NANYANG
THE INNOVATIVE POLYTECHNIC

2024S2 - IT2313

Programming for Data Science

NumPy for Numerical Computations (Part 2)

Basics of Linear Algebra

Basics of Linear Algebra

In this module, we'll unravel the power of linear algebra in the realm of Python for Data Science, equipping you with essential tools to manipulate and understand data with precision and efficiency.

In linear algebra, data is mostly represented using **vectors** and **matrices**. Linear Algebra

Vector

A vector is a one-dimensional array that has only one row - called a row vector, or just one column, also called a column vector.

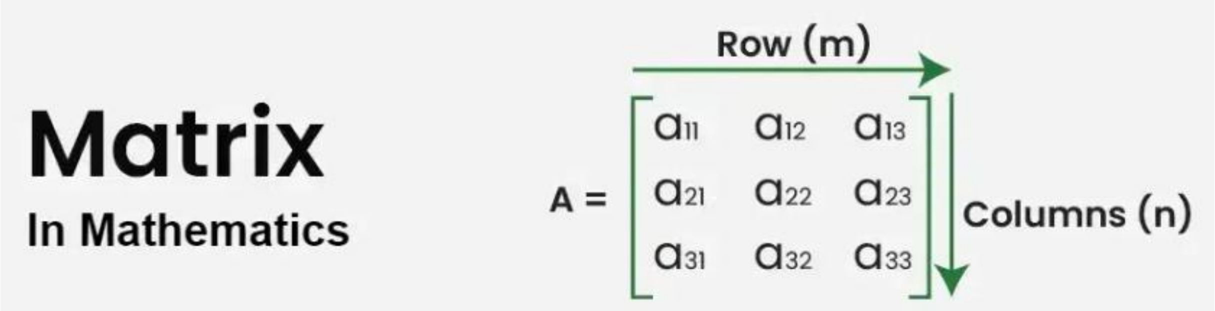
Matrix

Matrices (plural of matrix) are used throughout machine learning algorithms, specifically for input variables, i.e, the variables we try to understand in machine learning. An m×n matrix is a two-dimensional array that has m rows and n columns.

Matrices

A matrix is a rectangular array of elements (numbers, symbols, points, or characters) arranged in rows and columns, with its order defined by the number of rows and columns. Each element’s location within the matrix is identified by its specific row and column.

Matrices are essential tools in mathematics and data representation, widely used to organize data efficiently. They are integral in fields like computer graphics, machine learning, and scientific computations, where they enable the compact representation of complex data structures.



Applications of Linear Algebra in Machine Learning

Applications of Linear Algebra in Machine Learning (ML)

Linear Algebra Application in Linear Equation System

Solving the system of linear equations using matrices. Suppose that we have the system below, a typical way to compute the value of a and b is to eliminate one element at a time.

3a + 2b = 7

a - b = -1

An alternative solution is to represent it using the dot product between matrix and vector. We can package all the coefficients into a matrix and all the variable into a vector, hence we get following:

$$\begin{bmatrix} 3 & 2 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 7 \\ -1 \end{bmatrix}$$

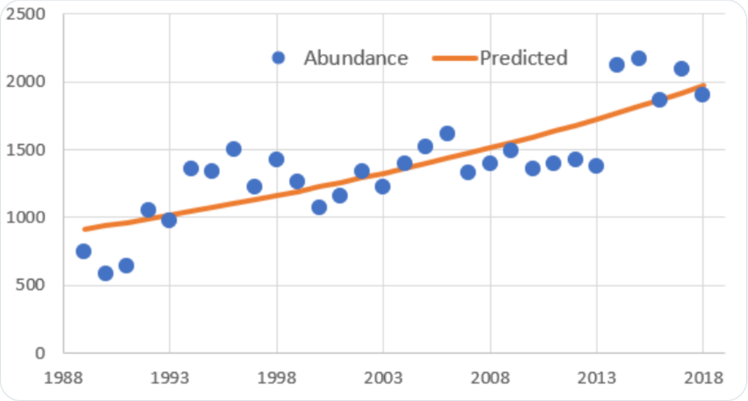
By representing the linear equation systems using matrices, this increase the computational speed of the process significantly. if we expand it to machine learning or even deep learning, it makes drastic increase in efficiency.

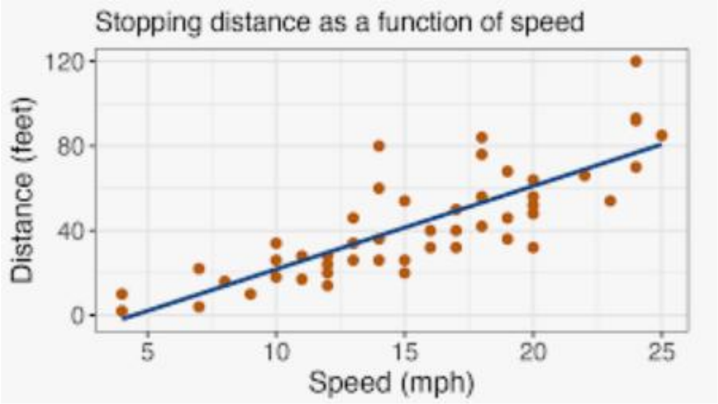
Applications of Linear Algebra in Machine Learning (ML)

Linear Algebra Application in Linear Regression

Linear regression is a typical regression algorithm which is responsible for numerous prediction. It is distinct to classification models - such as decision tree, support vector machine, neural network.

A linear regression finds the optimal linear relationship between independent variables and dependent variables, thus makes prediction accordingly.

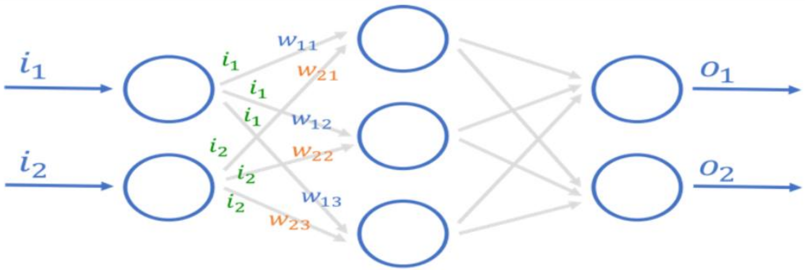




Applications of Linear Algebra in Machine Learning (ML)

Linear Algebra Application in Neural Network

Neural network is composed of multiple layers of interconnected nodes, where the outputs of nodes from the previous layers are weighted and then aggregated to form the input of the subsequent layers. If we zoom into the interconnected layers of a neural network, we can see some components of the regression model.



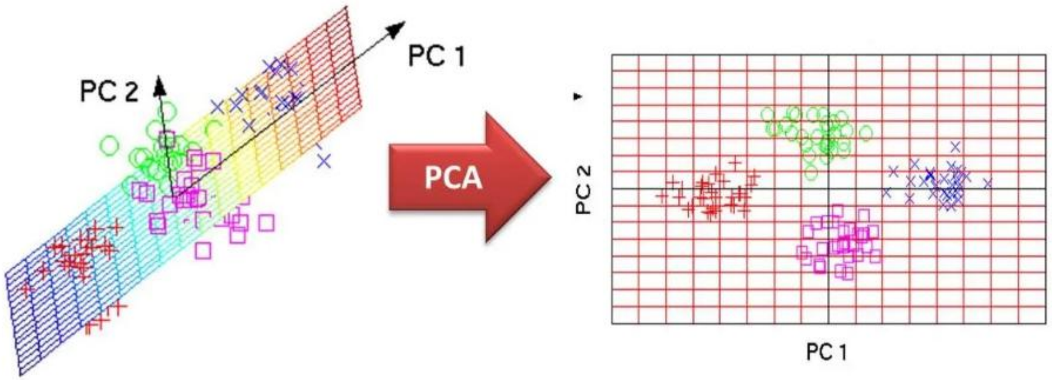
$$\begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} \cdot \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = \left.\begin{bmatrix} (w_{11} \times i_1) + (w_{21} \times i_2) \\ (w_{12} \times i_1) + (w_{22} \times i_2) \\ (w_{13} \times i_1) + (w_{23} \times i_2) \end{bmatrix}\right\} \boxed{\textbf{Matrices}}$$

Applications of Linear Algebra in Machine Learning (ML)

Dimensionality Reduction / Principal Component Analysis

Matrices are employed in feature extraction techniques such as Principal Component Analysis (PCA) and Singular Value Decomposition (SVD). [Matrices in Machine Learning]

These methods transform high-dimensional data into a lower-dimensional space using matrix operations, facilitating data compression and noise reduction. For this you should have knowledge of basis of vector.



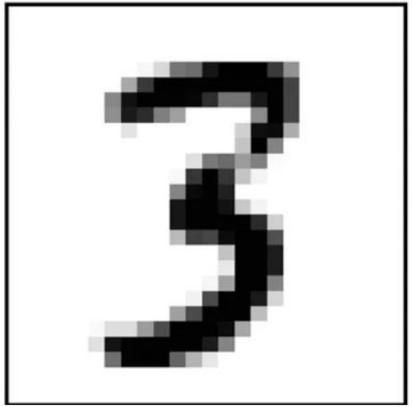
Applications of Linear Algebra in Machine Learning (ML)

Data and Image Representation:

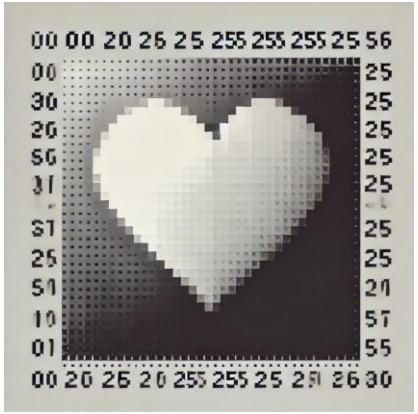
[Matrices in Machine Learning]

In machine learning, matrices are commonly used to represent datasets and images.

For datasets, rows represent individual samples, while columns represent features, enabling efficient data storage and processing. Images are also stored as matrices, with each element indicating pixel intensity. In grayscale images, pixel values range from 0 (black) to 255 (white), representing brightness levels.



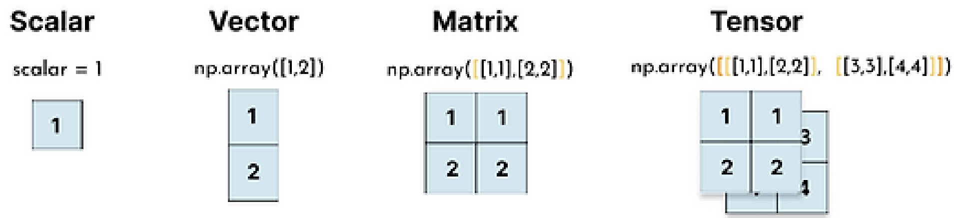
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	8	38	137	146	232	254	255	255	197	109	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	87	197	253	253	253	253	253	253	253	253	188	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	120	237	253	253	253	248	209	139	139	230	253	188	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	112	229	210	128	96	0	0	0	0	117	253	188	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	28	0	0	0	0	0	0	45	241	245	82	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	60	253	125	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	32	134	148	98	127	4	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	7	201	253	200	59	12	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	122	246	253	223	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	233	253	253	236	55	11	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	233	253	253	253	253	210	48	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	68	193	185	243	253	253	173	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	59	253	253	226	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	22	253	253	226	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	93	253	253	123	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	14	180	253	253	228	42	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	95	181	253	253	253	66	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	6	33	33	100	178	253	253	253	241	88	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	32	236	253	253	253	253	253	228	122	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	143	253	253	253	154	76	24	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



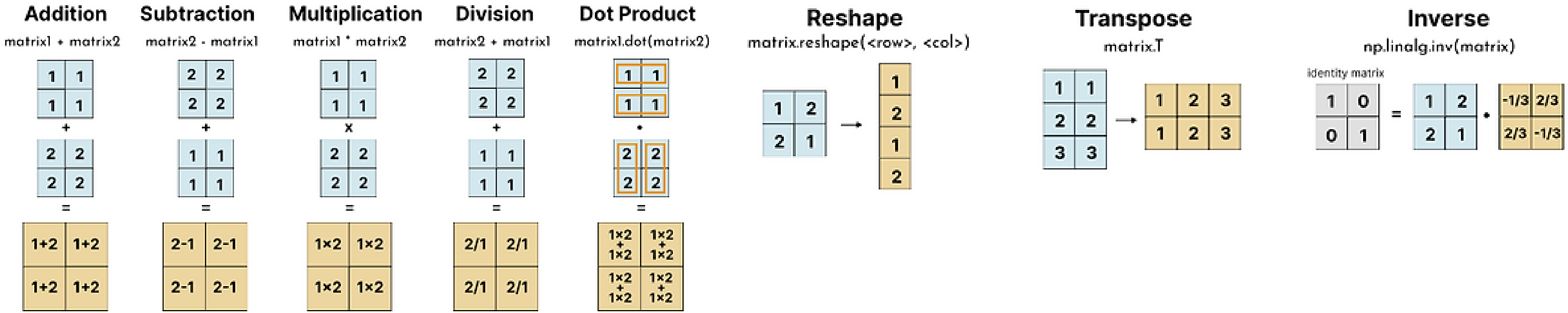
Matrices Operation

Matrix and Vector Operations

The following Matrix and Vector operations in Python will be covered in this course.



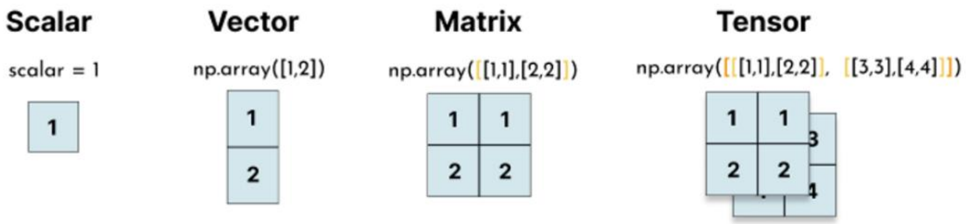
[Linear Algebra for Machine Learning]()



Definition of Scalar, Vector, Matrix and Tensor

Firstly, let's address the building blocks of linear algebra - scalar, vector, matrix and tensor.

- scalar: a single number
- vector: an one-dimensional array of numbers
- matrix: a two-dimensional array of numbers
- tensor: a multi-dimensional array of numbers



We can implement them using Numpy array np.array() in python

```python
# Implement Numpy array np.array() in python.
scalar = 1
vector = np.array([1,2])
matrix = np.array([[1,1],[2,2]])
tensor = np.array([[[1,1],[2,2]],
                   [[3,3],[4,4]]])
```

```python
# Shape of the vector, matrix and tensor generated.
print('vector shape:', vector.shape)
print('matrix shape:', matrix.shape)
print('tensor shape:', tensor.shape)
```

```
vector shape: (2,)
matrix shape: (2, 2)
tensor shape: (2, 2, 2)
```

Addition, Subtraction, Multiplication, Division

Similar to how we perform operations on numbers, the same logic also works for matrices and vectors.

- Note that these operations on matrix have restrictions on two matrices being the same size.
- This is because they are operated in an element-wise manner, which is different from matrix dot product.

```python
## Addition, Subtraction, Multiplication, Division
matrix1 = np.array([[1, 1],
                    [1, 1]])

matrix2 = np.array([[2, 2],
                    [2, 2]])

print('Matrix Addition\n', matrix1 + matrix2, '\n')
print('Matrix Subtraction\n', matrix1 - matrix2, '\n')
print('Matrix Multiplication\n', matrix1 * matrix2, '\n')
print('Matrix Division\n', matrix1 / matrix2, '\n')
```

```
Matrix Addition
 [[3 3]
 [3 3]]

Matrix Subtraction
 [[-1 -1]
 [-1 -1]]

Matrix Multiplication
 [[2 2]
 [2 2]]

Matrix Division
 [[0.5 0.5]
 [0.5 0.5]]
```

Dot Product

Dot product is often being confused with matrix element-wise multiplication (which is demonstrated earlier). However, in fact it is a more commonly used operations on matrices and vectors.

Dot product operates by multiplying each **row of the first matrix** to the **column of the second matrix**, therefore the dot product between a j x k matrix and k x i matrix is a j x i matrix.

When you multiply a matrix A of shape (m, n) by another matrix B of shape (n, p), the resulting matrix C will have shape (m, p). Let's consider a 3x2 matrix A and a 2x3 matrix B:

.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \bullet B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \Rightarrow C = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} & a_{21}b_{13} + a_{22}b_{23} \\ a_{31}b_{11} + a_{32}b_{21} & a_{31}b_{12} + a_{32}b_{22} & a_{31}b_{13} + a_{32}b_{23} \end{bmatrix}$$

Dot Product

```
# Define matrices A and B
matrix_A = np.array([[1, 2],
                     [3, 4],
                     [5, 6]])

matrix_B = np.array([[2, 0, 1],
                     [1, 3, 2]])

# Perform the dot product
matrix_C = np.dot(matrix_A, matrix_B)

# Alternatively, you can use the @ operator for matrix
# multiplication, matrix_C = matrix_A @ matrix_B

print("Matrix A:")
print(matrix_A)

print("\nMatrix B:")
print(matrix_B)

print("\nResult of Matrix (Dot Product):")
print(matrix_C)
```

```
Matrix A:
[[1 2]
 [3 4]
 [5 6]]

Matrix B:
[[2 0 1]
 [1 3 2]]

Result of Matrix (Dot Product):
[[ 4  6  5]
 [10 12 11]
 [16 18 17]]
```

$$\begin{bmatrix} (1\cdot 2+2\cdot 1) & (1\cdot 0+2\cdot 3) & (1\cdot 1+2\cdot 2) \\ (3\cdot 2+4\cdot 1) & (3\cdot 0+4\cdot 3) & (3\cdot 1+4\cdot 2) \\ (5\cdot 2+6\cdot 1) & (5\cdot 0+6\cdot 3) & (5\cdot 1+6\cdot 2) \end{bmatrix}$$

Reshape

A vector is often seen as a matrix with one column and it can be reshaped into matrix by specifying the number of columns and rows using reshape(). We can also reshape the matrix into a different layout. For example, we can use the code below to transform the 2x2 matrix to 4 rows and 1 column.

```
# Creating a 2x2 matrix
matrix_2x2 = np.array([[1, 2],
                       [3, 4]])

# Reshaping to a 4x1 matrix
reshaped_4x1 = matrix_2x2.reshape(4, 1)

# Reshaping to a 1x4 matrix
reshaped_1x4 = matrix_2x2.reshape(1, 4)

print("Original 2x2 matrix:")
print(matrix_2x2)

print("\nReshaped 4x1 matrix:")
print(reshaped_4x1)

print("\nReshaped 1x4 matrix:")
print(reshaped_1x4)
```

```
Original 2x2 matrix:
[[1 2]
 [3 4]]

Reshaped 4x1 matrix:
[[1]
 [2]
 [3]
 [4]]

Reshaped 1x4 matrix:
[[1 2 3 4]]
```

When the size of the matrix is unknown, reshape(-1) is also commonly used to reduce the matrix dimension and “flatten” the array into one row.

Reshaping matrices can be widely applied in neural network in order to fit the data into the neural network architecture.

Transpose

Let's define two matrices, A and B, and then illustrate the transpose operation on them. Matrix transpose is an operation that flips the rows and columns of a matrix, effectively transforming its columns into rows and vice versa

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & 0 & 1 \\ 1 & 3 & 2 \end{bmatrix} \quad B^T = \begin{bmatrix} 2 & 1 \\ 0 & 3 \\ 1 & 2 \end{bmatrix}$$

```python
# Creating a 3x2 matrix
matrix_3x2 = np.array([[1, 2],
                       [3, 4],
                       [5, 6]])

# Transposing the matrix using transpose()
transposed_matrix = np.transpose(matrix_3x2)

# Alternatively, you can use .T for transpose
# transposed_matrix = matrix_3x2.T

print("Original 3x2 matrix:")
print(matrix_3x2)

print("\nTransposed matrix:")
print(transposed_matrix)
```

```
Original 3x2 matrix:
[[1 2]
 [3 4]
 [5 6]]

Transposed matrix:
[[1 3 5]
 [2 4 6]]
```

Identity Matrix

The identity matrix, often denoted as I_n , is a special ***square*** matrix that has ones on its main diagonal and zeros elsewhere. The identity matrix is multiplicative identity for matrices, meaning that when any matrix ***A*** A is multiplied by the identity matrix, the result is the original matrix ***A***.

This concept holds for identify matrices of any size. If I_n is an *n x m* identify matrix, and A is an *n x m* or *m x n* matrix.

$$A \cdot I_n = I_n \cdot A = A$$

```
# Define a 2x2 identity matrix
identity_matrix = np.eye(2)

# Define matrix B
matrix_B = np.array([[2, 3],
                     [4, 5]])

# Multiply B by the identity matrix
result = np.dot(matrix_B, identity_matrix)

print("Matrix B:")
print(matrix_B)

print("\nIdentity Matrix I2:")
print(identity_matrix)

print("\nResult of Matrix Multiplication (B * I2):")
print(result)
```

```
Matrix B:
[[2 3]
 [4 5]]

Identity Matrix I2:
[[1. 0.]
 [0. 1.]]

Result of Matrix Multiplication (B * I2):
[[2. 3.]
 [4. 5.]]
```

$$I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$I_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Inverse Matrix

The inverse of a square matrix ***A***, denoted as ***A*** $^{-1}$, is a matrix such that when it is multiplied by ***A***, the result is the Identify matrix ***I***, In mathematical terms: $A \cdot A^{-1} = A^{-1} \cdot A = I$

Not all matrices have inverse, and a matrix must be square to have an inverse.

Let's consider a 2x2 matrix ***A***, $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ the inverse of ***A***, if it exists is given by $A^{-1} = \frac{1}{ad-bc} \cdot \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$

```
# Define matrix B
matrix_B = np.array([[2, 3],
                     [4, 5]])

# Calculate the inverse of B
inverse_B = np.linalg.inv(matrix_B)

print("Matrix B:")
print(matrix_B)

print("\nInverse of Matrix B:")
print(inverse_B)
```

```
Matrix B:
[[2 3]
 [4 5]]

Inverse of Matrix B:
[[-2.5  1.5]
 [ 2.  -1. ]]
```

Determinant of Matrix

The determinant of a matrix is a single numerical value which is used when calculating the inverse or when solving systems of linear equations. The determinant of a matrix A is denoted |A| , or sometimes det(A) . The determinant is only defined for square matrices. A matrix is said to be singular if its determinant is zero.

Determinant of a 2×2 matrix

$$\text{Let } \mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \Rightarrow |\mathbf{A}| = ad - bc$$

```
# Define a matrix C
matrix_C = np.array([[6, 8],
                     [3, 9]])

# Calculate the determinant of C
determinant = np.linalg.det(matrix_C)
print(determinant)
```

```
29.999999999999
```

Determinant of Matrix

Determinant of a 3×3 matrix

The determinant of a 3×3 matrix can be calculated by breaking it down into smaller 2×2 matrices, as follows

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} \Rightarrow a\begin{vmatrix} e & f \\ h & i \end{vmatrix} - b\begin{vmatrix} d & f \\ g & i \end{vmatrix} + c\begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \Rightarrow a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12}\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13}\begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

```python
# Define a matrix C
matrix_C = np.array([[1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9]])

# Calculate the determinant of C
determinant = np.linalg.det(matrix_C)
print(determinant)
```

```
-9.51619735392994e-16
```

Rank of Matrix

Rank of Matrix

The rank of a matrix represents the maximum number of linearly independent rows or columns in the matrix.

In simpler terms, it measures the dimension of the vector space spanned by the rows or columns of the matrix.

Let's consider a matrix ***A***, with dimensions ***m*** x ***n***,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

The rank of ***A***, denoted as ***rank (A)***, is the maximum number of linearly independent rows or columns in the matrix.

```python
# Define a matrix D
matrix_D = np.array([[1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9]])

# Calculate the rank of D
rank_D = np.linalg.matrix_rank(matrix_D)

print("Matrix D:")
print(matrix_D)

print("\nRank of Matrix D:")
print(rank_D)
```

```
Matrix D:
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Rank of Matrix D:
2
```

The matrix D has only 2 linearly independent rows/columns, making it rank 2 instead of 3 (full rank).

Rank of Matrix

Key Points

- A matrix's rank reveals insights into **solvability** in linear equations.
- The rank determines if a matrix is **full rank** (rank equals the smallest dimension) or **rank-deficient** (has dependencies among rows/columns).

Rules in Matrix Rank

- Rank ≤ Min (Number of Rows, Number of Columns)
  The rank can't exceed the matrix's smallest dimension.
- Rank of Zero Matrix = 0
  A zero matrix (all entries are zero) has a rank of 0.
- Full Rank for Square Matrices
  A square matrix with linearly independent rows and columns has full rank equal to its dimension.
- Rank Consistency
  Row rank and column rank are always equal.

Rank of Matrix

Full Rank Matrix

- A matrix is full rank if its rank equals the smallest of its dimensions (number of rows or columns).
- Implication: In a full-rank matrix, all rows and columns are linearly independent, meaning there are no redundant rows or columns.

$$E = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Rank (E) $\Rightarrow$ **2**

The rank is 2, which equals the smallest dimension of Matrix E (2x2 matrix), so it is full rank.

Rank-Deficient Matrix

- A matrix is rank-deficient if its rank is less than the smallest dimension, indicating dependencies among rows or columns.
- Implication: Rank deficiency means some rows or columns can be represented as combinations of others, which reduces the matrix's "information content."

$$F = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$$

Rank (F) $\Rightarrow$ **1**

The second row is a multiple of the first row, so the rank is 1 (less than 2), making Matrix F rank-deficient.

Rank of Matrix

Applications in AI and Machine Learning

- Crucial for dimensionality reduction techniques like **PCA (Principal Component Analysis).**
- Optimizes data processing by identifying and reducing redundant information.
- Full-rank matrices allow for better data representation and more robust calculations.
- Rank-deficient matrices indicate redundancy and potential for dimensionality reduction to save computation.

Calculating Matrix Rank in Python

```python
# Define the matrix
H = np.array([[1, 0, 2],
              [0, 1, 3],
              [4, 5, 6]])

# Calculate rank
rank = np.linalg.matrix_rank(H)
print("Rank of the matrix:", rank)
```

```
Rank of the matrix: 3
```

Aggregating Methods

Aggregating Methods

Summation, Maximum and Minimum

```
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# 1. Summation
sum_result = np.sum(matrix)
print("Sum:", sum_result)

# 2. Maximum and Minimum
max_value = np.max(matrix)
min_value = np.min(matrix)
print("Maximum:", max_value)
print("Minimum:", min_value)

# 3. Row and Column-wise Aggregation
column_sum = np.sum(matrix, axis=0)
row_sum = np.sum(matrix, axis=1)

print("Column-wise Sum:", column_sum)
print("Row-wise Sum:", row_sum)
```

```
Sum: 45
Maximum: 9
Minimum: 1
Column-wise Sum: [12 15 18]
Row-wise Sum: [ 6 15 24]
```

Aggregating Methods

Product, Count, Mean, Median, Standard Deviation and Variance

```
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
# 4. Product
product_result = np.prod(matrix)
print("Product:", product_result)

# 5. Count
count_nonzero = np.count_nonzero(matrix)
print("Number of Non-Zero Elements:", count_nonzero)

# 6. Mean (Average)
mean_result = np.mean(matrix)
print("Mean:", mean_result)

# 7. Median
median_result = np.median(matrix)
print("Median:", median_result)

# 8. Standard Deviation and Variance
std_deviation = np.std(matrix)
variance = np.var(matrix)
print("Standard Deviation:", std_deviation)
print("Variance:", variance)
```

```
Sum: 45
Maximum: 9
Minimum: 1
Column-wise Sum: [12 15 18]
Row-wise Sum: [ 6 15 24]
```

Aggregating Methods

Range, 25th and 75th Percentiles

```
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# 9. Range (maximum - minimum)
data_range = np.ptp(matrix)
print("Range:", data_range)

# 10 Calculate the 25th and 75th percentiles (Q1 and Q3)
q1 = np.percentile(matrix, 25)
q3 = np.percentile(matrix, 75)
print("Q1:", q1)
print("Q3:", q3)
```

```
Range: 8
Q1: 3.0
Q3: 7.0
```

Percentiles are a way of understanding the distribution of values in a dataset.

25th percentile, also known as the first quartile, is the value below which 25% of the data falls. So, it's like the border between the lowest 25% and the rest.

75th percentile, or the third quartile, is the value below which 75% of the data falls. It marks the boundary between the lowest 75% and the top 25%.

Aggregating Methods

Square Root, Square, Exponential and Power

```python
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# 11. Square Root
sqrt_result = np.sqrt(matrix)
print("Square Root:\n", sqrt_result)

# 12. Square
square_result = np.square(matrix)
print("Square:\n", square_result)

# 13. Exponential (e^x)
exp_result = np.exp(matrix)
print("Exponential:\n", exp_result)

# 14. Power
power_result = np.power(matrix, 2)  # Squaring each element
print("Power (Squared):\n", power_result)
```

```
Square Root:
 [[1.         1.41421356 1.73205081]
 [2.         2.23606798 2.44948974]
 [2.64575131 2.82842712 3.        ]]
Square:
 [[ 1  4  9]
 [16 25 36]
 [49 64 81]]
Exponential:
 [[2.71828183e+00 7.38905610e+00 2.00855369e+01]
 [5.45981500e+01 1.48413159e+02 4.03428793e+02]
 [1.09663316e+03 2.98095799e+03 8.10308393e+03]]
Power (Squared):
 [[ 1  4  9]
 [16 25 36]
 [49 64 81]]
```

Aggregating Methods

Natural Logarithm (base e) and Base 10 Logarithm

```python
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# 15. Natural Logarithm (log base e)
log_result = np.log(matrix)
print("Natural Logarithm (base e):\n", log_result)

# 16. Base 10 Logarithm
log10_result = np.log10(matrix)
print("Base 10 Logarithm:\n", log10_result)
```

```
Natural Logarithm (base e):
 [[0.         0.69314718 1.09861229]
 [1.38629436 1.60943791 1.79175947]
 [1.94591015 2.07944154 2.19722458]]
Base 10 Logarithm:
 [[0.         0.30103    0.47712125]
 [0.60205999 0.69897    0.77815125]
 [0.84509804 0.90308999 0.95424251]]
```

Matrix Broadcasting

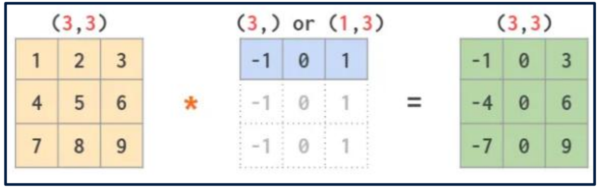
Matrix Broadcasting

Broadcasting is an operation of matching the dimensions of differently shaped arrays in order to be able to perform further operations on those arrays. The simplest example of broadcasting is the multiplication of an n-dimensional array by a scalar, which evidently means per-element multiplication of its elements by the scalar value:

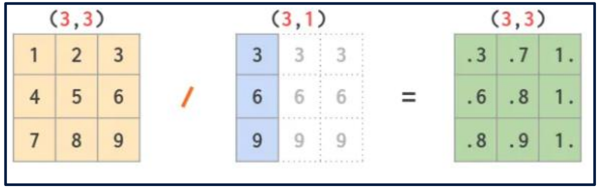
```
np.array([[1, 2],
          [3, 4]]) * 2
```

```
array([[2, 4],
       [6, 8]])
```

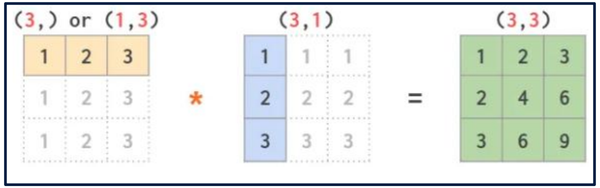
In 2D (eg dividing a matrix by a vector) the broadcasting is somewhat trickier since the result of the operation depends on the particular shapes of the operands:



Multiplying several columns at once



Row-wise normalization

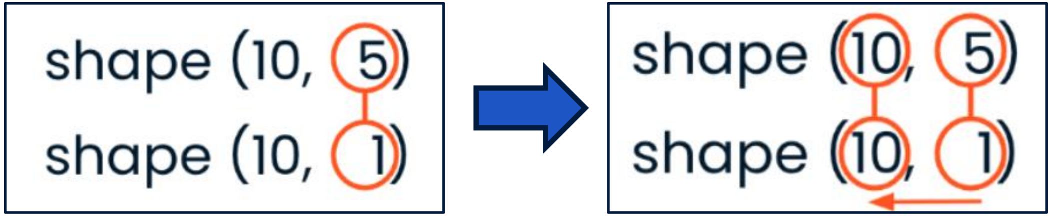


Outer Product

Matrix Broadcasting

The operations on arrays of non-similar shapes is still possible in NumPy, because of the broadcasting capability. The smaller array is broadcast to the size of the larger array so that they have compatible shapes.

NumPy compares sets of array dimensions from right to left.



Two dimensions are compatible when:

- One of them has a length of one or
- They are of equal lengths

All dimension sets must be compatible.

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions and works its way forward.

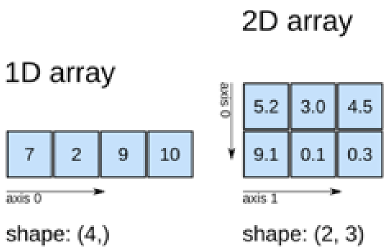
Matrix Broadcasting

Broadcastable shapes:

- `(10, 5)` and `(10, 1)`
- `(10, 5)` and `(5, )`

Shapes which are not broadcastable:

- `(10, 5)` and `(5, 10)`
- `(10, 5)` and `(10, )`



An array with 10 rows and 5 columns is broadcastable with array with 10 rows and 1 column since one of the trailing dimension lengths is **1**, and the length of the first dimensions are **equal**.

A 10 rows and 5 columns is also broadcastable with a five-element 1D array since the **right most dimensions** are both **5**.

Two arrays need not have the same number of dimensions to be broadcastable. A 10 by 5 array is not broadcastable with an array with 5 rows and 10 columns since **neither set of dimensions is compatible**. Finally, a 10 by 5 array is not compatible with a 10-element 1D array since the **right-most set of dimensions** is not compatible.

Matrix Broadcasting

In this example, B is broadcasted across the rows of A, and element-wise addition is performed. The output will be:

```python
# Example arrays
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([10, 20, 30])

# Broadcasting
result = A + B

# Display the result
print(result)
```

```
[[11 22 33]
 [14 25 36]]
```

This is a simple example, but broadcasting becomes very handy when dealing with more complex operations involving arrays of different shapes.

Matrix Broadcasting

NumPy's broadcasting is a powerful feature that allows you to perform operations on arrays of different shapes and sizes. Here's a simple example to illustrate broadcasting with NumPy. We create a 3x3 matrix and then add the scalar value 10 to every element of the matrix. NumPy automatically broadcasts the scalar to the shape of the matrix, making the operation straightforward.

```python
# Creating a 3x3 matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Adding a constant to each element using broadcasting
result = matrix + 10

print("Original Matrix:")
print(matrix)
print("\nResult after Broadcasting:")
print(result)
```

```
Original Matrix:
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Result after Broadcasting:
[[11 12 13]
 [14 15 16]
 [17 18 19]]
```

Matrix Broadcasting

In this example, the row vector is added to each row of the matrix. NumPy automatically broadcasts the row vector to match the shape of the matrix, allowing the addition operation to be performed element-wise.

```python
# Creating a 3x3 matrix
matrix_a = np.array([[1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9]])

# Creating a 1x3 array (row vector)
row_vector = np.array([10, 20, 30])

# Performing broadcasting to add the row vector to
# each row of matrix_a
result = matrix_a + row_vector

print("Matrix A:")
print(matrix_a)
print("\nRow Vector:")
print(row_vector)
print("\nResult after Broadcasting:")
print(result)
```

```
Matrix A:
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Row Vector:
[10 20 30]

Result after Broadcasting:
[[11 22 33]
 [14 25 36]
 [17 28 39]]
```

Matrix Broadcasting

Let's consider an example where we multiply a matrix by a column vector using broadcasting:

```
# Creating a 3x3 matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Creating a column vector
column_vector = np.array([[10],
                          [20],
                          [30]])

# Performing broadcasting to multiply the matrix by
# the column vector
result = matrix * column_vector

print("Matrix:")
print(matrix)
print("\nColumn Vector:")
print(column_vector)
print("\nResult after Broadcasting:")
print(result)
```

```
Matrix:
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Column Vector:
[[10]
 [20]
 [30]]

Result after Broadcasting:
[[ 10  20  30]
 [ 80 100 120]
 [210 240 270]]
```

Matrix Broadcasting

Let's consider an example where broadcasting would lead to an error. We'll try to add a 2x3 matrix to a 3x2 matrix. According to broadcasting rules, these shapes are not compatible for broadcasting.

```python
# Example matrices
matrix1 = np.array([[1, 2, 3],
                    [4, 5, 6]])

matrix2 = np.array([[10, 20],
                    [30, 40],
                    [50, 60]])

# Attempt to perform broadcasting addition
result = matrix1 + matrix2
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[19], line 10
      5 matrix2 = np.array([[10, 20],
      6                     [30, 40],
      7                     [50, 60]])
      9 # Attempt to perform broadcasting addition
---> 10 result = matrix1 + matrix2

ValueError: operands could not be broadcast together with shapes (2,3) (3,2)
```

Let's examine the shapes of these matrices:

matrix1 has a shape of (2, 3), and matrix2 has a shape of (3, 2)

According to the broadcasting rules, for two dimensions to be compatible, they must either be equal or one of them must be 1. In this case, neither dimension in matrix1 nor matrix2 is 1, and they are not equal.

Therefore, broadcasting cannot be applied, and it would result in a ValueError.

Matrix Broadcasting - Example

Let’s consider the following 2 matrices:

```
A = [[1, 2, 3],
     [4, 5, 6]]

B = [10, 20, 30]
```

1. Shape Compatibility: Explain why it's possible to perform element-wise addition between A and B even though they have different shapes.
2. Broadcasting Rules: Describe the specific broadcasting rules that allow NumPy to perform this operation.
3. Resultant Array: What is the shape of the resulting array after performing A + B?
4. Code Implementation: Write a Python code snippet to perform the addition and print the resulting array.

Matrix Broadcasting - Example

```
A = [[1, 2, 3],
     [4, 5, 6]]

B = [10, 20, 30]
```

1. **Shape Compatibility:**

The array A has a shape of (2, 3), which means it has 2 rows and 3 columns, while the array B has a shape of (3,), which is a one-dimensional array with 3 elements. Broadcasting allows NumPy to treat B as if it were a two-dimensional array with the same number of rows as A, essentially "stretching" it to match the shape of A during the operation.

2. **Broadcasting Rules:**

If the arrays have different numbers of dimensions, the shape of the smaller-dimensional array is padded with ones on the left side until both shapes are the same length. Here, B (shape (3,)) is treated as having a shape of (1, 3) for the purpose of broadcasting.

After padding,

A: (2, 3) and B: (1, 3) → stretches to (2, 3)

Matrix Broadcasting - Example

```
A = [[1, 2, 3],
     [4, 5, 6]]

B = [10, 20, 30]
```

3. Resultant Array:

The resulting array after performing the operation A + B will have the same shape as A, which is (2, 3).

4. Code Implementation:

The Python code as shown:

```python
import numpy as np

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

B = np.array([10, 20, 30])

# Perform element-wise addition
result = A + B

# Print the resulting array
print("Resulting Array:")
print(result)
```

```
Resulting Array:
[[11 22 33]
 [14 25 36]]
```

Eigenvalues and Eigenvectors

Eigenvalues and Eigenvectors

Imagine a transformation that stretches or compresses a vector but does not change its direction. The scaling factor is the eigenvalue, and the vector itself is the eigenvector.

- **Eigenvalues** are special scalars associated with a matrix that, when multiplied by their respective eigenvectors, yield the same direction as the eigenvector.
- **Eigenvectors** are non-zero vectors that only change in scale, not direction, when a linear transformation (like a matrix) is applied.

For a square matrix A and vector v,

$$Av = \lambda v$$

λ is the eigenvalue, and v is the eigenvector.

Eigenvectors are linearly independent.

Eigenvalues may be real or complex.

If A is an $n \times n$ matrix, it can have up to n eigenvalues and eigenvectors.

Eigenvalues and Eigenvectors

Importance in AI and Machine Learning

- Used in techniques like Principal Component Analysis (PCA) for dimensionality reduction.
- Help identify dominant patterns in datasets.

Applications

- Data Analysis: Capturing principal components in PCA.
- Systems Stability: Assessing stability in dynamical systems.
- Image Processing: Identifying important features or patterns.

Eigenvalues and Eigenvectors

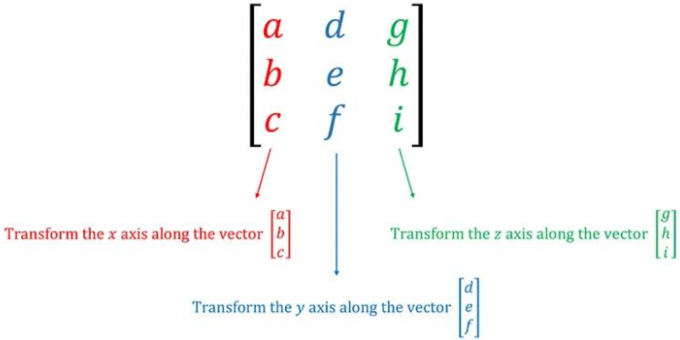
Calculation in Python (3x3 Matrix)

```python
# Define the matrix
J = np.array([[4, 1, 2],
              [1, 3, 0],
              [2, 0, 5]])

# Calculate eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(J)

print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

```
Eigenvalues: [6.66907909 1.85489731 3.4760236 ]
Eigenvectors:
 [[-0.63117897 -0.67931306 -0.37436195]
 [-0.17202654  0.59323331 -0.7864357 ]
 [-0.75632002  0.43198148  0.49129626]]
```



Eigenvalues Interpretation

6.669: Largest eigenvalue; indicates the direction of maximum variance in the data.

3.476: Moderate variance; less influential than the first but still significant.

1.855: Smallest eigenvalue; indicates minimal variance, suggesting this direction has less impact.

Eigenvalues and Eigenvectors

Eigenvectors Interpretation

For 6.669: [−0.631,−0.172,−0.756] captures the most variance; indicates key data relationships.

For 3.476: [−0.679,0.593,0.432] represents a secondary important direction.

For 1.855: [−0.374,−0.786,0.491] indicates less significance in variance.

Practical Implications

<u>Dimensionality Reduction:</u> Retaining eigenvectors of larger eigenvalues (like the first two) helps reduce data complexity while maintaining key information.

<u>Insights:</u> Understanding these vectors aids in identifying trends and relationships in data.

Eigenvalues show the importance of eigenvectors in variance capture, while eigenvectors define the key directions in the data. This is valuable for data analysis and machine learning.

Eigenvalues and Eigenvectors

For information Only

Determine Eigenvalues:

Find the eigenvalues by solving the characteristic polynomial:

$$\det(A - \lambda I) = 0$$

where I is the identity matrix of the same dimension as A.

Determine Eigenvectors:

For each eigenvalue λ, solve:

$$(A - \lambda I)v = 0$$

to find the corresponding eigenvectors v.

Verify:

Substitute the eigenvalues and eigenvectors back into the equation $Av = \lambda v$ to confirm.

Eigenvalues and Eigenvectors

For information Only

$$\det(A - \lambda I) = 0$$

$$(A - \lambda I)x = 0$$

Finding the eigenvalue is then a task of solving a quadratic. For 3+ dimension matrices, a different form of the determinant formula must be used.

$$A = \begin{bmatrix} 1 & 4 \\ 3 & 2 \end{bmatrix}$$

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$$

$$\det(A - \lambda I) = 0$$

$$\det\left(\begin{bmatrix} 1 & 4 \\ 3 & 2 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = 0$$

$$\det\left(\begin{bmatrix} 1-\lambda & 4 \\ 3 & 2-\lambda \end{bmatrix}\right) = 0$$

$$(1-\lambda)(2-\lambda) - 12 = 0$$

$$\lambda^2 - 3\lambda - 10 = 0$$

$$(\lambda - 5)(\lambda + 2) = 0$$

$$\lambda = 5, -2$$

Matrix we are finding the eigenvector/eigenvalue of

eigenvalue

$$Ax = \lambda x$$

$$(A - \lambda I)x = 0$$

identity matrix

In this case, the eigenvalues of the matrix `[[1, 4], [3, 2]]` are 5 and -2. This means that when the eigenvectors of the matrix are multiplied by the matrix, their vector length will be stretched by a factor of 5 and -2, respective to each of the eigenvectors. By plugging in the discovered eigenvalues into our originally derived equation, we can find the eigenvectors.

$$\lambda = 5, -2$$

$$A = \begin{bmatrix} 1 & 4 \\ 3 & 2 \end{bmatrix}$$

$$(A - \lambda I)x = 0$$

$$\begin{bmatrix} 1-\lambda & 4 \\ 3 & 2-\lambda \end{bmatrix} x = 0 \qquad \begin{bmatrix} 1-\lambda & 4 \\ 3 & 2-\lambda \end{bmatrix} x = 0$$

$$\begin{bmatrix} -4 & 4 \\ 3 & -3 \end{bmatrix} x = 0 \qquad \begin{bmatrix} 3 & 4 \\ 3 & 4 \end{bmatrix} x = 0$$

$$x = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad x = \begin{bmatrix} -4 \\ 3 \end{bmatrix}$$

Thank You!

NANYANG
THE INNOVATIVE POLYTECHNIC

www.nyp.edu.sg