Numerical Simulation and Scientific Computing I

Lecture 5:
FP Arithmetics, Direct Solvers, LA Libraries, Jacobi Method

Xaver Klemenschits, Paul Manstetten, and Josef Weinbub

Institute for Microelectronics
TU Wien

nssc@iue.tuwien.ac.at

WS 2022

Quiz

- Q1: What are the consequences/differences when using the Maximum norm or Euclidean norm to quantify the residual?
- Q2: Which of the discussed matrix norms is the ‘cheapest’ in terms of computational effort?
- Q3: What is the binary representation of “1000” in the IEEE 754 16bit/32bit/64bit FP format?
- Q4: What is the difference between BLAS routine ‘dgemm’ and ‘sgemm’ / What does the LAPACK routine ‘dsysv’ do?
- Q5: When would you advise to perform a LU decomposition of a matrix instead of a QR decomposition?

Q3

- IEEE FP32 “1000.0f”

```
template <typename FLOAT> union bitview {
  FLOAT f;
  constexpr static std::size_t bitsize = sizeof(FLOAT) * 8;
  std::bitset<bitsize> bits;
}
```

01000100011110100000000000000000
0,10001000,11110100000000000000000

Outline

- IEEE FP representations/arithmetic
- IEEE FP code snippets
- Matrix properties
- Matrix decompositions (direct solvers)
- LA libraries
- LA libraries code snippets
- Jacobi method

Finite Precision

$$u_h - \hat{u}_h = e_{\text{algeb.}} \qquad u - u_h = e_{\text{disc.}}$$

- We assumed algebraic error << discretization error
  - Condition number of the problem
  - Solver (e.g., number of iterations)
  - Finite representations and arithmetics used during calculation (influence on solving procedure)

- Example
  - Ill-conditioned system (for small beta)
  - Double precision FP
    - ~16 digits decimal precision
  - Single precision FP
    - ~7digits decimal precision

$$\underbrace{\begin{bmatrix} 0 & -1 \\ 0+\beta & -1 \end{bmatrix}}_{A} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 1+\beta \end{bmatrix}$$

$$y = -1$$

$$x = \frac{1+\beta-1}{\beta} = 1$$

$$\beta = 1\mathrm{e}^{-7} \rightarrow \kappa(A) = 2\mathrm{e}^{7}$$

$$\beta = 1\mathrm{e}^{-15} \rightarrow \kappa(A) = 2\mathrm{e}^{15}$$

Base 2 Floating Point Representation

- Advantages
  - Hardware implementation
  - Error analysis has tight bounds
  - Extra bit of precision (through normalization)

$$(\pm)\cdot 1.dddddddddd\cdot 2^{eeeee}$$

- Number of significant decimal digits

$$2^{(binary\ precision)} = 10^{(decimal\ precision)}$$

$$\log_{10}(2^{(binary\ precision)}) = (decimal\ precision)$$

$$\log_{10}(2^{52+1}) \approx 16$$

$$\log_{10}(2^{23+1}) \approx 7.2$$

$$\log_{10}(2^{10+1}) \approx 3.3$$

Floating Point Representation

s	e	e	e	e	e	d	d	d	d	d	d	d	d	d	d

- IEEE 754 16bit floating point representation
  - Exponent with 5 digits “e”
  - Significant with 10 digits “d” (precision=10)
  - Sign encoded with 1 digits “s”
  - Base is 2, so digits are bits with state 0 or 1
  - Exponent
    - 00000 = subnormal numbers (for significant>0), otherwise zero
    - 00001 = min, 01111 = bias (=0), 11110 = max
    - 11111 = NaN (for significant>0), otherwise +-infinity

$$(\pm)\cdot d.dddddddddd\cdot 2^{eeeee}$$

$$(\pm)\cdot 2^{00000}\cdot 0.0000000000=\pm 1\cdot 2^{0}\cdot\left(0+\frac{0}{2^{10}}\right)=\pm 0$$

$$(\pm)\cdot 2^{01111}\cdot 1.0000000000=\pm 1\cdot 2^{15-15}\cdot\left(1+\frac{0}{2^{10}}\right)=\pm 1$$

$$(\pm)\cdot 2^{11110}\cdot 1.1111111111=\pm 1\cdot 2^{30-15}\cdot\left(1+\frac{2^{10}-1}{2^{10}}\right)=\pm 65504$$

Floating Point Representation

s	e	e	e	e	e	e	e	e	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d	d

- IEEE 754 32bit floating point representation
  - Exponent with 8 digits “e”
  - Significant with 23 digits “d” (precision=10)
  - Exponent
    - 00000000 = subnormal numbers (for significant>0), otherwise zero
    - 00000001 = min, 01111111 = bias, 11111110 = max
    - 11111111 = NaN (for significant>0), otherwise +-infinity

$$(\pm)\cdot 2^{eeeeeeee}\cdot d.ddd\cdots dddd$$

$$(\pm)\cdot 2^{00000000}\cdot 0.000\cdots 0000 = \pm 1\cdot 2^{0}\cdot\left(0+\frac{0}{2^{23}}\right) = \pm 0$$

$$(\pm)\cdot 2^{01111111}\cdot 1.000\cdots 0000 = \pm 1\cdot 2^{127-127}\cdot\left(1+\frac{0}{2^{23}}\right) = \pm 1$$

$$(\pm)\cdot 2^{11111110}\cdot 1.111\cdots 11111 = \pm 1\cdot 2^{254-127}\cdot\left(1+\frac{2^{23}-1}{2^{23}}\right) = \pm 3.4028234664\mathrm{e}^{38}$$

Floating Point Representation

seeeeeeeeeeedddddddddddddddddddddddddddddddddddddddddddddddddddd

- IEEE 754 64bit floating point representation
  - Exponent with 11 digits “e”
  - Significant with 52 digits “d” (precision=52)

$$(\pm)\cdot 2^{ee\cdots eee}\cdot d.ddd\cdots dddd$$

$$(\pm)\cdot 2^{00\cdots 000}\cdot 0.000\cdots 0000=\pm 1\cdot 2^{0}\cdot\left(0+\frac{0}{2^{52}}\right)=\pm 0$$

$$(\pm)\cdot 2^{00\cdots 000}\cdot 0.000\cdots 0001=\pm 1\cdot 2^{1-1023}\cdot\left(0+\frac{1}{2^{52}}\right)=\text{smallest subnormal number}$$

$$(\pm)\cdot 2^{01\cdots 111}\cdot 1.000\cdots 0000=\pm 1\cdot 2^{1023-1023}\cdot\left(1+\frac{0}{2^{52}}\right)=\pm 1$$

$$(\pm)\cdot 2^{11\cdots 110}\cdot 1.111\cdots 11111=\pm 1\cdot 2^{2046-1023}\cdot\left(1+\frac{2^{52}-1}{2^{52}}\right)=\pm 1.7976931348623157\mathrm{e}^{308}$$

Rounding

- Rounding to nearest representable number
  - Maximum relative rounding error

```
double eps = std::numeric_limits<double>::epsilon();
```

$$\frac{1}{2^{(binary\ precision)}} = eps$$

$$\frac{1}{2^{(53)}} \approx 1.1e-16$$

$$\frac{1}{2^{(24)}} \approx 6.0e-8$$

$$\frac{1}{2^{(11)}} \approx 4.9e-4$$

- IEEE 754 requires result of operations is exactly rounded
  - For operators +, -,* ,/
  - That is: result is required to match result when performing operation with infinite precision and round the result
  - Reproducibility of results on different machines

Finite Subtraction

- Floating point hardware operates on a fixed number of digits
  - For efficient subtraction, the operand with the lower exponent is right shifted until operands are aligned (equal exponent)

Infinite precision for operation

$$a = 1.000 \cdot 2^{0110}$$

$$b = 1.110 \cdot 2^{0001}$$

$$a_2 = 1.0000000 \cdot 2^{0110}$$

$$b_2 = 0.0000111 \cdot 2^{0110}$$

$$a_2 - b_2 = 0.1111001 \cdot 2^{0110}$$

$$a_2 - b_2 = 1.1110010 \cdot 2^{0101}$$

$$a - b = 1.111 \cdot 2^{0101}$$

"Binary precision" for operation

$$a = 1.000 \cdot 2^{0110}$$

$$b = 1.110 \cdot 2^{0001}$$

$$a_2 = 1.000 \cdot 2^{0110}$$

$$b_2 = 0.000 \cdot 2^{0110}$$

$$a_2 - b_2 = 1.000 \cdot 2^{0110}$$

$$a - b = 1.000 \cdot 2^{0110}$$

Guard, Round, Stick

- How is hardware implementing IEEE requirements for operations

Infinite precision for operation

$$a = 1.000 \cdot 2^{0110}$$
$$b = 1.110 \cdot 2^{0010}$$
$$a_2 = 1.0000000 \cdot 2^{0110}$$
$$b_2 = 0.0001110 \cdot 2^{0110}$$
$$a_2 - b_2 = 0.1110010 \cdot 2^{0110}$$
$$a_2 - b_2 = 1.11001 \cdot 2^{0101}$$
$$a - b = 1.110 \cdot 2^{0101}$$

"Binary precision +3" for operation:
guard, round, sticky bits

$$a = 1.000 \cdot 2^{0110}$$
$$b = 1.110 \cdot 2^{0010}$$
$$a_2 = 1.000\overbrace{000}^{G\,R\,S} \cdot 2^{0110}$$
$$b_2 = 0.000111 \cdot 2^{0110}$$
$$a_2 - b_2 = 0.111001 \cdot 2^{0110}$$
$$a_2 - b_2 = 1.11001\_ \cdot 2^{0101}$$
$$a - b = 1.110 \cdot 2^{0101}$$

"Binary precision" for operation

$$a = 1.000 \cdot 2^{0110}$$
$$b = 1.110 \cdot 2^{0010}$$
$$a_2 = 1.000 \cdot 2^{0110}$$
$$b_2 = 0.000 \cdot 2^{0110}$$
$$a_2 - b_2 = 1.000 \cdot 2^{0110}$$
$$a - b = 1.000 \cdot 2^{0110}$$

Cancellation

- Subtraction of two nearby quantities
  - Assume relative error of ‘eps’ for a and b
  - Relative error in result of subtraction is high
    - Bits of lower significance “bubble up”
  - Workaround
    - Guard, round, sticky bits don’t help
    - Reformulate expression/procedure, if possible

“Binary precision +3” for operation:
guard, round, sticky bits

$$a = 1.000 \cdot 2^{0110}$$

$$b = 1.001 \cdot 2^{0110}$$

$$a_2 \quad = 1.000000 \cdot 2^{0110}$$

$$b_2 \quad = 1.001000 \cdot 2^{0110}$$

$$a_2 - b_2 = (-)0.001000 \cdot 2^{0110}$$

$$a - b = (-)1.000 \cdot 2^{0010}$$

“Binary precision” for operation

$$a = 1.000 \cdot 2^{0110}$$

$$b = 1.001 \cdot 2^{0110}$$

$$a - b = (-)0.001 \cdot 2^{0110}$$

$$a - b = (-)1.000 \cdot 2^{0010}$$

Finite Precision Examples

- Associativity

```
int main() { // associative math
  float a = -500000000;
  float b = 500000000;
  float c = 1;
  std::cout << "a + (b + c) is equal to " << a + (b + c) << std::endl;
  std::cout << "(a + b) + c is equal to " << (a + b) + c << std::endl;
}
```

- Output

a + (b + c) is equal to 0.0000000000000000
(a + b) + c is equal to 1.0000000000000000

Finite Precision Examples

- Rounding

```
int main() { // guard, round, and sticky bits
  float a = 1.0f;
  float b = 6e-8;
  float c = a + b;

  float eps = std::numeric_limits<float>::epsilon();
  float ref = 1.0f;
  float refp = ref + eps;
  float refm = ref - eps;

  std::cout << std::setprecision(16) << std::fixed;
  std::cout << "float    : " << a << " + " << b << " = " << c << std::endl;
  std::cout << "exact    : " << "1.00000006 " << std::endl;
  std::cout << "ref+eps  : " << refp << std::endl;
  std::cout << "ref      : " << ref << std::endl;
  std::cout << "ref-eps  : " << refm << std::endl;
}
```

- Output

```
float    : 1.0000000000000000 + 0.0000000599999979 = 1.0000001192092896
exact    : 1.00000006
ref+eps  : 1.0000001192092896
ref      : 1.0000000000000000
ref-eps  : 0.9999998807907104
```

Finite Precision Examples

- Compare

```
{ // compare
   using namespace std;
   double eps =
   numeric_limits<double>::epsilon();

   bool equal1 = a – b == 0;
   bool equal2 = fabs(a - b) < 10 * eps;
   bool equal3 = fabs(a - b) < 100 * eps;
   bool equal4 = fabs(a - b) < eps * (fabs(a) + fabs(b));
   bool equal5 = fabs(a - b) <=
                 ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a) ) * eps);
}
```

Finite Precision Examples

- Large condition number

$$\underbrace{\begin{bmatrix} 0 & -1 \\ 0+\beta & -1 \end{bmatrix}}_{A} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 1+\beta \end{bmatrix} \qquad \underbrace{\begin{bmatrix} -1/\beta & 1/\beta \\ -1 & 0 \end{bmatrix}}_{A^{-1}} \cdot \begin{bmatrix} 1 \\ 1+\beta \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$y = -1$$

$$x = \frac{-1}{\beta} + \frac{1+\beta}{\beta} = \frac{-1+\beta+1}{\beta} = 1$$

$$\beta = 1\mathrm{e}^{-7} \rightarrow \kappa(A) = 2\mathrm{e}^{7}$$

$$\beta = 1\mathrm{e}^{-15} \rightarrow \kappa(A) = 2\mathrm{e}^{15}$$

```
double beta = 1e-15;
std::cout << "x1  = " << (1.0 + beta - 1.0) / beta << std::endl;
std::cout << "x2  = " << (-1.0 + beta + 1.0) / beta << std::endl;
std::cout << "x3  = " << -1.0 / beta + (1.0 + beta) / beta << std::endl;
double eps = std::numeric_limits<double>::epsilon();
std::cout << "eps = " << eps << std::endl;
std::cout << "x4  = "
          << -1.0 / (beta + eps) + (1.0 + beta - eps) / (beta + eps)
          << std::endl;
```

- Output

```
x1 = 1.1102230246251565
x2 = 0.9992007221626408
x3 = 1.1250000000000000
eps = 0.0000000000000002
x4 = 0.7500000000000000
```

Matrix Properties

- A is square nxn
  - Singular/degenerate or nonsingular/invertible/full rank

$$B \text{ exists so that } AB = I$$

  - Non-symmetric or symmetric/(self-adjoint/hermitian)

$$A = A^T$$

  - Indefinite or positive / negative (semi-)definite

$$z^T A z > 0 \qquad z^T A z < 0 \qquad \geq \quad \leq \qquad and \quad A = A^T$$

$$\underbrace{\begin{bmatrix} - & - & - \\ - & - & - \\ - & - & - \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}}_{b}$$

Matrix Decomposition/Factorization

$$\underbrace{\begin{bmatrix} - & - & - \\ - & - & - \\ - & - & - \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}}_{b}$$

- A is square nxn
  - Nonsingular/invertible/full rank $AB = I$
    - LU decomposition
    - QR decomposition
  - Symmetric positive(negative) (semi-)definite $A = A^T \quad z^T A z > 0$
    - LLT decomposition (Cholesky decomposition, special version of LU)

- A is rectangular nxm
    - LU decomposition
    - QR decomposition

under-determined/under-constrained

$$\begin{bmatrix} - & - & - & - \\ - & - & - & - \\ - & - & - & - \end{bmatrix} \cdot \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}}_{b}$$

option: pick one of many solutions (e.g. minimum)

over-determined/over-constraint

$$\underbrace{\begin{bmatrix} - & - & - \\ - & - & - \\ - & - & - \\ - & - & - \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}}_{b}$$

option: Least-squares solution

Problem Transformations

- Row scaling (D= diag.)
  - Does not change solution

$$Ax = b \quad \rightarrow \quad (DA)x = Db$$

- Column scaling (D= diag.)
  - Changes solution

$$Ax = b \quad \rightarrow \quad AD(D^{-1}x) = ADz = b \qquad x = Dz$$

- Premultiply (M= nonsingular)
  - Does not change solution

$$Ax = b \quad \rightarrow \quad MAx = Mb$$

- Row permutation (P= permutation matrix)
  - Does not change solution, but reorders

$$Ax = b \quad \rightarrow \quad PAx = Pb$$

- Column permutation (P= permutation matrix)
  - Changes solution

$$Ax = b \quad \rightarrow \quad (AP)P^{-1}x = b$$

- Inverse of permutation matrices

$$P = \begin{bmatrix} 0 & 3 \\ 1 & 0 \end{bmatrix} \quad P^{-1} = \begin{bmatrix} 0 & 1 \\ 1/3 & 0 \end{bmatrix}$$

LU Decomposition

- Lower triangular matrix
- Upper triangular matrix
- LU Decomposition
  - Step1: decompose A=LU
  - Step2: forward substitution
  - Step3: backward substitution
- Example: w/o and w/ row permutations

$$L = \begin{bmatrix} - & 0 & 0 \\ - & - & 0 \\ - & - & - \end{bmatrix} \quad U = \begin{bmatrix} - & - & - \\ 0 & - & - \\ 0 & 0 & - \end{bmatrix}$$

$$Ax = b \quad A = LU$$

$$L(Ux) = b$$

$$Lz = b \rightarrow z = L^{-1}b \quad \text{(forward subs.)}$$

$$Ux = z \rightarrow x = U^{-1}z \quad \text{(backward subs.)}$$

$$A = \begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix} \quad \varepsilon \approx \text{eps}$$

$$E = \begin{bmatrix} 1 & 0 \\ -1/\varepsilon & 1 \end{bmatrix} = L^{-1} \quad \text{"subtract } \frac{1}{\varepsilon} \text{ row1 from row2"}$$

$$U = \begin{bmatrix} \varepsilon & 1 \\ 0 & 1-1/\varepsilon \end{bmatrix} \underset{eps}{\approx} \begin{bmatrix} \varepsilon & 1 \\ 0 & -1/\varepsilon \end{bmatrix}$$

$$LU = \begin{bmatrix} \varepsilon & 1 \\ 1 & 0 \end{bmatrix} \neq A$$

$$A = \begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix} \quad \varepsilon \approx \text{eps} \quad \text{PA} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \text{A} = \begin{bmatrix} 1 & 1 \\ \varepsilon & 1 \end{bmatrix}$$

$$E = \begin{bmatrix} 1 & 0 \\ -\varepsilon & 1 \end{bmatrix} = L^{-1} \quad \text{"subtract } -\varepsilon \text{ row1 from row2"}$$

$$U = \begin{bmatrix} 1 & 1 \\ 0 & 1-\varepsilon \end{bmatrix} \underset{eps}{\approx} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

$$PLU = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \varepsilon & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = PA$$

Implementations

- A small detour on (B)asic (L)inear (A)lgebra (S)ubprograms = BLAS interface
  - Many implementations: Reference (Fortran), ATLAS, Intel, OpenBLAS
  - Many projects use compatible interface
  - Low-level routines for common LA operations
  - Level1: vector-vector

$$\vec{y} = \alpha\vec{x} + \vec{y}$$

  - Level2: matrix-vector

$$\vec{y} = \alpha A\vec{x} + \beta\vec{y}$$

  - Level3: matrix-matrix

$$C = \alpha A^{(T)} B^{(T)} + \beta C$$

- C interfaces are available “cblas.h”
- API documentation: http://www.netlib.org/lapack/explore-html/d9/d0e/group__level1.html

BLAS Examples

- “daxpy” – X plus Y

```
size_t N = 10;
std::cout << "cblas_daxpy" << std::endl;
// generate vectors
std::vector<double> x(N, 1);
std::vector<double> y(N, 2);
// y := alpha*x + beta*b
cblas_daxpy(N, 8, x.data(), 1, y.data(), 1);
// print result
for (auto item : y)
  std::cout << item << " ";
std::cout << std::endl;
```

BLAS Examples

- “dnrm2” – Euclidean Norm

```
size_t N = 10;
std::cout << "cblas_dnrm2" << std::endl;
// ||x||_2
std::vector<double> x(N, 1);
auto norm2 = cblas_dnrm2(N, x.data(), 1);
// print result
std::cout << norm2 << std::endl;
```

BLAS Examples

- “dtrsv” – solving for triangular A

```
{
  std::cout << "cblas_dtrsv" << std::endl;
  std::vector<double> A(N * N, 0);
  for (size_t i = 0; i != N; ++i)
    for (size_t j = 0; j != N; ++j)
      if (j >= i)
        A[j + i * N] = N - i;

  std::vector<double> bx(N, 10);

  // backward/forward substitution of  Ax=b , A is L or U
  cblas_dtrsv(CblasRowMajor, CblasUpper, CblasNoTrans, CblasNonUnit, N,
              A.data(), N, bx.data(), 1);

  // print result
  for (auto item : bx)
    std::cout << item << " ";
  std::cout << std::endl;
}
```

BLAS Examples

- “dgemv” – Matrix-Vector mult.

```
size_t N = 10;
std::cout << "cblas_dgemv" << std::endl;
size_t M = N - 5;
std::vector<double> A(M * N, 1);
std::vector<double> x(M, 1);
std::vector<double> y(N, 1);
double alpha = 2;
double beta = 10;
// y := alpha*A**T*x + beta*y
cblas_dgemv(CblasRowMajor, CblasTrans, M, N, alpha, A.data(), N, x.data(), 1,
            beta, y.data(), 1);
// print result
for (auto item : y)
  std::cout << item << " ";
std::cout << std::endl;
```

BLAS Examples

- “dgemm” – Matrix-Matrix mult.

```
size_t N = 10;
std::cout << "cblas_dgemm" << std::endl;
// generate matrices
std::vector<double> A(N * N, 1);
std::vector<double> B(N * N, 2);
std::vector<double> C(N * N, 0);
// C := alpha*op(A)*op(B) + beta*C
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1, A.data(),
            N, B.data(), N, 0, C.data(), N);
```

LU Implementations

- Lapack API: (L)inear (A)lgebra (Pack)age
  - Fortran library, uses BLAS backend
  - C bindings are available (“lapacke.h”)
  - API documentation: http://www.netlib.org/lapack/explore-html/d8/d70/group__lapack.html

- Eigen library
  - Open source
  - C++ header only library
  - Expression Templates
  - Overlap with BLAS/LAPACK
  - Can be configured to use BLAS backend
  - API documentation: https://eigen.tuxfamily.org/dox/group__DenseLinearSolvers__chapter.html

LU Implementations

- Lapack: “dgetrf” -- A = P L U

```
size_t N = 10;
std::cout << "LAPACKE_dgetrf (LU)" << std::endl;
std::vector<double> A(N * N, 1);
std::vector<int> pivots(N, 0);
for (size_t i = 0; i != N; ++i)
  for (size_t j = 0; j != N; ++j)
    if (j >= i)
      A[j + i * N] = j + i * N;
// A = P * L * U
auto info =
    LAPACKE_dgetrf(LAPACK_ROW_MAJOR, N, N, A.data(), N, pivots.data());
std::cout << info << std::endl;
// print pivots
for (auto item : pivots)
  std::cout << item << " ";
std::cout << std::endl;
// LU is stored in A, unit diagonal of L is not stored (implicit)
```

LU Implementations

- Eigen A = P L U

```
int N = 10;
using namespace Eigen;
std::cout << "Eigen LU" << std::endl;
std::vector<double> A(N * N,1);
for (size_t i = 0; i != N; ++i)
  for (size_t j = 0; j != N; ++j)
    if (j >= i)
      A[j + i * N] = j + i * N;

typedef Matrix<double, Dynamic, Dynamic,RowMajor> EigenMatrix;
Map<EigenMatrix> EA(A.data(), N, N);
auto decomposition = PartialPivLU<EigenMatrix>(EA);
EigenMatrix U = decomposition.matrixLU().triangularView<UpLoType::Upper>();
EigenMatrix L = decomposition.matrixLU().triangularView<UpLoType::UnitLower>();
EigenMatrix P = decomposition.permutationP().inverse();
std::cout << P << std::endl << std::endl;
std::cout << L << std::endl << std::endl;
std::cout << U << std::endl << std::endl;
std::cout << P*L*U << std::endl << std::endl;
std::cout << EA << std::endl << std::endl;
```

QR Decomposition

- Orthonormal matrix Q
- Upper triangular matrix R
- QR Decomposition
  - Step1: decompose A=QR
  - Step2: invert Q
  - Step3: backward substitution
- Methods for QR
  - Gram-Schmidt
  - Householder transformations
  - …

$$Q = \begin{bmatrix} \begin{bmatrix} - \\ - \\ - \end{bmatrix} \begin{bmatrix} - \\ - \\ - \end{bmatrix} \begin{bmatrix} - \\ - \\ - \end{bmatrix} \end{bmatrix} \quad R = \begin{bmatrix} - & - & - \\ 0 & - & - \\ 0 & 0 & - \end{bmatrix}$$

$$Ax = b \quad A = QR$$

$$QRx = b$$

$$Rx = Q^{-1}b = Q^{T}b = z \rightarrow \quad \text{(invert Q)}$$

$$Rx = z \rightarrow x = R^{-1}z \quad \text{(backward subs.)}$$

QR Decomposition

- QR using Gram-Schmidt
  - Step1: generate set of normalized orthogonal basis vectors from the columns of A
  - Step2: Calculate R

$$Q = \begin{bmatrix} \begin{bmatrix} - \\ - \\ - \end{bmatrix} \begin{bmatrix} - \\ - \\ - \end{bmatrix} \begin{bmatrix} - \\ - \\ - \end{bmatrix} \end{bmatrix} \quad R = \begin{bmatrix} - & - & - \\ 0 & - & - \\ 0 & 0 & - \end{bmatrix}$$

$$A = [a_1 \mid a_2 \mid a_3]$$

$$Q = [q_1 \mid q_2 \mid q_3]$$

$$q_1 = \frac{a_1}{\|a_1\|} \quad q_2 = \frac{a_2 - (a_2 \cdot q_1)q_1}{\|\cdots\|} \quad q_3 = \frac{a_3 - (a_3 \cdot q_1)q_1 - (a_3 \cdot q_2)q_2}{\|\cdots\|}$$

$$QR = A$$

$$R = Q^{-1}A$$

- Comment
  - Simple implementation
  - Not used in this form (instable)
  - Alternatives to Gram-Schmidt exist

QR Implementations

- Eigen: A = Q R

```
size_t N = 10;
//   using namespace Eigen;
std::cout << "Eigen QR" << std::endl;
std::vector<double> A(N * N, 1);
for (size_t i = 0; i != N; ++i)
  for (size_t j = 0; j != N; ++j)
    if (j >= i)
      A[j + i * N] = j + i * N;

typedef Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
    EigenMatrix;
Eigen::Map<EigenMatrix> EA(A.data(), N, N);
auto decomposition = Eigen::HouseholderQR<EigenMatrix>(EA);
EigenMatrix Q = decomposition.householderQ();
EigenMatrix R = Q.transpose() * EA;
std::cout << Q << std::endl << std::endl;
std::cout << R << std::endl << std::endl;
std::cout << Q * R << std::endl << std::endl;
std::cout << EA << std::endl << std::endl;
```

Cholesky Implementations

- Eigen/Lapack

```
{
  std::cout << "Cholesky LLT (symmetric definite matrices)" << std::endl;
  // A = L * L**T

  // LAPACKE_dpotrf(...,A,...)

  // LLT<EigenMatrix> decomposition(A);
  // EigenMatrix L = decomposition.matrixL();
}
{
  std::cout << "Lapack Cholesky variant LDLT (symmetric matrices)" << std::endl;
  // A = L * D * L**T
  // LAPACKE_dsysv(...)

}
{
  std::cout << "Eigen Cholesky variant LDLT (semidefinite matrices)" << std::endl;
  // A = P**T L D L**T P
  // LDLT<EigenMatrix> decomposition(A);
}
```

Jacobi Method

- Simplest form of iterative method for solving Ax=b

$$x_{k+1} = Gx_k + c$$

- Chose G and c so that a *fixed point* of

$$g(x) = Gx + c$$

  solves Ax=b

- Method is called *stationary* if G and c are constant over all iterations
- Method converges if

$$\rho(G) < 1$$

  - The smaller the spectral radius the faster the convergence

Jacobi Method

- How to obtain a fixed point for the solution of Ax=b?
- Splitting of A

$$x_{k+1} = Gx_k + c$$

$$A = M - N \quad \text{M invertible}$$

- Results in

$$Ax = b$$

$$(M - N)x = b$$

$$Mx = Nx + b$$

$$x_{k+1} = M^{-1}Nx + M^{-1}b$$

- Define

$$G := M^{-1}N$$

$$c := M^{-1}b$$

$$x_{k+1} = Gx_k + c$$

- Converges for

$$\rho(M^{-1}N) < 1$$

Jacobi Method

- Simplest choice for M is diagonal of A (=D)

$$A = D + R$$

$$\mathrm{M} = \mathrm{D}$$

$$\mathrm{N} = -\mathrm{R} = -(L + U)$$

- For A with no zero diagonal entries, D is nonsingular
- The Jacobi method is defined as

$$x_{k+1} = M^{-1}Nx + M^{-1}b$$

$$x_{k+1} = D^{-1}(-R)x + D^{-1}b$$

$$x_{k+1} = D^{-1}(b - Rx)$$

Jacobi Method

- Component wise rewriting of the Jacobi method

$$x_{k+1} = D^{-1}(b - Rx)$$

$$A = D + R = \begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{bmatrix} + \begin{bmatrix} 0 & - & - \\ - & 0 & - \\ - & - & 0 \end{bmatrix}$$

$$D^{-1} = \begin{bmatrix} 1/a_{11} & 0 & 0 \\ 0 & 1/a_{22} & 0 \\ 0 & 0 & 1/a_{33} \end{bmatrix}$$

$$x_i^{k+1} = \left(b_i - \sum_{i \neq j} a_{ij} x_j^k \right) \cdot \frac{1}{a_{ii}} \quad i = 1, \ldots, n$$

- Note: two storage locations for x are required
  - Otherwise newly computed values would be used on the RHS

Summary

- IEEE FP
  - Arithmetics/rounding/cancellation/eps
- Matrix properties
  - Square/symmetric/definite
- Matrix decompositions (direct solvers)
  - LU LLT LDLT
  - QR
- LA libraries
  - BLAS/LAPACK/Eigen
- LA libraries code snippets
  - Vector-vector/matrix-vector/matrix-matrix/LU/QR/LLT
- Jacobi method
  - Convergence
  - Element-wise formulation

Quiz

Q1: Imagine you track the balance of a bank account using a single precision floating point number representing EUR. Starting with balance 0 EUR, each day 0.1EUR are transferred to the account, after how many days will the balance not increase anymore?

Q2: What are potential advantages/disadvantages when using BLAS/LAPACK or Eigen?

Q3: How could you calculate an upper bound for the spectral radius for a given iteration matrix of the Jacobi method?

Q4: Assume a large matrix has mostly zero entries, how to store it efficiently in terms off memory footprint?

Q5: What is the benefit of preconditioning a problem before solving?