Student Name: Linxuan Jia *Penn ID: 61896199*

1 Homogeneous Transforms (20 points)

1. The Rotation Matrix is:

$$R_1 = \begin{bmatrix} b_1^a & b_2^a & b_3^a \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The Coordinate Matrix is:

$$O_1 = \begin{bmatrix} 0 \\ 3 \\ 0 \end{bmatrix}$$

Hence, the Transformation Matrix is:

$$T_1 = \begin{bmatrix} R_1 & O_1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. The Rotation Matrix is:

$$R_2 = \begin{bmatrix} b_1^a & b_2^a & b_3^a \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

The Coordinate Matrix is:

$$O_2 = \begin{bmatrix} 0 \\ 2 \\ 0 \end{bmatrix}$$

Hence, the Transformation Matrix is:

$$T_2 = \begin{bmatrix} R_2 & O_2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.

$$T_3 = T_1 T_2 = \begin{bmatrix} -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} & \sqrt{2} \\ \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} & 3+\sqrt{2} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2 Rotation Matrix Sudoku (20 points)

see code 2.py

A rotation matrix R has following properties:

1. **Orthogonomality**: $RR^T = R^T R = I$;
2. **Unit vectors**: the row and column vectors of R are all unit vectors;
3. **Cross-multiplication**: every cross-multiplication of two column vectors is equal to the third column;
4. **Determinant**: The determinant of the matrix is 1(right-hand) or -1(left-hand)

For clarity, I set the unknown elements as $x[i]$, like below:

$$R = \begin{bmatrix} x[1] & 0.892 & 0.423 \\ x[2] & x[3] & x[5] \\ -0.186 & x[4] & x[6] \end{bmatrix}$$

From the property one, we know that (**below calculations are by hand**):

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x[1] & 0.892 & 0.423 \\ x[2] & x[3] & x[5] \\ -0.186 & x[4] & x[6] \end{bmatrix} \begin{bmatrix} x[1] & x[2] & -0.186 \\ 0.892 & x[3] & x[4] \\ 0.423 & x[5] & x[6] \end{bmatrix}$$

$$= \begin{bmatrix} x_1^2 + 0.974593 & x_1x_2 + 0.892x_3 + 0.423x_5 & -0.186x_1 + 0.892x_4 + 0.423x_6 \\ x_1x_2 + 0.892x_3 + 0.423x_5 & x_2^2 + x_3^2 + x_5^2 & -0.186x_2 + x_3x_4 + x_5x_6 \\ -0.186x_1 + 0.892x_4 + 0.423x_6 & -0.186x_2 + x_3x_4 + x_5x_6 & x_4^2 + x_6^2 + 0.034596 \end{bmatrix}$$

$$= \begin{bmatrix} x[1] & x[2] & -0.186 \\ 0.892 & x[3] & x[4] \\ 0.423 & x[5] & x[6] \end{bmatrix} \begin{bmatrix} x[1] & 0.892 & 0.423 \\ x[2] & x[3] & x[5] \\ -0.186 & x[4] & x[6] \end{bmatrix}$$

$$= \begin{bmatrix} x_1^2 + x_2^2 + 0.034596 & 0.892x_1 + x_2x_3 - 0.186x4 & 0.423x_1 + x_2x_5 - 0.186x_6 \\ 0.892x_1 + x_2x_3 - 0.186x_4 & x_3^2 + x_4^2 + 0.795664 & x_3x_5 + x_4x_6 + 0.377316 \\ 0.423x_1 + x_2x_5 - 0.186x_6 & x_3x_5 + x_4x_6 + 0.377316 & x_5^2 + x_6^2 + 0.178929 \end{bmatrix}$$

Hence, we will have $6+6 = 12$ equations (there are repeating equations in positions symmetric along the diagonals):

$$x_1^2 + 0.974593 = 1 \tag{1}$$
$$x_2^2 + x_3^2 + x_5^2 = 1 \tag{2}$$
$$x_4^2 + x_6^2 + 0.034596 = 1 \tag{3}$$
$$x_1x_2 + 0.892x_3 + 0.423x_5 = 0 \tag{4}$$
$$-0.186x_1 + 0.892x_4 + 0.423x_6 = 0 \tag{5}$$
$$-0.186x_2 + x_3x_4 + x_5x_6 = 0 \tag{6}$$
$$x_1^2 + x_2^2 + 0.034596 = 1 \tag{7}$$
$$x_3^2 + x_4^2 + 0.795664 = 1 \tag{8}$$
$$x_5^2 + x_6^2 + 0.178929 = 1 \tag{9}$$
$$0.892x_1 + x_2x_3 - 0.186x4 = 0 \tag{10}$$
$$0.423x_1 + x_2x_5 - 0.186x_6 = 0 \tag{11}$$
$$x_3x_5 + x_4x_6 + 0.377316 = 0 \tag{12}$$

1. from equation (1), we can get that $x[1] = \pm 0.159$;
2. put it into equation (7), we can get that $x[2] = \pm 0.970$;

3. use equations (2), (3), (8), (9) to solve $x[3]^2$ to $x[6]^2$:

$$x_3^2 + x_5^2 = 1 - 0.970^2 \tag{13}$$
$$x_4^2 + x_6^2 = 1 - 0.034596 \tag{14}$$
$$x_3^2 + x_4^2 = 1 - 0.795664 \tag{15}$$
$$x_5^2 + x_6^2 = 1 - 0.178929 \tag{16}$$

```python
# to solve x[3] ~ x[6]
## matrix representation of the equations
### coefficient matrix
A = np.array([
    [1, 1, 0, 0],
    [0, 0, 1, 1],
    [1, 0, 1, 0],
    [0, 1, 0, 1]
])
### constant matrix
B = np.array([1 - 795664, 1 - 0.178928, 1 - 0.970**2, 1 - 0.034596])

## solve the linear equations
solution = np.linalg.solve(A, B)
print("The solution is:", solution)
```

However, since the coefficient matrix is singular(rank=3), there is no unique solution (more than one solution):

```
Traceback (most recent call last):
  File "c:\Upenn\24Spring\MEAM6200\Proj0\test.py", line 24, in <module>
    solution = np.linalg.solve(A, B)
               ^^^^^^^^^^^^^^^^^^^^^
  File "<__array_function__ internals>", line 200, in solve
  File "C:\Anaconda\Anaconda\Lib\site-packages\numpy\linalg\linalg.py", line 386, in solve
    r = gufunc(a, b, signature=signature, extobj=extobj)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Anaconda\Anaconda\Lib\site-packages\numpy\linalg\linalg.py", line 89, in _raise_linalgerror_singular
    raise LinAlgError("Singular matrix")
numpy.linalg.LinAlgError: Singular matrix
```

4. considering add other equations (equations (4),(5),(10),(11)) into the group of the equation:

$$\begin{cases} x_3^2 + x_5^2 = 1 - 0.970^2 \\ x_4^2 + x_6^2 = 1 - 0.034596 \\ x_3^2 + x_4^2 = 1 - 0.795664 \\ x_5^2 + x_6^2 = 1 - 0.178929 \\ 0.892x_3 + 0.423x_5 = -x_1x_2 \\ 0.892x_4 + 0.423x_6 = 0.186x_1 \\ x_2x_3 - 0.186x_4 = -0.892x_1 \\ x_2x_5 - 0.186x_6 = -0.423x_1 \end{cases}$$

since $x[1] = \pm 0.159, x[2] = \pm 0.970$, I will discuss each combination:

(a) if x[1]=0.159, x[2]=0.970, we have:

$$\begin{cases} 0.892x_3 + 0.423x_5 = -0.15423 \\ 0.892x_4 + 0.423x_6 = 0.029574 \\ 0.970x_3 - 0.186x_4 = -0.141828 \\ 0.970x_5 - 0.186x_6 = -0.067257 \end{cases}$$

Together with equations (13), (14), (15), and (16), we can get the following solutions:
left-hand frame (det=-1):

$$x[1] = 0.159, x[2] = 0.970, x[3] = -0.221, x[4] = -0.394, x[5] = 0.103, x[6] = 0.900$$

right-hand frame (det=1):

$$x[1] = 0.159, x[2] = 0.970, x[3] = -0.061, x[4] = 0.448, x[5] = -0.237, x[6] = -0.875$$

(b) if x[1]=0.159, x[2]=-0.970

$$\begin{cases} 0.892x_3 + 0.423x_5 = 0.15423 \\ 0.892x_4 + 0.423x_6 = 0.029574 \\ -0.970x_3 - 0.186x_4 = -0.141828 \\ -0.970x_5 - 0.186x_6 = -0.067257 \end{cases}$$

Together with equations (13), (14), (15), and (16), we can get the following solutions:
left-hand frame (det=-1):

$$x[1] = -0.159, x[2] = 0.970, x[3] = 0.061, x[4] = -0.448, x[5] = 0.237, x[6] = 0.875$$

right-hand frame (det=1):

$$x[1] = -0.159, x[2] = 0.970, x[3] = 0.221, x[4] = 0.394, x[5] = -0.103, x[6] = -0.900$$

(c) if x[1]=-0.159, x[2]=0.970

$$\begin{cases} 0.892x_3 + 0.423x_5 = 0.15423 \\ 0.892x_4 + 0.423x_6 = -0.029574 \\ 0.970x_3 - 0.186x_4 = 0.141828 \\ 0.970x_5 - 0.186x_6 = 0.067257 \end{cases}$$

Together with equations (13), (14), (15), and (16), we can get the following solutions:
left-hand frame (det=-1):

$$x[1] = 0.159, x[2] = -0.970, x[3] = 0.061, x[4] = 0.448, x[5] = 0.237, x[6] = -0.875$$

right-hand frame (det=1):

$$x[1] = 0.159, x[2] = -0.970, x[3] = 0.221, x[4] = -0.394, x[5] = -0.103, x[6] = 0.900$$

(d) if x[1]=-0.159, x[2]=-0.970

$$\begin{cases} 0.892x_3 + 0.423x_5 = -0.15423 \\ 0.892x_4 + 0.423x_6 = -0.029574 \\ -0.970x_3 - 0.186x_4 = 0.141828 \\ -0.970x_5 - 0.186x_6 = 0.067257 \end{cases}$$

Together with equations (13), (14), (15), and (16), we can get the following solutions:
left-hand frame (det=-1):

$$x[1] = -0.159, x[2] = -0.970, x[3] = -0.221, x[4] = 0.394, x[5] = 0.103, x[6] = -0.900$$

right-hand frame (det=1):

$$x[1] = -0.159, x[2] = -0.970, x[3] = -0.061, x[4] = -0.448, x[5] = -0.237, x[6] = 0.875$$

Code 2.py:

```
import numpy as np
from scipy.optimize import fsolve

print("1. solve possible values of x[1] and x[2]:")
# 1. to solve x[1]
x_1 = np.round(np.sqrt(1-0.892**2-0.423**2),3)
print("x[1] = ±", x_1)


# 2. to solve x[2]
x_2 = np.round(np.sqrt(1-x_1**2-(-0.186)**2),3)
print("x[2] = ±", x_2)


# # 3. to solve x[3] ~ x[6] using equations 2,3,8,9
# # 3.1 coefficient matrix
# A = np.array([
#     [1, 1, 0, 0],
#     [0, 0, 1, 1],
#     [1, 0, 1, 0],
#     [0, 1, 0, 1]
# ])
# # 3.2 constant matrix
# B = np.array([1 - 795664, 1 - 0.178928, 1 - 0.970**2, 1 - 0.034596])

# # 3.3 solve the linear equations
# print("The rank of the matrix is:", np.linalg.matrix_rank(A))
# solution = np.linalg.solve(A, B)
# print("The solution is:", solution)


# 4. solve unlinear equations:
print("\n2. solve possible values of x[3] and x[4]:")
x1 = [x_1, -x_1, x_1, -x_1]   # 0.159
x2 = [x_2, x_2, -x_2, -x_2]   # 0.970
x3 = []
x4 = []

def equations(vars):
    x3, x4 = vars
    eq1 = -0.9695 * x4 + 0.186 * x3 - 0.423
    eq2 = 0.892**2 + x3**2 + x4**2 - 1
    return [eq1, eq2]

initial_guess1 = [-1, -1]
solution1 = fsolve(equations, initial_guess1)
x_3, x_4 = solution1
x3.append(np.round(x_3,3))
x4.append(np.round(x_4,3))
initial_guess2 = [1, 1]
solution2 = fsolve(equations, initial_guess2)
x_3, x_4 = solution2
```

```
x3.append(np.round(x_3,3))
x4.append(np.round(x_4,3))

print("x3:",x3,"; x4:",x4)

# right-hand frame results
print("\nright-hand frame results:")
for x_1,x_2 in zip(x1,x2):
    for x_3,x_4 in zip(x3,x4):
        for c1,c2 in zip([1,-1,1,-1],[1,1,-1,-1]):
            x_3_ = x_3 * c1
            x_4_ = x_4 * c2
            _, x_5, x_6 = np.cross(np.array([x_1, x_2, -0.186]), np.array([0.892, x_3_, x_4_]))
            x_5 = np.round(x_5,3)
            x_6 = np.round(x_6,3)
            R = np.array([
                        [x_1, 0.892, 0.423],
                        [x_2, x_3_, x_5],
                        [-0.186, x_4_, x_6]
                        ])
            det = np.round(np.linalg.det(R),3)
            if np.abs(np.abs(det)-1)<0.005:
                print(x_1,x_2,x_3_,x_4_,x_5,x_6,";",np.round(np.linalg.det(R),3))

# left-hand frame results
print("left-hand frame results:")
for x_1,x_2 in zip(x1,x2):
    for x_3,x_4 in zip(x3,x4):
        for c1,c2 in zip([1,-1,1,-1],[1,1,-1,-1]):
            x_3_ = x_3 * c1
            x_4_ = x_4 * c2
            _, x_5, x_6 = -np.cross(np.array([x_1, x_2, -0.186]), np.array([0.892, x_3_, x_4_]))
            x_5 = np.round(x_5,3)
            x_6 = np.round(x_6,3)
            R = np.array([
                        [x_1, 0.892, 0.423],
                        [x_2, x_3_, x_5],
                        [-0.186, x_4_, x_6]
                        ])
            det = np.round(np.linalg.det(R),3)
            if np.abs(np.abs(det)-1)<0.005:
                print(x_1,x_2,x_3_,x_4_,x_5,x_6,";",np.round(np.linalg.det(R),3))
```

Code results:

```
1. solve possible values of x[1] and x[2]:
x[1] = ± 0.159
x[2] = ± 0.97

2. solve possible values of x[3] and x[4]:
x3: [-0.061, 0.222] ; x4: [-0.448, -0.394]

right-hand frame results:
0.159 0.97 -0.061 0.448 -0.237 -0.875 ; 1.001
-0.159 0.97 0.222 0.394 -0.103 -0.901 ; 1.001
0.159 -0.97 0.222 -0.394 -0.103 0.901 ; 1.001
-0.159 -0.97 -0.061 -0.448 -0.237 0.875 ; 1.001
left-hand frame results:
0.159 0.97 -0.222 -0.394 0.103 0.901 ; -1.001
-0.159 0.97 0.061 -0.448 0.237 0.875 ; -1.001
0.159 -0.97 0.061 0.448 0.237 -0.875 ; -1.001
-0.159 -0.97 -0.222 0.394 0.103 -0.901 ; -1.001
```

3 Rodrigues' Formula (20 points)

1. Not every rotation has uniquely defined axes and angles. This depends largely on the nature of the rotation and the representation used.

   (a) **Multiple solutions**: For some rotations, there may be more than one combination of axes and angles that can describe the same rotation transformation;

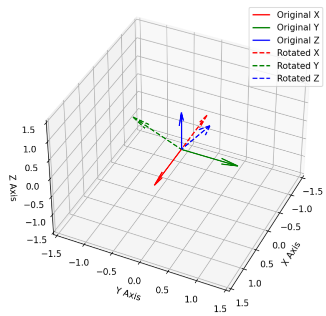
   (b) **Periodicity**: Angles are periodic, hence, a given rotation can be expressed in terms of a given axis and a specific angle, or in terms of the same axis and angle plus $n \times 360$ degree;

   (c) **Coordinate frame**: Different choices of coordinate frame may result in different axis and angle representations, this can be described by a homogeneous transformation matrix like question 1;

2. **see code 3-2.py**

```
import numpy as np
import matplotlib.pyplot as plt

# rotation matrix R
R = np.array([[-1, 0, 0],
              [0, -np.cos(np.pi/6), np.sin(np.pi/6)],
              [0, np.sin(np.pi/6), np.cos(np.pi/6)]])

# create 3d figure
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# axes before rotation
x_axis = np.array([1, 0, 0])
y_axis = np.array([0, 1, 0])
z_axis = np.array([0, 0, 1])
ax.quiver(0, 0, 0, x_axis[0], x_axis[1], x_axis[2], color='r', label='Original X')
ax.quiver(0, 0, 0, y_axis[0], y_axis[1], y_axis[2], color='g', label='Original Y')
ax.quiver(0, 0, 0, z_axis[0], z_axis[1], z_axis[2], color='b', label='Original Z')

# axes after rotation
x_axis_rot = R @ x_axis
y_axis_rot = R @ y_axis
z_axis_rot = R @ z_axis
ax.quiver(0, 0, 0, x_axis_rot[0], x_axis_rot[1], x_axis_rot[2], color='r', linestyle='dashed', label=
ax.quiver(0, 0, 0, y_axis_rot[0], y_axis_rot[1], y_axis_rot[2], color='g', linestyle='dashed', label=
ax.quiver(0, 0, 0, z_axis_rot[0], z_axis_rot[1], z_axis_rot[2], color='b', linestyle='dashed', label=

# set attributes
ax.set_xlim([-1.5, 1.5])
ax.set_ylim([-1.5, 1.5])
ax.set_zlim([-1.5, 1.5])
ax.set_xlabel('X Axis')
ax.set_ylabel('Y Axis')
ax.set_zlabel('Z Axis')
ax.legend()

plt.show()
```

$$R = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -cos(\frac{\pi}{6}) & sin(\frac{\pi}{6}) \\ 0 & sin(\frac{\pi}{6}) & cos(\frac{\pi}{6}) \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -\frac{\sqrt{3}}{2} & \frac{1}{2} \\ 0 & \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix}$$



(a) **axis-angle representation**

$$Tr(R) = -1 - cos(\frac{\pi}{6}) + cos(\frac{\pi}{6}) = -1$$

$$\theta = arccos(\frac{Tr(R) - 1}{2}) = arccos(-1) = \pi$$

$$u_x = \sqrt{\frac{1 + R_{11}}{2}} = \sqrt{\frac{1 + (-1)}{2}} = 0$$

$$u_y = \sqrt{\frac{1 + R_{22}}{2}} = \sqrt{\frac{1 + (-\frac{\sqrt{3}}{2})}{2}} \approx 0.2588$$

$$u_z = \sqrt{\frac{1 + R_{33}}{2}} = \sqrt{\frac{1 + (\frac{\sqrt{3}}{2})}{2}} \approx 0.9659$$

Hence, we have the axis as $[0, 0.2588, 0.9659]$ or $[0, -0.2588, -0.9659]$, and the angle is π

(b) **quaternion**
Since the definition of quaternion is:

$$q = [cos(\frac{\theta}{2}), u_x sin(\frac{\theta}{2}), u_y sin(\frac{\theta}{2}), u_z sin(\frac{\theta}{2})]$$

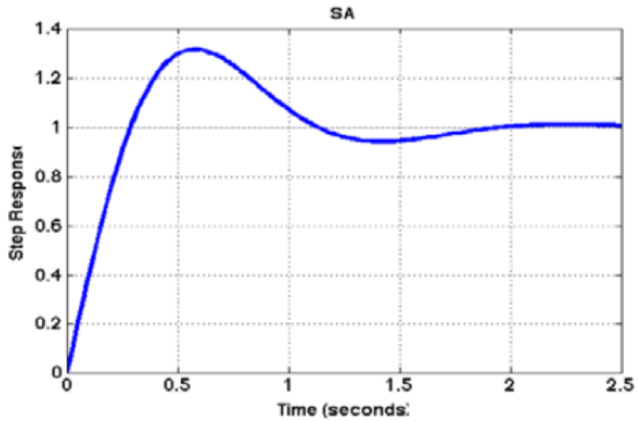
and we have $\theta = \pi, u_x = 0, u_y = 0.2588, u_z = 0.9659$ or $[0, -0.2588, -0.9659]$; Hence:

$$q = [0, 0, 0.2588, 0.9659]$$

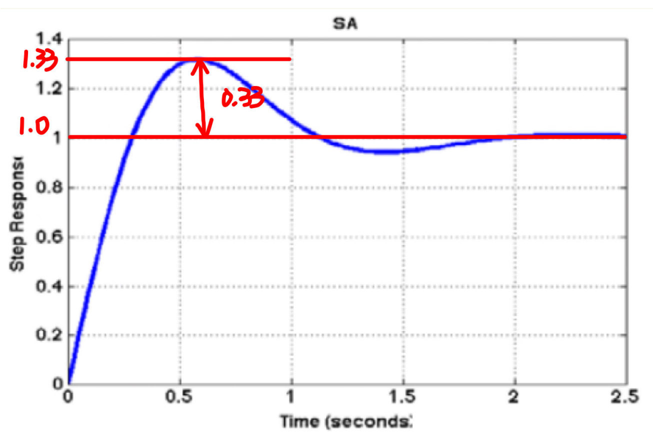
or

$$q = [0, 0, -0.2588, -0.9659]$$

4 Signal Processing (20 points)



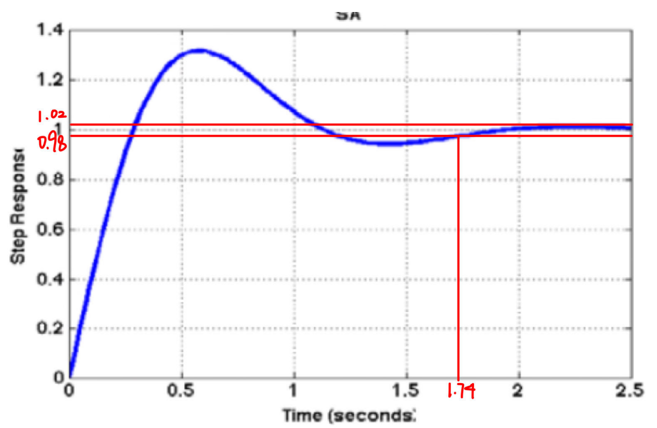
1. This is an under-damped signal since it oscillates and overshoots. To make it clear, in a step response:
   (a) **under-damped signal**($\zeta < 1$): (1)overshoots, (2)oscillates;
   (b) **critically-damped signal** ($\zeta = 1$): (1)no oscillates, (2)reaches the final steady-state value as fast as possible without overshooting;
   (c) **over-damped signal** ($\zeta > 1$): (1)no oscillates, (2)responds slowly and takes a long time to reach the final steady-state value;
2. (a) **percentage overshoot**: max value≈1.33, steady-state value≈1.0, hence percentage overshoot $\approx \frac{1.33-1.0}{1.0} =$ 33%



   (b) **damping ratio**: The value of the first peak is approximately 0.33 (P_1), and the value of the second peak is approximately 0.01 (P_2); Hence, using the Peak Method, we can have:

$$\zeta = \frac{-ln(\frac{PO}{100})}{\sqrt{\pi^2 + ln^2(\frac{PO}{100})}} = \frac{-ln(0.33)}{\sqrt{\pi^2 + ln^2(0.33)}} = \frac{1.108}{\sqrt{\pi^2 + (-1.108)^2}} \approx 0.332$$

(c) **2% settling time**:2% settling time means the time spent from initial state to 2% range around the steady state value



From the graph, we can see that the 2% settling time is about 1.74s.

5 Transforms and and Python Plotting (30 points)

1.

$$d(t) = \begin{bmatrix} cos(0.1t) \\ sin(0.12t) \\ sin(0.08t) \end{bmatrix} \; R(t) = \begin{bmatrix} cos(t) & -sin(t) & 0 \\ sin(t) & cos(t) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$v(t) = \frac{d}{dt}d(t) = \begin{bmatrix} -0.1sin(0.1t) \\ 0.12cos(0.12t) \\ 0.08cos(0.08t) \end{bmatrix}$$

2. The homogeneous matrix of the robot with respect to the world frame is:

$$H_r^w = \begin{bmatrix} cos(t) & -sin(t) & 0 & cos(0.1t) \\ sin(t) & cos(t) & 0 & sin(0.12t) \\ 0 & 0 & 1 & sin(0.08t) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The homogeneous matrix of the camera with respect to the robot frame is:

$$H_c^r = \begin{bmatrix} 1 & 0 & 0 & c \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

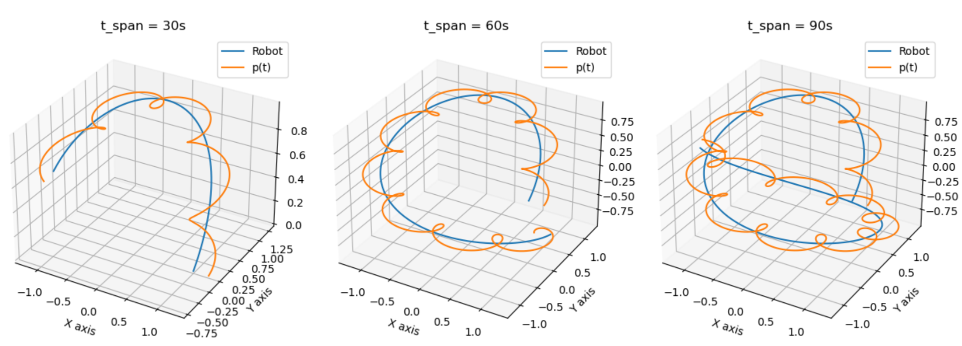
Hence the homogeneous matrix of the camera with respect to the world frame is:

$$H_c^w = H_r^w H_c^r = \begin{bmatrix} \cos(t) & -\sin(t) & 0 & \cos(0.1t) + c\cos(t) \\ \sin(t) & \cos(t) & 0 & \sin(0.12t) + c\sin(t) \\ 0 & 0 & 1 & \sin(0.08t) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Hence, we have:

$$^{A}p(t) = \begin{bmatrix} \cos(0.1t) + c\cos(t) \\ \sin(0.12t) + c\sin(t) \\ \sin(0.08t) \end{bmatrix}$$

3. **see code 5-c.py**
The plots with different time spans are shown below(frequency: 100Hz)



4. **see code 5-d.py**
   Since the gravity matrix under the world frame should always be the same, we can have the following equation:

$$g_w = R(1)g_r(1) = R(5)g_r(5)$$

Since $R(t)$ is rotation matrix, $R(t)^{-1} = R(t)^T$, we have:

$$g_r(5) = R(5)^T R(1) g_r(1) = \begin{bmatrix} -1.4001 \\ 0.2442 \\ -9.7 \end{bmatrix}$$

Codes:

5-c.py:

```python
import matplotlib.pyplot as plt
import numpy as np

# set three different t_span values
t_span_values = [30, 60, 90]
freq = 100   # hz

# iterate each t_span
for i, t_span in enumerate(t_span_values):
    t = np.linspace(0, t_span, t_span * freq)  # time array

    d_x = np.cos(0.1 * t)
    d_y = np.sin(0.12 * t)
    d_z = np.sin(0.08 * t)

    Ap_x = 0.25 * np.cos(t) + np.cos(0.1 * t)
    Ap_y = 0.25 * np.sin(t) + np.sin(0.12 * t)
    Ap_z = np.sin(0.08 * t)

    # create subplot
    plt.subplot(1, len(t_span_values), i + 1, projection='3d')

    # Robot position
    plt.plot(d_x, d_y, d_z, label='Robot')

    # A_p(t) position
    plt.plot(Ap_x, Ap_y, Ap_z, label='p(t)')

    # Labels and title
    plt.xlabel('X axis')
    plt.ylabel('Y axis')
    plt.clabel('Z axis')
    plt.title(f't_span = {t_span}s')
    plt.legend()

plt.tight_layout()
plt.show()
```

5-d.py:

```python
import numpy as np

# the gravity vector under robot frame at t=1
gv_r_1 = np.array([1.1, 0.9, -9.7])

# calculate the rotation matrix R(t) as a function of t
def calculate_rotation_matrix(t):
    R = np.array([
        [np.cos(t), -np.sin(t), 0],
        [np.sin(t), np.cos(t), 0],
        [0, 0, 1]
        ])
    return R

# calculate the rotation matrix at t=1 and t=5
R_1 = calculate_rotation_matrix(1)
R_5 = calculate_rotation_matrix(5)

# Since the Gravity vector under the world frame should always be the same,
# we have: gv_w = R_5 @ gv_r_5 = R_1 @ gv_r_1, hence we have:
# gv_r_5 = R_5^T @ R_1 @ gv_r_1

gv_r_5 = R_5.T @ R_1 @ gv_r_1

print(gv_r_5)
```