

NumCore — C++ Numerical Computing Library

NumCore is a header-only C++ numerical computing library exposed to Python via pybind11. It provides linear algebra, ODE solvers, and optimization algorithms — designed to be fast, dependency-light, and easy to use from both C++ and Python.

Table of Contents

1. [Project Structure](#)
 2. [Installation](#)
 - [Requirements](#)
 - [Python \(pip install\)](#)
 - [Manual CMake Build](#)
 3. [Quick Start](#)
 4. [Matrix & Vector](#)
 - [Matrix](#)
 - [Vector](#)
 5. [Linear Algebra](#) — `linalg`
 6. [ODE Solvers](#) — `ode`
 7. [Optimization](#) — `optim`
 8. [C++ API](#)
 9. [Running Tests](#)
 10. [NumPy Interoperability](#)
-

Project Structure

```
NumCore/
├── include/
│   └── NumCore/
│       ├── NumCore.hpp      ← umbrella header (include this in your C++ code)
│       ├── matrix.hpp
│       ├── vector.hpp
│       ├── linalg.hpp
│       ├── ode.hpp
│       └── optim.hpp
├── python/
│   └── bindings.cpp          ← pybind11 bindings (sole translation unit)
└── tests/
```

```
|   |─ tests_cpp.cpp           ← C++ unit tests (30 tests)
|   |─ tests_python.py        ← Python pytest suite (24 tests)
|   └─ CMakeLists.txt
|   └─ setup.py / pyproject.toml
```

Installation

Requirements

Dependency	Version
C++ compiler	C++17 or later (GCC 9+, Clang 10+, MSVC 2019+)
CMake	3.15+
Python	3.8+
pybind11	2.10+
NumPy	Any recent version (for Python usage)

Install pybind11 if not already installed:

```
bash

pip install pybind11
```

Python (pip install)

The recommended way — builds the extension and installs it into your Python environment:

```
bash

git clone https://github.com/yourusername/NumCore.git
cd NumCore
pip install .
```

Then import it:

```
python

from numcore import Matrix, Vector, linalg, ode, optim
```

Manual CMake Build

Use this if you want to build the C++ test binary alongside the Python module.

```
bash

git clone https://github.com/yourusername/NumCore.git
cd NumCore
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
cmake --build . --config Release
```

This produces:

- `_NumCore*.so` (Linux/macOS) or `_NumCore*.pyd` (Windows) — the Python extension
- `tests_cpp` — the C++ test binary

To use the built extension without installing:

```
bash

# From the build directory
python -c "import sys; sys.path.insert(0, '.'); import _NumCore"
```

Or run via `pip install .` from the repo root after building.

Quick Start

```
python
```

```

from numcore import Matrix, Vector, linalg, ode, optim
import numpy as np

# Solve a linear system Ax = b
A = Matrix(2, 2, [2.0, 1.0, 5.0, 7.0])
b = Vector([11.0, 13.0])
x = linalg.solve(A, b)
print(x)  # Vector([7.11, -3.22])

# Minimize f(x, y) = x^2 + y^2
f = lambda v: v[0]**2 + v[1]**2
g = lambda v: Vector([2.0*v[0], 2.0*v[1]])
result = optim.gradient_descent(f, g, Vector([5.0, 3.0]))
print(result.x)  # Vector(~[0, 0])

# Solve dy/dt = -y, y(0) = 1
decay = lambda t, y: Vector([-y[0]])
sol = ode.rk4(decay, 0.0, 5.0, Vector([1.0]), dt=0.01)
print(sol.y[-1][0])  # ≈ e^{-5} ≈ 0.00674

```

Matrix & Vector

Matrix

Construction

python

```
from numcore import Matrix
import numpy as np

# Zero matrix
m = Matrix.zeros(3, 4)

# All-ones matrix
m = Matrix.ones(2, 2)

# Identity matrix
I = Matrix.eye(4)

# From flat data (row-major order)
A = Matrix(2, 2, [1.0, 2.0, 3.0, 4.0])
# Represents:
# | 1  2 |
# | 3  4 |

# From a 2D NumPy array
arr = np.array([[1.0, 2.0], [3.0, 4.0]])
A = Matrix(arr)
```

Element Access

python

```
val = A[(0, 1)]      # get element at row 0, col 1
A[(1, 0)] = 9.0      # set element
```

Properties

python

```
A.rows()             # number of rows
A.cols()              # number of columns
A.shape()             # (rows, cols) tuple
```

Arithmetic

python

```
C = A + B          # element-wise addition
C = A - B          # element-wise subtraction
C = A * 2.0        # scalar multiplication
C = 2.0 * A        # scalar multiplication (right)
C = A / 2.0        # scalar division
C = -A            # negation
C = A @ B          # matrix multiplication (also A.matmul(B))
T = A.T()          # transpose
```

Utility

```
python

A.sum()            # sum of all elements
A.max()            # maximum element
A.min()            # minimum element
A.norm_frobenius() # Frobenius norm
A.apply(func)      # apply a Python callable element-wise
A.row(i)           # get row i as Vector
A.col(j)           # get col j as Vector
A.numpy()          # convert to 2D numpy array
```

Vector

Construction

```
python

from numcore import Vector
import numpy as np

v = Vector(3)          # zero vector of size 3
v = Vector(3, 5.0)     # vector filled with 5.0
v = Vector([1.0, 2.0, 3.0]) # from Python list
v = Vector(np.array([1.0, 2.0])) # from 1D NumPy array
```

Element Access

```
python

val = v[0]           # get element
v[1] = 7.0           # set element
len(v)               # number of elements
v.size()             # same as len(v)
```

Arithmetic

```
python

w = u + v          # element-wise add
w = u - v          # element-wise subtract
w = v * 2.0        # scalar multiply
w = 2.0 * v        # scalar multiply (right)
w = v / 2.0        # scalar divide
w = -v            # negation
```

Operations

```
python

v.dot(u)           # dot product
v.norm()           # L2 norm (Euclidean length)
v.normalized()      # unit vector (returns new Vector)
v.cross(u)         # cross product (3D vectors only)
v.sum()            # sum of elements
v.mean()           # mean of elements
v.max()            # maximum element
v.min()            # minimum element
v.apply(func)      # apply Python callable element-wise
v.as_column()      # convert to column Matrix (n×1)
v.as_row()         # convert to row Matrix (1×n)
v.numpy()          # convert to 1D NumPy array
```

matvec — Matrix-Vector Multiply

```
python

from numcore import matvec

result = matvec(A, v)  # equivalent to A @ v, returns Vector
```

Linear Algebra — **linalg**

```
python

from numcore import linalg
```

linalg.solve(A, b) — Solve linear system

Solves $A @ x = b$ using LU decomposition with partial pivoting.

python

```
A = Matrix(2, 2, [2.0, 1.0, 5.0, 7.0])
b = Vector([11.0, 13.0])

x = linalg.solve(A, b)
# Raises RuntimeError if A is singular
```

Raises: `RuntimeError` if the matrix is singular (pivot below threshold).

`linalg.lu(A)` — LU Decomposition

Returns an `LUResult` with fields `L`, `U`, `perm`, `sign`.

python

```
res = linalg.lu(A)

res.L      # lower triangular Matrix (unit diagonal)
res.U      # upper triangular Matrix
res.perm   # permutation list (row indices)
res.sign   # determinant sign (+1 or -1)
```

`linalg.det(A)` — Determinant

python

```
d = linalg.det(A)    # returns float
```

`linalg.inv(A)` — Matrix Inverse

python

```
A_inv = linalg.inv(A)    # returns Matrix

# Verify: A @ inv(A) ≈ I
import numpy as np
np.testing.assert_allclose((A @ A_inv).numpy(), np.eye(2), atol=1e-8)
```

Raises: `RuntimeError` if `A` is singular.

Solves $\text{argmin} \|A @ x - b\|_2$ using the normal equations $(A^T A) x = A^T b$. Suitable for overdetermined systems (more rows than columns).

```
python

# Fit y = a*x + c through 4 points
A = Matrix(4, 2, [1.0, 1.0,
                  2.0, 1.0,
                  3.0, 1.0,
                  4.0, 1.0])
y = Vector([3.0, 5.0, 7.0, 9.0])

coeff = linalg.lstsq(A, y)
# coeff[0] = slope ≈ 2.0
# coeff[1] = intercept ≈ 1.0
```

`linalg.gauss_jordan(A, b)` — Gauss-Jordan Solver

Alternative direct solver using Gauss-Jordan elimination.

```
python

x = linalg.gauss_jordan(A, b)
```

`linalg.norm(v)` — Vector Norm

```
python

n = linalg.norm(v)    # same as v.norm()
```

`linalg.norm_frobenius(A)` — Frobenius Norm

```
python

n = linalg.norm_frobenius(A)    # same as A.norm_frobenius()
```

ODE Solvers — `ode`

```
python
```

```
from numcore import ode
```

All solvers accept a right-hand-side function `f(t, y) -> Vector` and return an `ODESolution`.

`ODESolution` fields:

```
python

sol.t          # list of time points
sol.y          # list of state vectors (as lists of floats)
sol.t_array()  # sol.t as 1D NumPy array
sol.y_array()  # sol.y as 2D NumPy array, shape (n_steps, n_vars)
```

`ode.euler(f, t0, t_end, y0, dt)` — Euler Method

First-order fixed-step method. Fast but low accuracy.

```
python

def f(t, y):
    return Vector([-y[0]])    # dy/dt = -y

sol = ode.euler(f, t0=0.0, t_end=1.0, y0=Vector([1.0]), dt=0.001)

print(sol.y[-1][0])    # ≈ e^{-1} ≈ 0.368
```

`ode.rk4(f, t0, t_end, y0, dt)` — Classic 4th-Order Runge-Kutta

High accuracy for smooth problems with fixed step size. Recommended for most use cases.

```
python

sol = ode.rk4(f, t0=0.0, t_end=5.0, y0=Vector([1.0]), dt=0.01)

import math
print(sol.y[-1][0])    # ≈ e^{-5} ≈ 0.00674 (error < 1e-4)
```

Multi-variable example — harmonic oscillator:

```
python
```

```
def harmonic(t, y):
    # y[0] = position, y[1] = velocity
    #  $d^2x/dt^2 + x = 0 \rightarrow [x', v'] = [v, -x]$ 
    return Vector([y[1], -y[0]])

import math
sol = ode.rk4(harmonic, 0.0, 2*math.pi, Vector([1.0, 0.0]), dt=0.01)

# Energy should be conserved:  $x^2 + v^2 = 1$ 
x, v = sol.y[-1][0], sol.y[-1][1]
print(x**2 + v**2)    #  $\approx 1.0$ 
```

`ode.rk45(f, t0, t_end, y0, opts)` — Adaptive Dormand-Prince RK45

Adaptive step-size control using embedded 4th/5th order error estimation. Automatically adjusts step size to meet error tolerances. Best for stiff or high-accuracy problems.

```
python

# Default options
sol = ode.rk45(f, t0=0.0, t_end=5.0, y0=Vector([1.0]))

# Custom options
opts = ode.RK45Options()
opts.rtol      = 1e-6    # relative tolerance (default: 1e-3)
opts.atol      = 1e-9    # absolute tolerance (default: 1e-6)
opts.dt_init   = 0.01    # initial step size (default: 0.01)
opts.dt_min    = 1e-8    # minimum step size (default: 1e-8)
opts.dt_max    = 0.1     # maximum step size (default: 1.0)
opts.max_steps = 100000  # step limit (default: 100000)

sol = ode.rk45(f, 0.0, 5.0, Vector([1.0]), opts)
print(sol.y[-1][0])    #  $\approx e^{-5}$ , error < 1e-5
```

`ode.extract_component(sol, index)` — Extract One Variable

Extracts a single component from a multi-variable solution as a list.

```
python
```

```
sol = ode.rk4(harmonic, 0.0, 10.0, Vector([1.0, 0.0]), 0.01)

positions = ode.extract_component(sol, 0) # x(t)
velocities = ode.extract_component(sol, 1) # v(t)
```

Plotting example:

```
python

import numpy as np
import matplotlib.pyplot as plt

t = sol.t_array()
y = sol.y_array()

plt.plot(t, y[:, 0], label="position")
plt.plot(t, y[:, 1], label="velocity")
plt.legend()
plt.show()
```

Optimization — `optim`

```
python

from numcore import optim
```

All optimizers return an `OptimResult`:

```
python

res.x          # Vector — solution found
res.f_val      # float — function value at solution
res.iterations # int — number of iterations taken
res.converged  # bool — True if convergence criterion was met
res.message    # str — status message
```

`optim.gradient_descent(f, grad, x0, opts)` — Gradient Descent

Fixed learning-rate gradient descent. Works well on smooth, convex objectives.

```
python
```

```
f      = lambda v: v[0]**2 + 4.0*v[1]**2
grad = lambda v: Vector([2.0*v[0], 8.0*v[1]])

opts = optim.GDOptions()
opts.lr      = 0.1      # learning rate (default: 0.01)
opts.tol     = 1e-6     # gradient norm stopping threshold (default: 1e-6)
opts.max_iter = 10000   # iteration limit (default: 10000)

res = optim.gradient_descent(f, grad, Vector([5.0, 3.0]), opts)
print(res.x)      # ≈ [0, 0]
print(res.converged) # True
```

`optim.adam(f, grad, x0, opts)` — Adam Optimizer

Adaptive moment estimation. Robust to noisy gradients, fast convergence on deep learning-style objectives.

```
python

f      = lambda v: v[0]**2 + v[1]**2
grad = lambda v: Vector([2.0*v[0], 2.0*v[1]])

opts = optim.AdamOptions()
opts.lr      = 0.001   # learning rate (default: 0.001)
opts.beta1   = 0.9     # 1st moment decay (default: 0.9)
opts.beta2   = 0.999   # 2nd moment decay (default: 0.999)
opts.eps     = 1e-8     # numerical stability (default: 1e-8)
opts.tol     = 1e-6     # convergence threshold (default: 1e-6)
opts.max_iter = 10000   # iteration limit (default: 10000)

res = optim.adam(f, grad, Vector([5.0, -3.0]))
print(res.x)      # ≈ [0, 0]
```

`optim.newton(f, x0, opts)` — Newton's Method

Uses a numerically computed Hessian for second-order convergence. Very fast near the optimum. Requires the objective to be locally convex.

```
python
```

```
f = lambda v: v[0]**2 + v[1]**2

opts = optim.NewtonOptions()
opts.tol      = 1e-8    # gradient norm threshold (default: 1e-8)
opts.max_iter = 100     # iteration limit (default: 100)
opts.h        = 1e-5    # finite-difference step for Hessian (default: 1e-5)
opts.lambda_  = 1e-4    # Levenberg-Marquardt regularization (default: 1e-4)

res = optim.newton(f, Vector([3.0, 4.0]), opts)
print(res.x)          # ≈ [0, 0]
print(res.iterations) # typically < 10
```

Note: `lambda_` adds a small regularization (λI) to the Hessian to handle near-singular cases.

`optim.bfgs(f, grad, x0, opts)` — BFGS with Armijo Line Search

Quasi-Newton method with BFGS Hessian approximation. Strong convergence on smooth non-convex objectives. The recommended method for general use.

```
python

# Rosenbrock function: minimum at (1, 1)
def f(v):
    x, y = v[0], v[1]
    return (1 - x)**2 + 100*(y - x**2)**2

def grad(v):
    x, y = v[0], v[1]
    return Vector([
        -2*(1 - x) - 400*x*(y - x**2),
        200*(y - x**2)
    ])

opts = optim.BFGSOptions()
opts.tol      = 1e-6    # gradient norm threshold (default: 1e-6)
opts.max_iter = 1000    # iteration limit (default: 1000)

res = optim.bfgs(f, grad, Vector([0.0, 0.0]), opts)
print(res.x)    # ≈ [1.0, 1.0]
```

`optim.numerical_grad(f, x, eps)` — Numerical Gradient

Computes the gradient of f at x using central finite differences. Useful when an analytical

gradient is not available.

python

```
f = lambda v: v[0]**2 + v[1]**2

x = Vector([3.0, 4.0])
g = optim.numerical_grad(f, x, eps=1e-7) # eps is optional, default 1e-7
# g ≈ [6.0, 8.0]
```

You can combine it with any optimizer:

python

```
grad = lambda v: optim.numerical_grad(f, v)
res = optim.bfgs(f, grad, Vector([0.0, 0.0]))
```

C++ API

Include the umbrella header in your C++ project:

cpp

```
#include "NumCore/NumCore.hpp"
using namespace NumCore;
```

The API mirrors the Python interface closely.

Matrix & Vector

cpp

```

Matrix A(2, 2, 0.0);           // 2x2 zero matrix
A(0, 0) = 1.0; A(0, 1) = 2.0;
A(1, 0) = 3.0; A(1, 1) = 4.0;

Matrix B = A + A;
Matrix C = A.matmul(A);
Matrix T = A.T();
double d = A.sum();

Vector v{1.0, 2.0, 3.0};
double n = v.norm();
double dp = v.dot(v);
Vector cross = v.cross(Vector{0.0, 1.0, 0.0});
Vector unit  = v.normalize();

```

Linear Algebra

```

cpp

Matrix A{{2, 1}, {5, 7}};
Vector b{11, 13};

Vector x      = linalg::solve(A, b);
double det    = linalg::det(A);
Matrix inv    = linalg::inv(A);
Vector coeff  = linalg::lstsq(A, b);

auto lu_res = linalg::lu(A);
// lu_res.L, lu_res.U, lu_res.perm, lu_res.sign

```

ODE Solvers

```

cpp

auto f = [](double t, const Vector& y) -> Vector {
    return Vector{-y[0]}; // dy/dt = -y
};

auto sol_euler = ode::euler(f, 0.0, 1.0, Vector{1.0}, 0.001);
auto sol_rk4   = ode::rk4 (f, 0.0, 5.0, Vector{1.0}, 0.01);
auto sol_rk45  = ode::rk45 (f, 0.0, 5.0, Vector{1.0});

double final_val = sol_rk4.y.back()[0]; // ≈ e^{-5}

```


Optimization

cpp

```
auto quad = [](const Vector& v) { return v[0]*v[0] + v[1]*v[1]; };
auto gd    = [](const Vector& v) { return v * 2.0; };

auto res_n      = optim::newton(quad, Vector{3.0, 4.0});
auto res_bfgs   = optim::bfgs(quad, gd, Vector{3.0, 4.0});
auto res_adam   = optim::adam(quad, gd, Vector{3.0, 4.0});

std::cout << res_bfgs.x[0] << "\n"; // ≈ 0.0
std::cout << res_bfgs.converged << "\n"; // 1
```

Running Tests

C++ Tests (30 tests)

bash

```
cd build
cmake .. && cmake --build .
./tests_cpp
```

Expected output:

```
=== numcpp C++ unit tests ===

[matrix]
  PASS  element access
  PASS  sum
  ...

Results: 30 passed, 0 failed
```

Python Tests (24 tests)

bash

```
pip install pytest numpy scipy
pytest tests/tests_python.py -v
```

Expected output:

```
tests/tests_python.py::TestMatrix::test_zeros PASSED
tests/tests_python.py::TestMatrix::test_eye PASSED
...
24 passed in X.XXs
```

NumPy Interoperability

All `Matrix` and `Vector` objects convert to/from NumPy with zero extra dependencies:

```
python

import numpy as np
from numcore import Matrix, Vector

# NumPy → NumCore
arr = np.array([[1.0, 2.0], [3.0, 4.0]])
M = Matrix(arr)          # 2D array → Matrix

v_arr = np.array([1.0, 2.0, 3.0])
v = Vector(v_arr)        # 1D array → Vector

# NumCore → NumPy
M.numpy()                 # Matrix → 2D ndarray
v.numpy()                 # Vector → 1D ndarray

# Validate against SciPy
import scipy.linalg
from numcore import linalg

A_np = np.array([[2., 1.], [5., 7.]])
b_np = np.array([11., 13.])

x_nc = linalg.solve(Matrix(A_np), Vector(b_np)).numpy()
x_sp = scipy.linalg.solve(A_np, b_np)

np.testing.assert_allclose(x_nc, x_sp, atol=1e-10) # passes
```

Choosing the Right Tool

Task	Recommended
Dense linear system (small-medium)	<code>linalg.solve</code>

Task	Recommended
Overdetermined / least-squares fit	<code>linalg.lstsq</code>
Smooth ODE, fixed accuracy needed	<code>ode.rk4</code>
Stiff ODE or high accuracy	<code>ode.rk45</code>
Convex smooth optimization	<code>optim.gradient_descent</code>
Noisy gradients / ML-style training	<code>optim.adam</code>
Smooth function, fast convergence	<code>optim.bfgs</code>
Near optimum, convex	<code>optim.newton</code>
No analytical gradient available	<code>optim.numerical_grad</code> + BFGS