

ETH Lectures 401-0663-00L Numerical Methods for Computer Science
401-2673-00L Numerical Methods for CSE

Homework Problems and Projects

Prof. R. Hiptmair, SAM, ETH Zurich

Autumn Term 2021, Version of October 2, 2022
(C) Seminar für Angewandte Mathematik, ETH Zürich

[Link](#) to the current version of this homework collection

0.1 General Information

0.1.1 Weekly Homework Assignments

All problems will be published in this single “.pdf” file. Every week, we publish a list of problem numbers on the lecture Moodle page. These are the current assignments for that week. Details are given on the lecture Moodle page.

0.1.2 Importance of Homework

Homework assignments are **not** mandatory. However, it is **very important** that you constantly exercise with the material you learn. “Solving” the homework assignments one week before the main exam by looking at the solutions will likely result in failure.

We provide hints and solutions from the beginning. It is your responsibility to look at the solutions only if you are stuck with a difficult problem and you tried to find a solution for a sufficient amount of time.

Most homework problems come with an estimate for the time required for its solution. These times are what we expect a bright and well-prepared student to require for that problem during an exam.

0.1.3 Corrections and Grading of Assignments

Homework assignments will not be graded.

However, you may submit your handwritten solutions by handing them to your tutor directly or uploading them via the Moodle upload interface, see the lecture Moodle page for instructions. The assistants will

Problem 2-13: Conversion of Triplet (COO) Matrix Format to CRS Format

This exercise is about sparse matrices. Make sure you are prepared on the subject by reading [Lecture → Section 2.7] in the lecture notes. In particular, read through the various *sparse storage formats* discussed in class (cf. [Lecture → Section 2.7.1]).

Focus of the problem is implementation in C++, using EIGEN in one part.

Remark. The ultimate goal is to devise a function that converts a matrix given in **triplet (or COOrdinate) format** (COO, see [Lecture → § 2.7.1.1]) to the **compressed row storage format** (CRS, see [Lecture → § 2.7.1.4]). You do not need to rely on EIGEN to solve this problem.

Recall that the COO format stores a collection of triplets (i, j, v) , with $i, j \in \mathbb{N}$, $i, j \geq 0$ (the indices) and $v \in \mathbb{R}$ (the value in cell (i, j)). Multiple triplets corresponding to the same cell (i, j) are allowed, meaning that multiple values with the same indices (i, j) should be summed up when fully expressing the matrix.

In C++ the data for a matrix stored in COO format and with entry type **SCALAR** can be stored in the following data structure

```
template <typename SCALAR>
struct TripletMatrix {
    std::size_t n_rows {0}; // Number of rows
    std::size_t n_cols {0}; // Number of columns
    std::vector<std::tuple<std::size_t, std::size_t, SCALAR>> triplets;
};
```

Also remember that the CRS format uses three vectors:

1. `val`, which stores the values of the nonzero cells, one row after the other.
2. `col_ind`, which stores the column indices of the nonzero cells, one row after the other.
3. `row_ptr`, which stores the index of the entry in `val/col_ind` containing the first element of each row.

The case of rows composed of only zero elements requires special consideration. The convention is that `row_ptr` stores two consecutive entries with the same values, meaning that in vectors `val` and `col_ind` the next row begins at the same index of the previous row (which implies that the previous row must be empty). This convention should be taken into account when e.g. iterating through `row_ptr`. However, to simplify things, in this problem you may always consider *matrices without all-zero rows*.

A suitable C++ data structure for storing a matrix with entries of type **SCALAR** in CRS format is the following

```
template <typename SCALAR>
struct CRSMatrix {
    std::size_t n_rows {0}; // Number of rows
    std::size_t n_cols {0}; // Number of columns
    std::vector<SCALAR> val; // Value array
    std::vector<std::size_t> col_ind; // Column indices
    std::vector<std::size_t> row_ptr; // Row pointers
};
```

(2-13.a)  (30 min.) Implement the C++ functions

```
template <typename SCALAR>
```

```

Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic>
    densify(const TripletMatrix<SCALAR> &M);
template <typename SCALAR>
Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic>
    densify(const CRSMatrix<SCALAR> &M);

```

that convert the argument matrices into EIGEN's dense matrix storage type.

SOLUTION of (2-13.a):

- For **TripletMatrix** the conversion is easy: Run through the vectors of triplets and add the values of a triplet to the matrix element with its indices.
- For **CRSMatrix** you fill the dense matrix row by row while iterating over the `val` and `col_ind` vectors.

C++ code 2.13.1: Implementation of `densify()` for **TripletMatrix**.

```

2  template <typename SCALAR>
3  Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic> densify(
4      const TripletMatrix<SCALAR>& M) {
5      Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic> dense =
6          Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic>::Zero(M.n_rows,
7                                                                      M.n_cols);
8
9      // TODO: (2-13.a) Densify the TripletMatrix.
10     // START
11     for (auto& triplet : M.triplets) {
12         dense(std::get<0>(triplet), std::get<1>(triplet)) += std::get<2>(triplet);
13     }
14     // END
15     return dense;
16 }

```

Get it on  [GitLab \(triplettoCRS.hpp\)](#).

C++ code 2.13.2: Implementation of `densify()` for **CRSMatrix**

```

2  template <typename SCALAR>
3  Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic> densify(
4      const CRSMatrix<SCALAR>& M) {
5      Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic> dense =
6          Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic>::Zero(M.n_rows,
7                                                                      M.n_cols);
8
9      // TODO: (2-13.a) Densify the CRSMatrix.
10     // START
11     std::vector<std::size_t> row_ptr_end = M.row_ptr;
12     row_ptr_end.push_back(M.col_ind.size());
13
14     for (std::size_t i = 0; i < M.row_ptr.size(); ++i) {
15         for (std::size_t j = row_ptr_end[i]; j < row_ptr_end[i + 1]; ++j) {
16             dense(i, M.col_ind[j]) = M.val[j];
17         }
18     }
19     // END
20     return dense;
21 }

```

Get it on  GitLab ([triplettoCRS.hpp](#)).



(2-13.b)  (60 min.) Write a C++ function that converts a matrix **T** in COO format to a matrix in CRS format and returns the latter:

```
template <typename SCALAR>
CRSMatrix<SCALAR> tripletToCRS(const TripletMatrix<SCALAR>& T);
```

Your implementation should have (almost, maybe up to a logarithm) optimal complexity, that is, its computational effort should scale linearly with the number of triplets in the COO format.

HINT 1 for (2-13.b): It might be a good idea to sort the vector of triplets beforehand in a suitable way. To that end you can invoke `std::sort()`.



SOLUTION of (2-13.b):

The following steps can be implemented to realize `tripletToCRS()`:

1. Copy the vector of triplets and **sort** the copy using `std::sort` with a predicate comparing two triplets according to

$$(i_1, j_1, a_1) < (i_2, j_2, a_2) \Leftrightarrow (i_1 < i_2) \vee (i_1 = i_2 \wedge j_1 < j_2).$$

2. Loop over the sorted vector and count the numbers of non-zero entries per row:

$$n_i := \#\{j : (i, j, *) \in \text{triplet vector}\}, \quad i = 1, \dots, n_{\text{rows}}.$$

Use these values to initialize the `row_ptr` and add them to obtain the length of `col_ind` and `val`.

3. Run through the sorted triplet vector again, summing the value entries of repeated index pairs and storing the sum in the `val` vector. The corresponding column indices have to be inserted into `col_ind`.

C++ code 2.13.3: Listing of `tripletToCRS()`

```
2  template <typename SCALAR>
3  CRSMatrix<SCALAR> tripletToCRS(const TripletMatrix<SCALAR>& T) {
4      CRSMatrix<SCALAR> crs;
5      crs.n_rows = T.n_rows;
6      crs.n_cols = T.n_cols;
7
8      // TODO: (2-13.b) Convert the TripletMatrix to a CRSMatrix. Make
9      // sure that you
10     // handle repeated index pairs.
11     // START
12     // sort the triplets according to their indices over a copy of the
13     // vector
14     auto triplets_copy = T.triplets;
15     auto sorting_predicate =
16         [](const std::tuple<std::size_t, std::size_t, SCALAR>& a,
17            const std::tuple<std::size_t, std::size_t, SCALAR>& b) {
```

```

16         return (std::get<0>(a) < std::get<0>(b)) ||
17                ((std::get<0>(a) == std::get<0>(b)) &&
18                 (std::get<1>(a) < std::get<1>(b)));
19     };
20     std::sort(triplets_copy.begin(), triplets_copy.end(), sorting_predicate);
21
22     // count the number of non-zeroes per row
23     crs.row_ptr = std::vector<std::size_t>(crs.n_rows + 1, 0);
24     std::size_t k = 0;
25     std::size_t last_j = -1;
26     std::size_t last_i = -1;
27     for (std::size_t i = 1; i <= crs.n_rows; ++i) {
28         while (k < triplets_copy.size() && std::get<0>(triplets_copy[k]) == i - 1) {
29             // do only when no duplicate index pair
30             if (std::get<1>(triplets_copy[k]) != last_j ||
31                 std::get<0>(triplets_copy[k]) != last_i) {
32                 last_j = std::get<1>(triplets_copy[k]);
33                 last_i = std::get<0>(triplets_copy[k]);
34                 ++crs.row_ptr[i];
35             }
36             ++k;
37         }
38         crs.row_ptr[i] += crs.row_ptr[i - 1];
39     }
40
41     // now, we know nnz; reserve in advance
42     crs.val.reserve(crs.row_ptr[crs.n_rows]);
43     crs.col_ind.reserve(crs.row_ptr[crs.n_rows]);
44
45     crs.val.push_back(std::get<2>(triplets_copy[0]));
46     crs.col_ind.push_back(std::get<1>(triplets_copy[0]));
47     std::size_t i = 0;
48     for (k = 1; k < triplets_copy.size(); ++k) {
49         // repeated index pairs
50         if (std::get<0>(triplets_copy[k]) == std::get<0>(triplets_copy[k - 1]) &&
51             std::get<1>(triplets_copy[k]) == std::get<1>(triplets_copy[k - 1])) {
52             crs.val[i] += std::get<2>(triplets_copy[k]);
53         } else {
54             crs.val.push_back(std::get<2>(triplets_copy[k]));
55             crs.col_ind.push_back(std::get<1>(triplets_copy[k]));
56             ++i;
57         }
58     }
59     // END
60     return crs;
61 }

```

Get it on  GitLab ([triplettoCRS.hpp](#)).



(2-13.c)  What is the worst-case complexity of your function `tripletToCRS` ((2-13.b)) in the number of triplets?

HINT 1 for (2-13.c): If you use the C++ standard library functions consult the documentation to find out about their asymptotic computational effort.

SOLUTION of (2-13.c):

According to the C++ standard the algorithm underlying `std::sort()` has to feature an asymptotic complexity of $O(n \log n)$ for $n \rightarrow \infty$, n the number of array elements to be sorted. The sorting is the most expensive operation in Code 2.13.3. The computational cost of all other parts scales linearly in the number of triplets.



(2-13.d) 🕒 (20 min.) [depends on Sub-problem (2-13.a)]

Test the correctness of your implementation of `tripletToCRS()` by checking whether the function `densify()` from Sub-problem (2-13.a) yield the same matrix when applied before or after the call to `tripletToCRS()`.

For testing use a **TripletMatrix<double>** object containing an $n \times n$ -matrix defined by $5n$ triplets with random indices and value 1. To generate random indices use `std::rand() % n`. Implement the test as a C++ function

```
bool testTripletToCRS(std::size_t n);
```

which accepts the matrix dimension as arguments and returns **true**, if the test is passed.

HINT 1 for (2-13.d): Note that the conversion of the matrix formats may involved adding floating point numbers, which may introduce round-off errors. Thus [Lecture → Rem. 1.5.3.15] applies. ┘

SOLUTION of (2-13.d):



End Problem 2-13, 110 min.