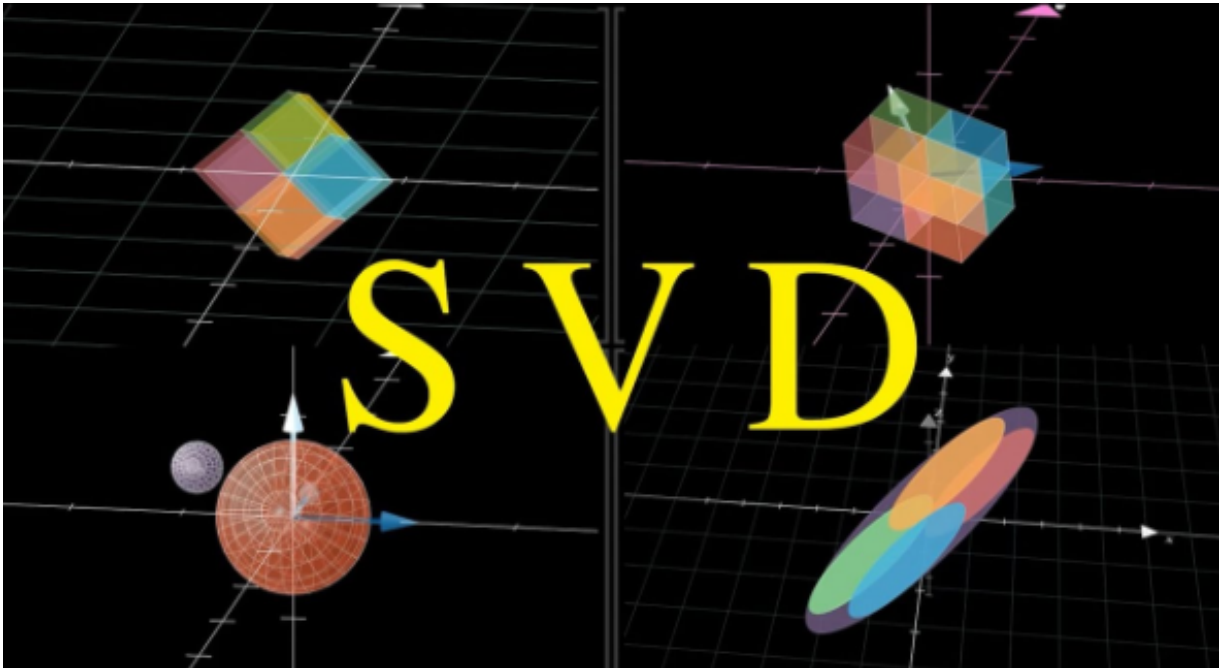


What is singular value decomposition?

Singular Value Decomposition (SVD) is a mathematical technique used to break down a matrix into three separate components: a left singular vector matrix, a singular value diagonal matrix, and a right singular vector matrix. It is a powerful tool in linear algebra and numerical analysis for analyzing and manipulating matrices, and it has applications in various fields such as data analysis, image processing, and signal processing. SVD allows for the representation of a matrix in a form that simplifies calculations and reveals important characteristics of the original matrix.



Singular Value Decomposition (SVD) can be applied to **non-square matrices** as well. In the case of a non-square matrix A with dimensions $m \times n$ (m rows and n columns), the SVD decomposition is expressed as:

$$A = U\Sigma V^T$$

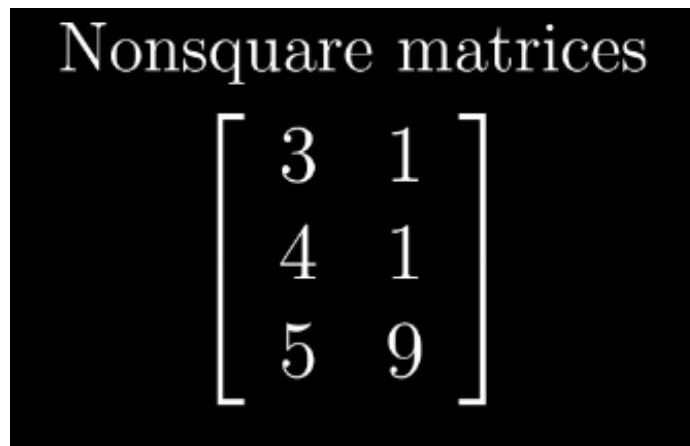
Here's a breakdown of the components:

- **U:** The left singular vector matrix is an $m \times m$ orthogonal matrix. Its columns are the left singular vectors of A . The number of columns in U is equal to the number of rows in A .
- **Σ :** The singular value diagonal matrix is an $m \times n$ matrix. It has non-negative elements along the diagonal and zeros elsewhere. The diagonal elements of Σ are the singular values of A . The number of singular values is equal to the minimum of m and n .
- **V:** The right singular vector matrix is an $n \times n$ orthogonal matrix. Its columns are the right singular vectors of A . The number of columns in V is equal to the number of columns in A .

SVD for non-square matrices is commonly used in applications such as data compression, image processing, and dimensionality reduction. It allows for the extraction of valuable information from the matrix and provides a useful representation that simplifies calculations and analysis.

What is Non square matrix ?

A non-square matrix is a matrix that does not have the same number of rows and columns. In other words, the matrix has a different dimension for the number of rows than it does for the number of columns. For example, a 3x2 matrix has 3 rows and 2 columns, while a 2x3 matrix has 2 rows and 3 columns.


$$\begin{bmatrix} 3 & 1 \\ 4 & 1 \\ 5 & 9 \end{bmatrix}$$

Non-square matrices are often used to represent data that is not naturally rectangular. For example, a non-square matrix can be used to represent a set of images, where each row represents an image and each column represents a pixel in the image.

Non-square matrices can also be used to represent relationships between different sets of data. For example, a non-square matrix can be used to represent a table of data, where each row represents a different data point and each column represents a different variable.

There are many different types of non-square matrices, including:

- **Rectangular matrices:** These matrices have different dimensions for the number of rows and columns.
- **Wide matrices:** These matrices have more columns than rows.
- **Tall matrices:** These matrices have more rows than columns.
- **Sparse matrices:** These matrices have a lot of zeros.
- **Banded matrices:** These matrices have non-zero elements only in a narrow band around the main diagonal.

Non-square matrices are a powerful tool that can be used to represent and analyze data in a variety of ways.

In [5]:

```

# The code then defines a 2x3 transformation matrix, and applies the transformation to
# The code then creates two figures, one for the original cube and one for the transformed cube

import numpy as np
import plotly.graph_objects as go

# Define the 8 vertices of a cube in 3D space
cube = np.array([
    [0, 0, 0],
    [1, 0, 0],
    [1, 1, 0],
    [0, 1, 0],
    [0, 0, 1],
    [1, 0, 1],
    [1, 1, 1],
    [0, 1, 1]
])

# Create a list of edges to build the cube
edges = [(0, 1), (1, 2), (2, 3), (3, 0), # edges on the bottom face
         (4, 5), (5, 6), (6, 7), (7, 4), # edges on the top face
         (0, 4), (1, 5), (2, 6), (3, 7)] # edges on the sides

# Define a 2x3 transformation matrix
A = np.array([[1, 2, 1], [2, 1, 0]])

# Apply the transformation
transformed_cube = np.dot(A, cube.T).T

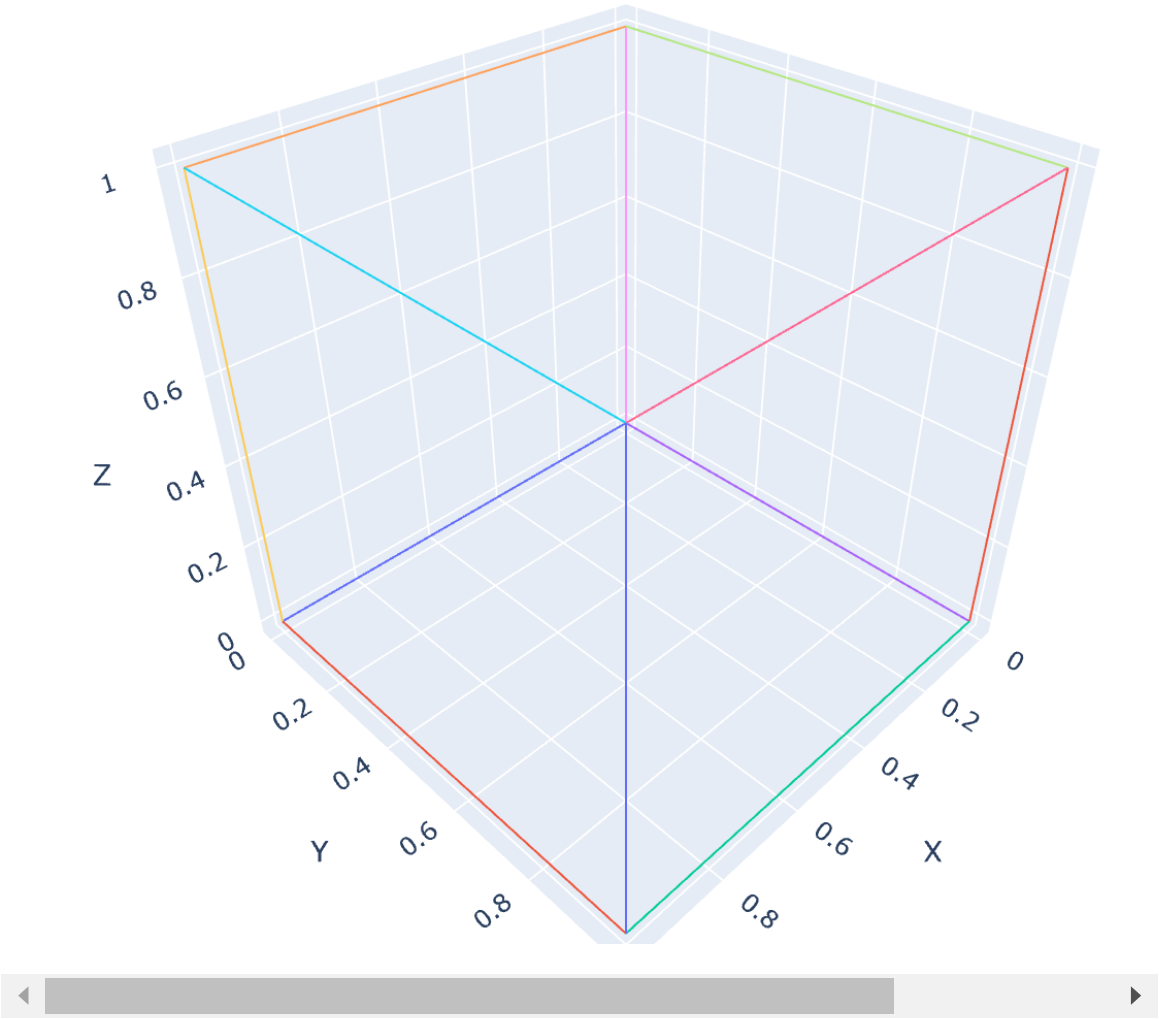
# Create a 3D plot for the original cube
fig1 = go.Figure()

for edge in edges:
    x0, y0, z0 = cube[edge[0]]
    x1, y1, z1 = cube[edge[1]]
    fig1.add_trace(go.Scatter3d(x=[x0, x1], y=[y0, y1], z=[z0, z1], mode='lines'))

fig1.update_layout(scene=dict(xaxis_title='X', yaxis_title='Y', zaxis_title='Z'),
                    width=700, margin=dict(r=20, l=10, b=10, t=10))

fig1.show()

```

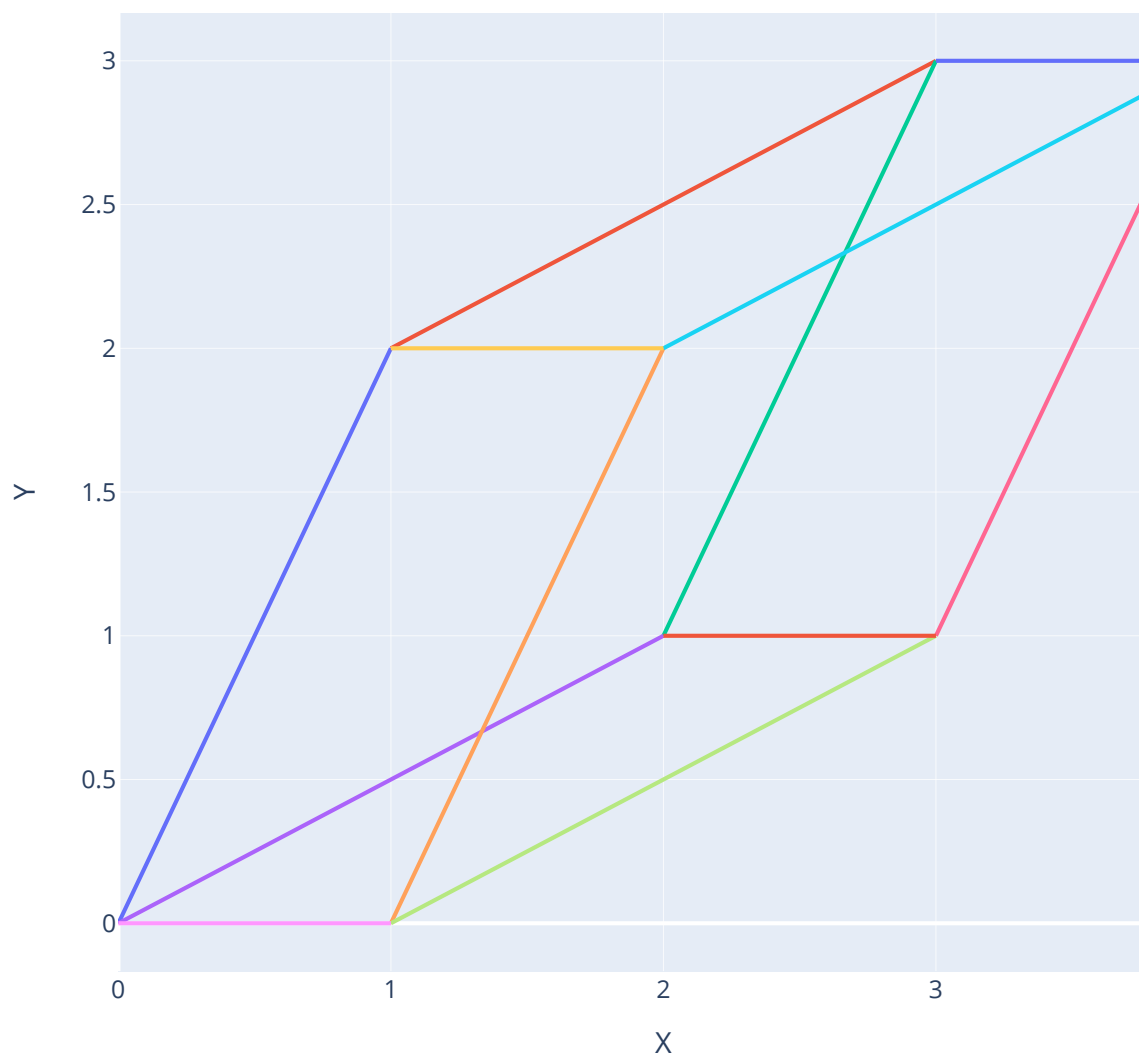


In [6]:

```
# Create a 2D plot for the transformed cube
fig2 = go.Figure()

for edge in edges:
    x0, y0 = transformed_cube[edge[0]]
    x1, y1 = transformed_cube[edge[1]]
    fig2.add_trace(go.Scatter(x=[x0, x1], y=[y0, y1], mode='lines'))

fig2.update_layout(xaxis_title='X', yaxis_title='Y',
                    width=700, margin=dict(r=20, l=10, b=10, t=10))
fig2.show()
```



Explanation

This code creates two figures to visualize a 3D cube and its transformed version in 2D space.

1. First, the code defines a 3D cube by specifying its 8 vertices and creates a list of edges to construct the cube.
2. Next, a 2×3 transformation matrix A is defined. This matrix will be used to transform the original 3D cube into a new configuration.

3. The transformation is applied to the original cube using matrix multiplication, resulting in the transformed_cube.
4. For the 3D visualization, the code uses the Plotly library to create a figure (fig1) representing the original cube. It iterates through the edges of the cube and adds traces to the figure to draw lines connecting the vertices, creating the wireframe representation of the cube.
5. The fig1 is then displayed, showing the 3D visualization of the original cube.
6. Now, a new 2D figure fig2 is created for the transformed cube visualization. Similar to the previous step, it iterates through the edges of the transformed_cube and adds traces to the figure to draw lines connecting the vertices, creating the 2D wireframe representation of the transformed cube.
7. Finally, fig2 is displayed, showing the 2D visualization of the transformed cube.

What is Rectangular Diagonal Matrix ?

A rectangular diagonal matrix is a matrix in which all the entries outside the main diagonal are zero, and the main diagonal entries can be either zero or nonzero. The main diagonal is the diagonal that runs from the top left corner of the matrix to the bottom right corner.

A matrix that would be diagonal if it were square, but instead is rectangular due to extra rows or columns of zeros.

There are two main transformations that can be applied to a rectangular diagonal matrix:

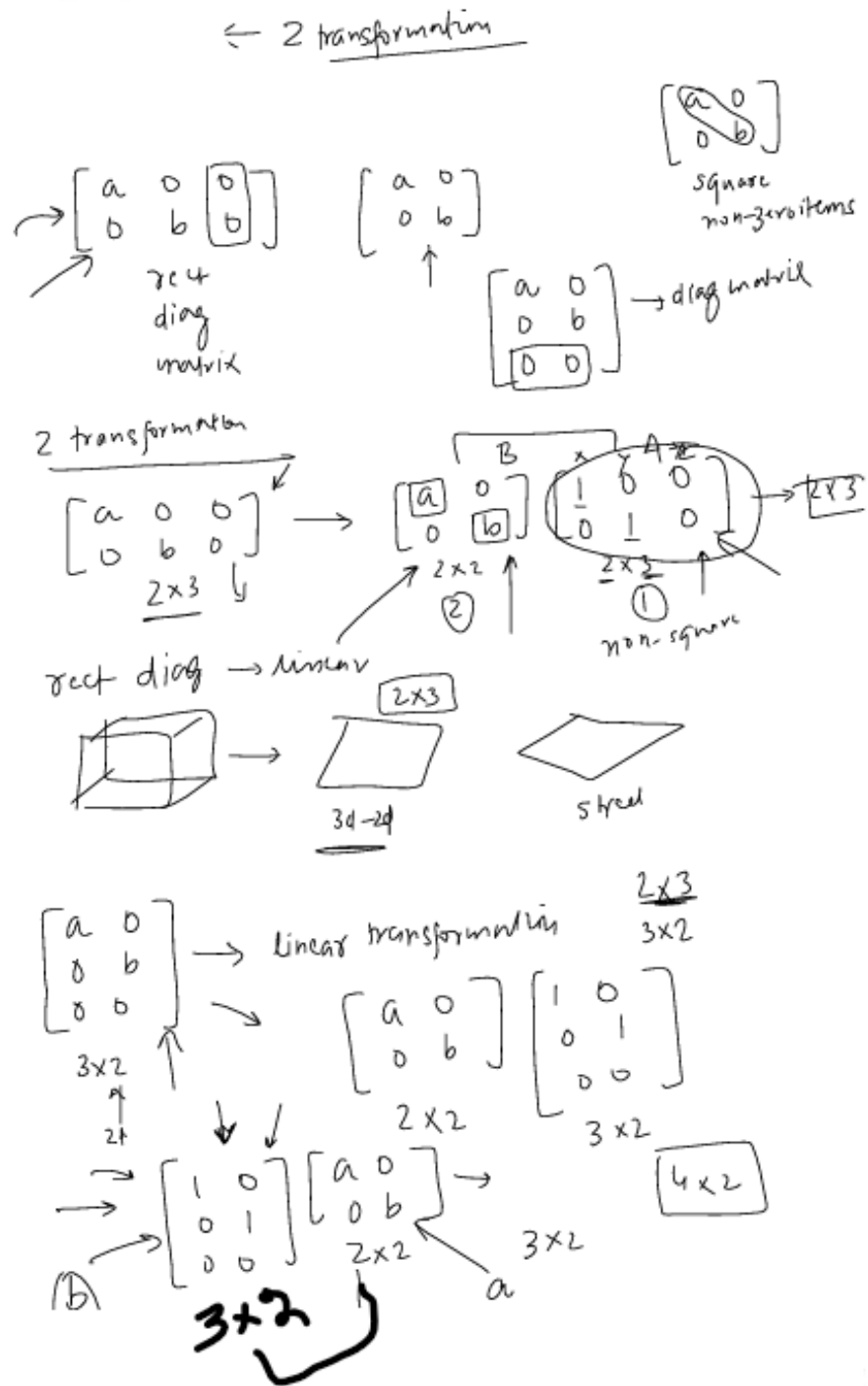
- **Diagonalization:** This transformation converts the matrix into a diagonal matrix, where all the off-diagonal entries are zero and the diagonal entries are the eigenvalues of the original matrix.
- **Scaling:** This transformation multiplies the matrix by a scalar, which has the effect of scaling the diagonal entries of the matrix.

Diagonalization is a powerful tool that can be used to simplify the analysis of a matrix. For example, if a matrix is diagonalizable, then its eigenvalues can be easily found and used to understand the behavior of the matrix.

Scaling is a simpler transformation that can be used to change the scale of a matrix. For example, if a matrix is scaled by a factor of 2, then all the diagonal entries of the matrix will be doubled.

Here are some examples of how these transformations can be used:

- **Diagonalization:** If a matrix represents a system of linear equations, then diagonalization can be used to solve the system of equations.
- **Scaling:** If a matrix represents a covariance matrix, then scaling can be used to adjust the variance of the data.
- **Diagonalization and scaling:** These transformations can be used together to simplify the analysis of a matrix and to make it easier to understand the behavior of the matrix.



Singular Value Decomposition Equation

The Singular Value Decomposition (SVD) equation represents the decomposition of a matrix A into three separate matrices. Given a matrix A with dimensions $m \times n$ (m rows and n columns), the SVD equation is:

$$A = U \Sigma V^T$$

where Q is an orthogonal matrix consisting of eigenvectors of A , and Λ is a diagonal matrix containing the eigenvalues of A .

SVD, on the other hand, can be applied to any matrix A , square or non-square. The SVD equation is:

$$A = U \Sigma V^T$$

where U and V are orthogonal matrices, and Σ is a diagonal matrix containing the singular values of A .

The relationship between SVD and eigen decomposition lies in the fact that:

- If A is a square matrix, the singular values of A in the SVD are the square roots of the eigenvalues of $A^T A$ or $A A^T$ (which are positive). The singular vectors in U and V are related to the eigenvectors of $A^T A$ and $A A^T$.
- If A is a non-square matrix, the singular values and singular vectors of A in the SVD are related to the eigenvalues and eigenvectors of $A^T A$ or $A A^T$.

In summary, while eigen decomposition is specific to square matrices, SVD is a more general matrix factorization technique that can handle both square and non-square matrices. The singular values and vectors in SVD provide information about the eigenvalues and eigenvectors of certain related square matrices.

Handwritten notes showing the derivation of SVD from eigen decomposition of $A^T A$ and $A A^T$.

Left side (Eigen decomposition of $A^T A$):

$$A^T A = U \Sigma U^T \quad \text{where } \Sigma = \begin{bmatrix} a^2 & 0 \\ 0 & b^2 \end{bmatrix}$$

$$A A^T = V \Sigma V^T \quad \text{where } \Sigma = \begin{bmatrix} a^2 & 0 \\ 0 & b^2 \end{bmatrix}$$

Right side (Eigen decomposition of $A A^T$):

$$A A^T = V \Sigma V^T \quad \text{where } \Sigma = \begin{bmatrix} a^2 & 0 \\ 0 & b^2 \end{bmatrix}$$

Final SVD formula:

$$A = U \Sigma V^T$$

Geometric Intuition

The geometric intuition behind the Singular Value Decomposition (SVD) lies in understanding how it transforms a matrix and captures its underlying structure.

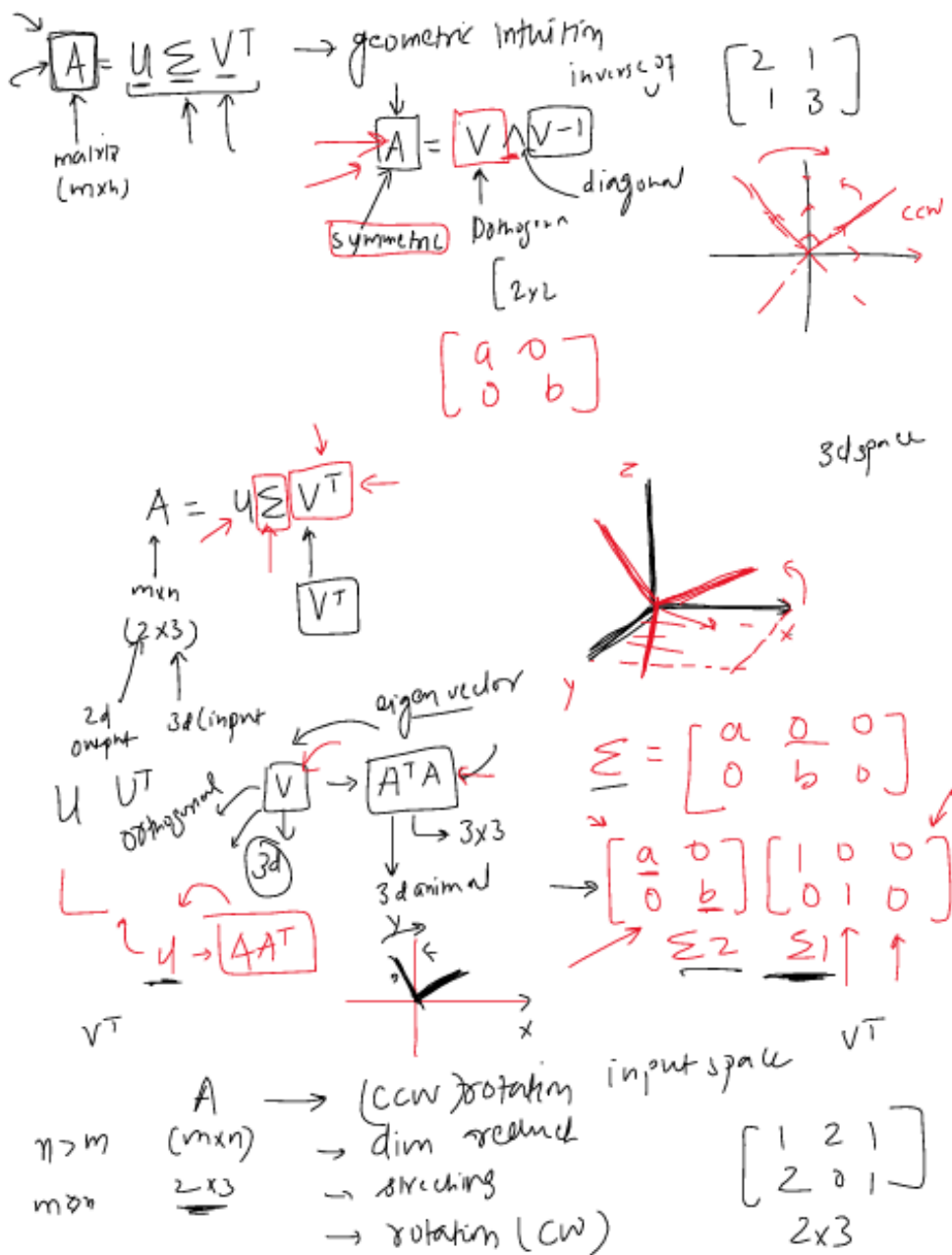
Imagine a matrix A as a linear transformation that acts on a vector space. SVD provides a way to decompose this transformation into three fundamental operations: rotation/reflection, scaling, and another rotation/reflection.

1. **Rotation/Reflection (U and V):** The matrices U and V in the SVD equation, $A = U \Sigma V^T$, represent rotations or reflections in the input and output vector spaces, respectively. They capture the orthogonal transformation aspect of the matrix A . The columns of U and V are the left and right singular vectors, respectively, which form an orthonormal basis for their respective spaces.

2. **Scaling (Σ):** The diagonal matrix Σ in the SVD equation contains the singular values of A . These singular values represent the scaling factors applied to the transformed vectors. The singular values provide information about the stretching or compression along the orthogonal directions defined by the singular vectors.

Together, the SVD equation decomposes the original matrix A into these three components: U , Σ , and V^T . The left singular vectors in U and the right singular vectors in V^T define orthogonal coordinate systems that span the input and output spaces, respectively. The singular values in Σ scale the vectors along these coordinate systems.

From a geometric perspective, SVD allows us to understand how the original matrix A distorts or transforms vectors, and how this transformation can be decomposed into fundamental geometric operations. It provides a way to analyze the shape, orientation, and scaling behavior of the linear transformation represented by A .



3 * 2 matrix

In [7]:

```
# Create a 3x2 matrix

import numpy as np

a = np.array([[1,2,1],[2,0,1]])

# Perform Singular Value Decomposition (SVD) on the given matrix.

U,S,V_t = np.linalg.svd(a)
```

In [8]:

U

Out[8]:

```
array([[ 0.76301998, -0.6463749 ],
       [ 0.6463749 ,  0.76301998]])
```

In [9]:

S

Out[9]:

```
array([2.92256416, 1.56799832])
```

In [10]:

```
# Create a 3x2 matrix filled with zeros
S_full = np.zeros((U.shape[1], V_t.shape[0]))

# Fill the diagonal of S_full with the values from S
S_full[:S.shape[0], :S.shape[0]] = np.diag(S)
```

In [11]:

S_full[:,[0,1]]

Out[11]:

```
array([[2.92256416, 0.          ],
       [0.          , 1.56799832]])
```

In [12]:

```
V_t
```

Out[12]:

```
array([[ 0.70341305,  0.5221579 ,  0.482246  ],
       [ 0.56101149, -0.82445866,  0.07439108],
       [-0.43643578, -0.21821789,  0.87287156]])
```

In [13]:

```

import numpy as np
import plotly.graph_objects as go

# define eight vertices for the unit cube
vertices = np.array([
    [-2.5, -2.5, -2.5],
    [2.5, -2.5, -2.5],
    [2.5, 2.5, -2.5],
    [-2.5, 2.5, -2.5],
    [-2.5, -2.5, 2.5],
    [2.5, -2.5, 2.5],
    [2.5, 2.5, 2.5],
    [-2.5, 2.5, 2.5]
])

# define the twelve edges of the cube
edges = [
    (0, 1), (1, 2), (2, 3), (3, 0), # edges in the bottom face
    (4, 5), (5, 6), (6, 7), (7, 4), # edges in the top face
    (0, 4), (1, 5), (2, 6), (3, 7) # edges connecting top and bottom faces
]

# create a plotly graph object
fig = go.Figure()

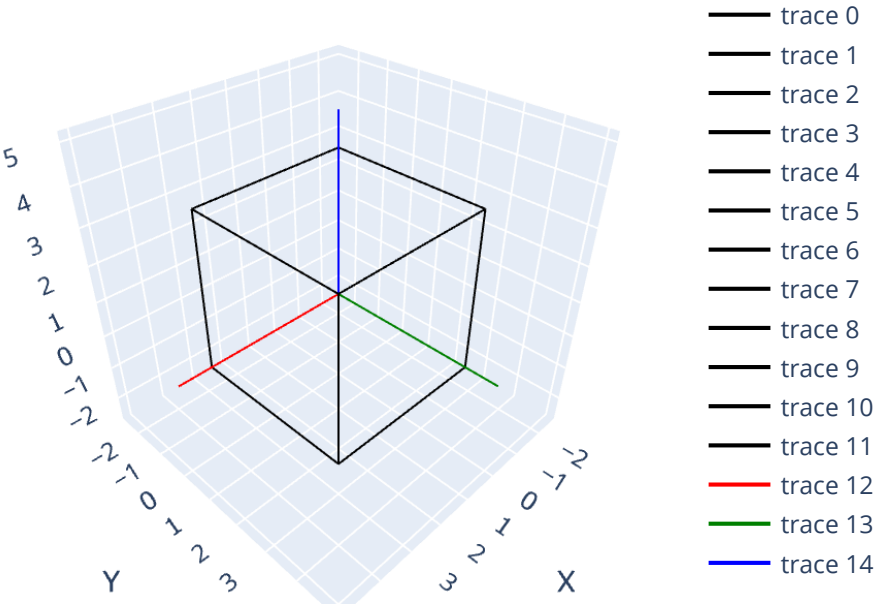
# add each edge to the graph
for edge in edges:
    x_values = [vertices[edge[0], 0], vertices[edge[1], 0]]
    y_values = [vertices[edge[0], 1], vertices[edge[1], 1]]
    z_values = [vertices[edge[0], 2], vertices[edge[1], 2]]
    fig.add_trace(go.Scatter3d(x=x_values, y=y_values, z=z_values, mode='lines', line:

# Draw axis lines
for i, color in enumerate(['red', 'green', 'blue']): # i=0 is X, i=1 is Y, i=2 is Z
    axis = np.zeros((2, 3))
    axis[1, i] = 1
    axis_line = axis * 5 # scale to the desired length
    fig.add_trace(go.Scatter3d(x=axis_line[:, 0], y=axis_line[:, 1], z=axis_line[:, 2],

fig.update_layout(scene=dict(xaxis_title='X', yaxis_title='Y', zaxis_title='Z'), autos:
                    margin=dict(l=50, r=50, b=100, t=100, pad=4))

fig.show()

```



In [14]:

```

# Transformation matrix
T = np.array([
    [-0.70341305, -0.5221579, -0.482246 ],
    [ 0.56101149, -0.82445866,  0.07439108],
    [-0.43643578, -0.21821789,  0.87287156]
])

# create a plotly graph object
fig = go.Figure()

# Add each edge of the original cube to the graph
for edge in edges:
    x_values = [vertices[edge[0], 0], vertices[edge[1], 0]]
    y_values = [vertices[edge[0], 1], vertices[edge[1], 1]]
    z_values = [vertices[edge[0], 2], vertices[edge[1], 2]]
    fig.add_trace(go.Scatter3d(x=x_values, y=y_values, z=z_values, mode='lines', line=

# Apply transformation
vertices_transformed = np.dot(vertices, T.T)

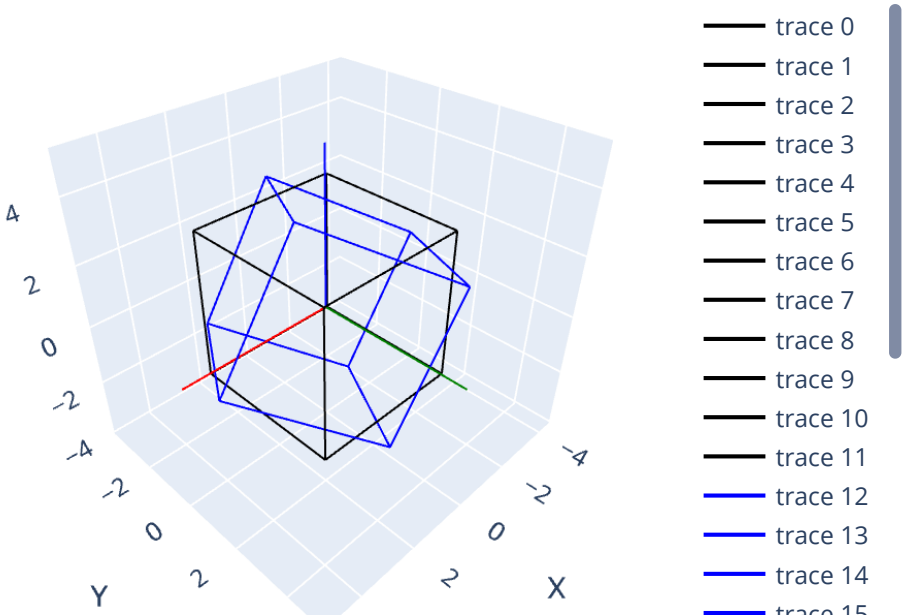
# Add each edge of the transformed cube to the graph
for edge in edges:
    x_values = [vertices_transformed[edge[0], 0], vertices_transformed[edge[1], 0]]
    y_values = [vertices_transformed[edge[0], 1], vertices_transformed[edge[1], 1]]
    z_values = [vertices_transformed[edge[0], 2], vertices_transformed[edge[1], 2]]
    fig.add_trace(go.Scatter3d(x=x_values, y=y_values, z=z_values, mode='lines', line=

# Draw axis lines
for i, color in enumerate(['red', 'green', 'blue']): # i=0 is X, i=1 is Y, i=2 is Z
    axis = np.zeros((2, 3))
    axis[1, i] = 1
    axis_line = axis * 5 # scale to the desired length
    fig.add_trace(go.Scatter3d(x=axis_line[:, 0], y=axis_line[:, 1], z=axis_line[:, 2],

fig.update_layout(scene=dict(xaxis_title='X', yaxis_title='Y', zaxis_title='Z'), autos
                    margin=dict(l=50, r=50, b=100, t=100, pad=4))

fig.show()

```

In [15]:

```
# Transformation matrix for projection onto X-Y plane
P = np.array([
    [1, 0, 0],
    [0, 1, 0]
])

# Apply transformation
vertices_transformed_2d = np.dot(vertices_transformed, P.T)

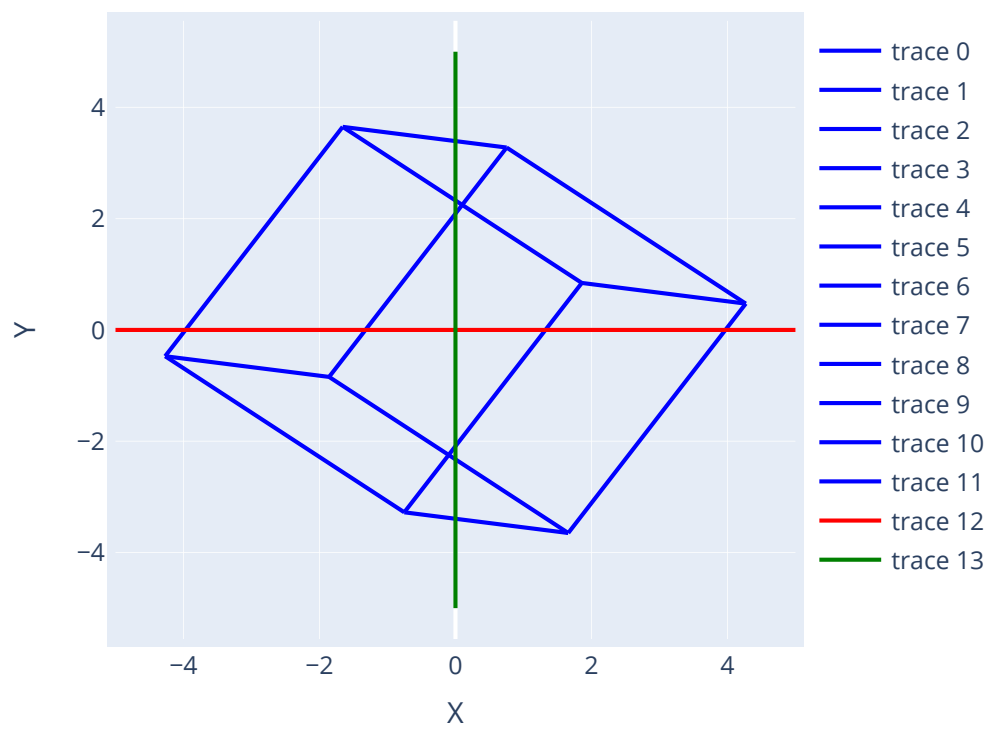
# create a plotly graph object for 2D
fig2d = go.Figure()

# Add each edge of the transformed cube to the graph
for edge in edges:
    x_values = [vertices_transformed_2d[edge[0], 0], vertices_transformed_2d[edge[1],
    y_values = [vertices_transformed_2d[edge[0], 1], vertices_transformed_2d[edge[1],
    fig2d.add_trace(go.Scatter(x=x_values, y=y_values, mode='lines', line=dict(color=

# Draw axis lines
axis = np.array([[ -5, 5], [0, 0]])
fig2d.add_trace(go.Scatter(x=axis[0, :], y=axis[1, :], mode='lines', line=dict(color=
axis = np.array([[0, 0], [ -5, 5]])
fig2d.add_trace(go.Scatter(x=axis[0, :], y=axis[1, :], mode='lines', line=dict(color=

fig2d.update_layout(xaxis_title='X', yaxis_title='Y', autosize=False, width=500, height=500)

fig2d.show()
```



In [16]:

```
# Transformation matrix for scaling
S = np.array([
    [2.92256416, 0],
    [0, 1.56799832]
])

# Apply transformation
vertices_transformed_2d_scaled = np.dot(vertices_transformed_2d, S.T)

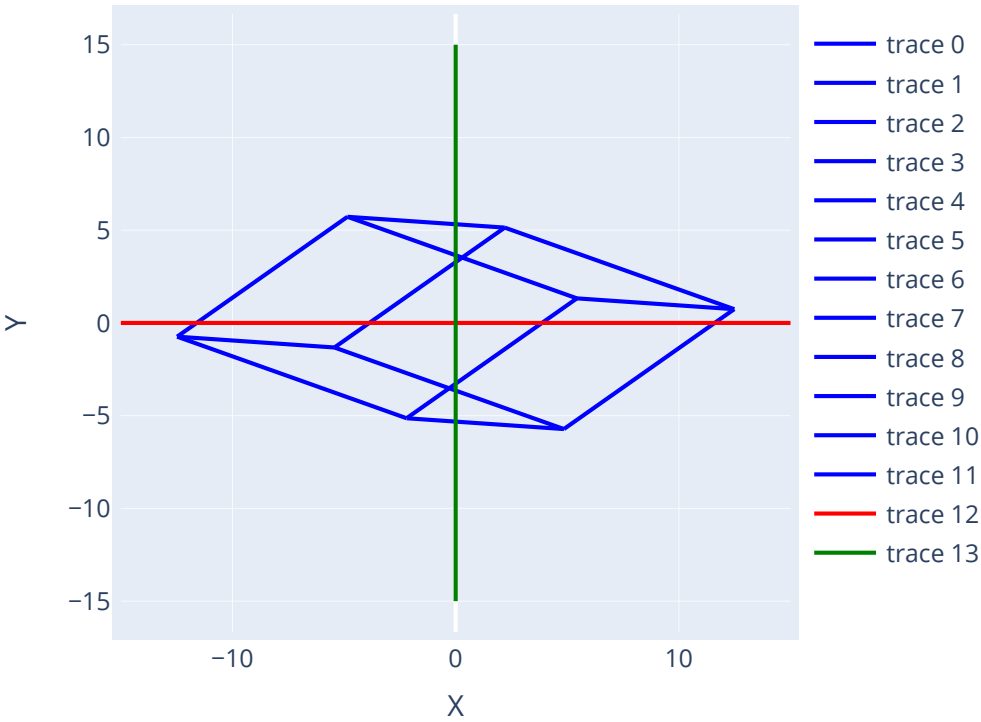
# create a new plotly graph object for 2D
fig2d_scaled = go.Figure()

# Add each edge of the transformed cube to the graph
for edge in edges:
    x_values = [vertices_transformed_2d_scaled[edge[0], 0], vertices_transformed_2d_scaled[edge[1], 0], vertices_transformed_2d_scaled[edge[2], 0]]
    y_values = [vertices_transformed_2d_scaled[edge[0], 1], vertices_transformed_2d_scaled[edge[1], 1], vertices_transformed_2d_scaled[edge[2], 1]]
    fig2d_scaled.add_trace(go.Scatter(x=x_values, y=y_values, mode='lines', line=dict(color='red', width=2)))

# Draw axis lines
axis = np.array([[-15, 15], [0, 0]])
fig2d_scaled.add_trace(go.Scatter(x=axis[0, :], y=axis[1, :], mode='lines', line=dict(color='blue', width=2)))
axis = np.array([[0, 0], [-15, 15]])
fig2d_scaled.add_trace(go.Scatter(x=axis[0, :], y=axis[1, :], mode='lines', line=dict(color='blue', width=2)))

fig2d_scaled.update_layout(xaxis_title='X', yaxis_title='Y', autosize=False, width=500, height=500)

fig2d_scaled.show()
```



In [17]:

```
# Transformation matrix for rotation
R = np.array([
    [-0.76301998, -0.6463749],
    [-0.6463749, 0.76301998]
])

# Apply transformation
vertices_transformed_2d_scaled_rotated = np.dot(vertices_transformed_2d_scaled, R.T)

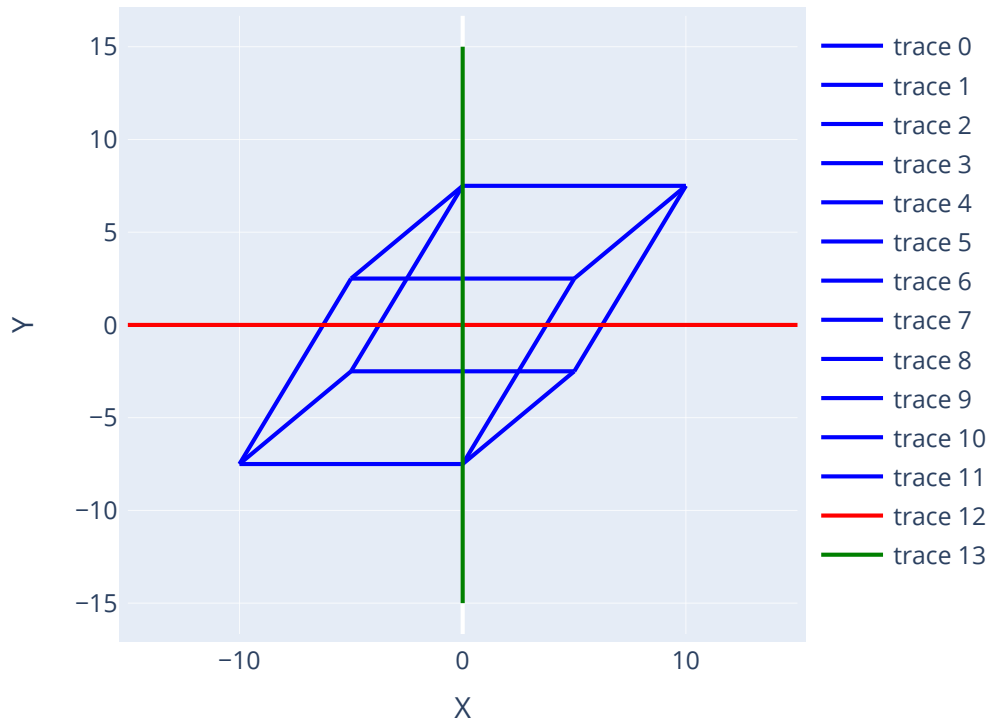
# create a new plotly graph object for 2D
fig2d_scaled_rotated = go.Figure()

# Add each edge of the transformed cube to the graph
for edge in edges:
    x_values = [vertices_transformed_2d_scaled_rotated[edge[0], 0], vertices_transformed_2d_scaled_rotated[edge[1], 0]]
    y_values = [vertices_transformed_2d_scaled_rotated[edge[0], 1], vertices_transformed_2d_scaled_rotated[edge[1], 1]]
    fig2d_scaled_rotated.add_trace(go.Scatter(x=x_values, y=y_values, mode='lines', line=dict(color='red')))

# Draw axis lines
axis = np.array([[-15, 15], [0, 0]])
fig2d_scaled_rotated.add_trace(go.Scatter(x=axis[0, :], y=axis[1, :], mode='lines', line=dict(color='red')))
axis = np.array([[0, 0], [-15, 15]])
fig2d_scaled_rotated.add_trace(go.Scatter(x=axis[0, :], y=axis[1, :], mode='lines', line=dict(color='red')))

fig2d_scaled_rotated.update_layout(xaxis_title='X', yaxis_title='Y', autosize=False, width=500, height=500)

fig2d_scaled_rotated.show()
```



Explanation

In the FOCAL CELL, the code applies a series of transformations to a 3D cube to visualize it in 2D. Let's go through it step by step:

1. Rotation Transformation:

The first transformation is a rotation. The code defines a 2x2 rotation matrix R to rotate the 2D coordinates of the transformed and scaled cube. The matrix R is applied to the 2D coordinates in `vertices_transformed_2d_scaled` using matrix multiplication, and the result is stored in `vertices_transformed_2d_scaled_rotated`.

2. Plotting the Rotated 2D Cube:

The code creates a new plotly graph object `fig2d_scaled_rotated` to visualize the rotated 2D cube.

- It iterates through each edge in the edges list to plot the edges of the cube in blue on the graph
- The x and y values of each edge are extracted from `vertices_transformed_2d_scaled_rotated`.
- These edge points are then added to the graph using `go`. Scatter with the mode set to 'lines' and the color set to 'blue'.

3. Adding Axis Lines:

- The code adds red and green axis lines to the graph to represent the X and Y axes, respectively.

4. Plot Layout:

- The code sets the layout of the graph with axis titles, fixed width and height, and margin settings.

5. Displaying the Graph:

- Finally, the graph `fig2d_scaled_rotated` is displayed using `fig2d_scaled_rotated.show()`. executed.

3*2 Matrix

In [18]:

```
import numpy as np

A = np.array([
    [3, 1],
    [1, 0],
    [1, 1]
])

# Compute SVD
U, S, V_T = np.linalg.svd(A)

# Full  $\Sigma$  matrix
S_full = np.zeros(A.shape)
for i in range(len(S)):
    S_full[i, i] = S[i]

print("U =")
print(U)
print("\nFull  $\Sigma$  matrix =")
print(S_full)
print("\nV^T =")
print(V_T)
```

```
U =
[[-0.89291197 -0.18984612 -0.40824829]
 [-0.26415956 -0.51337419  0.81649658]
 [-0.36459285  0.83690226  0.40824829]]
```

```
Full  $\Sigma$  matrix =
[[3.53847386 0.          ]
 [0.          0.69224469]
 [0.          0.          ]]
```

```
V^T =
[[-0.9347217 -0.35538056]
 [-0.35538056  0.9347217 ]]
```

Explanation

In this code, the user is performing Singular Value Decomposition (SVD) on a 3x2 matrix A. SVD is a mathematical technique commonly used in linear algebra and data analysis.

1. The code imports the NumPy library, which provides support for array operations and linear algebra.

2. The user defines the matrix A as a 3x2 NumPy array.
3. SVD is then applied to the matrix A using the `np.linalg.svd()` function, which returns three matrices: U, S, and V_T.
4. The u matrix contains the left singular vectors, s contains the singular values in descending order, and V_T contains the transposed right singular vectors.
5. The code creates the s_full matrix, which is a diagonal matrix containing the singular values of A.
6. The output of the SVD computation is printed to the console, showing the values of U, S_full, and V_T.
7. The u matrix shows how the original vectors were transformed to form the basis of the input space.
8. The s_full matrix contains the singular values, with the other elements set to zero to make it a full-size square matrix.
9. The V_T matrix represents how the input vectors are transformed to form the basis of the output space.

```
U =  
[[-0.89291197 -0.18984612 -0.40824829]  
 [-0.26415956 -0.51337419  0.81649658]  
 [-0.36459285  0.83690226  0.40824829]]
```

```
Full  $\Sigma$  matrix =  
[[3.53847386 0.          ]  
 [0.          0.69224469]  
 [0.          0.          ]]
```

```
V^T =  
[[-0.9347217  -0.35538056]  
 [-0.35538056  0.9347217  ]]
```

Explanation

In this code, the user is performing Singular Value Decomposition (SVD) on a 3x2 matrix A. SVD is a mathematical technique commonly used in linear algebra and data analysis.

1. The code imports the NumPy library, which provides support for array operations and linear algebra.
2. The user defines the matrix A as a 3x2 NumPy array.
3. SVD is then applied to the matrix A using the `np.linalg.svd()` function, which returns three matrices: U, S, and V_T.
4. The u matrix contains the left singular vectors, s contains the singular values in descending order, and V_T contains the transposed right singular vectors.
5. The code creates the s_full matrix, which is a diagonal matrix containing the singular values of A.
6. The output of the SVD computation is printed to the console, showing the values of U, S_full, and V_T.
7. The u matrix shows how the original vectors were transformed to form the basis of the input space.
8. The s_full matrix contains the singular values, with the other elements set to zero to make it a full-size square matrix.
9. The V_T matrix represents how the input vectors are transformed to form the basis of the output space.

```
In [19]: import plotly.graph_objects as go
import numpy as np

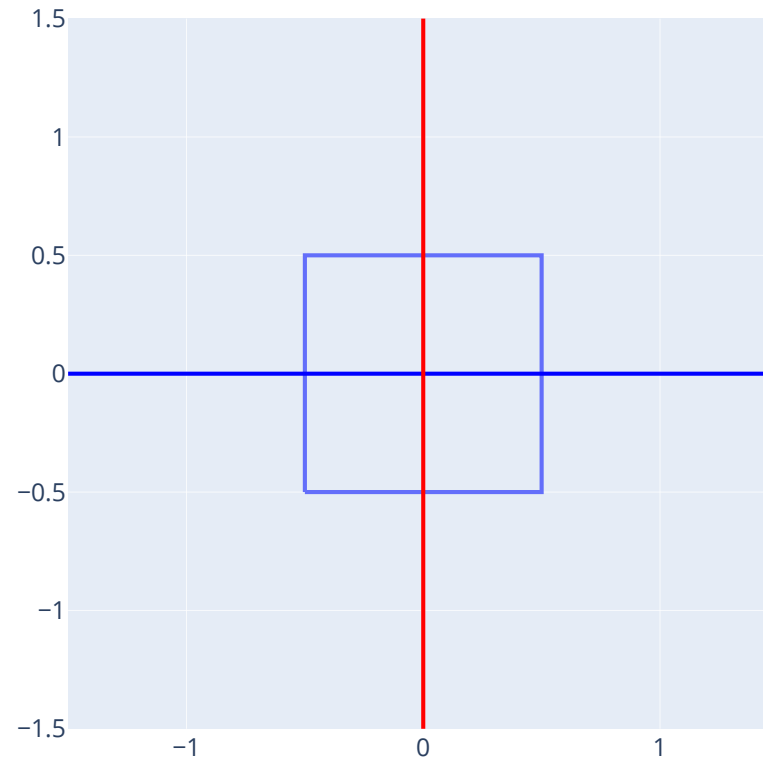
# Define vertices of the unit square centered at the origin
vertices = np.array([
    [-0.5, -0.5],
    [0.5, -0.5],
    [0.5, 0.5],
    [-0.5, 0.5],
    [-0.5, -0.5]
])

# Create a scatter plot for the square
fig = go.Figure(data=go.Scatter(x=vertices[:, 0], y=vertices[:, 1], mode='lines'))

# Add x and y axes
fig.add_shape(
    type="line", line=dict(width=2, color="Blue"),
    x0=-1.5, x1=1.5, y0=0, y1=0,
)
fig.add_shape(
    type="line", line=dict(width=2, color="Red"),
    x0=0, x1=0, y0=-1.5, y1=1.5
)

# Update axes
fig.update_xaxes(range=[-1.5, 1.5], constrain="domain")
fig.update_yaxes(range=[-1.5, 1.5], scaleanchor="x", scaleratio=1)

# Show figure
fig.show()
```



Explanation

This code generates a plot of a unit square centered at the origin, and then applies a scaling and rotation transformation to the square, followed by plotting the transformed square in 2D using Plotly. Let's break down the code in the focal cell step-by-step:

1. Importing Libraries: The code starts by importing the necessary libraries, `plotly.graph_objects` as `go` and `numpy` as `np`.

2. Defining the Vertices of the Unit Square: o The code defines the vertices of the unit square centered at the origin. The square is defined using an array called vertices. Each row of the array represents a vertex, and each vertex is specified by its (x, y) coordinates.
3. Creating the Initial Plot: o The code creates a scatter plot for the unit square using Plotly. It uses the go. Scatter function to plot the vertices of the square as connected lines.
4. Adding X and Y Axes: • Two lines are added to represent the X and Y axes using fig.add_shape(). The X-axis is represented by a blue line, and the Y-axis is represented by a red line.
5. Updating Axis Ranges: • The code updates the X and Y axis ranges to ensure that the plot covers a square area. It sets the X-axis range from -1.5 to 1.5 and the Y-axis range from -1.5 to 1.5. The constrain="domain" and scaleanchor="x", scaleratio=1 properties ensure that the plot maintains a square aspect ratio.
6. Displaying the Plot: o Finally, the fig.show() function is called to display the plot with the unit square and the X, Y axes. Since the output of this focal cell is a plot, there will not be any STDOUT or explicit result printed in the console. The plot will be displayed directly in the output area.

Please note that the code in the focal cell doesn't apply the scaling and rotation transformation as mentioned in the previous code section. If you want to see the transformed square plot, the transformation code from the previous code section should be included before displaying


```
In [20]: import plotly.graph_objects as go
import numpy as np

# Define vertices of the unit square centered at the origin
vertices = np.array([
    [-0.5, -0.5],
    [0.5, -0.5],
    [0.5, 0.5],
    [-0.5, 0.5],
    [-0.5, -0.5]
])

# Define the linear transformation
transformation = np.array([
    [-0.9347217, -0.35538056],
    [-0.35538056, 0.9347217]
])

# Apply the transformation to the square
transformed_vertices = vertices @ transformation

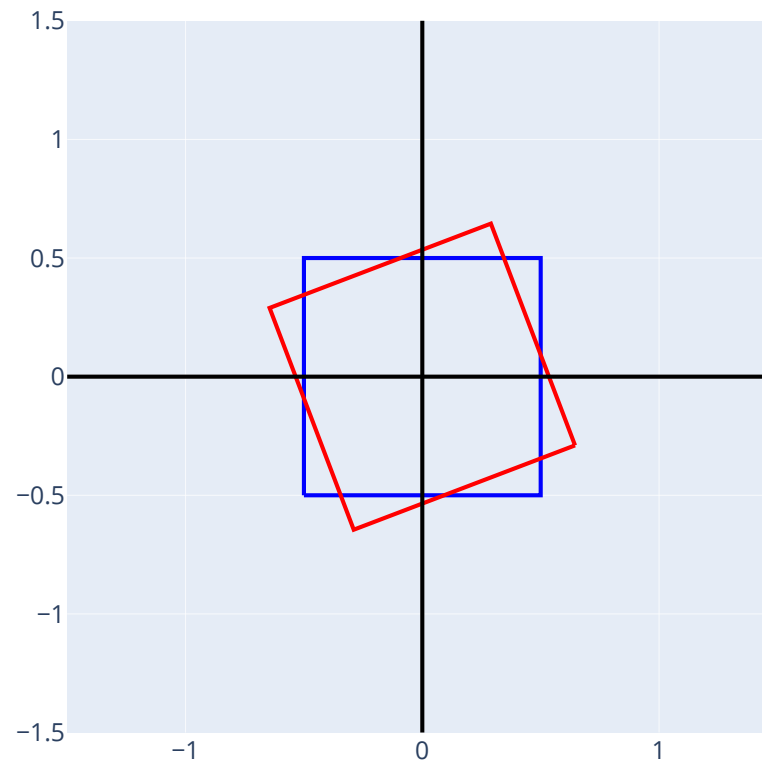
# Create a scatter plot for the original square
fig = go.Figure(data=go.Scatter(x=vertices[:, 0], y=vertices[:, 1], mode='lines', line=dict(color='blue'), name='Original Square'))

# Add the transformed square to the plot
fig.add_trace(go.Scatter(x=transformed_vertices[:, 0], y=transformed_vertices[:, 1], mode='lines', line=dict(color='red'), name='Transformed Square'))

# Add x and y axes
fig.add_shape(
    type="line", line=dict(width=2, color="black"),
    x0=-1.5, x1=1.5, y0=0, y1=0,
)
fig.add_shape(
    type="line", line=dict(width=2, color="black"),
    x0=0, x1=0, y0=-1.5, y1=1.5
)

# Update axes
fig.update_xaxes(range=[-1.5, 1.5], constrain="domain")
fig.update_yaxes(range=[-1.5, 1.5], scaleanchor="x", scaleratio=1)
```

```
# Show figure  
fig.show()
```



Explanation

The code in the focal cell uses Python with the Plotly library to visualize the transformation of a unit square in 2D space. Let's go through it step by step:

1. Importing Libraries: The code begins by importing the necessary libraries: `plotly.graph_objects` as `go` for visualization, and `numpy` as `np` for mathematical operations.
2. Defining Vertices: The code defines the vertices of the unit square centered at the origin. The vertices are represented as a NumPy array containing five points in 2D space.
3. Defining Transformation Matrix: A transformation matrix transformation is defined as a 2x2 NumPy array. This matrix represents a linear transformation that will be applied to the original square.
4. Applying the Transformation: The transformation is applied to the original square by performing matrix multiplication between the vertices and the transformation matrix. The result is stored in `transformed_vertices`.
5. Creating the Plot: The code creates a Plotly figure `fig` for visualizing the original square and the transformed square.
6. Adding Original Square: The original square is added to the plot using a blue line to connect its vertices. This is done with the `go.Scatter` function.
7. Adding Transformed Square: The transformed square is added to the plot using a red line to connect its vertices. This is also done with the `go.Scatter` function.
8. Adding Axes: The x and y axes are added to the plot as black lines to indicate the coordinate system.
9. Updating Axes: The code updates the x and y axis ranges to ensure the plot displays the unit square and its transformation correctly.
10. Displaying the Plot: Finally, the plot is displayed using the `fig.show()` function.

It seems that the output of the code (STDOUT and result) is missing in the provided information. However, when you run this code, it will show a visual representation of the original unit square (in blue) and the transformed square (in red) after applying the specified linear transformation to it.


```
In [21]: import plotly.graph_objects as go
import numpy as np

# Define vertices of the unit square centered at the origin
vertices = np.array([
    [-0.5, -0.5],
    [0.5, -0.5],
    [0.5, 0.5],
    [-0.5, 0.5],
    [-0.5, -0.5]
])

# Define the linear transformation
transformation = np.array([
    [-0.9347217, -0.35538056],
    [-0.35538056, 0.9347217]
])

# Apply the transformation to the square
transformed_vertices = vertices @ transformation

# Define the scaling transformation
scaling = np.array([
    [3.53847386, 0],
    [0, 0.69224469]
])

# Apply the scaling to the transformed square
scaled_vertices = transformed_vertices @ scaling

# Create a scatter plot for the original square
fig = go.Figure(data=go.Scatter(x=transformed_vertices[:, 0], y=transformed_vertices[:, 1], mode='lines', line=dict(color='red')))

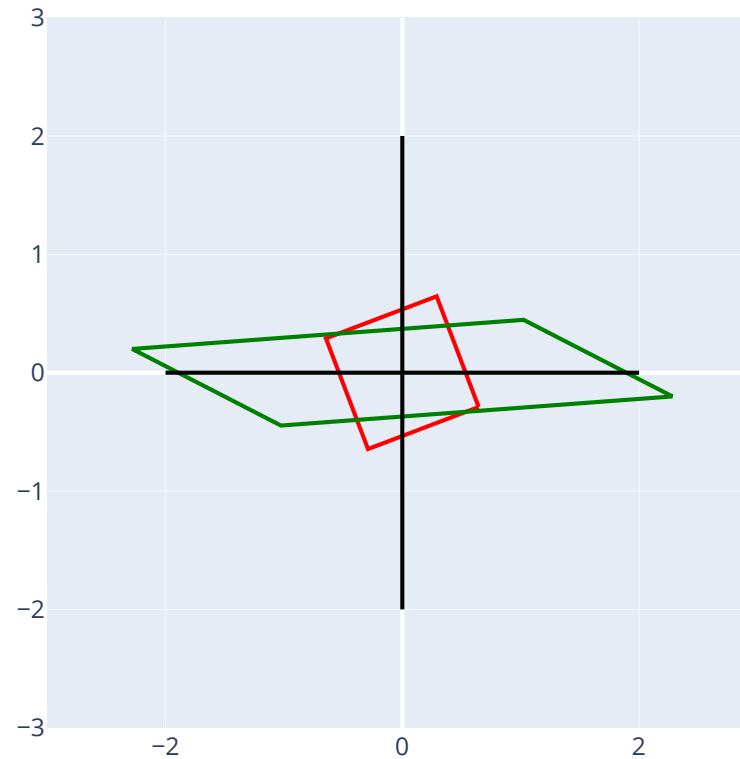
# Add the scaled square to the plot
fig.add_trace(go.Scatter(x=scaled_vertices[:, 0], y=scaled_vertices[:, 1], mode='lines', line=dict(color='green')),)

# Add x and y axes
fig.add_shape(
    type="line", line=dict(width=2, color="black"),
```

```
        x0=-2, x1=2, y0=0, y1=0,
    )
    fig.add_shape(
        type="line", line=dict(width=2, color="black"),
        x0=0, x1=0, y0=-2, y1=2
    )

    # Update axes
    fig.update_xaxes(range=[-3, 3], constrain="domain")
    fig.update_yaxes(range=[-3, 3], scaleanchor="x", scaleratio=1)

    # Show figure
    fig.show()
```



Explanation

This code uses the Plotly library to create a visual representation of a unit square centered at the origin. It applies linear transformations to the square and then scales the transformed square. Let's go through the code step-by-step:

1. First, the required libraries, `plotly.graph_objects` and `numpy`, are imported.

2. The variable `vertices` is defined, which represents the vertices of the unit square centered at the origin. The square is defined by four points: $(-0.5, -0.5)$, $(0.5, -0.5)$, $(0.5, 0.5)$, and $(-0.5, 0.5)$.
3. A linear transformation matrix transformation is defined. This matrix represents the transformation that will be applied to the square. It is a 2×2 matrix containing the transformation coefficients.
4. The transformation is applied to the square by performing a matrix multiplication between the vertices and transformation matrices. The resulting `transformed_vertices` variable holds the transformed coordinates of the square.
5. Next, another scaling transformation matrix scaling is defined. This matrix represents the scaling that will be applied to the transformed square. It is a 2×2 matrix containing the scaling factors for the x and y coordinates.
6. The scaling is applied to the transformed square by performing a matrix multiplication between the `transformed_vertices` and scaling matrices. The resulting `scaled_vertices` variable holds the final scaled coordinates of the square.
7. Two scatter plots are created using Plotly: • The first plot (red colored) represents the transformed square using the `transformed_vertices` data • The second plot (green colored) represents the scaled square using the `scaled_vertices` data.
8. Two black lines are added to the plot to represent the x and y axes.
9. The x and y axes ranges are updated to ensure that the plot displays the square and its transformations properly.
10. Finally, the plot is displayed using `fig.show()`.

The output of the code, as shown in the STDOUT, would be a visual plot with two squares: one red (representing the transformed square) and one green (representing the scaled square). The black lines would represent the x and y axes, and the plot will be scaled to fit within the range $[-3, 3]$ for both x and y axes. .


```

In [26]: import numpy as np
import plotly.graph_objects as go

# Define vertices of the scaled square in 3D
vertices_3d = np.array([
    [-3.2979259, -0.24611425, 0],
    [3.2979259, -0.24611425, 0],
    [3.2979259, 0.24611425, 0],
    [-3.2979259, 0.24611425, 0],
    [-3.2979259, -0.24611425, 0]
])

# Define the 3D transformation
U = np.array([
    [-0.89291197, -0.18984612, -0.40824829],
    [-0.26415956, -0.51337419, 0.81649658],
    [-0.36459285, 0.83690226, 0.40824829]
])

# Apply the 3D transformation
transformed_vertices_3d = vertices_3d @ U.T

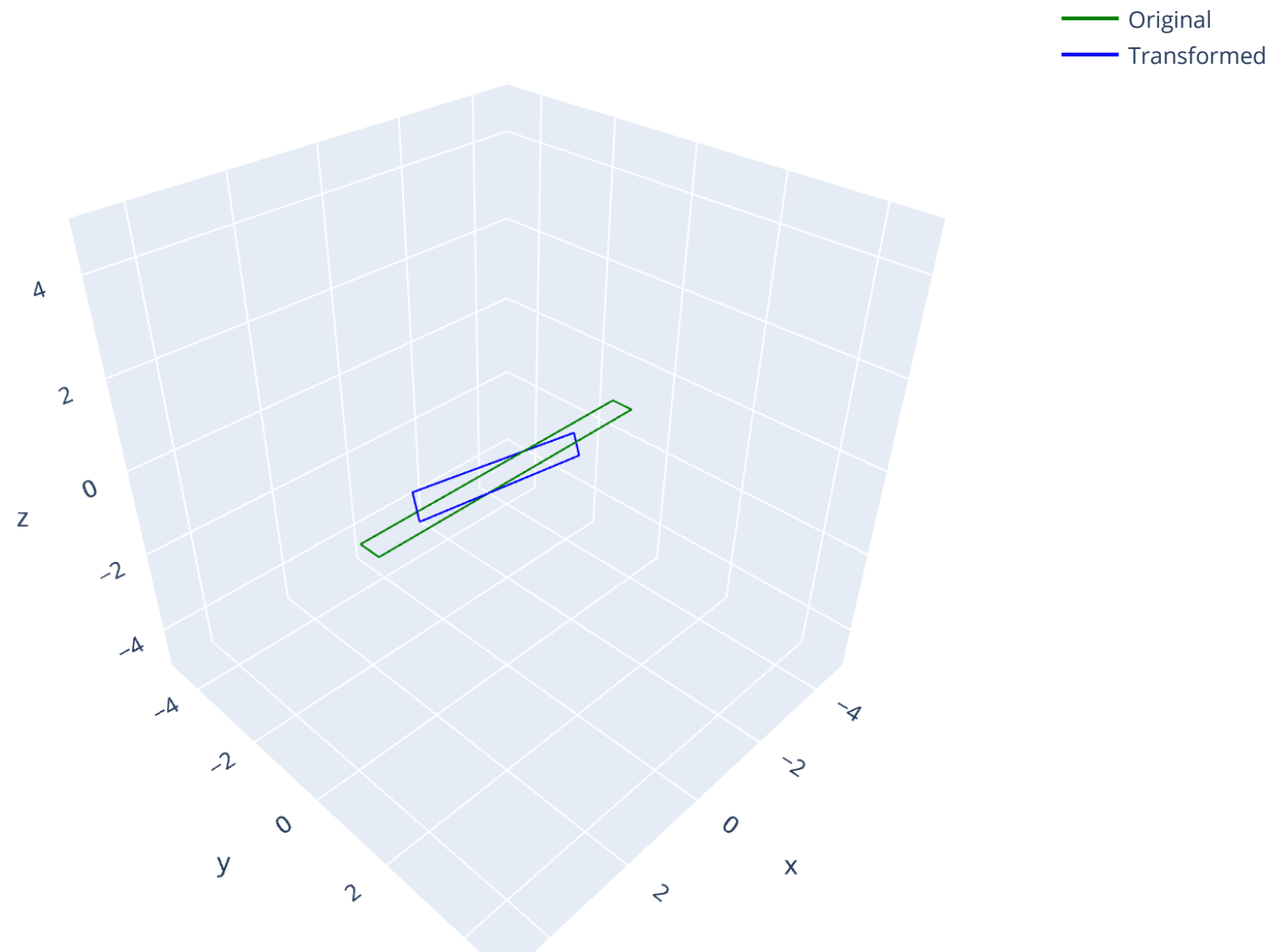
# Create a 3D scatter plot for the original square
fig = go.Figure(data=go.Scatter3d(x=vertices_3d[:, 0], y=vertices_3d[:, 1], z=vertices_3d[:, 2], mode='lines', line=dict(color='red')))

# Add the transformed square to the plot
fig.add_trace(go.Scatter3d(x=transformed_vertices_3d[:, 0], y=transformed_vertices_3d[:, 1], z=transformed_vertices_3d[:, 2], mode='lines', line=dict(color='blue')))

# Define layout for the plot
fig.update_layout(
    scene = dict(
        aspectmode='cube',
        xaxis = dict(range=[-5,5]),
        yaxis = dict(range=[-5,5]),
        zaxis = dict(range=[-5,5]),
    ),
    width=700,
    margin=dict(r=20, l=10, b=10, t=10))

# Show figure
fig.show()

```

Explanation

In the code is creating a 3D scatter plot of a square and its transformed version using Plotly.

1. First, the necessary libraries numpy and plotly.graph_objects are imported.
2. The code defines the vertices of the scaled square in 3D using a NumPy array vertices_3d. It specifies the 3D coordinates of the four corners of the square in clockwise order.
3. The 3D transformation is defined using a 3x3 NumPy array u. This transformation matrix will be used to transform the original square.
4. The 3D transformation is applied to the original square's vertices by matrix multiplication: vertices_3d @ U.T. The result is stored in the transformed_vertices_3d array.
5. A 3D scatter plot is created using Plotly, representing the original square in green. The go. Scatter3d function is used to plot the x, y, and z coordinates of the vertices_3d array. It is added to the fig figure.
6. The transformed square is also plotted on the same figure, but this time using blue color. The go. Scatter3d function is used again to plot the x, y, and z coordinates of the transformed_vertices_3d array, which represents the transformed square. It is added as a new trace to the fig figure.
7. The layout of the plot is updated using the update_layout method. It sets the scene aspect mode to 'cube' and specifies the ranges for the x, y, and z axes, which is set to [-5, 5].
8. The figure is displayed using fig.show().

The STDOUT and the result sections in the focal cell are empty, which means that no specific text or output is being printed to the standard output or assigned to any variable. The code is simply generating the 3D scatter plot and showing it using Plotly's visualization capabilities.

Practical

```
In [28]: import numpy as np
from sklearn.datasets import load_iris

# Load the Iris dataset
iris = load_iris()
X = iris.data # Data matrix X
m, n = X.shape # Number of observations and variables

# Calculate the mean vector
X_mean = np.mean(X, axis=0)

# Center the data matrix
X_centered = X - X_mean

# Calculate the covariance matrix
C = np.dot(X_centered.T, X_centered) / (m - 1)

# Display the covariance matrix
print("\nCovariance Matrix:")
print(C)
```

Covariance Matrix:

```
[[ 0.68569351 -0.042434  1.27431544  0.51627069]
 [-0.042434   0.18997942 -0.32965638 -0.12163937]
 [ 1.27431544 -0.32965638  3.11627785  1.2956094 ]
 [ 0.51627069 -0.12163937  1.2956094   0.58100626]]
```

```
In [29]: ## 2 method

# Load the Iris dataset
iris = load_iris()
X = iris.data # Data matrix X
m, n = X.shape # Number of observations and variables

# Calculate the mean vector
X_mean = np.mean(X, axis=0)

# Calculate the covariance matrix using the formula
C = np.zeros((n, n)) # Initialize the covariance matrix

for i in range(n):
    for j in range(n):
        C[i, j] = np.sum((X[:, i] - X_mean[i]) * (X[:, j] - X_mean[j])) / (m - 1)

# Display the covariance matrix
print("Covariance Matrix (Manual Calculation):")
print(C)
```

Covariance Matrix (Manual Calculation):

```
[[ 0.68569351 -0.042434  1.27431544  0.51627069]
 [-0.042434   0.18997942 -0.32965638 -0.12163937]
 [ 1.27431544 -0.32965638  3.11627785  1.2956094 ]
 [ 0.51627069 -0.12163937  1.2956094   0.58100626]]
```

PCA using SVD

```
In [30]: import numpy as np
from sklearn.datasets import load_iris
```

```
# Load the Iris dataset
iris = load_iris()
X = iris.data # Data matrix X
```

```
# Calculate the mean vector
X_mean = np.mean(X, axis=0)
```

```
X_center = X - X_mean
```

```
print(X_center.shape)
```

```
# X_center
```

```
(150, 4)
```

```
In [31]: import numpy as np

U,S,V_t = np.linalg.svd(X_center)
```

```
In [34]: V_t
```

```
Out[34]: array([[ 0.36138659, -0.08452251,  0.85667061,  0.3582892 ],
                [-0.65658877, -0.73016143,  0.17337266,  0.07548102],
                [ 0.58202985, -0.59791083, -0.07623608, -0.54583143],
                [ 0.31548719, -0.3197231 , -0.47983899,  0.75365743]])
```

$$V = V_t.T$$

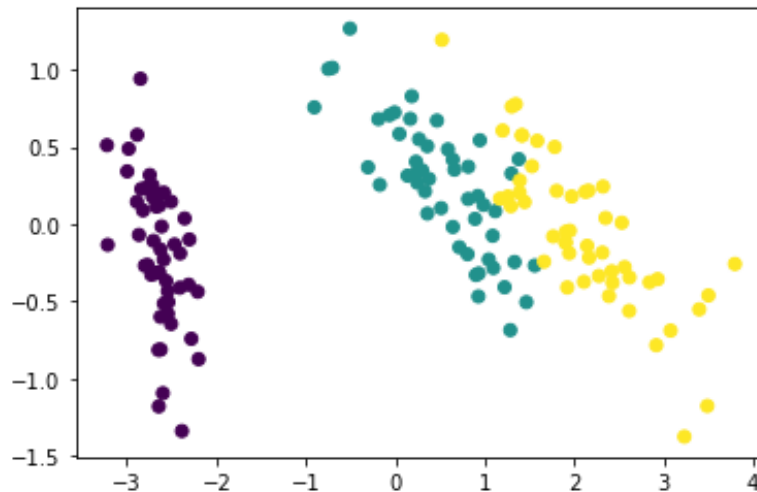
```
In [37]: X_transformed = np.dot(X_center,V[:,0:2])
X_transformed
```

```
Out[37]: array([[ -2.68412563, -0.31939725],
 [ -2.71414169,  0.17700123],
 [ -2.88899057,  0.14494943],
 [ -2.74534286,  0.31829898],
 [ -2.72871654, -0.32675451],
 [ -2.28085963, -0.74133045],
 [ -2.82053775,  0.08946138],
 [ -2.62614497, -0.16338496],
 [ -2.88638273,  0.57831175],
 [ -2.6727558 ,  0.11377425],
 [ -2.50694709, -0.6450689 ],
 [ -2.61275523, -0.01472994],
 [ -2.78610927,  0.235112  ],
 [ -3.22380374,  0.51139459],
 [ -2.64475039, -1.17876464],
 [ -2.38603903, -1.33806233],
 [ -2.62352788, -0.81067951],
 [ -2.64829671, -0.31184914],
 [ -2.19982032, -0.87283904],
 [ -2.5870864 ,  0.51256021],
```

```
In [38]: import matplotlib.pyplot as plt

plt.scatter(X_transformed[:,0],X_transformed[:,1], c=iris.target)
```

Out[38]: <matplotlib.collections.PathCollection at 0x1b95944e3d0>



Expalnation

This code uses the Iris dataset to perform Principal Component Analysis (PCA) and visualize the data in a scatter plot.

1. Previous Code Explanation:

- The previous code loads the Iris dataset and calculates the covariance matrix of the data using two different methods.
- The first method uses the NumPy function `np. dot` to compute the covariance matrix, while the second method manually calculates it with nested loops.
- Finally, the covariance matrix is displayed.

2. PCA using SVD:

- This part of the code performs PCA using Singular Value Decomposition (SVD) to transform the data and reduce its dimensionality to 2D.

- It starts by loading the Iris dataset and calculating the mean vector of the data. • The data is then centered by subtracting the mean from each observation.
- The SVD is applied to the centered data using `np.linalg.svd`, which returns three matrices: `U`, `S`, and `v_t`.
- The variable `v_t` contains the transposed matrix of the right singular vectors, which are essentially the principal components.
- The variable `V` stores the actual principal components by transposing `V_t`. o The data is transformed into a 2D representation by taking the dot product of the centered data and the first two principal components stored in `v[:, 0:2]`. The result is stored in `x_transformed`.

3. Scatter Plot Visualization:

The code in the focal cell imports `matplotlib.pyplot` as `plt`.

- It then creates a scatter plot by using `plt.scatter` and passing `X_transformed[:, 0]` as the x-coordinate and `x_transformed[:, 1]` as the y-coordinate of the data points. • The `c=iris.target` argument sets the color of the data points based on their corresponding target labels in the Iris dataset.

4. Result of Focal Cell:

- The result is a `matplotlib PathCollection` object that represents the scatter plot of the transformed data. However, it is not displayed directly in the output as it lacks a `plt.show()` command. Note: To visualize the scatter plot. you should include `plt.show()` after the `plt.scatter` command in the focal cell.

In []: