NumCSE

Autumn Semester 2017

Prof. Rima Alaifari

Exercise sheet 2
Linear systems, sparse matrices

A. Dabrowski

Problem 2.1: Solving sequential linear systems

Given a matrix $\mathbf{A}$ and vectors $\mathbf{b}_1, \ldots, \mathbf{b}_m$ we want to find (efficiently) vectors $\mathbf{x}_1, \ldots, \mathbf{x}_m$ such that $\mathbf{A}\mathbf{x}_i = \mathbf{b}_i$ for every $i = 1, \ldots, m$.

Template: `solveSeqSystems.cpp`

(a) Implement a `C++` function which accepts as input an $n \times n$ matrix $\mathbf{A}$, an $n \times m$ matrix $\mathbf{B}$, an $n \times m$ matrix $\mathbf{X}$, and overwrites $\mathbf{X}$ so that

$$\mathbf{A}(\mathbf{X}^i) = \mathbf{B}^i \tag{2.1}$$

holds for every i. The superscript notation $\mathbf{X}^i$ indicates the i-th column of $\mathbf{X}$. Use a decomposition and solver of your choice (for example `FullPivLU()` from `Eigen`) at **every** step i.

(b) In another `C++` function, design an efficient implementation of Point (a) using the LU-decomposition of $\mathbf{A}$.

(c) What is the complexity of each of your implementations?

Problem 2.2: Blockwise linear solver

We want to solve a linear system where the matrix has a structure particularly suitable for block operations.

Template: `blockLinSolvers.cpp`

Let

$$\mathbf{A} = \begin{bmatrix} \mathbf{R} & \mathbf{v} \\ \mathbf{u}^T & 0 \end{bmatrix} \tag{2.2}$$

where $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ and $\mathbf{R} \in \mathbb{R}^{n,n}$ is upper triangular and invertible.

We first use an approach which relies on LU-decomposition.

(a) Compute the blockwise LU-decomposition of $\mathbf{A}$.

(b) Show that $\mathbf{A}$ is invertible if and only if $\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v} \neq 0$.

(c) Implement a `C++` function which accepts as input $\mathbf{R}, \mathbf{u}, \mathbf{v}, \mathbf{b}, \mathbf{x}$ and writes in $\mathbf{x}$ the solution of $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is the matrix given in (2.2). Make use of the blockwise LU-decomposition you derived in Point (a) and only use elementary operations (no solvers from `Eigen`).

(d) What is the asymptotic complexity of your implementation?

We move on to an approach which relies on blockwise Gaussian elimination.

(e) Determine expressions for $\mathbf{z} \in \mathbb{R}^n, \xi \in \mathbb{R}$ such that

$$\begin{bmatrix} \mathbf{R} & \mathbf{v} \\ \mathbf{u}^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{z} \\ \xi \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \beta \end{bmatrix}$$

for arbitrary $\mathbf{b} \in \mathbb{R}^n, \beta \in \mathbb{R}$.

Hint: Use blockwise Gaussian elimination.

(f) Implement a `C++` function as in Point (c) which computes the solution to $\mathbf{Ax} = \mathbf{b}$. This time however, use the blockwise decomposition from Point (e).

Hint: You can rely on the `triangularView()` function to instruct EIGEN of the triangular structure of $\mathbf{R}$.

(g) What is the asymptotic complexity of your implementation?

Problem 2.3: Resistance to impedance map

We apply the Sherman-Morrison-Woodbury update formula to analyze an electric circuit.

Template: `circuitImpedance.cpp`

We want to compute the impedance of the circuit drawn in Fig. 1 as a function of a variable resistance of a **single** circuit element.

The circuit contains 27 identical linear resistors with resistance $R = 1$, and a variable resistance R_x between nodes 14 and 15. Excitation is provided by a voltage V imposed at node 16. We consider only direct current operation (stationary setting), that means all currents and voltages are real-valued.

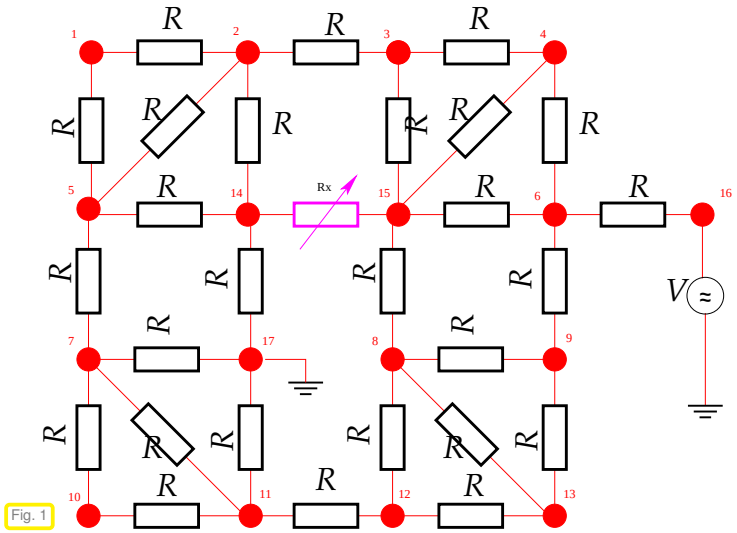

Fig. 1

(a) (*optional*) Compute voltages and currents in the circuit by means of nodal analysis. Understand how this leads to a linear system of equations for the unknown nodal potentials (the fundamental laws of circuit analysis should be known from physics as well as the principles of nodal analysis). The circuit matrix $\mathbf{A}_{R_x}$ and the right-hand-side b of the resulting linear system is already coded in the template; if you do not want to derive it yourself you can directly skip to the next point.

Hint: use Kirchhoff's current law and Ohm's law to determine the coefficients of the matrix $\mathbf{A}_{R_x}$ such that $\mathbf{A}_{R_x}\mathbf{v} = \mathbf{b}$, where $\mathbf{v}_i$ is the voltage at node i, and $\mathbf{b}$ is the zero vector (except for the position corresponding to the source term at node 16).

(b) Characterize the change in the circuit matrix $\mathbf{A}_{R_x}$ induced by a change in the value of R_x as a low-rank modification of the circuit matrix $\mathbf{A}_1$ (that is the matrix $\mathbf{A}_{R_x}$ with $R_x = 1$). Use the matrix $\mathbf{A}_1$ as your "base state".

Hint: four entries of the circuit matrix will change. This amounts to a rank-1-modification for suitable vectors.

(c) Using EIGEN, implement a C++ function which returns the impedance of the circuit from Figure 1, when supplied with $\mathbf{A}_1^{-1}$ and a specific value for R_x. Recall that the impedance of the circuit is the quotient of the voltage at node 16 with the current through node 16. This function should be implemented efficiently using the Sherman-Morrison-Woodbury formula.

(d) Test your function using $V = 1$ and $R_x = 1, 2, 2^2, \ldots, 2^{10}$ (as a reference value, for $R_x = 1024$ you should obtain an impedance of 2.65744).

Problem 2.4: Triplet format to CRS format

We want to devise a function that converts a matrix given in *triplet/coordinate list* (COO) format to the *compressed row storage* (CRS) format.

Template: `COOtoCRS.cpp`

Let $\mathbf{A}$ indicate an arbitrary matrix.

The COO format of $\mathbf{A}$ stores a collection of triplets (i, j, v), with $i, j \in \mathbb{N}$ the indices, and $v \in \mathbb{R}$ the non-zero value which contributes to the element in position (i, j) of $\mathbf{A}$. Multiple triplets corresponding to the same position are allowed, meaning that multiple values with the same indices (i, j) should be summed together to obtain the actual content in position (i, j) of $\mathbf{A}$.

The CRS format uses three vectors:

1. `val`, which stores the values of the nonzero entries of $\mathbf{A}$, read from left to right and then from top to bottom;
2. `col_ind`, which stores the column indices of the elements in `val`;
3. `row_ptr`, which stores in position j the index of the entry in `val` which is the first element of the row j of $\mathbf{A}$.

The case of rows only made by zero elements require special consideration. The usual convention, which we adopt, is that if the j-th row of the matrix is empty, then `row_ptr` has in position j the same entry which is in position $j + 1$.

(a) Implement two `C++` functions which respectively convert the COO and the CRS format to EIGEN dense matrices. Implement two other `C++` functions which convert EIGEN dense matrices to COO and CRS.

Hint 1: For COO, you can use **Eigen::**`Triplet<`**double**`>` to store each triplet and **std::vector** to store the collection of triplets. For CRS, you can use three **std::vector**. You can either implement wrapper classes for COO and CRS objects or work with the raw data structures.

Hint 2: If it is convenient, you can append to `row_ptr` the length of `val` minus 1.

(b) Write a C++ function that converts a matrix in COO format to a matrix in CRS format. Try to be as efficient as possible.

Hint: use **std::**`sort`.

(c) What is the worst-case complexity of your function which converts COO to CRS?

(d) Test the correctness of your functions on the matrix provided in the template.

Problem 2.5: Multiplication in COO format

We want to design an algorithm which computes efficiently the multiplication of two sparse matrices in COO format.

Template: `COOMult.cpp`

(a) Is the product of two sparse matrices always sparse? If not, is it always dense?

Hint: think about simple matrices, only made of columns, rows or diagonals of ones.

(b) Implement a `C++` function which computes the product between two matrices in COO format and returns the result in COO format. Do not care about efficiency at this point.

(c) What is the asymptotic complexity of your naive implementation of Point (b)? Assume there are no duplicates in the input triplet vectors.

(d) Implement a C++ function which computes the product between two matrices in COO format in an efficient way.

Hint: sort the two lists of triplets in a convenient way.

(e) What is the asymptotic complexity of your efficient implementation? Assume there are no duplicates in the input triplet vectors.

(f) Compare the timing of your functions for random matrices with different dimensions. Perform the comparison first for products between sparse matrices, then for any kind of matrix.