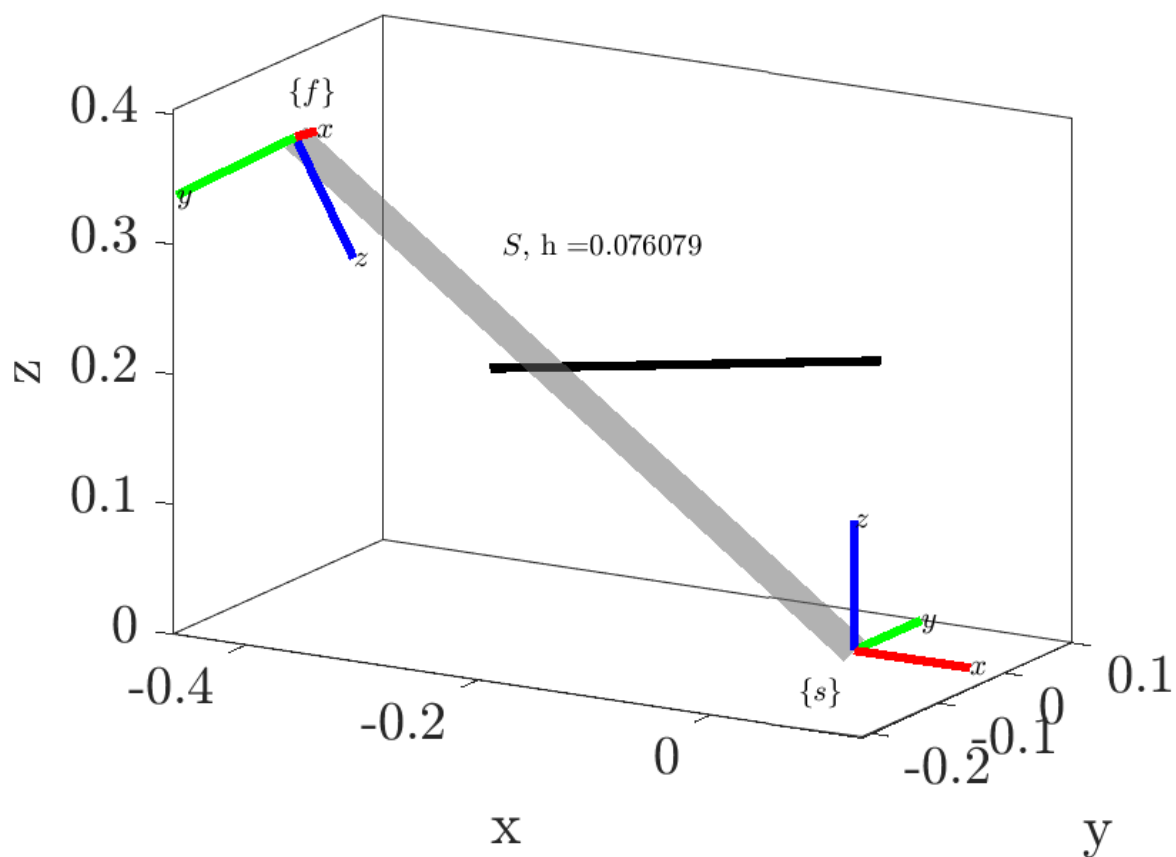


Figure 8. FK_body test case 3

The poses of the base and end-effector frames agree with the space FK solution, as does the calculated screw axis between the two frames.

$$[0, -L1*\sin(t1), L1*\cos(t1)*\sin(t2), L1*\cos(t3)*\sin(t1) + L2*\cos(t1)*\sin(t3) - a*\sin(t1)*\sin(t2) + L1*\cos(t1)*\cos(t2)*\sin(t3) + L2*\cos(t2)*\cos(t3)*\sin(t1), \\ (\cos(t2)*\cos(t4) + \cos(t3)*\sin(t2)*\sin(t4))*(\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3))*(\sin(t4)*(L1 + L2) - a*(\cos(t4) - 1)) + L1*\cos(t1)*\sin(t2) + \\ \cos(t1)*\sin(t2)*((\cos(t4) - 1)*(L1 + L2) + a*\sin(t4))) - (\sin(t4)*(\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3)) + \cos(t1)*\cos(t4)*\sin(t2))*(\cos(t2)*((\cos(t4) - 1)*(L1 + L2) + a*\sin(t4))) - (\sin(t4)*(\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3)) + \cos(t1)*\cos(t4)*\sin(t2)))]$$

$$\begin{aligned}
& + a*\sin(t4)) + L1*(\cos(t2) - 1) + \cos(t3)*\sin(t2)*(\sin(t4)*(L1 + L2) - a*(\cos(t4) - 1))), (\cos(t5)*(\cos(t4)*(\cos(t1)*\sin(t3) + \cos(t2)*\cos(t3)*\sin(t1)) + \\
& \sin(t1)*\sin(t2)*\sin(t4)) + \sin(t5)*(\cos(t1)*\cos(t3) - \cos(t2)*\sin(t1)*\sin(t3)))*(L1 + L2 + L3) + (\sin(t5)*(\cos(t2)*\sin(t4) - \cos(t3)*\cos(t4)*\sin(t2)) - \\
& \cos(t5)*\sin(t2)*\sin(t3)))*((\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3))*(\sin(t4)*(L1 + L2) - a*(\cos(t4) - 1)) + L1*\cos(t1)*\sin(t2) + \cos(t1)*\sin(t2)*((\cos(t4) - 1)*(L1 + \\
& L2) + a*\sin(t4))) + (\sin(t5)*(\cos(t4)*(\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3)) - \cos(t1)*\sin(t2)*\sin(t4)) - \cos(t5)*(\cos(t3)*\sin(t1) + \\
& \cos(t1)*\cos(t2)*\sin(t3)))*(\cos(t2)*((\cos(t4) - 1)*(L1 + L2) + a*\sin(t4)) + L1*(\cos(t2) - 1) + \cos(t3)*\sin(t2)*(\sin(t4)*(L1 + L2) - a*(\cos(t4) - 1))), \\
& (\cos(t6)*(\sin(t4)*(\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3)) + \cos(t1)*\cos(t4)*\sin(t2)) + \sin(t6)*(\cos(t5)*(\cos(t4)*(\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3)) - \\
& \cos(t1)*\sin(t2)*\sin(t4)) + \sin(t5)*(\cos(t3)*\sin(t1) + \cos(t1)*\cos(t2)*\sin(t3))))*(\cos(t4) - 1)*(L1 + L2) + a*\sin(t4)) + L1*(\cos(t2) - 1) + (\cos(t6) - \\
& 1)*(\cos(t2)*\cos(t4) + \cos(t3)*\sin(t2)*\sin(t4))*(L1 + L2 + L3) - \sin(t6)*(\cos(t5)*(\cos(t2)*\sin(t4) - \cos(t3)*\cos(t4)*\sin(t2)) + \sin(t2)*\sin(t3)*\sin(t5))*(L1 + L2 + L3) + \\
& \cos(t3)*\sin(t2)*(\sin(t4)*(L1 + L2) - a*(\cos(t4) - 1))) - L4*(\sin(t5)*(\cos(t4)*(\cos(t1)*\sin(t3) + \cos(t2)*\cos(t3)*\sin(t1)) + \sin(t1)*\sin(t2)*\sin(t4)) - \\
& \cos(t5)*(\cos(t1)*\cos(t3) - \cos(t2)*\sin(t1)*\sin(t3))) + (\sin(t6)*(\cos(t5)*(\cos(t2)*\sin(t4) - \cos(t3)*\cos(t4)*\sin(t2)) + \sin(t2)*\sin(t3)*\sin(t5)) - \cos(t6)*(\cos(t2)*\cos(t4) + \\
& \cos(t3)*\sin(t2)*\sin(t4)))*((\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3))*(\sin(t4)*(L1 + L2) - a*(\cos(t4) - 1)) + L1*\cos(t1)*\sin(t2) + \cos(t1)*\sin(t2)*((\cos(t4) - 1)*(L1 + \\
& L2) + a*\sin(t4)) + (\sin(t4)*(\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3)) + \cos(t1)*\cos(t4)*\sin(t2))*(\cos(t6) - 1)*(L1 + L2 + L3) + \\
& \sin(t6)*(\cos(t5)*(\cos(t4)*(\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3)) - \cos(t1)*\sin(t2)*\sin(t4)) + \sin(t5)*(\cos(t3)*\sin(t1) + \cos(t1)*\cos(t2)*\sin(t3))))*(L1 + L2 + L3)]]
\end{aligned}$$

$$\begin{aligned}
& [0, \quad 0, \quad 0, \quad -a*\cos(t2) - L2*\cos(t3)*\sin(t2), -\sin(t2)*\sin(t3)*(L2*\sin(t4) - a + a*\cos(t4)), (\cos(t5)*(\cos(t2)*\sin(t4) - \cos(t3)*\cos(t4)*\sin(t2)) + \\
& \sin(t2)*\sin(t3)*\sin(t5))*(L1 + L2 + L3) - (\sin(t5)*(\cos(t4)*(\cos(t1)*\sin(t3) + \cos(t2)*\cos(t3)*\sin(t1)) + \sin(t1)*\sin(t2)*\sin(t4)) - \cos(t5)*(\cos(t1)*\cos(t3) - \\
& \cos(t1)*\sin(t2)*\sin(t3)))*((\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3))*(\sin(t4)*(L1 + L2) - a*(\cos(t4) - 1)) + L1*\cos(t1)*\sin(t2) + \cos(t1)*\sin(t2)*((\cos(t4) - 1)*(L1 + \\
& L2) + a*\sin(t4))) - (\sin(t5)*(\cos(t4)*(\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3)) - \cos(t1)*\sin(t2)*\sin(t4)) - \cos(t5)*(\cos(t3)*\sin(t1) + \\
& \cos(t1)*\cos(t2)*\sin(t3)))*(L1*\sin(t1)*\sin(t2) - (\cos(t1)*\sin(t3) + \cos(t2)*\cos(t3)*\sin(t1))*(\sin(t4)*(L1 + L2) - a*(\cos(t4) - 1)) + \sin(t1)*\sin(t2)*((\cos(t4) - 1)*(L1 + \\
& L2) + a*\sin(t4))), (\cos(t6)*(\sin(t4)*(\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3)) + \cos(t1)*\cos(t4)*\sin(t2)) + \sin(t6)*(\cos(t5)*(\cos(t4)*(\sin(t1)*\sin(t3) - \\
& \cos(t1)*\sin(t2)*\sin(t4)) + \sin(t5)*(\cos(t3)*\sin(t1) + \cos(t1)*\cos(t2)*\sin(t3))))*((\cos(t1)*\sin(t3) + \cos(t2)*\cos(t3)*\sin(t1)))*(\sin(t4)*(L1 + \\
& L2) - a*(\cos(t4) - 1)) - L1*\sin(t1)*\sin(t2) - \sin(t1)*\sin(t2)*((\cos(t4) - 1)*(L1 + L2) + a*\sin(t4)) + (\sin(t4)*(\cos(t1)*\sin(t3) + \cos(t2)*\cos(t3)*\sin(t1)) - \\
& \cos(t4)*\sin(t1)*\sin(t2))*(\cos(t6) - 1)*(L1 + L2 + L3) + \sin(t6)*(\cos(t5)*(\cos(t4)*(\cos(t1)*\sin(t3) + \cos(t2)*\cos(t3)*\sin(t1)) + \sin(t1)*\sin(t2)*\sin(t4)) + \\
& \sin(t5)*(\cos(t1)*\cos(t3) - \cos(t2)*\sin(t1)*\sin(t3)))*(L1 + L2 + L3)) - (\cos(t6)*(\sin(t4)*(\cos(t1)*\sin(t3) + \cos(t2)*\cos(t3)*\sin(t1)) - \cos(t4)*\sin(t1)*\sin(t2)) + \\
& \sin(t6)*(\cos(t5)*(\cos(t4)*(\cos(t1)*\sin(t3) + \cos(t2)*\cos(t3)*\sin(t1)) + \sin(t5)*(\cos(t1)*\cos(t3) - \cos(t2)*\sin(t1)*\sin(t3)))*((\sin(t1)*\sin(t3) - \\
& \cos(t1)*\cos(t2)*\cos(t3))*(\sin(t4)*(L1 + L2) - a*(\cos(t4) - 1)) + L1*\cos(t1)*\sin(t2) + \cos(t1)*\sin(t2)*((\cos(t4) - 1)*(L1 + L2) + a*\sin(t4)) + (\sin(t4)*(\sin(t1)*\sin(t3) - \\
& \cos(t1)*\cos(t2)*\cos(t3)) + \cos(t1)*\cos(t4)*\sin(t2))*(\cos(t6) - 1)*(L1 + L2 + L3) + \sin(t6)*(\cos(t5)*(\cos(t4)*(\sin(t1)*\sin(t3) - \cos(t1)*\cos(t2)*\cos(t3)) - \\
& \cos(t1)*\sin(t2)*\sin(t4)) + \sin(t5)*(\cos(t3)*\sin(t1) + \cos(t1)*\cos(t2)*\sin(t3)))*(L1 + L2 + L3)) - L4*(\sin(t5)*(\cos(t2)*\sin(t4) - \cos(t3)*\cos(t4)*\sin(t2)) - \\
& \cos(t5)*\sin(t2)*\sin(t3)]]
\end{aligned}$$

Body Jacobian:

$$\begin{aligned}
& [\cos(t2)*(\sin(t4)*(\sin(t5)*\sin(t7) + \cos(t5)*\cos(t6)*\cos(t7)) + \cos(t4)*\cos(t7)*\sin(t6)) - \sin(t2)*(\cos(t3)*(\cos(t4)*(\sin(t5)*\sin(t7) + \cos(t5)*\cos(t6)*\cos(t7)) - \\
& \cos(t7)*\sin(t4)*\sin(t6)) + \sin(t3)*(\cos(t5)*\sin(t7) - \cos(t6)*\cos(t7)*\sin(t5))), \sin(t3)*(\cos(t4)*(\sin(t5)*\sin(t7) + \cos(t5)*\cos(t6)*\cos(t7)) - \cos(t7)*\sin(t4)*\sin(t6)) - \\
& \cos(t3)*(\cos(t5)*\sin(t7) - \cos(t6)*\cos(t7)*\sin(t5)), \sin(t4)*(\sin(t5)*\sin(t7) + \cos(t5)*\cos(t6)*\cos(t7)) + \cos(t4)*\cos(t7)*\sin(t6), \quad \cos(t5)*\sin(t7) - \\
& \cos(t6)*\cos(t7)*\sin(t5), \quad \cos(t7)*\sin(t6), \quad \sin(t7), 0]
\end{aligned}$$

$$\begin{aligned}
& [\cos(t2)*(\sin(t4)*(\cos(t7)*\sin(t5) - \cos(t5)*\cos(t6)*\sin(t7)) - \cos(t4)*\sin(t6)*\sin(t7)) - \sin(t2)*(\cos(t3)*(\cos(t4)*(\cos(t7)*\sin(t5) - \cos(t5)*\cos(t6)*\sin(t7)) + \\
& \sin(t4)*\sin(t6)*\sin(t7)) + \sin(t3)*(\cos(t5)*\cos(t7) + \cos(t6)*\sin(t5)*\sin(t7))), \sin(t3)*(\cos(t4)*(\cos(t7)*\sin(t5) - \cos(t5)*\cos(t6)*\sin(t7)) + \sin(t4)*\sin(t6)*\sin(t7)) - \\
& \cos(t3)*(\cos(t5)*\cos(t7) + \cos(t6)*\sin(t5)*\sin(t7)), \sin(t4)*(\cos(t5)*\sin(t7) - \cos(t6)*\sin(t5)*\sin(t7)) - \cos(t5)*\cos(t6)*\sin(t7)) - \cos(t4)*\sin(t6)*\sin(t7), \\
& \cos(t5)*\cos(t7) + \cos(t6)*\sin(t5)*\sin(t7), \quad -\sin(t6)*\sin(t7), \quad \cos(t7), 0]
\end{aligned}$$

$$\begin{aligned}
& [-\sin(t2)*(\cos(t3)*(\cos(t6)*\sin(t4) + \cos(t4)*\cos(t5)*\sin(t6)) - \sin(t3)*\sin(t5)*\sin(t6)) - \cos(t2)*(\cos(t4)*\cos(t6) - \cos(t5)*\sin(t4)*\sin(t6)), \\
& \sin(t3)*(\cos(t6)*\sin(t4) + \cos(t4)*\cos(t5)*\sin(t6)) + \cos(t3)*\sin(t5)*\sin(t6), \cos(t5)*\sin(t4)*\sin(t6) - \cos(t4)*\cos(t6), \\
& -\sin(t5)*\sin(t6), -\cos(t6), \quad 0, 1]
\end{aligned}$$

$$\begin{aligned}
& [L4*(\sin(t3)*(\cos(t4)*(\sin(t5)*\sin(t7) + \cos(t5)*\cos(t6)*\cos(t7)) - \cos(t7)*\sin(t4)*\sin(t6)) - \cos(t3)*(\cos(t5)*\sin(t7) - \cos(t6)*\cos(t7)*\sin(t5))) - \\
& (\sin(t2)*(\cos(t3)*(\cos(t6)*\sin(t4) + \cos(t4)*\cos(t5)*\sin(t6)) - \sin(t3)*\sin(t5)*\sin(t6)) + \cos(t2)*(\cos(t4)*\cos(t6) - \cos(t5)*\sin(t4)*\sin(t6)))*(\sin(t7)*(L5*\sin(t6) - \\
& L4*(\cos(t6) - 1)) + (\sin(t4)*(\cos(t7)*\sin(t5) - \cos(t5)*\cos(t6)*\sin(t7)) - \cos(t4)*\sin(t6)*\sin(t7))*(\cos(t2) - 1)*(L2 + L3 - L5) + L4*\sin(t2)) + (\cos(t7)*\sin(t5) - \\
& \cos(t5)*\cos(t6)*\sin(t7))*((\cos(t4) - 1)*(L4 - a) + \sin(t4)*(L3 - L5)) + (L4*(\cos(t2) - 1) - \sin(t2)*(L2 + L3 - L5))*(\cos(t3)*(\cos(t4)*(\cos(t7)*\sin(t5) - \\
& \cos(t5)*\cos(t6)*\sin(t7)) + \sin(t3)*(\cos(t5)*\cos(t7) + \cos(t6)*\sin(t5)*\sin(t7))) + L4*(\cos(t4)*(\cos(t7)*\sin(t5) - \cos(t5)*\cos(t6)*\sin(t7)) + \\
& \sin(t4)*\sin(t6)*\sin(t7))*(\cos(t3) - 1) + L4*\sin(t3)*(\cos(t5)*\cos(t7) + \cos(t6)*\sin(t5)*\sin(t7)) + \sin(t6)*\sin(t7)*(\sin(t4)*(L4 - a) - (\cos(t4) - 1)*(L3 - L5)) + \\
& L4*\cos(t7)*\sin(t5) - L4*\cos(t6)*\sin(t7)*(\cos(t5) - 1)) - (\cos(t2)*(\sin(t4)*(\cos(t7)*\sin(t5) - \cos(t5)*\cos(t6)*\sin(t7)) - \cos(t4)*\sin(t6)*\sin(t7)) - \\
& \sin(t2)*(\cos(t3)*(\cos(t4)*(\cos(t7)*\sin(t5) - \cos(t5)*\cos(t6)*\sin(t7)) + \sin(t4)*\sin(t6)*\sin(t7)) + \sin(t3)*(\cos(t5)*\cos(t7) + \\
& \cos(t6)*\sin(t5)*\sin(t6)))*(\cos(t6)*\sin(t4) + \cos(t4)*\cos(t5)*\sin(t6)) + \cos(t3)*(\cos(t6)*\sin(t4) + \cos(t4)*\cos(t5)*\sin(t6)) - \\
& \sin(t3)*\sin(t5)*\sin(t6))*(L4*(\cos(t2) - 1) - \sin(t2)*(L2 + L3 - L5)) + L5*(\cos(t6) - 1) - ((\cos(t2) - 1)*(L2 + L3 - L5) + L4*\sin(t2))*(\cos(t4)*\cos(t6) - \\
& \cos(t5)*\sin(t4)*\sin(t6)) + \cos(t5)*\sin(t6)*((\cos(t4) - 1)*(L4 - a) + \sin(t4)*(L3 - L5)) + L4*(\cos(t3) - 1)*(\cos(t6)*\sin(t4) + \cos(t4)*\cos(t5)*\sin(t6)) + \\
& L4*\sin(t6)*(\cos(t5) - 1) - L4*\sin(t3)*\sin(t5)*\sin(t6)), (\sin(t3)*(\cos(t6)*\sin(t4) + \cos(t4)*\cos(t5)*\sin(t6)) + \cos(t3)*\sin(t5)*\sin(t6))*(\sin(t7)*(L5*\sin(t6) - L4*(\cos(t6) \\
& - 1)) + (\cos(t7)*\sin(t5) - \cos(t5)*\cos(t6)*\sin(t7))*((\cos(t4) - 1)*(L4 - a) + \sin(t4)*(L3 - L5)) + L4*(\cos(t4)*(\cos(t7)*\sin(t5) - \cos(t5)*\cos(t6)*\sin(t7)) + \\
& \sin(t4)*\sin(t6)*\sin(t7))*(\cos(t3) - 1) + L4*\sin(t3)*(\cos(t5)*\cos(t7) + \cos(t6)*\sin(t5)*\sin(t7)) + \sin(t6)*\sin(t7)*(\sin(t4)*(L4 - a) - (\cos(t4) - 1)*(L3 - L5)) + \\
& L4*\cos(t7)*\sin(t5) - L4*\cos(t6)*\sin(t7)*(\cos(t5) - 1)) + (\cos(t3)*(\cos(t4)*(\sin(t5)*\sin(t7) + \cos(t5)*\cos(t6)*\cos(t7)) - \cos(t7)*\sin(t4)*\sin(t6)) + \\
& \sin(t3)*(\cos(t5)*\cos(t7) - \cos(t6)*\cos(t7)*\sin(t5)))*(L2 + L3 - L5) - (\sin(t3)*(\cos(t4)*(\cos(t7)*\sin(t5) - \cos(t5)*\cos(t6)*\sin(t7)) + \sin(t4)*\sin(t6)*\sin(t7)) - \\
& \cos(t3)*(\cos(t5)*\cos(t7) + \cos(t6)*\sin(t5)*\sin(t7)))*(\cos(t6)*(\sin(t4)*(L4 - a) - (\cos(t4) - 1)*(L3 - L5)) + L4*\sin(t6) + L5*(\cos(t6) - 1) + \cos(t5)*\sin(t6)*((\cos(t4) - \\
& 1)*(L4 - a) + \sin(t4)*(L3 - L5)) + L4*(\cos(t3) - 1)*(\cos(t6)*\sin(t4) + \cos(t4)*\cos(t5)*\sin(t6)) + L4*\sin(t6)*(\cos(t5) - 1) - L4*\sin(t3)*\sin(t5)*\sin(t6)) - \\
& L4*(\sin(t4)*(\sin(t5)*\sin(t7) + \cos(t5)*\cos(t6)*\cos(t7)) + \cos(t4)*\cos(t7)*\sin(t6)), -L4*(\cos(t5)*\sin(t7) - \cos(t6)*\cos(t7)*\sin(t5)) - (\cos(t4)*\cos(t6) - \\
& \cos(t5)*\sin(t4)*\sin(t6))*(\sin(t7)*(L5*\sin(t6) - L4*(\cos(t6) - 1)) + (\cos(t7)*\sin(t5) - \cos(t5)*\cos(t6)*\sin(t7))*((\cos(t4) - 1)*(L4 - a) + \sin(t4)*(L3 - L5)) + \\
& \sin(t6)*\sin(t7)*(\sin(t4)*(L4 - a) - (\cos(t4) - 1)*(L3 - L5)) + L4*\cos(t7)*\sin(t5) - L4*\cos(t6)*\sin(t7)*(\cos(t5) - 1)) - (\sin(t4)*(\cos(t7)*\sin(t5) - \cos(t5)*\cos(t6)*\sin(t7)) - \\
& \cos(t4)*\sin(t6)*\sin(t7))*(\cos(t6)*(\sin(t4)*(L4 - a) - (\cos(t4) - 1)*(L3 - L5)) + L4*\sin(t6) + L5*(\cos(t6) - 1) + \cos(t5)*\sin(t6)*((\cos(t4) - 1)*(L4 - a) + \sin(t4)*(L3 - \\
& L5)) + L4*\sin(t6)*(\cos(t5) - 1)), L5*\cos(t5)*\sin(t7) - L3*\sin(t5)*\sin(t7) - a*(\cos(t7)*\sin(t6) - L3*\cos(t5)*\cos(t6)*\cos(t7) + L5*\cos(t6)*\sin(t5)*\sin(t7) - \\
& L4*\sin(t5)*\sin(t6)*\sin(t7), -\sin(t7)*(L4*\cos(t6) + L5*\sin(t6)), L5*\cos(t7), 0]
\end{aligned}$$

$$\begin{aligned}
& [L4*(\sin(t3)*(\cos(t4)*(\cos(t7)*\sin(t5) - \cos(t5)*\cos(t6)*\sin(t7)) + \sin(t4)*\sin(t6)*\sin(t7)) - \cos(t3)*(\cos(t5)*\cos(t7) + \cos(t6)*\sin(t5)*\sin(t7))) + \\
& (\sin(t2)*(\cos(t3)*(\cos(t6)*\sin(t4) + \cos(t4)*\cos(t5)*\sin(t6)) - \sin(t3)*\sin(t5)*\sin(t6)) + \cos(t2)*(\cos(t4)*\cos(t6) - \cos(t5)*\sin(t4)*\sin(t6)))*((\sin(t4)*(\sin(t5)*\sin(t7) + \\
& \cos(t5)*\cos(t6)*\cos(t7)) + \cos(t4)*\cos(t7)*\sin(t6))*((\cos(t2) - 1)*(L2 + L3 - L5) + L4*\sin(t2)) - \cos(t7)*(L5*\sin(t6) - L4*(\cos(t6) - 1)) + (\sin(t5)*\sin(t7) +
\end{aligned}$$

$$\begin{aligned}
& \cos(t_5) \cos(t_6) \cos(t_7) * ((\cos(t_4) - 1) * (L_4 - a) + \sin(t_4) * (L_3 - L_5)) + (L_4 * (\cos(t_2) - 1) - \sin(t_2) * (L_2 + L_3 - L_5)) * (\cos(t_3) * (\cos(t_4) * (\sin(t_5) * \sin(t_7) + \\
& \cos(t_5) * \cos(t_6) * \cos(t_7)) - \cos(t_7) * \sin(t_4) * \sin(t_6)) + \sin(t_3) * (\cos(t_5) * \sin(t_7) - \cos(t_6) * \cos(t_7) * \sin(t_5))) + L_4 * \sin(t_3) * (\cos(t_5) * \sin(t_7) - \cos(t_6) * \cos(t_7) * \sin(t_5)) - \\
& \cos(t_7) * \sin(t_6) * (\sin(t_4) * (L_4 - a) - (\cos(t_4) - 1) * (L_3 - L_5)) + L_4 * \sin(t_5) * \sin(t_7) + L_4 * (\cos(t_4) * (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) - \\
& \cos(t_7) * \sin(t_4) * \sin(t_6)) * (\cos(t_3) - 1) + L_4 * \cos(t_6) * \cos(t_7) * (\cos(t_5) - 1)) + (\cos(t_2) * (\sin(t_4) * (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) + \cos(t_4) * \cos(t_7) * \sin(t_6)) - \\
& \sin(t_2) * (\cos(t_3) * (\cos(t_4) * (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) - \cos(t_7) * \sin(t_4) * \sin(t_6)) + \sin(t_3) * (\cos(t_5) * \sin(t_7) - \\
& \cos(t_6) * \cos(t_7) * \sin(t_5)))) * (\cos(t_6) * (\sin(t_4) * (L_4 - a) - (\cos(t_4) - 1) * (L_3 - L_5)) + L_4 * \sin(t_6) + (\cos(t_3) * (\cos(t_6) * \sin(t_4) + \cos(t_4) * \cos(t_5) * \sin(t_6)) - \\
& \sin(t_3) * \sin(t_5) * \sin(t_6)) * (\cos(t_2) - 1) - ((\cos(t_2) - 1) * (L_2 + L_3 - L_5) + L_4 * \sin(t_2)) * (\cos(t_4) * \cos(t_6) - \\
& \cos(t_5) * \sin(t_4) * \sin(t_6)) + \cos(t_5) * \sin(t_6) * ((\cos(t_4) - 1) * (L_4 - a) + \sin(t_4) * (L_3 - L_5)) + L_4 * (\cos(t_3) - 1) * (\cos(t_6) * \sin(t_4) + \cos(t_4) * \cos(t_5) * \sin(t_6)) + \\
& L_4 * \sin(t_6) * (\cos(t_5) - 1) - L_4 * \sin(t_3) * \sin(t_5) * \sin(t_6)), (\sin(t_3) * (\cos(t_4) * (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) - \cos(t_7) * \sin(t_4) * \sin(t_6)) - \cos(t_3) * (\cos(t_5) * \sin(t_7) - \\
& \cos(t_6) * \cos(t_7) * \sin(t_5))) * (\cos(t_6) * (\sin(t_4) * (L_4 - a) - (\cos(t_4) - 1) * (L_3 - L_5)) + L_4 * \sin(t_6) + L_5 * (\cos(t_6) - 1) + \cos(t_5) * \sin(t_6) * ((\cos(t_4) - 1) * (L_4 - a) + \sin(t_4) * (L_3 - \\
& L_5)) + L_4 * (\cos(t_3) - 1) * (\cos(t_6) * \sin(t_4) + \cos(t_4) * \cos(t_5) * \sin(t_6)) + L_4 * \sin(t_6) * (\cos(t_5) - 1) - L_4 * \sin(t_3) * \sin(t_5) * \sin(t_6)) + (\cos(t_3) * (\cos(t_4) * (\cos(t_7) * \sin(t_5) - \\
& \cos(t_5) * \cos(t_6) * \sin(t_7)) + \sin(t_4) * \sin(t_6) * \sin(t_7)) + \sin(t_3) * (\cos(t_5) * \cos(t_7) + \cos(t_6) * \sin(t_5) * \sin(t_7))) * (L_2 + L_3 - L_5) - L_4 * (\sin(t_4) * (\cos(t_7) * \sin(t_5) - \\
& \cos(t_5) * \cos(t_6) * \sin(t_7)) - \cos(t_4) * \sin(t_6) * \sin(t_7)) - (\sin(t_3) * (\cos(t_6) * \sin(t_4) + \cos(t_4) * \cos(t_5) * \sin(t_6)) + \cos(t_3) * \sin(t_5) * \sin(t_6)) * ((\sin(t_5) * \sin(t_7) + \\
& \cos(t_5) * \cos(t_6) * \cos(t_7)) * (\cos(t_4) - 1) * (L_4 - a) + \sin(t_4) * (L_3 - L_5)) - \cos(t_7) * (L_5 * \sin(t_6) - L_4 * (\cos(t_6) - 1)) + L_4 * \sin(t_3) * (\cos(t_5) * \sin(t_7) - \cos(t_6) * \cos(t_7) * \sin(t_5)) - \\
& \cos(t_7) * \sin(t_6) * (\sin(t_4) * (L_4 - a) - (\cos(t_4) - 1) * (L_3 - L_5)) + L_4 * \sin(t_5) * \sin(t_7) + L_4 * (\cos(t_4) * (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) - \\
& \cos(t_7) * \sin(t_4) * \sin(t_6)) * (\cos(t_3) - 1) + L_4 * \cos(t_6) * \cos(t_7) * (\cos(t_5) - 1)), (\cos(t_4) * \cos(t_6) - \cos(t_5) * \sin(t_4) * \sin(t_6)) * ((\sin(t_5) * \sin(t_7) + \\
& \cos(t_5) * \cos(t_6) * \cos(t_7)) * ((\cos(t_4) - 1) * (L_4 - a) + \sin(t_4) * (L_3 - L_5)) - \cos(t_7) * (L_5 * \sin(t_6) - L_4 * (\cos(t_6) - 1)) - \cos(t_7) * \sin(t_6) * (\sin(t_4) * (L_4 - a) - (\cos(t_4) - 1) * (L_3 - \\
& L_5)) + L_4 * \sin(t_5) * \sin(t_7) + L_4 * \cos(t_6) * \cos(t_7) * (\cos(t_5) - 1)) - L_4 * (\cos(t_5) * \cos(t_7) + \cos(t_6) * \sin(t_5) * \sin(t_7)) + (\sin(t_4) * (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) + \\
& \cos(t_4) * \cos(t_7) * \sin(t_6)) * (\cos(t_6) * (\sin(t_4) * (L_4 - a) - (\cos(t_4) - 1) * (L_3 - L_5)) + L_4 * \sin(t_6) + L_5 * (\cos(t_6) - 1) + \cos(t_5) * \sin(t_6) * ((\cos(t_4) - 1) * (L_4 - a) + \sin(t_4) * (L_3 - \\
& L_5)) + L_4 * \sin(t_6) * (\cos(t_5) - 1)), a * \sin(t_6) * \sin(t_7) - L_5 * \cos(t_5) * \sin(t_7) - L_3 * \cos(t_7) * \sin(t_5) + L_3 * \cos(t_5) * \cos(t_6) * \sin(t_7) + L_5 * \cos(t_6) * \cos(t_7) * \sin(t_5) - \\
& L_4 * \cos(t_7) * \sin(t_5) * \sin(t_6), -\cos(t_7) * (L_4 * \cos(t_6) + L_5 * \sin(t_6)), -L_5 * \sin(t_7), 0]
\end{aligned}$$

$$\begin{aligned}
& [(\cos(t_2) * (\sin(t_4) * (\cos(t_7) * \sin(t_5) - \cos(t_5) * \cos(t_6) * \sin(t_7)) - \cos(t_4) * \sin(t_6) * \sin(t_7)) - \sin(t_2) * (\cos(t_3) * (\cos(t_4) * (\cos(t_7) * \sin(t_5) - \cos(t_5) * \cos(t_6) * \sin(t_7)) + \\
& \sin(t_4) * \sin(t_6) * \sin(t_7)) + \sin(t_3) * (\cos(t_5) * \cos(t_7) + \cos(t_6) * \sin(t_5) * \sin(t_7))) * ((\sin(t_4) * (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) + \cos(t_4) * \cos(t_7) * \sin(t_6)) * ((\cos(t_2) \\
& - 1) * (L_2 + L_3 - L_5) + L_4 * \sin(t_2)) - \cos(t_7) * (L_5 * \sin(t_6) - L_4 * (\cos(t_6) - 1)) + (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) * ((\cos(t_4) - 1) * (L_4 - a) + \sin(t_4) * (L_3 - L_5)) + \\
& (L_4 * (\cos(t_2) - 1) - \sin(t_2) * (L_2 + L_3 - L_5)) * (\cos(t_3) * (\cos(t_4) * (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) - \cos(t_7) * \sin(t_4) * \sin(t_6)) + \sin(t_3) * (\cos(t_5) * \sin(t_7) - \\
& \cos(t_6) * \cos(t_7) * \sin(t_5))) + L_4 * \sin(t_3) * (\cos(t_5) * \sin(t_7) - \cos(t_6) * \cos(t_7) * \sin(t_5)) - \cos(t_7) * \sin(t_6) * (\sin(t_4) * (L_4 - a) - (\cos(t_4) - 1) * (L_3 - L_5)) + L_4 * \sin(t_5) * \sin(t_7) + \\
& L_4 * (\cos(t_4) * (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) - \cos(t_7) * \sin(t_4) * \sin(t_6)) * (\cos(t_3) - 1) + L_4 * \cos(t_6) * \cos(t_7) * (\cos(t_5) - 1)) + L_4 * (\sin(t_3) * (\cos(t_6) * \sin(t_4) + \\
& \cos(t_4) * \cos(t_5) * \sin(t_6)) + \cos(t_3) * \sin(t_5) * \sin(t_6)) - (\cos(t_2) * (\sin(t_4) * (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) + \cos(t_4) * \cos(t_7) * \sin(t_6)) - \\
& \sin(t_2) * (\cos(t_3) * (\cos(t_4) * (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) - \cos(t_7) * \sin(t_4) * \sin(t_6)) + \sin(t_3) * (\cos(t_5) * \sin(t_7) - \\
& \cos(t_6) * \cos(t_7) * \sin(t_5)))) * (\sin(t_7) * (L_5 * \sin(t_6) - L_4 * (\cos(t_6) - 1)) + (\sin(t_4) * (\cos(t_7) * \sin(t_5) - \cos(t_5) * \cos(t_6) * \sin(t_7)) - \cos(t_4) * \sin(t_6) * \sin(t_7)) * ((\cos(t_2) - 1) * (L_2 + \\
& L_3 - L_5) + L_4 * \sin(t_2)) + (\cos(t_5) * \cos(t_6) * \sin(t_5) - \cos(t_5) * \cos(t_6) * \sin(t_7)) * ((\cos(t_4) - 1) * (L_4 - a) + \sin(t_4) * (L_3 - L_5)) + (L_4 * (\cos(t_2) - 1) - \sin(t_2) * (L_2 + L_3 - \\
& L_5)) * (\cos(t_3) * (\cos(t_4) * (\cos(t_7) * \sin(t_5) - \cos(t_5) * \cos(t_6) * \sin(t_7)) + \sin(t_4) * \sin(t_6) * \sin(t_7)) + \sin(t_3) * (\cos(t_5) * \cos(t_7) + \cos(t_6) * \sin(t_5) * \sin(t_7))) + \\
& L_4 * (\cos(t_4) * (\cos(t_7) * \sin(t_5) - \cos(t_5) * \cos(t_6) * \sin(t_7)) + \sin(t_4) * \sin(t_6) * \sin(t_7)) * (\cos(t_3) - 1) + L_4 * \sin(t_3) * (\cos(t_5) * \cos(t_7) + \cos(t_6) * \sin(t_5) * \sin(t_7)) + \\
& \sin(t_6) * \sin(t_7) * (\sin(t_4) * (L_4 - a) - (\cos(t_4) - 1) * (L_3 - L_5)) + L_4 * \cos(t_7) * \sin(t_5) - L_4 * \cos(t_6) * \sin(t_7) * (\cos(t_5) - 1)), L_4 * (\cos(t_4) * \cos(t_6) - \cos(t_5) * \sin(t_4) * \sin(t_6)) + \\
& (\cos(t_3) * (\cos(t_6) * \sin(t_4) + \cos(t_4) * \cos(t_5) * \sin(t_6)) - \sin(t_3) * \sin(t_5) * \sin(t_6)) * (L_2 + L_3 - L_5) + (\sin(t_3) * (\cos(t_4) * (\cos(t_7) * \sin(t_5) - \cos(t_5) * \cos(t_6) * \sin(t_7)) + \\
& \sin(t_4) * \sin(t_6) * \sin(t_7)) - \cos(t_3) * (\cos(t_5) * \cos(t_7) + \cos(t_6) * \sin(t_5) * \sin(t_7))) * ((\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) * ((\cos(t_4) - 1) * (L_4 - a) + \sin(t_4) * (L_3 - L_5)) - \\
& \cos(t_7) * (L_5 * \sin(t_6) - L_4 * (\cos(t_6) - 1)) + L_4 * \sin(t_3) * (\cos(t_5) * \sin(t_7) - \cos(t_6) * \cos(t_7) * \sin(t_5)) - \cos(t_7) * \sin(t_6) * (\sin(t_4) * (L_4 - a) - (\cos(t_4) - 1) * (L_3 - L_5)) + \\
& L_4 * \sin(t_5) * \sin(t_7) + L_4 * (\cos(t_4) * (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) - \cos(t_7) * \sin(t_4) * \sin(t_6)) * (\cos(t_3) - 1) + L_4 * \cos(t_6) * \cos(t_7) * (\cos(t_5) - 1)) - \\
& (\sin(t_3) * (\cos(t_4) * (\sin(t_5) * \sin(t_7) + \cos(t_5) * \cos(t_6) * \cos(t_7)) - \cos(t_7) * \sin(t_4) * \sin(t_6)) - \cos(t_3) * (\cos(t_5) * \sin(t_7) - \cos(t_6) * \cos(t_7) * \sin(t_5))) * (\sin(t_7) * (L_5 * \sin(t_6) - \\
& L_4 * (\cos(t_6) - 1)) + (\cos(t_7) * \sin(t_5) - \cos(t_5) * \cos(t_6) * \sin(t_7)) * ((\cos(t_4) - 1) * (L_4 - a) + \sin(t_4) * (L_3 - L_5)) + L_4 * (\cos(t_4) * (\cos(t_7) * \sin(t_5) - \cos(t_5) * \cos(t_6) * \sin(t_7)) + \\
& \sin(t_4) * \sin(t_6) * \sin(t_7)) * (\cos(t_3) - 1) + L_4 * \sin(t_3) * (\cos(t_5) * \cos(t_7) + \cos(t_6) * \sin(t_5) * \sin(t_7)) + \sin(t_6) * \sin(t_7) * (\sin(t_4) * (L_4 - a) - (\cos(t_4) - 1) * (L_3 - L_5)) + \\
& L_4 * \cos(t_7) * \sin(t_5) - L_4 * \cos(t_6) * \sin(t_7) * (\cos(t_5) - 1)), -\sin(t_5) * (L_4 * \sin(t_4) - a * \sin(t_6) + L_3 * \sin(t_4) * \sin(t_6) + a * \cos(t_4) * \sin(t_6)), \\
& a * \cos(t_6) - L_4 * \cos(t_5) - L_3 * \cos(t_5) * \sin(t_6), \quad 0, \quad -L_4, 0]
\end{aligned}$$

e) *J_space.m* and *J_body.m*

Problem Description:

This problem requests the creation of functions to calculate the space and body form jacobians, which have been titled *SpaceJacobian.m* and *BodyJacobian.m*, respectively.

Method for Solution:

This solution saw the implementation of the equations spelled out in section D in code form.

Explanation of Program:

The following function, *SpaceJacobian.m*, was developed to complete this problem.

```

function [Js] = SpaceJacobian(S_mat, thetas, is_symbolic)
%Converts matrix of column vectors corresponding to space frame screw axes
%and a vector of theta values corresponding to each joint and returns the
%space Jacobian
    if nargin < 3
        is_symbolic = false;
    end

    S_bank = cell(1, size(S_mat, 2));
    for i = 1:size(S_mat, 2)
        S_bank{i} = S_mat(:,i);
    end

    %    symbolic variation
    if is_symbolic == true
        Js = sym(zeros(6, size(S_mat,2)));
    else
        Js = zeros(6, size(S_mat,2));
    end

    Js(:,1) = S_bank{1};
    for i = 2:size(S_mat, 2)
        Ti = eye(4);
        for j = 1:i - 1
            Ti = Ti * S2T(S_bank{j}, thetas(j), is_symbolic);
        %            fprintf('column: %i, matrix: %i\n', i, j); % debug stream
        end
        Js(:,i) = Ad(Ti, is_symbolic) * S_bank{i};
    end
end

```

This function takes three inputs: the matrix of screw axes and thetas, as used previously for sections B and E, as well as a boolean *is_symbolic*, which like before, functions to allow the code to solve both problems D and E. The iteratively calculates the columns of the space jacobian, following the equation described in section D.

The following function, *BodyJacobian.m*, was also developed for this problem.

```

function [Jb] = BodyJacobian(B_mat, thetas, is_symbolic)
%Converts matrix of column vectors corresponding to body frame screw axes
%and a vector of theta values corresponding to each joint and returns the
%body Jacobian
    if nargin < 3
        is_symbolic = false;
    end

    B_bank = cell(1, size(B_mat, 2));
    for i = 1:size(B_mat, 2)
        B_bank{i} = B_mat(:,i);
    end

    %    symbolic variation

```

```

if is_symbolic == true
Jb = sym(zeros(6, size(B_mat,2)));
else
Jb = zeros(6, size(B_mat,2));
end

for i = size(B_mat, 2) - 1:-1:1
Ti = eye(4);
for j = size(B_mat,2):-1:i + 1
    Ti = Ti * S2T(-1 * B_bank{j}, thetas(j), is_symbolic);
%         fprintf('column: %i, iteration: %i\n', i, j); % debug stream
end
Jb(:,i) = Ad(Ti, is_symbolic) * B_bank{i};
end
Jb(:,end) = B_bank{end};
end

```

Nearly identical to *SpaceJacobian.m*, *BodyJacobian.m* takes the same inputs (with the slight difference that the matrix of screws are now in the body frame) and outputs a body-frame jacobian.

Answer / Test Cases:

The Jacobian calculating functions were tested extensively with many robot configurations. Some of these test cases are shown below. With each test case, the outputs J_space and J_body are compared by using the identity $V_s = Ad_{T_{sb}}(V_b)$, and J_diff is defined as

$$J_diff = J_space - Ad(FK_solution_space) * J_body$$

If J_diff is zero, that means that the two Jacobians are equivalent, and, because they were calculated independently, suggests that the Jacobians are correct.

Test Case 1: The robot is in the zero configuration, with $Thetas = [0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0]$

The function returns:

Numeric Space Jacobian:

$J_space =$

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	-0.0825	0	0	0

Numeric Body Jacobian:

J_body =

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	-0.0055	0	-0.0880	0

Demonstrating equivalency:

J_diff =

1.0e-15 *

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	-0.0555	0	0.1110	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Note that the scale of J_diff is 10^{-15} , implying that the Jacobians are equivalent.

As demonstrated by J_diff, which is calculated using the adjoint matrix of the transformation matrix between the end effector and base frames of the robot, the calculated Jacobian matrices are equivalent when expressed in the same frame.

Test Case 2: The robot is in the “ready” configuration, and has $\Theta = [0 \ -\pi/4 \ 0 \ -3\pi/4 \ 0 \ \pi/2 \ \pi/4]$

The function returns:

Numeric Space Jacobian:

$J_{\text{space}} =$

0	0	-0.7071	0	1.0000	0	-0.0000
0	1.0000	0	-1.0000	0	-1.0000	0
1.0000	0	0.7071	0	0	0	-1.0000
0	-0.3330	0	0.6148	0	0.6973	0
0	0	-0.2355	0	0.6973	0	0.3069
0	0	0	0.1651	0	-0.2189	0

Numeric Body Jacobian:

$J_{\text{body}} =$

-0.0000	-0.7071	-0.5000	0.7071	0.7071	0.7071	0
0.0000	-0.7071	0.5000	0.7071	-0.7071	0.7071	0
-1.0000	0	-0.7071	0	-0.0000	0	1.0000
-0.2170	0.1819	-0.2821	0.0173	-0.0757	0.0757	0
-0.2170	-0.1819	-0.2821	-0.0173	-0.0757	-0.0757	0
0.0000	0.3069	0.0000	-0.4720	-0.0000	-0.0880	0

Demonstrating equivalency:

$J_{\text{diff}} =$

1.0e-15 *

0.0067	0	0	0	0	0	0
-0.0393	0	-0.0555	0	0	0	0
0	0	0.1110	0	-0.0000	0	0
0.0555	0	0.0971	0	0	0	0
-0.0555	-0.0278	-0.0278	0.0555	0.1110	0	0
-0.0035	0.1110	0.0416	-0.0833	-0.0278	-0.1388	0

Note that the scale of J_{diff} is 10^{-15} , implying that the Jacobians are equivalent.

Test Case 3: The robot has randomly generated Thetas = [4.7418 1.7343 4.2707 4.1161 1.0217 0.7477 3.1313]

The function returns:

Numeric Space Jacobian:

J_space =

0	0.9996	0.0290	0.4316	-0.7623	0.6367	0.4798
0	0.0294	-0.9862	-0.1345	0.4743	0.6719	0.0390
1.0000	0	-0.1628	0.8920	0.4404	0.3784	-0.8765
0	-0.0098	0.3284	-0.2425	-0.3264	-0.4110	0.1081
0	0.3329	0.0097	0.1947	-0.2013	0.4649	-0.1708
0	0	-0.0000	0.1467	-0.3481	-0.1340	0.0516

Numeric Body Jacobian:

J_body =

0.3014	0.5884	0.6912	0.6308	-0.6799	0.0103	0
-0.3753	-0.6501	0.7129	-0.5155	-0.0070	-0.9999	0
-0.8765	0.4807	0.1181	-0.5800	-0.7333	0	1.0000
0.3781	-0.0079	-0.2053	0.1440	-0.0014	-0.1070	0
0.1832	0.0751	0.1458	0.3132	0.1373	-0.0011	0
0.0516	0.1113	0.3211	-0.1217	-0.0000	-0.0880	0

Demonstrating equivalency:

J_diff =

1.0e-15 *

-0.0278	0	-0.0902	-0.1110	0.1110	-0.1110	0
-0.1110	-0.0555	0.3331	-0.0833	-0.1110	-0.2220	0
-0.2220	-0.0278	0.1110	-0.1110	0.0555	-0.1110	0
0.1943	0	-0.1665	0.1110	0.1110	0.1665	-0.0139
0.2220	-0.1110	0.0226	0.1388	0.1943	-0.3331	0
0.0666	-0.2220	-0.0295	-0.1110	0.0555	0.2498	0

Note that the scale of J_diff is 10^{-15} , implying that the Jacobians are equivalent.

f) Singularity Analysis and *singularity.m*

Problem Description:

This problem asks for a function that analytically calculates the singularity configurations of the robot.

Method for Solution:

The robot will be in a singularity configuration any time its Jacobian matrix loses full rank, which will be the case when the determinant of the Jacobian is equal to zero.

$$\det(J) = 0$$

For checking whether a single configuration is at singularity, this is as simple as calculating the Jacobian for that state and taking its determinant. If the robot is redundant or under-defined, and the determinant can not be taken, the determinant of JJ^T can be taken instead with the same result. This is the case for the Panda, our robot of choice.

Explanation of Program:

For this problem, the function *singularity.m* was created.

```
function [is_singular, result] = singularity(J, is_symbolic)
%Determines if robot is at singular configuration
% Takes in symbolic or numeric Jacobian and corresponding is_symbolic
% boolean argument. Outputs a boolean and the result of the calculation.
    if nargin < 2
        is_symbolic = false;
    end

    if is_symbolic == true
        J_simp = simplify(J);
        J_simp_sqrd = simplify(J_simp * J_simp');
        result = simplify(det(J_simp_sqrd));
        is_singular = "Unknown. Need to solve for thetas that lead to
singularity configuration.";
    else
        result = det(J * J');
        if result < 1e-04
            disp('The robot is singular.')
            is_singular = true;
        else
            disp('The robot is not singular.')
            is_singular = false;
        end
    end
end
```

This function takes an input of a Jacobian, as well as information if the Jacobian is numerical or symbolic. If the Jacobian is numerical, the function will calculate the determinant and return true if the robot is singular, false if not. If the jacobian is symbolic, the function will attempt to return the determinant of JJ^T , which would be the representation of every case where the robot would be singular.

Answer / Test Cases:

The singularity function was tested extensively with many robot configurations. Some of these test cases are shown below. Notably, while the function theoretically works for the symbolic case, because the Jacobian contains so many terms, the term JJ^T is too large to calculate the determinant for in a reasonable amount of time.

In general, the robot is in singularity configurations if three or more of its joints axes are coplanar, as is shown in the subsequent examples in this question and in part g), or if it is very near the edge of its workspace, at which point its manipulability will rapidly diminish. In essence, the Panda will be in singularity if any of joints 1, 3, 5, or 7 become collinear, or if joints 2, 4, and 6 become coplanar and parallel. Listed below are many of these singularity configurations for the robot:

1. If at the edge of the workspace

$$\theta = [\theta_1 \varepsilon[0, 2\pi] \quad \theta_2 \varepsilon[0, 2\pi] \quad 0 \quad 0 \quad 0 \quad \frac{3\pi}{4} \quad \theta_7 \varepsilon[0, 2\pi]]^T$$

2. If any of joints 1, 3, 5, 7 become collinear

$$\begin{aligned} \theta &= [\theta_1 \varepsilon[0, 2\pi] \quad 0 \quad \theta_3 \varepsilon[0, 2\pi] \quad 0 \quad \theta_5 \varepsilon[0, 2\pi] \quad \theta_6 \varepsilon[0, 2\pi] \quad \theta_7 \varepsilon[0, 2\pi]]^T \\ \theta &= [\theta_1 \varepsilon[0, 2\pi] \quad 0 \quad \theta_3 \varepsilon[0, 2\pi] \quad 0.2312 \quad \theta_5 \varepsilon[0, 2\pi] \quad \theta_4 + \pi \quad \theta_7 \varepsilon[0, 2\pi]]^T \end{aligned}$$

3. If joints 2, 4, and 6 become coplanar and parallel

$$\theta = [\theta_1 \varepsilon[0, 2\pi] \quad \theta_2 \varepsilon[0, 2\pi] \quad 0 \quad -0.4670 \quad 0 \quad 0 \quad \theta_7 \varepsilon[0, 2\pi]]^T$$

Test Case 1: The robot is in the zero configuration, with $\text{Thetas} = [0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0]$. This configuration finds joints 1 and 5 as collinear, so the function should return singular.

As expected, the function returns “The robot is singular.”

Test Case 2: The robot is in the “ready” configuration, and has $\text{Thetas} = [0 \ -\pi/4 \ 0 \ -3\pi/4 \ 0 \ \pi/2 \ \pi/4]$. There are no collinear joints and no coplanar and parallel joints, so the function should not return singular.

As expected, the function returns “The robot is not singular.”

Test Case 3: The robot is in a specially selected singular configuration, and has $\Theta = [0 \ 0 \ 0 \ ((\text{atan}(.316/.0825)+\text{atan}(.384/.0825))-\pi) \ 0 \ 0 \ 0]$. This configuration should return singular as joints 2 4 and 6 are coplanar and parallel. This configuration is reused in part g).

As expected, the function returns “The robot is singular.”

Forward Kinematics for Test3

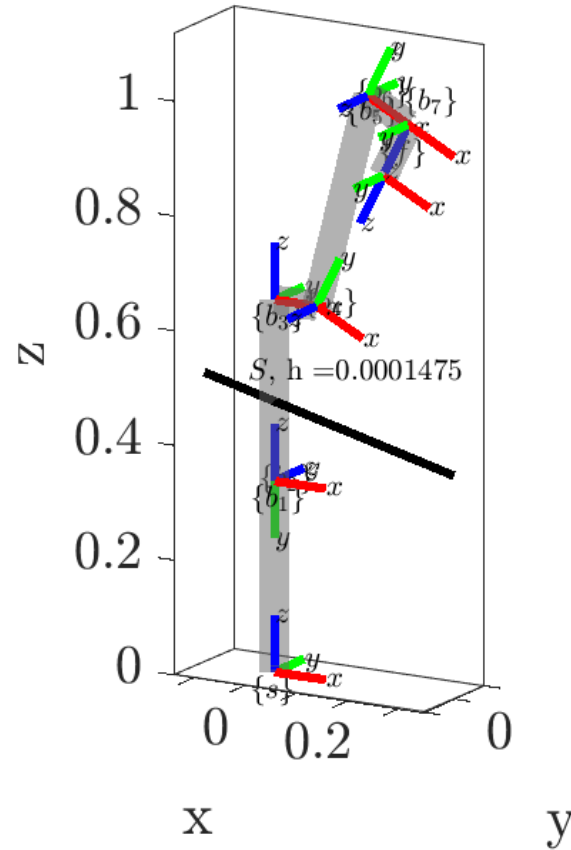


Figure 9. Singular Configuration for test case 3

g) Manipulability Ellipsoids and Jacobian Isotropy, Condition Number, and Volume

Problem Description:

This problem has two key parts. The first is to create a function that plots the manipulability ellipsoids for the linear and angular velocities and their axes at a given Panda configuration. The second part is to calculate the isotropy, condition number, and volumes of the ellipsoids.

Method for Solution:

To calculate the axes for the manipulability ellipsoids, we calculate the positive eigenvectors and eigenvalues of $A = JJ^T > 0$, where J is the first 3 rows of the space jacobian (the ω component) for the angular case, and the last 3 rows (the v component) for the linear case. The eigenvectors become the axes for the manipulability ellipsoids, and the square roots of the eigenvalues become their magnitudes.

To calculate the isotropy, for either the linear or the angular case, the equation is

$$\mu_1 = \frac{\sqrt{\lambda_{\max}(A)}}{\sqrt{\lambda_{\min}(A)}}$$

To calculate the condition number, we simply square the isotropy measure

$$\mu_2 = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}$$

Finally, the volumes of the ellipsoids are calculated by taking the square root of the determinant of A.

Explanation of Program:

For this problem, 5 functions were created. The first two functions, *ellipsoid_plot_linear.m* and *ellipsoid_plot_linear.m*, are identical except for one line, and are shown below.

```
function [eigenvalues, V] = ellipsoid_plot_linear(J, origin, delta,
show_ellipsoids, show_axes, should_create_plot)
    %Calculates and plots the linear manipulability ellipsoid of a Jacobian.
    % Option arguments include: delta, the scale of ellipsoid (default 1,
    recommended 0.5 or less), show_ellipsoids, allowing the user to toggle between
    showing just the axes or including the ellipsoids, show_axes, the same toggle
    in reverse, and should_create_plot, allowing the user to either create a new
    plot or overlay on an existing one.
    if nargin < 3
        delta = 1;
    end
    if nargin < 4
```

```

show_ellipsoids = true;
end
if nargin < 5
show_axes = true;
end
if nargin < 6
should_create_plot = false;
end

J_v = J(4:6, :);
A = J_v * J_v';
[V, D] = eig(A);
eigenvalues = diag(D);

if cross(V(:, 1), V(:, 2)) == -1 * V(:, 3)
V(:,3) = -1 * V(:, 3);
end

if should_create_plot == true
figure
view(3)
box on
hold on
grid on
axis equal;
xlabel('x'), ylabel('y'), zlabel('z');
end

if show_ellipsoids == true
[x, y, z] = ellipsoid(origin(1), origin(2), origin(3), delta *
sqrt(eigenvalues(1)), delta * sqrt(eigenvalues(2)), delta *
sqrt(eigenvalues(3)));
lmesh = surf(x, y, z, 'FaceAlpha', 0.05, 'EdgeAlpha', 0.00,
'FaceColor','cyan','EdgeColor','k');

% checking if eigenvector matrix is I because axis angle rotation
representation is undefined in that case
if not(isequal(V, eye(3)))
[w, theta] = RotationMatrix2AxisAngle(V);
rotate(lmesh, w, theta * 180 / pi, origin);
end
end

if show_axes == true
% Plotting ellipsoid axes and adding scaling label
line('XData', [origin(1) origin(1) + delta * sqrt(eigenvalues(1)) *
V(1,1)], 'YData', [origin(2) origin(2) + delta * sqrt(eigenvalues(1)) *
V(2,1)], ...
'ZData', [origin(3) origin(3) + delta * sqrt(eigenvalues(1)) *
V(3,1)], 'Color','cyan','LineWidth',3);
line('XData', [origin(1) origin(1) - delta * sqrt(eigenvalues(1)) *
V(1,1)], 'YData', [origin(2) origin(2) - delta * sqrt(eigenvalues(1)) *
V(2,1)], ...

```



```

        'ZData', [origin(3) origin(3) - delta *sqrt(eigenvalues(1))*
V(3,1)], 'Color','cyan','LineWidth',3);

        line('XData', [origin(1) origin(1) + delta *sqrt(eigenvalues(2))*
V(1,2)], 'YData', [origin(2) origin(2) + delta *sqrt(eigenvalues(2))*
V(2,2)], ...
        'ZData', [origin(3) origin(3) + delta *sqrt(eigenvalues(2))*
V(3,2)], 'Color','cyan','LineWidth',3);
        line('XData', [origin(1) origin(1) - delta *sqrt(eigenvalues(2))*
V(1,2)], 'YData', [origin(2) origin(2) - delta *sqrt(eigenvalues(2))*
V(2,2)], ...
        'ZData', [origin(3) origin(3) - delta *sqrt(eigenvalues(2))*
V(3,2)], 'Color','cyan','LineWidth',3);

        line('XData', [origin(1) origin(1) + delta *sqrt(eigenvalues(3))*
V(1,3)], 'YData', [origin(2) origin(2) + delta *sqrt(eigenvalues(3))*
V(2,3)], ...
        'ZData', [origin(3) origin(3) + delta *sqrt(eigenvalues(3))*
V(3,3)], 'Color','cyan','LineWidth',3);
        line('XData', [origin(1) origin(1) - delta *sqrt(eigenvalues(3))*
V(1,3)], 'YData', [origin(2) origin(2) - delta *sqrt(eigenvalues(3))*
V(2,3)], ...
        'ZData', [origin(3) origin(3) - delta *sqrt(eigenvalues(3))*
V(3,3)], 'Color','cyan','LineWidth',3);
    end
    text(origin(1) + 0.5*delta, 0, origin(3) + 0.5*delta,
strcat('$\delta_{lin}$ = ', string(delta)), 'Color','cyan');
end

```

```

function [eigenvalues, V] = ellipsoid_plot_angular(J, origin, delta,
show_ellipsoids, show_axes, should_create_plot)
%Calculates and plots the angular manipulability ellipsoid of a Jacobian.
% Option arguments include: delta, the scale of ellipsoid (default 1,
recommended 0.5 or less), show_ellipsoids, allowing the user to toggle between
showing just the axes or including the ellipsoids, show_axes, the same toggle
in reverse, and should_create_plot, allowing the user to either create a new
plot or overlay on an existing one.
    if nargin < 3
        delta = 1;
    end
    if nargin < 4
        show_ellipsoids = true;
    end
    if nargin < 5
        show_axes = true;
    end
    if nargin < 6
        should_create_plot = false;
    end

    J_w = J(1:3, :);

```

```

A = J_w * J_w';
[V, D] = eig(A);
eigenvalues = diag(D);

if cross(V(:, 1), V(:, 2)) == -1 * V(:, 3)
V(:, 3) = -1 * V(:, 3);
end

if should_create_plot == true
figure
view(3)
box on
hold on
grid on
axis equal;
xlabel('x'), ylabel('y'), zlabel('z');
end

if show_ellipsoids == true
[x, y, z] = ellipsoid(origin(1), origin(2), origin(3), delta *
sqrt(eigenvalues(1)), delta * sqrt(eigenvalues(2)), delta *
sqrt(eigenvalues(3)));
hMesh = surf(x, y, z, 'FaceAlpha', 0.05, 'EdgeAlpha', 0.00,
'FaceColor', 'magenta', 'EdgeColor', 'k');

    % checking if eigenvector matrix is I because axis angle rotation
representation is undefined in that case
    if not(isequal(V, eye(3)))
        [w, theta] = RotationMatrix2AxisAngle(V);
        rotate(hMesh, w, theta * 180 / pi, origin);
    end
end

if show_axes == true
    % Plotting ellipsoid axes and adding scaling label
    line('XData', [origin(1) origin(1) + delta * sqrt(eigenvalues(1))] *
V(1,1)], 'YData', [origin(2) origin(2) + delta * sqrt(eigenvalues(1))] *
V(2,1)], ...
        'ZData', [origin(3) origin(3) + delta * sqrt(eigenvalues(1))] *
V(3,1)], 'Color', 'magenta', 'LineWidth', 3);
    line('XData', [origin(1) origin(1) - delta * sqrt(eigenvalues(1))] *
V(1,1)], 'YData', [origin(2) origin(2) - delta * sqrt(eigenvalues(1))] *
V(2,1)], ...
        'ZData', [origin(3) origin(3) - delta * sqrt(eigenvalues(1))] *
V(3,1)], 'Color', 'magenta', 'LineWidth', 3);

    line('XData', [origin(1) origin(1) + delta * sqrt(eigenvalues(2))] *
V(1,2)], 'YData', [origin(2) origin(2) + delta * sqrt(eigenvalues(2))] *
V(2,2)], ...
        'ZData', [origin(3) origin(3) + delta * sqrt(eigenvalues(2))] *
V(3,2)], 'Color', 'magenta', 'LineWidth', 3);
    line('XData', [origin(1) origin(1) - delta * sqrt(eigenvalues(2))] *
V(1,2)], 'YData', [origin(2) origin(2) - delta * sqrt(eigenvalues(2))] *

```

```

V(2,2)],...
    'ZData', [origin(3) origin(3) - delta *sqrt(eigenvalues(2))*
V(3,2)], 'Color','magenta','LineWidth',3);

    line('XData', [origin(1) origin(1) + delta *sqrt(eigenvalues(3))*
V(1,3)], 'YData', [origin(2) origin(2) + delta *sqrt(eigenvalues(3))*
V(2,3)],...
    'ZData', [origin(3) origin(3) + delta *sqrt(eigenvalues(3))*
V(3,3)], 'Color','magenta','LineWidth',3);
    line('XData', [origin(1) origin(1) - delta *sqrt(eigenvalues(3))*
V(1,3)], 'YData', [origin(2) origin(2) - delta *sqrt(eigenvalues(3))*
V(2,3)],...
    'ZData', [origin(3) origin(3) - delta *sqrt(eigenvalues(3))*
V(3,3)], 'Color','magenta','LineWidth',3);
end

text(origin(1) + 0.5*delta, 0, origin(3) + 0.5*delta,
strcat('$\Delta_{ang}$ = ', string(delta)), 'Color','magenta');
end

```

These functions each take 6 inputs. The first two are required inputs; the Jacobian and the ellipsoid's origin (i.e. the end effector coordinates), which are used to plot the ellipsoid as described above. The next four inputs are optional. The first is delta, which allows the user to scale down the ellipsoids for visibility. Show_ellipsoids and show_axes are used to toggle the visibility of the ellipsoids and their axes, respectively. Finally, should_create_plot is used to determine whether a new figure is used. The only difference between these two functions is in lines 17 and 18, where the linear ellipsoid uses the last 3 rows of the Jacobian for its calculations, and the angular does the opposite.

The next three functions are all short, and have similar implementations.

```

function [mu] = J_isotropy(J, type)
%Calculates the isotropy of a Jacobian. 'type' is either "angular" for the
%isotropy of the angular velocity components or "linear" for the isotropy
%of the linear velocity components.
    J_w = J(1:3, :);
    J_v = J(4:6, :);
    if type == "angular"
        A = J_w * J_w';
    elseif type == "linear"
        A = J_v * J_v';
    end
    [V, D] = eig(A);
    eigenvalues = diag(D);

    mu = sqrt(max(eigenvalues) / min(eigenvalues));
end

```

```

function [condition_number] = J_condition(J, type)
    %Calculates the condition number of a Jacobian. 'type' is either "angular" for
the
    %condition number of the angular velocity components or "linear" for the
    %condition number of the linear velocity components.
    J_w = J(1:3, :);
    J_v = J(4:6, :);
    if type == "angular"
        A = J_w * J_w';
    elseif type == "linear"
        A = J_v * J_v';
    end
    [V, D] = eig(A);
    eigenvalues = diag(D);

    condition_number = max(eigenvalues) / min(eigenvalues);
end

```

```

function [ellipsoid_volume] = J_ellipsoid_volume(J, type)
    %Calculates the volume of a Jacobian manipulability ellipsoid. 'type' is
    %either "angular" for the
    %angular velocity components or "linear" for the linear velocity components.
    J_w = J(1:3, :);
    J_v = J(4:6, :);
    if type == "angular"
        A = J_w * J_w';
    elseif type == "linear"
        A = J_v * J_v';
    end
    ellipsoid_volume = sqrt(det(A));
end

```

These three functions input the robot's jacobian and the type (angular or linear), and output the corresponding isotropy, condition number, and ellipsoid volume using the equations defined above.

Answer / Test Cases:

The manipulability functions were tested extensively with many robot configurations. Some of these test cases are shown below.

Test Case 1: The robot is in the zero configuration, with $\text{Thetas} = [0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0]$

The functions return:

isotropy_angular_space =

Inf

isotropy_linear_space =

```

17.8798
condition_angular_space =
    Inf
condition_linear_space =
    319.6872
ellipsoid_volume_angular_space =
    0
ellipsoid_volume_linear_space =
    0.0079

```

Manipulability Ellipsoids for Test1

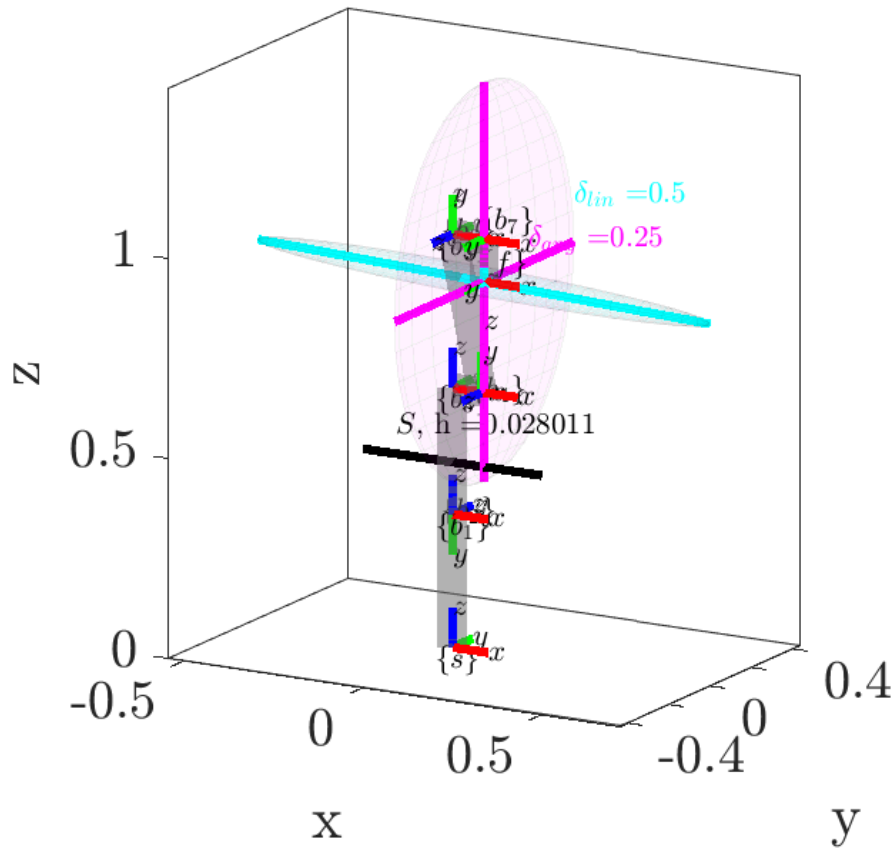


Figure 10. Manipulability ellipsoids for test case 1

In this test case, the robot is at singularity, as its angular velocity manipulability (shown in magenta) ellipsoid has collapsed to 2D. At this configuration, the robot has three coplanar joint

axes, so this singularity is not unexpected. The robot has no ability to generate instantaneous angular velocity about the current x-axis, as all of its joints rotate about either the current y- or z- axes in this configuration, so this result is to be expected. Accordingly, the isotropy and condition number of the angular velocity portions of the Jacobian are infinity, and the volume of the ellipsoid is zero. The linear velocity manipulability ellipsoid (shown in cyan) is nonzero, as the offset in its second-to-last link gives it the ability to generate small linear velocities in the y- and z- directions in this configuration, and its second joint from the base gives it the ability to create much higher instantaneous velocity in the x-direction. Accordingly, the isotropy of the linear ellipsoid is quite high, and its condition number is also fairly high, indicating the robot is near singularity, and the volume is quite small.

Test Case 2: The robot is in the “ready” configuration, and has $\Theta = [0 \ -\pi/4 \ 0 \ -3\pi/4 \ 0 \ \pi/2 \ \pi/4]$

The functions return:

isotropy_angular_space =

1.5233

isotropy_linear_space =

3.6783

condition_angular_space =

2.3204

condition_linear_space =

13.5301

ellipsoid_volume_angular_space =

3.2404

ellipsoid_volume_linear_space =

0.2120

Manipulability Ellipsoids for Test2

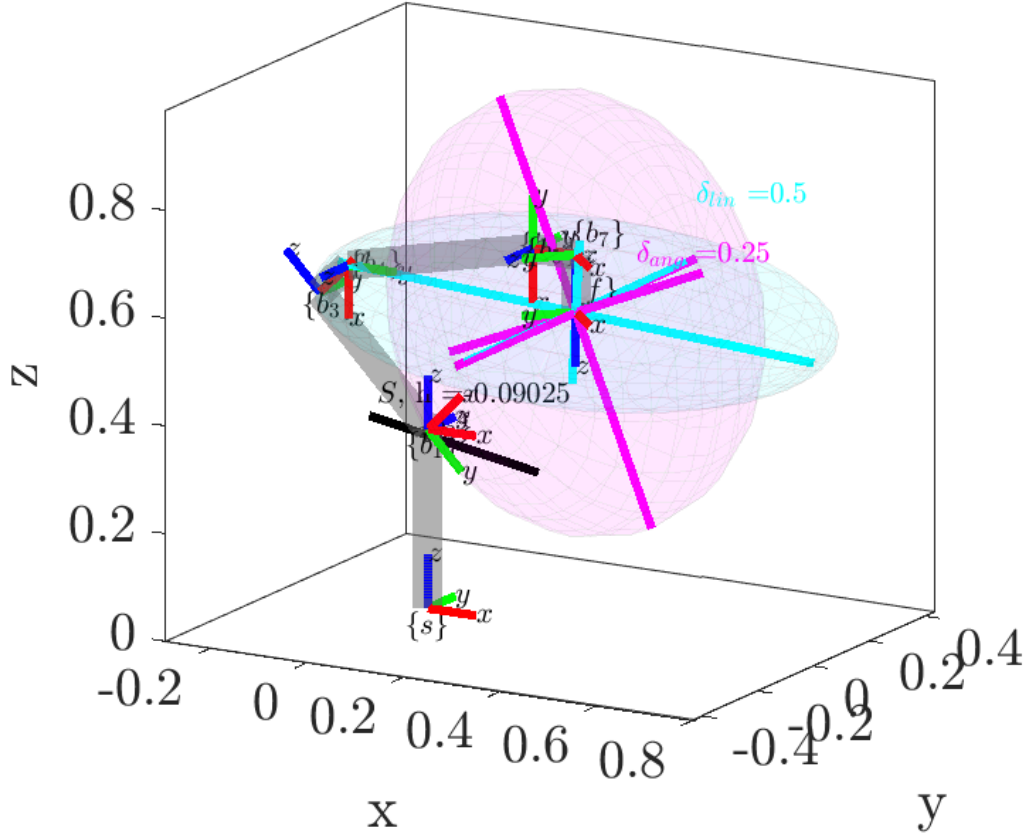


Figure 11. Manipulability ellipsoids for test case 2

In this test case, the end-effector is well within the workspace of the robot and its joints are far from their extreme values, so, as expected, the manipulability ellipsoids are quite large, with much more uniform shaping around each axis. This is reflected in the isotropy values, which are much closer to 1 than in the previous case, and the condition numbers are also much closer to 0, indicating this configuration is further from singularity. The volume of each ellipsoid is also significantly larger, as the robot is much more dexterous and maneuverable in this configuration than in the first test case.

Test Case 3: The robot has $\text{Thetas} = [0 \ 0 \ 0 \ ((\text{atan}(.316/.0825)+\text{atan}(.384/.0825))-\pi) \ 0 \ 0 \ 0]$, which were calculated to place the robot near singularity.

The functions return:

isotropy_angular_space =

1.4345

isotropy_linear_space =

2.9254

condition_angular_space =

2.0578

condition_linear_space =

8.5578

ellipsoid_volume_angular_space =

3.3435

ellipsoid_volume_linear_space =

0.1269

Manipulability Ellipsoids for Test3

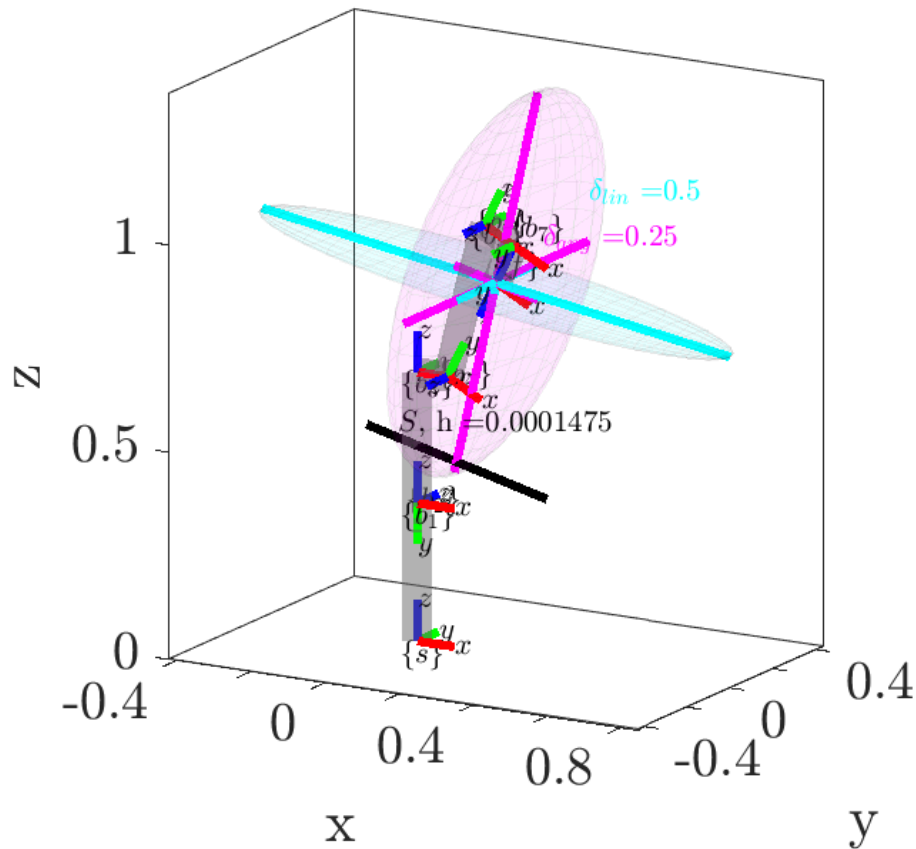


Figure 12. Manipulability ellipsoids for test case 3

In this test case, the robot is nearly at singularity. It has three coplanar joint axes, which has caused the linear velocity manipulability ellipsoid to collapse to nearly 2D. As such, its volume is small and the rate of change of its isotropy and condition number are rapidly increasing as it moves closer to the singularity configuration. The angular velocity ellipsoid is also not far from singularity, as this configuration is close to the known singularity configuration of the first test case.

h) Inverse Kinematics and *J_inverse_kinematics.m*

Problem Description:

This problem requests the creation of a function that uses the iterative numerical inverse kinematics algorithm to control the robot from an arbitrary configuration to a desired configuration.

Method for Solution:

Numerical inverse kinematics revolves around the concept of minimizing the error $g(\theta)$, defined as the difference between the goal state and the robot's current state. Utilizing this error, at each time step we want to solve for $\Delta\theta$ as follows

$$\Delta\theta = J^{-1}(\theta^0)g(\theta)$$

In many cases, and indeed, in our case, the jacobian J is not invertible, so we must also define the Moore-Penrose pseudoinverse Jacobian J^\dagger . When J is fat (redundant, the case of the Panda),

$$J^\dagger = J^T(JJ^T)^{-1}$$

When J is tall (at singularity),

$$J^\dagger = (J^T J)^{-1} J^T$$

To implement numerical inverse kinematics, the desired configuration is define as

$$T_{bd}(\theta^i) = T_{sb}^{-1}(\theta^i)T_{sd}$$

Therefore, the desired motion $g(\theta)$ is solved for using the matrix logarithm as

$$[v_b] = \log(T_{bd}(\theta^i))$$

In implementation we see

$$\theta^{i+1} = \theta^i + J_b^\dagger(\theta^i)v_b$$

i is then incremented until the error measure is sufficiently small.

Explanation of Program:

The following function, `J_inverse_kinematics.m`, was developed to complete this problem.

```
function [theta_0] = J_inverse_kinematics(T_sd, theta_0, M, S_mat, should_plot,
M_intermediates, alpha_0, w_tol, v_tol, fading_memory_size,
fading_memory_tolerance)
    %This function performs iterative numerical inverse kinematics to drive the
    robot from one state to another.
    % Inputs: should_plot defaults to false, and is used to animate the motion.
    M_intermediates is used for plotting the robot's links. alpha_0, w_tol, and
    v_tol are all tolerancing and tuning parameters used to change how quickly the
    robot state converges and how precise it needs to be. Fading_memory_size is
    used to determine how far into the past the fading memory filter looks.
    if nargin < 8
        w_tol = 1e-02;
        v_tol = 1e-02;
```

```

end
if nargin < 7
alpha_0 = 4e-02;
end
if nargin < 6
should_plot = false;
end
if nargin < 10
fading_memory_size = 10;
end
if nargin < 11
fading_memory_tolerance = 1e-03;
end

w_mag = w_tol + 1;
v_mag = v_tol + 1;

[T_0] = FK_space(M, S_mat, theta_0);
x = [];
y = [];
z = [];

previous_delta_thetas = zeros(size(S_mat,2), fading_memory_size);
count = 0;
while w_mag > w_tol || v_mag > v_tol
if should_plot == true
cla;
[T_sb] = FK_space(M, S_mat, theta_0, false, true, M_intermediates,
false);
x = [x, T_sb(1, 4)];
y = [y, T_sb(2, 4)];
z = [z, T_sb(3, 4)];
plot3(x, y, z, 'k')
plotFrame_T(T_0, 'start', 0.1)
plotFrame_T(T_sd, 'goal', 0.1)
pause(0.03);
else
[T_sb] = FK_space(M, S_mat, theta_0);
end

J_s = SpaceJacobian(S_mat, theta_0);
J_b = Ad(inv(T_sb)) * J_s;

J_diff = J_s - Ad(T_sb) * J_b;

T_bd = T_sb \ T_sd;
nu_b = T2S(T_bd);
delta_theta = alpha_0 * J_dagger(J_b) * nu_b;
theta_0 = theta_0 + delta_theta';

count = count + 1;

% Fading memory implementation

```

```

    if count <= fading_memory_size
        previous_delta_thetas(:, count) = delta_theta;
    else
        for i = 2:fading_memory_size
            previous_delta_thetas(:, i - 1) = previous_delta_thetas(:, i);
        end
        previous_delta_thetas(:, fading_memory_size) = delta_theta;

        % now checking to see if average of previous thetas
        means = zeros(size(S_mat,2), 1);
        for i = 1:size(S_mat,2)
            means(i) = mean(previous_delta_thetas(i, :));
        end
        if max(means) < fading_memory_tolerance
            w_mag = w_tol;
            v_mag = v_tol;
        end
    end
    end
    count
end

```

This function makes use of *J_dagger.m*, which is shown below:

```

function [J_dagger] = J_dagger(J)
%Calculates the Moore-Penrose pseudoinverse of a Jacobian
% If pseudoinverse is unnecessary, will return proper inverse of input
% Jacobian
    m = size(J, 1);
    n = size(J, 2);

    %   if J is square -> true inverse
    if n == m
        J_dagger = inv(J);
    %   redundant or 'fat' case -> right inverse
    elseif n > m
        J_dagger = J' / (J * J');
    %   'tall' case -> left inverse
    elseif n < m
        J_dagger = (J' * J) \ J';
    end
end

```

J_dagger calculates either the left or right matrix pseudo-inverse, depending on whether the input Jacobian is tall or fat (ie. the robot is redundant or not), or returns the true inverse if the Jacobian is square.

J_inverse_kinematics takes inputs of a current and desired position, as well as the set of Screws and end effector matrix M that define the robot. Using these, it calculates the transformations T_{sb} and T_{sd} , uses those to calculate T_{bd} , the error v_b , and iterates the robot pose using θ^{i+1} . The rate at which it converges can be tuned using α_0 , which is a parameter analogous to the learning rate or step size parameter used in gradient descent-style optimization, and seems to have the best results at about $4e-2$ from our trials. After nearly converging to the desired end-effector pose, we often observed oscillations around the desired pose that took a very long time to converge. This is also a common problem in local optimization problems, and in the control and machine learning fields, gain/learning rate scheduling methods are often incorporated to help the system converge when this type of behavior is observed. However, early results showed that the different scheduling approaches we tested did not generalize well to new configurations, which is also a common issue with non-adaptive forecasting. Hence, we opted to instead use a basic fading memory stopping condition that, once the average change in theta over a set period of measurements is approximately zero, will cause the algorithm to stop running and prevent this oscillatory behavior from continuing.

Answer / Test Cases:

The iterative numerical inverse kinematic functions were tested extensively with many robot configurations. Some of these test cases are shown below. To view the full gifs, explore the “GIFS” folder of the submission zip.

Note that these test cases are reused for each of the IK problems. They were selected because they represent realistic motions of a robotic manipulator, with several of them taking the manipulator close to or through singularity situations, allowing the differences in performance of the methods to be easily compared.

Test case 1: The robot moves from pose obtained by setting arbitrary $\theta_0 = [2\pi/3 \ \pi/6 \ 0 \ -\pi/4 \ \pi/4 \ -\pi/2 \ 0]$ to pose obtained by setting $\theta_d = [0 \ \pi/2 \ 0 \ 0 \ 0 \ 0 \ 0]$:

Jacobian Numerical IK Test1

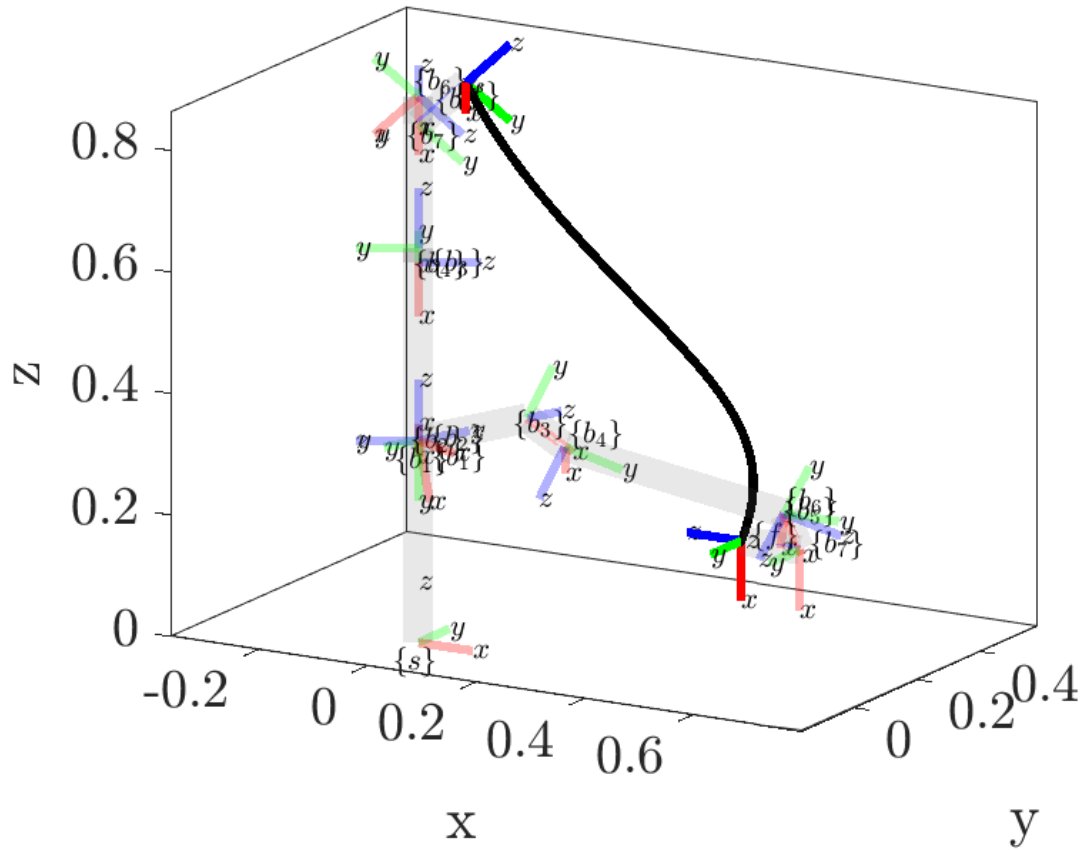


Figure 13: Iterative numerical inverse kinematics test case 1

Test case 2: The robot performs a “pick and place” motion from the “ready” configuration $\theta_0 = [0 \ -\pi/4 \ 0 \ -3\pi/4 \ 0 \ \pi/2 \ \pi/4]$ to the world cartesian coordinate $(0.3, 0.3, 0)$ with a twist of the end effector.

Jacobian Numerical IK Test2

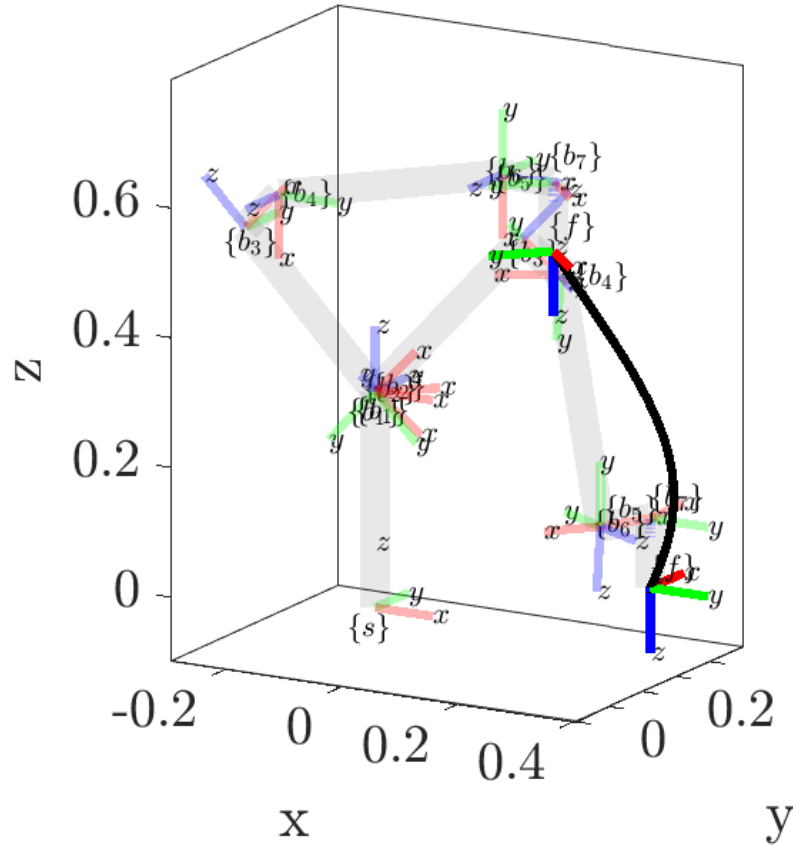


Figure 14: Iterative numerical inverse kinematics test case 2

Test case 3: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

Jacobian Numerical IK Test3

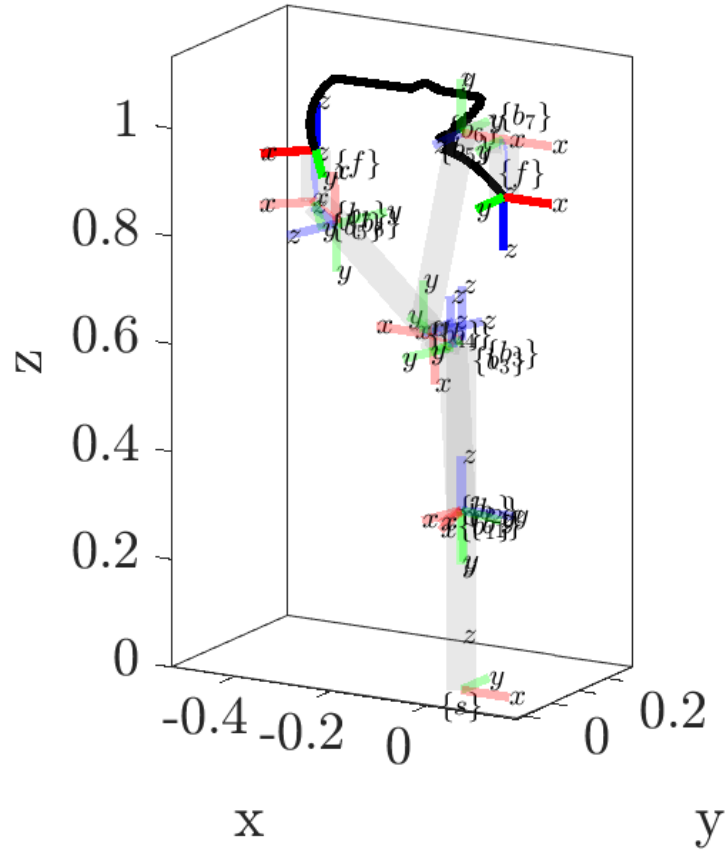


Figure 15: Iterative numerical inverse kinematics test case 3

Test case 4: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

Jacobian Numerical IK Test4

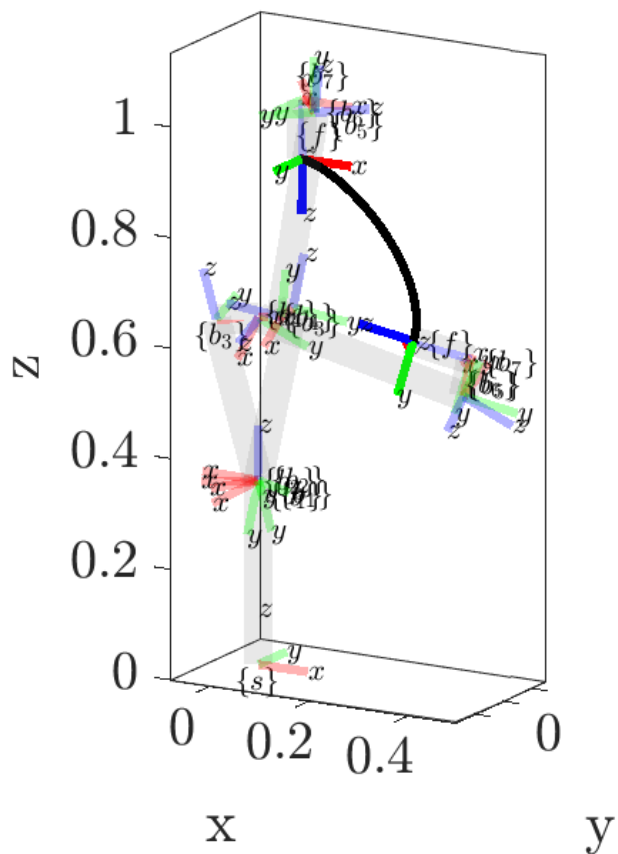


Figure 16: Iterative numerical inverse kinematics test case 4

Test case 5: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

Jacobian Numerical IK Test5

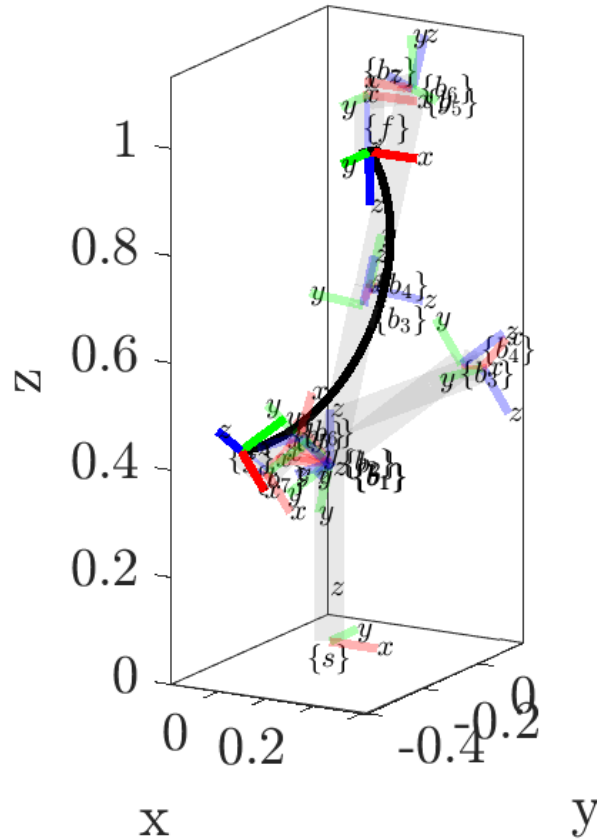


Figure 17: Iterative numerical inverse kinematics test case 5

In these tests, the robot generally follows a smooth, curved path between the starting and goal poses. The obvious exception to this is test case 3, which features a highly irregular and discontinuous path to the goal pose. This is because the robot is moving through singularity to reach this goal, and there are no considerations for these types of situations in this initial algorithm.

This performance will be compared to the performance of the remaining algorithms in the following sections. The reader is referred to the GIFs in the included files with this report to better understand the performance of these algorithms.

i) Jacobian Transpose Inverse Kinematics and *J_transpose_kinematics.m*

Problem Description:

This problem requests the creation of a function that uses the jacobian transpose algorithm to control the robot from an arbitrary configuration to a desired configuration.

Method for Solution:

This problem is functionally similar to the numerical inverse kinematics problem. The two main differences are that it uses the jacobian transpose J^T instead of the Moore-Penrose pseudoinverse Jacobian J^\dagger , and instead of a constant alpha, the jacobian transpose algorithm utilizes a symmetric positive definite matrix K . The error $g(\theta)$ can be defined identically to that used in the numerical inverse kinematic method. The joint updates $\Delta\theta$ are then defined as

$$\Delta\theta = J_b^T K g(\theta) = J_b^T K v_b$$

Explanation of Program:

The following function, *J_transpose_kinematics.m*, was developed to complete this problem.

```
function [theta_0] = J_transpose_kinematics(T_sd, theta_0, M, S_mat,
should_plot, M_intermediates, alpha_0, w_tol, v_tol, fading_memory_size,
fading_memory_tolerance)
    %This function performs iterative numerical inverse kinematics to drive the
    robot from one state to another.
    % Inputs: should_plot defaults to false, and is used to animate the
    % motion. M_intermediates is used for plotting the robot's links.
    % alpha_0, w_tol, and v_tol are all tolerancing and tuning parameters
    % used to change how quickly the robot state converges and how precise it
    % needs to be. Fading_memory_size is used to determine how far into the
    % past the fading memory filter looks.
    if nargin < 8
        w_tol = 1e-02;
        v_tol = 1e-02;
    end
    if nargin < 7
        alpha_0 = 4e-02;
    end
    if nargin < 6
        should_plot = false;
    end
    if nargin < 10
        fading_memory_size = 10;
    end
    if nargin < 11
```

```

        fading_memory_tolerance = 1e-03;
    end

    w_mag = w_tol + 1;
    v_mag = v_tol + 1;

    [T_0] = FK_space(M, S_mat, theta_0);
    x = [];
    y = [];
    z = [];
    previous_delta_thetas = zeros(size(S_mat,2), fading_memory_size);
    count = 0;
    while w_mag > w_tol || v_mag > v_tol
        if should_plot == true
            cla;
            [T_sb] = FK_space(M, S_mat, theta_0, false, true, M_intermediates,
false);
            x = [x, T_sb(1, 4)];
            y = [y, T_sb(2, 4)];
            z = [z, T_sb(3, 4)];
            plot3(x, y, z, ':k')
            plotFrame_T(T_0, 'start', 0.1)
            plotFrame_T(T_sd, 'goal', 0.1)
            pause(0.03);
        else
            [T_sb] = FK_space(M, S_mat, theta_0);
        end

        J_s = SpaceJacobian(S_mat, theta_0);
        J_b = Ad(inv(T_sb)) * J_s;

        T_bd = T_sb \ T_sd;
        nu_b = T2S(T_bd);
        K = eye(6)*alpha_0;
        delta_theta = (J_b') * K * nu_b;
        theta_0 = theta_0 + delta_theta';

        count = count + 1;

        % Fading memory implementation
        if count <= fading_memory_size
            previous_delta_thetas(:, count) = delta_theta;
        else
            for i = 2:fading_memory_size
                previous_delta_thetas(:, i - 1) = previous_delta_thetas(:, i);
            end
            previous_delta_thetas(:, fading_memory_size) = delta_theta;

            % now checking to see if average of previous thetas
            means = zeros(size(S_mat,2), 1);
            for i = 1:size(S_mat,2)
                means(i) = mean(previous_delta_thetas(i, :));
            end
        end
    end

```

```

        if max(means) < fading_memory_tolerance
            w_mag = w_tol;
            v_mag = v_tol;
        end
    end
end
count
end

```

This function takes inputs of a current and desired position, as well as the set of Screws and end effector matrix M that define the robot. Using these, it calculates the transformations T_{sb} and T_{sd} , uses those to calculate T_{bd} , the error v_b , and iterates the robot pose using θ^{i+1} . The rate at which it converges can be tuned using K , which works well at $I_6 * 4e-2$. Other options for symmetric, positive-definite matrices were tested, though we did not find an alternative to K that performed better than this diagonal scalar matrix, which is effectively equivalent to

$$\Delta\theta = kJ_b^T v_b.$$

Answer / Test Cases:

The Jacobian transpose inverse kinematic functions were tested extensively with many robot configurations. Some of these test cases are shown below. To view the full gifs, explore the “GIFS” folder of the submission zip.

Note that these test cases are reused for each of the IK problems. They were selected because they represent realistic motions of a robotic manipulator, with several of them taking the manipulator close to or through singularity situations, allowing the differences in performance of the methods to be easily compared.

Test case 1: The robot moves from pose obtained by setting arbitrary $\theta_0 = [2\pi/3 \ \pi/6 \ 0 \ -\pi/4 \ \pi/4 \ -\pi/2 \ 0]$ to pose obtained by setting $\theta_d = [0 \ \pi/2 \ 0 \ 0 \ 0 \ 0 \ 0]$:

Jacobian Transpose IK Test1

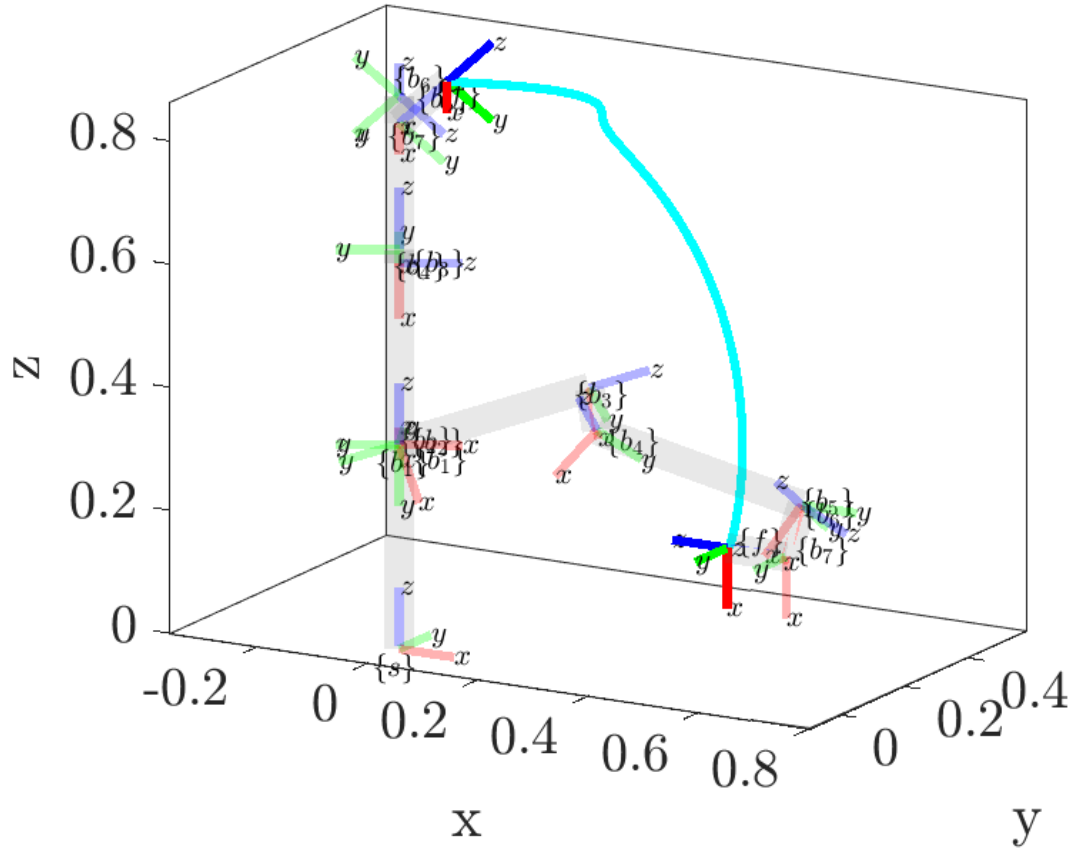


Figure 18: Jacobian transpose inverse kinematics test case 1

Test case 2: The robot performs a “pick and place” motion from the “ready” configuration $\theta_0 = [0 \ -\pi/4 \ 0 \ -3\pi/4 \ 0 \ \pi/2 \ \pi/4]$ to the world cartesian coordinate $(0.3, 0.3, 0)$ with a twist of the end effector.

Jacobian Transpose IK Test2

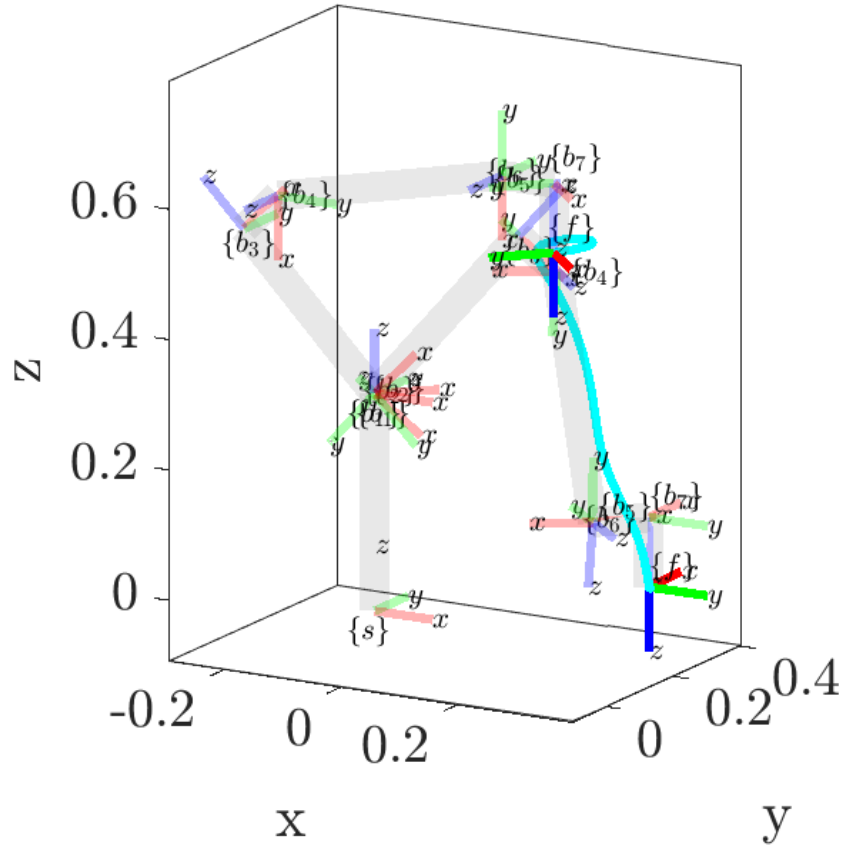


Figure 19: Jacobian transpose inverse kinematics test case 2

Test case 3: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

Jacobian Transpose IK Test3

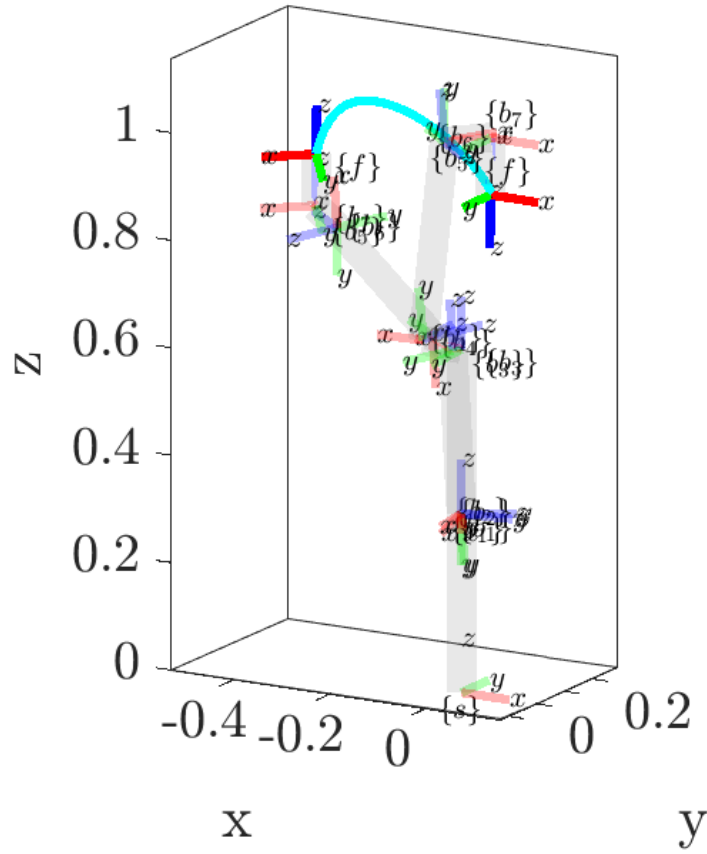


Figure 20: Jacobian transpose inverse kinematics test case 3

Test case 4: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

Jacobian Transpose IK Test4

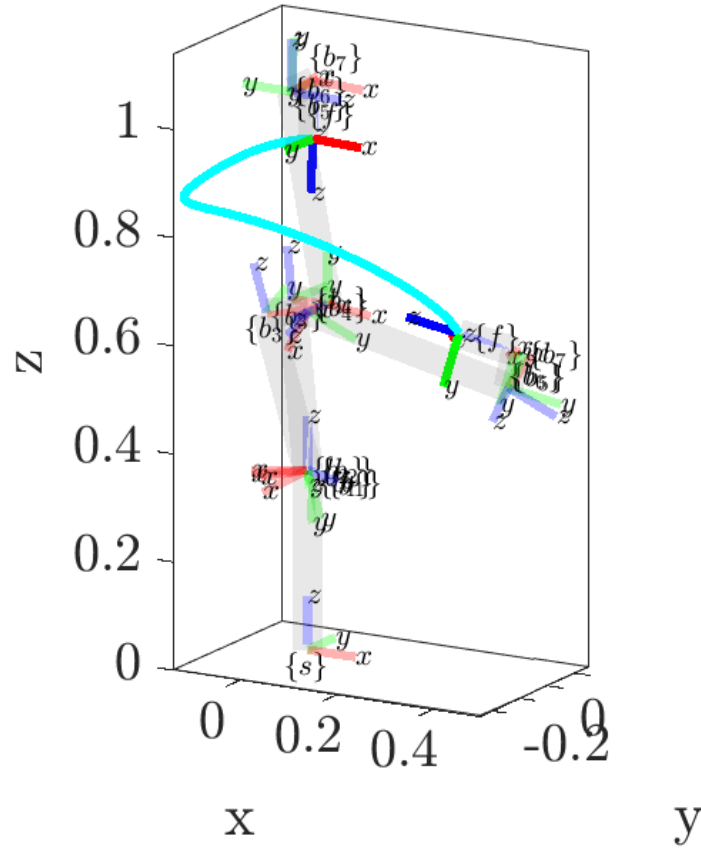


Figure 21: Jacobian transpose inverse kinematics test case 4

Test case 5: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

Jacobian Transpose IK Test5

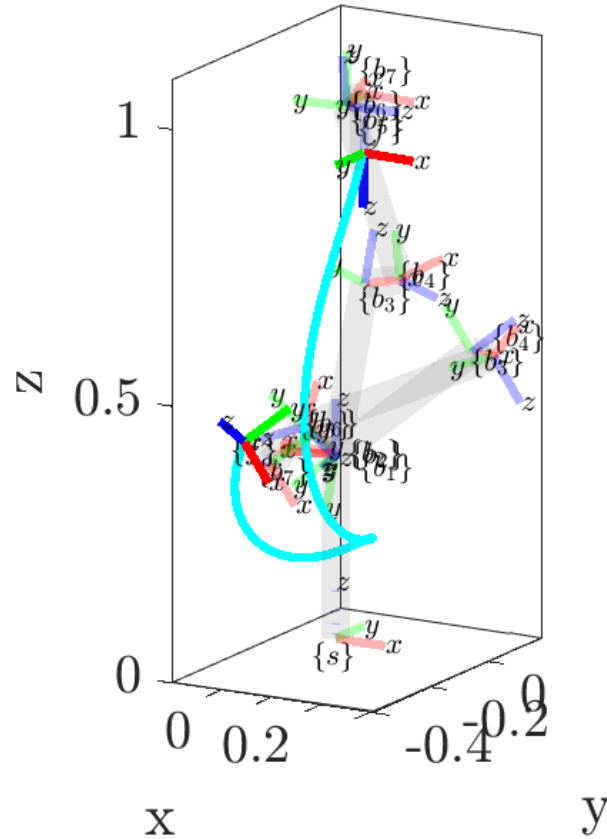


Figure 22: Jacobian transpose inverse kinematics test case 5

As shown in these results, the robot follows a smooth, continuous path for each of these results. This is not unlike the results from part h), with the exception of case 3, in which the robot, unlike using the original IK algorithm, was able to find a smooth, continuous path to achieve the desired pose. One notable difference between the performance of the transpose algorithm and the original algorithm is that the transpose algorithm tends to produce paths that are less direct to the desired pose, with many of these cases featuring significant deviations from the more spatially optimal paths taken by the original algorithm. As a result, the transpose algorithm also took longer to converge, especially when very near the desired pose.

This performance will be compared to the performance of the remaining algorithms in the following sections. The reader is referred to the GIFs in the included files with this report to better understand the performance of these algorithms.

j) Redundancy Resolution Inverse Kinematics and *redundancy_resolution.m*

Problem Description:

This problem requests the extension of the solution from part h, using redundancy resolution, to define a secondary objective function of the Panda's joint variables that maximizes the manipulability measure and drives the robot away from singularities.

Method for Solution:

First, we define the manipulability measure objective function as

$$w(q) = \det(J(q)J^T(q))$$

The goal is to find a desired motion

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

where \dot{q}_0 is defined as

$$\dot{q}_0 = k_0 \left(\frac{\partial w(q)}{\partial q} \right)^T$$

with $k_0 > 0$. Substituting the desired motion back into our solution from part h, we now have an updated inverse kinematics solution which avoids singularities.

Explanation of Program:

The function *redundancy_resolution.m*, which is shown below, was used to solve this problem. It is an extension of the previously shown *J_inverse_kinematics.m* function that additionally calculates the time-rate-of-change of the manipulability measure and incorporates it into the update of the joint angles.

```
function [theta_0, x, y, z] = redundancy_resolution(T_sd, theta_0, M, S_mat,
k0, should_plot, M_intermediates, alpha_0, w_tol, v_tol, fading_memory_size,
fading_memory_tolerance)
    %This function performs iterative numerical inverse kinematics to drive the
    robot from one state to another.
    % Inputs: should_plot defaults to false, and is used to animate the
    % motion. M_intermediates is used for plotting the robot's links.
    % alpha_0, w_tol, and v_tol are all tolerancing and tuning parameters
    % used to change how quickly the robot state converges and how precise it
    % needs to be. Fading_memory_size is used to determine how far into the
    % past the fading memory filter looks.
    if nargin < 9
```

```

w_tol = 1e-02;
v_tol = 1e-02;
end
if nargin < 8
alpha_0 = 1e-02;
end
if nargin < 7
should_plot = false;
end
if nargin < 11
fading_memory_size = 10;
end
if nargin < 12
fading_memory_tolerance = 1e-03;
end

w_mag = w_tol + 1;
v_mag = v_tol + 1;

[T_0] = FK_space(M, S_mat, theta_0);
x = [];
y = [];
z = [];
previous_delta_thetas = zeros(size(S_mat,2), fading_memory_size);
previous_thetas = zeros(size(S_mat,2), 2);
previous_ws = zeros(1, 2);
count = 0;

while w_mag > w_tol || v_mag > v_tol
if should_plot == true
cla;
[T_sb] = FK_space(M, S_mat, theta_0, false, true,
M_intermediates, false);
x = [x, T_sb(1, 4)];
y = [y, T_sb(2, 4)];
z = [z, T_sb(3, 4)];
plot3(x, y, z, 'k')
plotFrame_T(T_0, 'start', 0.1)
plotFrame_T(T_sb, 'goal', 0.1)
pause(0.03);
else
[T_sb] = FK_space(M, S_mat, theta_0);
end

J_s = SpaceJacobian(S_mat, theta_0);
J_b = Ad(inv(T_sb)) * J_s;

%         if should_plot == true
%             ellipsoid_plot_angular(J_s, T_sb(1:3,4), 0.25, false);
%             ellipsoid_plot_linear(J_s, T_sb(1:3,4), 0.5, false);
%             pause(0.03);
%         end

```

```

J_diff = J_s - Ad(T_sb) * J_b;

T_bd = T_sb \ T_sd;
nu_b = T2S(T_bd);

% Using redundancy resolution if able to calculate the derivative
if count > 2
    d_theta = previous_thetas(:,2) - previous_thetas(:,1);
    w = sqrt(det(J_b * J_b'));
    d_w = previous_ws(2) - previous_ws(1);
    dw_dtheta = d_w ./ d_theta';
    q0_dot = k0 * dw_dtheta';
else
    w = 0;
    q0_dot = zeros(7,1);
end

standard_update = alpha_0 * J_dagger(J_b) * nu_b;
additional_update = (eye(size(J_b' * J_b, 2)) - J_dagger(J_b) * J_b) *
q0_dot;
delta_theta = standard_update + additional_update;
theta_0 = theta_0 + delta_theta';

count = count + 1;

% Storing w and thetas for derivative calculation
w_old = w;
if count <= 2
    previous_thetas(:, count) = theta_0;
    previous_ws(count) = w;
else
    previous_thetas(:, 1) = previous_thetas(:, 2);
    previous_thetas(:, 2) = theta_0;
    previous_ws(1) = previous_ws(2);
    previous_ws(2) = w;
end

% Fading memory implementation
if count <= fading_memory_size
    previous_delta_thetas(:, count) = delta_theta;
else
    for i = 2:fading_memory_size
        previous_delta_thetas(:, i - 1) = previous_delta_thetas(:,
i);
    end
    previous_delta_thetas(:, fading_memory_size) = delta_theta;

    % now checking to see if average of previous thetas
    means = zeros(size(S_mat,2), 1);
    for i = 1:size(S_mat,2)
        means(i) = mean(previous_delta_thetas(i, :));
    end
    if max(means) < fading_memory_tolerance

```

```

w_mag = w_tol;
v_mag = v_tol;

end
end
end
end

```

Answer / Test Cases:

The same several test cases used for parts h) and i) of this assignment were again used here, to produce the following results. To view the full gifs, explore the “GIFS” folder of the submission zip.

Note that these test cases are reused for each of the IK problems. They were selected because they represent realistic motions of a robotic manipulator, with several of them taking the manipulator close to or through singularity situations, allowing the differences in performance of the methods to be easily compared.

Test case 1: The robot moves from pose obtained by setting arbitrary $\theta_{a0} = [2\pi/3 \ \pi/6 \ 0 \ -\pi/4 \ \pi/4 \ -\pi/2 \ 0]$ to pose obtained by setting $\theta_{ad} = [0 \ \pi/2 \ 0 \ 0 \ 0 \ 0]$:

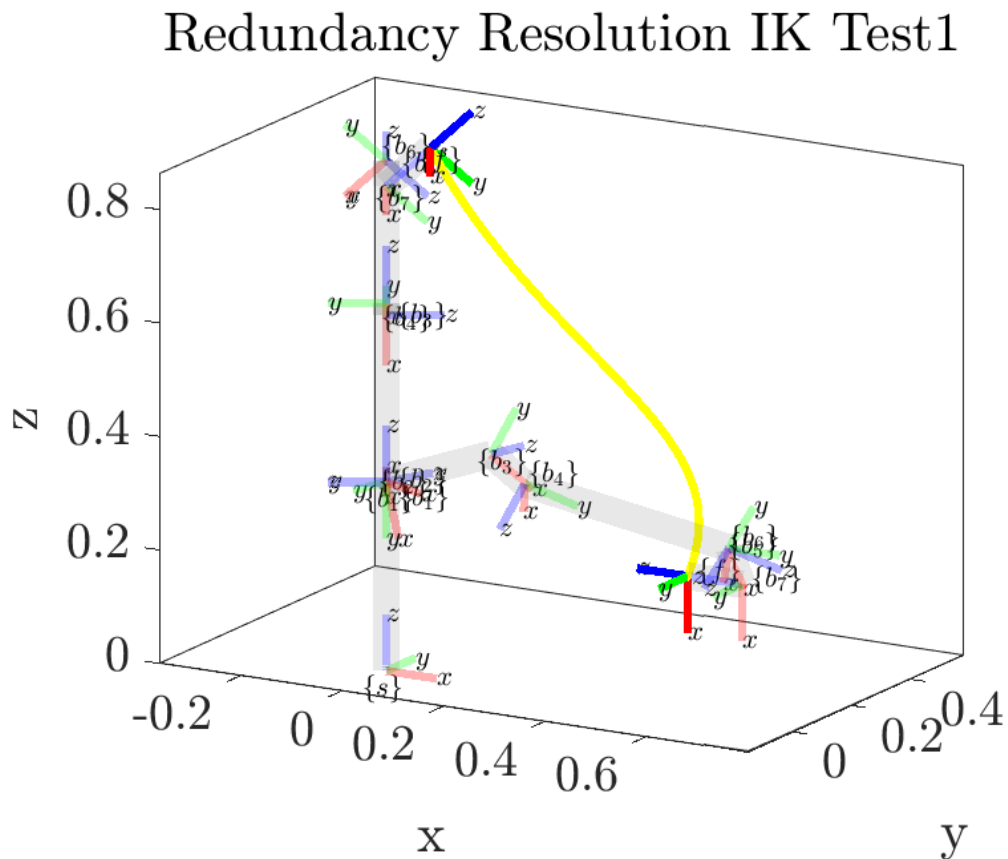


Figure 23: Redundancy resolution inverse kinematics test case 1

Test case 2: The robot performs a “pick and place” motion from the “ready” configuration $\theta_0 = [0 \ -\pi/4 \ 0 \ -3\pi/4 \ 0 \ \pi/2 \ \pi/4]$ to the world cartesian coordinate $(0.3, 0.3, 0)$ with a twist of the end effector.

Redundancy Resolution IK Test2

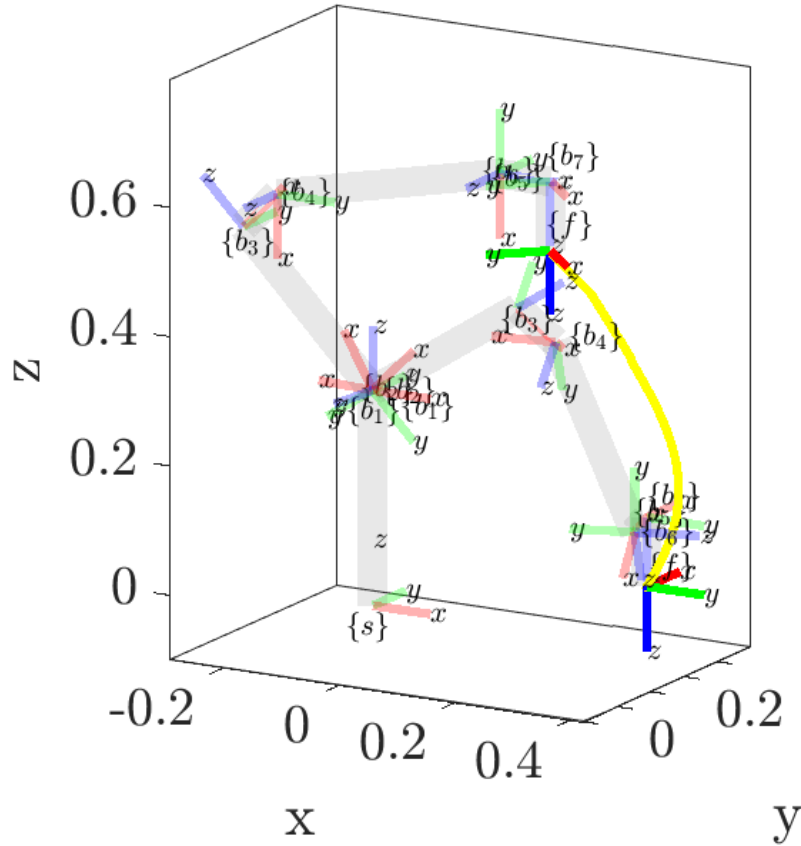


Figure 24: Redundancy resolution inverse kinematics test case 2

Test case 3: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

Redundancy Resolution IK Test3

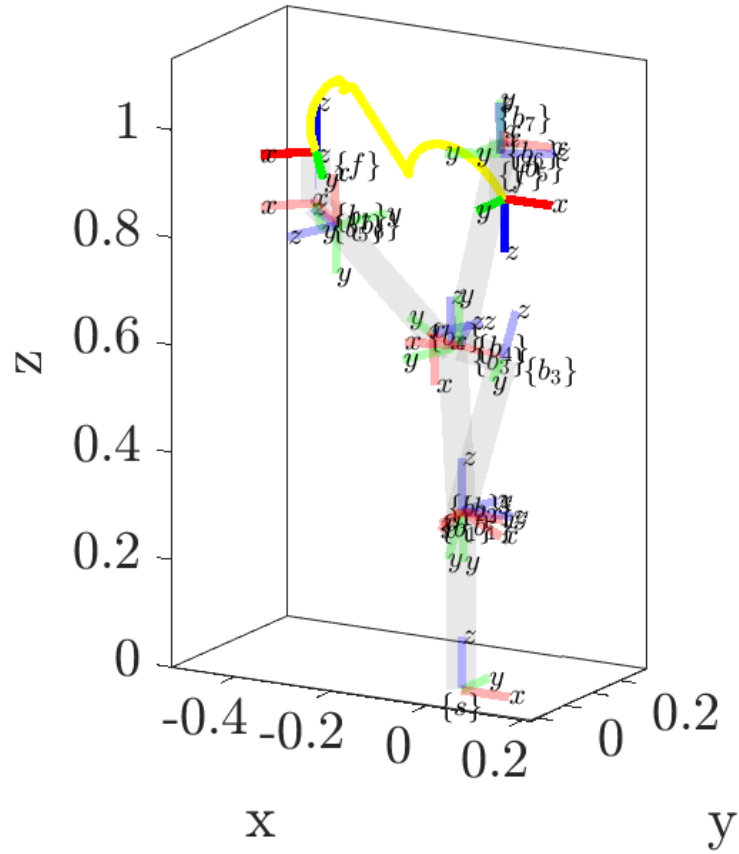


Figure 25: Redundancy resolution inverse kinematics test case 3

Test case 4: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

Redundancy Resolution IK Test4

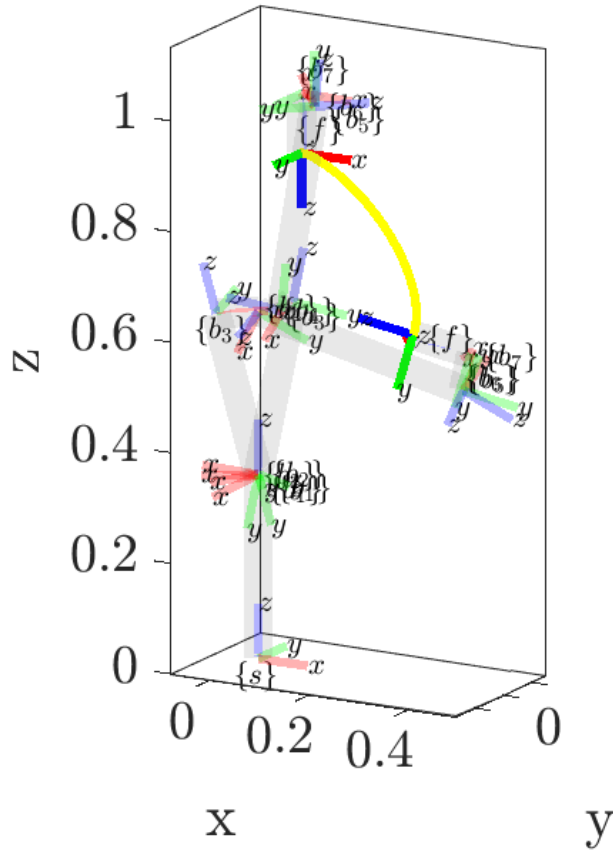


Figure 26: Redundancy resolution inverse kinematics test case 4

Test case 5: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

Redundancy Resolution IK Test5

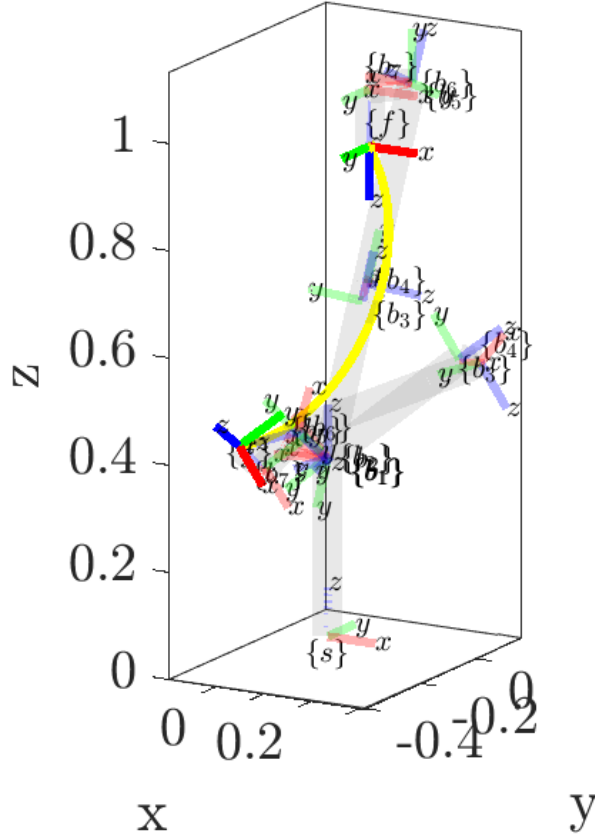


Figure 27: Redundancy resolution inverse kinematics test case 5

These results are quite similar to those of the original IK algorithm, which is logical as the robot successfully avoids singularity situations for most of the tests using the original algorithm. The most notable exception to this is test case 3, which does show the redundancy resolution in effect, as the algorithm initially drives the robot away from possible singularity near the edge of its workspace before finding a significantly smoother path to the desired pose. In testing, the scalar $k\theta$ was adjusted to be significantly higher than the settled-on value of 0.1, which produced significantly exaggerated singularity-phobic behavior in which the end-effector would jump across the workspace when even remotely near singularity configurations. This made this algorithm difficult to tune, though it did verify that the additions of the redundancy resolution algorithm did force the robot away from singularity configurations.

This performance will be compared to the performance of the remaining algorithms in the following sections. The reader is referred to the GIFs in the included files with this report to better understand the performance of these algorithms.

k) Damped Least-Squares Inverse Kinematics and *DLS_inverse_kinematics.m*

Problem Description:

This problem requests the extension of the solution from part h, using the damped least square approach to control the robot near singularity.

Method for Solution:

The damped least squares jacobian, J^* , is defined as

$$J^* = J^T(JJ^T + k^2I)^{-1}$$

Implementing the damped least squares method is as simple as replacing J^\dagger in our problem h solution with the new J^* .

Explanation of Program:

The function *DLS_inverse_kinematics.m*, which is shown below, was used to solve this problem. It is an extension of the previously shown *J_inverse_kinematics.m* function that additionally calculates uses the J_{star} matrix as opposed to J_{dagger} when the robot is detected to be near singularity situations.

```
function [theta_0, x, y, z] = DLS_inverse_kinematics(T_sd, theta_0, M, S_mat,
k, should_plot, M_intermediates, produce_gif, gif_name, alpha_0, w_tol, v_tol,
fading_memory_size, fading_memory_tolerance)
%This function performs iterative numerical inverse kinematics to drive the
robot from one state to another.
% Inputs: should_plot defaults to false, and is used to animate the
% motion. M_intermediates is used for plotting the robot's links.
% alpha_0, w_tol, and v_tol are all tolerancing and tuning parameters
% used to change how quickly the robot state converges and how precise it
% needs to be. Fading_memory_size is used to determine how far into the
% past the fading memory filter looks.
    if nargin < 11
        w_tol = 1e-02;
        v_tol = 1e-02;
    end
    if nargin < 10
        alpha_0 = 1e-02;
    end
    if nargin < 7
        should_plot = false;
    end
    if nargin < 13
```

```

fading_memory_size = 10;
end
if nargin < 14
fading_memory_tolerance = 1e-03;
end
if nargin < 8
produce_gif = false;
end
if nargin < 9
gif_name = "dls_ik.gif";
end

w_mag = w_tol + 1;
v_mag = v_tol + 1;

[T_0] = FK_space(M, S_mat, theta_0);
x = [];
y = [];
z = [];
previous_delta_thetas = zeros(size(S_mat,2), fading_memory_size);
count = 0;

if produce_gif == true
filename = strcat(gif_name, ".gif");
end

while w_mag > w_tol || v_mag > v_tol
if should_plot == true
cla;
[T_sb] = FK_space(M, S_mat, theta_0, false, true, M_intermediates,
false);

x = [x, T_sb(1, 4)];
y = [y, T_sb(2, 4)];
z = [z, T_sb(3, 4)];
plot3(x, y, z, 'k')
plotFrame_T(T_0, 'start', 0.1)
plotFrame_T(T_sd, 'goal', 0.1)
pause(0.03);

% creating gif
if produce_gif == true
frame = getframe(gcf);
im = frame2im(frame);
[imind,cm] = rgb2ind(im,256);
if count == 0
imwrite(imind,cm,filename,'gif', 'Loopcount',inf,...
'DelayTime',0.1);
else
imwrite(imind,cm,filename,'gif','WriteMode','append',...
'DelayTime',0.1);
end
end
else

```

```

        [T_sb] = FK_space(M, S_mat, theta_0);
    end

    J_s = SpaceJacobian(S_mat, theta_0);
    J_b = Ad(inv(T_sb)) * J_s;

    J_diff = J_s - Ad(T_sb) * J_b;

    T_bd = T_sb \ T_sd;
    nu_b = T2S(T_bd);

    %         if J_isotropy(J_b, "linear") < 0.1 || J_ellipsoid_volume(J_b,
"linear") < 0.1 || J_isotropy(J_b, "angular") < 0.1 || J_ellipsoid_volume(J_b,
"angular") < 0.1
        if J_ellipsoid_volume(J_b, "linear") < 0.1 || J_ellipsoid_volume(J_b,
"angular") < 0.1
            %                 theta_bank(:, i + 1) = theta_bank(:, i) + J_star(J_b, k) *
(alpha .* nu_b); % updating thetas
            delta_theta = alpha_0 * J_star(J_b, k) * nu_b;
        else
            delta_theta = alpha_0 * J_dagger(J_b) * nu_b;
        end
        theta_0 = theta_0 + delta_theta';

        count = count + 1;

        % Fading memory implementation
        if count <= fading_memory_size
            previous_delta_thetas(:, count) = delta_theta;
        else
            for i = 2:fading_memory_size
                previous_delta_thetas(:, i - 1) = previous_delta_thetas(:,
i);

            end
            previous_delta_thetas(:, fading_memory_size) = delta_theta;

            % now checking to see if average of previous thetas
            means = zeros(size(S_mat,2), 1);
            for i = 1:size(S_mat,2)
                means(i) = mean(previous_delta_thetas(i, :));
            end
            if max(means) < fading_memory_tolerance
                w_mag = w_tol;
                v_mag = v_tol;
            end
        end
    end
end
end
end

```

$J_{star:m}$ is used in the near-singularity update form of this algorithm, which is triggered when the Jacobian is near singularity (as determined by a threshold on the volume of the manipulability ellipsoid), and is shown below:

```
function [J_star] = J_star(J, k)
    %Calculates the damped least-squares inverse using a damping factor k
    if nargin < 2
        k = 1;
    end
    n = size(J*J',1);
    J_star = J' / (J*J' + k^2*eye(n));
end
```

Answer / Test Cases:

The same several test cases used for parts h), i), and j) of this assignment were again used here, to produce the following results. To view the full gifs, explore the “GIFS” folder of the submission zip.

Note that these test cases are reused for each of the IK problems. They were selected because they represent realistic motions of a robotic manipulator, with several of them taking the manipulator close to or through singularity situations, allowing the differences in performance of the methods to be easily compared.

Test case 1: The robot moves from pose obtained by setting arbitrary $\theta_0 = [2\pi/3 \ \pi/6 \ 0 \ -\pi/4 \ \pi/4 \ -\pi/2 \ 0]$ to pose obtained by setting $\theta_d = [0 \ \pi/2 \ 0 \ 0 \ 0 \ 0 \ 0]$:

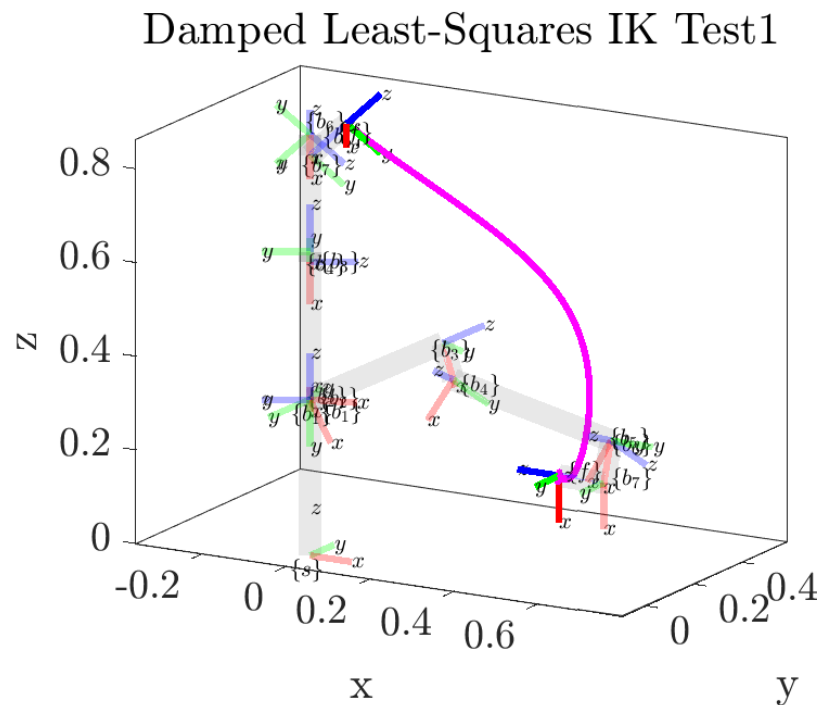


Figure 28: DLS inverse kinematics test case 1

Test case 2: The robot performs a “pick and place” motion from the “ready” configuration $\theta_0 = [0 \ -\pi/4 \ 0 \ -3\pi/4 \ 0 \ \pi/2 \ \pi/4]$ to the world cartesian coordinate $(0.3, 0.3, 0)$ with a twist of the end effector.

Damped Least-Squares IK Test2

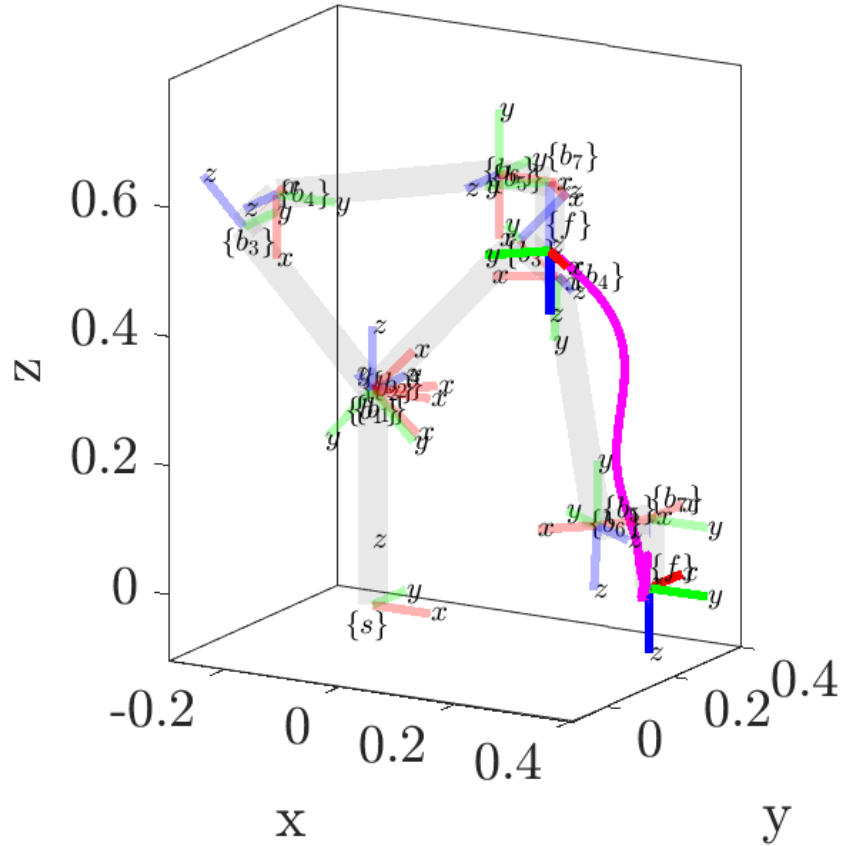


Figure 29: DLS inverse kinematics test case 2

Test case 3: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

Damped Least-Squares IK Test3

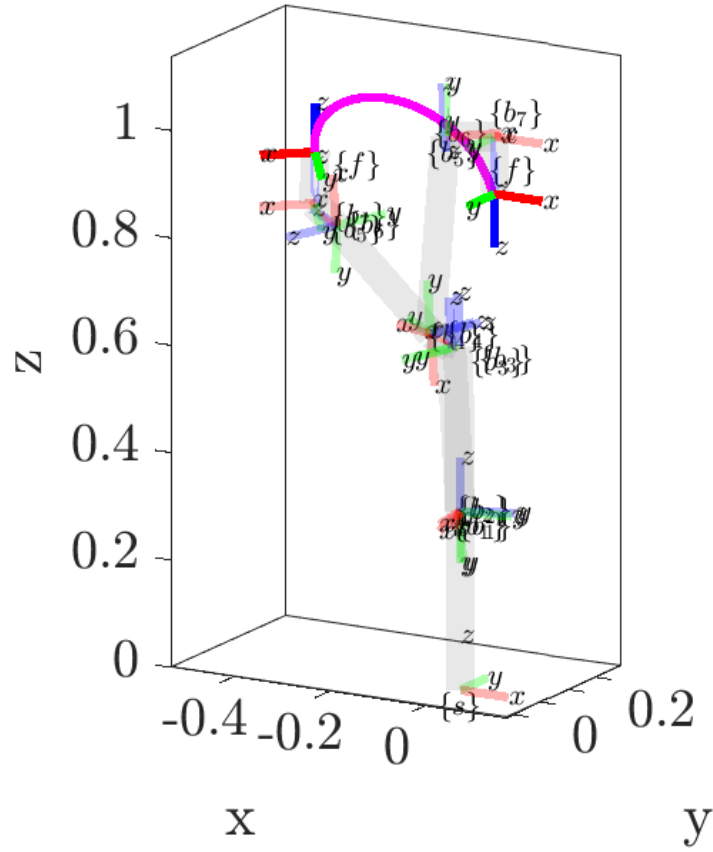


Figure 30: DLS inverse kinematics test case 3

Test case 4: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

Damped Least-Squares IK Test4

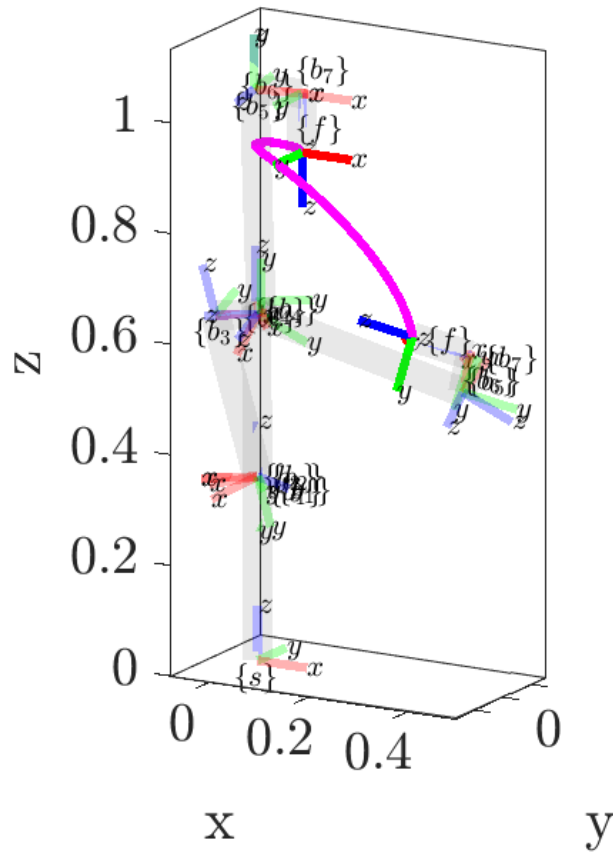


Figure 31: DLS inverse kinematics test case 4

Test case 5: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

Damped Least-Squares IK Test5

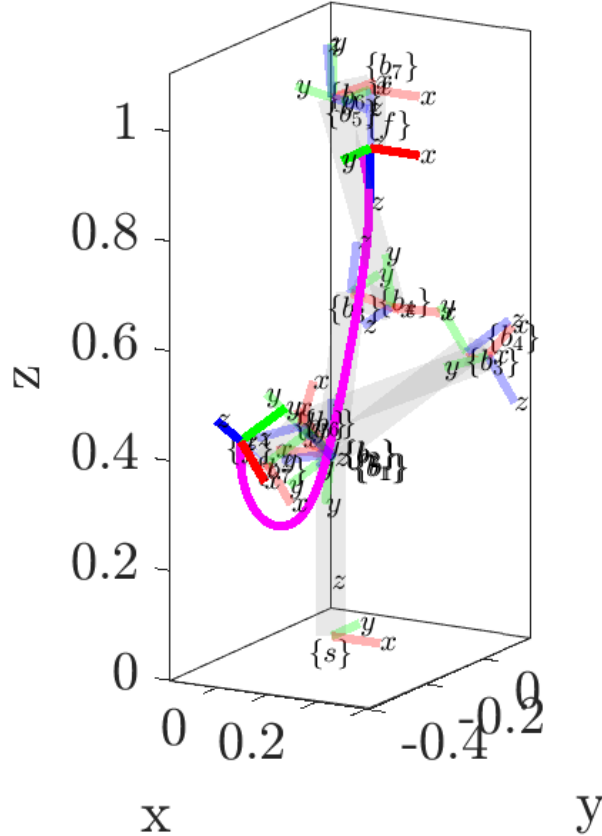


Figure 32: DLS inverse kinematics test case 5

These results are surprisingly similar to those produced by the transpose algorithm, featuring similar behavior with taking longer paths to reach the goal pose than the original or redundancy resolution algorithms. The paths are not quite as deviant, however, and do successfully avoid the singularity configurations around the edge of the workspace that proved problematic previously in test case 3. The robot also generally takes more conservative paths, as seen in test cases 2, 4, and 5 than those of the original or RR algorithms, preferring to move the robot deeper within its workspace where it has higher manipulability before moving toward the goal. For these tests, a k of 0.5 was used.

This performance of each of these algorithms is visually compared in the following section. The reader is referred to the GIFs in the included files with this report to better understand the performance of these algorithms.

Summary of Inverse Kinematics Results

Explanation of Program:

The script *IK_comparison.m* was used to successively view and then plot the trajectories of the manipulator using the various IK methods covered here. The results are plots that show the trajectories of each method with representations of the robot at its initial position and one possible configuration to reach the desired end-effector pose.

```
% Format workspace
clc; clear; format compact; clf; close all;
set(0, 'defaultTextInterpreter', 'latex');
set(0, 'defaultAxesTickLabelInterpreter', 'latex');
set(0, 'defaultLegendInterpreter', 'latex');
set(0, 'defaultAxesFontSize', 18);
set(0, 'DefaultLineLineWidth', 2);
set(groot, 'defaultFigureUnits', 'pixels', 'defaultFigurePosition', [440 278
560 420]);

disp('-----Comparison of IK Methods-----')
[M, thetas, S_mat, B_mat, M_intermediates] = instantiate_robot();

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% % Test case 1
% theta_0 = [2*pi/3 pi/6 0 -pi/4 pi/4 -pi/2 0];
% theta_d = [0 pi/2 0 0 0 0 0];
% [FK_solution_space] = FK_space(M, S_mat, theta_d);
% target_orientation = FK_solution_space(1:3, 1:3);
% target_position = FK_solution_space(1:3, 4);
% test_name = "Test1";

% % Test case 2
% theta_0 = [0 -pi/4 0 -3*pi/4 0 pi/2 pi/4];
% target_orientation = [0 1 0; 1 0 0; 0 0 -1];
% target_position = [0.3, 0.3, 0]';
% test_name = "Test2";

% % Test case 3
% rng(10)
% theta_0 = 2 * pi * rand(1, 7);
% theta_d = [0 0 0 0 0 0 0];
% [FK_solution_space] = FK_space(M, S_mat, theta_d);
% target_orientation = FK_solution_space(1:3, 1:3);
% target_position = FK_solution_space(1:3, 4);
% test_name = "Test3";

% % Test case 4
% rng(51)
% theta_0 = 2 * pi * rand(1, 7);
% theta_d = [0 0 0 0 0 0 0];
```

```

% [FK_solution_space] = FK_space(M, S_mat, theta_d);
% target_orientation = FK_solution_space(1:3, 1:3);
% target_position = FK_solution_space(1:3, 4);
% test_name = "Test4";

% Test case 5
rng(67)
theta_0 = 2 * pi * rand(1, 7);
theta_d = [0 0 0 0 0 0 0];
[FK_solution_space] = FK_space(M, S_mat, theta_d);
target_orientation = FK_solution_space(1:3, 1:3);
target_position = FK_solution_space(1:3, 4);
test_name = "Test5";
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Target pose
T_sd = [target_orientation target_position;
        zeros(1,3), 1];

% Creating plot
[T_sb] = FK_space(M, S_mat, theta_0, false, true, M_intermediates);
pause(1);

% IK using pseudo inverse
[theta_f, x1, y1, z1] = J_inverse_kinematics(T_sd, theta_0, M, S_mat, true,
M_intermediates);

% IK using transpose
K = 1e-02 * eye(6);
[theta_f, x2, y2, z2] = J_transpose_kinematics(T_sd, theta_0, M, S_mat, K,
true, M_intermediates);

% IK using RR
k0 = 0.1;
[theta_f, x3, y3, z3] = redundancy_resolution(T_sd, theta_0, M, S_mat, k0,
true, M_intermediates);

% IK using DLS
k = 0.5;
[theta_f, x4, y4, z4] = DLS_inverse_kinematics(T_sd, theta_0, M, S_mat, k,
true, M_intermediates);

% Comparison
cla;
plot3(x1, y1, z1, '-', 'Color', '#000000', 'LineWidth', 3)
plot3(x2, y2, z2, '-', 'Color', '#00FFFF', 'LineWidth', 3)
plot3(x3, y3, z3, '-', 'Color', '#FFFF00', 'LineWidth', 3)
plot3(x4, y4, z4, '-', 'Color', '#FF00FF', 'LineWidth', 3)
[FK_solution_space] = FK_space2(M, S_mat, theta_0, true, M_intermediates, 0.3,
false, false);
[FK_solution_space] = FK_space2(M, S_mat, theta_f, true, M_intermediates, 0.3,
false, false);
title(strcat('IK Method Comparison', {' '}, test_name))

```

```

legend('Pseudo-Inverse', 'Transpose', 'Redundancy Resolution', 'Damped
Least-Squares')
legend('location', 'best')

```

Test case 1: The robot moves from pose obtained by setting arbitrary $\theta_{a0} = [2\pi/3 \ \pi/6 \ 0 \ -\pi/4 \ \pi/4 \ -\pi/2 \ 0]$ to pose obtained by setting $\theta_{ad} = [0 \ \pi/2 \ 0 \ 0 \ 0 \ 0 \ 0]$:

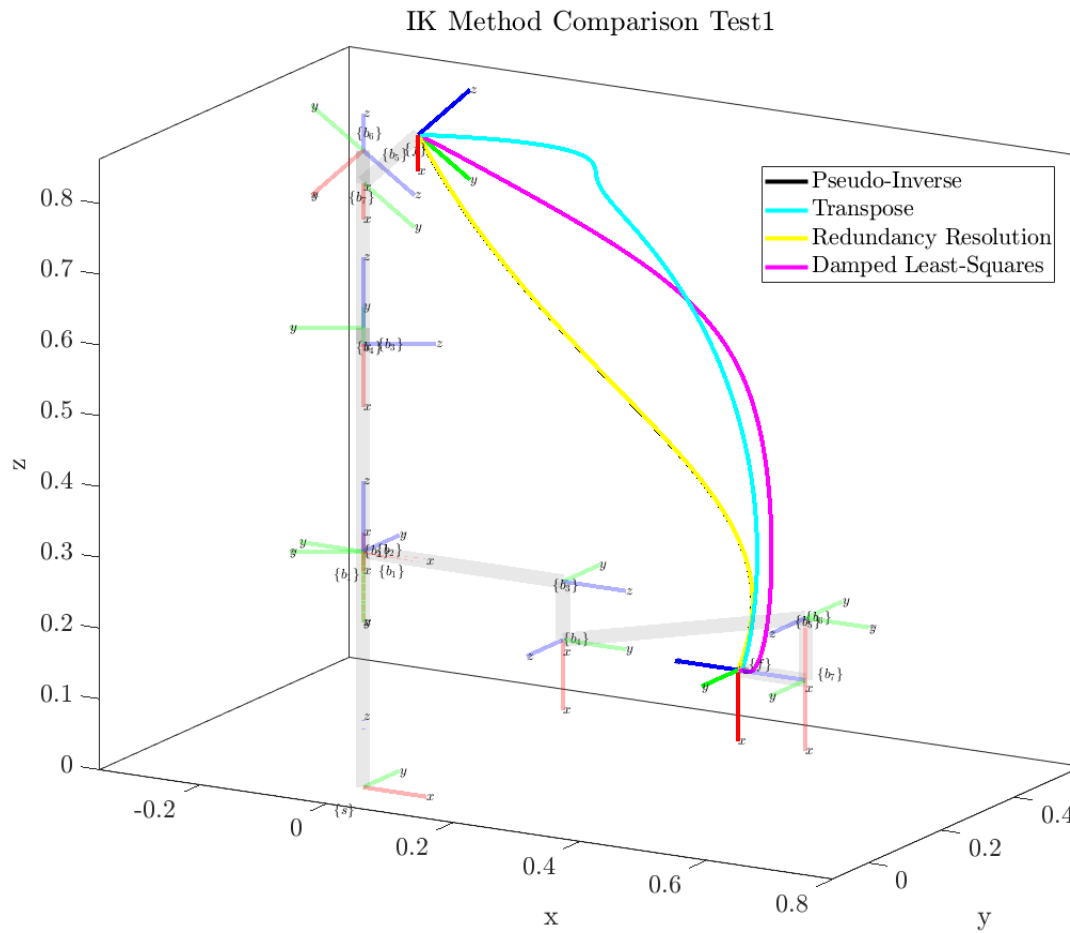
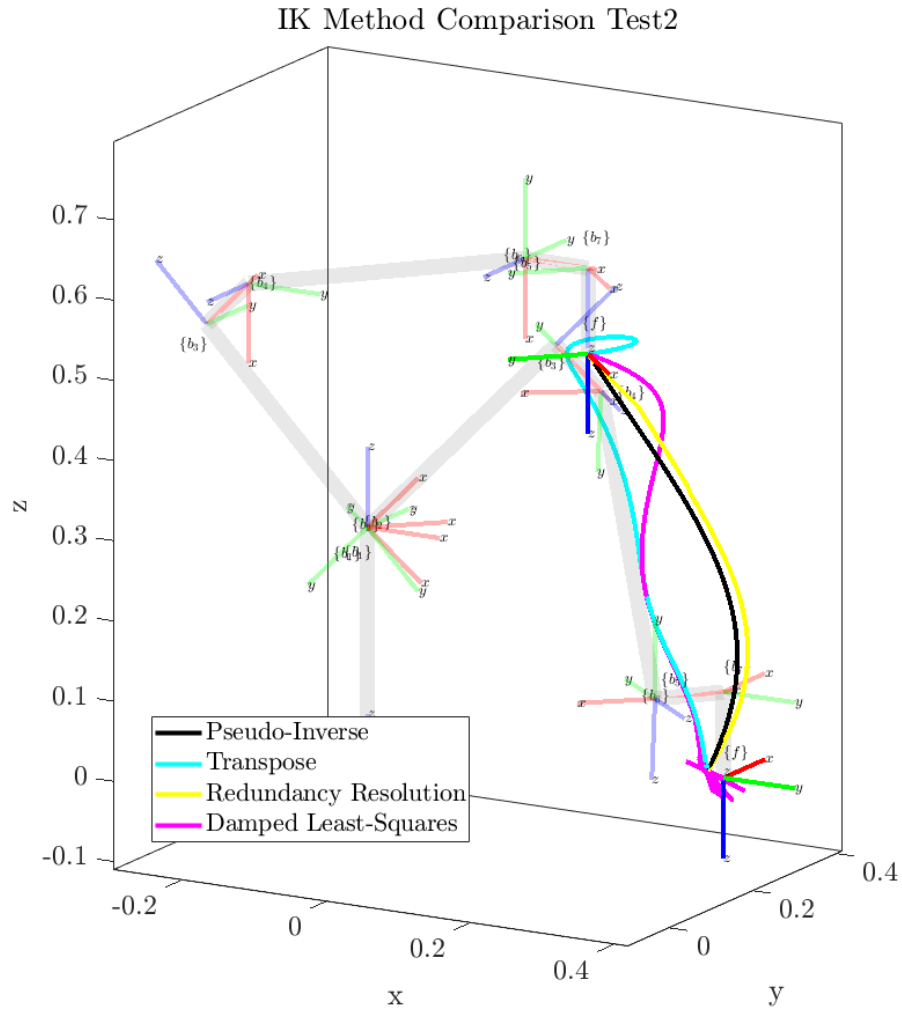


Figure 33: IK comparison for test case 1

Test case 2: The robot performs a “pick and place” motion from the “ready” configuration $\theta_0 = [0 \ -\pi/4 \ 0 \ -3\pi/4 \ 0 \ \pi/2 \ \pi/4]$ to the world cartesian coordinate $(0.3, 0.3, 0)$ with a twist of the end effector.



Test case 3: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

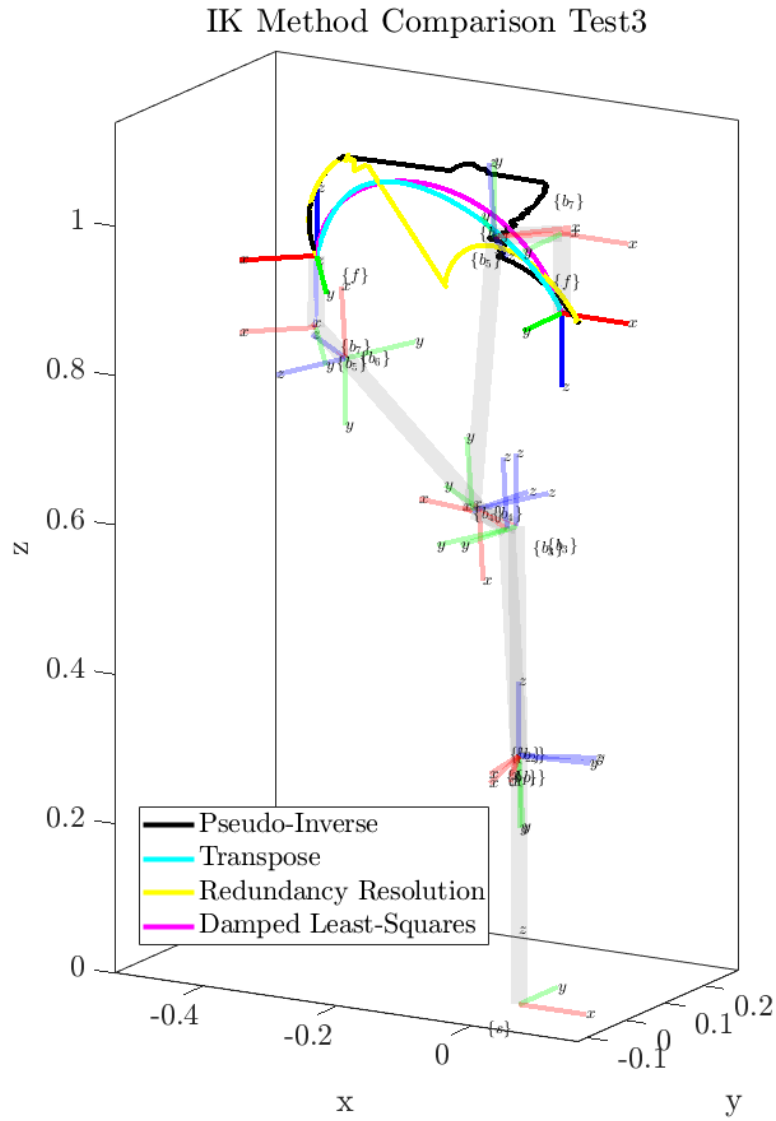


Figure 35: IK comparison for test case 3

Test case 4: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

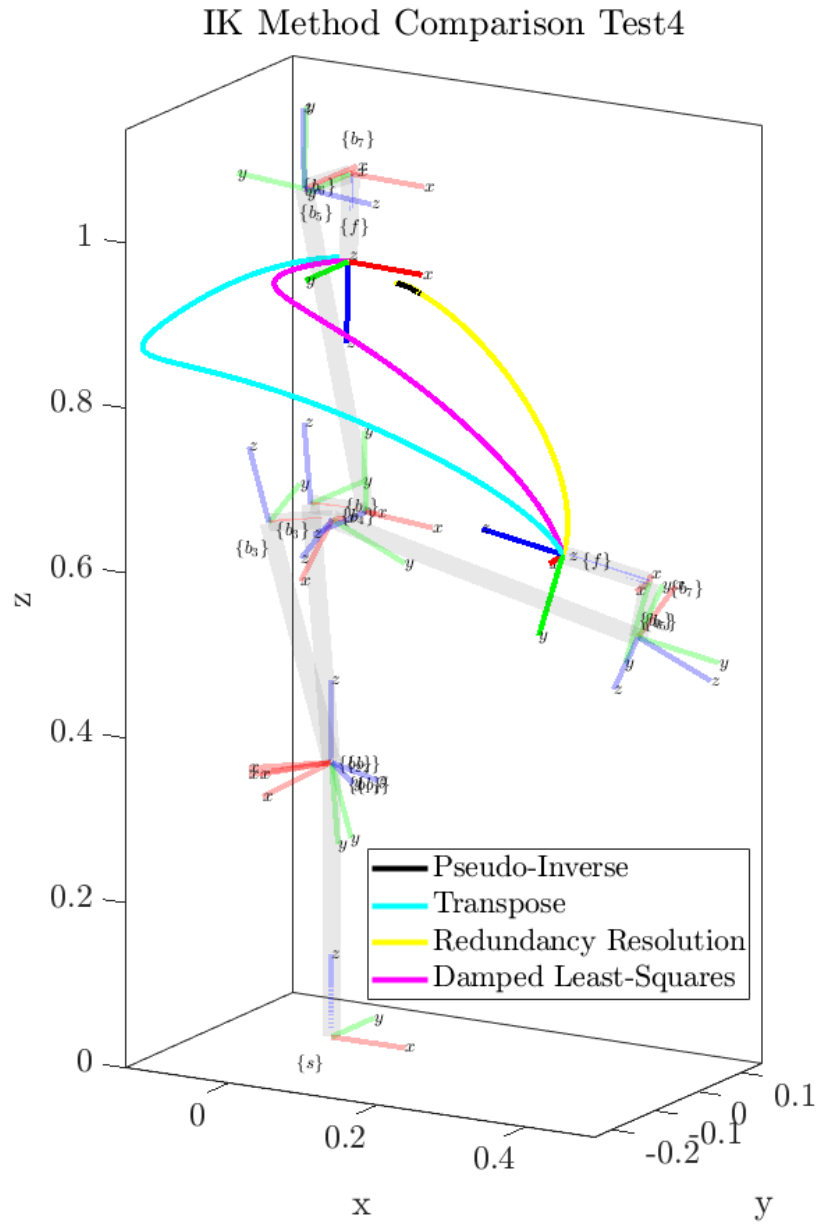


Figure 36: IK comparison for test case 4

Test case 5: The robot moves from a randomly chosen valid pose to its zero configuration, moving near singularity.

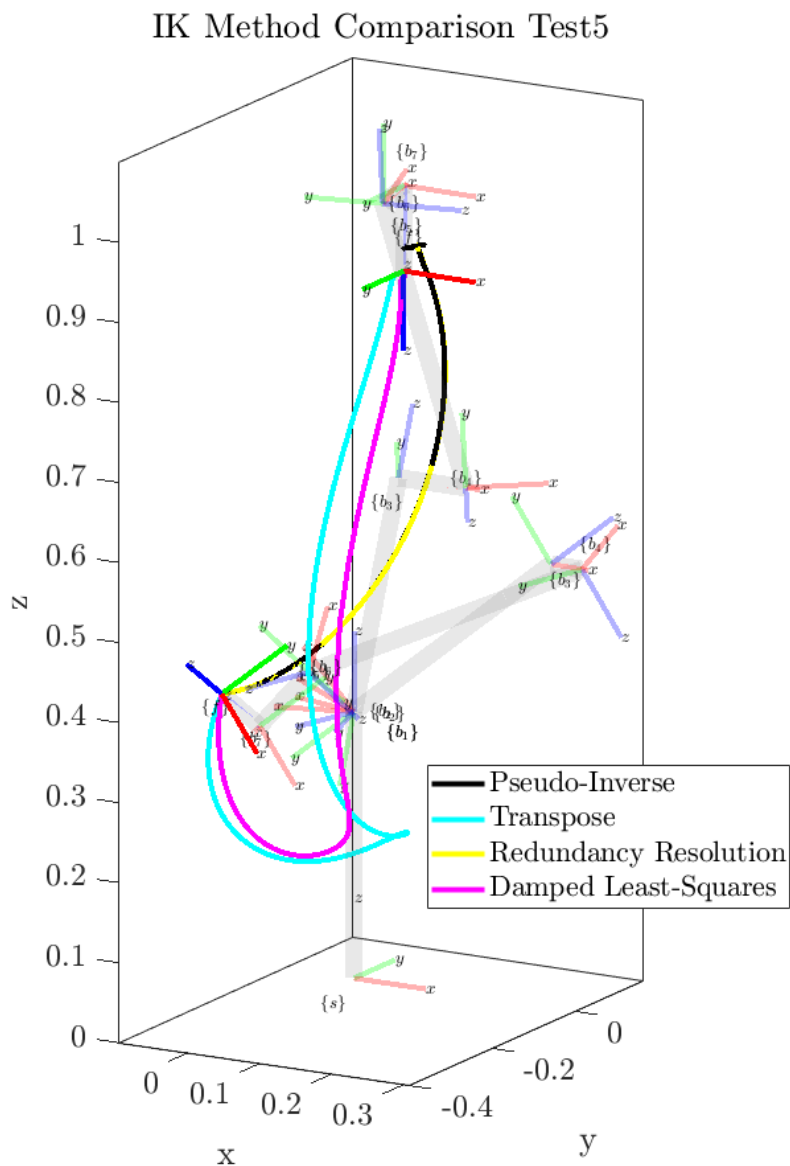


Figure 37: IK comparison for test case 5

For most of these cases, the original and the redundancy resolution approaches take very similar paths. As the robot is not passing overly close to singularity, this performance is not unexpected. The exception to this is case 3, in which the RR approach does successfully drive the robot away from singularity. The transpose and DLS approaches also take similar paths, though the path produced by the DLS approach is more efficient. The DLS pathing moves the robot more

through the portions of its workspace with maximum manipulability, which is expected as the conditional in the algorithm that switches between the default and DLS-calculated joint updates was set quite high such that the robot favors the DLS approach.

m) Graphical Simulation of Robot

Problem Description:

This problem requests the graphical simulation of the robot.

Method for Solution:

We chose to graphically simulate the robot entirely with our own MATLAB code. These graphical representations and relevant software are introduced in sections A (simulation) and G (ellipsoids), and used to produce the images and GIFs shown in this report and included in the attached files. The reason we chose to do this as we did (rather than using the MATLAB Robotics Toolbox or ROS), is because these files will generalize to produce a rough visualization of any robot given initial screw axis parameters and base configuration. Hence, they can be reused for any such robot, only requiring these parameters to be added to the created *instantiate_robot.m* script.

Explanation of Program:

Beyond the previously shown equations for statically or dynamically viewing the robot (ie. for FK, manipulability ellipsoids, and IK solutions), an additional function *animate_joint_goals.m* was created to take in a series of joint goals and animate the motion of the robot between them.

```
function animate_joint_goals(robot_name, theta_series, show_ellipsoids,
num_steps, delay, should_create_plot)
%Animates the motion of a known robot given a series of joint goals to
%reach
% Accepts a string for the name of the robot, which must be known to the
% "instantiate_robot()" function. Theta_series is a matrix of column
% vectors representing the desired joint values of the robot.
% show_ellipsoids will toggle the visible, possibly erroneous ellipsoids
% onto the plot, defaults to false. num_steps is the number of linear
% interpolation steps taken between joint values, defaults to 100. delay is
% time delay between updates of the plot, defaults to 0.05s.
% should_create_plot decides whether a new figure will be created, defaults
% to true.
    if nargin < 6
        should_create_plot = true;
    end
    if nargin < 5
        delay = 0.05;
```

```

end
if nargin < 4
num_steps = 100;
end
if nargin < 3
show_ellipsoids = false;
end

[M, ~, S_mat, ~, M_intermediates] = instantiate_robot(robot_name);

if should_create_plot == true
figure
view(3)
box on
hold on
grid on
axis equal;
xlabel('x'), ylabel('y'), zlabel('z');
xlim([-1, 1]);
ylim([-1, 1]);
    zlim([-0.5, 1.5]);
view([30, 15]);
end

for i = 1:size(theta_series, 2) - 1
current_thetas = theta_series(:, i);
delta_theta = zeros(size(theta_series, 1), 1);
for j = 1:size(theta_series, 1)
    delta_theta(j) = (theta_series(j, i + 1) - theta_series(j, i)) /
num_steps;
end
for j = 1:num_steps
cla;
[FK_solution_space, ~, ~] = FK_space(M, S_mat, current_thetas,
false, true, M_intermediates, false);
J_space = SpaceJacobian(S_mat, current_thetas);
    ellipsoid_plot_angular(J_space, FK_solution_space(1:3,4), 0.25,
show_ellipsoids);
    ellipsoid_plot_linear(J_space, FK_solution_space(1:3,4), 0.5,
show_ellipsoids);
    pause(delay);

    current_thetas = current_thetas + delta_theta;

end
end
end

```

Answer / Test Cases:

This function is provided with random valid joint values using the `ASBR_THA2_PA_{}.m` file corresponding to this problem to produce animations of the robot through various joint configurations. Because of the previously described errors with ellipsoid rotations, the ellipsoids are not included in these animations, though their axes are. GIFs of some of these animations can be found in the included files for this assignment.

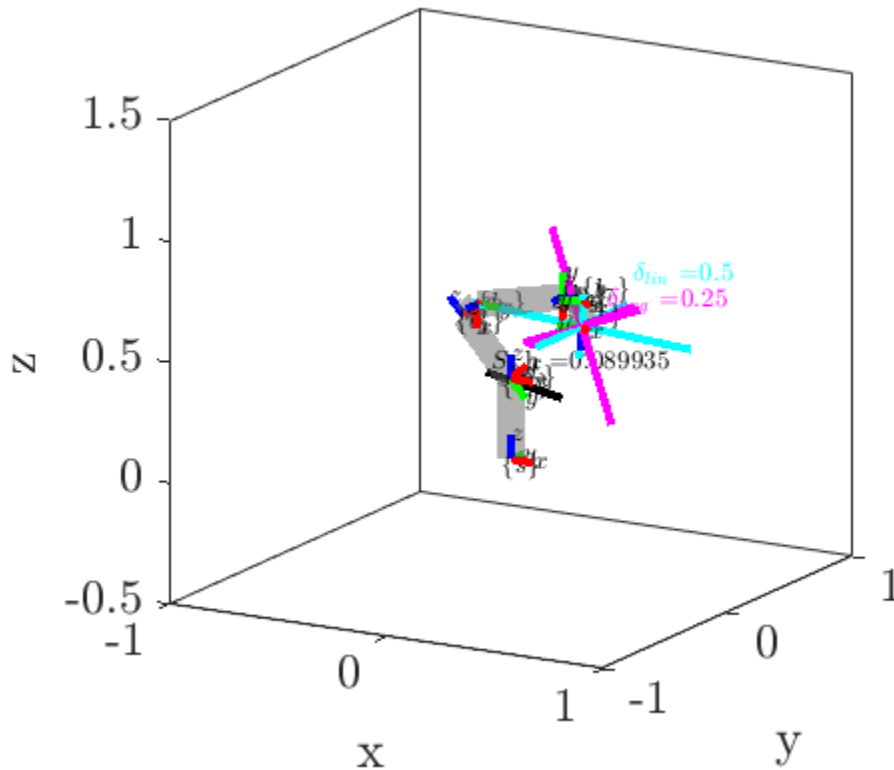


Figure 38: Still of our Franka manipulator as taken from one of the produced animation GIFs

Other Relevant Code

`Instantiate_robot.m` is a script used to, as the name implies, instantiate the robot. This allows us to confirm that our scripts are compatible with multiple robots (implemented are two options; the Franka-Emika Panda and a two link, 2D test robot). The function accepts the robot's name and a request for numerical or symbolic outputs, and returns the robot's end effector positional matrix M , thetas, screw axes in space and body form, and intermediate M matrices for plotting. The purpose of this function is to streamline the rest of the code and make it modular, plus localizing all the areas where changes need to be made if the user wishes to use any other function we show

in this report with a new robot they want to define. Because this script is incredibly long (6 pages in text form), we have chosen not to include it in the report - it can be found in the submitted folder.

ASBR_THA2_PA_x.m are a series of scripts ranging from $x \in [a, k]$. These individual scripts are used to run and test the above problems. An example of a script *ASBR_THA2_PA_a.m* is found below, and the rest can be found in the submitted folder as they are fairly redundant.

```
% Format workspace
clc; clear; format compact; clf; close all;

disp('-----a)-----')
disp('Symbolic Space FK:')

% Instantiating the robot object symbolically
[M, thetas, S_mat, B_mat] = instantiate_robot("franka", true);

% Performing space FK symbolically
[FK_solution_space, T_bank_space, T_total_bank_space] = FK_space(M, S_mat,
thetas, true);
simplify(FK_solution_space)
```

An integral piece of code is below:

```
set(0, 'defaultTextInterpreter', 'latex');
set(0, 'defaultAxesTickLabelInterpreter', 'latex');
set(0, 'defaultLegendInterpreter', 'latex');
set(0, 'defaultAxesFontSize', 18);
set(0, 'DefaultLineLineWidth', 2);
set(groot, 'defaultFigureUnits', 'pixels', 'defaultFigurePosition', [440
278 560 420]);
```

Running this code once at the instantiation of a MATLAB instance allows the plots to be graphed as intended, using LaTeX formatting and other similar formatting methods to produce the figures as they are shown here.

Contributions of Group Members

Jared Rosenbaum completed Homework Assignment (HA) Q1, HA Q2, HA Q3, HA Q4.

Steven Swanbeck completed Programming Assignment (PA) QB, PA QD, PA QE, PA QF, PA QH, PA QM.

Responsibility was shared on PA QA, PA QC, PA QG, PA QI, PA QJ, PA QK, and PA QL (This document).

References

1. “Robot and Interface Specifications.” *Robot and Interface Specifications - Franka Control Interface (FCI) Documentation*, https://frankaemika.github.io/docs/control_parameters.html.
2. Murray, R.M., Li, Z., & Sastry, S.S. (1994). *A Mathematical Introduction to Robotic Manipulation* (1st ed.). CRC Press. <https://doi.org/10.1201/9781315136370>
3. Lynch, K. M., & Park, F. C. (2017). *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press.