rust_num_lib

A scientific computing library written in Rust, with Python bindings.

Modules: **linalg** (matrices, factorizations, solving) and **stats** (descriptive statistics, sampling).

```
from rust_num_lib import linalg, stats

a = linalg.matrix([[2, 1], [1, 3]])
b = linalg.vector([1, 2])
x = linalg.solve(a, b)
print(stats.mean(x))
```

rust_num_lib.linalg

Linear algebra module: matrices, factorizations, eigenvalues, solving systems.

class Matrix

A dense row-major matrix of f64 values.

__init__

```
def __init__(self, rows: int, cols: int) -> None
```

Create a new zero-filled matrix of shape (rows, cols).

zeros

```
@staticmethod
def zeros(rows: int, cols: int) -> 'Matrix'
```

Create a (rows x cols) matrix filled with zeros.

identity

```
@staticmethod
def identity(size: int) -> 'Matrix'
```

Create a (size x size) identity matrix.

random

```
@staticmethod
def random(rows: int, cols: int, min: float, max: float) -> 'Matrix'
```

Create a (rows x cols) matrix with values drawn uniformly from [min, max).

from_data

```
@staticmethod
def from_data(data: list[float], rows: int, cols: int) -> 'Matrix'
```

Create a matrix from a flat row-major list. len(data) must equal rows * cols.

rows

```
@property
def rows(self) -> int
```

Number of rows.

cols

```
@property
def cols(self) -> int
```

Number of columns.

transpose

```
def transpose(self) -> 'Matrix'
```

Return the transpose as a new matrix.

determinant

```
def determinant(self) -> float
```

Determinant of a square matrix (via LU factorization).

trace

```
def trace(self) -> float
```

Sum of the diagonal elements.

norm

```
def norm(self) -> float
```

Frobenius norm: sqrt of the sum of squared elements.

rank

```
def rank(self) -> int
```

Rank: number of non-zero pivots from LU factorization.

eigenvalues

```
def eigenvalues(self, steps: Optional[int]=None) -> list[float]
```

Eigenvalues, approximated with the QR algorithm.

steps — number of QR iterations (default: built-in value).

return — list of eigenvalues.

eigenvectors

```
def eigenvectors(self, steps: Optional[int]=None) -> tuple[list[float], list['Matrix']]
```

Eigenvalues and eigenvectors.

steps — number of QR iterations.

return — tuple (eigenvalues, eigenvectors) where each eigenvector is a column Matrix.

row

```
def row(self, i: int) -> 'Matrix'
```

Return row i as a (1 x cols) matrix.

col

```
def col(self, i: int) -> 'Matrix'
```

Return column i as a (rows x 1) matrix.

slice

```
def slice(self, row_range: Optional[tuple[int, int]]=None, row_step: Optional[int]=None, col_range
```

Extract a sub-matrix with optional (start, end) ranges and steps per axis.

Negative steps reverse the axis. None selects the full axis.

apply_along_axis

```
def apply_along_axis(self, f: 'callable', axis: int) -> 'Matrix'
```

Apply f to each row or column, reducing it to a single value.

1 applies f to each row (result rows x 1).

f — callable taking a Matrix (row/column) and returning a float.

axis — 0 applies f to each column (result 1 x cols),

__matmul__

```
def __matmul__(self, other: 'Matrix') -> 'Matrix'
```

Matrix multiplication (the @ operator).

__add__

```
def __add__(self, other: 'Matrix') -> 'Matrix'
```

Element-wise matrix addition.

__mul__

```
def __mul__(self, scalar: float) -> 'Matrix'
```

Scalar multiplication (matrix * scalar).

__rmul__

```
def __rmul__(self, scalar: float) -> 'Matrix'
```

Scalar multiplication (scalar * matrix).

__getitem__

```
def __getitem__(self, idx: tuple[int, int]) -> float
```

Element access: m[i, j].

__setitem__

```
def __setitem__(self, idx: tuple[int, int], value: float) -> None
```

Element assignment: m[i, j] = value.

matrix

```
def matrix(data: list[list[float]]) -> Matrix
```

Build a matrix from a 2D list, inferring dimensions from its shape.

All rows must have the same length.

vector

```
def vector(data: list[float]) -> Matrix
```

Build a column vector (n x 1 matrix) from a flat list.

matmul_chained

```
def matmul_chained(matrices: list[Matrix]) -> Matrix
```

Multiply a chain of matrices using an optimal multiplication order.

solve

```
def solve(a: Matrix, b: Matrix) -> Matrix
```

Solve the linear system A x = b for x.

solve_multiple

```
def solve_multiple(a: Matrix, bs: list[Matrix]) -> list[Matrix]
```

Solve A x = b for multiple right-hand sides, factorizing A once.

qr_factorize

```
def qr_factorize(a: Matrix) -> tuple[Matrix, Matrix]
```

QR factorization. Returns (Q, R) with Q orthonormal, R upper triangular, A = Q R.

lu_factorize

```
def lu_factorize(a: Matrix) -> tuple[Matrix, Matrix, list[int]]
```

LU factorization with partial pivoting. Returns (L, U, perm) such that P A = L U.

cholesky

```
def cholesky(a: Matrix) -> Matrix
```

Cholesky factorization of a symmetric positive-definite matrix.

Returns lower-triangular L such that A = L L■.

rust_num_lib.stats

Statistics module: descriptive statistics and sampling from distributions.

mean

```
def mean(m: Matrix) -> float
```

Arithmetic mean of all elements in the matrix.

median

```
def median(m: Matrix) -> float
```

Median of all elements in the matrix.

std_dev

```
def std_dev(m: Matrix) -> float
```

Standard deviation of all elements in the matrix.

variance

```
def variance(m: Matrix) -> float
```

Variance of all elements in the matrix.

normal_sample

```
def normal_sample(n: int, mu: float, sigma: float) -> Matrix
```

Draw n samples from a normal distribution N(mu, sigma^2).

return — an (n x 1) column vector of samples.

multivariate_normal_sample

```
def multivariate_normal_sample(n_samples: int, mu: Matrix, sigma: Matrix) -> Matrix
```

Draw samples from a multivariate normal distribution N(mu, Sigma).

mu — mean vector.

sigma — covariance matrix (symmetric positive-definite).

return — an (n_samples x dim) matrix where each row is one sample.