Software libraries: Eigen

Victor Eijkhout, Susan Lindsey

Fall 2024

`last formatted: November 24, 2024`

1. Linear algebra

- Linear algebra is very important in science
- You're not the first person to need linear algebra software
- $\Rightarrow$ many libraries in existence
- many old libraries in Fortran, C

2. Eigen library

```
// eigen/matvec.cpp
#include <Eigen/Dense>
using namespace Eigen;
```

- Eigen library: idiomatic C++
- Many common operations
- High performance

Linear algebra objects

3. Dense matrices and vectors

- Matrix size as template parameter:

```
// eigen/matvec.cpp
const int siz=5;
Matrix<double,siz,siz> A;
Vector<double,siz> sol,rhs,tmp;
```

- Dynamic definition:

```
// eigen/matvec.cpp
Matrix<float,Dynamic,Dynamic> Af(20,20);
// or: MatrixXf Af(20,20);
```

(last character: use '`f`' for float, '`i`' for `int`, and '`d`' for `double`)

4. Elementary operations

- Indexing with parentheses: `A(i,j)*v(j)`
- Methods `rows`,`cols`,`size`
- Range over vector, output:

```
// eigen/matvec.cpp
for ( auto& v : rhs )
  v = 1.;
cout << rhs << '\n';
```

- `seq(f,t)` is the sequence $f, f+1, \ldots, t$:

```
A( seq(0,5),seq(3,10) );
```

- Subblock of size $p \times q$ starting at (i,j):

```
A.block(i,j,p,q);
```

Exercise 1: Matrix vector product

Code the matrix-vector product with a matrix `A` and vector `x`.
Compare the result to writing

```
y = A*x;
```

5. Jacobi method

Matrix-vector product with x unknown:

$$Ax = f \Leftrightarrow \forall_i \colon \sum_j a_{ij} x_j = f_i$$

repeatedly solve:

$$x_i^{(n+1)} = \bigl(f_i - \sum_{j\neq i} a_{ij} x_j^{(n)}\bigr) / a_{ii}$$

Speed of convergence depends on matrix sizes and diagonal dominance:

$$\forall_i \colon a_{ii} > \sum_{j\neq i} |a_{ij}|$$

Exercise 2: Jacobi method

- Code the jacobi; iterate a few steps
- Write a second version using *seq* or *block*
  compare to the first for correctness
- Optionally, can you write a version using ranges?

(Use the *matvec* skeleton in the repository.)

- Since you know the exact solution, iterate until a certain precision.
- How does the number of iterations depend on matrix size and amount of diagonal dominance?

Exercise 3: Optional: commandline options

Use the *cxxopts* library to let your code accept commandline argument;
supply a script that invokes your code with some set of inputs

Operations

6. Solving $Ax = b$

Multiply by inverse: $x = A^{-1}b$:

```
// eigen/invert.cpp
auto sol1 = A.inverse() * rhs;
```

Use Gaussian elimination:

$$A = LU, \qquad Ly = f, \qquad Ux = y$$

```
// eigen/invert.cpp
PartialPivLU<MatrixXd> LU(A);
auto sol2 = LU.solve(rhs);
```

Exercise 4: System solving

Diagonal dominance:

$$\forall_i\colon a_{ii} > \sum_{j\neq i} |a_{ij}|$$

- Solve the linear system by both methods above. How accurate is the solution?
- Experimment with different matrix sizes and amounts of diagonal dominance.

Exercise 5: Vandermonde matrix

$$a_{ij} = x_i^j$$

where $\{x_i\}_i$ sequence without duplicates

- Again solve the system both ways. Observations?

7. Singular values

```
// eigen/invert.cpp
JacobiSVD<MatrixXd> svd(A);
VectorXd sigmas = svd.singularValues();
```

- Singular values are a little like eigenvalues
- The spread in singular values indicates how 'difficult' the matrix is
- $\Rightarrow$ 'condition number': largest divided by smallest eigenvalue

Exercise 6: Commandline options

Use commandline options to:

- set the size of the matrix
- switch between diagonal dominant and vandermonde matrix
- for the former case set the diagonal dominance