CS100
Introduction to Programming

Lecture 19. Introduction to Eigen

Outline

- Introduction & Motivation
- Platforms
- Eigen vs BLAS/Lapack
- Installing Eigen
- How it works
- Implementation of Eigen

Introduction

- A C++ template library for linear algebra
- Header only, nothing to install or compile
- Provides good speed, simple interface and usage
- Open-source

Why Another Library?

- Multi-platform and good compiler support
- A single unified library
- Most libraries specialized in one of the features or modules
- Eigen satisfies all criteria:

  -free, fast, versatile, reliable, decent API, support for both sparse and dense matrices, vectors and arrays, linear algebra algorithms (LU, QR, ...), geometric transformations, etc.

Platforms

- Supported compilers:
  - GCC (from 3.4 to 4.6) , MSVC (2005,2008,2010) , Intel ICC, Clang/LLVM
- Supported systems:
  - x86/x86_64 (Linux, Windows, OSX)
  - ARM (Linux), PowerPC
- Supported SIMD vectorization engines:
  - SSE2, SSE3, SSSE3, SSE4
  - NEON (ARM)
  - Altivec (PowerPC)

Eigen vs BLAS/Lapack

- Fixed size matrices, vectors
- Sparse matrices and vectors
- More features like Geometry module, Array module
- Most operations are faster or comparable with MKL and GOTO
- Better API
- Complex operations are faster

Installing Eigen

- We'll use Git to download the source files for Eigen into the user's home directory.

```
$ cd ~

$ git clone https://github.com/eigenteam/eigen-git-mirror
```

- We don't need to build anything here. Eigen uses pure header libraries:
  - simply `#include` these header files at the beginning of your own code.
  - You can find these headers in `~/eigen-git-mirror/Eigen/src`

Installing Eigen

- The compiler needs to be told the location on disk of any header files you include
- You can copy the ~/eigen-git-mirror/Eigen folder to each new project's directory
- Or, you can add the folder once to the compiler's default include path
  - Can do this by running the following command on Unix-based systems

  - `$ sudo ln -s /usr/local/include ~/eigen-git-mirror/Eigen`

Installing Eigen

- If you've followed the steps above, you should be able to compile the following piece of code without any additional configuration:

```
#include <Eigen/Core>

#include <iostream>

using namespace std;

using namespace Eigen;

int main() {

   cout << "Eigen version:" << EIGEN_MAJOR_VERSION << "."

                                  << EIGEN_MINOR_VERSION << endl;

   return 0;

}
```

How it works

- In Eigen, all matrices and vectors are objects of the Matrix template class
- Vectors are just a special case of matrices, with either 1 row or 1 column
- Takes 3 compulsory and 3 optional arguments

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

How it works

- Could be different types

```
typedef Matrix<float, 4, 4> Matrix4f;
typedef Matrix<double, Dynamic, Dynamic> MatrixXd;
typedef Matrix<float, 3, 1> Vector3f;
typedef Matrix<int, 1, 2> RowVector2i;
```

How it works

- The Array class provides general-purpose arrays, as opposed to the Matrix class which is intended for linear algebra
- The Array class provides an easy way to perform coefficient-wise operations, which might not have a linear algebraic meaning, such as adding a constant to every coefficient in the array or multiplying two arrays coefficient-wise
- Array is a class template taking the same template parameters as Matrix. As with Matrix, the first three template parameters are mandatory:

```
Array<typename Scalar, int RowsAtCompileTime, int
ColsAtCompileTime>
```

The special value Dynamic

- Eigen is not limited to matrices whose dimensions are known at compile time
- For example, the convenience typedef **`MatrixXd`**, meaning a matrix of doubles with dynamic size, is defined as follows:

```
typedef Matrix<double, Dynamic, Dynamic> MatrixXd;
```

- And similarly, it defines a self-explanatory typedef **`VectorXi`** as follows:

```
typedef Matrix<int, Dynamic, 1> VectorXi;
```

Constructors

- A default constructor is always available, never performs any dynamic memory allocation, and never initializes the matrix coefficients. You can do:

```
Matrix3f a;    MatrixXf b;
```

- Constructors taking sizes are also available. They allocate the array of coefficients with the given size, but don't initialize the 系数 coefficients themselves:

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

- Eigen also offers some constructors to initialize the coefficients of small fixed-size vectors up to size 4:

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

Comma-initialization

- Matrix and vector coefficients can be conveniently set using comma-initialization syntax.

- Example:

```
Matrix3f m;
m << 1, 2, 3,
     4, 5, 6,
     7, 8, 9;
std::cout << m;
```

Output:

```
1 2 3
4 5 6
7 8 9
```

Special matrices and array initialization

- The Matrix and Array classes have static methods like `Zero()`, which can be used to initialize all coefficients to zero
- Similarly, the static method `Constant(value)` sets all coefficients to a certain value
  - If the size of the object needs to be specified, the additional arguments go before the value argument, as in `MatrixXd::Constant(rows, cols, value)`
- The method `Random()` fills the matrix or array with random coefficients
- The identity matrix can be obtained by calling `Identity();`

Coefficient accessors

- The primary coefficient accessors and mutators in Eigen are the overloaded parenthesis operators.
- For matrices, the row index is always passed first. For vectors, just pass one index. The numbering starts at 0.
- Example:

```
#include <iostream>
#include <Eigen/Dense>
using namespace Eigen;
int main()
{
  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 << "Here is the matrix m:\n" << m << std::endl;
  VectorXd v(2);
  v(0) = 4;
  v(1) = v(0) - 1;
  std::cout << "Here is the vector v:\n" << v << std::endl;
}
```

Output:

```
Here is the matrix m:
  3  -1
2.5 1.5
Here is the vector v:
4
3
```

Assignment and Resizing

- Size of a matrix is retrieved by **`rows(), cols()`** and **`size()`**
- Resizing a dynamic-size matrix done by the **`resize()`** method

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

- Assignment is the action of copying a matrix into another, using **`operator =`**. Eigen resizes the matrix on the left-hand side automatically so that it matches the size of the matrix on the right-hand size.
- If the left-hand side is of fixed size, resizing is not allowed

```
MatrixXf a(2,2);
MatrixXf b(3,3);
a = b;
```

Matrix and vector arithmetic

- **Addition and subtraction**
  - The left hand side and right hand side must have the same numbers of rows and of columns
  - They must also have the same Scalar type, as Eigen doesn't do automatic type promotion. The operators at hand here are:
    - binary operator + as in a+b
    - binary operator - as in a-b
    - unary operator - as in -a
    - compound operator += as in a+=b
    - compound operator -= as in a-=b

Matrix and vector arithmetic

- **Scalar multiplication and division**
  - Multiplication and division by a scalar is very simple too. The operators at hand here are:
    - binary operator * as in matrix*scalar
    - binary operator * as in scalar*matrix
    - binary operator / as in matrix/scalar
    - compound operator *= as in matrix*=scalar
    - compound operator /= as in matrix/=scalar

Matrix and vector arithmetic

- **Dot product and cross product**
  - For dot and cross product, you need the dot() and cross() methods. Of course, the dot product can also be obtained as a 1x1 matrix as u.adjoint()*v:
- Example:

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

Output:

```
Dot product: 8
Dot product via a matrix
product: 8
Cross product:
 1
-2
 1
```

Matrix and vector arithmetic

- **Basic arithmetic reduction operations**
  - Eigen provides some reduction operations to reduce a given matrix or vector to a **single** value such as the sum (computed by sum()), product (prod()), or the maximum (maxCoeff()) and minimum (minCoeff()) of all its coefficients:
- Example:

```
#include <iostream>
#include <Eigen/Dense>
using namespace std;
int main()
{
  Eigen::Matrix2d mat;
  mat  << 1, 2, 3, 4;
  cout << "Here is mat.sum(): " << mat.sum() << endl;
  cout << "Here is mat.prod(): " << mat.prod() << endl;
  cout << "Here is mat.mean(): " << mat.mean() << endl;
  cout << "Here is mat.minCoeff(): " << mat.minCoeff() << endl;
  cout << "Here is mat.maxCoeff(): " << mat.maxCoeff() << endl;
  cout << "Here is mat.trace(): " << mat.trace() << endl;
}
```

Output:

```
Here is mat.sum(): 10
Here is mat.prod(): 24
Here is mat.mean(): 2.5
Here is mat.minCoeff(): 1
Here is mat.maxCoeff(): 4
Here is mat.trace(): 5
```

Matrix and vector arithmetic

- **Norm computations**
  - Eigen provides the norm() method, which returns the square root of squaredNorm()
    - squaredNorm() is equal to the dot product of the vector by itself, and equivalently to the sum of squares of its coefficients
  - These operations can also operate on matrices; in that case, a n-by-p matrix is seen as a vector of size (n*p), so for example the norm() method returns the "Frobenius" or "Hilbert-Schmidt" norm.
  - Example:

```
VectorXf v(2);
MatrixXf m(2,2);
v << -1,2;
m << 1,-2,-3,4;
cout << "v.norm() = " << v.norm() << endl;
cout << "m.norm() = " << m.norm() << endl;
```

Output:

```
v.norm() = 2.23607
m.norm() = 5.47723
```

Block operations

- **Using block operations**
  - The most general block operation in Eigen is called .block() . There are two versions, whose syntax is as follows:
    - Block of size **`(p,q)`**, starting at **`(i,j)`**

    `matrix.block(i,j,p,q);` or **`matrix.block<p,q>(i,j);`**
  - Both versions can be used on fixed-size and dynamic-size matrices and arrays
  - These two expressions are semantically equivalent. The fixed-size version will typically give you faster code if the block size is small, but requires this size to be known at compile time

Block operations

- **Columns and rows**
  - Individual columns and rows are special cases of blocks. Eigen provides methods to easily address them: .col() and .row():
    - ith row * `matrix.row(i);`
    - jth column * `matrix.col(j);`
  - The argument for col() and row() is the index of the column or row to be accessed. As always in Eigen, indices start at 0

Block operations

- **Block operations for vectors**
  - Eigen also provides a set of block operations designed specifically for the special case of vectors and one-dimensional arrays:
    - Block containing the first n elements *

      ```
      vector.head(n);
      ```
    - Block containing the last n elements *

      ```
      vector.tail(n);
      ```
    - Block containing n elements, starting at position i *

      ```
      vector.segment(i,n);
      ```

Block operations

- Use special accessors for corner-located blocks
  - Faster!

Block operation	Version constructing a dynamic-size block expression	Version constructing a fixed-size block expression
Top-left p by q block *	`matrix.topLeftCorner(p,q);`	`matrix.topLeftCorner<p,q>();`
Bottom-left p by q block *	`matrix.bottomLeftCorner(p,q);`	`matrix.bottomLeftCorner<p,q>();`
Top-right p by q block *	`matrix.topRightCorner(p,q);`	`matrix.topRightCorner<p,q>();`
Bottom-right p by q block *	`matrix.bottomRightCorner(p,q);`	`matrix.bottomRightCorner<p,q>();`
Block containing the first q rows *	`matrix.topRows(q);`	`matrix.topRows<q>();`
Block containing the last q rows *	`matrix.bottomRows(q);`	`matrix.bottomRows<q>();`
Block containing the first p columns *	`matrix.leftCols(p);`	`matrix.leftCols<p>();`
Block containing the last q columns *	`matrix.rightCols(q);`	`matrix.rightCols<q>();`

Type casting

- Sometimes matrices are of different type
- A copy of a matrix with casted scalar type can be obtained as follows

```
Matrix3d aMatrix;
Matrix3f anotherMatrix =
    aMatrix.base().cast<float>();
```

Linear algebra and decompositions

- **Basic linear solving**
  - **The problem:** You have a system of equations, that you have written as a single matrix equation:

    ```
    Ax = b
    ```

    Where A and b are matrices (b could be a vector, as a special case). You want to find a solution x.
  - **The solution:** You can choose between various decompositions, depending on what your matrix A looks like, and depending on whether you favor speed or accuracy.

Linear algebra and decompositions

- Example:

```
#include <iostream>
#include <Eigen/Dense>
using namespace std;
using namespace Eigen;
int main()
{
   Matrix3f A;
   Vector3f b;
   A << 1,2,3,  4,5,6,  7,8,10;
   b << 3, 3, 4;
   cout << "Here is the matrix A:\n"
        << A << endl;
   cout << "Here is the vector b:\n"
        << b << endl;
   Vector3f x = A.colPivHouseholderQr().solve(b);
   cout << "The solution is:\n"
        << x << endl;
}
```

Output:

```
Here is the matrix A:
1 2 3
4 5 6
7 8 10
Here is the vector b:
3
3
4
The solution is:
-2
1
1
```

Linear algebra and decompositions

- In this example, the colPivHouseholderQr() method returns an object of class ColPivHouseholderQR
- Here is a table of some other decompositions that you can choose from, depending on your matrix and the trade-off you want to make:

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	-	- - -	+++

Linear algebra and decompositions

- **Computing eigenvalues and eigenvectors**
  - The computation of eigenvalues and eigenvectors does not necessarily converge, but such failure to converge is very rare. The call to info() is to check for this possibility

Example:	Output:

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

Linear algebra and decompositions

- **Computing inverse and determinant**
  - Inverse computations are often advantageously replaced by solve() operations, and the determinant is often not a good way of checking if a matrix is invertible
  - However, for very small matrices, the above is not true, and inverse and determinant can be very useful

Example:	Output:

`#include <iostream>`
`#include <Eigen/Dense>`
`using namespace std;`
`using namespace Eigen;`
`int main()`
`{`
`Matrix3f A;`
`A << 1, 2, 1,`
`2, 1, 0,`
`-1, 1, 2;`
`cout << "Here is the matrix A:\n" << A << endl;`
`cout << "The determinant of A is " << A.determinant() << endl;`
`cout << "The inverse of A is:\n" << A.inverse() << endl;`
`}`
	`Here is the matrix A:`
	` 1 2 1`
	` 2 1 0`
	`-1 1 2`
	`The determinant of A is -3`
	`The inverse of A is:`
	`-0.667     1  0.333`
	`  1.33    -1 -0.667`
	`    -1     1      1`

Solving linear least squares systems

- An overdetermined system of equations, say $Ax = b$, has no solutions.
  - In this case, it makes sense to search for the vector x which is closest to being a solution, in the sense that the difference $Ax - b$ is as small as possible. This x is called the least squares solution (if the Euclidean norm is used)
  - The three methods discussed are the SVD decomposition, the QR decomposition and normal equations
  - The SVD decomposition is generally the most accurate but the slowest, normal equations is the fastest but least accurate, and the QR decomposition is a trade-off

Solving linear least squares systems

- **Using the SVD decomposition**
  - The solve() method in the BDCSVD class can be directly used to solve linear squares systems. It is not enough to compute only the singular values (the default for this class); you also need the singular vectors but the thin SVD decomposition suffices for computing least squares solutions

Example:	Output:

`#include <iostream>`
`#include <Eigen/Dense>`
`using namespace std;`
`using namespace Eigen;`
`int main()`
`{`
`MatrixXf A = MatrixXf::Random(3, 2);`
`cout << "Here is the matrix A:\n" << A << endl;`
`VectorXf b = VectorXf::Random(3);`
`cout << "Here is the right hand side b:\n" << b << endl;`
`cout << "The least-squares solution is:\n"`
`<< A.bdcSvd(ComputeThinU | ComputeThinV).solve(b)`
`<< endl;`
`}`
	`Here is the matrix A:`
	`  0.68  0.597`
	`-0.211  0.823`
	` 0.566 -0.605`
	`Here is the right hand side b:`
	` -0.33`
	` 0.536`
	`-0.444`
	`The least-squares solution is:`
	` -0.67`
	` 0.314`

Solving linear least squares systems

- **Using the QR decomposition**
  - The solve() method in QR decomposition classes also computes the least squares solution
  - There are three QR decomposition classes: HouseholderQR (no pivoting, so fast but unstable), ColPivHouseholderQR (column pivoting, thus a bit slower but more accurate) and FullPivHouseholderQR (full pivoting, so slowest and most stable).

Example:	Output:

`MatrixXf A = MatrixXf::Random(3, 2);`
`VectorXf b = VectorXf::Random(3);`
`cout << "The solution using the QR decomposition is:\n"`
`<< A.colPivHouseholderQr().solve(b) << endl;`
	The solution using the QR decomposition is:
	-0.67
	0.314

Solving linear least squares systems

- **Using normal equations**
  - Finding the least squares solution of $Ax = b$ is equivalent to solving the normal equation $A^TAx = A^Tb$. This leads to the following code.

Example:	Output:

`MatrixXf A = MatrixXf::Random(3, 2);`
`VectorXf b = VectorXf::Random(3);`
`cout << "The solution using normal equations is:\n"`
`<< (A.transpose() * A).ldlt().solve(A.transpose() * b) << endl;`
	The solution using normal equations is:
	-0.67
	0.314

  - If the matrix A is ill-conditioned, then this is not a good method, because the condition number of A^TA is the square of the condition number of A. This means that you lose twice as many digits using normal equation than if you use the other methods