Advanced Linear Algebra AOL Documentation

Nathanael Romaloburju[1], Richie Rizawardana[2], Robben Wijanathan[3], and Weneville[4]

[1]**2802538132, *Computer Science & Mathematics, Binus University, Jakarta, Indonesia***
[2]**2802403100, *Computer Science & Mathematics, Binus University, Jakarta, Indonesia***
[3]**2802461681, *Computer Science & Mathematics, Binus University, Jakarta, Indonesia***
[4]**2802549016, *Computer Science & Mathematics, Binus University, Jakarta, Indonesia***

ABSTRACT

This project presents a simple Python-based Matrix Analyzer program designed to perform key linear algebra operations. The system provides an interactive menu with four main features: checking if a matrix is diagonalizable, performing LU decomposition, computing the dominant eigenvalue using the Power Method, and conducting Singular Value Decomposition (SVD). Each operation is implemented using fundamental numerical methods in Python.

1 MENU USER INTERFACE

```
# Clear Screen
def clear():
    if os.name == 'nt':
        os.system('cls')
    else:
        os.system('clear')

def mainMenu():

    clear()

    print(r"""
 __  __       _        _               _                _
|  \/  |     | |      (_)     /\      | |
| \  / | __ _| |_ _ __ ___  __   /  \   _ __   __ _| |_   _ _______ _ __
| |\/| |/ _` | __| '__| \ \/ /  / /\ \ | '_ \ / _` | | | | |_  / _ \ '__|
| |  | | (_| | |_| |  | |>  <  / ____ \| | | | (_| | | |_| |/ /  __/ |
|_|  |_|\__,_|\__|_|  |_/_/\_\/_/    \_\_| |_|\__,_|_|\__, /___\___|_|
                                                       __/ |
                                                      |___/
""")

    print("1. Check Diagonalizability")
    print("2. Perform LU Decomposition")
    print("3. Find Dominant Eigenvalue")
    print("4. Singular Value Decomposition (SVD)")
    print("5. Exit")

    while True:
        choice = input("Enter your choice (1-5): ")
        if choice == "1":
            isDiagonalizable()
            break
        elif choice == "2":
            LUDecomposition()
            break
        elif choice == "3":
            dominantEigenvalue()
            break
        elif choice == "4":
            svd()
            break
        elif choice == "5":
```

```
            print("\nExiting Matrix Analyzer Tool. Goodbye!")
            break
        else:
            print("\nInvalid choice. Please enter a number from 1 to 5.")
```

- The function `clear()` clears the terminal screen depending on the operating system (Windows or Linux/macOS).
- The function `mainMenu()` displays an ASCII art banner and lists five main options for matrix operations:
  1. Check Diagonalizability
  2. Perform LU Decomposition
  3. Find Dominant Eigenvalue
  4. Perform Singular Value Decomposition (SVD)
  5. Exit the program
- It then continuously prompts the user for input (`1-5`) and calls the corresponding function (e.g., `isDiagonalizable()`, `LUDecomposition()`, etc.).
- If the input is invalid, an error message is displayed until a valid choice is entered.

Output

```
 __  __       _        _                              _
|  \/  |     | |      (_)              /\             | |
| \  / | __ _| |_ _ __ ___  __        /  \   _ __   __ _| |_   _ _______ _ __
| |\/| |/ _` | __| '__| \ \/ /       / /\ \ | '_ \ / _` | | | | |_  / _ \ '__|
| |  | | (_| | |_| |  | |>  <       / ____ \| | | | (_| | | |_| |/ /  __/ |
|_|  |_|\__,_|\__|_|  |_/_/\_\     /_/    \_\_| |_|\__,_|_|\__, /___\___|_|
                                                            __/ |
                                                           |___/

1. Check Diagonalizability
2. Perform LU Decomposition
3. Find Dominant Eigenvalue
4. Singular Value Decomposition (SVD)
5. Exit
Enter your choice (1-5): a

Invalid choice. Please enter a number from 1 to 5.
Enter your choice (1-5): 0

Invalid choice. Please enter a number from 1 to 5.
Enter your choice (1-5): 1
```

2 INPUT MATRIX FUNCTIONS

These two functions handle user input for matrix creation, ensuring valid and properly formatted entries before performing matrix operations.

2.1 $m \times n$ Matrix (For Singular Value Decomposition)

```
def input_matrix():
    while True:
        try:
            rows = int(input("Enter number of rows: "))
            cols = int(input("Enter number of columns: "))
            break
        except ValueError:
            print("Please enter valid integers for rows and columns.\n")

    print("\nEnter the matrix values row by row (separated by spaces):")
    matrix = []

```

```python
    for i in range(rows):
        while True:
            row_input = input(f"Row {i + 1}: ").split()

            if len(row_input) != cols:
                print(f"Row {i + 1} must have exactly {cols} entries. Try again.")
                continue

            try:
                row = [float(x) for x in row_input]
                matrix.append(row)
                break
            except ValueError:
                print(f"Row {i + 1} contains non-numeric values. Try again.")

    return matrix
```

The function `input_matrix()` is designed to input a general rectangular matrix of size $m \times n$

- The user is first prompted to enter the number of rows (m) and columns (n) n), with input validation to ensure they are integers.
- Then, for each row, the user must input numeric values separated by spaces.
- The program checks that:
  - Each row contains exactly n entries.
  - All entries are valid numeric values (converted to `float`).
- If invalid input is detected, an error message is displayed and the user is asked to re-enter that row.
- Once all inputs are valid, the function returns the complete matrix as a list of lists.

2.2 $n \times n$ Square Matrix (For Checking Diagonalizability, LU Decomposition, & Finding Dominant Eigenvalue)

```python
def input_squareMatrix():
    while True:
        try:
            n = int(input("Enter the size of the square matrix (n x n): "))
            if n <= 0:
                print("Matrix size must be a positive integer.\n")
                continue
            break
        except ValueError:
            print("Please enter a valid integer for the matrix size.\n")

    print(f"\nEnter the {n}x{n} matrix values row by row (separated by spaces):")
    matrix = []

    for i in range(n):
        while True:
            row_input = input(f"Row {i + 1}: ").split()

            if len(row_input) != n:
                print(f"Row {i + 1} must have exactly {n} entries. Try again.")
                continue

            try:
                row = [float(x) for x in row_input]
                matrix.append(row)
                break
            except ValueError:
                print(f"Row {i + 1} contains non-numeric values. Try again.")

    return matrix
```

The function `input_squareMatrix()` is used for creating a square matrix of size $n \times n$.

- It starts by asking the user for the matrix size n and checks that the input is a positive integer.
- The user is then prompted to enter each row of the matrix, with input separated by spaces.
- The program validates that:
  - Each row contains exactly n numeric entries.
  - Non-numeric or mismatched inputs trigger an error message and require re-entry.
- After successful input, the matrix is returned as a list of lists to be used in other operations such as diagonalizability checking, LU decomposition, or eigenvalue calculation.

3 CORE MAIN FUNCTIONS

3.1 Diagonalizable Matrix

Theory A square matrix A of size $n \times n$ is said to be **diagonalizable** if it can be expressed as:

$$A = PDP^{-1}$$

where D is a diagonal matrix whose diagonal elements are the eigenvalues of A, and P is a matrix whose columns are the corresponding eigenvectors of A.

- **Diagonalizability:** A matrix is diagonalizable if there exists an invertible matrix P such that $A = PDP^{-1}$. This means the linear transformation represented by A can be simplified to scaling operations along independent directions defined by the eigenvectors.
- **Eigenvalues:** The eigenvalues of a matrix A are scalars λ that satisfy the equation:

  $$A\mathbf{v} = \lambda\mathbf{v}$$

  where $\mathbf{v}$ is a non-zero vector. Each eigenvalue represents how its corresponding eigenvector is scaled when transformed by A.
- **Eigenvectors:** The eigenvectors are non-zero vectors that do not change direction when multiplied by A; only their magnitude changes by the factor of their eigenvalue.

A matrix is diagonalizable if and only if it has n linearly independent eigenvectors, allowing it to be represented as a diagonal matrix in the basis of those eigenvectors.

```
def isDiagonalizable():
    clear()
    print(r"""
___  _                         _ _         _    _    ___
|   \(_)__ _ __ _ ___ _ _  __ _| (_)______ _| |__| |__|__ \
| |) | / _` / _` / _ \ ' \/ _` | | |_ / _` | '_ \ / -_)/_/
|___/|_\__,_\__, \___/_||_\__,_|_|_/__\__,_|_.__/_\___(_)
            |___/
""")

    matrix = input_squareMatrix()
    n = len(matrix)
    A = np.array(matrix, dtype=float)

    eigvals, eigvecs = np.linalg.eig(A)

    rank = np.linalg.matrix_rank(eigvecs)

    unique_eigs = np.unique(np.round(eigvals, 6))
    diagonalizable = True

    for eig in unique_eigs:
        algebraic_mult = np.sum(np.isclose(eigvals, eig))

```

```python
        B = A - eig * np.eye(n)
        geometric_mult = n - np.linalg.matrix_rank(B)

        if geometric_mult < algebraic_mult:
            diagonalizable = False


    print("\nEigenvalues:")
    print(np.round(eigvals, 4))

    print("\nEigenvectors:")
    print(np.round(eigvecs, 4))

    if diagonalizable:
        print("\nMatrix IS diagonalizable!")
    else:
        print("\nMatrix is NOT diagonalizable.")

    print()
    print("Press Enter to go back to main menu...")
    input()
    mainMenu()
```

Explanation The function `isDiagonalizable()` determines whether a given square matrix is diagonalizable by analyzing its eigenvalues and eigenvectors.

- The function begins by clearing the screen and displaying an ASCII art title for visual clarity.
- It then calls the `input_squareMatrix()` function to receive user input for a square matrix, which is converted into a NumPy array A.
- The eigenvalues and eigenvectors of A are computed using `np.linalg.eig(A).`
- The function checks the **algebraic multiplicity** (number of times each eigenvalue appears) and the **geometric multiplicity** (dimension of the eigenspace for each eigenvalue).
- For each unique eigenvalue λ_i:
  - The algebraic multiplicity is calculated by counting how many times λ_i occurs in the eigenvalue list.
  - The geometric multiplicity is computed as $n - \text{rank}(A - \lambda_i I)$.
- If for any eigenvalue the geometric multiplicity is less than its algebraic multiplicity, the matrix is declared **not diagonalizable**.
- Otherwise, if all multiplicities match, the matrix is considered **diagonalizable**.
- Finally, the function prints the eigenvalues, eigenvectors, and the result (diagonalizable or not), before prompting the user to return to the main menu.

Output (Not Diagonalizable)

```
 ___  _                     _ _          _    _      ___ 
|   \(_)__ _ __ _ ___ _ _  __ _| (_)______ _| |__| |___|__ \
| |) | / _` / _` / _ \ ' \/ _` | | |_ / _` | '_ \ / -_)/_/
|___/|_\__,_\__, \___/_||_\__,_|_|_/__\__,_|_.__/_\___(_) 
            |___/                                         

Enter the size of the square matrix (n x n): a
Please enter a valid integer for the matrix size.

Enter the size of the square matrix (n x n): 1.5
Please enter a valid integer for the matrix size.

Enter the size of the square matrix (n x n): 3

Enter the 3x3 matrix values row by row (separated by spaces):
Row 1: a b c
```

```
Row 1 contains non-numeric values. Try again.
Row 1: 1 2
Row 1 must have exactly 3 entries. Try again.
Row 1: 1 2 3
Row 2: 6 5 3
Row 3: 7 2 4

Eigenvalues:
[10.3975 -2.629   2.2315]

Eigenvectors:
[[-0.3455 -0.7001  0.0863]
 [-0.7192  0.2949 -0.8122]
 [-0.6029  0.6503  0.5769]]

Matrix is NOT diagonalizable.

Press Enter to go back to main menu...
```

Output (Diagonalizable)

```
 ____  _                               _ _          _     _      ___
|  _ \(_) __ _  __ _  ___  _ __   __ _| (_)______ _| |__ | | ___|__ \
| | | | |/ _` |/ _` |/ _ \| '_ \ / _` | | |_  / _` | '_ \| |/ _ \ / /
| |_| | | (_| | (_| | (_) | | | | (_| | | |/ / (_| | |_) | |  __/|_|
|____/|_|\__,_|\__, |\___/|_| |_|\__,_|_|_/___\__,_|_.__/|_|\___|(_)
               |___/

Enter the size of the square matrix (n x n): 2

Enter the 2x2 matrix values row by row (separated by spaces):
Row 1: 4 1
Row 2: 0 2

Eigenvalues:
[4. 2.]

Eigenvectors:
[[ 1.      -0.4472]
 [ 0.       0.8944]]

Matrix IS diagonalizable!

Press Enter to go back to main menu...
```

3.2 LU Decomposition

Theory The **LU Decomposition** (or **LU Factorization**) is a technique used to decompose a square matrix A into the product of two simpler matrices:

$$A = LU$$

where:

- L is a **lower triangular matrix** with ones on its diagonal.
- U is an **upper triangular matrix**.

This factorization expresses the matrix A in a form that simplifies various numerical computations. It is especially useful in linear algebra, numerical analysis, and computational mathematics because it allows complex operations to be broken down into simpler steps.

- **Applications:**
  - **Solving Systems of Linear Equations:** Instead of directly solving $A\mathbf{x} = \mathbf{b}$, LU decomposition rewrites it as:

    $$L\mathbf{y} = \mathbf{b} \quad \text{and} \quad U\mathbf{x} = \mathbf{y}$$

    which can be solved efficiently using forward and backward substitution.

- **Computing Determinants:** The determinant of A can be easily computed as:

$$\det(A) = \det(L) \times \det(U)$$

Since $\det(L) = 1$, it follows that $\det(A) = \prod_i U_{ii}$, the product of the diagonal elements of U.

- **Finding Inverses:** LU decomposition also aids in finding the inverse of a matrix by solving multiple systems of linear equations for each column of the identity matrix.

- **Numerical Stability:** When row exchanges are required for stability, a **permutation matrix** P is introduced:

$$PA = LU$$

This process, known as **partial pivoting**, minimizes numerical errors in floating-point computations.

```
def LUDecomposition():
    clear()
    print(r"""
__   __  __  ___                                   _ __  _
/ /  / / / / / _ \___ _______  __ _  ___  ___  ___ (_) /_(_)__  ___
/ /__/ /_/ / / // / -_) __/ _ \/  ' \/ _ \/ _ \(_-</ / __/ / _ \/ _ \
/____/\____/ /____/\__/\__/\___/_/_/_/ .__/\___/___/_/\__/_/\___/_//_/
                                    /_/
""")


    matrix = input_squareMatrix()
    n = len(matrix)

    lower = [[0.0 for _ in range(n)] for _ in range(n)]
    upper = [[0.0 for _ in range(n)] for _ in range(n)]

    for i in range(n):
        # Upper Triangular
        for k in range(i, n):
            temp_sum = sum(lower[i][j] * upper[j][k] for j in range(i))
            upper[i][k] = matrix[i][k] - temp_sum

        # Lower Triangular
        for k in range(i, n):
            if i == k:
                lower[i][i] = 1.0
            else:
                temp_sum = sum(lower[k][j] * upper[j][i] for j in range(i))
                if upper[i][i] == 0:
                    print("Matrix is singular, cannot perform LU decomposition.")
                    print()
                    print("Press Enter to go back to main menu...")
                    input()
                    mainMenu()
                lower[k][i] = (matrix[k][i] - temp_sum) / upper[i][i]

    print("\nLower Triangular Matrix:")
    for row in lower:
        print([f"{x:.2f}" for x in row])

    print("\nUpper Triangular Matrix:")
    for row in upper:
        print([f"{x:.2f}" for x in row])

    print()
    print("Press Enter to go back to main menu...")
    input()
    mainMenu()
```

Explanation The function `LUDecomposition()` performs an LU decomposition of a user-input square matrix without using external libraries. It breaks the given matrix A into a **lower triangular matrix** L and an **upper triangular matrix** U such that:

$$A = LU$$

- The function first clears the screen and displays an ASCII art title for better presentation.
- It then calls the `input_squareMatrix()` function to receive the matrix input from the user.
- Two empty matrices, `lower` and `upper`, of size $n \times n$ are initialized with zeros.
- The decomposition process is done in two nested loops:
  - **Upper Triangular Calculation:** For each row i, the elements of U are computed using:

    $$U_{i,k} = A_{i,k} - \sum_{j=0}^{i-1} L_{i,j} U_{j,k}$$

    This ensures that U contains nonzero values only on and above its main diagonal.
  - **Lower Triangular Calculation:** For each column i, the diagonal of L is set to 1. The remaining elements below the diagonal are computed as:

    $$L_{k,i} = \frac{A_{k,i} - \sum_{j=0}^{i-1} L_{k,j} U_{j,i}}{U_{i,i}}$$

    This ensures that L contains ones on its diagonal and nonzero values below it.
- The function also checks if any pivot element $U_{i,i} = 0$, in which case the matrix is **singular** and LU decomposition cannot be performed.
- After the decomposition, both L and U are printed with values rounded to two decimal places for readability.
- Finally, the program waits for the user to press Enter before returning to the main menu.

Output

```
__    __   __   ___                                          _  __   _
/ /  / / / / / _ \___  _______   __ _  ___  ___  ___ (_) /_(_)__  ___
/ /__/ /_/ / / // / -_) __/ _ \/  ' \/ _ \/ _ \(_-</ / __/ / _ \/ _ \
/____/\____/ /____/\__/\__/\___/_/_/_/ .__/\___/___/_/\__/_/\___/_//_/
                                     /_/

Enter the size of the square matrix (n x n): 3

Enter the 3x3 matrix values row by row (separated by spaces):
Row 1: 1 2 3
Row 2: 3 4 5
Row 3: 5 6 7

Lower Triangular Matrix:
['1.00', '0.00', '0.00']
['3.00', '1.00', '0.00']
['5.00', '2.00', '1.00']

Upper Triangular Matrix:
['1.00', '2.00', '3.00']
['0.00', '-2.00', '-4.00']
['0.00', '0.00', '0.00']

Press Enter to go back to main menu...
```

Output (Singular)

```
__    __   __   ___                                          _  __   _
/ /  / / / / / _ \___  _______   __ _  ___  ___  ___ (_) /_(_)__  ___
/ /__/ /_/ / / // / -_) __/ _ \/  ' \/ _ \/ _ \(_-</ / __/ / _ \/ _ \
/____/\____/ /____/\__/\__/\___/_/_/_/ .__/\___/___/_/\__/_/\___/_//_/
```

```
                                        / _/

Enter the size of the square matrix (n x n): 3

Enter the 3x3 matrix values row by row (separated by spaces):
Row 1: 2 4 6
Row 2: 3 6 1
Row 3: 7 8 5
Matrix is singular, cannot perform LU decomposition.

Press Enter to go back to main menu...
```

3.3 Dominant Eigenvalue

Theory In the context of a square matrix $A \in \mathbb{R}^{n\times n}$, the *dominant eigenvalue* is usually understood to be an eigenvalue λ_1 satisfying

$$|\lambda_1| > |\lambda_j| \quad \text{for all } j = 2,3,\ldots,n.$$

That is, the magnitude of λ_1 strictly exceeds that of every other eigenvalue of A. A corresponding eigenvector $\mathbf{v}_1 \neq \mathbf{0}$ satisfies

$$A\,\mathbf{v}_1 = \lambda_1\,\mathbf{v}_1.$$

A common numerical method to find λ_1 (and $\mathbf{v}_1$) is the *Power Method*. The basic algorithm proceeds as follows:

- Choose an initial non-zero vector $\mathbf{b}_0$.
- For $k = 0,1,2,\ldots$:

$$\mathbf{b}_{k+1} = \frac{A\,\mathbf{b}_k}{\|A\,\mathbf{b}_k\|}.$$

- After each iteration one may estimate the eigenvalue via the Rayleigh quotient:

$$\mu_k = \frac{\mathbf{b}_k^T\,(A\,\mathbf{b}_k)}{\mathbf{b}_k^T\mathbf{b}_k}.$$

Convergence Criteria and Remarks:

- The method converges (i.e., $\mathbf{b}_k \to \pm\mathbf{v}_1$ and $\mu_k \to \lambda_1$) provided that:
  1. A has a unique eigenvalue λ_1 such that $|\lambda_1| > |\lambda_j|$ for all $j > 1$.
  2. The initial vector $\mathbf{b}_0$ has a non-zero component in the direction of $\mathbf{v}_1$; equivalently, when writing $\mathbf{b}_0 = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_n\mathbf{v}_n$, the coefficient $c_1 \neq 0$.
- The rate of convergence is governed by the ratio

$$\left|\frac{\lambda_2}{\lambda_1}\right|,$$

  where λ_2 is the eigenvalue of next largest absolute value (i.e., the "sub-dominant" eigenvalue). If $\left|\frac{\lambda_2}{\lambda_1}\right|$ is close to zero, convergence is very rapid; if it is close to one, convergence is slow. :contentReference[oaicite:0]index=0
- If $|\lambda_1| = |\lambda_2|$ (or there's a dominant eigenspace of dimension ¿ 1), or if $\mathbf{b}_0$ is orthogonal to $\mathbf{v}_1$, the power method may fail to converge to the correct eigen-pair. :contentReference[oaicite:1]index=1

```python
def power_method(A, iter=2000, tolerance=1e-9):
    n, _ = A.shape
    x = np.ones(n, dtype=float)
    x = x / np.linalg.norm(x)

    eigen_start = 0.0
    x_prevprev = None

    for _ in range(iter):
```

```
        Ax = A @ x
        normAx = np.linalg.norm(Ax)
        if normAx == 0.0:
            return 0.0, x

        x_new = Ax / normAx
        eigen_new = float(np.dot(x_new, Ax) / np.dot(x_new, x_new))

        if x_prevprev is not None:
            if abs(np.dot(x_new, x_prevprev)) > 1.0 - 1e-8:
                return None, None  # signal: no unique dominant eigenvalue (tie/complex)

        if abs(eigen_new - eigen_start) < tolerance:
            return eigen_new, x_new

        x_prevprev = x
        x = x_new
        eigen_start = eigen_new

    return None, None  # non-convergent within iter

def dominantEigenvalue():
    clear()
    print(r"""
___             _                __
/ _ \___  __ _  (_)__  ___ ____  / /_
/ // / _ \/  ' \/ / _ \/ _ `/ _ \/ __/
/____/\___/_/_/_/_/_//_/\_,_/_//_/\__/
/ __(_)__ ____ ___ _   _____ _/ /_ _____
/ _// / _ `/ -_) _ \ |/ / _ `/ / // / -_)
/___/_/\_, /\__/_//_/___/\_,_/_/\_,_/\__/
    /___/
""")

    matrix = input_squareMatrix()
    matrix = np.array(matrix, dtype=float)

    eig, x = power_method(matrix, iter=2000, tolerance=1e-9)

    if eig is None:
        print("Power method did not converge to a unique dominant eigenvalue.")
    else:
        print(f"Dominant eigenvalue: {eig}")
        print(f"Dominant eigenvector: {x}")
    print("Press Enter to go back to main menu...")
    input()
    mainMenu()
```

Explanation The `dominantEigenvalue()` function computes the **dominant eigenvalue** and its corresponding **eigenvector** of a given square matrix using the **Power Method**, a simple iterative numerical algorithm.

- The function first clears the screen and displays an ASCII title for presentation purposes.
- It then calls `input_squareMatrix()` to allow the user to input a square matrix, which is converted into a NumPy array for computation.
- The core logic is handled by the helper function `power_method(A, iter, tolerance)`, which performs the iterative computation:
  - The algorithm begins with an initial guess vector x (initialized as a vector of ones) and normalizes it.
  - In each iteration, the algorithm multiplies the matrix A by the current vector x:

$$\mathbf{Ax} = \mathbf{Ax}$$

– The resulting vector is normalized to produce the next approximation $\mathbf{x}_{\text{new}}$.

– The eigenvalue estimate is updated using the **Rayleigh quotient**:

$$\lambda_{\text{new}} = \frac{\mathbf{x}_{\text{new}}^T (A\mathbf{x}_{\text{new}})}{\mathbf{x}_{\text{new}}^T \mathbf{x}_{\text{new}}}$$

– Convergence is checked by comparing the change in eigenvalue estimates between iterations. If

$$|\lambda_{\text{new}} - \lambda_{\text{old}}| < \text{tolerance},$$

the algorithm stops and returns the approximate dominant eigenvalue and eigenvector.

- The function then prints the computed dominant eigenvalue and corresponding normalized eigenvector.
- Finally, it waits for user input before returning to the main menu.

Output

```
___              _                __
/ _ \___  __ _  (_)__  ___ ____  / /_
/ // / _ \/  ' \/ / _ \/ _ `/ _ \/ __/
/____/\___/_/_/_/_/_//_/\_,_/_//_/\__/
/ __(_)__ ____ ___ _  _____ _/ /_ _____
/ _// / _ `/ -_) _ \ |/ / _ `/ / // / -_)
/___/_/\_, /\__/_//_/___/\_,_/_/\_,_/\__/
    /___/

Enter the size of the square matrix (n x n): 3

Enter the 3x3 matrix values row by row (separated by spaces):
Row 1: 2 4 5
Row 2: 5 3 4
Row 3: 1 2 3
Dominant eigenvalue: 9.219264347529903
Dominant eigenvector: [0.62158879 0.7110826  0.32861668]
Press Enter to go back to main menu...
```

Output (Repeated Eigenvalues)

```
___              _                __
/ _ \___  __ _  (_)__  ___ ____  / /_
/ // / _ \/  ' \/ / _ \/ _ `/ _ \/ __/
/____/\___/_/_/_/_/_//_/\_,_/_//_/\__/
/ __(_)__ ____ ___ _  _____ _/ /_ _____
/ _// / _ `/ -_) _ \ |/ / _ `/ / // / -_)
/___/_/\_, /\__/_//_/___/\_,_/_/\_,_/\__/
    /___/

Enter the size of the square matrix (n x n): 2

Enter the 2x2 matrix values row by row (separated by spaces):
Row 1: 2 1
Row 2: 0 2
Power method did not converge to a unique dominant eigenvalue.
Press Enter to go back to main menu...
```

Output (Non-Dominant Eigenvalues)

```
___              _                __
/ _ \___  __ _  (_)__  ___ ____  / /_
/ // / _ \/  ' \/ / _ \/ _ `/ _ \/ __/
/____/\___/_/_/_/_/_//_/\_,_/_//_/\__/
/ __(_)__ ____ ___ _  _____ _/ /_ _____
/ _// / _ `/ -_) _ \ |/ / _ `/ / // / -_)
/___/_/\_, /\__/_//_/___/\_,_/_/\_,_/\__/
    /___/

Enter the size of the square matrix (n x n): 2
```

```
Enter the 2x2 matrix values row by row (separated by spaces):
Row 1: 0 -1
Row 2: 1 0
Power method did not converge to a unique dominant eigenvalue.
Press Enter to go back to main menu...
```

3.4 Singular Value Decomposition

Theory The **Singular Value Decomposition (SVD)** is a powerful matrix factorization technique that generalizes the concept of eigen-decomposition to any real or complex matrix $A \in \mathbb{R}^{m \times n}$, whether square or rectangular. It expresses A as the product of three matrices:

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

where:

- $U \in \mathbb{R}^{m \times m}$ is an **orthogonal matrix** whose columns are the **left singular vectors** of A.
- $\Sigma \in \mathbb{R}^{m \times n}$ is a **diagonal matrix** containing the **singular values** $\sigma_1, \sigma_2, \ldots, \sigma_r$ (where $r = \text{rank}(A)$).
- $V \in \mathbb{R}^{n \times n}$ is an **orthogonal matrix** whose columns are the **right singular vectors** of A.

Mathematical Background

The singular values of A are defined as the non-negative square roots of the eigenvalues of $A^\top A$ (or equivalently $AA^\top$):

$$A^\top A v_i = \sigma_i^2 v_i \quad \text{and} \quad AA^\top u_i = \sigma_i^2 u_i$$

Here, v_i and u_i are the right and left singular vectors corresponding to the singular value σ_i, respectively. The singular values are typically arranged in descending order:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$$

Each singular value measures the amount of “stretching” that A applies in the direction of its corresponding singular vector.

Geometric Interpretation

The SVD represents the linear transformation A as a sequence of three simpler transformations:

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

$$(\text{Rotation by } V^\top) \quad \longrightarrow \quad (\text{Scaling by } \Sigma) \quad \longrightarrow \quad (\text{Rotation by } U)$$

This means that A first rotates the input space, then scales it along orthogonal axes (by the singular values), and finally rotates it again.

Applications

SVD has a wide range of applications across mathematics, engineering, and data science:

- **Dimensionality Reduction:** Used in **Principal Component Analysis (PCA)** to reduce large datasets into smaller, more informative components.
- **Image Compression:** Images represented as matrices can be approximated using only the largest singular values, reducing storage while preserving quality.
- **Noise Filtering:** Small singular values often correspond to noise and can be truncated to enhance signal clarity.
- **Solving Linear Systems:** Provides stable solutions for ill-conditioned or non-square systems using the pseudoinverse:

  $$A^+ = V \Sigma^+ U^\top$$

- **Data Analysis and Recommender Systems:** Helps identify latent structures or patterns (e.g., in collaborative filtering like the Netflix algorithm).

```python
def svd():
    clear()
    print(r"""
 ,---.,--.   ,--.,-------.
'   .-'\  `.'  / |  .-.  \
```

```
`.  `-. \      /   |   |   \   :
.-'     | \   /    |   '--'   /
`-----'    `-'     `-------'
""")
    np.set_printoptions(precision=4, suppress=True)
    A = np.array(input_matrix()) #A = U  VT
    m, n = A.shape
    r = min(m, n)

    ATA = A.T @ A
    lamV, V = np.linalg.eigh(ATA)
    idxV = lamV.argsort()[::-1]
    lamV = lamV[idxV]
    V = V[:, idxV]

    #lamU = np.clip(lamU, 0.0, None)
    lamV = np.clip(lamV, 0.0, None)
    sigma = np.sqrt(lamV) #averaging in case of mismatch due to floating point rounding errors

    eps = np.finfo(float).eps
    tol = max(m, n) * eps * (sigma[0] if r else 0.0)
    k = int(np.sum(sigma > tol)) # numerical rank
    k = min(k, r)

    U = np.zeros((m, m))
    U[:, :k] = (A @ V[:, :k]) / sigma[:k].reshape(1, -1)

    d = np.sign(np.diag(U[:, :k].T @ A @ V[:, :k]))
    d[d == 0] = 1.0
    U[:, :k] *= d.reshape(1, -1)
    V[:, :k] *= d.reshape(1, -1)

    if m > k:
        Z = np.random.randn(m, m - k)
        Z -= U[:, :k] @ (U[:, :k].T @ Z)
        U2, _ = np.linalg.qr(Z)
        U[:, k:] = U2

    Sigma = np.zeros((m, n))
    Sigma[np.arange(r), np.arange(r)] = sigma[:r]

    def zapsmall(M, Aref=None, rel=50*np.finfo(float).eps, abs_=0.0):
        scale = np.linalg.norm(A if Aref is None else Aref, ord=np.inf)
        thr = abs_ + rel * scale
        M[np.abs(M) < thr] = 0.0
        return M

    U = zapsmall(U, A)
    V = zapsmall(V, A)
    Sigma = zapsmall(Sigma, A)

    print(f"\n\nU = {U};\n\n = {Sigma};\n\nV_T = {V.T}")
    print(f"Check\nA = U  V_T = {zapsmall(U @ Sigma @ V.T, A)};")
    print("Press Enter to go back to main menu...")
    input()
    mainMenu()
```

Explanation The function `svd()` performs the **Singular Value Decomposition (SVD)** of a user-input matrix A, decomposing it into three matrices U, Σ, and $V^\top$, such that:

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

This implementation computes the SVD manually using NumPy operations, providing detailed insight into the mathematical steps behind the decomposition process. Code snippets are presented before the explanation.

```
clear()
print(r"""
 ,---.,--.    ,--.,------.
'   .-'\  `.'  / |  .-.  \
`.  `-. \     /  | |  \  :
.-'    | \   /   |  '--'  /
`-----'   `-'    `-------'
""")
```

1. The function starts by clearing the screen and displaying an ASCII banner for presentation.

```
np.set_printoptions(precision=4, suppress=True)
A = np.array(input_matrix()) #A = U  VT
m, n = A.shape
r = min(m, n)
```

2. The decimal precision is set to four digits, to make results more comprehensible.
   The user inputs a matrix using the `input_matrix()` function, which is then stored as a NumPy array A.
   The matrix dimensions m and n are determined from the shape of the input matrix, and the variable $r = \min(m,n)$ represents the smaller dimension (the rank limit).

```
ATA = A.T @ A
lamV, V = np.linalg.eigh(ATA)
idxV = lamV.argsort()[::-1]
lamV = lamV[idxV]
V = V[:, idxV]
```

3. Next, the function computes $A^\top A$, which is a symmetric matrix, using the `np.linalg.eigh()` function, which is specially designed for hermitian/symmetric matrices. Its eigenvalues and eigenvectors are calculated using:

   $$(A^\top A)V = V\Lambda$$

   The eigenvalues in Λ correspond to the squares of the singular values, and the eigenvectors form the columns of V (the **right singular vectors**).

4. The eigenvalues (λ_i) are sorted in descending order, and any negative rounding errors are corrected using:

   $$\lambda_i = \max(\lambda_i, 0)$$

   The singular values (σ_i) are then obtained as:

   $$\sigma_i = \sqrt{\lambda_i}$$

```
lamV = np.clip(lamV, 0.0, None)
sigma = np.sqrt(lamV)
```

5. Mathematically $\lambda_i \geq 0$ and this correction wouldn't be necessary, however in the implementation, this is done to correct any floating-point round-off errors that may produce negative, yet practically zero, values, that indicate that the value is probably meant to be exactly zero.

```
eps = np.finfo(float).eps
tol = max(m, n) * eps * (sigma[0] if r else 0.0)
k = int(np.sum(sigma > tol)) # numerical rank
k = min(k, r)
```

6. A tolerance check is applied to determine the **numerical rank** of the matrix (handling floating-point precision issues), and QR decomposition is used to complete the orthonormal basis of U if necessary (for cases where $m > n$).

```
U = np.zeros((m, m))
U[:, :k] = (A @ V[:, :k]) / sigma[:k].reshape(1, -1)
```

7. To find the **left singular vectors** (U), the function uses the relationship:

$$U_i = \frac{AV_i}{\sigma_i}$$

This ensures that each column of U is orthonormal and corresponds to a singular direction of A. We do not compute U and the corresponding eigenvalues separately using `np.linalg.eigh()` because it would most probably generate a set of eigenvectors that, though valid, do not align with the equality $U_i = \frac{AV_i}{\sigma_i}$.

```
d = np.sign(np.diag(U[:, :k].T @ A @ V[:, :k]))
d[d == 0] = 1.0
U[:, :k] *= d.reshape(1, -1)
V[:, :k] *= d.reshape(1, -1)
```

8. This step is also unnecessary in pure mathematics. We know that

$$Av_i = \sigma_i u_i, \quad A^\top u_i = \sigma_i v_i$$

if we take any triplet (u_i, σ_i, v_i) and replace **both** u_i and v_i with their negatives, the equalities still hold. In the implementation, the problem again lies in round-off errors that can flip either one of u_i or v_i within the triple, but not the other. The code above fixes this, ensuring their signs uphold the equality

$$u_i = \frac{Av_i}{\sigma_i}$$

specifically, since in our implementation, U is derived from V.

```
if m > k:
    Z = np.random.randn(m, m - k)
    Z -= U[:, :k] @ (U[:, :k].T @ Z)
    U2, _ = np.linalg.qr(Z)
    U[:, k:] = U2
```

9. We have previously calculated the numerical rank k, which means we only have k values (u_i, σ_i, v_i) that are *true*, i.e., not as a result of round-off errors. So now

$$U \in \mathbb{R}^{m\times k}, \quad \Sigma \in \mathbb{R}^{k\times k}, \quad V \in \mathbb{R}^{n\times k}$$

, but we want to construct the full matrices

$$U \in \mathbb{R}^{m\times m}, \quad \Sigma \in \mathbb{R}^{m\times n}, \quad V \in \mathbb{R}^{n\times n}$$

This is precisely what the code snippet above does.

```
Sigma = np.zeros((m, n))
Sigma[np.arange(r), np.arange(r)] = sigma[:r]
```

10. The diagonal matrix Σ is then constructed by placing the singular values along its diagonal:

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \cdots \\ 0 & \sigma_2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

```
def zapsmall(M, Aref=None, rel=50*np.finfo(float).eps, abs_=0.0):
    scale = np.linalg.norm(A if Aref is None else Aref, ord=np.inf)
    thr = abs_ + rel * scale
    M[np.abs(M) < thr] = 0.0
    return M

U = zapsmall(U, A)
V = zapsmall(V, A)
Sigma = zapsmall(Sigma, A)
```

11. The helper function `zapsmall()` removes very small numerical values (near zero) to clean up floating-point noise and improve readability of the output.

```
print(f"\n\nU = {U};\n\n = {Sigma};\n\nV_T = {V.T}")
print(f"Check\nA = U  V_T = {zapsmall(U @ Sigma @ V.T, A)};")
print("Press Enter to go back to main menu...")
input()
mainMenu()
```

12. Finally, the program prints the resulting matrices U, Σ, and $V^\top$, and verifies the decomposition by reconstructing A through:

$$A \approx U\Sigma V^\top$$

This check confirms the correctness of the SVD.

Output

```
 ,---.,--.   ,--.,-------.
'   .-'\   `.'  / |  .-.  \
`.  `-. \     /  |  | \  :
.-'    | \   /   |  '--'  /
`-----'   `-'    `-------'

Enter number of rows: 2
Enter number of columns: 3

Enter the matrix values row by row (separated by spaces):
Row 1: 1 2 3
Row 2: 4 5 6


U = [[-0.3863  0.9224]
 [-0.9224 -0.3863]];

Σ = [[9.508  0.     0.    ]
 [0.     0.7729 0.    ]];

V_T = [[-0.4287 -0.5663 -0.7039]
 [-0.806  -0.1124  0.5812]
 [ 0.4082 -0.8165  0.4082]]
Check
A = U Σ V_T = [[1. 2. 3.]
 [4. 5. 6.]];
Press Enter to go back to main menu...
```

Output (Zero Singular Values)

```
 ,---.,--.   ,--.,-------.
'   .-'\   `.'  / |  .-.  \
`.  `-. \     /  |  | \  :
.-'    | \   /   |  '--'  /
`-----'   `-'    `-------'

Enter number of rows: 2
Enter number of columns: 3
```

```
Enter the matrix values row by row (separated by spaces):
Row 1: 1 2 3
Row 2: 2 4 6


U = [[0.4472 0.     ]
 [0.8944 0.     ]];

Σ = [[8.3666 0.     0.     ]
 [0.     0.     0.     ]];

V_T = [[ 0.2673  0.5345  0.8018]
 [-0.9554  0.0386  0.2927]
 [-0.1255  0.8443 -0.521 ]]
Check
A = U Σ V_T = [[1. 2. 3.]
 [2. 4. 6.]];
Press Enter to go back to main menu...
```