Fast, Easy Linear Algebra in C++ with Eigen

Benjamin S. Skrainka

University of Chicago
The Harris School of Public Policy
skrainka@uchicago.edu

May 16, 2012

Objectives

This talk's objectives:

- State benefits of using Eigen from linear algebra
- List key components of Eigen
- Use Eigen to perform typical linear algebra operations
- Generate high quality pseudo-random numbers with Mersenne Twister

Plan for this Talk

Getting Started

Eigen's Containers: Matrix<T> and Array<T>

Advanced Operations

Mersenne Twister

Benefits of Eigen

Eigen makes dense linear algebra easy in C++:

- All basic features and many more advanced ones as well
- Template-based $\Rightarrow$ works with all arithmetic types
- Syntax is almost as easy to use as MATLAB's
- Blazing fast performance because of expression templates:
  - 10-30x speed up over MATLAB
  - Comparable to Intel MKL, the gold standard
  - Produces efficient code so arrays are only traversed once!
- Easy to install
- Good documentation
- For a single core
- Use PETSc or equivalent for MPI-based linear algebra on multiple cores

Eigen Benchmark

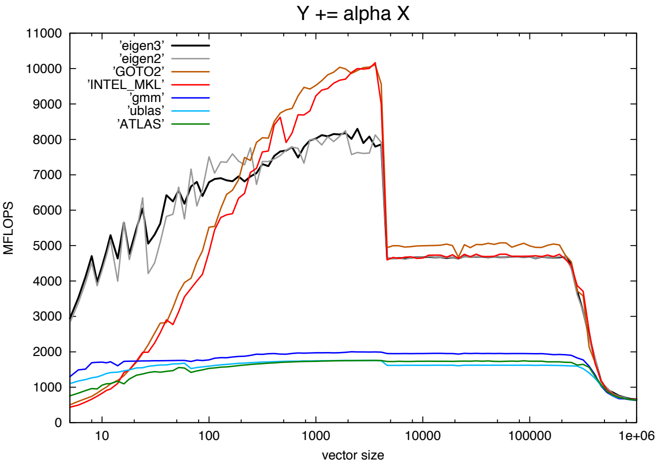

Figure is taken from Eigen documentation.

Eigen Benchmark

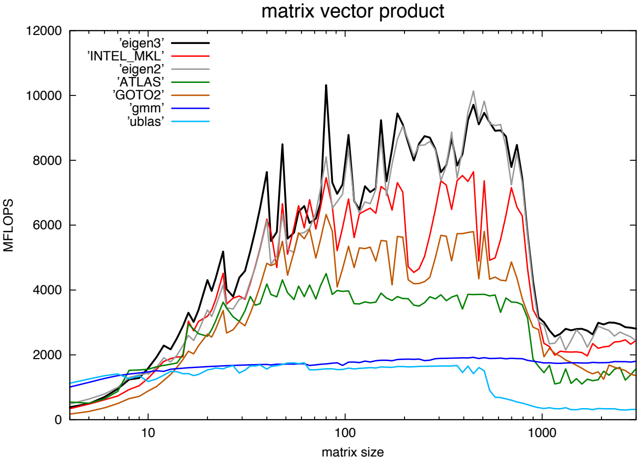

Figure is taken from Eigen documentation.

Eigen Benchmark

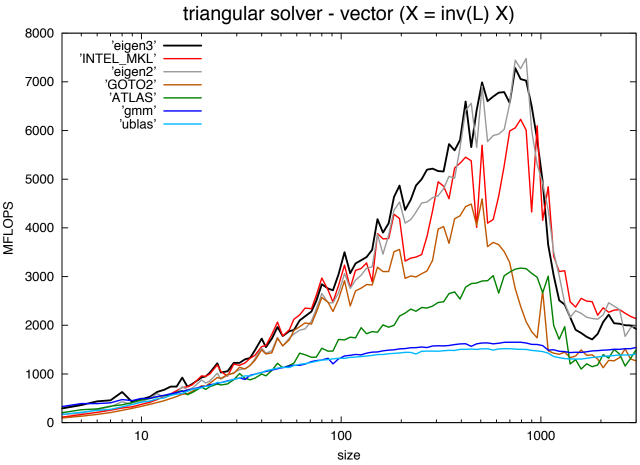

Figure is taken from Eigen documentation.

Overview of Eigen Libraries

Eigen provides several templated Matrix and Array types:

- Matrix<T>: generic matrix class
- MatrixXd: matrix of dynamically allocated doubles
- VectorXd: column vector of dynamically allocated doubles
- RowVectorXd: row vector of dynamically allocated doubles
- Plus common fixed sizes and types: Matrix3f, Matrix2i, etc.
- Map<T>: provides Matrix<T> interface to an existing slab of data

Matrix<T> operations are matrix arithmetic (like * in MATLAB) whereas Array<T> operations are coefficient-wise (like .* in MATLAB).

Math Operations

Eigen supports all the standard math functions via member functions:

- Includes:

`abs()`	`pow()`
`min()`	`exp()`
`max()`	`log()`
`square()`	`sqrt()`

- Must convert `Matrix<T>` to `Array<T>` using `array()` member function in order to perform coefficient-wise operations
- Performed coefficient-wise, i.e. like MATLAB where `m.array().exp()` is $\begin{pmatrix} \exp(a_{12}) & \exp(a_{12}) \\ \exp(a_{21}) & \exp(a_{22}) \end{pmatrix}$

Overview of Basic Linear Solving

Eigen also contains many linear solvers:

Decomposition	Speed	Accuracy
PartialPivLU	++	+
FullPivLU	-	+++
HouseholderQR	++	+
ColPivHouseholderQR	+	++
FullPivHouseholderQR	-	+++
LLT (Cholesky)	+++	+
LDLT (Robust Cholesky)	+++	++

There are also SVD and eigenvalue decompositions

Array<T> vs. Matrix<T>

Select Array<T> or Matrix<T> depending on desired behavior:

- `Matrix<T>`:
  - Designed for linear algebra
  - Supports matrix arithmetic
  - Has some coefficient-wise member functions
  - Convert to an `Array<T>` with `Matrix<T>::array()`
- `Array<T>`:
  - General purpose array
  - Coefficient-wise arithmetic
  - Can perform coefficient-wise operations, such as add a number to all elements
  - Convert to `Matrix<T>` with `Array<T>::matrix()`

Getting Started

First Steps

Let's start by discussing:

- Installation
- Configuration
- Getting Help

Installation

Eigen is incredibly easy to install:

- Entire library consists of header files
- No need to compile or build any binary libraries
- Just download and copy it to where you want it

To install,

- Download latest stable release from
  http://bitbucket.org/eigen/eigen/get/3.0.5.tar.gz
- Unpack:

```
$ tar xzvf eigen-eigen-6e7488e20373.tar.gz
$ mv eigen-eigen-6e7488e20373 eigen-3.0.5
```

Help

There are many resources for help:

- Online documentation
- Online Getting Started Tutorial
- Online Quick Reference

A First Application

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

int main()
{
  Eigen::MatrixXd m(2,2) ;
  m(0,0) = 3 ;
  m(1,0) = 2.5 ;
  m(0,1) = -1 ;
  m(1,1) = m(1,0) + m(0,1) ;
  std::cout << m << std::endl ;

  m.diagonal() << -1, -2 ;
  std::cout << m < std::endl ;
  return 0 ;
}
```

Building an Application with Eigen

To build an application with Eigen, use a supported C++ compiler such as GNU or MSVC++:

- Specify path to Eigen header files with `-I/path/to/Eigen`
- Do not compile with `-pedantic`:
  - `-pedantic` forces strict conformance to ANSI/ISO C++ standards
  - But, Eigen requires `long long` which is not yet officially supported...
- Disable debugging checks in product code:
  - Compile with `-DEIGEN_NO_DEBUG`
  - Huge speed up!

Example Compilation

Here is an example compilation

- For debugging (development):

```
$ g++ -Wall -Wextra -O0 -g -m64          \
  -I/home/skrainka/public/sw/include \
  eigenExample.cpp -o eigenExample
```

- For production code:

```
$ g++ -Wall -Wextra -O2 -m64             \
  -DEIGEN_NO_DEBUG                   \
  -I/home/skrainka/public/sw/include \
  eigenExample.cpp -o eigenExample
```

Practical Tips on Use

For production code, compile with -DEIGEN_NO_DEBUG:

- Removes checks for conformability, etc.
- Results in much faster code (10-30x!)
- Make sure your code is debugged first!

Use typedefs:

- For floating point type
- For key matrix and vector types

BLP Example:

```
typedef long double BLPReal_t ;
typedef Eigen::Matrix< BLPReal_t, Eigen::Dynamic,
  Eigen::Dynamic, Eigen::ColMajor | Eigen::AutoAlign >
  BLPMatrixXd ;
typedef Eigen::Matrix< BLPReal_t, Eigen::Dynamic, 1,
  Eigen::ColMajor | Eigen::AutoAlign > BLPVectorXd ;
```

Configuration Options

Eigen is highly configurable using compile-time macros. Some options are:

- `EIGEN_DEFAULT_TO_ROW_MAJOR`
- `EIGEN_INITIALIZE_MATRICES_BY_ZERO`
- `EIGEN_NO_DEBUG`

See the manual for details.

Eigen's Containers: Matrix<T> and Array<T>

Matrix<T>

`Matrix< typename Scalar, int nRows, int nCols >`:

- Scalar: data type to store in Matrix (double, int, etc.)
- nRows and nCols:
  - Number of rows and columns
  - Pre-specified number of rows and columns at compile time $\Rightarrow$ 'fixed size
  - Can use Eigen::Dynamic if size is unknown $\Rightarrow$ 'dynamic' size
  - Use fixed size for smaller matrices – i.e. size <= 32
- Can also set Eigen::RowMajor or Eigen::ColMajor

Vectors

A vector is just a Matrix:

- A column vector is

```
Eigen::Matrix< typename Scale, int nRows, 1 >
```

- A row vector is

```
Eigen::Matrix< typename Scale, 1, int nCols >
```

- Orientation matters:
  - VectorXd is a column vector
  - RowVectorXd is a row vector

Predefined Types

Eigen provides several predefined types:

- Matrix**Nt**
- Vector**Nt**
- RowVector**Nt**
- where

`N: Storage`	`t: type`
X: dynamic	**d**: double
n: number of rows and/or columns (2, 3, or 4)	**f**: float
	i: int
	cd: complex double
	cf: complex float

- Examples:
  - MatrixXd: dynamically allocated matrix of doubles
  - Vector3i: vector with three ints

Coefficient Accessors

Element access is intuitive:

- Access elements using operator():

```
Matrix3i mMat ;
mMat( 1, 2 ) = 3 ;
```

- Remember: 0-based indexing!
- Can also access sub-matrices (slices) like the ':' operator in MATLAB

Initialization

Initialization is easy and intuitive:

- Use operator<< to fill by column then row:

```
Matrix3d m3 ;
m3 << 1, 2, 3,
      4, 5, 6,
      7, 8, 9 ;
```

- Can use member functions
  - `setZero()`
  - `setOnes()`
  - `setConstant()`
  - `setRandom()`
  - `setLinSpaced()`
- See manual for more complex options

Matrix Arithmetic

Operators are overloaded so that arithmetic behaves as expected:

```
Eigen::Matrix3d m1, m2 ;
Eigen::Vector3d v1, v2 ;

v2 = m1 * v1 ;  // Matrix multiplication
m2 = 5.0 * m1 ; // Scalar multiplication
v1 += v2 ;
m1 = m2.transpose() ;
```

Handy Member Functions

`Matrix<T>` has several handy member functions:

- Ctors:
  - `MatrixNt( nRows, nCols )`
  - `VectorNt( nElem )`
- `m.size()` - number of elements
- `m.cols()` - number of columns
- `m.rows()` - number of rows
- `m.resize()` - resize matrix

Reductions

Eigen provides several convenient matrix reduction operations:

`m.minCoeff()`	minimum coefficient
`m.maxCoeff()`	maximum coefficient
`m.prod()`	product of coefficients
`m.sum()`	sum of coefficients
`m.mean()`	mean of coefficients
`m.trace()`	trace of matrix
`m.colwise()`	peform reduction on each column
`m.rowwise()`	perform reduction on each row
`m.all()`	true iff all elements are true
`m.any()`	true iff any element is true
`m.lpNorm<p>()`	compute the ℓ^p norm
`m.lpNorm<Infinity>()`	compute the ℓ^∞ norm

Sub-Matrices & Sub-Vectors

	Matrix<T>	
`col()`	`topLeftCorner()`	`topRightCorner()`
`row()`	`bottomLeftCorner()`	`bottomRightCorner()`
`block()`	`topRows()`	`leftCols()`
`diagonal()`	`bottomRows()`	`rightCols()`

Vector<T>
`head()`
`tail()`
`segment()`

Array<T>

Array<T> has same template parameters as Matrix:

- Array<typename Scalar, int nRows, int nCols >
- Use for coefficient-wise operations
- Use Matrix<T> for linear algebra!
- * and / are like .* or ./ in MATLAB
- Use m.array() and a.matrix() to convert back and forth
- Sometimes you need both:

```
MatrixXf m(2,2);
MatrixXf n(2,2);
MatrixXf result(2,2);
m << 1,2,
     3,4 ;
n << 5,6,
     7,8 ;
result = (m.array() + 4).matrix() * m ;
result = (m.array() * n.array()).matrix() * m ;
```

Coefficient-Wise Operations

Can often avoid converting a `Matrix<T>` to an `Array<T>` by using coefficient-wise operations which operate on two matrices:

- `m.cwiseMin( m2 )`
- `m.cwiseMax( m2 )`
- `m.cwiseProduct( m2 )`
- `m.cwiseQuotient( m2 )`
- `m.cwiseAbs( m2 )`
- `m.cwiseInverse( m2 )`
- `m.cwiseSqrt( m2 )`

See manual for details

Advanced Operations

Overview of Common Operations

Overview of common operations:

- Solving $Ax = b$:
  - Do not compute the inverse of a matrix unless you need it!
  - Solve in place if possible
  - Choose an algorithm with the appropriate speed and accuracy
  - Can use m.inverse() to compute inverse for small matrices
- Common decompositions and algorithms:
  - LU
  - QR
  - Cholesky
  - Eigenvalues
- Check for errors or numerical problems such as ill-conditioning

Basic Linear Solving

To solve $Ax = b$ for x:

1. Choose a suitable solver:

   ```
   Matrix3d A ;
   Vector3d b ;
   ...
   FullPivLU< Matrix3d > lu( A ) ;
   ```

   Can also access most decompositions via a member function.

2. Solve the problem:

   ```
   Vector3d x = lu.solve( b ) ;
   ```

3. Check that solver was successful:

   ```
   if( lu.rank() != A.cols() )
     exit( -1 ) ;
   ```

Checking Linear Solution

The different decompositions provide several member functions which you can use to check if a solution was successful:

- m.rank() - use to check solution is full rank
- m.setThreshold() - sets threshold on rank-revealing algorithms to determine rank

Can also compute the relative error using the ℓ^2 norm() member function:

```
double dwRelErr = ( A * x - b ).norm() / b.norm() ;
```

Least Squares

To solve least squares problems:

- SVD is usually best
- LDLT can be faster but less reliable

```
MatrixXf A = MatrixXf::Random(3, 2) ;
cout << "A:\n" << A << endl ;
VectorXf b = VectorXf::Random(3) ;
cout << "b:\n" << b << endl ;
cout << "Solution:\n"
  << A.jacobiSvd( ComputeThinU | ComputeThinV ).solve(b)
  << endl ;
```

Include Files

Can optimize build time by choosing just the right header files:

- Easiest to #include “Eigen/Dense” for dense linear algebra
- May wish to choose header files with finer granularity to accelerate build times and use less memory:
  - #include “Eigen/Core”
  - #include other Eigen header files as appropriate
- See manual for details

Map<T>

Sometimes you have data from a source external to Eigen (such as a solver) and it does not make sense to copy data into a `Matrix<T>`:

- External library expects an array, such as a solver
- But also need to perform linear algebra
- Use `Eigen::Map<T>` to create an Eigen wrapper for accessing the data as if it were an `Eigen::Matrix<T>`:

```
double vJac[ 16 ] ;
...
Eigen::Map< Eigen::Matrix4d > mJac( vJac ) ;
Eigen::Map< Eigen::MatrixXd > mJac( vJac, 16 ) ;
```

- Make sure you have correct ordering – column major or row major!

Map<T> Example

```
integer n  = nTheta1 + nTheta2 + nShares + nIV ;
double *x  = double[n] ;
Eigen::Map< Eigen::Matrix< double,
            Eigen::Dynamic, 1 > > vXOptTmp( x, n ) ;
```

Map<T> Example II

```
int nElem = 3 ;
int m3i[ nElem ][ nElem ] ;
int count = 1 ;
for( int ix = 0 ; ix < nElem * nElem ; ++ix )
{
  for( int jx = 0 ; jx < nElem ; ++jx )
  {
    m3i[ ix ][ jx ] = count++ ;
  }
}
Eigen::Map<
    Eigen::Matrix<int, Eigen::Dynamic, Eigen::Dynamic,
                  Eigen::RowMajor >
          > mMap( &(m3i[0][0]), nElem , nElem ) ;
std::cout << "mMap:\n" << mMap << std::endl ;
```

Other Features

Eigen has several other features you may need:

- cast<T> member function casts all elements in a matrix:

```
BLPVectorXd vXOpt = vXOptTmp.cast< blp::BLPReal_t >() ;
```

- SparseMatrix:
  - Handy for representing matrices which are large and mostly zero
  - Eigen provides some sparse solvers as well
- See the manual for details

Performance Enhancements

To maximize performance:

- Compile without debug symbols and enable -O2
- Disable conformance checking via -DEIGEN_NO_DEBUG
- Use .noalias() if RHS can be safely evaluated into LHS without aliasing:

  ```
  c.noalias() += a * b ;
  ```

- Write matrix expressions to favor either row major or column major ordering
- Initialize matrix and use member functions with fixed sizes specified at compile time for m.size() <= 32.

Mersenne Twister

Random Numbers

Generating good random numbers is necessary for research:

- Monte Carlo studies
- Synthetic Data
- Starting guesses

Machine-generated random numbers are *pseudo*-random:

- Not truly random
- Pass some set of statistical tests
- Mersenne Twister is currently the best algorithm
- M. Matsumoto and T. Nishimura, *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1, January 1998.

Installation

Mersenne Twister is easy to install:

1. Download `Mersenne-1.1.tar.gz` from the course website
2. Unpack with tar:

   ```
   tar xzf Mersenne-1.1.tar.gz
   ```

3. `#include` “`MersenneTwister.h`” in your code – there is no need to build any libraries

This implementation is by Richard J. Wagner

Documentation

There is no documentation:

- See README file in installation
- See `example.cpp` file
- See MersenneTwister.h header file

Use

This implementation is very easy to use:

1. Instantiate an MTRand object
2. Use member functions to generate random numbers on desired interval (open, closed; normal, uniform)

Recommended usage:

```
#include ‘‘MersenneTwister.h’’
...
MTRand mtrand;    // automatically generate seed
double s = mtrand();                        // in [0,1]
double t = mtrand.randExc(0.5);             // in [0,0.5)
unsigned long u = mtrand.randInt(10); // in [0,10]
```

Other Issues

You should always set a seed so your work is reproducible:

```
MTRand mtrand( 1945 ) ; // Seed with 1945
```

- Test that results do not depend on the seed you have chosen.
- Do not use the system clock as a seed. It is not random...
- See `example.cpp` for using a more complex seed

Can generate Normally distributed random numbers:

mtrand.randNorm(μ, σ) ;

Other features:

- Can save and load streams of random numbers
- Can scale random numbers for desired interval