Eigen: a practical approach to linear algebra
http://eigen.tuxfamily.org

Benoît Jacob, Dept. of Mathematics,
bjacob@math.toronto.edu

NA seminar, University of Toronto
23 january 2009

Eigen is co-developed with
Gaël Guennebaud (INRIA Bordeaux)
and a handful of occasional contributors

Eigen:

- is a C++ library for linear algebra.
- is versatile, all-in-one
- has a great C++ API
- has great performance
- is lightweight
- is LGPL licensed and cross-platform
- is used in real world projects

Eigen is **versatile**

All 3 kinds of matrices/vectors:

- Dense, fixed-size
- Dense, dynamic-size
- Sparse (experimental)

Moreover these 3 kinds are fully **integrated** with one another.

The fixed-size case is crucial:

- Very commonly used
- Huge optimization opportunity

Eigen is **all-in-one**

- Basic matrix/vector functionality
- Linear algebra algorithms (LU, QR, SVD, ...)
- Geometry framework (projective, quaternions, ...)
- Array manipulation

Fully **self-contained** for **dense** algorithms.

Allows optional **backends** for **sparse** algorithms.

Eigen has a **great C++ API**

We'll give many examples shortly.

Teaser:

```
matrix.row(i) += lambda * matrix.row(j);
```

Notice: in C++, a+=b means a=a+b.

Eigen works on expressions and decides **automatically** when to use lazy evaluation.

Eigen has **great performance**

- Works at the level of expressions
- Explicit vectorization: SSE2+ and AltiVec
- Cache-friendly algorithms
- Takes advantage of fixed dimensions

We'll show some benchmarks...

Eigen is **lightweight**

It is a **pure template** library.

Compiled directly into the user code.

Only the code that's **actually used**, is compiled.

Compilation times are still reasonable.

No binary library to link to.

Eigen itself is small: 16,000 LOC.

Eigen is **available**

License: LGPL 3+ (alternatively GPL 2+)

Closed-source software may use Eigen.

Supported operating systems:
Linux, BSD, MacOSX, Windows

Supported compilers:
GCC 3.3+, MSVC 2005+, ICC

For vectorization:
Instruction set: SSE2+, AltiVec
Compiler: GCC 4.2+, MSVC 2008+, ICC

Eigen is used in real world apps:

- Computer graphics: MeshLab, VcgLib, Krita, libmv
- Robotics companies: Yujin, Willow Garage
- Desktop: KDE, KOffice (including Krita)
- Chemistry: Avogadro, soon Open Babel

Example program:

```
#include <Eigen/Core>
using namespace Eigen;
using namespace std;

int main()
{
  Matrix4f m = Matrix4f::Zero();
  m.diagonal().end(2) << 1, 2;
  cout << m << endl;
}
```

Matrix4f is a shortcut for:

```
Matrix<float,4,4>
```

Matlab code:

```
m = eye(20);
m = rand(20);
```

Eigen/C++ equivalent:

```
MatrixXf m = MatrixXf::Identity(20,20);
MatrixXf m = MatrixXf::Random(20,20);
```

MatrixXf is a shortcut for:

```
Matrix<float,Dynamic,Dynamic>
```

Matlab code:

```
v = rand(20, 1);
w = rand(20, 1); w = w / norm(w);
m = diag(rand(20, 1));
```

Eigen/C++ equivalent:

```
VectorXf v = VectorXf::Random(20);
VectorXf w = VectorXf::Random(20).normalized();
MatrixXf m = VectorXf::Random(20).asDiagonal();
```

VectorXf is a shortcut for:

```
Matrix<float,Dynamic,1>
```

Algebraic expressions work as expected

```
m4 = m3 * (x1 * m1 + x2 * m2);
```

and produce optimized code.

No bad aliasing effects:

```
m = m * m;
```

Eigen is conservative with operator overloading.

Matlab code:

```
m(i, j) = 0;
m = zeros(size(m));
m(i, :) = 0;
```

Eigen/C++ equivalent:

```
m(i,j) = 0;
m.setZero();
m.row(i).setZero();
```

Matlab code:

```
m(i, :) = m(i, :) + x*m(j, :);
m(:, [i j]) = m(:, [j i]);
```

Eigen/C++ equivalent:

```
m.row(i) += x * m.row(j);
m.col(i).swap(m.col(j));
```

Matlab code:

```
m(i:i+n-1, j:j+n-1) = eye(n);
m(i:i+n-1, j:j+n-1) = m2(i:i+n-1, j:j+n-1) * m3;
```

Eigen/C++ equivalent:

```
m.block(i,j,n,n).setIdentity();
m.block(i,j,n,n) = m2.block(i,j,n,n) * m3;
```

Matlab code:

```
result = sum(m(:));
result = sum(m(:, i));
result = sum(m');
result = sum(m(i, :).^3);
```

Eigen/C++ equivalent:

```
result = m.sum();
result = m.col(i).sum();
result = m.rowwise().sum();
result = m.row(i).cwise().cube().sum();
```

Matlab code:

```
m = [eye(n, p),    zeros(n, p);
     random(n, p), m2+m3];

v(i:i+3) = [a b c d];
```

Eigen/C++ equivalent:

```
m << MatrixXf::Identity(n,p), MatrixXf::Zero(n,p),
     MatrixXf::Random(n,p),   m2+m3;

v.segment(i,4) << a,b,c,d;
```

SHOW BENCHMARKS AS EXPOSED ON
http://eigen.tuxfamily.org/index.php?title=Benchmark

Eigen internals

With Eigen, an operation like

```
v3 = v1 + v2;
```

Is entirely performed in the "=".

The "+" by itself does nothing. It just returns a "sum expression" object.

The whole expression tree is encoded in the **type** of this object.

Example: the expression

```
m1 + m2.transpose()
```

gives an object of type

```
Sum< MatrixXf, Transpose<MatrixXf> >
```

The expression types carry a lot of recursive **meta-data** about the expressions:

- Scalar type
- Dimensions at compile-time, or Dynamic
- Maximum dimensions at compile-time, or Dynamic
- Linearity: index-based access
- Cost of reading a coefficient
- Hints for/against lazy evaluation
- Vectorizability: packet access
- Data alignment
- Prefered storage order
- Shape (diagonal...)

By default we **lazy evaluate** expression.

However this is not always good.

Eigen automatically takes the decision to evaluate an intermediate step into a temporary.

Eigen then takes advantage of that to split the expression tree, limiting its depth.

Lazy Evaluation:

- Can be **necessary**.

Example: m1.block(r,c,nr,nc) = m2.

- Can be **very good**.

Example: v4=v1+v2+v3.

- Can be **bad**.

Example: m4=(m1+m2)*m3

- Can be **terrible**.

Example: m4=(m1*m2)*m3

- Can be **dangerous**.

Example: m = m*m

We conducted **experiments**.

Lazy evaluation must be justified by a **clear** gain.

Vectorization

Dynamic-size matrices: we allow ourselves some runtime logic (taking care of unaligned boundaries...)

Fixed-size matrices: no runtime overhead allowed

The expression metadata allows to choose the right vectorization strategy.

When cache friendliness matters, nothing replaces handwritten code. So we have handwritten code for e.g. matrix product.

Portable abstractions for SIMD instructions and packets.

Vectorization Example.

```
#include<Eigen/Core>
using namespace Eigen;

void foo(Matrix2f& u,
         float a, const Matrix2f& v,
         float b, const Matrix2f& w)
{
  u = a*v + b*w - u;
}
```

Comments:

- Neither rows not columns fit in 128bit packets
- So necessary to understand that the expression is **linearizable**
- Notice how lazy evaluation allows to do only 1 traversal

```
movl  8(%ebp), %edx
movss 20(%ebp), %xmm0
movl  24(%ebp), %eax
movaps  %xmm0, %xmm2
shufps  $0, %xmm2, %xmm2
movss 12(%ebp), %xmm0
movaps  %xmm2, %xmm1
mulps (%eax), %xmm1
shufps  $0, %xmm0, %xmm0
movl  16(%ebp), %eax
mulps (%eax), %xmm0
addps %xmm1, %xmm0
subps (%edx), %xmm0
movaps  %xmm0, (%edx)
```

Comments? Questions?

http://eigen.tuxfamily.org

bjacob@math.toronto.edu

Core developers: Gaël Guennebaud, Benoît Jacob

Contributors: David Benjamin, Armin Berres, Daniel Gómez, Konstantinos Margaritis, Christian Mayer, Keir Mierle, Michael Olbrich, Kenneth Riddile, Alex Stapleton