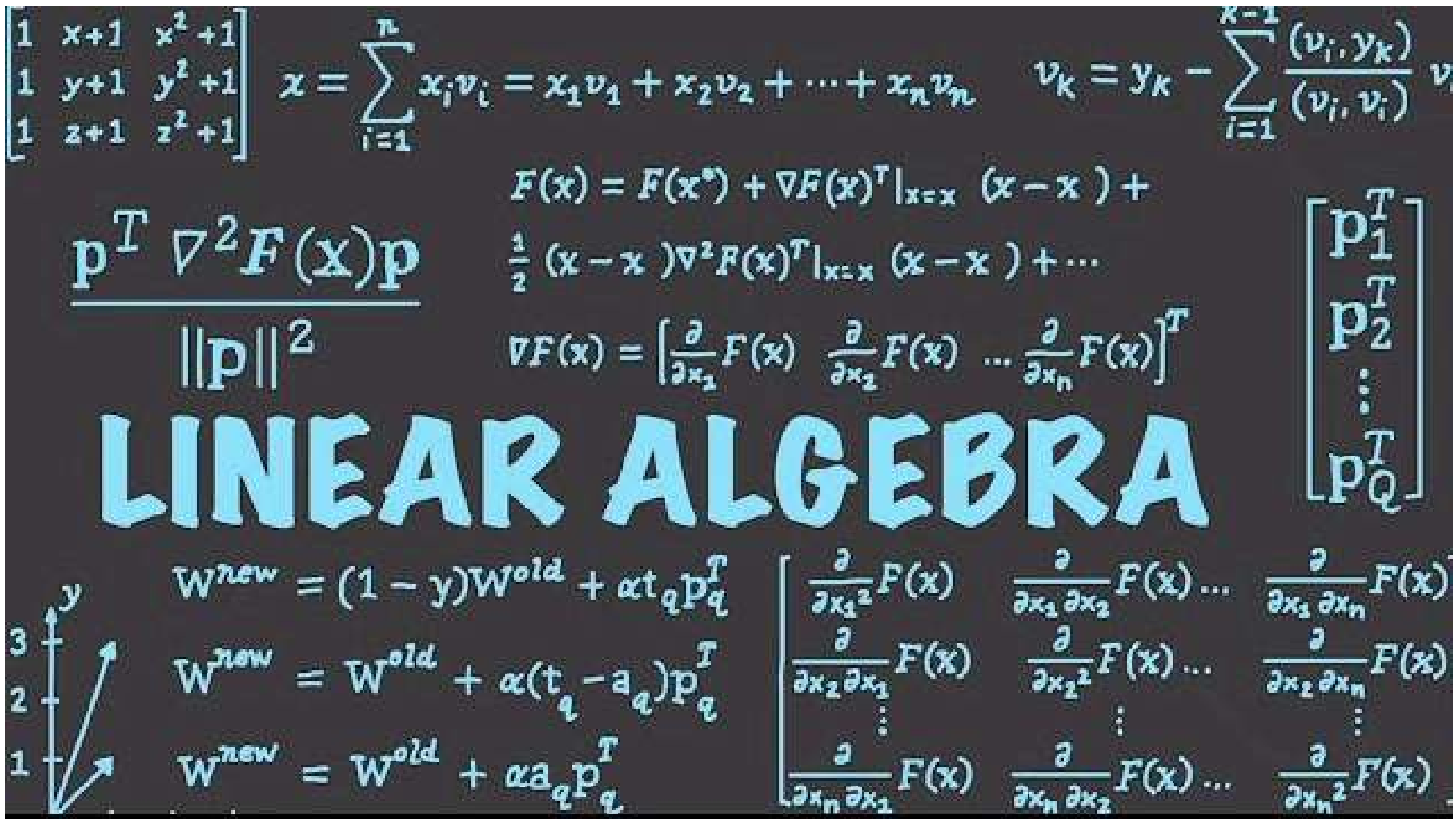
Mathematics Handbook for Machine Learning

Linear Algebra using Numpy & Matplotlib

Ashraful Islam Mahi

Full Stack Python Developer | ML & Robotics Enthusiast | Founder & CEO, PytronLab



Understanding the Role of Linear Algebra in Data Science & Machine Learning.

What is Linear Algebra?

- Linear Algebra is a branch of mathematics that studies vectors, matrices, and linear equations, forming the backbone for many areas of science and engineering.

Why use Linear Alegbra in Ai?

- Data Representation: Models data as vectors and matrices.
- Efficiency: Enables fast computations.
- Transformations: Facilitates data manipulation and feature transformation.
- Fundamental Algorithms: Underpins many AI and machine learning algorithms.

Understanding Vector & Matrices:

1. Definition:

- **Vector:** It is a 1D array, representing quantities with both magnititude & direction.
    - Example: [2, 3, 4]
- **Matrix:** It is a 2D array, arranged in rows & columns.
    - Example:

$$\begin{bmatrix} 2 & 2 & 3 \\ 3 & -2 & 3 \\ 1 & 6 & -3 \end{bmatrix}$$

2. Properties:

- Matrices are read out by m x n (m by n); where m = number of rows & n = number of columns.
- Vectors are special case of matrix with 1 row or 1 column.

Basic Matrix Operations:

In [5]:
```
#importing numpy libraries for matrice operations
import numpy as np
```

1. **Addition & Substraction:**

In [4]:
```
#Declearing matrices

A = np.array([[1,2],[3,4]])
B = np.array([[5,6],[7,8]])

print(f"Addition:\n{A+B}")
print(f"Substraction:\n{A-B}")
```

```
Addition:
[[ 6  8]
 [10 12]]
Substraction:
[[-4 -4]
 [-4 -4]]
```

2. **Scaler Multiplication:**

In [ ]:
```
A = np.array([[1,2],[3,4]])

#Declearing a scaler for multiplication
```

```
scaler = 5

print(f"Scaler Multiplication:\n{scaler*A}")
```

```
Scaler Multiplication:
[[ 5 10]
 [15 20]]
```

3. **Matrix Multiplication:**
    - Condition: Number of Row in 1st Matrix = Number of Column in 2nd MAtrix

In [7]:
```
A = np.array([[1,2],[3,4]])
B = np.array([[5,6],[7,8]])


print(f"Multiplication:\n{np.dot(A,B)}")
```

```
Multiplication:
[[19 22]
 [43 50]]
```

Special Matrices:

1. Identity Matrix(I): I.A = A
2. Zero Matrix(O): O.A = A
3. Diagonal Matrix (D): Every elements except the diagonal are zero.

1. Identity Matrix:

In [10]:
```
# Creating Identity Matrix
n = 4 #order
I = np.eye(n)
print(f"Identity Matrix of Order {n} X {n}:\n {I}")
```

```
Identity Matrix of Order 4 X 4:
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

2. Zero Matrix:

In [13]:
```
# Creating Zero Matrix
#order
m = 3
n = 4
Z = np.zeros((m,n))
print(f"Zero Matrix of Order {m} X {n}:\n {Z}")
```

```
Zero Matrix of Order 3 X 4:
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

3. Diagonal Matrix:

In [7]:
```
D = np.diag([1,2,3])
print(f"Diagonal Matrix:\n{D}")
```

```
Diagonal Matrix:
[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

Determinants & Inverse of a Matrix:

- **Determinants:**
  - Scaler value that provides information about a matrix's properties.
  - Only for square matrices.
  - det(A) = 0, the matrix A is singular.
  - det(A) = 0, A is invertible.
  - Geometric Interpretation:
    - For a 2x2 matrix, the determinant represents the scaling factor of the area formed by it's column vector.

In [11]:
```python
# Example 3x3 matrix
A = np.array([[1, 2, 3],
              [0, 1, 4],
              [5, 6, 0]])

# Determinant Calculation:
det_A = np.linalg.det(A)
print(f"Determinant: {det_A:.2f}")
```

```
Determinant: 1.00
```

- **Inverse of a Matrix:**
  - denoted as A^{-1}
  - Product of a matrix and its inverse is identity matrix: A x A^{-1} = I
  - Matrix is invertible only if det(A) != 0

In [12]:
```python
A = np.array([[1, 2, 3],
              [0, 1, 4],
              [5, 6, 0]])

# Inverse (only if the matrix is non-singular)
if det_A != 0:
    A_inverse = np.linalg.inv(A)
    print("Inverse:\n", A_inverse)
else:
    print("Matrix is singular; no inverse exists.")
```

```
Inverse:
 [[-24.  18.   5.]
 [ 20. -15.  -4.]
 [ -5.   4.   1.]]
```

Eigenvalues & Eigenvectors

What are *Eigenvalues* & *Eigenvectors*?

- Properties of square matrices & describe transformations.
- If A.v = λ.v, then

- v is an eigenvector
- λ is an eigenvalue

Geometric Interpretation:

- Eigenvectors point in the direction where the matrix transformation streches or compresses vectors.
- Eigenvalues indicate the factor of streching or compression.

Properties:

- Matrix of n x n has *n* eigenvalues & eigenvectors.
- Eigenvalues can be real or complex.
- For a symmetric matrix, eigenvalues are always real.

1. Calculate Eigene Values & Vectors:

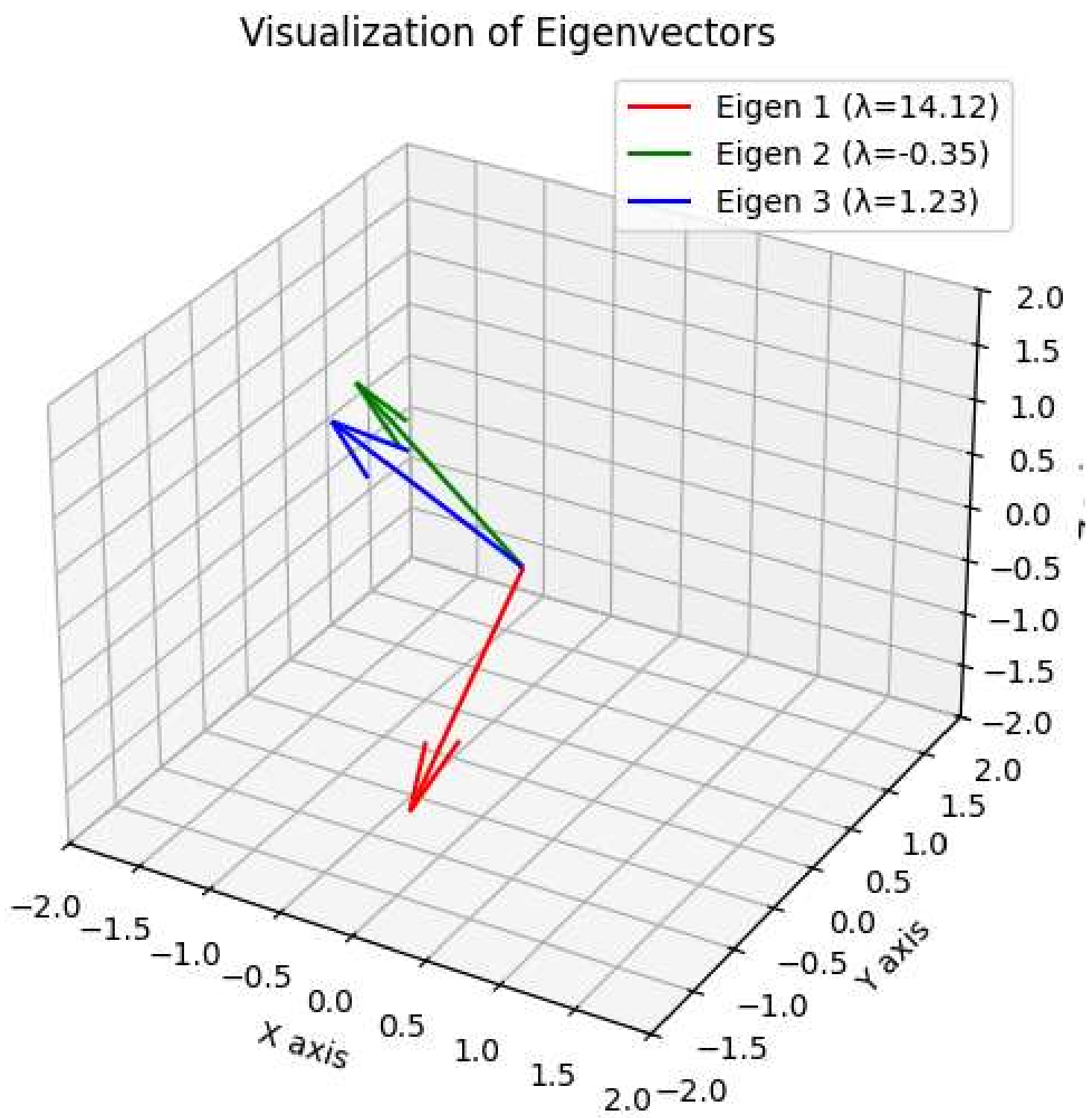
In [8]:
```python
import numpy as np
# Define a 3x3 matrix with nonzero eigenvalues.

A = np.array([[1, 2, 1],
              [4, 5, 4],
              [7, 8, 9]])

# Compute eigenvalues and eigenvectors
eigen_val, eigen_vec = np.linalg.eig(A)

print("Eigenvalues:")
print(eigen_val)
print("\nEigenvectors:")
print(eigen_vec)
```

```
Eigenvalues:
[14.12005078 -0.34647647  1.22642569]

Eigenvectors:
[[-0.13555758 -0.85625712 -0.38528935]
 [-0.4472136   0.4472136  -0.4472136 ]
 [-0.8840951   0.25850288  0.80718778]]
```

2. Visualize Eigene Values & Vectors:

In [9]:
```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define distinct colors for each eigenvector
colors = ['r', 'g', 'b']

# Create a 3D plot to visualize the eigenvectors
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

# Set the origin for our vectors
origin = [0, 0, 0]

# Plot each eigenvector with a different color
for i in range(len(eigen_val)):
    vec = eigen_vec[:, i]  # select the i-th eigenvector
    ax.quiver(origin[0], origin[1], origin[2],
```

```
                vec[0], vec[1], vec[2],
                length=2, normalize=True,
                color=colors[i % len(colors)],  # cycle colors if needed
                label=f'Eigen {i+1} (λ={eigen_val[i]:.2f})')

# Customize plot appearance
ax.set_xlim([-2, 2])
ax.set_ylim([-2, 2])
ax.set_zlim([-2, 2])
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
ax.set_title('Visualization of Eigenvectors')
ax.legend()

plt.show()
```



Introduction to Matrix Decomposition

What is Matrix Decomposition?

- Process of breaking a matrix into simpled componets to analyze or solve problems.

Singular Value Decomposition(SVD)

- SVD decomposes a matrix into three matrices: $A = U. \sum. V^T$
  - U: Left singular vectors(orthogonal matrix)
  - ∑: Diagonal Matrix of singualr values (non - negative)

- V^T : Right Singular Vectors(orthogonal matrix)

Applications of SVD:

1. Dimentonality reduction
2. Noise redcution in dataset
3. Image completation

1. Calculate SVD:

In [10]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define a 3x3 matrix with nonzero singular values.
# We use a symmetric tridiagonal matrix as an example.
A = np.array([[1, 2, 1],
              [4, 5, 4],
              [7, 8, 9]])

# Compute the SVD: A = U * Σ * V^T
U, s, Vt = np.linalg.svd(A)
V = Vt.T  # Right singular vectors (columns of V)

print("Singular values:")
print(s)
print("\nLeft singular vectors (U):")
print(U)
print("\nRight singular vectors (V):")
print(V)
```

```
Singular values:
[15.98232127  1.21222901  0.30968967]

Left singular vectors (U):
[[-0.14560165  0.59784838  0.788275  ]
 [-0.46959274  0.65953039 -0.58694321]
 [-0.87079436 -0.45562811  0.18471654]]

Right singular vectors (V):
[[-0.50803216  0.03841943 -0.86048084]
 [-0.60100919  0.69980491  0.38608425]
 [-0.61700185 -0.7133001   0.33243296]]
```

2. Visualize SVD:

In [11]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Create a 3D plot to visualize the SVD components
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

origin = [0, 0, 0]

# Colors for differentiation
colors_right = ['b', 'c', 'm']   # for right singular vectors (input)
colors_trans = ['r', 'orange', 'g']  # for transformed vectors = s_i * u_i

# Plot the right singular vectors (input basis)
```

```
for i in range(3):
    # right singular vector v_i (blue arrows)
    vec_v = V[:, i]
    ax.quiver(origin[0], origin[1], origin[2],
              vec_v[0], vec_v[1], vec_v[2],
              length=1, normalize=True,
              color=colors_right[i],
              label=f'v{i+1}')

    # Transform v_i: A * v_i, which equals s_i * u_i (red arrows)
    transformed = A @ vec_v  # alternatively: s[i] * U[:, i]
    ax.quiver(origin[0], origin[1], origin[2],
              transformed[0], transformed[1], transformed[2],
              length=1, normalize=True,
              color=colors_trans[i],
              label=f'scaled u{i+1} (s={s[i]:.2f})')

    # Optionally, you can also plot the left singular vector U[:, i] for confirmation
    # ax.quiver(origin[0], origin[1], origin[2],
    #           U[0, i], U[1, i], U[2, i],
    #           length=1, normalize=True,
    #           color='k', linestyle='dashed', label=f'u{i+1}')

# Customize plot appearance
ax.set_xlim([-1.5, 1.5])
ax.set_ylim([-1.5, 1.5])
ax.set_zlim([-1.5, 1.5])
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
ax.set_title('Visualization of the SVD of a 3x3 Matrix')
ax.legend()

plt.show()
```

Visualization of the SVD of a 3x3 Matrix