THA Programming Assignment #1

ME 397 ABSR, Spr 23, Dr. Farshid Alambeigi

Jared Rosenbaum and Steven Swanbeck

**PA 1**

*Problem Description:*

This problem requests the creation of three functions that each accept a rotation matrix and return a) an equivalent axis-angle representation, b) an equivalent quaternion representation, and c) both equivalent ZYZ and RPY angle representations.

*Method for Solution:*

For part a), the matrix logarithm of rotations is used to find a unit axis *ω* and rotation angle *θ* that parameterize the rotation such that . The algorithm for this is given in several steps, as follows:

1. If *R = I*, there is no rotation, so *θ = 0* and *ω* is undefined.
2. If the trace of *R* is -1, *θ = π* and is given by any of:
3. In other cases:

Using this algorithm will produce the *ω* and *θ* that describe an equivalent rotation of an input *R*.

For part b), the unit quaternion representation of a rotation from a rotation matrix *R* is given by:

Where *sgn()* is the sign or signum function. Using these equations will produce a unit quaternion that describes an equivalent rotation to *R* so long as

For part c), the *ZYZ* angle representation equivalent to a given *R* can be calculated as:

Where the rotations are applied according to the body frame of the rotating object and angles *φ*, *θ*, and *ψ* are calculated as:

For and:

For There are special cases, including:

1. If *sin(θ) = 0*, only the sum or difference of *φ* and *ψ* can be calculated. This occurs when *θ = 0, π*, which means that the first and third rotations are about parallel axes and therefore not well defined with this representation.
2. If *R = I*, there is no rotation, and *ZYZ* angle representation is singular here.

The RPY(*ZYX*) angle representation equivalent to a given *R* is calculated as:

Where the rotations are applied according to a fixed frame coincident with the center of mass of the rotating object and angles *φ*, *θ*, and *ψ* are calculated as:

For and:

For .

RPY angle representation is singular when *θ = ± π/2.*

*Explanation of Program:*

The first function developed is used for part a) and converts *R* to an equivalent *ω* and *θ*:

function [w, theta] = RotationMatrix2AxisAngle(R)  
% Converts 3x3 rotation matrix into equivalent axis representation  
% Accepts 3x3 orthonormal matrix and returns a theta value in the range  
% [0, pi] radians and a column vector w representing the unit rotation  
% axis  
 if R == eye  
 disp('[UNDEFINED WARNING] infinitely many representations for identity rotation.')  
 theta = 0;  
 w = [NaN NaN NaN]';  
 elseif trace(R) == -1  
 theta = pi;  
 if R(3, 3) ~= -1  
 w = 1 / (sqrt(2 \* (1 + R(3, 3)))) \* [R(1, 3); R(2, 3); 1 + R(3, 3)];  
 elseif R(2, 2) ~= -1  
 w = 1 / (sqrt(2 \* (1 + R(2, 2)))) \* [R(1, 2); 1 + R(2, 2); R(3, 2)];  
 elseif R(1, 1) ~= -1  
 w = 1 / (sqrt(2 \* (1 + R(1, 1)))) \* [1 + R(1, 1); R(2, 1); R(3, 1)];  
 end  
 else  
 theta = acos(0.5 \* (trace(R) - 1));  
 w\_hat = 1 / (2 \* sin(theta)) \* (R - R');  
 w = [w\_hat(3, 2) w\_hat(1, 3) w\_hat(2, 1)]';  
 end  
end

*RotationMatrix2AxisAngle* accepts a 3x3 orthonormal rotation matrix and implement the equations previously shown for the matrix logarithm of rotations. If *R* is an identity matrix, the user is presented a warning stating the rotation is not well defined and the resulting *ω* returns full of *NaN.* Following this check, if the trace of *R* is -1, the special equations previously shown are used. To increase robustness, there is an additional check of the diagonal components of *R* to make sure these equations are not singular, and the algorithm will progress to the next equation if the checked element of *R* is *0.* Since *tr(R) = -1* in this case, it is guaranteed that at least of the diagonal elements of *R* is nonzero and one of these equations will be executed. Finally, if neither of these previous checks are true of the input *R*, the general equations are used to produce *ω* and *θ.*

The second function is used for part b) and converts *R* into an equivalent unit quaternion representation:

function [q] = RotationMatrix2Quaternion(R)  
%receives a 3x3 orthonormal rotation matrix and returns the equivalent  
%unit quaternion  
% accepts 3x3 matrix and returns a 4x1 column vector of quaternion  
% components [q0 q1 q2 q3]'  
 q = zeros(4,1);  
 q(1) = 1/2 \* sqrt(R(1,1) + R(2,2) + R(3,3) + 1);  
 q(2) = 1/2 \* sign(R(3,2) - R(2,3)) \* sqrt(R(1,1) - R(2,2) - R(3,3) + 1);  
 q(3) = 1/2 \* sign(R(1,3) - R(3,1)) \* sqrt(R(2,2) - R(3,3) - R(1,1) + 1);  
 q(4) = 1/2 \* sign(R(2,1) - R(1,2)) \* sqrt(R(3,3) - R(1,1) - R(2,2) + 1);  
end

This function directly applies the previously shown equations to the input matrix *R*. Since there are no singularity conditions for the unit quaternion representation of an input rotation matrix, there are no guard clauses to handle edge cases. So long as a 3x3 orthonormal rotation matrix is the input to this function, an equivalent quaternion representation will be produced.

The third function used for part c) calculates an equivalent *ZYZ* angle representation for an input *R*:

function [phi, theta, psi] = RotationMatrix2ZYZAngles(R, side)  
%converts an orthonormal 3x3 rotation matrix into ZYZ equivalent rotation angle components  
% Returns a vector of Euler angles [phi, theta, psi] where phi is the initial rotation  
% about the body Z axis, theta is the secondary rotation about the body Y  
% axis, and psi is the tertiary rotation about the body Z axis when  
% provided a 3x3 orthonormal rotation matrix.  
 if nargin < 2  
 side = 'upper';  
 end  
 if R == eye(3)  
 disp('[UNDEFINED WARNING] infinitely many representations for identity rotation.')  
 phi = NaN;  
 theta = NaN;  
 psi = NaN;  
 else  
 if strcmp(side, 'upper')  
 phi = atan2(R(2, 3), R(1, 3));  
 theta = atan2(sqrt(R(1, 3)^2 + R(2, 3)^2), R(3, 3));  
 psi = atan2(R(3, 2), -1 \* R(3, 1));  
 elseif strcmp(side, 'lower')  
 phi = atan2(-1 \* R(2, 3), -1 \* R(1, 3));  
 theta = atan2(-1 \* sqrt(R(1, 3)^2 + R(2, 3)^2), R(3, 3));  
 psi = atan2(-1 \* R(3, 2), R(3, 1));  
 else  
 disp("[UNKNOWN SIDE ARGUMENT] use either 'upper' for quadrants 1 and 2 or 'lower' for quadrants 3 and 4")  
 end  
 end  
 if abs(R(3,3)) == 1  
 disp('[SINGULARITY WARNING] theta is 0 or pi, resulting in singularity')  
 phi = NaN;  
 theta = NaN;  
 psi = NaN;  
 end  
end

This function takes in a 3x3 orthonormal rotation matrix *R* and an additional argument called “side” which can be either “upper” or “lower” and indicates the quadrants within which the rotation angle *θ* lies on the unit circle, with “upper” indicating and “lower” indicating . The function defaults to “upper” for this argument. If *R* is an identity matrix, a warning is printed that the angles cannot be determined and *NaN*s are returned for each of the Euler angles. Additionally, if as checked by looking at the *cos* component of *R33,* the user is warned of the singularity condition and *NaN*s are returned for each of the Euler angles.

The final function used for part c) calculates an equivalent RPY (*ZYX*) angle representation for an input *R*:

function [roll, pitch, yaw] = RotationMatrix2RPYAngles(R, side)  
%converts an orthonormal rotation matrix into RPY equivalent rotation angle components  
% Returns a vector of Euler angles [phi, theta, psi] where phi is the initial rotation  
% about the body fixed Z axis, theta is the secondary rotation about the body fixed Y  
% axis, and psi is the tertiary rotation about the body fixed X axis when  
% provided a 3x3 orthonormal rotation matrix.  
 if nargin < 2  
 side = 'right';  
 end  
 if strcmp(side, 'right')  
 roll = atan2(R(2, 1), R(1, 1));  
 pitch = atan2(-1 \* R(3, 1), sqrt(R(3, 2)^2 + R(3, 3)^2));  
 yaw = atan2(R(3, 2), R(3, 3));  
 elseif strcmp(side, 'left')  
 roll = atan2(-1 \* R(2, 1), -1 \* R(1, 1));  
 pitch = atan2(-1 \* R(3, 1), -1 \* sqrt(R(3, 2)^2 + R(3, 3)^2));  
 yaw = atan2(-1 \* R(3, 2), -1 \* R(3, 3));  
 else  
 disp("[UNKNOWN SIDE ARGUMENT] use either 'right' for quadrants 4 and 1 or 'left' for quadrants 2 and 3")  
 end  
 if abs(R(3,1)) == 1  
 disp('[SINGULARITY WARNING] theta is pi/2, resulting in singularity')  
 roll = NaN;  
 pitch = NaN;  
 yaw = NaN;  
 end  
end

Like the previous function, it accepts a 3x3 orthonormal rotation matrix *R* and an additional argument called “side” which can be either “right” or “left” and indicates the quadrants within which the rotation angle *θ* lies on the unit circle, with “right” indicating and “left” indicating . The function defaults to “left” for this argument. If as checked by looking at the *sin* component of *R31,* the user is warned of the singularity condition and *NaN*s are returned for each of the Euler angles.

*Answer / Test Cases:*

For each of the functions created, test cases were assigned to check each of the known edge cases as well as an arbitrary case where the expected output is known for each function.

For a), the function *RotationMatrix2AxisAngle()* was given three test cases:

**a.1** The identity rotation was tested, as the rotation is not well defined in axis-angle representation. The output of the program for this input is:

[UNDEFINED WARNING] infinitely many representations for identity rotation.

w =  
 NaN  
 NaN  
 NaN  
theta =  
 0

Which is expected because the rotation is not well defined.

**a.2** Test case 2 tests the conditions where the trace of matrix *R* is equal to -1 using the input matrices:

The output for these cases is shown below:

w =  
 0.7071  
 0.7071  
 0  
theta =  
 3.1416  
w =  
 0  
 -0.7071  
 0.7071  
theta =  
 3.1416  
w =  
 0.7071  
 0  
 0.7071  
theta =  
 3.1416

As expected, the function produces real outputs that each represent rotations of π about a rotation axis defined such that the axes of each frame following transformation is in the same or opposite direction of one of the axes of the original frame.

**a.3** The function was then tested for the 2D transformation matrix about the z-axis

With an input *θ* of *π/2*. The output for this case is:

w =  
 0  
 0  
 1  
theta =  
 1.5708

This matches expectations, as the calculated axis is the z-axis and the rotation angle is *π/2.*

For b), the function *RotationMatrix2Quaternion()* was given two test cases:

**b.1** The identity rotation was tested, which should not result in singularity for quaternion representation. The output of the code is:

q =  
 1  
 0  
 0  
 0

Which matches expectations and did not result in singularity.

**b.2** The second test case then tested the output for a 2D rotation about the z-axis

With an input *θ* of *π/2*. The output for this case is:

q =  
 0.9659  
 0  
 0  
 0.2588

This again matches expectations, and will be confirmed in question 2 where this quaternion will be the input to the inverse function and does produce the original *R*.

For c), *RotationAngle2ZYZAngles()* was given three test cases:

**c.1.1** The first test case was the Identity rotation, which is undefined in *ZYZ* Euler angle representation. The output for this case is:

[UNDEFINED WARNING] infinitely many representations for identity rotation.

phi =

NaN

theta =

NaN

psi =

NaN

Which matches expectations.

**c.1.2** The second test case for *ZYZ* Euler angles tested an otherwise arbitrary rotation with , which, as described previously, results in singularity. To do this, input RPY angles were converted into an equivalent rotation matrix using the known formula:

Using input , the output for this case is:

[SINGULARITY WARNING] theta is 0 or pi, resulting in singularity  
phi =  
 NaN  
theta =  
 NaN  
psi =  
 NaN

Which accurately warns the user and again matches expectations.

**c.1.3** The third test case represents a non-edge case arbitrary rotation and assigns values to *φ*, *θ*, and *ψ* then computes the rotation matrix *R* according to the known formula:

Then calls *RotationAngle2ZYZAngles()* to confirm the output Euler angles match the input angles. For input , the output of this case is:

phi =  
 0.5236  
theta =  
 0.7854  
psi =  
 -1.5708

These angles equal the input angles, so *RotationAngle2ZYZAngles()* passes all its test cases.

Continuing for c), the function *RotationMatrix2RPYAngles()* was provided two test cases:

**c.2.1** The first test case tests the singular case for RPY angles when the pitch angle *θ* is *π/2*. To do this, input RPY angles were converted into an equivalent rotation matrix using the known formula:

Which was then used as the input to *RotationMatrix2RPYAngles()*  to confirm the output Euler angles match the input angles. For input , the output of this case is:

[SINGULARITY WARNING] theta is pi/2, resulting in singularity  
roll =  
 NaN  
pitch =  
 NaN  
yaw =  
 NaN

Which matches expectations, as *θ* is *π/2* and should therefore result in singularity.

**c.2.2** The second test case uses an arbitrary rotation not expected to result in singularity. *R* is again calculated for a given , and this *R* is taken as the input to the function. For input , the output of this case is:

roll =  
 0.7854  
pitch =  
 -1.0472  
yaw =  
 -1.0472

These output angles match the inputs, so the function is working as intended.

*All Other Relevant Code:*

The script used to perform all case testing described in the previous section is shown below:

clc; clear; format compact;  
  
% -------------------------------------------------------------------------  
% Problem 1  
disp('-------------- Problem 1 --------------')  
  
% a) converting rotation matrix to axis-angle representation  
disp('-------------- a) --------------')  
  
% Test case 1: Identity rotation  
disp('---Case 1---')  
R = eye(3);  
[w, theta] = RotationMatrix2AxisAngle(R)  
  
% Test case 2: trace(R) == -1 (all forms)  
disp('---Case 2---')  
R = [0 1 0; 1 0 0; 0 0 -1];  
[w, theta] = RotationMatrix2AxisAngle(R)  
  
R = [-1 0 0; 0 0 -1; 0 1 0];  
[w, theta] = RotationMatrix2AxisAngle(R)  
  
R = [0 0 1; 0 -1 0; 1 0 0];  
[w, theta] = RotationMatrix2AxisAngle(R)  
  
% Test case 3: arbitrary rotation with known expected result  
disp('---Case 3---')  
angle = pi/2;  
R = [cos(angle) -sin(angle) 0; sin(angle) cos(angle) 0; 0 0 1];  
[w, theta] = RotationMatrix2AxisAngle(R)  
  
% b) converting rotation matrix to quaternion representation  
disp('-------------- b) --------------')  
  
% Test 1: Identity rotation  
disp('---Case 1---')  
R = eye(3)  
[q] = RotationMatrix2Quaternion(R)  
  
% Test 2: arbitrary rotation with known expected result  
disp('---Case 2---')  
angle = pi/6;  
R = [cos(angle) -sin(angle) 0; sin(angle) cos(angle) 0; 0 0 1]  
[q] = RotationMatrix2Quaternion(R)  
  
% c) converting rotation matrix to ZYZ and roll-pitch-yaw representation  
disp('-------------- c) --------------')  
  
% ZYZ  
disp('-------------- ZYZ --------------')  
  
% Test 1: Identity rotation  
disp('---Case 1---')  
R = eye(3);  
[phi, theta, psi] = RotationMatrix2ZYZAngles(R)  
  
% Test 2: theta = 0, pi  
disp('---Case 2---')  
phi = pi/6  
theta = 0  
psi = -pi/2  
R = [cos(phi)\*cos(theta)\*cos(psi) - sin(phi)\*sin(psi) -1\*cos(phi)\*cos(theta)\*sin(psi) - sin(phi)\*cos(psi) cos(phi)\*sin(theta);  
 sin(phi)\*cos(theta)\*cos(psi) + cos(phi)\*sin(psi) -1\*sin(phi)\*cos(theta)\*sin(psi) + cos(phi)\*cos(psi) sin(phi)\*sin(theta);  
 -1\*sin(theta)\*cos(psi) sin(theta)\*sin(psi) cos(theta)];  
  
[phi, theta, psi] = RotationMatrix2ZYZAngles(R)  
  
% Test 3: arbitrary rotation with known expected result  
disp('---Case 3---')  
phi = pi/6  
theta = pi/4  
psi = -pi/2  
R = [cos(phi)\*cos(theta)\*cos(psi) - sin(phi)\*sin(psi) -1\*cos(phi)\*cos(theta)\*sin(psi) - sin(phi)\*cos(psi) cos(phi)\*sin(theta);  
 sin(phi)\*cos(theta)\*cos(psi) + cos(phi)\*sin(psi) -1\*sin(phi)\*cos(theta)\*sin(psi) + cos(phi)\*cos(psi) sin(phi)\*sin(theta);  
 -1\*sin(theta)\*cos(psi) sin(theta)\*sin(psi) cos(theta)];  
  
[phi, theta, psi] = RotationMatrix2ZYZAngles(R)  
  
% RPY  
disp('-------------- RPY --------------')  
  
% Test 1: theta = pi/2  
disp('---Case 1---')  
roll = pi/4  
pitch = -pi/2  
yaw = -pi/3  
R = [cos(roll)\*cos(pitch) cos(roll)\*sin(pitch)\*sin(yaw) - sin(roll)\*cos(yaw) cos(roll)\*sin(pitch)\*cos(yaw) + sin(roll)\*sin(yaw);  
 sin(roll)\*cos(pitch) sin(roll)\*sin(pitch)\*sin(yaw) + cos(roll)\*cos(yaw) sin(roll)\*sin(pitch)\*cos(yaw) - cos(roll)\*sin(yaw);  
 -1\*sin(pitch) cos(pitch)\*sin(yaw) cos(pitch)\*cos(yaw)];  
  
[roll, pitch, yaw] = RotationMatrix2RPYAngles(R)  
  
% Test 2: arbitrary rotation with known expected result  
disp('---Case 2---')  
roll = pi/4  
pitch = -pi/3  
yaw = -pi/3  
R = [cos(roll)\*cos(pitch) cos(roll)\*sin(pitch)\*sin(yaw) - sin(roll)\*cos(yaw) cos(roll)\*sin(pitch)\*cos(yaw) + sin(roll)\*sin(yaw);  
 sin(roll)\*cos(pitch) sin(roll)\*sin(pitch)\*sin(yaw) + cos(roll)\*cos(yaw) sin(roll)\*sin(pitch)\*cos(yaw) - cos(roll)\*sin(yaw);  
 -1\*sin(pitch) cos(pitch)\*sin(yaw) cos(pitch)\*cos(yaw)];  
  
[roll, pitch, yaw] = RotationMatrix2RPYAngles(R)

**PA 2**

*Problem Description:*

This problem sees the creation of two MATLAB functions, respectively converting an axis-angle representation and a quaternion representation of a transformation to their equivalent rotation matrices.

*Method for Solution:*

For part a), the matrix exponential of rotations (i.e., Rodrigues’ formula), which is given by

is used to calculate the rotation matrix with inputs of an axis of rotation and an angle .

For part b), the rotation matrix R is calculated as a rotation about the unit axis by an angle , given by

Using this algorithm, a rotation matrix R is derived from a unit quaternion .

*Explanation of Program:*

The first function developed is used for part a) and converts an axis-angle representation of rotation to a 3x3 rotation matrix.

function [R] = AxisAngle2RotationMatrix(w, theta)  
%converts an angle theta and axis w into an equivalent orthonormal rotation matrix  
% receives a 3x1 unit column vector w representing an axis of rotation  
% and an angle theta in radians and returns an equivalent 3x3 orthonormal rotation matrix  
 w\_hat = Axis2SkewSymmetricMatrix(w);  
 R = eye(3) + w\_hat \* sin(theta) + w\_hat \* w\_hat \* (1 - cos(theta));  
end

*AxisAngle2RotationMatrix* accepts a 1x3 axis of rotation and an angle , and implements the above equation to return the corresponding rotation matrix. Because the exponential map is surjective onto SO(3), there are no edge cases; every axis-angle representation has a respective rotation matrix.

The second function is used for part b), and converts a quaternion representation of a rotation to a 3x3 rotation matrix.

function [R] = Quaternion2RotationMatrix(q)  
% Converts 4x1 column vector representing quaternion components [q0 q1 q2 q3]'  
% into equivalent 3x3 rotation matrix  
 R = zeros(3);  
 R(1,1) = q(1)^2 + q(2)^2 - q(3)^2 - q(4)^2;  
 R(1,2) = 2 \* (q(2)\*q(3) - q(1)\*q(4));  
 R(1,3) = 2 \* (q(1)\*q(3) + q(2)\*q(4));  
 R(2,1) = 2 \* (q(1)\*q(4) + q(2)\*q(3));  
 R(2,2) = q(1)^2 - q(2)^2 + q(3)^2 - q(4)^2;  
 R(2,3) = 2 \* (q(3)\*q(4) - q(1)\*q(2));  
 R(3,1) = 2 \* (q(2)\*q(4) - q(1)\*q(3));  
 R(3,2) = 2 \* (q(1)\*q(2) + q(3)\*q(4));  
 R(3,3) = q(1)^2 - q(2)^2 - q(3)^2 + q(4)^2;  
end

*Quaternion2RotationMatrix* accepts a 1x4 unit quaternion and implements the above equation to return the corresponding rotation matrix. There are no edge cases, as the quaternion representation is immune to singularities.

*Answer & Debugging / Test Cases:*

For both a) and b), there are no edge cases as each axis-angle and quaternion representation will always have a corresponding rotation matrix. Because of this, edge cases are not a worry, and each function is tested three times to ensure consistency. Test cases 1 and 3 for both a) and b) are the inverses of test cases used in problem 1.

For part a), the first test case is an axis and an angle . The function correctly returns

The second test case is an axis and an angle , as demonstrated on slide 14 of lecture W2-L2. The function correctly returns

The final test case is an axis and an angle . The function correctly returns

For part b), the first test case is the quaternion , which correctly returns

The second test case is the quaternion , which correctly returns

The third test case is the quaternion , which correctly returns

*All Other Relevant Code:*

clc; clear; format compact;  
% Problem 2  
disp('-------------- Problem 2 --------------')  
  
% a) convert axis-angle representation into equivalent rotation matrix  
disp('-------------- a) --------------')  
  
% Test cases  
w = [0.7071 0 0.7071]';  
theta = 3.1416;  
R = AxisAngle2RotationMatrix(w, theta)  
  
w = [0 0.866 0.5]';  
theta = 30\*pi/180;  
R = AxisAngle2RotationMatrix(w, theta)  
  
w = [0 0 1]';  
theta = 1.5708;  
R = AxisAngle2RotationMatrix(w, theta)  
  
% b) convert quaternion representation into equivalent rotation matrix  
disp('-------------- b) --------------')  
  
% Test cases  
q = [0.9659 0 0 0.2588]'  
R = Quaternion2RotationMatrix(q)  
  
q = [0.5 0.5 0.5 0.5]'  
R = Quaternion2RotationMatrix(q)  
  
q = [1 0 0 0]'  
R = Quaternion2RotationMatrix(q)

**PA 3**

*Problem Description:*

This problem requests the used to calculate the final configuration of a rigid body using screw theory. The user inputs an initial configuration and the individual components defining a screw axis, and the program should calculate and plot the configurations at increments of . The program should then calculate a new screw axis and distance to return the new “final” configuration to the origin, and plot the new screw axis.

*Method for Solution:*

In the method for solution section, we will walk through each step of this function.

1. The screw axis S is calculated as follows
2. The screw axis S and angle are then used to find the respective transformation

Where the rotation R is found with Rodrigues’ formula

and the translation p is given by

1. The final configuration is found by pre-multiplying the initial configuration with the transformation matrix
2. The transformation from the final configuration to the origin is found as and is converted to a screw axis as follows:
   1. If R = I, then there is no rotation and
   2. Otherwise, the axis angle method is used to find *w* and . *v* is calculated as where is defined as

Finally, the equations

yield the direction and pitch of the screw axis, and a point *q* can be found along the screw axis as:

Thus, the screw axis *S* can be converted into equivalent representation, and the quantities and can be used to plot the screw axis for a given transformation as described in this problem.

*Explanation of Program:*

To complete this problem, three helper functions were created to be used by the main MATLAB script. The first of these functions is a function used to transform the qh screw axis representation to the representation.

function [S] = qsh2S(q, s\_hat, h)  
% Receives point q along the screw axis as a 3x1 column vector, s\_hat as a 3x1 unit column  
% vector for direction of the screw axis, and single value h for the pitch  
% Returns a Screw axis of the form [w; v].  
 w = s\_hat;  
 v = cross(-1 \* s\_hat, q) + h \* s\_hat;  
 S = [w; v];  
end

*qsh2S* accepts a point q, a unit vector , and a scalar h, and returns the screw axis S.

The second function developed converts the screw axis to a transformation matrix.

function [T] = S2T(S,theta)  
%converts screw axis S = [w;v] and theta representation into an equivalent  
%4x4 transformation matrix  
 T = zeros(4);  
 w = S(1:3);  
 v = S(4:6);  
 w\_hat = Axis2SkewSymmetricMatrix(w);  
 trans\_vect = (eye(3)\*theta+(1-cos(theta))\*w\_hat+(theta-sin(theta))\*w\_hat^2)\*v;  
 rot\_mat = eye(3)+w\_hat\*sin(theta)+w\_hat^2\*(1-cos(theta));  
 T(1:3, 1:3) = rot\_mat;  
 T(1:3, 4) = trans\_vect;  
 T(4,4) = 1;  
end

*S2T* accepts a screw axis S and an angle , and returns the equivalent transformation matrix T.

The third and final helper function converts a transformation matrix to a screw axis.

function [S] = T2S(T)  
%converts 4x4 transformation matrix into equivalent screw axis S = [w;v]  
%and theta represnetation  
 R = T(1:3, 1:3);  
 if R == eye(3)  
 S = [0;0;0;T(1,4);T(2,4);T(3,4)];  
 else  
 [w, theta] = RotationMatrix2AxisAngle(R);  
 w\_hat = Axis2SkewSymmetricMatrix(w);  
 v = (eye(3)/theta-w\_hat/2+((1/theta)-cot(theta/2)/2)\*w\_hat^2)\*[T(1,4);T(2,4);T(3,4)];  
 S = [w;v];  
 end  
end

*T2S* accepts a 4x4 transformation matrix T and returns a screw axis S and an angle .

The final script for this problem combines the three above functions with the given values and data organization to solve for and plot the transformation matrices and screw axes relevant to the problem statement.

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]);  
  
% givens  
q = [0, 2, 0]';  
s\_hat = [0, 0, 1]';  
h = 2;  
theta = pi;  
T = [1 0 0 2;  
 0 1 0 0;  
 0 0 1 0;  
 0 0 0 1];  
  
% 1) convert qsh to S = [w; v]  
S = qsh2S(q, s\_hat, h)  
w= S(1:3);  
v = S(4:6);  
  
% 2) matrix exponential to get T (repeat for each theta)  
theta = linspace(0, theta, 5);  
T\_stor\_s = cell(1, length(theta));  
T\_stor\_b = cell(1, length(theta));  
for i = 1:length(theta)  
 T\_stor\_s{i} = S2T(S, theta(i));  
 T\_stor\_b{i} = T\_stor\_s{i} \* T;  
end  
  
% 3) compute the final transform  
T1 = T\_stor\_s{end} \* T  
  
% 4) find transform to get back to the origin  
To = inv(T1)  
  
% 5) use matrix logarithm to convert To to screw axis  
So = T2S(To)  
  
s\_hato = So(1:3) / vecnorm(So(1:3))  
  
% Finding Point q to fully define axis  
v = So(4:6);  
h = So(1:3)' \* So(4:6);  
s\_hat = s\_hato;  
q = Axis2SkewSymmetricMatrix(s\_hat) \* (v - h \* s\_hat)  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% Plot results  
figure  
plot3(0, 0, 0, '.k', 1, 0, 0,'.k', 0, 1, 0,'.k', 0, 0, 1,'.k', markersize=10)  
view(3)  
box on  
hold on  
grid on  
xlabel('x'), ylabel('y'), zlabel('z');  
  
% origin  
origin = [0 0 0];  
% length of frame vectors  
delta = 1;  
% x-axis  
line('XData', [origin(1) origin(1) + delta], 'YData', [origin(2) origin(2)],...  
 'ZData', [origin(3) origin(3)], 'Color','r','LineWidth',3);  
% y-axis  
line('XData', [origin(1) origin(1)], 'YData', [origin(2) origin(2) + delta],...  
 'ZData', [origin(3) origin(3)], 'Color','g','LineWidth',3);  
% z-axis  
line('XData', [origin(1) origin(1)], 'YData', [origin(2) origin(2)],...  
 'ZData', [origin(3) origin(3) + delta], 'Color','b','LineWidth',3);  
text(origin(1) - 0.2, origin(2) - 0.2, origin(3) - 0.2, '$\lbrace s \rbrace$');  
text(origin(1) + delta, origin(2), origin(3), '$x$');  
text(origin(1), origin(2) + delta, origin(3), '$y$');  
text(origin(1), origin(2), origin(3) + delta, '$z$');  
  
for i = 1:length(theta)  
 origin = T\_stor\_b{i}(1:3, 4)';  
 % x-axis  
 line('XData', [origin(1) origin(1) + T\_stor\_b{i}(1,1)], 'YData', [origin(2) origin(2) + T\_stor\_b{i}(2,1)],...  
 'ZData', [origin(3) origin(3) + T\_stor\_b{i}(3,1)], 'Color','r','LineWidth',3);  
 % y-axis  
 line('XData', [origin(1) origin(1) + T\_stor\_b{i}(1,2)], 'YData', [origin(2) origin(2) + T\_stor\_b{i}(2,2)],...  
 'ZData', [origin(3) origin(3) + T\_stor\_b{i}(3,2)], 'Color','g','LineWidth',3);  
 % z-axis  
 line('XData', [origin(1) origin(1) + T\_stor\_b{i}(1,3)], 'YData', [origin(2) origin(2) + T\_stor\_b{i}(2,3)],...  
 'ZData', [origin(3) origin(3) + T\_stor\_b{i}(3,3)], 'Color','b','LineWidth',3);  
 % frame label  
 text(origin(1) - 0.2 \* T\_stor\_b{i}(1,1), origin(2) - 0.2 \* T\_stor\_b{i}(2,2), origin(3) - 0.2 \* T\_stor\_b{i}(3,3), strcat('$\lbrace b \rbrace$', ', $\theta = $', string(theta(i))));  
 % axes labels  
 text(origin(1) + T\_stor\_b{i}(1,1), origin(2) + T\_stor\_b{i}(2,1), origin(3) + T\_stor\_b{i}(3,1), '$x$');  
 text(origin(1) + T\_stor\_b{i}(1,2), origin(2) + T\_stor\_b{i}(2,2), origin(3) + T\_stor\_b{i}(3,2), '$y$');  
 text(origin(1) + T\_stor\_b{i}(1,3), origin(2) + T\_stor\_b{i}(2,3), origin(3) + T\_stor\_b{i}(3,3), '$z$');  
end  
  
% plotting screw axis  
origin = q';  
% length of frame vectors  
delta = 1;  
C = 1;  
% x-axis  
line('XData', [origin(1) origin(1) + C \* abs(s\_hat(1))], 'YData', [origin(2) origin(2)],...  
 'ZData', [origin(3) origin(3)], 'Color','k','LineWidth',3);  
line('XData', [origin(1) origin(1) - C \* abs(s\_hat(1))], 'YData', [origin(2) origin(2)],...  
 'ZData', [origin(3) origin(3)], 'Color','k','LineWidth',3);  
% y-axis  
line('XData', [origin(1) origin(1)], 'YData', [origin(2) origin(2) + C \* abs(s\_hat(2))],...  
 'ZData', [origin(3) origin(3)], 'Color','k','LineWidth',3);  
line('XData', [origin(1) origin(1)], 'YData', [origin(2) origin(2) - C \* abs(s\_hat(2))],...  
 'ZData', [origin(3) origin(3)], 'Color','k','LineWidth',3);  
% z-axis  
line('XData', [origin(1) origin(1)], 'YData', [origin(2) origin(2)],...  
 'ZData', [origin(3) origin(3) + C \* 8 \* abs(s\_hat(3))], 'Color','k','LineWidth',3);  
line('XData', [origin(1) origin(1)], 'YData', [origin(2) origin(2)],...  
 'ZData', [origin(3) origin(3) - C \* abs(s\_hat(3))], 'Color','k','LineWidth',3);  
text(origin(1) - 0.2, origin(2) - 0.2, origin(3) - 0.2, '$\hat{s}\_{sb}$');

*Answer:*

Running the above code, a 3D plot containing all the frames with labels and the screw axis of the transformation between the final body and spatial frame is produced. Screen captures of this plot from various angles are shown below.

Chart, diagram

Description automatically generated

Fig. 1

Chart

Description automatically generated

Fig. 2

Chart, line chart

Description automatically generated

Fig. 3

These plots show the expected relative pose of the initial *{b}* frame relative to the origin, as well as the pose of *{b}* at its final configuration with , as well as intermediate locations with /4, 2, and 3. Fig. 3 clearly shows that the frame *{b}* is rotating about an axis through the given *q* of (0, 2, 0) with the given orientation of (0, 0, 1) to a final displacement of . Fig. 1 and Fig. 2 show the translation of *{b}* in direction along the screw axis, and the final z-coordinate of *{b}* is 6.28, or 2π, which is equal to the given pitch *h* of 2 times the final displacement of , so the movement of *{b}* about the given screw axis is as expected. Based on this, we can confidently state that the first part of the question calculating the *T* matrices for the final and intermediate transforms, is correct. The final transformation matrix, *T1*, is calculated to be:

Which again matches the graphical representations.

For the transform that takes frame *{b}* back to the origin, the transform, henceforth referred to as *To*, was calculated as:

This was then converted into a screw axis representation, *So*, using the matrix exponential of transformations to produce the screw axis:

And that describe the transformation between the final configuration of *{b}* and the origin. Then, this representation was converted to an equivalent representation according to the previously constructed equations, yielding:

These results also all make sense given the prior results, as rotation about the negative z-axis an amount π will return *{b}* to the orientation of *{s}* and a pitch of -2 means that for this rotation, the frame will decrease in the z-direction by 6.28, its current z-coordinate before transformation, and return to the x-y plane. Then, the calculated point on the axis *q* is found to be (-1, 2, 0) which is the midpoint between the origin and *{b}* in the plane perpendicular to the screw axis, and for a rotation of π about this axis, *{b}* will coincide with the origin, as anticipated.

*All Other Relevant Code:*

The functions *RotationMatrix2Axisangle* and *Axis2SkewSymmetricMatrix* are called, as shown in problem 1.

Contributions of Group Members

Jared Rosenbaum completed Programming Assignment (PA) Q2, Homework Assignment (HA) Q3, HA Q8, and HA Q9.

Steven Swanbeck completed PA Q1, HA Q1, HA Q2, HA Q4, and HA Q6.

Responsibility was shared on PA Q3, HA Q5, and HA Q7.

References

1. 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>
2. Lynch, K. M., & Park, F. C. (2017). Modern Robotics: Mechanics, Planning, and Control. Cambridge University Press.