Eigen

⇒ Eigen is a header only library.

* MatrixXd → (Entities are double)

⇒ It is in Eigen name space.

* Quick Reference Guide

⇒ The Eigen library is divided into a Core module and Several additional modules.

⇒ Each module has a corresponding header files which has to be included in order to use the module.

⇒ The Dense and Eigen header files are provided to Conveiently gain access to Several modules at once.

① Core
- → Matrix & Array classes
- → Basic linear algebra
- → array manipulation

② Geometry
- → Transform
- → Translation
- → Scaling
- → Rotation 2D and 3D rotation
  - → Quaternion.

(3) LU
- → Inverse
- → Determinant
- → LU decompositions

(4) Cholesky
- → Cholesky factorization

(5) Householder
- → Householder transform

(6) SVD
- → SVD decompositions with least-square solver.

(7) QR
- → QR decomposition with solver

(8) Eigenvalues
- → Eigen value, Eigen Vector decomposition

(9) Sparse
- → Sparse matrix storage and related basic linear algebra.

(10) Dense
- → Includes: Core, Geometry, LU, Cholesky SVD, QR and Eigenvalues.

(11) Eigen
- → Includes dense & Sparse headerfiles (The whole Eigen library)

* Array, matrix and vector types

⇒ Eigen provides two kinds of dense objects:
- Matrices
- Vectors

{ Both represented by the template class Matrix }

⇒ General 1D & 2D arrays represented by the template class Array.

```
typedef Matrix<Scalar, RowsAtCompileTime, ColsAtCompileTime
        ,options> MyMatrixType;
typedef Array<Scalar, RowsAtCompileTime, ColsAtCompileTime
        ,Options> MyArrayType;
```

Scalar → Eg float, double, bool, int etc.

⇒ RowsAtCompileTime & ColsAtCompileTime are the number of rows and columns of the matrix as known at the compile time or dynamic.

⇒ Options can be ColMajor or RowMajor, default is ColMajor.

⇒ Examples

① Matrix<double, 6, Dynamic>
↳ Dynamic number of columns (heap allocation)

② Matrix<double, Dynamic, Dynamic, RowMajor>
↳ Fully Dynamic, row major (heap allocation)

③ Matrix<double, Dynamic, 2>
↳ Dynamic number of rows (heap allocation)

④ Matrix<double, 13, 3>
↳ Fully fixed (usually allocated on stack)

⇒ In most cases, you can simply use one of the convenience typedefs for matrices and arrays.

Matrices

Matrix<float, Dynamic, Dynamic> ⇔ MatrixXf
Matrix<double, Dynamic, 1> ⇔ VectorXd
Matrix<int, 1, Dynamic> ⇔ RowVectorXi
Matrix<double, 3, 3> ⇔ Matrix3d
Matrix<float, 4, 1> ⇔ Vector4f

Arrays

Array<float, Dynamic, Dynamic> ⇔ ArrayXXf
Array<double, Dynamic, 1> ⇔ ArrayXd
Array<int, 1, Dynamic> ⇔ RowArrayXi
Array<float, 3, 3> ⇔ Array33f
Array<float, 4, 1> ⇔ Array4f

⇒ Conversion between the matrix and array worlds:

① Array44f a1, a2;

② Matrix4f m1, m2;

③ m1 = a1 * a2;

{ → Coeffwise product
→ Implicit conversion from array to matrix }

④ a1 = m1 * m2;

{ → Matrix product
→ Implicit Conversion from matrix to array }

⑤ a2 = a1 + m1.array();

⑥ m2 = a1.matrix() + m1;

{ → Mixing of array and matrix is forbidden
→ Explicit conversion is required }

⑦ ArrayWrapper<Matrix4f> m1a(m1);

{ → m1a is an alias for m1.array()
→ They share the same coefficients }

⑧ MatrixWrapper<Array44f> a1m(a1);

{ → a1m is an alias for a1.matrix()
→ They share the same coefficients }

Note

Linear Algebra ⇒ Matrix and Vector only

Object Only ⇐ Array

★ Basic matrix Manipulation

① By default, the coefficients are left uninitialized on Construction.

② Comma initializer

```
Vector3f v1;
v1 << x, y, z;
```

```
Matrix3f m1;
m1 << 1, 2, 3,
      4, 5, 6,
      7, 8, 9;
```

③ Comma initializer (Block initialization)

① int rows = 5, cols = 5;

② MatrixXf m(rows, cols);

③ m << (Matrix3f() << 1,2,3,4,5,6,7,8,9).finished(),
MatrixXf::Zeros(3, cols-3),
MatrixXf::Zeros(rows-3, 3),
MatrixXf::Identity(rows-3, cols-3);

④ Cout << m;

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \equiv \left[\begin{array}{ccc|cc} 1 & 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 \\ 7 & 8 & 9 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right]$$

m << A, B, C, D;

④ Coeff access with range checking

Vector(i)	Vector.x()	matrix(i,j)
Vector[i]	Vector.y()	
	Vector.z()	
	Vector.w()	

⑤ Coeff access without range checking

Vector.coeff(i)	matrix.coeff(i,j)
Vector.coeffRef(i)	matrix.coeffRef(i,j)

★ Predefined Matrices

① Fixed-Size matrix or Vector

```
typedef {Matrix3f | Array33f} FixedXD;
FixedXD x;

x = FixedXD::Zero();
x = FixedXD::Ones();
x = FixedXD::Constant(value);
x = FixedXD::Random();
x = FixedXD::LinSpaced(Size, low, high);
x.setZero();
x.setOnes();
x.setConstant(value);
x.setRandom();
x.setLinSpaced(size, low, high);
```

```
X = FixedXD::Identity();
X.setIdentity();

Vector3f::UnitX(); // 1 0 0
Vector3f::UnitY(); // 0 1 0
Vector3f::UnitZ(); // 0 0 1
```

(9) Dynamic-size matrix

```
typedef {MatrixXf | ArrayXXf} Dynamic2D;
Dynamic2D X;

X = Dynamic2D::Zero(rows, cols);
X = Dynamic2D::Ones(rows, cols);
X = Dynamic2D::Constant(rows, cols, value);
X = Dynamic2D::Random(rows, cols);
X.setZeros(rows, cols);
X.setOnes(rows, cols);
X.setConstant(rows, cols, Value);
X.setRandom(rows, cols);
X = Dynamic2D::Identity(rows, cols);
X.setIdentity(rows, cols);
```

③ Dynamic-size vector

```
typedef {VectorXf | ArrayXf} Dynamic1D;
Dynamic1D x;

X = Dynamic1D::Zeros(size);
X = Dynamic1D::Ones(size);
X = Dynamic1D::Constant(size, value);
X = Dynamic1D::Random(size);
X = Dynamic1D::LinSpaced(size, low, high);

X.setZeros(size);
X.setOnes(size);
X.setConstant(size, value);
X.setRandom(size);
X.setLinSpaced(size, low, high);

VectorXf::Unit(size, i);
VectorXf::Unit(4,1) ==
        Vector4f(0,1,0,0)
            ==
        Vector4f::UnitY()
```

★ Arithmetic Operations

① Add / Subtract

```
mat3 = mat1 ± mat2;
mat3 ±= mat1;
```

② Scalar product

```
mat3 = mat1 */ s1;
mat3 */= s1;
```

③ Matrix / Vector Products

```
Col2 = mat1 * Col1;
row2 = row1 * mat1;      row1 *= mat1;
mat3 = mat1 * mat2;      mat3 *= mat1;
```

④ Transpose, adjoint (Conjugate transpose)

```
mat1 = mat2.transpose();      mat1.transposeInPlace();
mat1 = mat2.adjoint();        mat1.adjointInPlace();
```

⑤ dot product / Inner Product

```
Scalar = vec1.dot(vec2);
Scalar = col1.adjoint() * col2;
Scalar = (col1.adjoint() * col2).value();
```

⑥ Outer product

```
mat = Col1 * Col2.transpose();
```

(7) Norm, Normalization

```
Scalar = vec1.norm();          | Scalar = vec1.squaredNorm();
Vec2 = Vec1.normalized();      | Vec1.normalize();
                               |                    // inplace
```

(8) Cross product

```
#include <Eigen/Geometry>
  Vec3 = vec1.cross(Vec2);
```

★ Coefficient-wise & Array operators

```
array1 * / array2        array1 * /= array2

array1 ± Scalar          array1 ±= Scalar
```

array1.abs()	sin(array1)
array1.sin()	abs(array1)
array1.square()	square(array1)
array1.floor()	floor(array1)

⇒ It is also very simple to apply any user defined function foo using DenseBase together with.

→ ① std::ptr_fun

```
mat1.unaryExpr(std::ptr_fun(foo));
```

→ ② std::ref

```
mat1.unaryExp(std::ref(foo));
```

→ ③ lambdas

```
mat1.unaryExp([](double x){return foo(x);});
```

★ Reductions

⇒ Eigen provides several reduction methods such as:

- minCoeff()
- maxCoeff()
- sum()
- prod()
- trace()
- norm()
- squaredNorm()
- all()
- any()

⇒ All reduction operations can be done matrix-wise, column-wise or row-wise.

Example

$$mat = \begin{matrix} 5 & 3 & 1 \\ 2 & 7 & 8 \\ 9 & 4 & 6 \end{matrix}$$

```
mat.minCoeff();          ——— 1
mat.colwise().minCoeff(); ——— 2 3 1
mat.rowwise().minCoeff(); ——— 1
                              2
                              4
```

```
if ((array1 > 0).all()) ...
```

↳ If all coefficients of array1 are greater than 0

```
if ((array1 < array2).any()) ...
```

↳ If there exist a pair i,j such that array1(i,j) < array2(i,j)

★ Sub-matrices

① Read-Write access to a column or a row of a matrix (or array)

```
mat.row(i) = mat.col(j);
mat1.col(j1).swap(mat1.col(j2));
```

② Read-Write access to sub-vectors

ⓐ Default version — Optimized version when the size is known at compile time

ⓐ `Vec1.head(n)` — `Vec1.head<n>()`

⇒ The first n coeffs

ⓑ `Vec1.tail(n)` — `Vec1.tail<n>()`

⇒ The last n coeffs

ⓒ `Vec1.segment(Pos,n)` — `Vec1.segment<n>(Pos)`

⇒ The n coeffs in the range [Pos : Pos+(n-1)]

③ Read-Write access to sub-matrices

Default Version — Optimized version when the size is known at Compile time

ⓐ `mat1.block(i,j,rows,cols)` — `mat1.block<rows,cols>(i,j)`

⇒ The rows x cols matrix starting from position (i,j)

(a) mat1.topLeftCorner(rows, cols) ——
(b) mat1.topRightCorner(rows, cols) ——
(c) mat1.bottomLeftCorner(rows, cols) ——
(d) mat1.bottomRightCorner(rows, cols) ——

⇒ The rows x cols sub matrix taken in one of the four corners.

(a) mat1.topRows(rows) ——
(b) mat1.bottomRows(rows) ——
(c) mat1.leftCols(cols) ——
(d) mat1.rightCols(cols) ——

⇒ Specialized versions of block() when the block fit two corners.

★ Miscellaneous Operations

① Reverse

⇒ Vectors, rows, and/or columns of a matrix can be reversed.

② Replicate

⇒ Vectors, matrices, rows, and/or columns can be replicated in any direction.

★ Diagonal matrices

⇒ View a vector as a diagonal matrix.

```
mat1 = Vec1.asDiagonal();
```

⇒ Access the diagonal and super/sub diagonals of a matrix as a vector.

ⓐ `Vec1 = mat1.diagonal();`
↳ Main diagonal

ⓑ `Vec1 = mat1.diagonal(+n);`
↳ nth super diagonal.

ⓒ `Vec1 = mat1.diagonal(-n);`
↳ nth sub diagonal

★ Triangular Views / Symmetric or Self adjoint Views

⇒ These special property of the matrix can be used to perform optimized operations.

Eigen / LU

⇒ This module defines the following MatrixBase methods:

- MatrixBase::inverse()
- MatrixBase::determinant()