C++ linear algebra: Eigen

Programming Concepts in Scientific Computing

EPFL, Master class

November 8, 2023

Compilation

```
cmake_minimum_required (VERSION 3.1)
project (Particles)

set(CMAKE_CXX_STANDARD 20)

include_directories(eigen)
add_executable(main main.cc)
```

what can you conclude ?

Declaring a vector

```
Eigen::VectorXd v(2);
```

Declaring a vector

```
Eigen::VectorXd v(2);
```

Accessing and using a vector

```
v(0) = 3;      // parentheses
v[1] += 2.5; // or square brackets

double res = v.transpose() * v;
auto norm = v.norm();
```

Eigen objects have two dimensions

Eigen objects have two dimensions

```
// access to shape
std::cout << "nb_rows: " << v.rows() << std::endl;
std::cout << "nb_cols: " << v.cols() << std::endl;
std::cout << "size: " << v.size() << std::endl;
```

Eigen objects have two dimensions

```
// access to shape
std::cout << "nb_rows: " << v.rows() << std::endl;
std::cout << "nb_cols: " << v.cols() << std::endl;
std::cout << "size: " << v.size() << std::endl;
```

```
// change size
v.resize(10);
```

Vector types

```
Eigen::VectorXi v_int(2);
Eigen::VectorXf v_float(2);
Eigen::VectorXcd v_complex_double(2);
auto real_part = v_complex_double.real();
```

Initialization of vectors

```cpp
int N = 5;
Eigen::VectorXd v(N);

// v = np.array([1,2,3,4,5])
v << 1, 2, 3, 4, 5;
// v = np.random.random(N)
v = Eigen::VectorXd::Random(N);
// v = np.random.random(N)
v = Eigen::VectorXd::LinSpaced(N, -1., 1.);
// v = np.zeros(N)
v = Eigen::VectorXd::Zero(N);
// v = np.ones(N)
v = Eigen::VectorXd::Ones(N);
// v = np.ones(N)*42
v = Eigen::VectorXd::Constant(N, 42.);
v = Eigen::VectorXd::Ones(N) * 42.;
```

Matrices

```
Eigen::MatrixXd m(3, 3);

// access to shape
std::cout << "nb_rows: " << m.rows() << std::endl;
std::cout << "nb_cols: " << m.cols() << std::endl;
std::cout << "size: " << m.size() << std::endl;
std::cout << "m = " << m << std::endl;
std::cout << "m.T = " << m.transpose() << std::endl;

// resize
m.resize(10, 2);
```

Other types

```
Eigen::MatrixXcd m_complex_double(3, 3);
Eigen::MatrixXi m_int(3, 3);
```

Dynamic size

```cpp
// variable size => dynamic allocation (of course!)
Eigen::VectorXd v_undefined_size;

// In Eigen, this is equivalent to
Eigen::Matrix<double, -1, 1> v_undefined_size2;
```

Dynamic size

```cpp
// variable size => dynamic allocation (of course!)
Eigen::VectorXd v_undefined_size;

// In Eigen, this is equivalent to
Eigen::Matrix<double, -1, 1> v_undefined_size2;
```

How is a Eigen matrix defined ?

Static(fixed) size

```
Eigen::Vector4d v4; // Eigen::Matrix<double, 4, 1>
Eigen::Vector3d v3; // Eigen::Matrix<double, 3, 1>
Eigen::Vector2d v2; // Eigen::Matrix<double, 2, 1>
Eigen::Matrix3d m;  // Eigen::Matrix<double, 3, 3>
```

Static(fixed) size

```
Eigen::Vector4d v4; // Eigen::Matrix<double, 4, 1>
Eigen::Vector3d v3; // Eigen::Matrix<double, 3, 1>
Eigen::Vector2d v2; // Eigen::Matrix<double, 2, 1>
Eigen::Matrix3d m;  // Eigen::Matrix<double, 3, 3>
```

- is static allocation helpful/necessary ?

Static(fixed) size

```
Eigen::Vector4d v4; // Eigen::Matrix<double, 4, 1>
Eigen::Vector3d v3; // Eigen::Matrix<double, 3, 1>
Eigen::Vector2d v2; // Eigen::Matrix<double, 2, 1>
Eigen::Matrix3d m;  // Eigen::Matrix<double, 3, 3>
```

- is static allocation helpful/necessary ?
- what are the limitations ?

Linear Algebra

```
Eigen::Vector3d v = Eigen::Vector3d::Ones();
Eigen::Matrix3d m = Eigen::Vector3d::Constant(3).asDiagon

// matrix vector multiplication
auto res = m * v;

// determinent
auto det = m.determinant();
// inversion
auto inv = m.inverse();
// norm
auto norm = m.norm();
```

Linear system solve

```
Eigen::Matrix3d m = Eigen::Vector3d::Constant(3).asDiagon
// matrix LU factorization
auto facto = m.fullPivLu();
// system resolution
auto res2 = facto.solve(res);
```

Slices, blocks and memory management

What if we want to store data with:

```
std::vector<double> v{1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Slices, blocks and memory management

What if we want to store data with:

```
std::vector<double> v{1, 2, 3, 4, 5, 6, 7, 8, 9};
```

can we use eigen then ?

Eigen::Map - Wrapping the memory

TBD

Idea: Associate Eigen structure with a pointer

Eigen::Map - Wrapping the memory

Idea: Associate Eigen structure with a pointer

From a std::vector v

```
Eigen::Map<Eigen::Matrix3d> m(&v[0]);
```

Eigen::Map - Wrapping the memory

Idea: Associate Eigen structure with a pointer

From a std::vector v

```
Eigen::Map<Eigen::Matrix3d> m(&v[0]);
```

From a C-array/pointer

```
auto *ptr = new double[9];
Eigen::Map<Eigen::Matrix3d> m_ptr(ptr);
```

Eigen::Map - Wrapping the memory

Idea: Associate Eigen structure with a pointer

From a std::vector v

```
Eigen::Map<Eigen::Matrix3d> m(&v[0]);
```

From a C-array/pointer

```
auto *ptr = new double[9];
Eigen::Map<Eigen::Matrix3d> m_ptr(ptr);
```

From a std::array

```
std::array<double, 9> array;
Eigen::Map<Eigen::Matrix3d> m_array(&array[0]);
```

Eigen::Map - Reshaping

Idea: Keep the pointer and change the structure

Eigen::Map - Reshaping

Idea: Keep the pointer and change the structure

Same to what happens in Python

Let us define a matrix

```
Eigen::MatrixXd m(3, 3);
m << 1, 2, 3, 4, 5, 6, 7, 8, 9;
```

Eigen::Map - Reshaping

Idea: Keep the pointer and change the structure

Let us define a matrix

```
Eigen::MatrixXd m(3, 3);
m << 1, 2, 3, 4, 5, 6, 7, 8, 9;
```

and shape it as a vector

```
// m_reshape = m.reshape(9)
Eigen::Map<Eigen::VectorXd> m_reshape(m.data(), m.
```

Eigen::Map - Reshaping

Or the reverse...

```
Eigen::Matrix<double, 3, 3, Eigen::RowMajor> m2;
m2 << 1, 2, 3, 4, 5, 6, 7, 8, 9;

Eigen::Map<Eigen::VectorXd> m2_reshape(m2.data(), m2.size
```

Eigen::Map - Reshaping

Or the reverse...

```
Eigen::Matrix<double, 3, 3, Eigen::RowMajor> m2;
m2 << 1, 2, 3, 4, 5, 6, 7, 8, 9;

Eigen::Map<Eigen::VectorXd> m2_reshape(m2.data(), m2.size
```

Eigen::Matrix::data() method returns the raw pointer

Blocks and slicing

Extraction of block matrices

```
Eigen::ArrayXXd a(20, 2);
```

Blocks and slicing

Extraction of block matrices

```
Eigen::ArrayXXd a(20, 2);
```

Static/Dynamic Extraction

```
auto submatrix_static = a.block<2, 2>(0, 0);
auto submatrix_dynamic = a.block(0, 0, 2, 2);
```

Blocks and slicing

Extraction of block matrices

```
Eigen::ArrayXXd a(20, 2);
```

Static/Dynamic Extraction

```
auto submatrix_static = a.block<2, 2>(0, 0);
auto submatrix_dynamic = a.block(0, 0, 2, 2);
```

What is the returned type ?

Blocks and slicing

Extraction of block matrices

```
Eigen::ArrayXXd a(20, 2);
```

Should be matrix

Static/Dynamic Extraction

```
auto submatrix_static = a.block<2, 2>(0, 0);
auto submatrix_dynamic = a.block(0, 0, 2, 2);
```

If you want to make a copy, specify the type instead of using auto

What is the returned type ?

Q: how to make a copy? A: use copy constructor

Slicing:

https://eigen.tuxfamily.org/dox/group__TutorialSlicingIndexing.html

Sparse matrices

Declaration

```
Eigen::SparseMatrix<double> mat(3, 3);
```

Sparse matrices

Declaration

```
Eigen::SparseMatrix<double> mat(3, 3);
```

setting coefficients

```
mat.coeffRef(i, j) = val;  // set a coefficient
```

Sparse matrices

Declaration

```
Eigen::SparseMatrix<double> mat(3, 3);
```

setting coefficients

```
mat.coeffRef(i, j) = val;  // set a coefficient
```

from a diagonal matrix

```
// set the sparse matrix as a diagonal
mat = Eigen::Vector3d::Constant(3).asDiagonal();
mat.makeCompressed();
```

Sparse matrices

Declaration

```
Eigen::SparseMatrix<double> mat(3, 3);
```

setting coefficients

```
mat.coeffRef(i, j) = val;  // set a coefficient
```

from a diagonal matrix

```
// set the sparse matrix as a diagonal
mat = Eigen::Vector3d::Constant(3).asDiagonal();
mat.makeCompressed();
```

Solving a linear system

```
Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
```

Sparse matrices

Declaration

```
Eigen::SparseMatrix<double> mat(3, 3);
```

setting coefficients

```
mat.coeffRef(i, j) = val;  // set a coefficient
```

from a diagonal matrix

```
// set the sparse matrix as a diagonal
mat = Eigen::Vector3d::Constant(3).asDiagonal();
mat.makeCompressed();
```

Solving a linear system

```
Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;

auto x = solver.solve(v);
```

Arrays ($\neq$ Vectors)

- Arrays do per-components operations

```
Eigen::ArrayXXd array(100, 3);
Eigen::ArrayX3d array2(100, 3);

auto res1 = array * array2; // component-wise operation
auto res2 = array - array2; // component-wise operation
auto res3 = array.col(0).sin();
auto res4 = array.col(0).sqrt();
auto res5 = array.col(0).exp();
auto res6 = array.col(0).tanh();
```

Arrays ($\neq$ Vectors)

- Arrays do per-components operations

```
Eigen::ArrayXXd array(100, 3);
Eigen::ArrayX3d array2(100, 3);

auto res1 = array * array2; // component-wise operation
auto res2 = array - array2; // component-wise operation
auto res3 = array.col(0).sin();
auto res4 = array.col(0).sqrt();
auto res5 = array.col(0).exp();
auto res6 = array.col(0).tanh();

// converting to matrix
Eigen::MatrixXd res7 = array.matrix();
// back to array
Eigen::ArrayXXd res8 = res7.array();
```

Documentation

`https://eigen.tuxfamily.org/dox/`

Documentation

https://eigen.tuxfamily.org/dox/

Array documentation:

https://eigen.tuxfamily.org/dox/group__TutorialArrayClass.html

Documentation

https://eigen.tuxfamily.org/dox/

Array documentation:

https://eigen.tuxfamily.org/dox/group__TutorialArrayClass.html

Matrix documentation:

https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html

Take away message

- **Eigen** is a highly templated linear algebra library
- **Dynamic&Static** allocation of arrays is possible
- **Template parameters** are scalar-type, and sizes
- **Wrapping** of contiguous data into eigen arrays/matrices is possible
- **Slicing/Sub-blocks** extraction is possible
- **Arrays** have per-component operators
- **Matrices** have linear algebra operators
- Enables **mathematical readability** in C++ : use it!