NumCSE exercise sheet 2
LU decomposition, sparsity, least squares

soumil.gurjar@sam.math.ethz.ch
oliver.rietmann@sam.math.ethz.ch

October 29, 2018

Exercise 2.1. *LU decomposition and pivoting.*

We consider the family of *invertible* matrices

$$\mathbf{A}_\epsilon := \begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix}, \qquad 0 \leq \epsilon < 1$$

and the permutation matrix

$$\mathbf{P} := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

We will examine the equation

$$\mathbf{A}_\epsilon \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{1}$$

with solution

$$x_1 = \frac{-1}{1-\epsilon}, \qquad x_2 = \frac{1}{1-\epsilon}. \tag{2}$$

(a) Show that $\mathbf{A}_0$ has no LU decomposition, while $\mathbf{P} \cdot \mathbf{A}_0$ does.

Solution: We show that there exist no $\ell_1, \ell_2, \ell_3, u_1, u_2, u_3 \in \mathbb{R}$ such that

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} \ell_1 & 0 \\ \ell_2 & \ell_3 \end{pmatrix} \begin{pmatrix} u_1 & u_2 \\ 0 & u_3 \end{pmatrix}. \tag{3}$$

Indeed, the $(1,1)$ component of (3) implies $0 = \ell_1 \cdot u_1$. Thus we have $\ell_1 = 0$ or $u_1 = 0$. The former contradicts $1 = \ell_1 \cdot u_2$ while the latter contradicts $1 = \ell_2 \cdot u_1$, which are exactly the $(1,2)$ and $(2,1)$ component of (3). Finally, we have the LU decomposition

$$\mathbf{P} \cdot \mathbf{A}_0 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

(b) If $0 < \epsilon < 1$, then $\mathbf{A}_\epsilon$ admits the LU decomposition $\mathbf{A}_\epsilon = \mathbf{L}_\epsilon \cdot \mathbf{U}_\epsilon$, where

$$\mathbf{L}_\epsilon := \begin{pmatrix} 1 & 0 \\ \frac{1}{\epsilon} & 1 \end{pmatrix}, \qquad \mathbf{U}_\epsilon := \begin{pmatrix} \epsilon & 1 \\ 0 & 1 - \frac{1}{\epsilon} \end{pmatrix}.$$

However for sufficiently[1] small $\epsilon > 0$, cancellation will occur: Instead of $\mathbf{U}_\epsilon$, we will end up with

$$\tilde{\mathbf{U}}_\epsilon := \begin{pmatrix} \epsilon & 1 \\ 0 & -\frac{1}{\epsilon} \end{pmatrix}.$$

[1]This issue occurs for instance if ϵ is of type `float` and has value `1.0e-8f`.

Solve (1) using the erroneous LU decomposition $(\mathbf{L}_\epsilon, \tilde{\mathbf{U}}_\epsilon)$ and compare this solution with (2).

Solution: The forward substitution equation

$$\begin{pmatrix} 1 & 0 \\ \frac{1}{\epsilon} & 1 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

admits the solution $z_1 = 1$ and $z_2 = -\frac{1}{\epsilon}$. Back substitution into

$$\begin{pmatrix} \epsilon & 1 \\ 0 & -\frac{1}{\epsilon} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix}$$

yields $x_1 = 0$ and $x_2 = 1$. We observe an error in x_1 of more than 100%.

(c) Now we do the same, but with pivoting: For all $0 < \epsilon < 1$, we have $\mathbf{P} \cdot \mathbf{A}_\epsilon = \mathbf{L}_\epsilon^{\mathbf{P}} \cdot \mathbf{U}_\epsilon^{\mathbf{P}}$, where

$$\mathbf{L}_\epsilon^{\mathbf{P}} := \begin{pmatrix} 1 & 0 \\ \epsilon & 1 \end{pmatrix}, \qquad \mathbf{U}_\epsilon^{\mathbf{P}} := \begin{pmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{pmatrix}.$$

Again, for sufficiently small $\epsilon > 0$, we have to consider $\tilde{\mathbf{U}}_\epsilon^{\mathbf{P}}$, where

$$\tilde{\mathbf{U}}_\epsilon^{\mathbf{P}} := \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

Solve (1) using the erroneous LU decomposition $(\mathbf{L}_\epsilon^{\mathbf{P}}, \tilde{\mathbf{U}}_\epsilon^{\mathbf{P}})$ of $\mathbf{P} \cdot \mathbf{A}_\epsilon$. Compare this solution with (2) and the one from (b).

Solution: Instead of (1), we solve the equivalent equation

$$\mathbf{P} \cdot \mathbf{A}_\epsilon \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Then the forward substitution equation

$$\begin{pmatrix} 1 & 0 \\ \epsilon & 1 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

admits the solution $z_1 = 0$ and $z_2 = 1$. Back substitution into

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix}$$

yields $x_1 = -1$ and $x_2 = 1$. We observe only a very small error for small ϵ.

Exercise 2.2. *LU decomposition and sparsity.*

Fix $n \in \mathbb{N}$ and let $\mathbf{T} \in \mathbb{R}^{n \times n}$ be the tridiagonal matrix

$$\mathbf{T} = \begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 2 & -1 \\ 0 & \dots & 0 & -1 & 2 \end{pmatrix}.$$

(a) How many real numbers are needed to store $\mathbf{T}$ for given n in CCS format?

Solution: The matrix $\mathbf{T}$ as $3n-2$ non-zero entries. Therefore, both `value`-vector and `row_ind`-vector are of size $3n-2$. The `col_ptr`-vector is always of size $n+1$ if the matrix has n columns. In total, we have to store

$$2(3n-2) + n + 1 = 7n - 3$$

numbers to represent $\mathbf{T}$ in CCS format.

(b) Write a function

```
std::vector<Eigen::Triplet<double>> MakeTripletList(int n)
```

which for given dimension `n` returns a vector of triplets representing $\mathbf{T}$.

Solution:

```
10 std::vector<Eigen::Triplet<double>> MakeTripletList(int n) {
11     int nnz = 3 * n - 2;
12     std::vector<Eigen::Triplet<double>> tripletList(nnz);
13 
14     for (int i = 0; i < n - 1; ++i) {
15         tripletList[3 * i + 0] = Eigen::Triplet<double>(i, i + 1, -1.0);
16         tripletList[3 * i + 1] = Eigen::Triplet<double>(i, i, 2.0);
17         tripletList[3 * i + 2] = Eigen::Triplet<double>(i + 1, i, -1.0);
18     }
19     tripletList[3 * (n - 1)] = Eigen::Triplet<double>(n - 1, n - 1, 2.0);
20 
21     return tripletList;
22 }
```

sparse.cpp

(c) Write a function

```
double Runtime(const std::function<void(void)> &f)
```

that returns the runtime of a function `void f()` in seconds.

Solution:

```
24 double Runtime(const std::function<void(void)> &f) {
25     std::chrono::time_point<std::chrono::high_resolution_clock> start, end;
26     std::chrono::duration<double, std::ratio<1>> duration;
27     int repetitions = 10;
28     double sum = .0;
29 
```

```
    for (int i = 0; i < repetitions; ++i) {
        start = std::chrono::high_resolution_clock::now();
        f();
        end = std::chrono::high_resolution_clock::now();
        duration = end - start;
        sum += (double)duration.count();
    }

    return sum / repetitions;
}
```

sparse.cpp

(d) Compare sparse vs. dense LU decomposition of $\mathbf{T}$ in terms of execution time as follows: Measure the runtime of `Eigen::SparseLU::compute` and `Eigen::FullPivLU::compute` for all $n \in \{64, 128, 256, 512\}$. What are the asymptotic complexities (large n)?

Solution:

```
    // matrix sizes for benchmark
    std::vector<int> N = {64, 128, 256, 512};
    std::cout << "LU decomposition of T, where n = " << N << std::endl;

    // set up variables for runtime measurement
    std::vector<double> runtimeSparse;
    std::vector<double> runtimeDense;

    for (int n : N) {
        tripletList = MakeTripletList(n);

        // sparse LU decomposition
        Eigen::SparseMatrix<double> S(n, n);
        S.setFromTriplets(tripletList.begin(), tripletList.end());
        Eigen::SparseLU<Eigen::SparseMatrix<double>> sparseLU;
        std::function<void(void)> SparseLU = [&S, &sparseLU] () {
            sparseLU.compute(S);
        };

        // dense LU decomposition
        Eigen::MatrixXd D(S);
        Eigen::FullPivLU<Eigen::MatrixXd> denseLU(n, n);
        std::function<void(void)> DenseLU = [&D, &denseLU] () {
            denseLU.compute(D);
        };

        // benchmark
        runtimeSparse.push_back(Runtime(SparseLU));
        runtimeDense.push_back(Runtime(DenseLU));
    }

    std::cout << "Runtime in seconds using storage format..." << std::endl;
    std::cout << "...sparse: " << runtimeSparse << std::endl;
    std::cout << "...dense:  " << runtimeDense << std::endl;
```

sparse.cpp

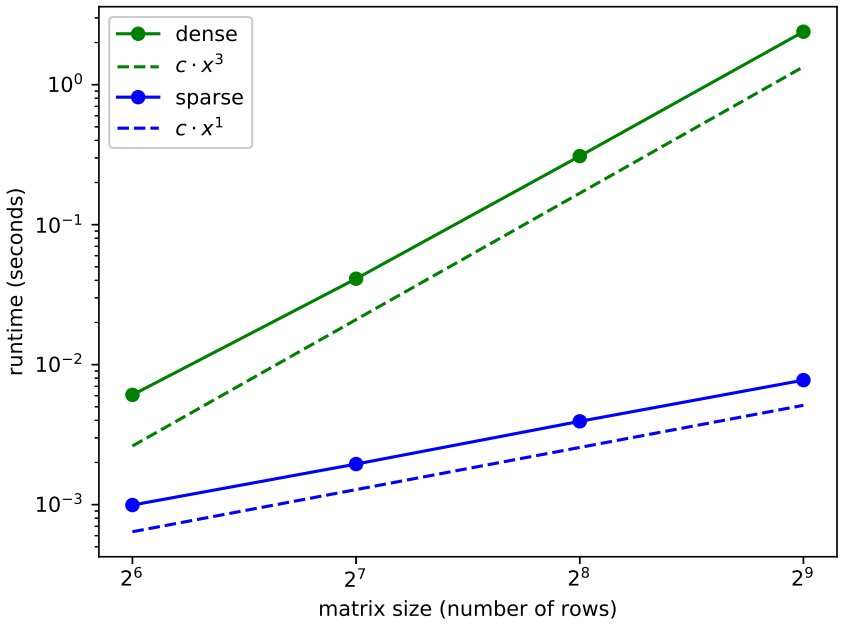

Figure 1: Runtime for sparse and dense LU decomposition of $\mathbf{T}$. We see that the former has complexity $O(n)$, while the latter has $O(n^3)$.

Exercise 2.3. *least squares.*

We consider the (scaled) *Runge*[2] function $r : [-1, 1] \to \mathbb{R}$ defined by

$$r(x) = \frac{1}{1 + 25x^2}$$

for all $x \in [-1, 1]$. Fix $m \in \mathbb{N}$ and suppose that r is given only by sampled data (x_i, y_i), where

$$x_i = \frac{2i}{m-1} - 1, \qquad y_i = r(x_i),$$

for all $i \in \{0, \dots, m-1\}$. For given $n \in \mathbb{N}$ we shall use this data to approximate r by a polynomial

$$p(x) = \sum_{i=0}^{n-1} a_i x^i,$$

where the coefficients $a_0, \dots, a_{n-1} \in \mathbb{R}$ are to be determined. To this end, we introduce the linear system

$$\underbrace{\begin{pmatrix} 1 & x_0^1 & \dots & x_0^{n-1} \\ 1 & x_1^1 & \dots & x_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m-1}^1 & \dots & x_{m-1}^{n-1} \end{pmatrix}}_{\mathbf{V}(x,n) :=} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{m-1} \end{pmatrix}, \tag{4}$$

which is equivalent to $p(x_i) = y_i$ for all $i \in \{0, \dots, m-1\}$. The matrix $\mathbf{V}(x, n)$ is called *Vandermonde*[3] matrix.

(a) Implement a function

```
Eigen::MatrixXd Vandermonde(const Eigen::VectorXd &x, int n)
```

which takes the vector `x` of sample points $x_0, \dots, x_{m-1}$ and the number `n` of coefficients of p and returns the associated Vandermonde matrix $\mathbf{V}(x, n)$.

Solution:

```
5  Eigen::MatrixXd Vandermonde(const Eigen::VectorXd &x, int n) {
6      int m = x.size();
7      Eigen::MatrixXd V(m, n);
8
9      V.col(0) = Eigen::VectorXd::Ones(x.size());
10     for (int i = 1; i < n; ++i) {
11         V.col(i) = V.col(i - 1).cwiseProduct(x);
12     }
13
14     return V;
15 }
```

least_squares.cpp

(b) We set $m = n$. Then $\mathbf{V}(x, n)$ is invertible and (4) admits a unique solution. Use `C++/Eigen` to find this solution and print the coefficients of p for $n = 11$.

Solution:

[2]Named after the German mathematician and physicist Carl David Tolmé Runge.
[3]Named after the French mathematician Alexandre-Théophile Vandermonde.

```
22      int n = 11;             // Number of polynomial coefficients
23      int m;                  // Number of samples
24      Eigen::MatrixXd V;      // Vandermonde matrix
25      Eigen::VectorXd x;      // Samples in [-1, 1]
26      Eigen::VectorXd y;      // r(x)
27      Eigen::VectorXd a(n);   // Polynomial coefficients
28
29      Eigen::IOFormat PythonFmt(Eigen::StreamPrecision, Eigen::DontAlignCols, ", ", ";\
        n", "[", "]", "[", "]");
30
31      std::cout << "Polynomial coefficients obtained by..." << std::endl;
32
33      // Compute overfitted polynomial coefficients
34      m = n;
35      x.setLinSpaced(m, -1.0, 1.0);
36      y = r(x);
37      V = Vandermonde(x, n);
38      a = V.lu().solve(y);
39      std::cout << "...overfitting:" << std::endl;
40      std::cout << a.transpose().format(PythonFmt) << std::endl;
```

least_squares.cpp

(c) Now we consider $m = 3n$ data points (yielding a tall $\mathbf{V}(x, n)$). In general, Equation (4) will no longer be solvable and we resort to a least squares solution. Write a short `C++/Eigen` code to obtain the least squares solution. Print the coefficients of p again for $n = 11$.

Solution:

```
42      // Compute least squares polynomial coefficients
43      m = 3 * n;
44      x.setLinSpaced(m, -1.0, 1.0);
45      y = r(x);
46      V = Vandermonde(x, n);
47      a = (V.transpose() * V).llt().solve(V.transpose() * y);
48      std::cout << "...least squares:" << std::endl;
49      std::cout << a.transpose().format(PythonFmt) << std::endl;
```

least_squares.cpp

(d) Plot the exact Runge function r and the polynomial p obtained in Tasks (b) and (c). What do you observe?

Solution: In Figure 2, we observe strong oscillations of the polynomial obtained in Task (b) at the edges of our interval. This is called Runge's phenomenon. On the other hand, the least squares solution from (c) exhibits no such behavior.

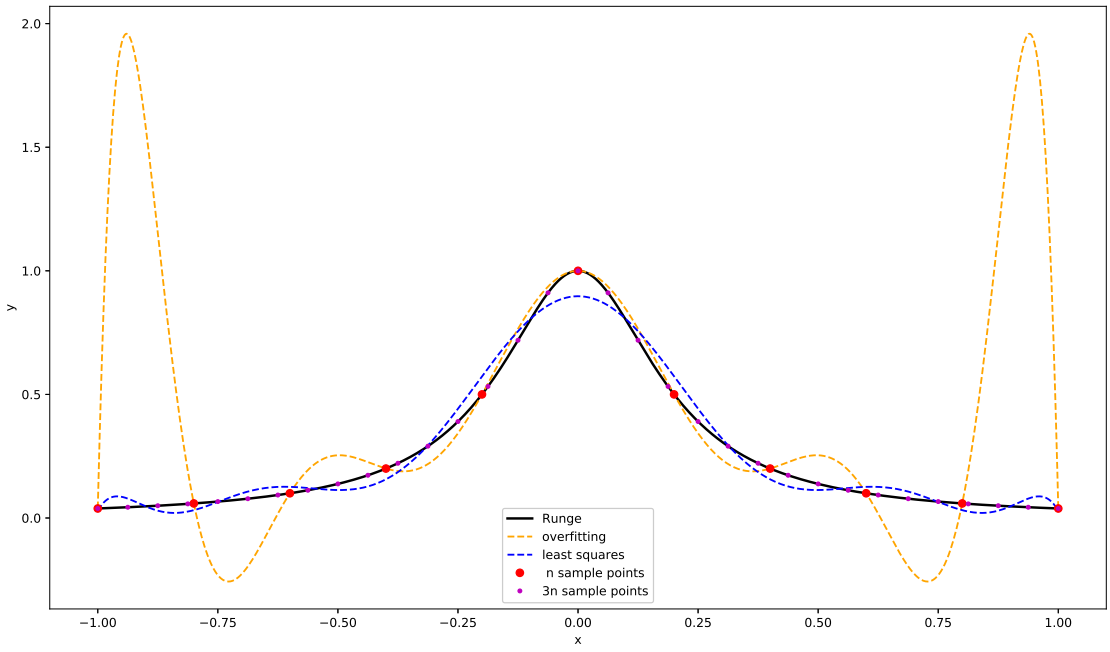

Figure 2: Runge's phenomenon.