(a) A reflection matrix for a line at angle θ

$$R_{reflect}(\theta) = \begin{bmatrix} \cos(2\theta) & \sin(2\theta) \\ \sin(2\theta) & -\cos(2\theta) \end{bmatrix} \quad (1)$$

If reflect about two lines at angle α, β

$$T = R_{reflect}(\beta) \cdot R_{reflect}(\alpha)$$

$$= \begin{bmatrix} \cos(2\beta) & \sin(2\beta) \\ \sin(2\beta) & -\cos(2\beta) \end{bmatrix} \begin{bmatrix} \cos(2\alpha) & \sin(2\alpha) \\ \sin(2\alpha) & -\cos(2\alpha) \end{bmatrix}$$

$$= \begin{bmatrix} \cos(2\beta-2\alpha) & -\sin(2\beta-2\alpha) \\ \sin(2\beta-2\alpha) & \cos(2\beta-2\alpha) \end{bmatrix}$$

$$= \begin{bmatrix} \cos(2(\beta-\alpha)) & -\sin(2(\beta-\alpha)) \\ \sin(2(\beta-\alpha)) & \cos(2(\beta-\alpha)) \end{bmatrix}$$

For rotation, when $\theta = 2(\beta-\alpha)$

$$R_{rotate}(2(\beta-\alpha)) = \begin{bmatrix} \cos(2(\beta-\alpha)) & -\sin(2(\beta-\alpha)) \\ \sin(2(\beta-\alpha)) & \cos(2(\beta-\alpha)) \end{bmatrix}$$

$$T = R_{rotate}(2(\beta-\alpha))$$

Thus, reflect about α followed by about β is equivalent to a rotation of $2(\beta-\alpha)$

(b)

A skew-symmetric matrix $\hat{s}$, associated with a vector $s = [s_1, s_2, s_3]^T$ is defined as $\hat{s} = \begin{bmatrix} 0 & -s_3 & s_2 \\ s_3 & 0 & -s_1 \\ -s_2 & s_1 & 0 \end{bmatrix}$

Compute the cross product $\hat{s}v = s \times v$, where v is any vector

Rodrigues' formula:

$$R = I + \sin(\phi)\hat{s} + (1 - \cos(\phi))\hat{s}^2 \qquad (1)$$

The matrix exponential of $\hat{s}\phi$ is given by

$$e^{\hat{s}\phi} = I + \frac{\hat{s}\phi}{1!} + \frac{(\hat{s}\phi)^2}{2!} + \frac{(\hat{s}\phi)^3}{3!} + \cdots$$

According to the property of skew-symmetric matrix that $\hat{s}^T = -\hat{s}$ and $\hat{s}^2 = -I$

The term that the power is odd will have $\pm \hat{s}$,

and the even ones will have $\pm I$

Then $e^{\hat{s}\phi} = I + \left(\phi\hat{s} + \frac{(\hat{s}\phi)^3}{3!} + \cdots\right) + \left(\frac{(\hat{s}\phi)^2}{2!} + \frac{(\hat{s}\phi)^4}{4!} + \cdots\right)$

$$= I + \sin(\phi)\hat{s} + (1 - \cos(\phi))\hat{s}^2 \qquad (2)$$

Equation (1) and (2) are equivalent.

(c) For python function to solve this, see below

```python
import numpy as np
from scipy.linalg import eig

def computeRotationMatrix(s, phi):
    s = s / np.linalg.norm(s)
    s1, s2, s3 = s
    I = np.eye(3)

    s_hat = np.array([
        [0, -s3, s2],
        [s3, 0, -s1],
        [-s2, s1, 0]
    ])

    # Rodriguez
    R = I + np.sin(phi) * s_hat + (1 - np.cos(phi)) * np.dot(s_hat, s_hat)
    return R


s = np.array([1, 1, 1])
phi = np.pi / 4

# Compute orthogonal matrix
R = computeRotationMatrix(s, phi)

# Eigenvalues and eigenvectors
eigenvalues, eigenvectors = eig(R)

# Verify cos(phi) = 1/2 * (trace(R) - 1)
cos_phi = 0.5 * (np.trace(R) - 1)

# Test points before and after rotation
points = np.array([
    [1, 0, 0],  # Along x-axis
    [0, 1, 0],  # Along y-axis
    [0, 0, 1],  # Along z-axis
    [1, 1, 1],  # Along the rotation axis
    [1, -1, 0], # Diagonal in xy-plane
    [0, 1, -1], # Diagonal in yz-plane
    [-1, 0, 1], # Diagonal in xz-plane
    [1, 2, 3],  # Arbitrary point
    [-2, -1, 0] # Arbitrary point
])
rotated_points = np.dot(R, points.T).T
```

For output result script,

```python
# Results
print("Rotation Matrix R:")
print(R,'\n')
print("Eigenvalues:")
print(eigenvalues,'\n')
print("Eigenvectors:")
print(eigenvectors,'\n')
print(f"1/2(trace(R)-1): {cos_phi}, cos(phi): {np.cos(phi)} \n")
print("Points before rotation:")
print(points, '\n')
print("Points after rotation:")
print(rotated_points)
```

For test results,

```
(openvla) PS C:\Users\16690\Desktop> & E:/Anaconda/envs/openvla/python.exe c:/Users/16690/Desktop/280hw1.py
Rotation Matrix R:
[[ 0.80473785 -0.31061722  0.50587936]
 [ 0.50587936  0.80473785 -0.31061722]
 [-0.31061722  0.50587936  0.80473785]]

Eigenvalues:
[0.70710678+0.70710678j 0.70710678-0.70710678j 1.        +0.j        ]

Eigenvectors:
[[-0.57735027+0.j  -0.57735027-0.j   0.57735027+0.j ]
 [ 0.28867513+0.5j  0.28867513-0.5j  0.57735027+0.j ]
 [ 0.28867513-0.5j  0.28867513+0.5j  0.57735027+0.j ]]

1/2(trace(R)-1): 0.7071067811865475, cos(phi): 0.7071067811865476

Points before rotation:
[[ 1  0  0]
 [ 0  1  0]
 [ 0  0  1]
 [ 1  1  1]
 [ 1 -1  0]
 [ 0  1 -1]
 [-1  0  1]
 [ 1  2  3]
 [-2 -1  0]]

Points after rotation:
[[ 0.80473785  0.50587936 -0.31061722]
 [-0.31061722  0.80473785  0.50587936]
 [ 0.50587936 -0.31061722  0.80473785]
 [ 1.          1.          1.        ]
 [ 1.11535507 -0.29885849 -0.81649658]
 [-0.81649658  1.11535507 -0.29885849]
 [-0.29885849 -0.81649658  1.11535507]
 [ 1.70114151  1.18350342  3.11535507]
 [-1.29885849 -1.81649658  0.11535507]]
```

verify

$\cos(\phi) = \frac{1}{2}\{\mathrm{trace}(R) - 1\}$

The relationship between eigenvalues, eigenvectors and the axis vector:

① Eigenvector corresponding to $\lambda = 1$ is the axis of rotation because points along this vector are not rotated

② $\lambda = e^{i\phi}$ and $\lambda = e^{-i\phi}$ (i.e. $0.7071 + 0.7071j$, $0.7071 - 0.7071j$ in this case)

represent how other points in the plane perpendicular to the rotation axis are transformed.

(d)

R is a rotation matrix if $R^T R = I$ & $\det(R) = 1$

The eigenvector corresponding to $\lambda = 1$ is the axis of rotation

The matrix $R - R^T$ is skew-symmetric

$$R - R^T = 2\sin(\phi)\,\hat{s}$$

if we need to derive s and ϕ,

we should ① compute $R - R^T$ and $\hat{s}$

② normalize $\hat{s}$ to find rotation axis

③ compute ϕ

The code is below:

```python
import numpy as np
from questionC import computeRotationMatrix

def computeAxisAndAngle(R):
    skew_symmetric = R - R.T

    sin_phi = np.linalg.norm(skew_symmetric) / 2
    cos_phi = (np.trace(R) - 1) / 2

    phi = np.arctan2(sin_phi, cos_phi)

    # Extract the axis of rotation
    s = np.array([
        skew_symmetric[2, 1],
        skew_symmetric[0, 2],
        skew_symmetric[1, 0]
    ])

    # Normalize
    s = s / (2 * sin_phi)
    return s, phi


phi = np.pi / 4
s = np.array([1, 1, 1]) / np.sqrt(3)
R = computeRotationMatrix(s, phi)
recovered_s, recovered_phi = computeAxisAndAngle(R)

print("Original Axis of Rotation (s):", s)
print("Recovered Axis of Rotation (s):", recovered_s)
print("Original Rotation Angle (phi):", phi)
print("Recovered Rotation Angle (phi):", recovered_phi)
```

```
(openvla) PS E:\Opensourced-Notes-SWE-MLE-JobCreator\Opensourced-Notes-SWE-MLE-JobCreator\Computer Vision> & E:/Anaconda/envs/openvla/python.exe "e:/Opensourced-Notes-SWE-MLE-JobCreator/Opensourced-Notes-SWE-MLE-JobCreator/Computer Vision/questionD.py"
Rotation Matrix R:
[[ 0.80473785 -0.31061722  0.50587936]
 [ 0.50587936  0.80473785 -0.31061722]
 [-0.31061722  0.50587936  0.80473785]]

Eigenvalues:
[0.70710678+0.70710678j 0.70710678-0.70710678j 1.        +0.j        ]

Eigenvectors:
[[-0.57735027+0.j  -0.57735027-0.j   0.57735027+0.j ]
 [ 0.28867513+0.5j  0.28867513-0.5j  0.57735027+0.j ]
 [ 0.28867513-0.5j  0.28867513+0.5j  0.57735027+0.j ]]

1/2(trace(R)-1): 0.7071067811865475, cos(phi): 0.7071067811865476

Points before rotation:
[[ 1  0  0]
 [ 0  1  0]
 [ 0  0  1]
 [ 1  1  1]
 [ 1 -1  0]
 [ 0  1 -1]
 [-1  0  1]
 [ 1  2  3]
 [-2 -1  0]]

Points after rotation:
[[ 0.80473785  0.50587936 -0.31061722]
 [-0.31061722  0.80473785  0.50587936]
 [ 0.50587936 -0.31061722  0.80473785]
 [ 1.          1.          1.        ]
 [ 1.11535507 -0.29885849 -0.81649658]
 [-0.81649658  1.11535507 -0.29885849]
 [-0.29885849 -0.81649658  1.11535507]
 [ 1.70114151  1.18350342  3.11535507]
 [-1.29885849 -1.81649658  0.11535507]]
Original Axis of Rotation (s): [0.57735027 0.57735027 0.57735027]
Recovered Axis of Rotation (s): [0.40824829 0.40824829 0.40824829]
Original Rotation Angle (phi): 0.7853981633974483
Recovered Rotation Angle (phi): 0.9553166181245093
```

(e) The transformation E can be written as

$$Eu_j = Ru_j + t$$

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \quad t = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

We want to minimize the error

$$\min(\text{error}) = \sum_{j=1}^{4} \| Ru_j + t - v_j \|^2$$

$$\bar{u} = \frac{1}{n}\sum_{j=1}^{n} u_j \qquad \bar{v} = \frac{1}{n}\sum_{j=1}^{n} v_j$$

$u_j' = u_j - \bar{u}$, $v_j' = v_j - \bar{v}$ are centered points.

Use SVD to solve R

$\downarrow$

then, $t = \bar{v} - R\bar{u}$

Code is in next page,

```python
import numpy as np

def findBestTransformation(u, v):
    u_mean, v_mean = np.mean(u, axis=0), np.mean(v, axis=0)
    u_centered, v_centered = u - u_mean, v - v_mean

    covariance_matrix = np.dot(u_centered.T, v_centered)

    U, S, Vt = np.linalg.svd(covariance_matrix)

    R = np.dot(Vt.T, U.T)
    t = v_mean - np.dot(R, u_mean)
    return R, t

# Test
u = np.array([[-3, 0], [1, 1], [1, 0], [1, -1]])
v = np.array([[0, 3], [1, 0], [0, 0], [-1, 0]])

R, t = findBestTransformation(u, v)


print("Optimal Rotation Matrix (R):")
print(R, '\n')
print("Optimal Translation Vector (t):")
print(t, '\n')

# Verify the transformation
u_transformed = np.dot(u, R.T) + t
print("Transformed Points:")
print(u_transformed, '\n')
print("Target Points:")
print(v)
```

```
(openvla) PS E:\Opensourced-Notes-SWE-MLE-JobCreator\Opensourced-Notes-SWE-MLE-JobCreator\Computer Vision> & E:/Anaconda/envs/openvla/python.exe "e:/Opensourced-Notes-SWE-MLE-JobCreator/Opensourced-Notes-SWE-MLE-JobCreator/Computer Vision/questionE.py"
Optimal Rotation Matrix (R):
[[ 0.  1.]
 [-1.  0.]]

Optimal Translation Vector (t):
[0.   0.75]

Transformed Points:
[[ 0.    3.75]
 [ 1.   -0.25]
 [ 0.   -0.25]
 [-1.   -0.25]]

Target Points:
[[ 0  3]
 [ 1  0]
 [ 0  0]
 [-1  0]]
```

(f)

A line on a plane can be written as

$$x(t) = p + td$$

p is the point on the line

d is the direction vector of the line

Assume the camera is at origin and the projection is central. A 3D point $X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$ projects to $X_{image} = \begin{bmatrix} \frac{x}{z} \\ \frac{y}{z} \end{bmatrix}$

For large t,

$$X_{vanishing} = \lim_{t \to \infty} \frac{p + td}{z} = \frac{d}{d_z}$$

Consider the plane $\quad n^T x + c = 0$

$n = \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix}$ is the normal vector of the plane

c is a constant.

all lines on the plane have direction d satisfying : $n^T d = 0$

All vanishing points lie on this line in the image plane where the plane intersects the image plane.

set $z=1$ for projection

then $$n^T [x, y, 1]^T + c = 0$$

$$\Rightarrow n_x x + n_y y + n_z + c = 0$$

The vanishing points of all lines on a plane lie on the vanishing line, which is the porjection of the plane's 3D geometry onto the image plane. The vanishing line is determined by the normal vector n and the plane constant c and its equation in 2D is

$$n_x x + n_y y + n_z + c = 0$$