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.

SOLUTION:

```
void solve_naive(const MatrixXd &A, const MatrixXd &b, MatrixXd &X) {

    for(int i = 0; i < b.cols(); i++) {
        X.col(i) = A.fullPivLu().solve(b.col(i));
    }
}
```

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

SOLUTION:

```
void solve_LU(const MatrixXd &A, const MatrixXd &b, MatrixXd &X) {

    FullPivLU<MatrixXd> LU = A.fullPivLu();

    for(int i = 0; i < b.cols(); i++) {
        X.col(i) = LU.solve(b.col(i));
    }
}
```

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

SOLUTION:

The complexity of the LU decomposition of an $n \times n$ matrix is $O(n^3)$. The complexity of backsubstitution for a matrix of size $n \times n$ is $O(n^2)$. Therefore performing both LU decomposition and backsubstitution m times has a cost of $O(mn^3)$ operations, while performing once the LU decomposition and m times only the backsubstitution has a cost of $O(mn^2 + n^3)$. If $m \simeq n$, the first strategy has complexity $O(n^4)$ while the second $O(n^3)$.

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 $\mathbf{A}$.

SOLUTION:

The LU-decomposition is:

$$\mathbf{A} = \mathbf{LU} = \begin{bmatrix} \mathbf{I} & 0 \\ \mathbf{u}^T\mathbf{R}^{-1} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{v} \\ 0 & -\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v} \end{bmatrix}$$

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

SOLUTION:

$\mathbf{A}$ is regular if and only if $\det(\mathbf{A}) \neq 0$.

$$\det(\mathbf{A}) = \det(\mathbf{L})\det(\mathbf{U}) = \det(\mathbf{L})\det(\mathbf{R})(\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v}) \tag{2.3}$$

$\det(\mathbf{L}) = 1$ and $\mathbf{R}$ is regular, so $\det(\mathbf{R}) \neq 0$, which leaves $\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v} \neq 0$ as necessary and sufficient condition.

(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`).

SOLUTION:

```
1  VectorXd solve_backSub(const MatrixXd& R, const VectorXd& c)
2  {
3      int n = R.rows();
4      VectorXd y(n);
5
6      // Since R is upper triangular, we can solve by backwards substitution
7      for (int i = n-1; i >= 0; --i)
8      {
9          y(i) = c(i);
10         for (int j = n-1; j > i ; --j)
```

```
        {
            y(i) -= R(i,j) * y(j);
        }
        y(i) /= R(i,i);
    }
    return y;
}

void solve_blockLU(const MatrixXd &R,
                   const VectorXd &u,
                   const VectorXd &v,
                   const VectorXd &b,
                   VectorXd &x) {

    int n = R.rows();
    VectorXd y(n+1);
    //
    // Solve Ly = b by forward substitution.
    y.head(n) = b.head(n);
    y(n) = b(n) - u.transpose() * solve_backSub(R, y.head(n));

    // Solve Ux = y by backward substitution.
    MatrixXd U(n+1,n+1);
    U << R,                                   v,
         VectorXd::Zero(n).transpose(), -u.transpose()*solve_backSub(R,v);

    x = solve_backSub(U,y);
}
```

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

SOLUTION:

The backward substitution for $\mathbf{R}^{-1}\mathbf{x}$ is $O(n^2)$, vector dot product and subtraction of vectors is $O(n)$, so that the complexity is dominated by the backward substitution: $O(n^2)$.

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.

SOLUTION:

Applying the block Gauss elimination, we obtain:

$$\begin{bmatrix} \mathbf{1} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{z} \\ \xi \end{bmatrix} = \begin{bmatrix} \mathbf{R}^{-1}(\mathbf{b} - \mathbf{v}s^{-1}b_s) \\ s^{-1}b_s \end{bmatrix}$$

with $s := -(\mathbf{u}^\top \mathbf{R}^{-1}\mathbf{v})$, $b_s := (\beta - \mathbf{u}^\top \mathbf{R}^{-1}\mathbf{b})$.

(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 $\mathbf{R}$.

SOLUTION:

```
void solve_blockGauss(const MatrixXd &R,
                      const VectorXd &u,
                      const VectorXd &v,
                      const VectorXd &b,
                      VectorXd &x) {

    int n = R.rows();

    const TriangularView<const MatrixXd, Upper> & triR =
       R.triangularView<Upper>();

    double sinv = -1. / u.dot(triR.solve(v));
    double bs = (b(n) - u.dot(triR.solve(b.head(n))));
    double sinvbs = sinv*bs;

    x << triR.solve(b.head(n) - v*sinvbs),
         sinvbs;
}
```

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

SOLUTION:

As before, the complexity is dominated by the backward substitution, this time hidden in the solver for the triangular view matrix, but still of complexity $O(n^2)$.

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.

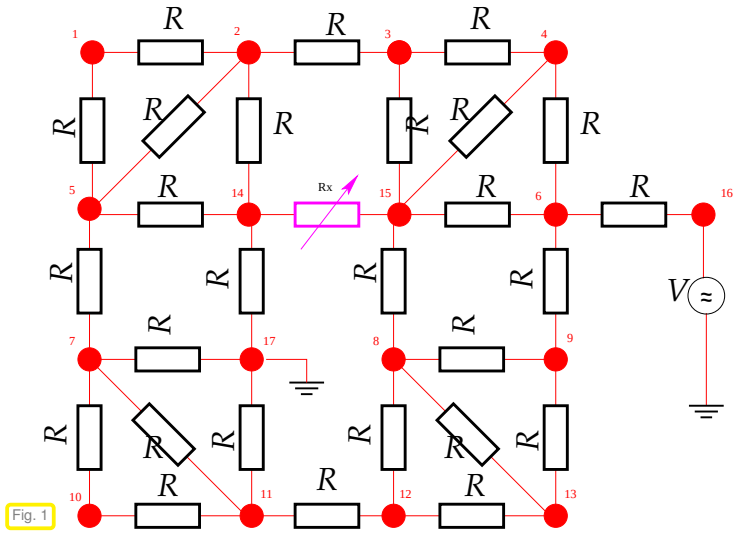

Fig. 1

(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).

SOLUTION:

We use Kirchhoff's first law, stating that the sum of the currents incident to a node is zero. We use Ohm's law to rewrite the current in terms of voltage and resistance: the current flowing in connection a to b is $I_{a,b} := \Delta_{a,b}v/R$. Here, we denote by $\Delta v_{i,j} := (v_i - v_j)$. Let $\mathbf{v} \in \mathbb{R}^{17}$ be the vector of voltages at the nodes. Set $f_{R_x} := R/R_x$. The system (obtained applying Kirchhoff's law for each node $i = 1, \ldots, 15$)

becomes:

$$\begin{cases} \Delta v_{1,2} + \Delta v_{1,5} & = 0 \\ \Delta v_{2,1} + \Delta v_{2,3} + \Delta v_{2,5} + \Delta v_{2,14} & = 0 \\ \Delta v_{3,2} + \Delta v_{3,4} + \Delta v_{3,15} & = 0 \\ \Delta v_{4,3} + \Delta v_{4,6} + \Delta v_{4,15} & = 0 \\ \Delta v_{5,1} + \Delta v_{5,2} + \Delta v_{5,7} + \Delta v_{5,14} & = 0 \\ \Delta v_{6,4} + \Delta v_{6,9} + \Delta v_{6,15} + \Delta v_{6,16} & = 0 \\ \Delta v_{7,5} + \Delta v_{7,10} + \Delta v_{7,11} + \Delta v_{7,17} & = 0 \\ \Delta v_{8,9} + \Delta v_{8,12} + \Delta v_{8,13} + \Delta v_{8,5} & = 0 \\ \Delta v_{9,6} + \Delta v_{9,8} + \Delta v_{9,13} & = 0 \\ \Delta v_{10,7} + \Delta v_{10,11} & = 0 \\ \Delta v_{11,7} + \Delta v_{11,10} + \Delta v_{11,12} + \Delta v_{11,17} & = 0 \\ \Delta v_{12,8} + \Delta v_{12,11} + \Delta v_{12,13} & = 0 \\ \Delta v_{13,8} + \Delta v_{13,9} + \Delta v_{13,12} & = 0 \\ \Delta v_{14,2} + \Delta v_{14,5} + \Delta v_{14,17} + f_{R_x} \Delta v_{14,15} & = 0 \\ \Delta v_{15,3} + \Delta v_{15,4} + \Delta v_{15,6} + \Delta v_{15,8} + f_{R_x} \Delta v_{15,14} & = 0. \end{cases} \tag{2.4}$$

We rescaled each equation multiplying by R. We have the extra conditions $v_{16} := V$ and $v_{17} = 0$, due to the ground/source voltages. The unknowns of the system are voltages v_j for $j = 1, \dots, 15$. Define $\mathbf{x} := [v_1, \dots, v_{15}]^\top$,

$$\mathbf{A}_{R_x} = \begin{bmatrix} 2 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & -1 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 3 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & -1 & 0 & 0 & 4 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 4 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 4 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 3 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 3 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & -1 & 3 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 + \frac{R}{R_x} & -\frac{R}{R_x} \\ 0 & 0 & -1 & -1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -\frac{R}{R_x} & 4 + \frac{R}{R_x} \end{bmatrix}$$

and $\mathbf{b} \in \mathbb{R}^{15}$ as the zero vector except in position 5 where it takes value V. Then the linear system which we want to solve is

$$\mathbf{A}_{R_x} \mathbf{x} = \mathbf{b}.$$

(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.

SOLUTION:

Define $f := R/R_x$ and the vectors

$$\mathbf{u} := \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ -1 \end{bmatrix}, \mathbf{v} := \begin{bmatrix} 0 \\ \vdots \\ 0 \\ f-1 \\ 1-f \end{bmatrix}.$$

Then

$$\mathbf{A}_{R_x} = \mathbf{A}_1 + \mathbf{u}\mathbf{v}^\top.$$

Then from the SMW formula

$$\mathbf{A}_{R_x}^{-1} = \mathbf{A}_1^{-1} - \frac{\mathbf{A}_1^{-1}\mathbf{u}\mathbf{v}^\top\mathbf{A}_1^{-1}}{1+\mathbf{v}^\top\mathbf{A}_1^{-1}\mathbf{u}}.$$

(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.

SOLUTION:

```
double calc_impedance(const MatrixXd &A1inv, const VectorXd &b, double Rx,
    double V) {

    // create u and v
    double f = 1/Rx;
    VectorXd u = VectorXd::Zero(b.size());
    VectorXd v = VectorXd::Zero(b.size());
    u(13) = 1;
    u(14) = -1;
    v(13) = f-1;
    v(14) = 1-f;

    // Use SMW formula
    double alpha = 1 + v.transpose() * A1inv * u;
    VectorXd y = A1inv * b;
    VectorXd x = y - A1inv * u * v.transpose() * y / alpha;

    return V / (V - x(5));
}
```

(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).

SOLUTION:

```
    for (int i = 0; i <= 10; i++) {
        std::cout << "For Rx = 2^" << i << " the impedance is "
                  << calc_impedance(A1inv, b, 1 << i, V) << std::endl;
    }
```

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.

SOLUTION:

From dense to COO.

```
MatrixCOO(MatrixXd &A) {
// constructor from Eigen dense matrix
    for (int i=0; i < A.rows(); i++) {
        for (int j=0; j < A.cols(); j++) {
            if (A(i,j) != 0) {
                Triplet<double> t(i, j, A(i,j));
                list.push_back(t);
            }
        }
    }
}
```

From COO to dense.

```
MatrixXd toDense() {
// returns an Eigen dense version
```

```
        MatrixXd dense(this->rows(), this->cols());
        for (auto t : list) {
            dense(t.row(), t.col()) = t.value();
        }
        return dense;
    }
```

From dense to CRS.

```
    MatrixCRS(MatrixXd &A) {
        // constructor from Eigen dense matrix
        int nnz = 0; // number of non-zero elements
        int prevNotEmptyRow = -1;

        // fill up val and colind by looping through triplets
        // some additional care must be taken for empty rows for rowptr
        for (int i=0; i < A.rows(); i++) {
            bool emptyRow = true;

            for (int j=0; j < A.cols(); j++) {
                if (A(i,j) != 0) {
                    nnz++;
                    val.push_back(A(i,j));
                    col_ind.push_back(j);
                    emptyRow = false;
                }
            }

            if (!emptyRow) {
                for (int c = prevNotEmptyRow; c < i; c++) {
                    row_ptr.push_back(nnz);
                }
                prevNotEmptyRow = i;
            }
        }
    }
```

From CRS to dense.

```
    MatrixXd toDense() {
        // conversion to Eigen dense matrix
        int nRows = this->rows();
        MatrixXd dense(nRows, this->cols());
        int row = 0;
        int i = 0;

        while (row < nRows) {
            do { // some extra care for empty rows
                row++;
            } while (row_ptr[row-1] == row_ptr[row]);

            while (i < row_ptr[row]) {
                // insert all elements of a certain row
                dense(row-1, col_ind[i]) = val[i];
                i++;
```

```
            }
        }

        while (i < val.size()) {
            // insert last row; this step would not be necessary if
            // we had directly appended to rowptr the size of val.
            dense(row-1, col_ind[i]) = val[i];
            i++;
        }

        return dense;
    }
```

(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`.

SOLUTION:

```
    void sort_byRow() {
        // std::sort can sort by applying a specific comparison function,
        // which is what is passed hereafter as third parameter using
        // the lambda expressions syntax.
        std::sort(list.begin(), list.end(),
            [] (auto t1, auto t2) {
                return t1.row() < t2.row();
            });
    }
```

```
    MatrixCRS(MatrixCOO &A) {
        // constructor from COO matrix
        A.sort_byRow(); // bring A to a more maneagable form

        int prevNotEmptyRow = -1;
        std::vector<Triplet<double>> &l = A.list;

        for (int i=0; i < l.size(); i++) {

            while (prevNotEmptyRow < l[i].row()) {
                // if there are some empty rows, fill up rowptr
                // following the conventions.
                prevNotEmptyRow++;
                this->row_ptr.push_back(i);
            }

            double currVal = l[i].value();

            while (l[i].col() == l[i+1].col() &&
                   l[i].row() == l[i+1].row()) {
                // add up the values corresponding to the same indices
                currVal += l[i].value();
```

```
23                      i++;
24                  }
25
26                  this->val.push_back(currVal);
27                  this->col_ind.push_back(l[i].col());
28              }
29          }
```

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

SOLUTION:

Let k be the number of triplets. The complexity is dominated by $O(k \log k)$ due to the sorting of the triplets, the rest of the operations are $O(k)$.

(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.

SOLUTION:

For example considering

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

we have that $\mathbf{AB}$ is dense, while $\mathbf{BA}$ is sparse:

$$\mathbf{AB} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}, \mathbf{BA} = \begin{bmatrix} 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

(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.

SOLUTION:

```
1   static MatrixCOO mult_naive(MatrixCOO &A, MatrixCOO &B) {
2       // naive implementation of COO matrices multiplication
3       MatrixCOO result;
4       for (auto t1 : A.list) {
5           for (auto t2: B.list) {
6               if (t1.col() == t2.row()) {
7                   Triplet<double> t(t1.row(), t2.col(), t1.value() *
                        t2.value());
8                   result.list.push_back(t);
9               }
10          }
11      }
12      return result;
13  }
```

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

SOLUTION:

The double loop iterates throughout all the triplets of A and B. As the COO format only stores nonzero cells, the complexity is $O(\mathrm{nnz}(A)\ \mathrm{nnz}(B))$, assuming that there are no duplicates in the input triplet vectors (more generally, if nT_A and nT_B are the numbers of triplets of A and B respectively, the complexity is $O(nT_A\ nT_B)$).

(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.

SOLUTION:

```
    void sort_byRow() {
        std::sort(list.begin(), list.end(),
            [] (auto t1, auto t2) {
                return t1.row() < t2.row();
            });
    }

    void sort_byCol() {
        std::sort(list.begin(), list.end(),
            [] (auto t1, auto t2) {
                return t1.col() < t2.col();
            });
    }
```

```
    static MatrixCOO mult_efficient(MatrixCOO &A1, MatrixCOO &A2) {
        // efficient implementation of COO matrices multiplication

        if (&A1 == &A2) {// to avoid complications when A1 and A2 are the
                         // same object, (deep) copy by value A2.
            MatrixCOO copyA2(A2.list);
            return mult_efficient(A1, copyA2);
        }

        MatrixCOO result;

        // convenience renamings
        vector<Triplet<double>> &l1 = A1.list;
        vector<Triplet<double>> &l2 = A2.list;
        vector<Triplet<double>> &lr = result.list;

        // useful way of sorting the triplets
        A1.sort_byCol();
        A2.sort_byRow();
```

```
20
21          // build vectors of indices b1 and b2, which contain the indices
22          // of elements which begin a new column for A1 and which begin a
23          // new row for A2.
24          vector<int> b1;
25          vector<int> b2;
26          b1.push_back(0);
27          b2.push_back(0);
28
29          for (int i=0; i < l1.size()-1; i++) {
30              if (l1[i].col() != l1[i+1].col()) {
31                  b1.push_back(i+1);
32              }
33          }
34          b1.push_back(l1.size());
35
36          for (int j=0; j < l2.size()-1; j++) {
37              if (l2[j].row() != l2[j+1].row()) {
38                  b2.push_back(j+1);
39              }
40          }
41          b2.push_back(l2.size());
42
43          // exploiting the special sorting of l1, l2 we are able to pass
44          // in a single loop all and only the elements of the matrices
45          // A1, A2 which will be paired when one computes A1*A2.
46          int i = 0;
47          int j = 0;
48
49          while (i < b1.size()-1 && j < b2.size()-1) {
50              if (l1[b1[i]].col() == l2[b2[j]].row()) {
51                  // in this case, all the associated couples should be
52                  // multiplied and the resulting triplets
53                  // have to be added to result.
54                  int c1, c2;
55                  for (c1 = b1[i]; c1 < b1[i+1]; c1++) {
56                      for (c2 = b2[j]; c2 < b2[j+1]; c2++) {
57
58                          Triplet<double> t(l1[c1].row(), l2[c2].col(),
59                              l1[c1].value() * l2[c2].value());
60
61                          lr.push_back(t);
62                      }
63                  }
64                  i++;
65                  j++;
66              } else {
67                  if (l1[b1[i]].col() < l2[b2[j]].row()) {
68                      // due to sorting, if the column of l1 is smaller than
69                      // the row of l2, eventual couples which can contribute
70                      // to the product can be found only by increasing i.
71                      i++;
72                  }
73                  if (l1[b1[i]].col() > l2[b2[j]].row()) {
```

```
                    // same as previous, only the roles are reversed.
                    j++;
                }
            }
        }
        return result;
    }
```

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

SOLUTION:

For simplicity let $n := \mathrm{nnz}(A) = \mathrm{nnz}(B)$. The sorting performed at the beginning has complexity $O(n \log n)$, while all the other operations, except the central loop of the algorithm, have linear complexity. The central loop multiplies all and only the couples which are necessary to compute the product matrix. If we denote as $\overline{a}$ the average number of times an entry of a matrix appears in an elementary operation necessary for the computation of the product matrix, then the central loop has complexity $O(\overline{a}n)$. Notice that $\overline{a}$ depends on the sparsity pattern of the matrices involved: for example when A, B are full then $\overline{a} = n$, while when A, B are diagonal then $\overline{a} = 1$. Therefore for matrices sparse enough to guarantee that $\overline{a} = O(\log n)$, the complexity is dominated by the initial sorting, and we expect the efficient implementation to perform sensibly better than the naive (at least for large enough inputs). For non-sparse/full matrices, we still obtain $O(n^2)$ complexity.

(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.

SOLUTION:

```
int main() {
    srand(time(0));

    int N = 100; // size of matrix
    double sparseCoeff = 1./std::log(N); // how much sparse the matrix
                                         // should be (btw 0 and 1)

    // generate full N*N matrix with random elements
    MatrixXd A = MatrixXd::Random(N,N);
    MatrixXd B = MatrixXd::Random(N,N);

    // convert to COO format
    MatrixCOO Acoo(A);
    MatrixCOO Bcoo(B);

    // define some random number generator
    std::random_device rd;
    std::mt19937 g(rd());

    // shuffle randomly the elements of the COO lists
```

```
21      std::shuffle(Acoo.list.begin(), Acoo.list.end(), g);
22      std::shuffle(Bcoo.list.begin(), Bcoo.list.end(), g);
23
24      // keep only part of COO lists, proportionally to sparseCoeff
25      int n = sparseCoeff*N*N;
26      Acoo.list.resize(n);
27      Bcoo.list.resize(n);
28      cout << "Multiplication of " << N << "x" << N << " matrices with "
29          << n << " non-zero elements. ";
30
31      // time naive multiplication
32      auto start_naive = std::chrono::system_clock::now();
33      MatrixCOO::mult_naive(Acoo,Bcoo);
34      auto end_naive = std::chrono::system_clock::now();
35
36      // time efficient multiplication
37      auto start_eff = std::chrono::system_clock::now();
38      MatrixCOO::mult_efficient(Acoo,Bcoo);
39      auto end_eff = std::chrono::system_clock::now();
40
41      cout << "Time ratio naive/efficient implementation: "
42          << (double) (end_naive-start_naive).count() /
                (end_eff-start_eff).count();
43  }
```

For high values of non-zero entries, the efficient implementation outperforms sensibly the naive algorithm, especially for sparse matrices. However for small values of non-zero entries the naive implementation is faster.
