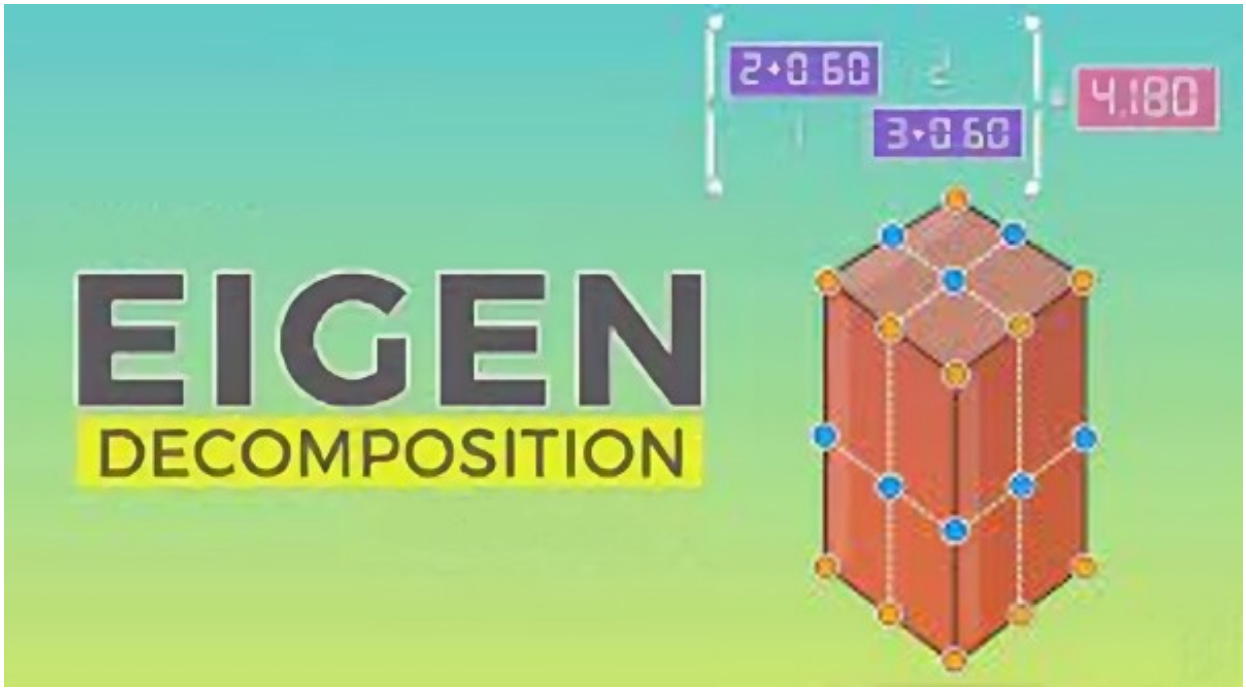
What is Eigen Decomposition?

Eigen decomposition, also known as eigendecomposition, is a fundamental concept in linear algebra. It involves decomposing a square matrix into a set of eigenvectors and eigenvalues.



The eigen decomposition of a matrix A is expressed as:

A = V * Λ * V^(-1)

Where:

- A is the square matrix being decomposed.
- V is a matrix whose columns are the eigenvectors of A.
- Λ is a diagonal matrix whose diagonal elements are the eigenvalues of A.

To put it simply, eigenvectors are special vectors that only change by a scalar factor when multiplied by a matrix. Eigenvalues, on the other hand, are the corresponding scalar factors. When you perform the eigen decomposition of a matrix, you find the eigenvectors and eigenvalues that satisfy the equation:

A * v = λ * v

Where:

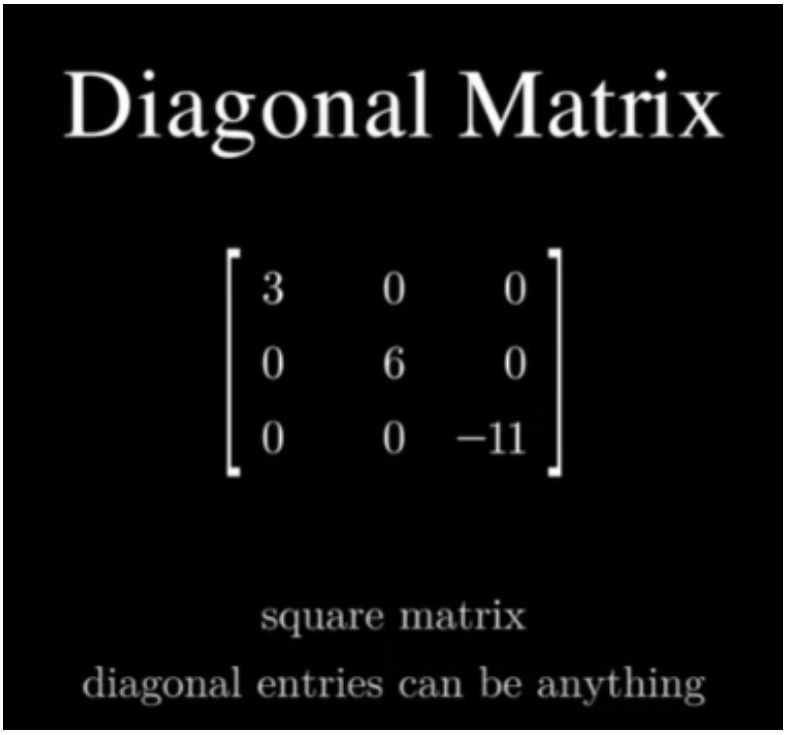
- A is the original matrix.
- v is an eigenvector of A.
- λ is the corresponding eigenvalue.

The eigen decomposition is useful in many areas of mathematics and science, including linear transformations, differential equations, data analysis, and quantum mechanics. It provides insights into the behavior and properties of matrices, allowing for efficient calculations and simplification of various mathematical operations.

Some Special Matrices

Diagonal Matrix

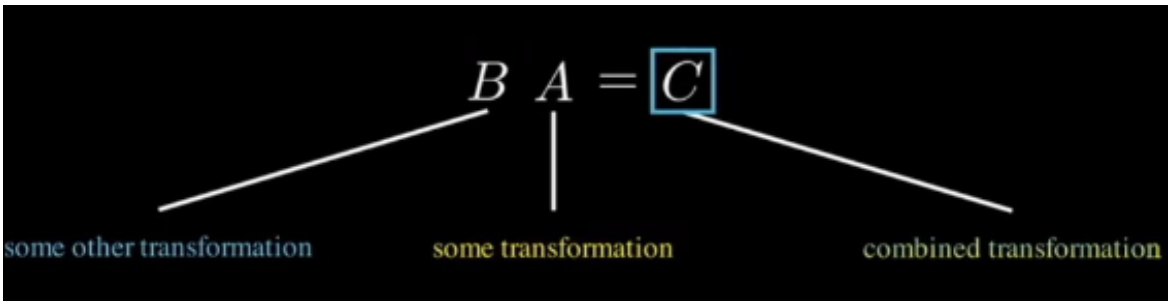
A diagonal matrix is a square matrix in which all the off-diagonal elements are zero. In other words, a matrix is diagonal if it has non-zero elements only on its main diagonal, which extends from the top-left to the bottom-right.



For example, consider a 3x3 diagonal matrix D:

D = | d1 0 0 | | 0 d2 0 | | 0 0 d3 |

Where d1, d2, and d3 are the diagonal elements of the matrix.



The main properties of diagonal matrices are as follows:

1. **Non-zero diagonal elements**: The diagonal elements of a diagonal matrix are the only non-zero elements, and all other elements are zero.

In [4]:

```
import numpy as np
A = np.diag([1, 2, 3])
A
```

Out[4]:

```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

2. **Operations on diagonal matrices**: Addition and subtraction of diagonal matrices can be done element-wise, and multiplication of diagonal matrices involves multiplying corresponding diagonal elements.

In [5]:

```
A = np.diag([1, 2, 3])
B = np.diag([4, 5, 6])
C = A + B
C
```

Out[5]:

```
array([[5, 0, 0],
       [0, 7, 0],
       [0, 0, 9]])
```

In [8]:

```
A = np.diag([1, 2, 3])
B = np.diag([4, 5, 6])
C = A * B
C
```

Out[8]:

```
array([[ 4,  0,  0],
       [ 0, 10,  0],
       [ 0,  0, 18]])
```

3. **Powers of diagonal matrices**: Raising a diagonal matrix to a power n involves raising each diagonal element to the power n.

In [13]:

```
A = np.diag([1, 2, 3])

A_squared = np.diag([1**2, 2**2, 3**2])

print(A_squared)
```

```
[[1 0 0]
 [0 4 0]
 [0 0 9]]
```

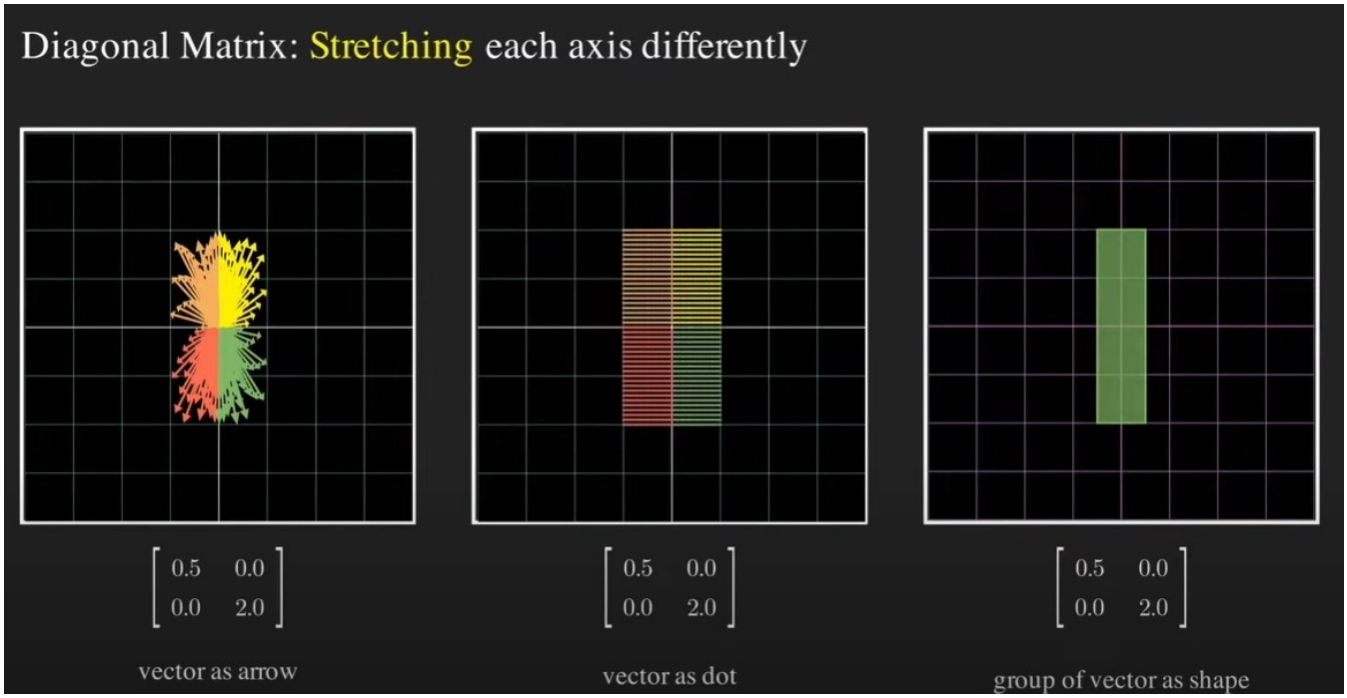
4. **Inverse of a diagonal matrix**: A diagonal matrix D is invertible if and only if all its diagonal elements are non-zero. The inverse of a diagonal matrix is obtained by taking the reciprocal of each non-zero diagonal element.

In [9]:

```
A = np.diag([1, 2, 3])
A.T
```

Out[9]:

```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

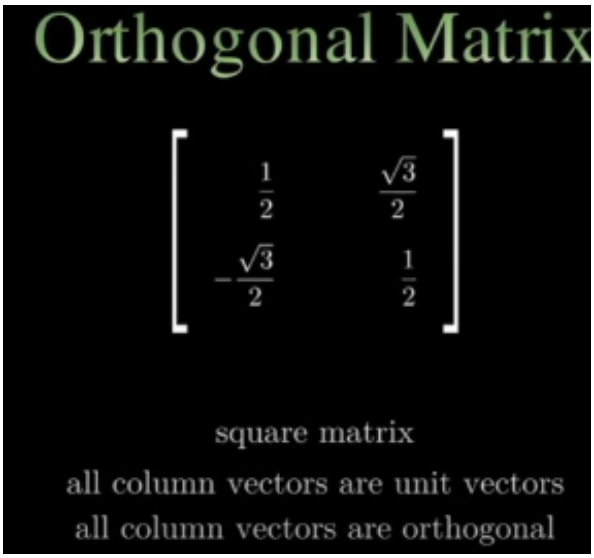


Diagonal matrices are often used in various mathematical operations and applications, including diagonalization of matrices, solving systems of linear equations, representing diagonalizable linear transformations, and performing element-wise operations efficiently. They simplify calculations and allow for easy manipulation of matrix operations due to their special structure.

Orthogonal Matrix

An orthogonal matrix is a square matrix whose columns (and rows) form an orthonormal set of vectors. In other words, an orthogonal matrix satisfies the following conditions:

1. **The dot product of any two distinct columns (or rows) is zero, indicating orthogonality.**
2. **Each column (or row) has a unit length, indicating normalization.**



Inverse Equals Transpose: The transpose of an orthogonal matrix equals its inverse,

i.e., A^T = A^(-1).

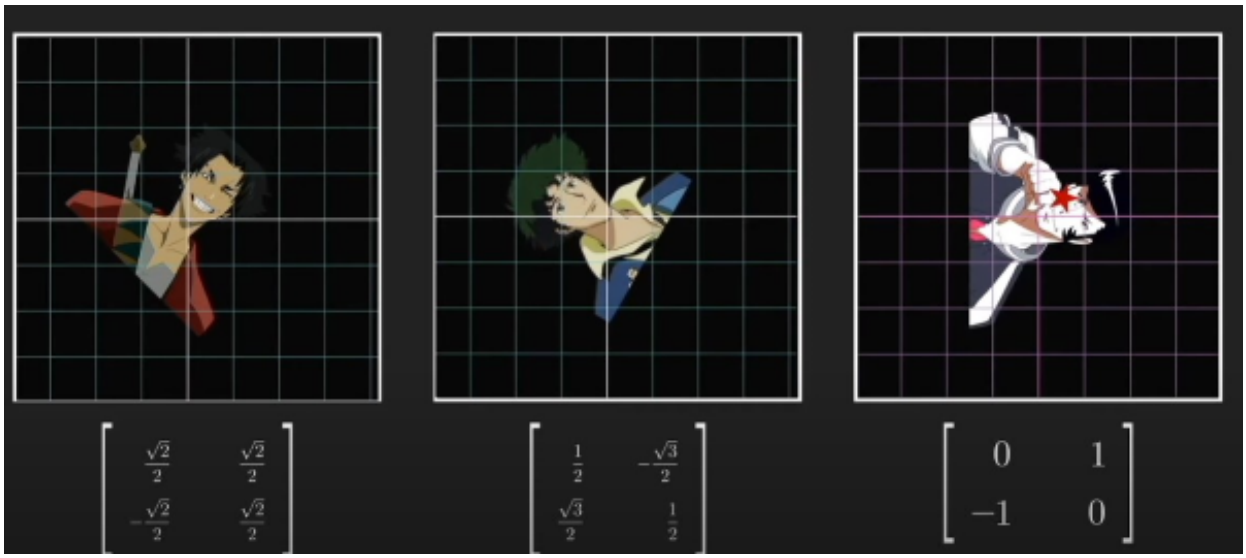
This property makes calculations with orthogonal matrices computationally efficient.

Mathematically, an orthogonal matrix Q satisfies the equation:

Q^T * Q = Q * Q^T = I

Where:

- Q is the orthogonal matrix.
- Q^T is the transpose of Q.
- I is the identity matrix.



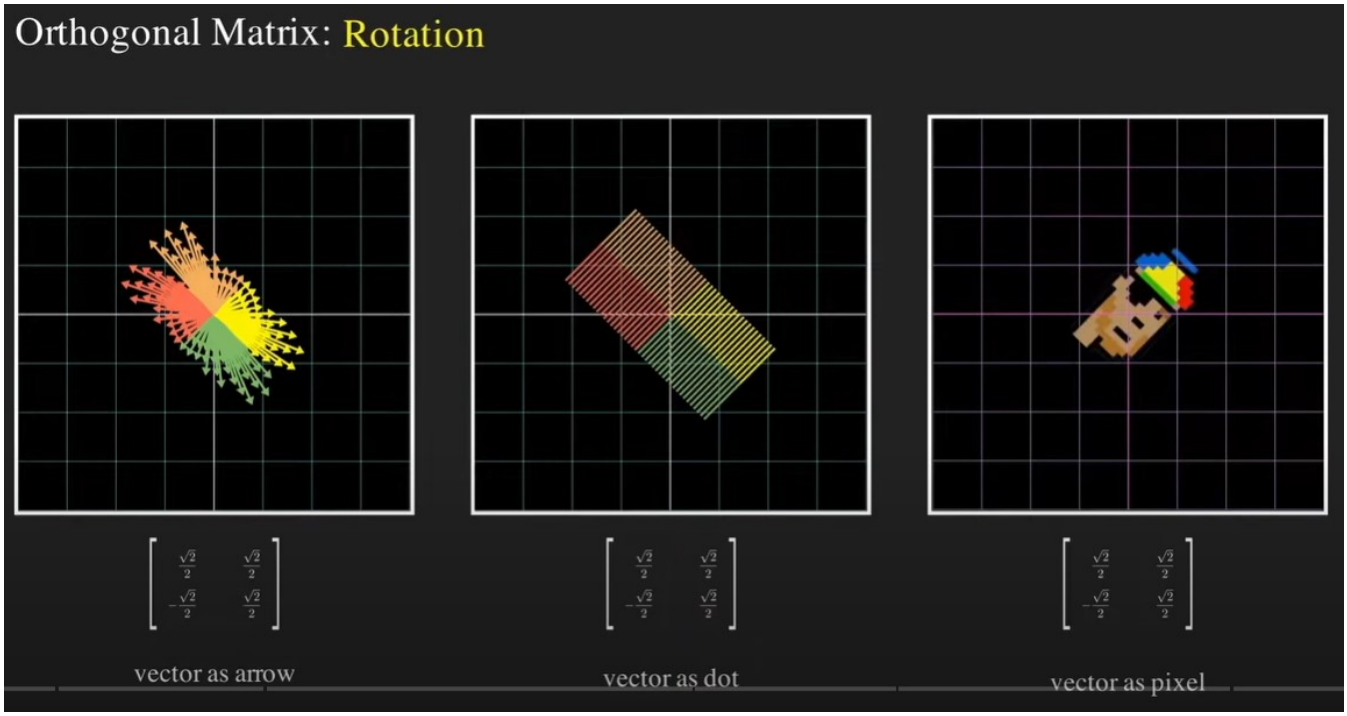
The properties of orthogonal matrices have several important implications:

1. **Orthogonal matrices preserve vector lengths**: If a vector x is multiplied by an orthogonal matrix Q, the length of the resulting vector Qx remains the same as the length of x.

2. **Orthogonal matrices preserve angles**: If two vectors u and v form an angle θ, then the angle between the vectors Qu and Qv is also θ after multiplication by an orthogonal matrix Q.

3. **Inverse of an orthogonal matrix**: The inverse of an orthogonal matrix is equal to its transpose. That is, if Q is an orthogonal matrix, then Q^(-1) = Q^T.

4. **Determinant of an orthogonal matrix**: The determinant of an orthogonal matrix is either +1 or -1. Since the columns (or rows) of an orthogonal matrix are orthonormal, their cross product (or dot product) results in ±1.



In [14]:

```
import numpy as np

def is_orthogonal(A):
  """Returns True if A is an orthogonal matrix, False otherwise."""
  if not np.allclose(A.dot(A.T), np.eye(A.shape[0])):
    return False
  if not np.allclose(np.linalg.det(A), 1):
    return False
  return True

def main():
  A = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
  print("A is orthogonal:", is_orthogonal(A))

  B = np.array([[1, 0, 0], [0, 0, 1], [0, 1, 0]])
  print("B is orthogonal:", is_orthogonal(B))

  C = np.array([[1, 0, 0], [0, 1, 0], [0, 0, -1]])
  print("C is orthogonal:", is_orthogonal(C))

if __name__ == "__main__":
  main()
```

```
A is orthogonal: True
B is orthogonal: False
C is orthogonal: False
```

Explanation:

1. The code starts by importing the NumPy library, which is commonly used for numerical computations in Python.

2. Next, there is a function called is_orthogonal(A). This function takes a matrix A as input and returns True if A is orthogonal, and False otherwise. The function is defined using a docstring, which provides a brief description of the function's purpose.
3. Inside the is_orthogonal function, there are three main steps to check if a matrix is orthogonal:

    - First, it uses NumPy's dot() function to calculate the dot product of A with its transpose (A.dot(A.T)). This operation checks if the matrix multiplication of A with its transpose results in the identity matrix (np.eye(A.shape[0])). The np.allclose() function is used to check if the two matrices are approximately equal. If they are not equal, it means the matrix A does not satisfy the condition of preserving vector lengths, so the function returns False.

    * The second step checks if the determinant of matrix A is equal to 1. It use
    s NumPy's linalg.det() function to calculate the determinant of A. If the det
    erminant is not equal to 1, it means the matrix A does not satisfy the condit
    ion of having a determinant of +1 or -1, so the function returns False.

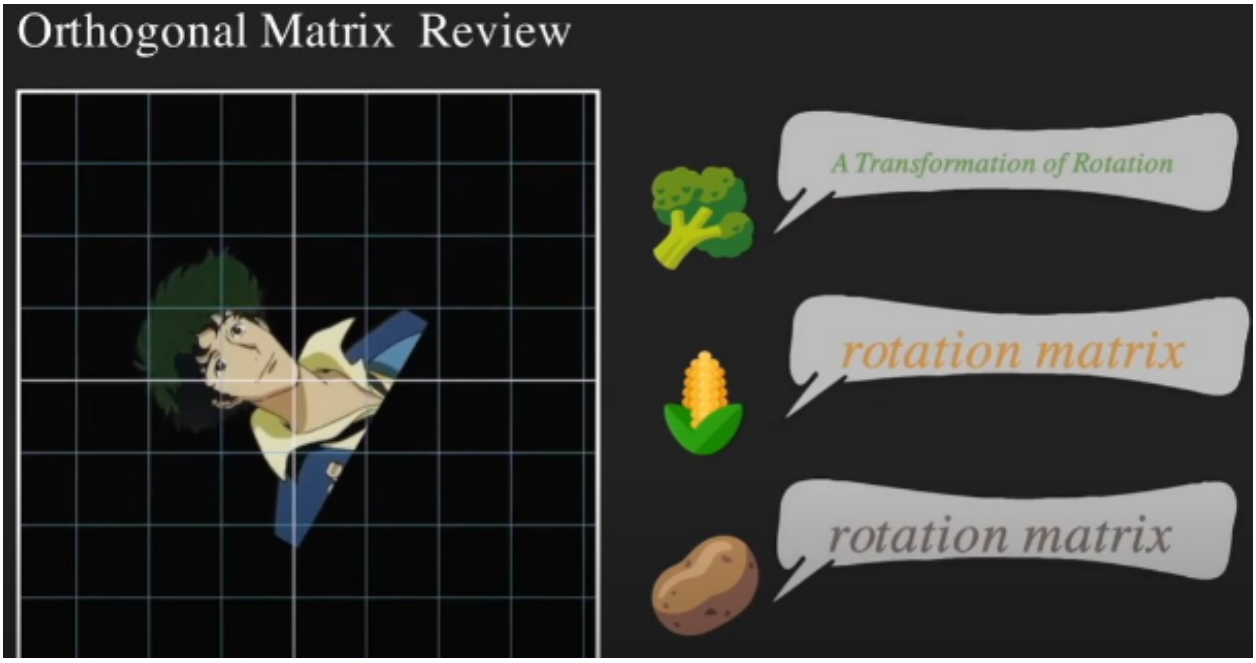
    *Finally, if the matrix A passes both checks, the function returns True, indi
    cating that A is orthogonal.

4. The code also defines a main() function. This function serves as the entry point of the script and contains the code that demonstrates the usage of the is_orthogonal() function.

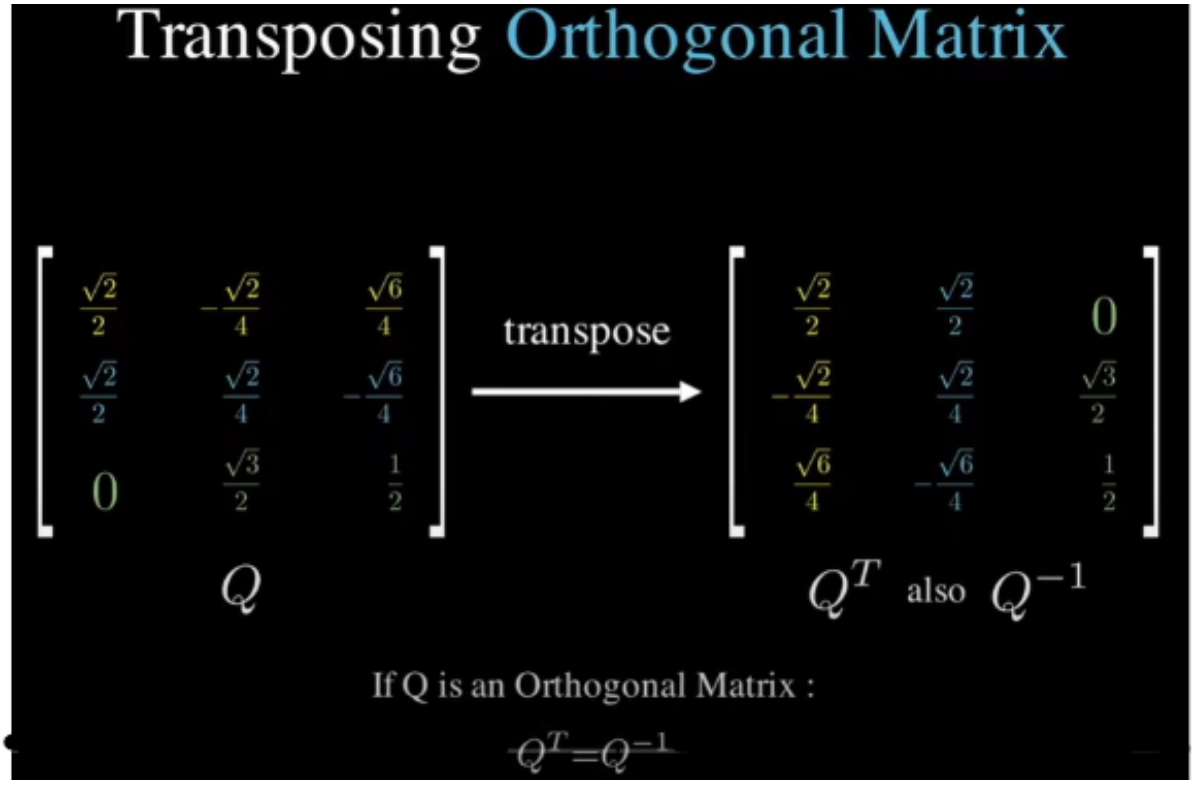
5. Inside the main() function, three matrices A, B, and C are defined using NumPy's array() function. These matrices are examples used to test the is_orthogonal() function.
6. Each matrix is then passed to the is_orthogonal() function, and the result is printed to the console using the print() function. The output indicates whether each matrix is orthogonal (True) or not (False).

7. Finally, the script checks if the **name** variable is equal to "**main**", which is true when the script is executed directly and not imported as a module. If it is the main script being executed, the main() function is called.

When you run the script, it checks the orthogonality of the matrices A, B, and C and prints the results to



Orthogonal matrices find numerous applications in various fields, including linear transformations, rotations, reflections, orthogonalization algorithms, solving linear systems of equations, data compression, and computer graphics, among others. They play a significant role in preserving geometric properties and simplifying calculations involving matrices.

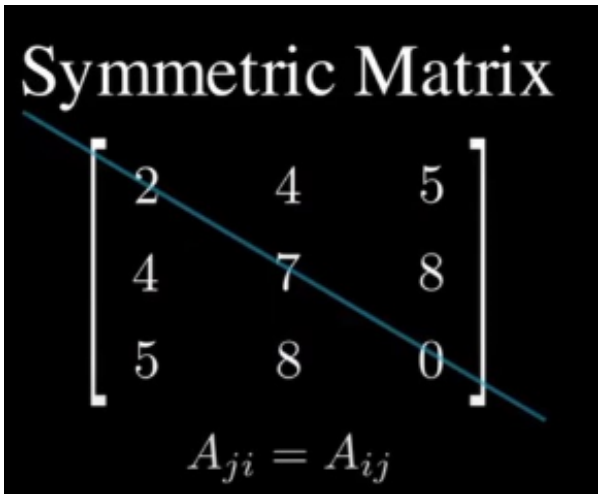


Symmertic Matrix

A symmetric matrix is a square matrix that is equal to its transpose. In other words, an n×n matrix A is symmetric if and only if for every element A(i, j), it holds that A(i, j) = A(j, i) for all i and j.

Mathematically, a symmetric matrix A satisfies the condition:

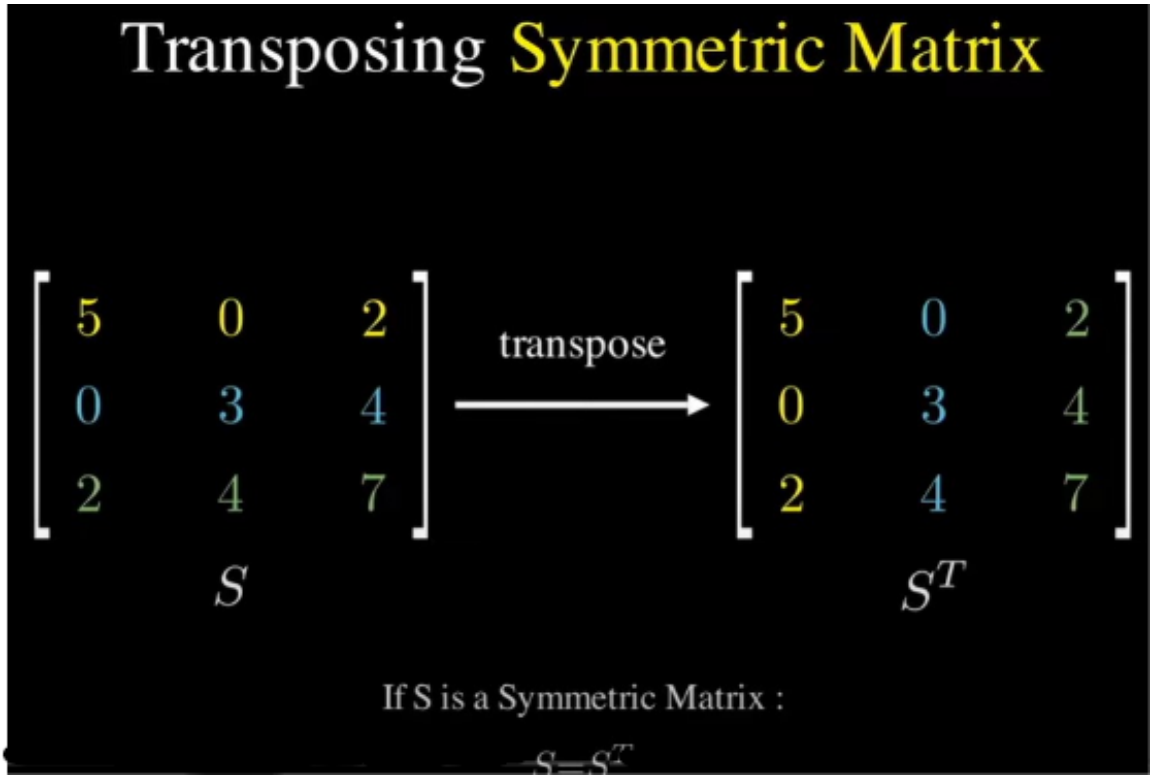
A = A^T



This means that the elements above and below the main diagonal are reflections of each other.

Here are some properties and examples of symmetric matrices:

1. **Diagonal elements**: The diagonal elements of a symmetric matrix remain the same when reflected along the main diagonal.
2. **Off-diagonal elements**: The off-diagonal elements of a symmetric matrix appear in pairs, where the element A(i, j) is equal to the element A(j, i).
3. **Eigenvalues**: Symmetric matrices have real eigenvalues. Additionally, eigenvectors corresponding to distinct eigenvalues are orthogonal to each other.
4. **Operations on symmetric matrices**: Addition and subtraction of symmetric matrices can be done element-wise, and the result remains symmetric. Multiplication of two symmetric matrices may not produce a symmetric matrix, except in the special case where the two matrices commute.



Examples of symmetric matrices:

- Example 1:

  A = | 3 1 4 |

  | 1 6 -2 |
  | 4 -2 8 |

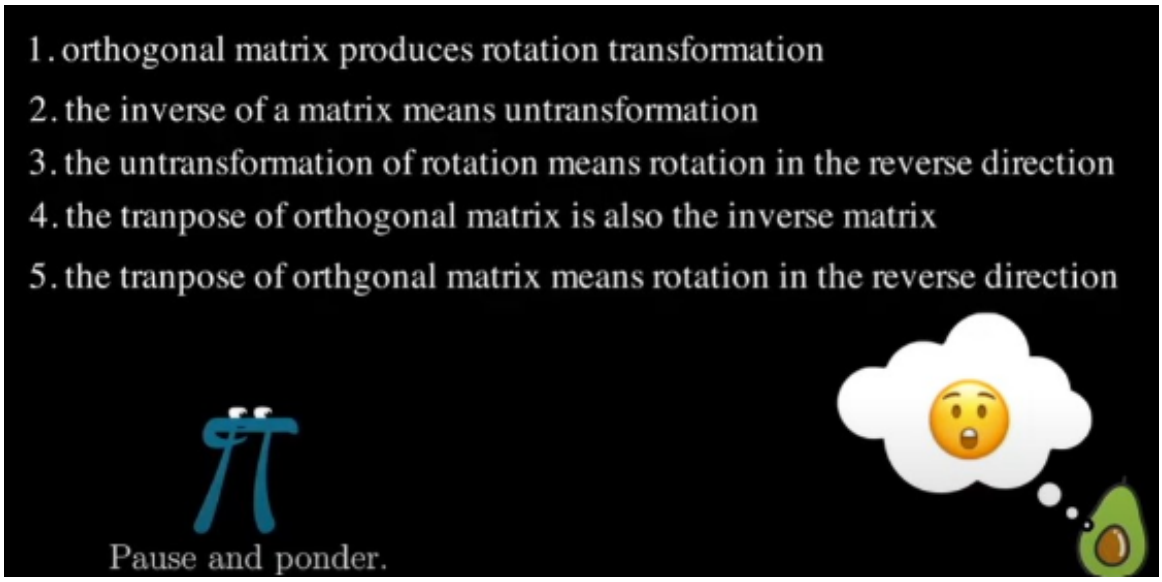
  This matrix is symmetric since A = A^T.
- Example 2:

  B = | 2 5 7 |

  | 5 1 -3 |
  | 7 -3 9 |

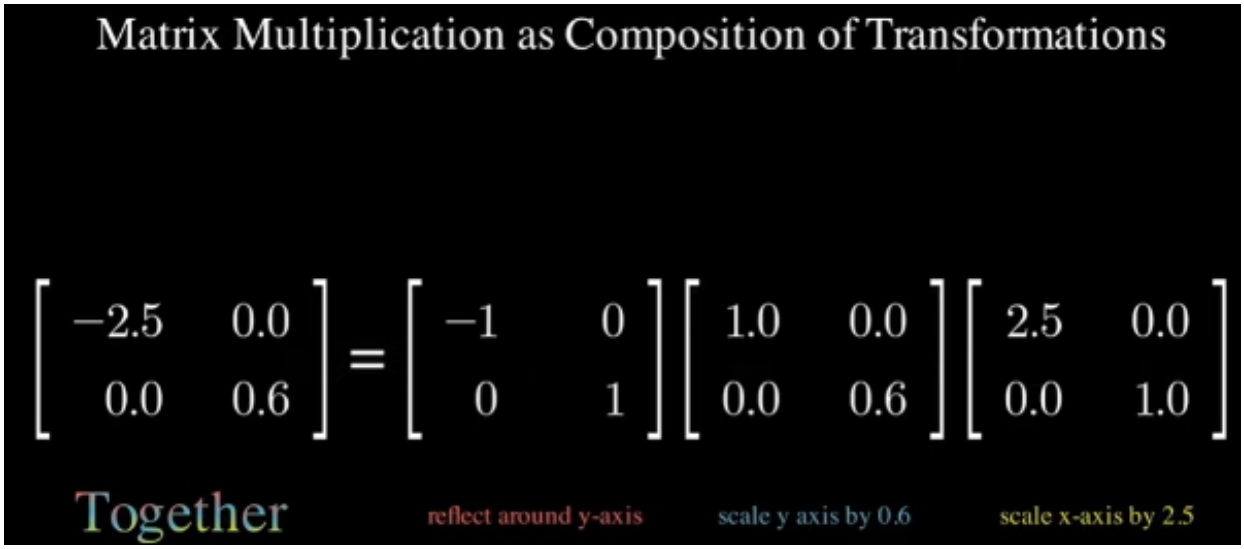
  This matrix is not symmetric since B ≠ B^T.

Symmetric matrices have important properties and applications in various areas, such as linear algebra, optimization, eigenvalue problems, quadratic forms, and physics. The symmetry property simplifies computations and allows for efficient algorithms and solutions in many mathematical and scientific contexts.

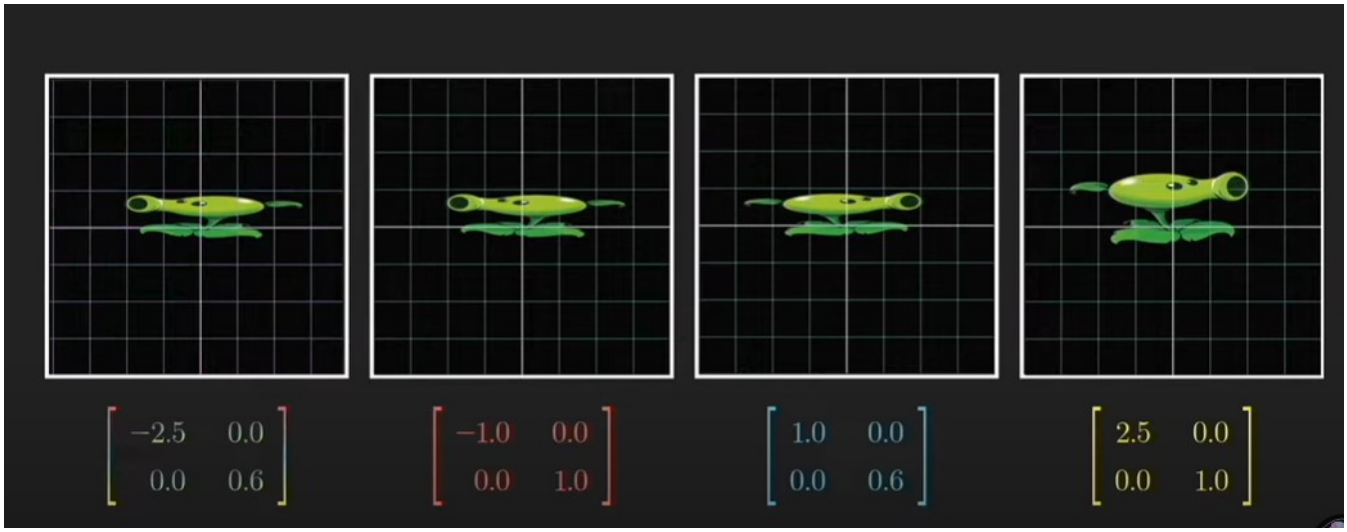


What is Eigen Composition?

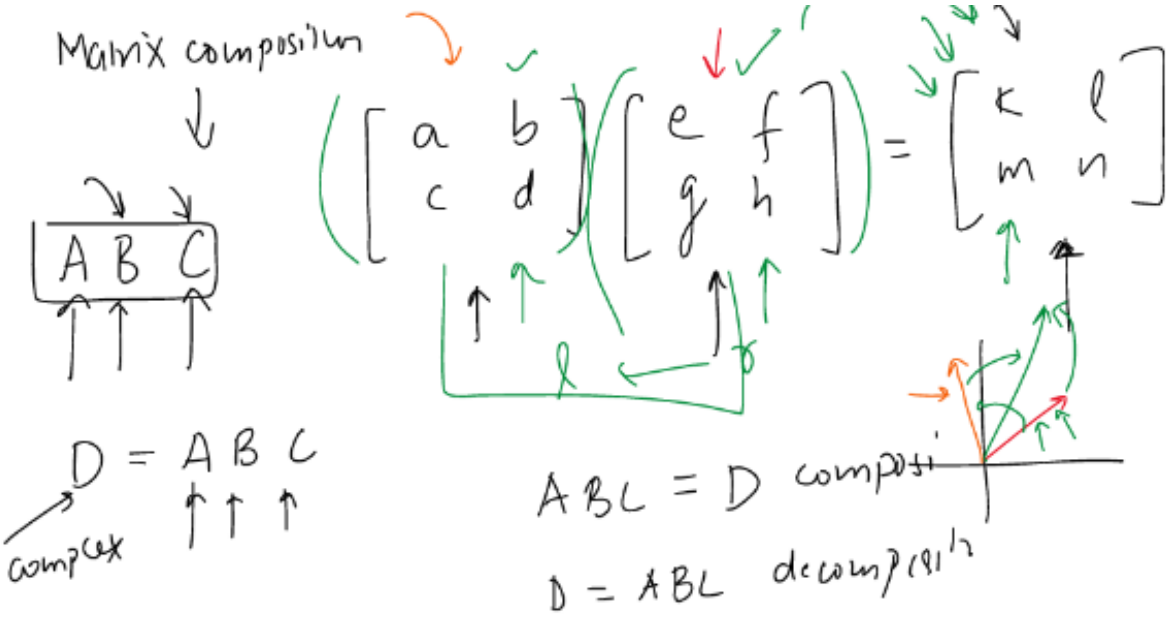
Eigen composition is a term commonly used in mathematics, specifically in linear algebra and matrix theory. In simple terms, eigen composition refers to the process of combining multiple eigenvalues and eigenvectors to create a new matrix or vector.



To understand eigen composition, we first need to understand eigenvalues and eigenvectors. In linear algebra, an eigenvector is a non-zero vector that, when multiplied by a matrix, only changes in scale (magnitude), but not in direction. The corresponding scalar value that represents the change in scale is called the eigenvalue. Eigenvalues and eigenvectors are fundamental concepts in the study of linear transformations.



Explanation



What is Matrix Decompostion?

Matrix decomposition is a technique that breaks down a matrix into smaller, more manageable matrices. This can be useful for a variety of tasks, such as:

There are many different types of matrix decomposition, but some of the most common include:

- **Eigendecomposition:** This decomposes a matrix into its eigenvalues and eigenvectors. The eigenvalues are the scaling factors that stretch or shrink the eigenvectors, and the eigenvectors are the directions along which the matrix stretches or shrinks the eigenvectors.
- **Singular value decomposition:** This decomposes a matrix into three matrices: a diagonal matrix of singular values, a matrix of left singular vectors, and a matrix of right singular vectors. The singular values are the square roots of the eigenvalues of the matrix product AA^T, and the left and right singular vectors are the eigenvectors of AA^T and A^TA, respectively.

- **Cholesky decomposition:** This decomposes a symmetric, positive-definite matrix into the product of a lower triangular matrix and its transpose. The lower triangular matrix is called the Cholesky factor of the matrix.
- **LU decomposition:** This decomposes a matrix into the product of a lower triangular matrix and an upper triangular matrix. The lower triangular matrix is called the L factor of the matrix, and the upper triangular matrix is called the U factor of the matrix.

Matrix decomposition is a powerful tool that can be used to solve a variety of problems. It is a relatively

what is Eigen Decomposition?

Eigen decomposition, also known as eigendecomposition, is a specific type of matrix decomposition that factors a square matrix A into the product of a diagonal matrix D and a matrix P consisting of eigenvectors.

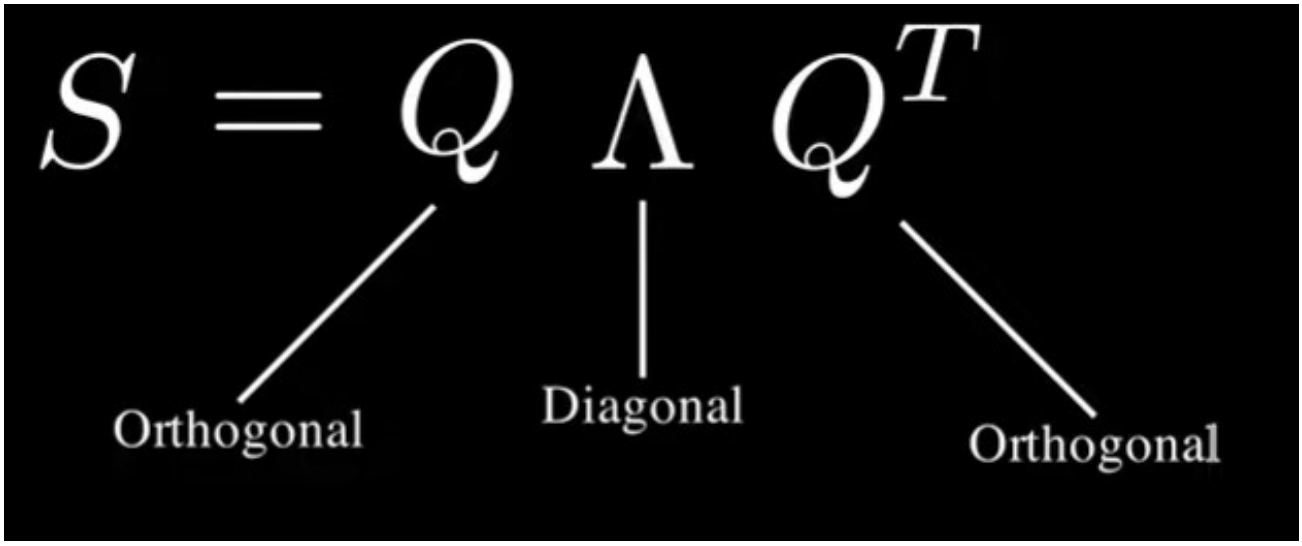
If A is a square matrix, eigen decomposition can be expressed as:

A = VΛV-1

where:

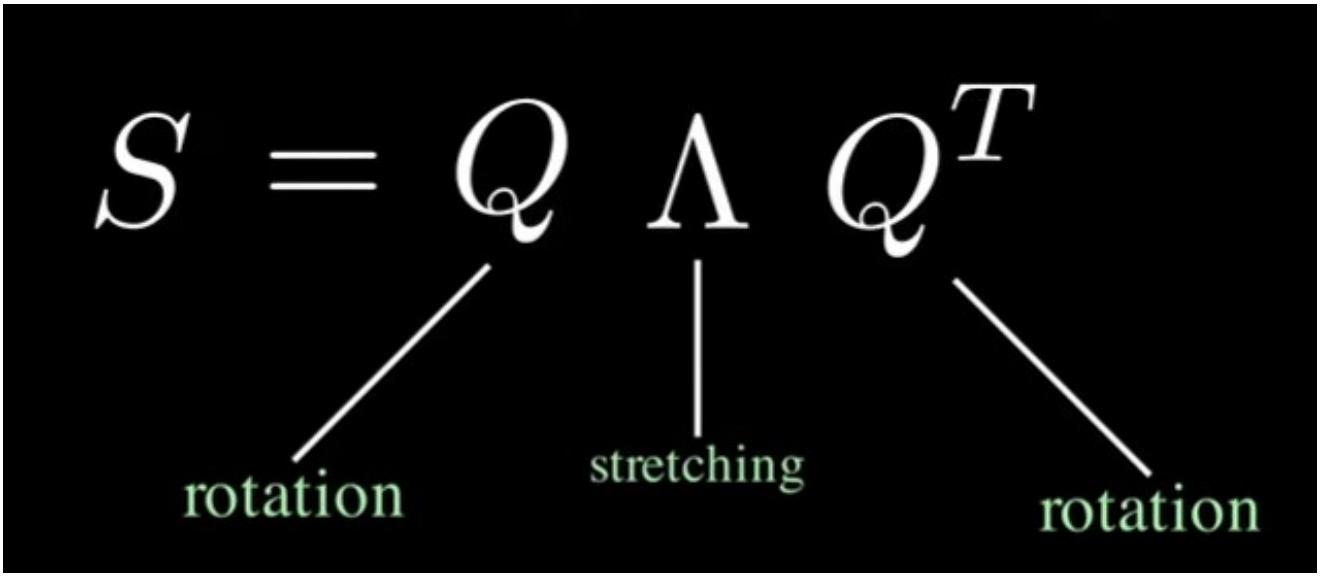
- V is a matrix whose columns are the eigenvectors of A•
- Λ is a diagonal matrix whose entries are the eigenvalues of A•
- V-1 is the inverse of V

also written as :



Assuming:

- **Square matrix**: Eigen decomposition is only defined for square matrices1.
- **Diagonalizability**: For a n*n matrix it should have n linearly independent eigen vectors



Mathematically, for a given eigenpair (λ, v), the eigen equation is defined as:

$Av = \lambda v$

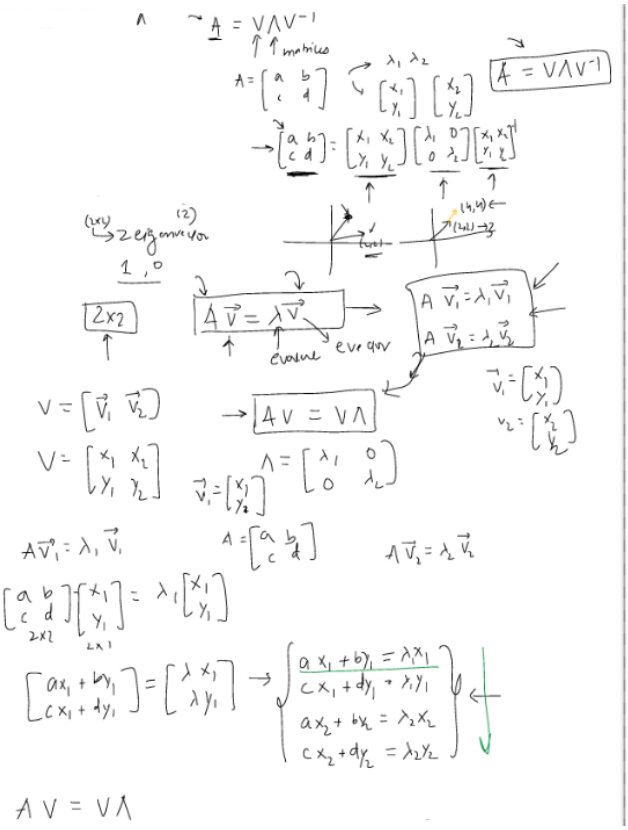
To compute the eigen decomposition, we follow these steps:

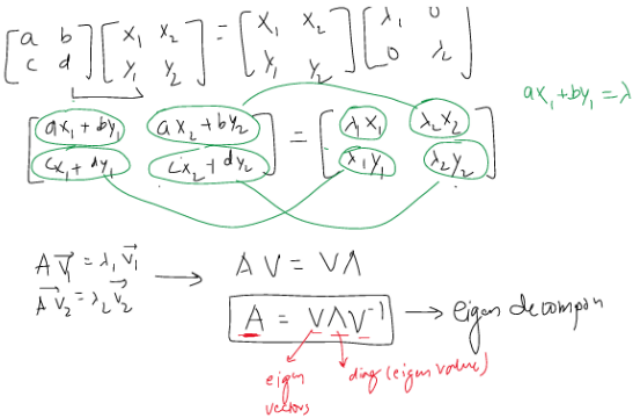
1. Compute the eigenvalues of A by solving the characteristic equation $\det(A - \lambda I) = 0$, where I is the identity matrix.
2. For each eigenvalue, find the corresponding eigenvector by solving the equation $(A - \lambda I)v = 0$.
3. Collect all the eigenvectors in matrix P, where each column represents an eigenvector.
4. Construct the diagonal matrix D using the eigenvalues. The eigenvalues are placed on the diagonal of D, while the off-diagonal elements are zero.

The eigen decomposition allows us to express the original matrix A in terms of its eigenvalues and eigenvectors. This decomposition is particularly useful for understanding the properties and behavior of linear transformations, analyzing the stability of dynamical systems, solving systems of linear differential equations, and diagonalizing matrices.

It is important to note that not all matrices have a complete eigen decomposition. Some matrices may have repeated eigenvalues, or they may be non-diagonalizable. In such cases, alternative decomposition techniques, such as Jordan decomposition, may be used.

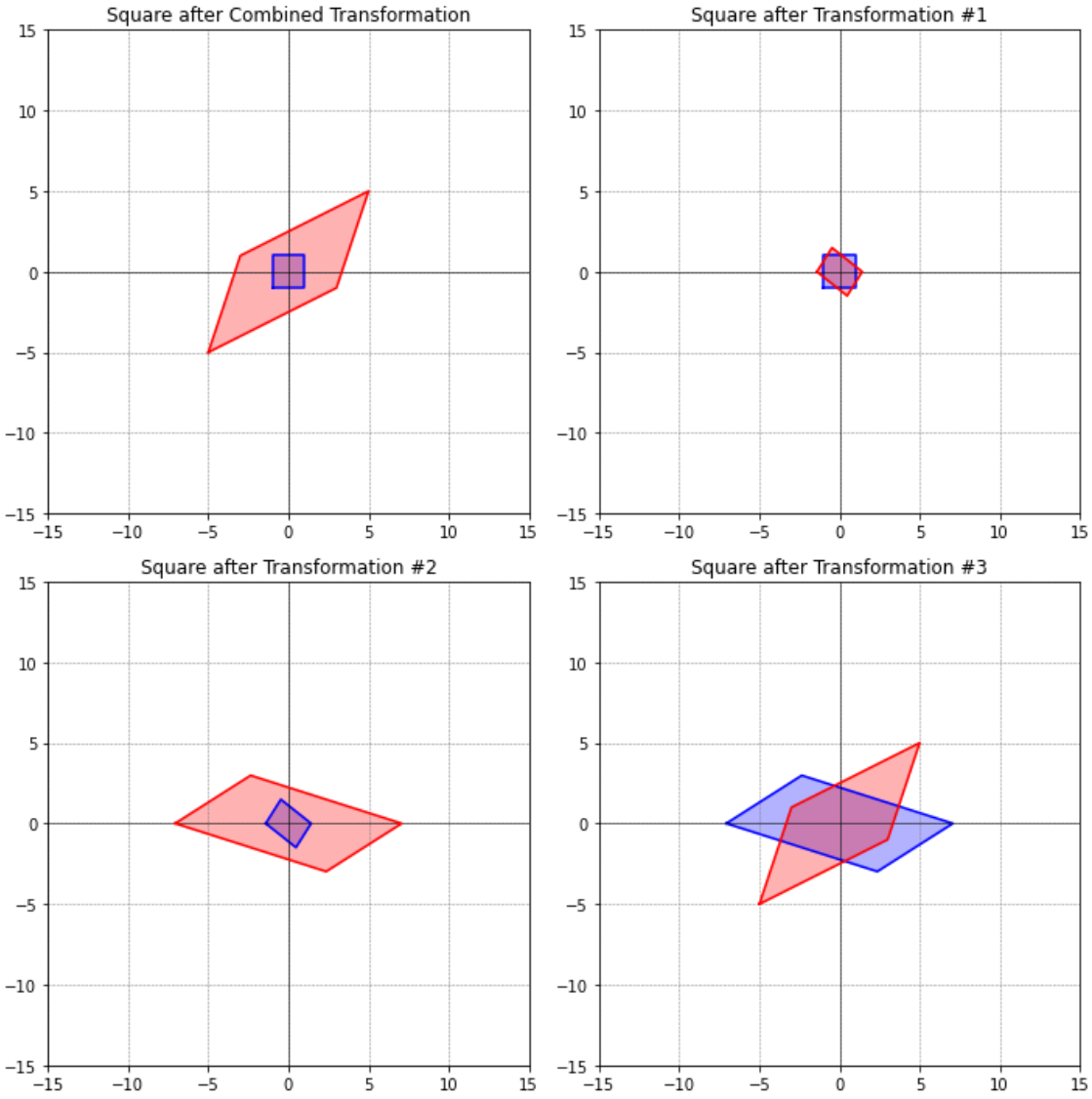
Explanation:





In [17]:

```
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import display, clear_output

# Plot initial coordinate axis and unit square
def plot_square(ax, square):
    ax.plot(square[0, :], square[1, :], 'b')
    ax.fill(square[0, :], square[1, :], 'blue', alpha=0.3)
    ax.set_xlim(-15, 15)
    ax.set_ylim(-15, 15)
    ax.axhline(0, color='black',linewidth=0.5)
    ax.axvline(0, color='black',linewidth=0.5)
    ax.grid(color = 'gray', linestyle = '--', linewidth = 0.5)

# Apply transformation and plot
def plot_transformed_square(ax, matrix, square):
    transformed_square = np.dot(matrix, square)
    ax.plot(transformed_square[0, :], transformed_square[1, :], 'r')
    ax.fill(transformed_square[0, :], transformed_square[1, :], 'red', alpha=0.3)
    return transformed_square

# Initialize unit square
square = np.array([[-1, -1], [-1, 1], [1, 1], [1, -1], [-1, -1]]).T

# Inputs for 3 2x2 matrices
matrix1 = np.array([[ 0.94280904, 0.47140452], [-0.74535599, 0.74535599]])  # Replace
matrix2 = np.array([[5, 0], [0, 2]])  # Replace with your matrix
matrix3 = np.array([[ 0.70710678, -0.4472136 ],[ 0.70710678, 0.89442719]])  # Replace
matrices = [matrix1, matrix2, matrix3]

# Calculate product of all three matrices
product_matrix = np.dot(matrix3, np.dot(matrix2, matrix1))

# Create 2x2 subplot grid
fig, axs = plt.subplots(2, 2, figsize=(10, 10))

# Plot combined transformation as the first graph
plot_square(axs[0, 0], square)
square_combined = plot_transformed_square(axs[0, 0], product_matrix, square)
axs[0, 0].set_title('Square after Combined Transformation')

# Plot individual transformations as the remaining graphs
for i, (matrix, ax) in enumerate(zip(matrices, axs.flat[1:]), start=1):
    plot_square(ax, square)
    square = plot_transformed_square(ax, matrix, square)
    ax.set_title(f'Square after Transformation #{i}')

# Adjust layout to prevent overlapping titles
plt.tight_layout()
plt.show()
```


Square after Combined Transformation
Square after Transformation #1
Square after Transformation #2
Square after Transformation #3

In [18]:

```
import numpy as np

# Define your square matrix
A = np.array([[4, 1], [2, 3]])

# Perform the eigen decomposition
eigenvalues, V = np.linalg.eig(A)

# Create the diagonal matrix of eigenvalues
Lambda = np.diag(eigenvalues)

# Compute the inverse of V
V_inv = np.linalg.inv(V)

# Verify that A = VΛV^(-1)
A_reconstructed = np.dot(V, np.dot(Lambda, V_inv))

# Print the matrices
print("Matrix V (Eigenvectors of A as columns):")
print(V)
print("\nMatrix Λ (Diagonal matrix of Eigenvalues):")
print(Lambda)
print("\nInverse of V:")
print(V_inv)
print("\nReconstructed A (Should be close to original A):")
print(A_reconstructed)
```

```
Matrix V (Eigenvectors of A as columns):
[[ 0.70710678 -0.4472136 ]
 [ 0.70710678  0.89442719]]

Matrix Λ (Diagonal matrix of Eigenvalues):
[[5. 0.]
 [0. 2.]]

Inverse of V:
[[ 0.94280904  0.47140452]
 [-0.74535599  0.74535599]]

Reconstructed A (Should be close to original A):
[[4. 1.]
 [2. 3.]]
```

Eigendecomposition of a symmetric matrix

Eigendecomposition of a symmetric matrix is a technique that decomposes a symmetric matrix into its eigenvalues and eigenvectors. The eigenvalues are the scaling factors that stretch or shrink the eigenvectors, and the eigenvectors are the directions along which the matrix stretches or shrinks the eigenvectors.

The eigendecomposition of a symmetric matrix can be written as follows:

$$A = QΛQ^T$$

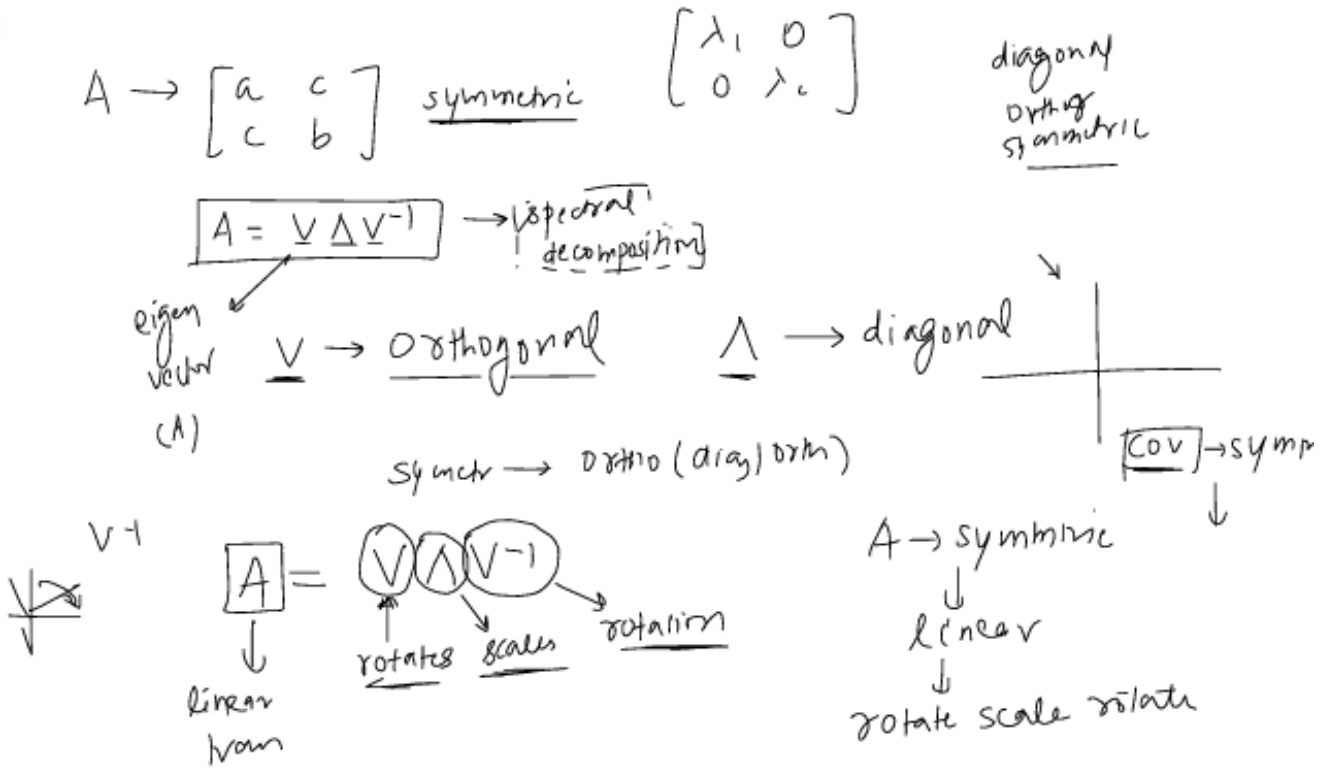
where A is the symmetric matrix, Q is a matrix of eigenvectors, and Λ is a diagonal matrix of eigenvalues.

The eigenvalues of a symmetric matrix are always real numbers. This is because the transpose of a symmetric matrix is equal to itself, which means that the eigenvalues of A are the same as the eigenvalues of A^T.

The eigenvectors of a symmetric matrix are orthogonal to each other. This means that the dot product of any two eigenvectors is equal to zero.

The eigendecomposition of a symmetric matrix can be used for a variety of tasks, such as:

- **Understanding the structure of a matrix:** The eigenvalues of a symmetric matrix can be used to understand the structure of the matrix. For example, if all of the eigenvalues of a matrix are equal, then the matrix is a scalar matrix.
- **Solving linear equations:** The eigendecomposition of a symmetric matrix can be used to solve linear equations more efficiently than other methods. This is because the inverse of a symmetric matrix can be written in terms of the eigenvalues and eigenvectors of the matrix.
- **Data compression:** The eigendecomposition of a symmetric matrix can be used to compress data by representing it in a more compact form. This is because the eigenvectors of a symmetric matrix can be used to represent the data in a lower-dimensional space.



Advantages of Eigen decomposition

Eigen decomposition offers several advantages and benefits in various areas of mathematics, science, and engineering. Here are some key advantages of eigen decomposition:

1. **Diagonalization**: Eigen decomposition allows a matrix to be diagonalized. By expressing a matrix in terms of its eigenvalues and eigenvectors, the resulting diagonal matrix represents the matrix in a

simpler and more understandable form. Diagonal matrices are particularly useful for computations and analysis, as they have many desirable properties and simplifications.

2. **Understanding Matrix Behavior**: Eigen decomposition provides insight into the behavior of a matrix. The eigenvectors represent the principal directions or basis vectors of the matrix, while the eigenvalues indicate the scaling or importance of each eigenvector. This information helps understand how a matrix transforms vectors and provides a geometric interpretation of its operations.

3. **Dimensionality Reduction**: Eigen decomposition is employed in techniques like Principal Component Analysis (PCA). PCA uses eigenvalues and eigenvectors to transform high-dimensional data into a lower-dimensional space while preserving the most important information. It aids in reducing data complexity, extracting meaningful features, and visualizing high-dimensional data.

4. **Solving Systems of Linear Equations**: Eigen decomposition can simplify the solution of systems of linear equations involving matrices. By decomposing a matrix into its eigenvectors and eigenvalues, the system of equations can be transformed into a diagonal system, making it easier to solve and analyze. This is particularly useful in solving linear differential equations and studying dynamic systems.

5. **Matrix Powers and Exponentiation**: Eigen decomposition facilitates the computation of matrix powers and exponentiation. Using the diagonalized form of a matrix, raising the matrix to a power or computing its exponential becomes straightforward. This simplification is utilized in various applications, including simulations, iterative algorithms, and dynamical systems analysis.

6. **Matrix Approximation and Compression**: Eigen decomposition can be used for matrix approximation and compression. By retaining only the most significant eigenvectors and eigenvalues, a matrix can be approximated using a smaller number of terms. This technique is used in data compression, signal processing, and image compression, where it reduces storage requirements and computational complexity while preserving essential information.

Overall, eigen decomposition is a versatile tool that aids in understanding the behavior of matrices, simplifying computations, extracting important features, and reducing the dimensionality of data. Its applications extend to various fields, including linear algebra, statistics, machine learning, physics, and engineering.

PCA (Principal Component Analysis) Varients

There are many different variants of PCA, but some of the most common include:

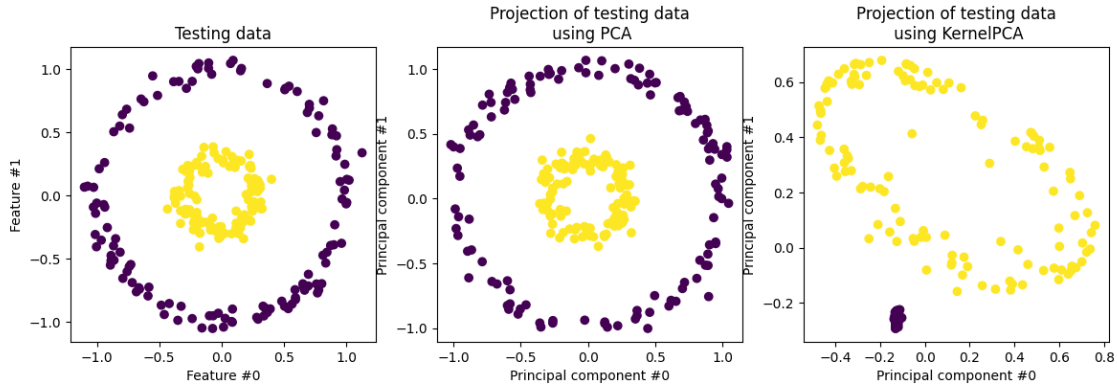
- **Kernel PCA:** Kernel PCA is a variant of PCA that uses a kernel function to map the data into a higher-dimensional space. This allows the algorithm to find more complex patterns in the data.
- **Sparse PCA:** Sparse PCA is a variant of PCA that encourages the eigenvectors to be sparse. This can be useful for data that is sparse, such as text data.
- **Incremental PCA:** Incremental PCA is a variant of PCA that can be used to process data incrementally. This can be useful for large datasets that cannot be loaded into memory all at once.
- **Randomized PCA:** Randomized PCA is a variant of PCA that uses randomized algorithms to find the eigenvectors. This can be faster than traditional PCA, but it may not be as accurate.

Each of these variants has its own advantages and disadvantages. Kernel PCA can find more complex patterns in the data, but it can be slower than traditional PCA. Sparse PCA can be useful for sparse data, but it can be less accurate than traditional PCA. Incremental PCA can be used to process large datasets, but it may not be as accurate as traditional PCA. Randomized PCA can be faster than traditional PCA, but it may not be as accurate.

The best variant of PCA to use depends on the specific application. For example, if you need to find complex patterns in the data, then kernel PCA may be a good choice. If you have sparse data, then sparse PCA may be a good choice. If you need to process a large dataset, then incremental PCA may be a good choice. If you need to find the eigenvectors quickly, then randomized PCA may be a good choice.

Kernel PCA (Principal Component Analysis)

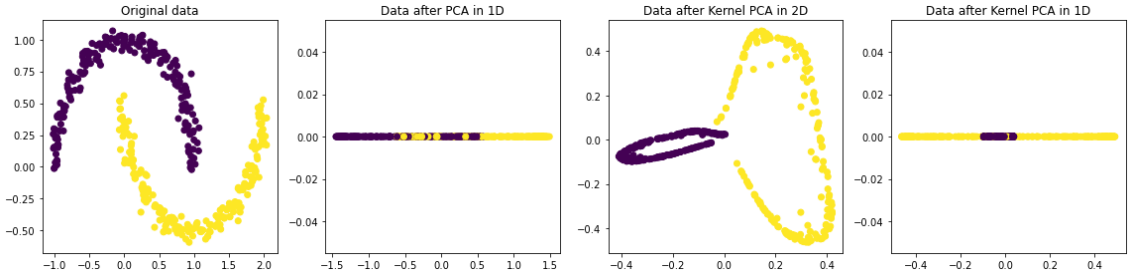
Kernel PCA (KPCA) is an extension of Principal Component Analysis (PCA) that allows for nonlinear dimensionality reduction. Traditional PCA is a linear technique that finds the principal components by performing a linear transformation on the data. In contrast, KPCA applies a nonlinear transformation to project the data into a higher-dimensional feature space where linear PCA can be performed.



The key idea behind KPCA is to use a kernel function to implicitly map the original data points into a higher-dimensional space. This kernel function computes the similarity or inner product between pairs of data points in the original feature space. By using a suitable kernel, KPCA can capture nonlinear relationships among the data points.

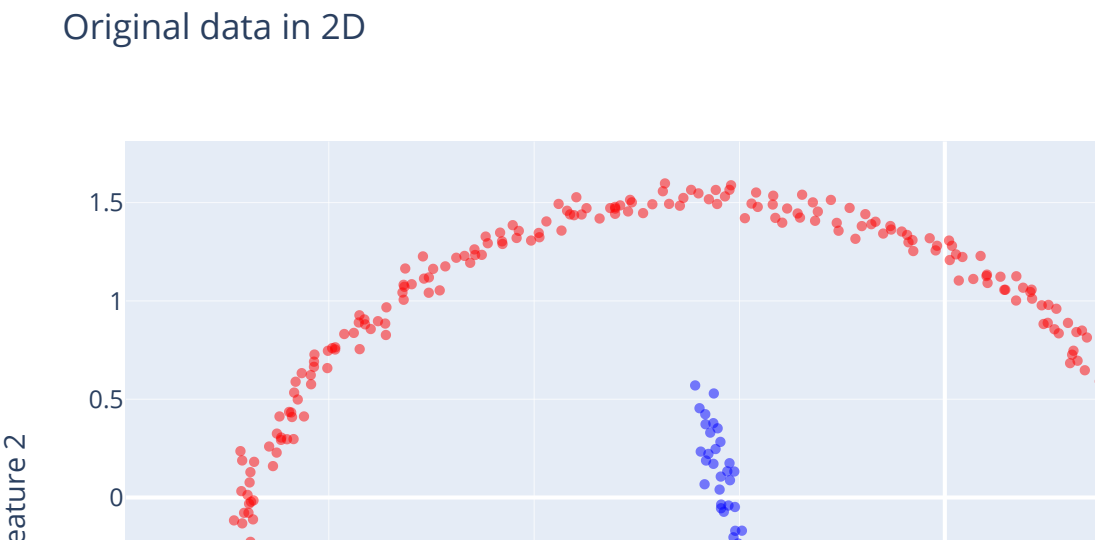
In [20]:

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA, KernelPCA
from sklearn.datasets import make_moons

# Generate the dataset
X, y = make_moons(n_samples=400, noise=.05)

# Apply PCA
pca = PCA()
X_pca = pca.fit_transform(X)

# Apply Kernel PCA
kpca = KernelPCA(kernel="rbf", gamma=15)
X_kpca = kpca.fit_transform(X)

# Original data plot
plt.figure(figsize=(16, 4))
plt.subplot(1, 4, 1)
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.title('Original data')

# Transformed data with PCA in 1D
plt.subplot(1, 4, 2)
plt.scatter(X_pca[:, 0], np.zeros((400,)), c=y)
plt.title('Data after PCA in 1D')

# Transformed data with Kernel PCA in 2D
plt.subplot(1, 4, 3)
plt.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y)
plt.title('Data after Kernel PCA in 2D')

# Transformed data with Kernel PCA in 1D
plt.subplot(1, 4, 4)
plt.scatter(X_kpca[:, 1], np.zeros((400,)), c=y)
plt.title('Data after Kernel PCA in 1D')

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



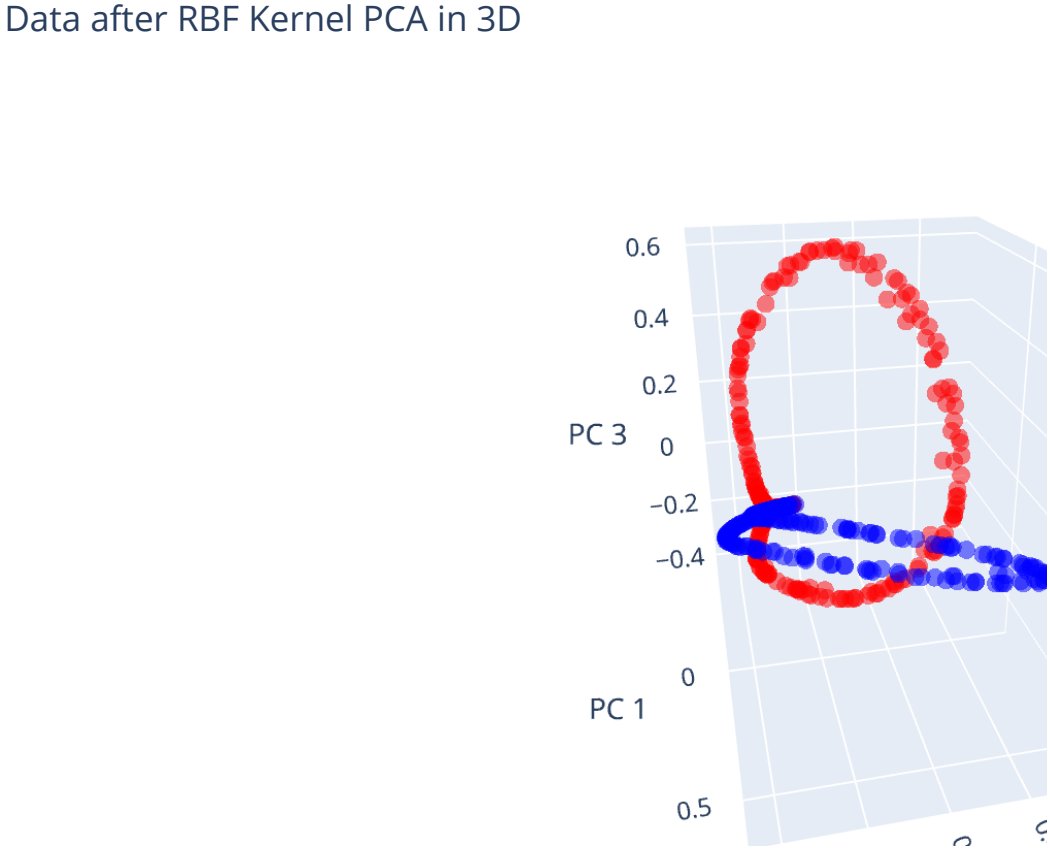
In [36]:

```
import plotly.graph_objs as go
from sklearn.datasets import make_moons
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import KernelPCA
import numpy as np

# Create the half moon data
X, y = make_moons(n_samples=500, noise=0.02)

# Standardize the data
scaler = StandardScaler()
X_std = scaler.fit_transform(X)

# Apply the RBF kernel PCA
kpca = KernelPCA(n_components=3, kernel='rbf', gamma=15)
X_kpca = kpca.fit_transform(X_std)

# Create a trace for the original data
trace1 = go.Scatter(x=X_std[y==0, 0], y=X_std[y==0, 1],
                    mode='markers', name='Class 0',
                    marker=dict(color='red', size=5, opacity=0.5))
trace2 = go.Scatter(x=X_std[y==1, 0], y=X_std[y==1, 1],
                    mode='markers', name='Class 1',
                    marker=dict(color='blue', size=5, opacity=0.5))

# Create a trace for the transformed data
trace3 = go.Scatter3d(x=X_kpca[y==0, 0], y=X_kpca[y==0, 1], z=X_kpca[y==0, 2],
                      mode='markers', name='Class 0',
                      marker=dict(color='red', size=5, opacity=0.5))
trace4 = go.Scatter3d(x=X_kpca[y==1, 0], y=X_kpca[y==1, 1], z=X_kpca[y==1, 2],
                      mode='markers', name='Class 1',
                      marker=dict(color='blue', size=5, opacity=0.5))

# Create the layouts
layout1 = go.Layout(title='Original data in 2D', autosize=True,
                    xaxis=dict(title='Feature 1'),
                    yaxis=dict(title='Feature 2'))
layout2 = go.Layout(title='Data after RBF Kernel PCA in 3D', autosize=True,
                    scene=dict(xaxis=dict(title='PC 1'),
                               yaxis=dict(title='PC 2'),
                               zaxis=dict(title='PC 3')))

# Create the figures and plot
fig1 = go.Figure(data=[trace1, trace2], layout=layout1)
fig2 = go.Figure(data=[trace3, trace4], layout=layout2)

fig1.show()
```


Original data in 2D
1.5
1
0.5
0
eature 2

In [37]:

```
fig2.show()
```



The KPCA algorithm can be summarized as follows:

1. **Compute the kernel matrix**: Given an input dataset with n data points, calculate an n x n kernel matrix K, where each element K(i, j) represents the kernel function applied to the i-th and j-th data points.

2. **Center the kernel matrix**: Center the kernel matrix K by subtracting the mean of each row and each column and adding the mean of the entire matrix.

3. **Perform eigen decomposition**: Compute the eigenvectors and eigenvalues of the centered kernel matrix K. The eigenvectors represent the principal components in the high-dimensional feature space.

4. **Select the principal components**: Choose the desired number of principal components to retain based on the explained variance or other criteria. These principal components can be used for further analysis or visualization.

5. **Project new data points**: To project new data points into the kernel PCA space, compute their inner product (similarity) with the eigenvectors obtained in step 3.

One advantage of KPCA is that it can capture complex nonlinear structures in the data without explicitly defining the transformation function. It allows for flexible modeling of data with intricate relationships and can be particularly useful when the underlying data distribution is nonlinear.

However, KPCA has some considerations to keep in mind. The choice of kernel function is crucial, as different kernels have different properties and capture different types of nonlinearities. Common kernel functions used in KPCA include Gaussian (RBF), polynomial, sigmoid, and Laplacian kernels. Additionally, KPCA is computationally more expensive than linear PCA, as it involves working with the kernel matrix and potentially high-dimensional feature spaces.

Overall, KPCA provides a powerful tool for nonlinear dimensionality reduction and extracting meaningful features from complex datasets. It has applications in various fields, such as computer vision, bioinformatics, and pattern recognition, where nonlinear relationships among data points need to be captured.

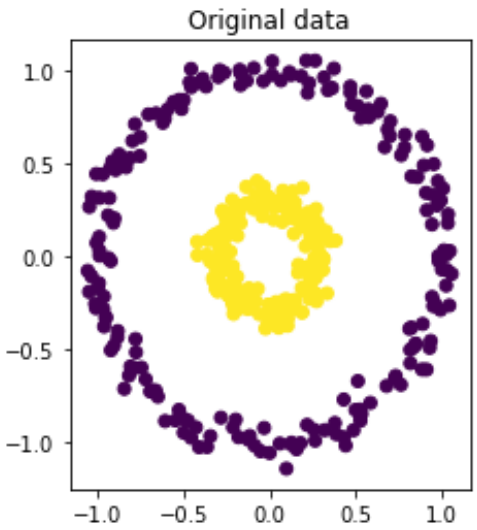
Kernel PCA step by step

In [22]:

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA, KernelPCA
from sklearn.datasets import make_circles
import numpy as np

# Generate the dataset
X, y = make_circles(n_samples=400, factor=.3, noise=.05)

# Original data plot
plt.figure(figsize=(16, 4))
plt.subplot(1, 4, 1)
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.title('Original data')
```

Out[22]:

```
Text(0.5, 1.0, 'Original data')
```



In [23]:

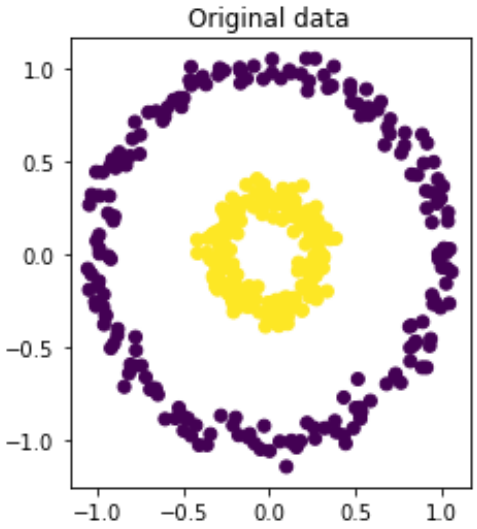
```
# Subtract the mean of X along axis 0 from X

X_centered = X - np.mean(X, axis=0)
```

In [24]:

```
plt.figure(figsize=(16, 4))
plt.subplot(1, 4, 1)
plt.scatter(X_centered[:, 0], X_centered[:, 1], c=y)
plt.title('Original data')
```

Out[24]:

```
Text(0.5, 1.0, 'Original data')
```



In [25]:

```
def rbf_kernel(x1, x2, gamma=0.1):
    distance = np.linalg.norm(x1 - x2) ** 2
    return np.exp(-gamma * distance)

# Create the kernel matrix
n_samples = X.shape[0]
K = np.zeros((n_samples, n_samples))
for i in range(n_samples):
    for j in range(n_samples):
        K[i, j] = rbf_kernel(X_centered[i], X_centered[j])
```

In [27]:

```
K.shape
```

Out[27]:

```
(400, 400)
```

In [28]:

```
from scipy.linalg import eigh
eigenvalues, eigenvectors = eigh(K)
```

In [29]:

```
K.shape
```

Out[29]:

```
(400, 400)
```

In [30]:

```
# Reverse the arrays as eigh returns them in ascending order
eigenvalues = eigenvalues[::-1]
eigenvectors = eigenvectors[:, ::-1]
```

In [31]:

```
eigenvalues.shape
```

Out[31]:

```
(400,)
```

In [32]:

```
eigenvectors.shape
```

Out[32]:

```
(400, 400)
```

In [33]:

```
k = 2
X_kpca = eigenvectors[:, :k]
```

In [34]:

```
X_kpca.shape
```

Out[34]:

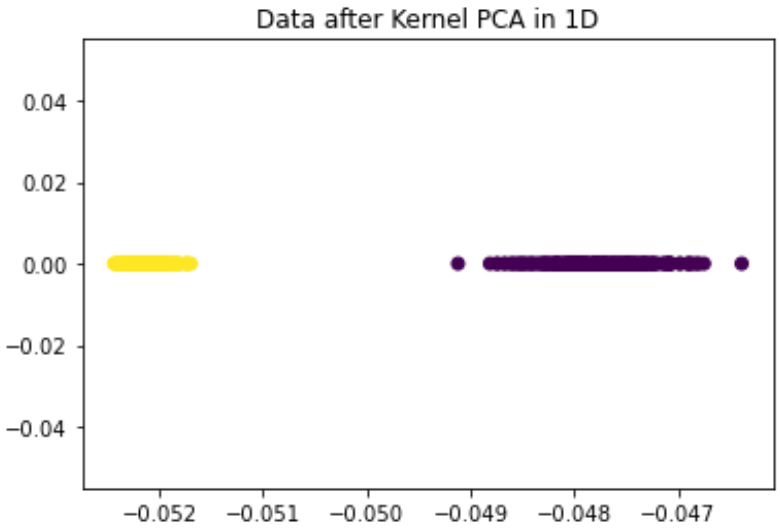
```
(400, 2)
```

In [35]:

```
plt.scatter(X_kpca[:, 0], np.zeros((400,)), c=y)
plt.title('Data after Kernel PCA in 1D')
```

Out[35]:

```
Text(0.5, 1.0, 'Data after Kernel PCA in 1D')
```



Explanation:

The code implementation of Kernel Principal Component Analysis (KPCA) using the scikit-learn library. Let's go through it step by step:

1. The code imports necessary libraries: matplotlib.pyplot for plotting, PCA and KernelPCA from sklearn.decomposition for PCA and KPCA respectively, make_circles from sklearn.datasets to generate a dataset with circles, and numpy as np for numerical operations.

2. The dataset is generated using the make_circles function, which creates a dataset with 400 samples, a factor of 0.3 for the size of the circles, and some noise.

3. The original data is plotted using plt.scatter to visualize the dataset.

4. The code calculates the centered dataset by subtracting the mean of x along axis 0 from x. This step is important for centering the data before applying KPCA.

5. The rbf_kernel function is defined, which computes the Radial Basis Function (RBF) kernel between two data points x1 and x2 using a specified gamma value. The RBF kernel is a commonly used kernel function in KPCA.

6. The code creates an empty kernel matrix K of size (n_samples, n_samples), where n_samples is the number of data points in the dataset. It then iterates over each pair of data points and calculates the RBF kernel value using the rbf_kernel function, storing the result in the corresponding element of the kernel matrix.

7. The shape of the kernel matrix K is printed to the standard output.

8. The code uses the eigh function from scipy.linalg to compute the eigenvalues and eigenvectors of the kernel matrix K. Note that eigh is used instead of eig because K is symmetric.

9. The shape of the kernel matrix K is printed again. This step is not necessary and seems to be redundant.
10. The eigenvalues and eigenvectors are reversed to be in descending order, as they are returned in ascending order by eigh.
11. The shapes of the eigenvalues and eigenvectors are printed to the standard output.
12. The code specifies the desired number of principal components k (in this case, 2) and selects the first k columns of the eigenvectors matrix, representing the principal components in the high-dimensional feature space.
13. The shape of the resulting X_kpca matrix is printed to the standard output.
14. Finally, the code plots the data after Kernel PCA in ID using plt.scatter. It visualizes the first column of x_kpca against an array of zeros with the same length as the number of samples, np. zeros((400,)). The color y is used to differentiate between the two classes in the dataset. The plot is titled "Data after Kernel PCA in ID".

reference:

https://www.youtube.com/@visualkernel4178 (https://www.youtube.com/@visualkernel4178)

In [ ]: