Using C++/Eigen3

Chris Richardson
chris@bpi.cam.ac.uk

What is Eigen3 and why use it?

- C++ library for matrix arithmetic
- “Header only” implementation: no libraries to compile and install (easy)
- Provides some “missing” features needed for scientific computing in C++
- Contains optimisations to get good performance out of ARM & Intel processors
- Easy to use interface
- Support for dense and sparse matrices, vectors and “arrays”
- Some support for ‘solvers’ (A.x = b)
- Download from eigen.tuxfamily.org or e.g. apt install libeigen3-dev
- If you know Python, it is a bit like a NumPy for C++

Basics

```
#include <Eigen/Dense>

int main()
{
   Eigen::Matrix<double, 10, 10> A;
   A.setZero();
   A(9, 0) = 1.234;
   std::cout << A << std::endl;
   return 0;
}
```

This is pretty similar to: `double A[10][10];`

Dynamic size

```
int n = 64;
int m = 65;
Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> A(n, m);

A.resize(20, 20);

std::cout << “Size is ”;
std::cout << A.rows() << “ x ” << A.cols() << std::endl;
```

So this is more like a 2D version of : `std::vector<double>;`

Convenience Typedefs

```
Eigen::Matrix3d = Eigen::Matrix<double, 3, 3>

Eigen::Matrix3i = Eigen::Matrix<int, 3, 3>

Eigen::MatrixXd = Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>

Eigen::VectorXd = Eigen::Matrix<double, Eigen::Dynamic, 1>

Eigen::RowVectorXd = Eigen::Matrix<double, 1, Eigen::Dynamic>

etc.
```

You can do Matrix arithmetic...

```
Eigen::MatrixXd A(5, 10);

Eigen::MatrixXd B(10, 2);

Eigen::VectorXd vec(10);

Eigen::MatrixXd C = A * B;

Eigen::VectorXd w = A * vec;
```

Also dot and cross products for vectors, transpose, and usual scalar arithmetic +-*/

You can also do ‘Array’ arithmetic for coefficient-wise ops

e.g.

```
Eigen::Matrix3d A, B;

A.array() = 2.0; // set all values to 2.0

A.array() = B.array().sin();  // set each element of A to sin() of the same element in B

Eigen::Array3d W;

W = W * A;  // Error - cannot mix Array with Matrix
```

Mix and match with std::vector or any contiguous layout

It is easy to “overlay” existing memory with an Eigen Array or Matrix:

```
std::vector<double> a(1000);

Eigen::Map<Eigen::Matrix<double, 100, 10>> a_eigen(a.data());

a_eigen(10, 0) = 1.0;

Eigen::Map<Eigen::MatrixXd> a2_eigen(a.data(), 10, 100);
```

Efficiency: Eigen does lots of checks in debug mode

- Turn optimisation on. -O2 etc.

Walk through example

Solve diffusion equation in 1D using an explicit method...

$$\frac{\partial T}{\partial t} = k\frac{\partial^2 T}{\partial x^2}$$

Each timestep can be solved by using $T^{(n+1)} = A.T^{(n)}$

1. Create an initial vector for T
2. Create a dense matrix for A
3. Matrix multiply several times

- Convert to an implicit method: $A.T^{(n+1)} = T^{(n)}$
- Sparse matrices

Follow/type-along with me, or go to http://bit.ly/2UTzixC

Diffusion equation (explicit)

T(i) = T(i) + (k*dt/dx^2)[T(i-1) - 2*T(i) + T(i+1)]

Left-hand side is unknown (next time step)

$$T^{(n+1)} = A.T^{(n)}$$

Let: delta = (k*dt/dx^2)

```
A = [ 1-□ □     0    0    0         ...]
    [ □   1-2□ □     0    0         ...]
    [ 0   □    1-2□ □     0  0      ...]
    [ 0   0    □    1-2□ □   0  0   ...]
    [ 0   0    0                   ...]
```

Diffusion equation (implicit)

T(i) - (k*dt/dx^2)[T(i-1) - 2*T(i) + T(i+1)] = T(i)

Left-hand side is unknown (next time step)

$$A.T^{(n+1)} = T^{(n)}$$

Let: delta = (k*dt/dx^2)