Advanced Programming for Scientific Computing (PACS)
Lecture title: A very short introduction to the Eigen Library

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2023/2024

The Eigen library

The Eigen library is a template library that provides classes for dense and sparse matrices as well as various algorithms for linear systems.
It is a header only library (since is based on templates) that uses Expression templates technique to be very fast while keeping the abstraction of high level operations, but you need to compile your program with optimization enabled (`-O3 -DNDEBUG`) to get the full speed.

Downloading the library

The `mk` module system has already Eigen installed. If you have a different linux box you can easily find the package. Or, go to the web site, download the library and follow the instructions. It is a template libary, so you do not have to compile anything, just install the eigen header files in the directory of choice (by default is `/usr/local/include/eigen3` (you have to use `sudo` to be able to write in that directory.

Using with `mk`

If you are using the `mk` module system, by loading the Eigen module you have automatically two environment variable set: (check with `env | grep Eigen`):

`mkEigenInc`	Where the include files are located.
`mkEigenPrefix`	Where the library is installed.

Compiling a code with the Eigen

You just have to indicate where the headers file are

```
g++ -c myprog.cpp -I<EigenDir>
```

For instance, if you use modules you can do

```
g++ -c myprog.cpp -I$mkEigenInc
```

In a `Makefile` you may set

```
CPPFLAGS+=$(mkEigenInc)
```

Remember the `()`!

Eigen

Creating a 2 by 2 matrix (fixed dimensions)

```
#include <iostream>
#include <Eigen/Dense>
int main()
{
 Eigen::Matrix<double,2,2> m;
 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;
}
```

Eigen

Using the overloaded streaming operator

Creating a 2 by 2 matrix (fixed dimensions)

```
#include <iostream>
#include <Eigen/Dense>
int main()
{
 Eigen::Matrix<double,2,2> m;
 m << 1.0, 2.0, 3.0, 4.0;
}
```

$$m = \begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix}$$

The namespace Eigen

All Eigen classes and objects are in the `Eigen` namespace. So their fully qualified name is always `Eigen::EigenObject`.

The various functionalities are obtained by including the relevant header files. The most important is

```
#include "Eigen/Core"
```

In the following slides, for the sake of space, we will omit `Eigen::`

Eigen

(Dense) Matrices

Generic matrix

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

Useful shortcuts

- Matrix with fixed dimensions

  ```
  using Matrix4f=Matrix<float, 4, 4>;
  ```

- Matrix with dynamic dimensions

  ```
  using MatrixXd=Matrix<double, Dynamic, Dynamic>;
  ```

- A column vector of integers

  ```
  using VectorXi=Matrix<int, Dynamic, 1>;
  ```

Dynamic matrices

```
// an empty 5x7 dynamic matrix
Eigen::Matrix<double,Dynamic,Dynamic> A(5,7);
...
A.resize(6,7) // matrix is resized
cout<<a.cols()// Will by 7
cout<<a.rows()// Will by 6
// A matrix with first dimensiondynamic
Eigen::Matrix<double,Dynamic,3> B;
B.resize(100,3);// second dymension must be 3
B.fill(30); // all elements=30
B(4,5)=-9.5;
```

Eigen

Initialization

- non-initialized matrix

```
MatrixXd m(2,2);
```

- initialized matrix

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

- assignment

```
m(0,1) = 7.0;
```

Note: If you want to see how the overloading of the comma operator is made, have a look at ReadingVectors/ReadingVector.hpp

Eigen

Arithmetics

Eigen implements all the arithmetic operators, and also

- transpose `a.transpose()`
- conjugate `a.conjugate()`
- adjoint `a.adjoint()`
- dot product `a.dot(b)`
- cross product `a.cross(b)`
- ...

You can extract the diagonal part or submatrices.

Eigen

Expression Templates

The code

```
VectorXf a(50), b(50), c(50), d(50);
...
a = 3*b + 4*c + 5*d;
```

is implemented in such a way that it is equivalent to the execution of

```
for(int i = 0; i < 50; ++i)
  a[i] = 3*b[i] + 4*c[i] + 5*d[i];
```

just like the most performant (and optimizable by the compiler) C code.

Eigen

Advantages & disadvantages

the efficiency of the code greatly improves, but expression of the type

```
a = b+ a.transpose();
```

are not allowed (no aliasing is allowed). Furthermore, compile time errors and execution time errors become mostly unreadable.

For dense matrices you have however `a.transposeInPlace()`, that transposes the matrix so `a=b+a.transposeInPlace()` is fine (but using transposition in place when not needed is less efficient).

Explanation

Methods like transpose() return a view of the matrix, embedded in a template expression, in this case of type TransposeReturnType. Often you have the `InPlace` alternative, which operates on the matrix, modifying it.

For sparse matrices, the API is less rich (for obvious reason) and the InPlace option is often noot viable. In that case the expression in the previous slides should be rewritten as (SpMAt is amn alias to the chosen sparse matrix type

```
SpMat at = a.transpose();
a=b+at;
// if you want to free memory
at.resize(0,0);
at.data().squeeze();
```

Random matrices

code

```
#include <iostream>
#include <Eigen/Dense>
int main() {
   Eigen::Matrix3f m = Eigen::Matrix3f::Random();
   std::cout << "m␣=" << std::endl << m ; }
```

You can define also other notable matrices

```
MatrixXd A;
A.setRandom(10,10);
MatrixXd B= MatrixXd::Random(100,100);
MatrixXd C= MatrixXd::Identity(100,100);
MatrixXd C= MatrixXd::Zero(100,100);
```

and more...

You have also the equivalent methods

code

```
cin>>n;
Matrix<double,3,Dynamic> m(3,n);
m.setZero();
m.row(0).setRandom();
```

You can extract blocks

```
Matrix<double,3,Dynamic> A(3,n);
...
Matrix<double,2,2>=A.block<2,2>(1,1);
Vector3d= A.col(2);// extract column
MatrixXd= A.topLeftCorner(2,2);
```

and more...

Column-major and row-major storage I

We say that a matrix is stored in row-major order if it is stored row by row. The entire first row is stored first, followed by the entire second row, and so on. Consider for example the matrix. it E.g.:

$$A = \begin{bmatrix} 8 & 2 & 2 & 9 \\ 9 & 1 & 4 & 4 \\ 3 & 5 & 4 & 5 \end{bmatrix}$$

Column major storage

$$[8\ 9\ 3\ 2\ 1\ 5\ 2\ 4\ 4\ 9\ 4\ 5]$$

Row major storage

$$[8\ 2\ 2\ 9\ 9\ 1\ 4\ 4\ 3\ 5\ 4\ 5]$$

Column-major and row-major storage II

Example:

```
Matrix<int, 3, 4, ColMajor> Acolmajor;
Acolmajor << 8, 2, 2, 9,
             9, 1, 4, 4,
             3, 5, 4, 5;
cout << "The matrix A:" << endl;
cout << Acolmajor << endl << endl;
cout << "In memory (column-major):" << endl;
for (int i = 0; i < Acolmajor.size(); i++)
  cout << *(Acolmajor.data() + i) << "  ";
cout << endl << endl;
Matrix<int, 3, 4, RowMajor> Arowmajor = Acolmajor;
cout << "In memory (row-major):" << endl;
for (int i = 0; i < Arowmajor.size(); i++)
  cout << *(Arowmajor.data() + i) << "  ";
cout << endl;
```

Column-major and row-major storage III

Output:

```
The matrix A:
8 2 2 9
9 1 4 4
3 5 4 5

In memory (column–major):
8   9   3   2   1   5   2   4   4   9   4   5

In memory (row–major):
8   2   2   9   9   1   4   4   3   5   4   5
```

Column major is the default!.

Interfacing with raw buffers: the Map class I

Eigen allows to work with "raw" C/C++ arrays using the Map class.

A Map object has a type defined by its Eigen equivalent

```
Map<Matrix<typename Scalar, int RowsAtCompTime, int ColsAtCompTime>>
```

Example:

```
using MapType=Map<VectorXd>; //a map to a vector
std::vector<double> a(10,10.);
MapType mappedV(a.data(),10);
std::cout<<mappedV<<std::endl;
VectorXd b(10);
b.setRandom();
std::cout<<(b-mappedV).squaredNorm()<<std::endl;
Map<Matrix<double,2,5>> map2(a.data()); // get a matrix map
```

Note: data() is a method of an Eigen matrix and it returns a pointer to the data buffer.

Eigen Matrices and Arrays

In the Eigen world a Matrix is a class that complies to the concept of a matrix in linear algebra: the representation of a linear operator $\mathbb{R}^n \rightarrow \mathbb{R}^m$.

However the Eigen library introduces also the Eigen::Array<> template classes to provide general-purpose arrays. The syntax of the constructors of a Eigen `Matrix` and a `Array` are similar. The main difference is that operation on arrays are made component-wise.

The interesting thing is that you can view a Eigen `Matrix` as an `Array` and viceversa.

Eigen Matrices and Arrays

```
int main()
{
// Alias for a dynamic array
  ArrayXf a(3,3), b(3,3);
  a << 1,2,3,
       4,5,6,
       7,8,9;
  b << 1,2,3,
       1,2,3,
       1,2,3;
  ArrayXXf c=a*b;
```

Array c will contain the component-wise product: $c_{ij} = a_{ij}b_{ij}$.

Converting between Eigen Array and Matrix expressions

Sometimes one wants to do component-wise operation on a Matrix expression (or viceversa). Eigen makes that simple:

```
MatrixXf m(2,2);
MatrixXf n(2,2);
MatrixXf result(2,2);
// this is standard matrix product
result = m * n;
// This is component-wise product
result = m.array() * n.array();
// But you can do it also in this way
result = m.cwiseProduct(n);
// Are adding 4 to all elements of m
// This operation is not allowed on Matrices.
result = m.array() + 4;
// After adding 4 we multiply the result with n
// using standard matrix product
result = (m.array() + 4).matrix() * n;
```

Sparse Matrices

The interface for sparse matrices is more recent, but it is now well developed. It is defined by

```
template<typename Scalar, int Options, typename StorageIndex>
class Eigen::SparseMatrix<Scalar, Options, StorageIndex >
```

where Scalar is the type of values contained in the matrix, Options is a flag that can be ColumnMajor (the default) or RowMajor and StorageIndex is the type used for the indexes (defaulted to **int**).

```
// column-major compressed sparse matrix of complex<float>
SparseMatrix<std::complex<float> > mat(1000,2000);
//1000x2000 row-major compressed sparse matrix of double
SparseMatrix<double,RowMajor> mat(1000,2000);
//a column sparse vector of complex<float> of size 1000
SparseVector<std::complex<float> > vec(1000);
```

Compressed and uncompressed state

Filling up a sparse matrix is a complex operation because the adjacency graph of the matrix may change during insertion of a new element.
So during insertion operations (with the method `insert(i,j)`) the matrix is kept in uncompressed mode: it uses a trick (explained in the web page) to make insertion less costly.

However, to have a operative matrix (for instance to solve a linear system) you need to have it in compressed mode. This is done with the method `makeCompressed()`. It is a very cheap method so use it also if in doubt.

Filling a sparse matrix

You can fill an element at a time using the method insert

```
A.insert(1,2)=0.4;
```

or (much more efficient), using Eigen::Triplets containing (i,j,value):

```
using T=Eigen::Triplet<double>;
std::vector<T> coeff;
...
coeff.emplace_back(3,5,8.9);// Fill coeff
...
A.setFromTriplets(coeff.begin(), coeff.end());
```

Note: setFromTriplets() returns the matrix in compressed form, insert does not. insert can be used only if the element is not already present! (see next slide)

Accessing an element in a sparse Eigen matrix

In a sparse Eigen Matrix you do not have **operator**()(**int int**) to access elements!. The reason is that the implementation of the const and non-const version of that operator is dangerous.
In a sparse matrix you must have a way to distinguish operation that just read a value from those that potentially insert a value.
You can access an element with coeff and coeffRef

```
double a = S.coeff(3,3);
```

coeff(i,j) returns the value of the element stored in the matrix or the value 0 if the element is not present. In both cases the matrix is unchanged.

```
S.coeffRef(3,3)=10.;
```

coeffRef(i,j) returns a reference to the element stored in the matrix, or the reference to a newly created element. In the latter case, the matrix will be left in an uncompressed state.

Compatibility with dense matrices and sparse vectors

For compatibility, coeff() and coeffRef() are available also in dense matrices, where however you have the more convenient call operator (whose use is advised). Note: try to use vectorised operations as far as possible, avoiding accessing single elements (it is more efficient, as is Matlab).

You also have Eigen::SparseVector, for sparse vectors. However, differently than in the dense counterpart, there are not just alias to a SparseMatrix with one column (but they are convertible to it).

Sparse matrix interface

Some useful methods

```
A.makeCompressed();// Make sure that the matrix is compressed
auto nz=A.nonZeros();// Number of non zeros
A.size();// The total size n X m
auto v = A.coeff(i,j);// the value A_{ij} (read only)
A.coeffRef(i,j)=5.4;// A reference to A_{ij}
A.resize(m,n);// resizes the matrix (old element canceled)
A.conservativeResize(m,n);// old elements kept
```

Beware: in the `coeffRef` statement if A_{ij} does not exist a new element will be inserted and the matrix turns out to a non-compressed state.

Sparse matrix interface

Iterating over non-zero elements

Iterating over elements of a sparse matrix is more complex because you need to address only the non zero elements. Moreover, you want to operate traversing rows or columns depending on the ordering. Here how to do it using iterators.

```
SparseMatrix<double> mat(rows,cols);// column ordering
...
for (int k=0; k<mat.outerSize(); ++k)
  for (SparseMatrix<double>::InnerIterator it(mat,k); it; ++it)
  {
    it.value();
    it.row();   // row index
    it.col();   // col index (here it is equal to k)
    it.index(); // inner index, here it is equal to it.row()
  }
```

Sparse matrix interface

sparse-dense op

```
dv2 = sm1 * dv1;
dm2 = dm1 * sm1.adjoint();
dm2 = 2. * sm1 * dm1;
```

sparse-sparse

```
sm3 = sm1 * sm2;
sm3 = 4 * sm1.adjoint() * sm2;
```

pruning

```
sm3 = (sm1 * sm2).pruned();// removes numerical zeros
//removes elements much smaller than ref
sm3 = (sm1 * sm2).pruned(ref);
// removes elements smaller than ref*epsilon
sm3 = (sm1 * sm2).pruned(ref,epsilon);
```

Map for sparse matrices

Also with sparse matrices we can map external buffer to an equivalent sparse matrix type.You need to give the inner and outer indexes, whose meaning depends on the chosen ordering:

- Values: stores the coefficient values of the non-zeros.
- InnerIndices: stores the row (resp. column) indices of the non-zeros.
- OuterStarts: stores for each column (resp. row) the index of the first non-zero in the previous two arrays.

In parenthesis the meaning for rowMajor ordering.

An example of map for sparse matrices

$$\begin{bmatrix} 0 & 0 & 4 & 4 \\ 1 & 0 & 0 & -3 \\ 0 & -1 & 0 & 0 \end{bmatrix}$$

```
using namespace Eigen;
using SpMap=Map<SparseMatrix<double,RowMajor>>;
std::vector<Index> innerIndices{2,3,0,3,1};
std::vector<Index> outerStart{0,2,4};
std::vector<double> values{4.,4.,1.,-3.,-1.};
Index nnz=5;
SpMap sm(3,4,5,outeStart.data(),innerIndeces.data(),values.data());
```

Now sm behaves as a sparse matrix and can be used in Eigen in any occasions where you can use a sparse matrix.
The chosen format corresponds to the Compressed Row Format, a standard format for sparse matrices. This allows interfacing with other linear algebra software easily. (The ColMajor Eigen internal compressed representation correspond to Compressed Sparse Column).

Much more...

You can do block operations, or activate views (upper/lower triangular...) with triangularView() and other view methods.

You can permute the matrix, do most common factorizations, solve linear systems.... well all is described in the web page, some information also i the next slides.

Solving linear system: direct methods for dense matrices

You have an extensive set of tools based on LU, Cholesky or QR factorization:

```
#include <Eigen/LU>
MatrixXd m;
VectorXd b;
...
FullPivLU<MatrixXd> lu(m);// full pivoting
VectorXd x = lu.solve(b); // solve Ax=b
```

Eigen direct solvers for sparse matrices

We have different direct solvers (superLU, UMFPACK) available....
The interface is uniform (see here for details)

```
#include <Eigen/RequiredModuleName>
// ...
SparseMatrix<double> A;
VectorXd b, x;
// Read A and b
SolverClassName<SparseMatrix<double> > solver;
solver.compute(A);
if(solver.info()!=Success) {
// decomposition failed ...}
x = solver.solve(b);
if(solver.info()!=Success) {//solving failed...}
// solve for another right hand side:
x1 = solver.solve(b1);
```

Direct solvers for sparse matrices

To use the UMFPACK eigen module you need to include `Eigen/UmfPackSupport` and link against the umfpack library!. The latter are normally shipped with the sparsesuite package.

The sparseLU solver is instead integrated directly in the Eigen (inspired by the original sparseLU pakage). You do not need to link against an external library, but UMFPACK is (in my opinion) better!

Iterative solvers and preconditioners

Eigen provides also a (not so extensive) set of iterative solvers and preconditioners and support for matrix free resolution.

An interface with the Iterative Method Library (IML++) is in LinearAlgebra/IML_Eigen.

In LinearAlgebra/IML_Eigen/SaddlePointSolver you have a working program for the iterative solution of saddle point matrices that makes use of the utility in LinearAlgebra/SparseBlockMatrix to handle sparse block matrices.

QR factorization

QR factorization is the method of choice for least square problems.
The Eigen library provide a full set of algorithms, some of which are rank revealing, i.e. can be used also in rank deficient matrices.
You have the description here.
Some QR factorizations are also implemented as methods of dense matrix classes.

The full set of matrix decompositions provided by the library (only for dense matrices) are here.

SVD

Of course you can also do singular value decomposition (useful for PCA). You have an accurate method based on Jacobi iterations (but not good if matrix is large) and a less precise method that however is faster on (relatively) large matrix.

You find them here.

And, of course, you have a set of tools for the computation of eigenvalues/vectors, see here

What about eigenvalues of sparse matrices?

Eigen provides also a sparseQR module for performing QR factorization on sparse matrices with reduced fill-in. To that purpose you need to link to an external library to do the appropriate reardering of the matrix. You may choose COLAMD or AMD (available with the suitesparse package mentioned previously) or METIS (a tool for graph partitioning).

I also provide in LinearAlgebra/spectra a fork of the spectra library for sparse eigenvalue computations and LinearAlgebra/redsvd-h, a tool for randomized singular value decompositions, good for principal component analysis on large matrices, when you are interested only on the most dominant components.

Type aliases for Eigen matrices

As customary in generic programming, the Eigen library contains also a set of useful type aliases:
Eigen::Index The type used for matrix indexes

In any Eigen Matrix type we have also the following type aliases:
Scalar The type of the elements stored in the matrix.
StorageKind Either Eigen::Dense or Eigen::Sparse

The boolean member variable IsRowMajor contained in all Matrix classes can be used to test storage ordering.

Eigen unsupported modules

Besides the modules of the stable versions of Eigen, you have a set of user provided module in Eigen unsupported. A lot of stuff, among which modules for tensors and automatic differentiation. Use with care, unsupported means... unsupported. Documentation is sometimes scarce.

Other Utilities

In LinearAlgebra/Utilities you have

MM_readers.hpp An enhanced tool to read matrices in Matrix Market Format (see later);

basicZeroFun.hpp Basic schemes for the zero of a scalar function;

basicOptimization.hpp Basic schemes for the minimum of a scalar function;

RotatingVector.hpp A structure that keep the last n inserted items. Available a version for matrices where you want to keep only the last n inserted columns.

In LinearAlgebra/CppNumericalSolvers you have a fork of this library for optimization problems, which uses Eigen matrices. Not a great documentation, but the examples are quite clear.

Looking for a matrix?

If you need to test your code on a matrix with certain characteristics, you can get them from MatrixMarket or the SuiteSparse Matrix Collection.

The Matrix Market format is one of the available formats for files containing sparse matrices. It is very simple. In LinearAlgebra/MatrixMarketReadersMatlab you have readers for Matlab. You have similar utilities also for R.

Some caveats: passing Eigen objects as argument

The use of expression templates in `Eigen` results in potentially every expression being of a different type. If you pass such an expression to a function taking a parameter of type Matrix, your expression will implicitly be evaluated into a temporary Matrix, which will then be passed to the function. This means that you lose the benefit of expression templates.
It is an issue if the matrix object has to be used in operations that benefit of expression templates.

Passing Eigen objects as arguments: the less efficient way

```
#include <Eigen/Core>
using namespace Eigen;
void print_size(MatrixXd const & b)
{
  std::cout << "size␣(rows,␣cols):␣" <<
    b.size() << "␣(" << b.rows()
    << ",␣" << b.cols() << ")" << std::endl;
}
int main()
{
    MatrixXd m;
    ...
    print_size(m);// here it is ok
// v.asDiagonal() returns a diagonal matrix expression
    print_size(m.asDiagonal()); // here we may be wasting memory!
}
```

Why is it inefficient?

The program works fine so you may think that nothing is wrong! However, v.asDiagonal() (which is the equivalent of the command `diag` in MATLAB) does not return a Eigen::Matrix, but a matrix expression, a view where the elements of v are not replicated but just referenced (remember the lecture on Expression Templates!).

However, if you pass it in the way described above you force your program to create a temporary matrix, in order to copy the elements of v.asDiagonal() in the function parameter b!

The right way of doing it

```
template <typename Derived>
void print_size(const MatrixBase<Derived>& b)
{
....
}
```

I refer to the base class of all dense eigen vectors and matrices, which is a template class called MatrixBase<D>.It is an implementation of Curiously Recurring Template Pattern and Expression Templates.

All basis classes of Eigen

MatrixBase<D>: The common base class for all dense matrix expressions (but not array expressions or sparse and special matrix classes). Use it in functions that are meant to work only on dense matrices.

ArrayBase<D>: The common base class for all dense array expressions. Use it in functions that are meant to work only on arrays.

DenseBase<T>: The common base class for all dense matrix/array expression. It can be used in functions that are meant to work on both matrices and arrays.

EigenBase<D>: The base class unifying all types of objects that can be evaluated into dense matrices or arrays (like special matrix classes such as diagonal matrices, permutation matrices, etc). It can be used in functions that are meant to work on any such general type.

All basis classes of Eigen

SparseMatrixBase<D>: The common base class for all sparse matrix expressions. Use it in functions that are meant to work only on sparse matrices.

A note: the technique of using constructs of the type

```
<template<class Derived>
void f(BaseClass<Derived> const & m)
```

to pass Eigen matrices/array etc to a function is not required when you do actually need the final object, not just the expression encapsulating it. However it does not any harm using it also in this case.
Often also simply

```
template<class EigenObject>
void f(EigenObject const & m)
```

works.

Returning Eigen objects

You may use decltype(**auto**) to be sure to return exactly what a Eigen function returns (use with care: check that you are not creating a dangling reference in the returned expression)

```
template<class M1, class M2>
decltype(auto) transposeProduct(const M1 & m1, const M2 & m2)
{
 return m1*m2.transpose();
}
```

Or, maybe better,

```
template<class M1, class M2>
decltype(auto) transposeProduct(const EigenBase<M1> & m1,
                                const EigenBase<M2> & m2)
{
return m1*m2.transpose();
}
```