

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 2-15: Ellpack sparse matrix format

The number of processing cores of high-performance computers has seen an exponential growth. However, the increase in *memory bandwidth*, which is the rate at which data can be read from or written into memory by a processor, has been slower. Hence, memory bandwidth is a bottleneck for several algorithms.

Several papers present storage formats to improve the performance of sparse matrix-vector multiplications. One example is the **Ellpack format**, where the number of nonzero entries per row is bounded and shorter rows are padded with zeros.

Related to [Lecture → Section 2.7.1], which should be studied before tackling this problem.

The C++ class **EllpackMat** whose definition is listed below implements the Ellpack sparse matrix format for a generic matrix $\mathbf{A} \in \mathbb{R}^{m,n}$:

C++ code 2.15.1: Declaration of EllpackMat class

```

2  class EllpackMat {
3  public:
4      EllpackMat(const Triplets &triplets, index_t m, index_t n);
5      double operator()(index_t i, index_t j) const;
6      void mvmult(const Vector &x, Vector &y) const;
7      void mtvmult(const Vector &x, Vector &y) const;
8      index_t get_maxcols() const;
9
10 private:
11     std::vector<double> val; //< Vector containing values
12     // corresponding to entries in 'col'
13     std::vector<index_t> col; //< Vector containing column
14     // indices of the entries in 'val'.
15     // The position of a cell in 'val' and 'col'
16     // is determined by its row number and original position in 'triplets'
17     index_t maxcols; //< Number of non-empty columns
18     index_t m, n;    //< Number of rows, number of columns
19 };

```

Get it on  GitLab (ellpack.hpp).

In the code above the following typedefs are used

```

using Triplet = Eigen::Triplet<double>;
using Triplets = std::vector<Triplet>;

```

Further, the data member `maxcols` is defined as the maximum number of non-zero entries in a row of the matrix:

$$\text{maxcols} := \max\{\#[(i, j) \mid \mathbf{A}_{i,j} \neq 0, j = 1, \dots, n], i = 1, \dots, m\}.$$

The entry read-access `operator()` method, which takes the entries $(\mathbf{A})_{i,j}$ row and column indices i and j as arguments, is implemented as follows. It completely “defines” the Ellpack sparse matrix format.

C++11-code 2.15.2: Definition of entry-access operator operator()

```

2  double EllpackMat::operator()(index_t i, index_t j) const {
3      assert(0 <= i && i < m && 0 <= j && j < n && "Index out of bounds!");
4
5      for (index_t l = i * maxcols; l < (i + 1) * maxcols; ++l) {
6          if (col.at(l) == j) return val.at(l);
7      }
8      return 0;
9  }

```

```

7   }
8   return 0;
9   }

```

Get it on  GitLab (ellpack.hpp).

It is well defined for $i = 0, \dots, m-1$ and for $j = 0, \dots, n-1$ (C++ indexing!).

(2-15.a)  (60 min.) Implement an efficient constructor of **EllpackMat** that builds an object of type **EllpackMat** from data forming a matrix in triplet format. In other words, implement the constructor with the following signature:

```
EllpackMat(const Triplets & triplets, index_t m, index_t n);
```

`triplets` is a **std::vector** of EIGEN triplets (see [Lecture → Section 2.7.2]). The arguments m and n represent the number of rows and columns of the matrix **A**.

- The data in the triplet vector must be compatible with the matrix size provided.
- to the constructor. No assumption is made on the ordering of the triplets.
- Values belonging to multiple occurrences of the same index pair are to be summed up.



Taking care of repeated index pairs requires sophisticated bookkeeping.

HINT 1 for (2-15.a): An object of `Eigen::Triplets<double>` type is a structure representing a triplet (r, c, v) , which are row, column and value of a cell in **A**. It provides the methods `index_t row()`, `index_t col()` and `double value()`, which return r , c and v . ┘

HINT 2 for (2-15.a): Contemplate a multi-pass implementation: The code traverses the vector of triplets several times, extracting different pieces of information.

The ultimate goal should be a code whose cost scales linearly with the number of triplets assuming that the matrix has only a small fixed number of non-zero entries per row. ┘

HINT 3 for (2-15.a): To begin with, you will have to determine the quantity `maxcols` based on `Eigen::Triplets<double> triplets` and the number of rows m . ┘

SOLUTION of (2-15.a):

The implementation below first scans the triplet vector in order to find out about the maximum number of non-zero entries in a row of the matrix.

C++11-code 2.15.3: Solution of Sub-problem (2-15.a): constructor of **EllpackMat**

```

2  EllpackMat::EllpackMat(const Triplets &triplets , index_t m, index_t n)
3      : m(m), n(n) {
4      // TODO: (2-15.a) implement the constructor for the class EllpackMat
5      // START
6
7      // Find maxcols, number of non-empty columns
8      // It is assumed that all triplets uniquely define one entry
9      std::vector<unsigned int> counters(m);
10     // Fill counters with 0
11     std::fill(counters.begin(), counters.end(), 0);
12
13     // counters especially for handling repeated index

```

```

14  std::vector<std::vector<unsigned int>> counters_col(m);
15
16  // 'maxcols' starts with no entry for each row
17  maxcols = 0;
18  // Loop over each triplet and increment the corresponding counter,
    updating
19  // maxcols if necessary
20  for (const Triplet &tr : triplets) {
21      std::vector<unsigned int> &col_idx = counters_col[tr.row()];
22      // if it's not a repeated index, then check if maxcols needs to be
    updated
23      if (std::find(col_idx.begin(), col_idx.end(), tr.col()) == col_idx.end()) {
24          // bookkeep the col of new(not repeated) triplet
25          col_idx.push_back(tr.col());
26          if (++counters[tr.row()] > maxcols) {
27              maxcols = counters[tr.row()];
28          }
29      }
30  }
31  std::cout << "Maxcols: " << maxcols << std::endl;
32
33  // Reserve space
34  col.resize(m * maxcols, -1);
35  val.resize(m * maxcols, 0);
36
37  // Loop over each triplet and find a column where to put the value
38  for (const Triplet &tr : triplets) {
39      assert(0 <= tr.row() && tr.row() < m && 0 <= tr.col() && tr.col() < n &&
    "Index out of bounds!");
40
41      index_t l;
42      for (l = tr.row() * maxcols; l < (tr.row() + 1) * maxcols; ++l) {
43          // Store the current triplet in the first
44          // empty position between 'tr.row()*maxcols'
45          // and '(tr.row()+1)*maxcols - 1'
46          if (col[l] == -1) {
47              col[l] = tr.col();
48              val[l] = tr.value();
49              break;
50          }
51      }
52
53      // if it's a repeated index, just sum up
54      if (col[l] == tr.col()) {
55          val[l] += tr.value();
56          break;
57      }
58  }
59  assert(l < (tr.row() + 1) * maxcols &&
    "You did not reserve enough columns!");
60
61  }
62  // END
63  }

```

Get it on  GitLab (ellpack.hpp).



(2-15.b)  (30 min.) Implement an efficient method of class **EllpackMat** that, given an input n -vector $\mathbf{x}$, returns the m -vector $\mathbf{y}$ from the matrix-vector product $\mathbf{Ax}$:

```
void mvmult(const Eigen::VectorXd &x, Eigen::VectorXd &y) const;
```

A is the matrix represented by `*this`. The implementation must run with the optimal complexity of $O(\text{nnz}(A))$.

HINT 1 for (2-15.b): Think about the way the Ellpack format stores the matrix values and how matrix-vector multiplication works. E.g., to obtain y_i , you have to compute the scalar product between row i of A and vector x , which is the sum of the products between each (nonzero) element of $A_{i,:}$ and x . Hence, you can do the following:

1. Iterate along the values of $A_{i,:}$ in the way Ellpack stores them.
2. Multiply each of them by the corresponding element of x , using `std::vector<index_t> col`.

SOLUTION of (2-15.b):

C++11-code 2.15.4: Solution of Sub-problem (2-15.b)

```
2 void EllpackMat::mvmult(const Vector &x, Vector &y) const {
3     assert(x.size() == n && "Incompatible vector x size!");
4     assert(y.size() == m && "Incompatible vector y size!");
5     // TODO: (2-15.b) implement the multiplication A*x using the
6     // class EllpackMat, with optimal complexity.
7
8     // START
9     for (index_t i = 0; i < m; ++i) {
10         y(i) = 0;
11         for (index_t l = i * maxcols; l < (i + 1) * maxcols; ++l) {
12             if (col[l] == -1) break;
13             y(i) += x[col[l]] * val[l];
14         }
15     }
16     // END
17 }
```

Get it on  GitLab (ellpack.hpp).

(2-15.c)  (20 min.) Parallel to what you did in Sub-problem (2-15.b), implement now the method `mtvmult` of **EllpackMat** that computes the matrix-vector product $A^T x$ with the transposed matrix.

HINT 1 for (2-15.c): The solution is very similar to that of Sub-problem (2-15.b). Think about which vector of $(A)_{i,:}$, x and y should be reordered using `std::vector<index_t> col` to obtain the correct result.

SOLUTION of (2-15.c):

C++11-code 2.15.5: Solution of (2-15.c)

```
2 void EllpackMat::mtvmult(const Vector &x, Vector &y) const {
3     assert(x.size() == m && "Incompatible vector x size!");
4     assert(y.size() == n && "Incompatible vector y size!");
```

```

5  // TODO: (2-15.c) implement the multiplication  $A^T * x$  using the
6  // class EllpackMat, with optimal complexity.
7
8  // START
9  y = Vector::Zero(n);
10 for (index_t i = 0; i < m; ++i) {
11     for (index_t l = i * maxcols; l < (i + 1) * maxcols; ++l) {
12         if (col[l] == -1) break;
13         y(col[l]) += x(i) * val[l];
14     }
15 }
16 // END
17 }

```

Get it on  GitLab (ellpack.hpp).



(2-15.d)  (10 min.) [depends on Sub-problem (2-15.a)]

Discuss the asymptotic complexity of your implementation of the constructor of **EllpackMat** in terms of growing number of triplets, $\# \text{triplets} \rightarrow \infty$.

SOLUTION of (2-15.d):

Due to the use of `std::find`, which does a sequential search, the computational effort (here measured by the numbers of comparisons needed) of the constructor of **EllpackMat** implemented in Code 2.15.3 scales quadratically in `maxcols`. However, under the *assumption* that that number is fixed and small, the overall effort is linear in the number of triplets: $O(\# \text{triplets})$.



End Problem 2-15, 120 min.