Matrix Class

Two simple examples

```cpp
#include <iostream>
#include <Eigen/Dense>

using Eigen::Matrix3d;
using Eigen::Vector3d;

int main() {
  Matrix3d m = Matrix3d::Random();
  m = (m + Matrix3d::Constant(1.2)) * 50;
  std::cout << "m =" << std::endl << m << std::endl;
  Vector3d v(1, 2, 3);

  std::cout << "m * v =" << std::endl << m * v << std::endl;
}
```

```cpp
#include <iostream>
#include <Eigen/Dense>

using Eigen::MatrixXd;
using Eigen::VectorXd;

int main() {
  MatrixXd m = MatrixXd::Random(3, 3);
  m = (m + MatrixXd::Constant(3, 3, 1.2)) * 50;
  std::cout << "m =" << std::endl << m << std::endl;
  VectorXd v(3);
  v << 1, 2, 3;
  std::cout << "m * v =" << std::endl << m * v << std::endl;
}
```

The output is

```
m =
94.8 76.7 82.3
70.3 36.5 28.4
39.3  106 61.2
m * v =
495
228
436
```

Matrix and Vector Type

```
Matrix<typename Scalar,
       int RowsAtCompileTime,
       int ColsAtCompileTime,
       int Options = 0,
       int MaxRowsAtCompileTime = RowsAtCompileTime,
       int MaxColsAtCompileTime = ColsAtCompileTime>
```

Eigen

```
- Matrix<double, 4, 5> a;
```

```
- typedef Matrix<float, 4, 4> Matrix4f;
- typedef Matrix<double, 3, 1> Vector3d;
- typedef Matrix<double, Dynamic, Dynamic> MatrixXd;
- typedef Matrix<int, Dynamic, 1> VectorXi;
```

- ‘i’ for integer - ‘d’ for double - ‘f’ for float - ‘c’ for complex

Declaration and Assignment

```
Matrix3f a;
MatrixXf b;

MatrixXf a(10,15);
VectorXf b(30);

Matrix3f a(3,3);
Matrix3f a(4,4); // illegal

Vector2d a(5.0, 6.0);
Vector3d b(5.0, 6.0, 7.0);
Vector4d c(5.0, 6.0, 7.0, 8.0);

Matrix3f m;
m << 1, 2, 3,
     4, 5, 6,
     7, 8, 9; // comma initialization
```

Matrix Size and Resize

```
#include <iostream>
#include <Eigen/Dense>

int main() {
  Eigen::MatrixXd m(2, 5);
  m.resize(4, 3);
```

```
  std::cout << "The matrix m is of size " << m.rows() << "x" << m.cols() << std::endl;
  std::cout << "It has " << m.size() << " coefficients" << std::endl;
  Eigen::VectorXd v(2);
  v.resize(5);
  std::cout << "The vector v is of size " << v.size() << std::endl;
  std::cout << "As a matrix, v is of size " << v.rows() << "x" << v.cols() << std::endl;
}
```

The output is

```
The matrix m is of size 4x3
It has 12 coefficients
The vector v is of size 5
As a matrix, v is of size 5x1
```

```
#include <iostream>
#include <Eigen/Dense>

int main() {
  Eigen::Matrix4d m;
  m.resize(4, 4);  // legal - no operation
  m.resize(3, 4);  // illegal
  std::cout << "The matrix m is of size " << m.rows() << "x" << m.cols() << std::endl;
}
```

Fixed vs. Dynamic size

- Use fixed sizes for very small sizes where you can, and use dynamic sizes for larger sizes or where you have to.
- For small sizes, especially for sizes smaller than (roughly) 16, using fixed sizes is hugely beneficial to performance, as it allows Eigen to avoid dynamic memory allocation and to unroll loops.
  –
  – Matrix4f Eigen 4x4 16
- for large enough sizes, say for sizes greater than (roughly) 32, the performance benefit of using fixed sizes becomes negligible. Worse, trying to create a very large matrix using fixed sizes inside a function could result in a stack overflow, since Eigen will try to allocate the array automatically as a local variable, and this is normally done on the stack.

Matrix and vector arithmetic

```
#include <iostream>
#include <Eigen/Dense>

int main() {
  Eigen::Matrix2d a;
```

```
  a << 1, 2, 3, 4;
  Eigen::MatrixXd b(2, 2);
  b << 2, 3, 1, 4;
  Eigen::Vector2d c(1., 2.);
  std::cout << "a + b =\n" << a + b << std::endl; // Addition
  std::cout << "a - b =\n" << a - b << std::endl; // Subtraction
  std::cout << "a * 2.5 =\n" << a * 2.5 << std::endl; // Scalar multiplication

  MatrixXcf a = MatrixXcf::Random(2, 2);
  cout << "Here is the matrix a\n" << a << endl;
  cout << "Here is the matrix a^T\n" << a.transpose() << endl; // Transposition
  cout << "Here is the conjugate of a\n" << a.conjugate() << endl; // Conjugation
  cout << "Here is the matrix a^*\n" << a.adjoint() << endl;

  cout << "mul of a and b\n" << a * b << endl; // Matrix multiplication
  cout << "mul of a and c\n" << a * c << endl; // Matrix-vector multiplication

  Eigen::Vector3d v(1, 2, 3);
  Eigen::Vector3d w(0, 1, 2);
  std::cout << "Dot product: " << v.dot(w) << std::endl; // Inner product
  double dp = v.adjoint() * w;  // automatic conversion of the inner product to a scalar
  std::cout << "Dot product via a matrix product: " << dp << std::endl;
  std::cout << "Cross product:\n" << v.cross(w) << std::endl; // cross product
}
```

- If you do b = a.transpose(), then the transpose is evaluated at the same time as the result is written into b. However, there is a complication here.
- If you do a = a.transpose(), then Eigen starts writing the result into a before the evaluation of the transpose is finished.

```
a = a.transpose() // Don't do this
a = a.transposeInPlace() // use this
```

Linear algebra and decompositions

Basic linear solving

```
#include <iostream>
#include <Eigen/Dense>

int main() {
  Eigen::Matrix3f A;
  Eigen::Vector3f b;
  A << 1, 2, 3, 4, 5, 6, 7, 8, 10;
  b << 3, 3, 4;
  Eigen::Vector3f x = A.colPivHouseholderQr().solve(b);
```

```
  std::cout << "The solution is:\n" << x << std::endl;
}
```

- ColPivHouseholderQR is a QR decomposition with column pivoting. It's a good compromise for this tutorial, as it works for all matrices while being quite fast.
- colPivHouseholderQr() method returns an object of class ColPivHouseholderQR, this line can be replaced by

```
ColPivHouseholderQR<Matrix3f> dec(A);
Vector3f x = dec.solve(b);
```

Decomposition	Method	Requirements on the matrix	Speed (small-to-medium)	Speed (large)	Accuracy
PartialPivLU	partialPivLu()	Invertible	++	++	+
FullPivLU	fullPivLu()	None	-	--	+++
HouseholderQR	householderQr()	None	++	++	+
ColPivHouseholderQR	colPivHouseholderQr()	None	+	-	+++
FullPivHouseholderQR	fullPivHouseholderQr()	None	-	--	+++
CompleteOrthogonalDecomposition	completeOrthogonalDecomposition()	None	+	-	+++
LLT	llt()	Positive definite	+++	+++	+
LDLT	ldlt()	Positive or negative semidefinite	+++	+	++
BDCSVD	bdcSvd()	None	-	-	+++
JacobiSVD	jacobiSvd()	None	-	---	+++

decompositions that you can choose from, depending on your matrix, the problem you are trying to solve, and the trade-off you want to make

Computing eigenvalues and eigenvectors

```
#include <iostream>
#include <Eigen/Dense>

int main() {
  Eigen::Matrix2f A;
  A << 1, 2, 2, 3;
  std::cout << "Here is the matrix A:\n" << A << std::endl;
  Eigen::SelfAdjointEigenSolver<Eigen::Matrix2f> eigensolver(A);
  if (eigensolver.info() != Eigen::Success) abort();
  std::cout << "The eigenvalues of A are:\n" << eigensolver.eigenvalues() << std::endl;
  std::cout << "Here's a matrix whose columns are eigenvectors of A \n" << "corresponding to
}
```

The output is

```
Here is the matrix A:
1 2
2 3
The eigenvalues of A are:
```

```
-0.236
  4.24
Here's a matrix whose columns are eigenvectors of A
corresponding to these eigenvalues:
-0.851 -0.526
 0.526 -0.851
```

Inverse and Determinant

```cpp
#include <iostream>
#include <Eigen/Dense>

int main() {
  Eigen::Matrix3f A;
  A << 1, 2, 1, 2, 1, 0, -1, 1, 2;
  std::cout << "The determinant of A " << A.determinant() << std::endl;
  std::cout << "The inverse of A \n" << A.inverse() << std::endl;
}
```