# Documentation for AnassLimpy Library

Programming M2 MMVAI

Anass Bouaicha

## Contents

# 1 Introduction

This document serves as the documentation for the AnassLimpy library, a Python library for linear algebra operations. It also includes instructions on how to implement and use the library.

# 2 Library Description

The AnassLimpy library is designed to provide users with a set of tools for performing various linear algebra operations. It simplifies tasks such as dot product, matrix multiplication, finding the determinant of a matrix, etc.

# 3 Installation Guide

## 3.1 Downloading the Library

1. Visit the AnassLimpy GitHub repository at `https://github.com/anassbh-dev/LinearAlgebraLib-MMVAI`
2. Navigate to the `AnassLimpy.py` file in the repository.
3. Open the file and click on the 'Raw' button to view the raw text version.
4. Save the file to your local machine by right-clicking on the page and selecting "Save Page As...". Make sure the file is named `AnassLimpy.py`.

## 3.2 Using the Library

After downloading the file, place `AnassLimpy.py` in the same directory as your Python script or notebook. Import the library using the following Python code:

```
import AnassLimpy as Lim
```

You can now use the functions and classes provided by the library. For example:

```
a = Lim.MyRandom().random(3, 4)
b = Lim.MyRandom().random(4, 2)
c = Lim.dot_product(a, b)
```

# 4 Class: MyRandom

## 4.1 Initialization

The class is initialized with an optional seed parameter, which defaults to the current time. This seed is used to start the sequence of random numbers. The constants $a$, $b$, and $c$ are internal parameters used by the LCG algorithm to generate the next number in the sequence.

## 4.2 Methods

### 4.2.1 rand

The `rand` method computes the next pseudo-random number in the sequence by updating the seed with theformula:

$$\text{seed} = (\text{a} \cdot \text{seed} + \text{b}) \mod \text{c}$$

This method ensures that the result is an integer and returns it as the next random number.

### 4.2.2 random

The `random` method generates a multi-dimensional array of random numbers. The dimensions of the array are specified by the arguments passed to the method. If no dimensions are provided, the method raises a `ValueError`.

#### 4.2.3 generate_array

This recursive helper method is used by `random` to build the multi-dimensional array. At each recursive call, it constructs one dimension of the array until the final dimension is reached, at which point it calls `rand` to fill the array with random numbers.

### 4.3 Example Usage

The following example demonstrates how to initialize the `MyRandom` class and generate a 2x3 array of random numbers:

```
rnd = MyRandom()
random_array = rnd.random(2, 3)
print(random_array)
```

The output will be a 2x3 array where each element is a random number.

## 5 Function: dot_product

### 5.1 Description

The `dot_product` function is designed to compute the scalar or dot product between two arrays. The function can handle the dot product operation between two vectors (1D arrays), a matrix and a vector (2D array and 1D array), and between two matrices (2D arrays). Additionally, it supports dot products for higher-dimensional tensors (3D arrays), provided they have matching dimensions.

### 5.2 Validation

To validate the correctness of the `dot_product` function, we compare its output against the established NumPy library's equivalent function. The following are test cases used for this purpose:

#### 5.2.1 1D Array Dot Product

```
# Example usage:
rnd = MyRandom()
a = rnd.random(4)
b = rnd.random(4)
c = dot_product(a, b)
```

For vectors `a` and `b`:

```
a = [5, 46, 3, 16]
b = [23, 34, 11, 42]
```

Our function's output: $c = 2384$

NumPy's output:

```
c_np = np.dot(a, b)
```

Comparison:

```
assert c == c_np
```

#### 5.2.2 2D Array and 1D Array Dot Product

```
a = rnd.random(4,2)
b = rnd.random(2)
c = dot_product(a, b)
```

For matrix `a` and vector `b`:

```
a = [[6, 14], [40, 7], [29, 30], [45, 35]]
b = [26, 32]
```

Our function's output: $c = [604, 1264, 1714, 2290]$

NumPy's output:

```
c_np = np.dot(a, b)
```

Comparison:

```
assert c == c_np.tolist()
```

#### 5.2.3 3D Array Dot Product

```
a = rnd.random(2,2,2)
b = rnd.random(2,2,2)
c = dot_product(a, b)
```

For tensors `a` and `b`:

```
a = [[[11, 42], [37, 9]], [[12, 10], [27, 0]]]
b = [[[18, 6], [14, 40]], [[7, 29], [30, 45]]]
```

Our function's output: $c = [[[1196, 3614], [1517, 2368]], [[754, 1292], [864, 1242]]]$

Comparison: In each case, the output from our `dot_product` function is asserted to be equal to the output from NumPy's equivalent function, validating the correctness of our implementation.

# 6 Function: inner_product

## 6.1 Description

The `inner_product` function calculates the inner product of two vectors, which is a single number obtained by performing a specific operation on two sequences of numbers. In the context of Euclidean space, the inner product is equivalent to the dot product, which combines corresponding elements of two vectors and sums the results.

In this implementation, the `inner_product` function is designed as a direct call to the `dot_product` function, reflecting the mathematical truth that for real-valued vectors, the inner product and dot product are identical operations. The terms "inner product" and "dot product" are often used interchangeably when dealing with real vector spaces. The inner product generalizes the dot product to more abstract vector spaces, where it can be defined in ways not limited to element-wise multiplication followed by a summation. However, in Euclidean spaces and with real vectors, both concepts converge into the same operation, which is why the `inner_product` function utilizes the existing `dot_product` implementation.

# 7 Function: outer_product

## 7.1 Description

The `outer_product` function calculates the outer product of two vectors, which are represented as 1D lists. The outer product of two vectors results in a matrix where each element $(i, j)$ is the product of elements $a_i$ from the first vector and $b_j$ from the second vector.

## 7.2 Validation

The function is validated by using it to calculate the outer product of two vectors generated by the `MyRandom` class's `random` method. The output is compared to that of NumPy's `np.outer` function to ensure correctness.

First, we initialize the random number generator and generate two vectors:

```
rnd = MyRandom()
a = rnd.random(4)
b = rnd.random(4)
```

The outer product of vectors `a` and `b` is then computed using our `outer_product` function:

```
c = outer_product(a, b)
```

For validation, we compute the outer product of the same vectors using NumPy's `np.outer` function:

```
import numpy as np
c_np = np.outer(a, b)
```

The resulting matrix from our function `c` and the matrix from NumPy `c_np` are compared to check for equality:

```
assert np.array_equal(c, c_np)
```

The assertion should not raise an error if our `outer_product` function is implemented correctly, indicating that the function produces an output consistent with NumPy's `np.outer` method.

# 8 Function: matrix_multiplication

## 8.1 Description

The `matrix_multiplication` function extends the dot product concept to operate between two vectors or two matrices. For vectors (1D arrays), it directly computes the dot product. When given matrices (2D arrays), it applies the dot product across corresponding rows and columns to obtain the resulting matrix entries.

It represents the matrix multiplication operation for 1D and 2D arrays while explicitly disallowing the multiplication of tensors (3D arrays or higher).

# 9 Function: tensor_dot_product_along_axes

## 9.1 Description

The `tensor_dot_product_along_axes` function performs a dot product on two tensors along specified axes. This is similar to the operation provided by NumPy's `tensordot` function.

## 9.2 Validation

Validation is performed by comparing the output to NumPy's `tensordot` function, using tensors generated with the `MyRandom` class.
Example Usage :

```
rnd = MyRandom()
tensor1 = rnd.random(2, 2, 2)
tensor2 = rnd.random(2, 2, 2)
result = tensor_dot_product_along_axes(tensor1, tensor2, 1, 0)
```

For validation with NumPy:

```
import numpy as np
result_np = np.tensordot(tensor1, tensor2, axes=(1, 0))
assert np.array_equal(result, result_np)
```

# 10 Function: matrix_power

## 10.1 Description

The `matrix_power` function raises a square matrix to a specified integer power, similar to NumPy's `numpy.linalg.matrix_power` function.

## 10.2 Validation

The matrix power calculation is validated by comparing the output with that from NumPy's `matrix_power` function.
Example Usage :

```
rnd = MyRandom()
matrix = rnd.random(2, 2)
n = 3  # Raise the matrix to the 3rd power
result = matrix_power(matrix, n)
```

For validation with NumPy:

```
import numpy as np
result_np = np.linalg.matrix_power(matrix, n)
assert np.array_equal(result, result_np)
```

# 11 Function: kronecker_product

## 11.1 Description

The `kronecker_product` function calculates the Kronecker product of two matrices. It replicates the functionality of NumPy's `np.kron`.

## 11.2 Validation

The Kronecker product computation is validated by comparing the result with NumPy's `kron` function.
Example Usage :

```
rnd = MyRandom()
matrix1 = rnd.random(2, 2)
matrix2 = rnd.random(2, 2)
result = kronecker_product(matrix1, matrix2)
```

For validation with NumPy:

```
import numpy as np
result_np = np.kron(matrix1, matrix2)
assert np.array_equal(result, result_np)
```

# 12 Function: cholesky_decomposition

## 12.1 Description

The `cholesky_decomposition` function performs a Cholesky decomposition of a square matrix, which is the factorization of a Hermitian, positive-definite matrix into a lower triangular matrix and its conjugate transpose.

## 12.2 Validation

We compare the output of our custom Cholesky decomposition function to NumPy's `np.linalg.cholesky` function.
Example Usage :

```
rnd = MyRandom()
A = rnd.random(3, 3)  # Generate a 3x3 random matrix
L = cholesky_decomposition(A)
```

For validation with NumPy:

```
import numpy as np
L_np = np.linalg.cholesky(A)
assert np.allclose(L, L_np)
```

# 13 Function: norm

## 13.1 Description

The `norm` function calculates the Euclidean norm (also known as the L2 norm or the Frobenius norm for matrices) of a vector or a matrix.

### 13.2 Validation

The function's output for both vectors and matrices is validated against the corresponding NumPy functions `np.linalg.norm` for vectors and matrices.
Example Usage :

```
rnd = MyRandom()
v = rnd.random(4)  # Generate a random vector of size 4
norm_v = norm(v)
```

For validation with NumPy:

```
import numpy as np
norm_v_np = np.linalg.norm(v)
assert np.allclose(norm_v, norm_v_np)
```

## 14 Function: multiplicative_inverse

### 14.1 Description

The `multiplicative_inverse` function calculates the multiplicative inverse of a square matrix.

### 14.2 Validation

The inverse matrix produced by our function is validated against NumPy's `np.linalg.inv` function.
Example Usage :

```
rnd = MyRandom()
A = rnd.random(3, 3)  # Generate a 3x3 random matrix
A_inv = multiplicative_inverse(A)
```

For validation with NumPy:

```
import numpy as np
A_inv_np = np.linalg.inv(A)
assert np.allclose(A_inv, A_inv_np)
```

## 15 Function: condition_number

### 15.1 Description

The `condition_number` function computes the condition number of a matrix, which measures the sensitivity of the solution of a system of linear equations to errors in the data.

### 15.2 Validation

The condition number calculated by our function is compared to the condition number computed by NumPy's `np.linalg.cond` function.
Example Usage :

```
rnd = MyRandom()
A = rnd.random(3, 3)  # Generate a 3x3 random matrix
cond_A = condition_number(A)
```

For validation with NumPy:

```
import numpy as np
cond_A_np = np.linalg.cond(A)
assert np.allclose(cond_A, cond_A_np)
```

# 16 Function: determinant

## 16.1 Description

The `determinant` function calculates the determinant of a square matrix, a scalar value that is a function of the entries of the matrix and gives information about the matrix's invertibility and its eigenvalues.

## 16.2 Validation

The determinant computed by our function is validated against the determinant computed by NumPy's `np.linalg.det` function.
Example Usage :

```
rnd = MyRandom()
A = rnd.random(3, 3)  # Generate a 3x3 random matrix
det_A = determinant(A)
```

For validation with NumPy:

```
import numpy as np
det_A_np = np.linalg.det(A)
assert np.isclose(det_A, det_A_np)
```

# 17 Function: calculate_eigen

The function `calculate_eigen` compute the eigenvalues and eigenvectors of a $2 \times 2$ matrix. This function takes a single argument, `matrix`, which is a list of lists representing a $2 \times 2$ square matrix. The function first checks if the provided matrix is indeed a $2 \times 2$ matrix and returns an error message if not.

Once verified as a $2 \times 2$ matrix, the function proceeds to calculate the trace and determinant of the matrix, which are then used to compute the eigenvalues. If the discriminant from the calculation is negative, indicating complex eigenvalues, the function returns a message stating that the eigenvalues are complex and that it does not compute eigenvectors for complex eigenvalues.

In cases where eigenvalues are real, the function computes both eigenvalues using the standard quadratic formula. The corresponding eigenvectors are determined depending on the structure of the matrix. Special cases, such as a diagonal matrix or matrices with zeros in certain positions, are handled separately to ensure correct computation of eigenvectors. If eigenvectors are found, they are normalized before being returned.

The function returns the calculated eigenvalues and eigenvectors of the input matrix.

This is an examlpe of usage :

```
# Initialize the custom random number generator
rnd = MyRandom()
# A = rnd.random(2, 2)  # Generate a random 2x2 matrix
# Calculate eigenvalues and eigenvectors for the square matrix A
# result = calculate_eigen(B)

# The result would contain eigenvalues and eigenvectors for matrix A.
```