Quick reference guide

Dense matrix and array manipulation

top

Modules and Header files

The **Eigen** library is divided in a Core module and several additional modules. Each module has a corresponding header file which has to be included in order to use the module. The `Dense` and **`Eigen`** header files are provided to conveniently gain access to several modules at once.

Module	Header file	Contents
Core	`#include <Eigen/Core>`	**Matrix** and **Array** classes, basic linear algebra (including triangular and selfadjoint products), array manipulation
Geometry	`#include <Eigen/Geometry>`	**Transform**, **Translation**, **Scaling**, **Rotation2D** and 3D rotations (**Quaternion**, **AngleAxis**)
LU	`#include <Eigen/LU>`	Inverse, determinant, LU decompositions with solver (**FullPivLU**, **PartialPivLU**)
Cholesky	`#include <Eigen/Cholesky>`	**LLT** and **LDLT** Cholesky factorization with solver
Householder	`#include <Eigen/Householder>`	Householder transformations; this module is used by several linear algebra modules
SVD	`#include <Eigen/SVD>`	SVD decomposition with least-squares solver (**JacobiSVD**)
QR	`#include <Eigen/QR>`	QR decomposition with solver (**HouseholderQR**, **ColPivHouseholderQR**, **FullPivHouseholderQR**)
Eigenvalues	`#include <Eigen/Eigenvalues>`	Eigenvalue, eigenvector decompositions (**EigenSolver**, **SelfAdjointEigenSolver**, **ComplexEigenSolver**)
Sparse	`#include <Eigen/Sparse>`	Sparse matrix storage and related basic linear algebra (**SparseMatrix**, DynamicSparseMatrix, **SparseVector**)
	`#include <Eigen/Dense>`	Includes Core, Geometry, LU, Cholesky, SVD, QR, and Eigenvalues header files
	`#include <Eigen/Eigen>`	Includes Dense and Sparse header files (the whole **Eigen** library)

top

Array, matrix and vector types

Recall: **Eigen** provides two kinds of dense objects: mathematical matrices and vectors which are both represented by the template class **Matrix**, and general 1D and 2D arrays represented by the template class **Array**:

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

- `Scalar` is the scalar type of the coefficients (e.g., `float`, `double`, `bool`, `int`, etc.).
- `RowsAtCompileTime` and `ColsAtCompileTime` are the number of rows and columns of the matrix as known at compile-time or `Dynamic`.
- `Options` can be `ColMajor` or `RowMajor`, default is `ColMajor`. (see class **Matrix** for more options)

All combinations are allowed: you can have a matrix with a fixed number of rows and a dynamic number of columns, etc. The following are all valid:

```
Matrix<double, 6, Dynamic>                  // Dynamic number of columns (heap allocation)
Matrix<double, Dynamic, 2>                  // Dynamic number of rows (heap allocation)
Matrix<double, Dynamic, Dynamic, RowMajor>  // Fully dynamic, row major (heap allocation)
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**. Some examples:

Matrices			Arrays		
`Matrix<float,Dynamic,Dynamic>`	<=>	`MatrixXf`	`Array<float,Dynamic,Dynamic>`	<=>	`ArrayXXf`
`Matrix<double,Dynamic,1>`	<=>	`VectorXd`	`Array<double,Dynamic,1>`	<=>	`ArrayXd`
`Matrix<int,1,Dynamic>`	<=>	`RowVectorXi`	`Array<int,1,Dynamic>`	<=>	`RowArrayXi`
`Matrix<float,3,3>`	<=>	`Matrix3f`	`Array<float,3,3>`	<=>	`Array33f`
`Matrix<float,4,1>`	<=>	`Vector4f`	`Array<float,4,1>`	<=>	`Array4f`

Conversion between the matrix and array worlds:

```
Array44f a1, a1;
Matrix4f m1, m2;
m1 = a1 * a2;                     // coeffwise product, implicit conversion from array to matrix.
a1 = m1 * m2;                     // matrix product, implicit conversion from matrix to array.
a2 = a1 + m1.array();             // mixing array and matrix is forbidden
m2 = a1.matrix() + m1;            // and explicit conversion is required.
ArrayWrapper<Matrix4f> m1a(m1);   // m1a is an alias for m1.array(), they share the same coefficients
MatrixWrapper<Array44f> a1m(a1);
```

In the rest of this document we will use the following symbols to emphasize the features which are specifics to a given kind of object:

- * linear algebra matrix and vector only
- * array objects only

Basic matrix manipulation

	1D objects	2D objects	Notes

Constructors	`Vector4d  v4;`
	`Vector2f  v1(x, y);`
	`Array3i   v2(x, y, z);`
	`Vector4d  v3(x, y, z, w);`
	
	`VectorXf  v5; // empty object`
	`ArrayXf   v6(size);`
		`Matrix4f  m1;`
		
		`MatrixXf  m5; // empty object`
		`MatrixXf  m6(nb_rows, nb_columns);`
			By default, the coefficients are left uninitialized

Comma initializer	`Vector3f  v1;     v1 << x, y, z;`
	`ArrayXf   v2(4);  v2 << 1, 2, 3, 4;`
		`Matrix3f  m1;   m1 << 1, 2, 3,`
		`                      4, 5, 6,`
		`                      7, 8, 9;`	

Comma initializer (bis)	`int rows=5, cols=5;`
	`MatrixXf m(rows,cols);`
	`m << (Matrix3f() << 1, 2, 3, 4, 5, 6, 7, 8, 9).finished(),`
	`     MatrixXf::Zero(3,cols-3),`
	`     MatrixXf::Zero(rows-3,3),`
	`     MatrixXf::Identity(rows-3,cols-3);`
	`cout << m;`
			output:
			`1 2 3 0 0`
			`4 5 6 0 0`
			`7 8 9 0 0`
			`0 0 0 1 0`
			`0 0 0 0 1`

Runtime info	`vector.size();`
	
	`vector.innerStride();`
	`vector.data();`
		`matrix.rows();          matrix.cols();`
		`matrix.innerSize();`
		`        matrix.outerSize();`
		`matrix.innerStride();`
		`        matrix.outerStride();`
		`matrix.data();`
			Inner/Outer* are storage order dependent

Compile-time	`ObjectType::Scalar` `ObjectType::RealScalar`	`ObjectType::RowsAtCompileTime` `ObjectType::ColsAtCompileTime`	

info	`ObjectType::Index`	`ObjectType::SizeAtCompileTime`	

Resizing	`vector.resize(size);`	`matrix.resize(nb_rows, nb_cols);`
		`matrix.resize(Eigen::NoChange, nb_cols);`
		`matrix.resize(nb_rows, Eigen::NoChange);`
			no-op if the new sizes match, otherwise data are lost

	`vector.resizeLike(other_vector);` `vector.conservativeResize(size);`	`matrix.resizeLike(other_matrix);` `matrix.conservativeResize(nb_rows, nb_cols);`	resizing with data preservation

Coeff access with range checking	`vector(i)` `vector.x()`
	`vector[i]` `vector.y()`
	`vector.z()`
	`vector.w()`
		`matrix(i,j)`	Range checking is disabled if NDEBUG or EIGEN_NO_DEBUG is defined

Coeff access without range checking	`vector.coeff(i)` `vector.coeffRef(i)`	`matrix.coeff(i,j)` `matrix.coeffRef(i,j)`	
Assignment/copy	`object = expression;` `object_of_float = expression_of_double.cast<float>();`		the destination is automatically resized (if possible)

Predefined Matrices

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);
```

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);
N/A

x.setZero(rows, cols);
x.setOnes(rows, cols);
x.setConstant(rows, cols, value);
x.setRandom(rows, cols);
N/A
```

Dynamic-size vector

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

x = Dynamic1D::Zero(size);
x = Dynamic1D::Ones(size);
x = Dynamic1D::Constant(size,
        value);
x = Dynamic1D::Random(size);
x = Dynamic1D::LinSpaced(size, low,
        high);

x.setZero(size);
x.setOnes(size);
x.setConstant(size, value);
x.setRandom(size);
x.setLinSpaced(size, low, high);
```

Identity and **basis vectors** *

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

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

```
x = Dynamic2D::Identity(rows, cols);
x.setIdentity(rows, cols);



N/A
```

```
N/A


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

Mapping external arrays

Contiguous memory	`float data[] = {1,2,3,4};`
	`Map<Vector3f> v1(data);       // uses v1 as a Vector3f object`
	`Map<ArrayXf>  v2(data,3);     // uses v2 as a ArrayXf object`
	`Map<Array22f> m1(data);       // uses m1 as a Array22f object`
	`Map<MatrixXf> m2(data,2,2);   // uses m2 as a MatrixXf object`

Typical usage of strides	`float data[] = {1,2,3,4,5,6,7,8,9};`
	`Map<VectorXf,0,InnerStride<2> >  v1(data,3);                      // = [1,3,5]`
	`Map<VectorXf,0,InnerStride<> >   v2(data,3,InnerStride<>(3));     // = [1,4,7]`
	`Map<MatrixXf,0,OuterStride<3> >  m2(data,2,3);                    // both lines     |1,4,7|`
	`Map<MatrixXf,0,OuterStride<> >   m1(data,2,3,OuterStride<>(3));   // are equal to:  |2,5,8|`

top

Arithmetic Operators

add subtract	`mat3 = mat1 + mat2;           mat3 += mat1;` `mat3 = mat1 - mat2;           mat3 -= mat1;`
scalar product	`mat3 = mat1 * s1;             mat3 *= s1;           mat3 = s1 * mat1;` `mat3 = mat1 / s1;             mat3 /= s1;`

matrix/vector products *	`col2 = mat1 * col1;`
	`row2 = row1 * mat1;           row1 *= mat1;`
	`mat3 = mat1 * mat2;           mat3 *= mat1;`

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

dot product
inner product *
	`scalar = vec1.dot(vec2);`
	`scalar = col1.adjoint() * col2;`
	`scalar = (col1.adjoint() * col2).value();`

outer product *	`mat = col1 * col2.transpose();`
norm **normalization** *	`scalar = vec1.norm();         scalar = vec1.squaredNorm()` `vec2 = vec1.normalized();     vec1.normalize(); // inplace`
cross product *	`#include <Eigen/Geometry>` `vec3 = vec1.cross(vec2);`

top

Coefficient-wise & Array operators

Coefficient-wise operators for matrices and vectors:

Matrix API *	**Via Array conversions**
`mat1.cwiseMin(mat2)`	`mat1.array().min(mat2.array())`
`mat1.cwiseMax(mat2)`	`mat1.array().max(mat2.array())`
`mat1.cwiseAbs2()`	`mat1.array().abs2()`
`mat1.cwiseAbs()`	`mat1.array().abs()`
`mat1.cwiseSqrt()`	`mat1.array().sqrt()`
`mat1.cwiseProduct(mat2)`	`mat1.array() * mat2.array()`
`mat1.cwiseQuotient(mat2)`	`mat1.array() / mat2.array()`

It is also very simple to apply any user defined function `foo` using DenseBase::unaryExpr together with std::ptr_fun:

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

Array operators:*

Arithmetic operators	`array1 * array2     array1 / array2     array1 *= array2    array1 /= array2` `array1 + scalar     array1 - scalar     array1 += scalar    array1 -= scalar`

Comparisons	`array1 < array2     array1 > array2     array1 < scalar     array1 > scalar`
	`array1 <= array2    array1 >= array2    array1 <= scalar    array1 >= scalar`
	`array1 == array2    array1 != array2    array1 == scalar    array1 != scalar`

	`array1.min(array2)`

Trigo, power, and misc functions and the STL variants	`array1.max(array2)`
	`array1.abs2()`
	`array1.abs()`
	`array1.sqrt()`
	`array1.log()`
	`array1.exp()`
	`array1.pow(exponent)`
	`array1.square()`
	`array1.cube()`
	`array1.inverse()`
	`array1.sin()`
	`array1.cos()`
	`array1.tan()`
	`array1.asin()`
	`array1.acos()`
		
		`abs(array1)`
		`sqrt(array1)`
		`log(array1)`
		`exp(array1)`
		`pow(array1,exponent)`
		
		`sin(array1)`
		`cos(array1)`
		`tan(array1)`
		`asin(array1)`
		`acos(array1)`

top

Reductions

Eigen provides several reduction methods such as: **minCoeff()** , **maxCoeff()** , **sum()** , **prod()** , **trace()** *, **norm()** *, **squaredNorm()** *, **all()** , and **any()** . All reduction operations can be done matrix-wise, **column-wise** or **row-wise** . Usage example:

`mat = 5 3 1`
`      2 7 8`
`      9 4 6`
	`mat.minCoeff();`	`1`

	`mat.colwise().minCoeff();`	`2 3 1`

	`mat.rowwise().minCoeff();`	`1`
		`2`
		`4`

Special versions of **minCoeff** and **maxCoeff** :

```
int i, j;
s = vector.minCoeff(&i);        // s == vector[i]
s = matrix.maxCoeff(&i, &j);    // s == matrix(i,j)
```

Typical use cases of all() and any():

```
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) ...
```

top

Sub-matrices

Read-write access to a **column** or a **row** of a matrix (or array):

```
mat1.row(i) = mat2.col(j);
mat1.col(j1).swap(mat1.col(j2));
```

Read-write access to sub-vectors:

Default versions	Optimized versions 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]
Read-write access to sub-matrices:		
`mat1.block(i,j,rows,cols)`	`mat1.block<rows,cols>(i,j)`	the rows x cols sub-matrix

(more)	**(more)**	starting from position (i,j)

`mat1.topLeftCorner(rows,cols)`
`mat1.topRightCorner(rows,cols)`
`mat1.bottomLeftCorner(rows,cols)`
`mat1.bottomRightCorner(rows,cols)`
	`mat1.topLeftCorner<rows,cols>()`
	`mat1.topRightCorner<rows,cols>()`
	`mat1.bottomLeftCorner<rows,cols>()`
	`mat1.bottomRightCorner<rows,cols>()`
		the rows x cols sub-matrix taken in one of the four corners

`mat1.topRows(rows)`
`mat1.bottomRows(rows)`
`mat1.leftCols(cols)`
`mat1.rightCols(cols)`
	`mat1.topRows<rows>()`
	`mat1.bottomRows<rows>()`
	`mat1.leftCols<cols>()`
	`mat1.rightCols<cols>()`
		specialized versions of block() when the block fit two corners

top

Miscellaneous operations

Reverse

Vectors, rows, and/or columns of a matrix can be reversed (see **DenseBase::reverse()**, **DenseBase::reverseInPlace()**, **VectorwiseOp::reverse()**).

```
vec.reverse()           mat.colwise().reverse()   mat.rowwise().reverse()
vec.reverseInPlace()
```

Replicate

Vectors, matrices, rows, and/or columns can be replicated in any direction (see **DenseBase::replicate()**, **VectorwiseOp::replicate()**)

```
vec.replicate(times)                                          vec.replicate<Times>
mat.replicate(vertical_times, horizontal_times)               mat.replicate<VerticalTimes, HorizontalTimes>()
mat.colwise().replicate(vertical_times, horizontal_times)     mat.colwise().replicate<VerticalTimes,
       HorizontalTimes>()
mat.rowwise().replicate(vertical_times, horizontal_times)     mat.rowwise().replicate<VerticalTimes,
       HorizontalTimes>()
```

top

Diagonal, Triangular, and Self-adjoint matrices

(matrix world *)

Diagonal matrices

Operation	Code
view a vector **as a diagonal matrix**	`mat1 = vec1.asDiagonal();`
Declare a diagonal matrix	`DiagonalMatrix<Scalar,SizeAtCompileTime> diag1(size);` `diag1.diagonal() = vector;`

Access the **diagonal** and **super/sub diagonals** of a matrix as a vector (read/write)	`vec1 = mat1.diagonal();` `// main diagonal` `mat1.diagonal() = vec1;`
	`vec1 = mat1.diagonal(+n);` `// n-th super diagonal` `mat1.diagonal(+n) = vec1;`
	`vec1 = mat1.diagonal(-n);` `// n-th sub diagonal` `mat1.diagonal(-n) = vec1;`
	`vec1 = mat1.diagonal<1>();` `// first super diagonal` `mat1.diagonal<1>() = vec1;`
	`vec1 = mat1.diagonal<-2>();` `// second sub diagonal` `mat1.diagonal<-2>() = vec1;`

Optimized products and inverse	`mat3  = scalar * diag1 * mat1;`
	`mat3 += scalar * mat1 * vec1.asDiagonal();`
	`mat3 = vec1.asDiagonal().inverse() * mat1`
	`mat3 = mat1 * diag1.inverse()`

Triangular views

TriangularView gives a view on a triangular part of a dense matrix and allows to perform optimized operations on it. The opposite triangular part is never referenced and can be used to store other information.

Note

The .triangularView() template member function requires the `template` keyword if it is used on an object of a type that depends on a template parameter; see **The template and typename keywords in C++** for details.

Operation	Code
Reference to a triangular with optional unit or null diagonal (read/write):	`m.triangularView<Xxx>()` Xxx = **Upper**, **Lower**, **StrictlyUpper**, **StrictlyLower**, **UnitUpper**, **UnitLower**
Writing to a specific triangular part: (only the referenced triangular part is evaluated)	`m1.triangularView<Eigen::Lower>() = m2 + m3`
Conversion to a dense matrix setting the opposite triangular part to zero:	`m2 = m1.triangularView<Eigen::UnitUpper>()`
Products:	`m3 += s1 * m1.adjoint().triangularView<Eigen::UnitUpper>() * m2` `m3 -= s1 * m2.conjugate() * m1.adjoint().triangularView<Eigen::Lower>()`

Solving linear equations:
$M_2 := L_1^{-1} M_2$
$M_3 := {L_1^*}^{-1} M_3$
$M_4 := M_4 U_1^{-1}$
	`L1.triangularView<Eigen::UnitLower>().solveInPlace(M2)`
	`L1.triangularView<Eigen::Lower>().adjoint().solveInPlace(M3)`
	`U1.triangularView<Eigen::Upper>().solveInPlace<OnTheRight>(M4)`

Symmetric/selfadjoint views

Just as for triangular matrix, you can reference any triangular part of a square matrix to see it as a selfadjoint matrix and perform special and optimized operations. Again the opposite triangular part is never referenced and can be used to store other information.

Note

The .selfadjointView() template member function requires the `template` keyword if it is used on an object of a type that depends on a template parameter; see **The template and typename keywords in C++** for details.

Operation	Code
Conversion to a dense matrix:	`m2 = m.selfadjointView<Eigen::Lower>();`
Product with another general matrix or vector:	`m3  = s1 * m1.conjugate().selfadjointView<Eigen::Upper>() * m3;` `m3 -= s1 * m3.adjoint() * m1.selfadjointView<Eigen::Lower>();`
Rank 1 and rank K update:	

$upper(M_1) \mathrel{+}= s_1 M_2 M_2^*$ $lower(M_1) \mathrel{-}= M_2^* M_2$	`M1.selfadjointView<Eigen::Upper>().rankUpdate(M2,s1);` `M1.selfadjointView<Eigen::Lower>().rankUpdate(M2.adjoint(),-1);`
Rank 2 update: ($M \mathrel{+}= suv^* + svu^*$)	`M.selfadjointView<Eigen::Upper>().rankUpdate(u,v,s);`

Solving linear equations:
($M_2 := M_1^{-1} M_2$)
	`// via a standard Cholesky factorization`
	`m2 = m1.selfadjointView<Eigen::Upper>().llt().solve(m2);`
	`// via a Cholesky factorization with pivoting`
	`m2 = m1.selfadjointView<Eigen::Lower>().ldlt().solve(m2);`