ETH Lecture 401-0663-00L Numerical Methods for **Computer Science**

Mid-Term Examination

Autumn Term 2021

Friday, Nov 12, 2021, 14:15, HG F 1

Don't panic!

Family Name		**Grade**
First Name		
Department		
Legi Nr.		
Date	Friday, Nov 12, 2021	

Points:

Prb. No.	1	2	3	Total
max	7	10	12	
achvd				

(100% = 29 pts. , ≈40% (passed) = 11 pts.)

- Upon entering the exam room take a seat at a desk on which you find an envelope (**without** a sticky note "CSE" on it).
- This is a **closed-book exam**, no aids are allowed.
- Keep only writing paraphernalia and your ETH ID card on the table.
- Turn off mobile phones, tablets, smartwatches, etc. and stow them away in your bag.
- When told to do so, take the exam paper out of the envelope, and fill in the cover sheet first. Do not turn pages yet!
- Make sure that your exam paper is for the course Lecture 401-0663-00L Numerical Methods for **Computer Science**, see the top of the front page.
- Turn the cover sheet only when instructed to do so.
- Make sure you have written your name number on every page.
- In your envelope you will find two blank sheets as scratch paper.
- **Write your answers in the appropriate (green) solution boxes on these problem sheets.**
- **Wrong ticks in multiple-choice boxes can lead to points being subtracted.** Hence, mere guessing is really dangerous! If you have no clue, leave all tickboxes empty.
- If you change your mind about an answer to a (multiple-choice) question, write a clear NO next to the old answer, draw fresh solution boxes/tickboxes and fill them.
- **Anything written outside the answer boxes will not be taken into account.**
- Do not write with red/green color or with pencil.
- **Duration: 30 minutes.**

- When the end of the exam is announced, make sure you have written your name on every sheet and put all of them back in the envelope.
- The exam proctors will collect the envelopes.

Special Covid-19 Safety Measures

- Only students in possession of a valid Covid Certificate are allowed to take the term exam.
- Protective masks covering nose and mouth have to be worn all the time.

Throughout the exam use the notations introduced in class, in particular [Lecture → Section 1.1.1]:

- $(\mathbf{A})_{i,j}$ to refer to the entry of the matrix $\mathbf{A} \in \mathbb{K}^{m,n}$ at position (i,j).
- $(\mathbf{A})_{:,i}$ to designate the ith-column of the matrix $\mathbf{A}$,
- $(\mathbf{A})_{i,:}$ to denote the i-th row of the matrix $\mathbf{A}$,
- $(\mathbf{A})_{i:j,k:\ell}$ to single out the sub-matrix $\left[(\mathbf{A})_{r,s}\right]_{\substack{i\leq r\leq j\\ k\leq s\leq \ell}}$ of the matrix $\mathbf{A}$,
- $(\mathbf{x})_k$ to reference the k-th entry of the vector $\mathbf{x}$,
- $\mathbf{e}_j \in \mathbb{R}^n$ to write the j-th Cartesian coordinate vector,
- $\mathbf{I}$ to denote the identity matrix,
- $\mathbf{O}$ to write a zero matrix,
- $\mathcal{P}_n$ for the space of (univariate polynomials of degree $\leq n$),
- and superscript indices in brackets to denote iterates: $\mathbf{x}^{(k)}$, etc.

By default, vectors are regarded as column vectors.

Problem 0-1: Asymptotic Cost of EIGEN-Based Functions

This short problem is about reading off the asymptotic computational cost of some C++ functions performing numerical linear algebra tasks based on EIGEN. It relies on information provided in [Lecture → Section 1.4.2], [Lecture → § 2.5.0.4], [Lecture → § 3.3.3.37], and [Lecture → Section 4.3].

Just reading of C++ code is required.

(0-1.a) (2 pts.) The function `slvtriag()` listed in Code 0.1.1 expects two vector arguments of equal length $n \in \mathbb{N}$. What is its asymptotic computational complexity for $n \to \infty$?

$$\text{cost}(\texttt{slvtriag}) = O(\quad\quad) \qquad \text{for} \quad n \to \infty\,.$$

C++ code 0.1.1: Function `slvtriag()`

```cpp
Eigen::VectorXd slvtriag(const Eigen::VectorXd &u, const Eigen::VectorXd &v) {
  assert((u.size() == v.size()) && "Size mismatch");
  const Eigen::MatrixXd A{u * v.transpose()};
  return A.triangularView<Eigen::Upper>().solve(u);
}
```

SOLUTION of (0-1.a):

- The function involves forming the tensor product of two vectors of length n, which created a densely populated $n \times n$-matrix at asymptotic cost of $O(n^2)$ for $n \to \infty$
- The solution of a *triangular* linear system of size n amounts to a simple elimination step with asymptotic cost $O(n^2)$ for $n \to \infty$, see [Lecture → § 2.3.2.15].

Hence the overall asymptotic complexity of `slvtriag()` is $O(n^2)$ for $n \to \infty$.

Remark. The function computes

$$\texttt{slvtriag}(\mathbf{u},\mathbf{v}) = \text{triu}(\mathbf{u}\mathbf{v}^\top)^{-1}\mathbf{u}\,.$$

▲

(0-1.b) (2 pts.) What is the asymptotic complexity of the C++ function `slvsymrom()` listed as Code 0.1.2 in terms of the length n of its argument vector `u`?

$$\text{cost}(\texttt{slvsymrom}) = O(\quad\quad) \qquad \text{for} \quad n \to \infty\,.$$

A sharp bound is expected.

C++ code 0.1.2: Function `slvsymrom()`

```cpp
Eigen::VectorXd slvsymrom(const Eigen::VectorXd &u) {
  const unsigned int n = u.size();
  return (Eigen::MatrixXd::Identity(n, n) + u * u.transpose()).lu().solve(u);
```

```
}
```

SOLUTION of (0-1.b):

A call to `slvsymrom()` involves the solution of a dense $n \times n$ linear system of equations by means of LU-decomposition and subsequent eliminations. The most costly step is the computation of the LU-decomposition with cost $O(n^3)$ for $n \to \infty$ [Lecture → Eq. (2.3.2.10)]. This step determines the asymptotic computational complexity of the function.

Remark. The result of the function is

$$\texttt{slvsymrom}(\mathbf{u}) = (\mathbf{I} + \mathbf{u}\mathbf{u}^\top)^{-1}\mathbf{u}\,.$$

▲

(0-1.c) (3 pts.) Code 0.1.3 lists the C++ function `getcompbas()`, whose argument `A` is a densely populated matrix $\mathbf{A} \in \mathbb{R}^{n,k}$, $k < n$. Give a sharp asymptotic bound for the asymptotic computational cost of `getcompbas()` for $n \to \infty$ assuming k to be small and fixed.

$$\mathrm{cost}(\texttt{getcompbas}) = O(\qquad) \qquad \text{for} \quad n \to \infty\,.$$

C++ code 0.1.3: Function `getcompbas()`

```
Eigen::MatrixXd getcompbas(const Eigen::MatrixXd &A) {
  const int n = A.rows();
  const int k = A.cols();
  Eigen::HouseholderQR<Eigen::MatrixXd> qr(A);
  return qr.householderQ() *
         (Eigen::MatrixXd(n, n - k) << Eigen::MatrixXd::Zero(k, n - k),
          Eigen::MatrixXd::Identity(n - k, n - k))
             .finished();
}
```

HINT 1 for (0-1.c): Note that the constructor of **Eigen::HouseholderQR<>** computes an economical QR-factorization stored in compressed format. The call `qr.householderQ()*` just applies Householder transformations. ⌟

SOLUTION of (0-1.c):

According to [Lecture → § 3.3.3.37] the computation of the (thin, which is the default in EIGEN) QR-decomposition of $\mathbf{A} \in \mathbb{R}^{n,k}$ incurs an asymptotic computational effort of $O(nk^2)$ for $n, k \to \infty$. Since k is small and fixed, this means $O(n)$ computational cost. Note that the Q-factor is never computed as a dense $n \times n$-matrix, but stored in compressed format as discussed in [Lecture → Rem. 3.3.3.21].

In the last step the Q-factor is multiplied with a matrix $\mathbf{M} := \begin{bmatrix} \mathbf{O} \\ \mathbf{I} \end{bmatrix}$ of size $n \times (n-k)$, resulting in a matrix of the same size. What is going on internally is that k Householder transformations are applied to $\mathbf{M}$, which creates a dense $n \times (n-k)$-matrix with $O(n^2)$ entries. This will take $O(n^2)$ operations for $n \to \infty$ and this determines the overall asymptotic cost of the function.

Remark. If $\mathbf{A}$ has full rank, the function `getcompbas()` returns an orthonormal basis of the orthogonal complement of $\mathcal{R}(\mathbf{A})$ in the columns of the result matrix.

▲

End Problem 0-1, 7 pts.

Problem 0-2: Matrix-Vector Product in CRS format

Sparse matrices have to be stored in special formats in order to save memory and inform algorithms about the position of non-zero entries. This problem will examine the CRS format

This problems is related to [Lecture → § 2.7.1.4] and assumes familiarity with C++.

The following data structure is used to store an $m \times n$-matrix in compressed row-storage (CRS) format:

```
struct CRSMatrix {
  unsigned int m;                      // number of rows
  unsigned int n;                      // number of columns
  std::vector<double> val;             // value array
  std::vector<unsigned int> col_ind;   // same length as value array
  std::vector<unsigned int> row_ptr;   // length m+1, row_ptr[m] ==
     val.size()
};
```

This format is defined by the relationship ("C++ indexing")

$$\texttt{val}[k] = a_{ij} \quad \Leftrightarrow \quad \begin{cases} \texttt{col\_ind}[k] = j\,, \\ \texttt{row\_ptr}[i] \le \texttt{k} < \texttt{row\_ptr}[i+1]\,, \end{cases} \quad 0 \le k \le \texttt{val.size()}\,,$$

for $i \in \{0,\dots,m-1\}$, $j \in \{0,\dots,n-1\}$.

(0-2.a) ⚃ (10 pts.) The function `crsmv` is supposed to return the product of a matrix in CRS format passed through `M` and of a vector given as argument `x`. Supplement the missing parts of the following listing code by writing valid C++ code into the boxes.

```
template <typename VECTORTYPE_I, typename VECTORTYPE_II>
VECTORTYPE_I crsmv(const CRSMatrix &M,
                   const VECTORTYPE_II &x) {
  assert((x.size() == M.[          ] )
         && "Size mismatch between x and M");
  VECTORTYPE_I y(M.m);
  for (int k = 0; k < [          ] ; ++k) {
    y[k] = 0;
    for (int j = M.[          ] ; j < [          ] ; ++j) {
      y[[          ]] += M.[          ] * x[[          ]];
    }
  }
  return y;
}
```

SOLUTION of (0-2.a):

The code processes the matrix row-wise (index varaible `k`) and, thus, sequentially runs through the `val` array and the `col_ind` array (index variable `j`).

C++ code 0.2.1: Function `crsmv()`

```
template <typename VECTORTYPE_I, typename VECTORTYPE_II>
VECTORTYPE_I crsmv(const CRSMatrix &M, const VECTORTYPE_II &x) {
  assert((x.size() == M.n) && "Size mismatch between x and M");
  VECTORTYPE_I y(M.m);
  for (int k = 0; k < M.m; ++k) {
    y[k] = 0;
    for (int j = M.row_ptr[k]; j < M.row_ptr[k + 1]; ++j) {
      y[k] += M.val[j] * x[M.col_ind[j]];
    }
  }
  return y;
}
```

▲

End Problem 0-2, 10 pts.

Problem 0-3: Economical Singular-Value Decomposition

For most applications requiring the singular-value decomposition (SVD) of matrix, its economical (thin) variant is sufficient. This problems examines some of its features.

This problems is based on the contents of [Lecture → Section 3.4.1].

Given a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $m, n \in \mathbb{N}$, $\mathbf{A} \neq \mathbf{O}$, we write $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ for its **economical (thin) SVD**.

(0-3.a) (3 pts.) Fill in the correct matrix sizes

$$\mathbf{U} \in \mathbb{R}^{\square,\square}\,,\quad \mathbf{\Sigma} \in \mathbb{R}^{\square,\square}\,,\quad \mathbf{V} \in \mathbb{R}^{\square,\square}\,.$$

HINT 1 for (0-3.a): Both cases $m \geq n$ and $m \leq n$ must be taken into account.

SOLUTION of (0-3.a):

$$\mathbf{U} \in \mathbb{R}^{m,\min\{m,n\}}\,,\quad \mathbf{\Sigma} \in \mathbb{R}^{\min\{m,n\},\min\{m,n\}}\,,\quad \mathbf{V} \in \mathbb{R}^{n,\min\{m,n\}}\,.$$

▲

(0-3.b) (6 pts.) Decide, whether the following statements are true for *every* $\mathbf{A} \in \mathbb{R}^{m,n} \setminus \{\mathbf{O}\}$, $m, n \in \mathbb{N}$.

1. The matrix $\mathbf{U}$ is an orthogonal matrix.

   ○ TRUE ○ FALSE

2. The set of all columns of $\mathbf{U}$ is an orthonormal basis of $\mathcal{R}(\mathbf{A})$.

   ○ TRUE ○ FALSE

3. $\mathbf{V}$ has orthogonal rows.

   ○ TRUE ○ FALSE

4. $\mathbf{V}^\top\mathbf{V} = \mathbf{I}$.

○ TRUE ○ FALSE

5. $\mathrm{nnz}(\boldsymbol{\Sigma}) := \sharp\{(i,j) : (\boldsymbol{\Sigma})_{i,j} \neq 0\} \leq n$.

○ TRUE ○ FALSE

6. $\mathbf{U}\mathbf{V}^\top = \mathbf{I}$.

○ TRUE ○ FALSE

SOLUTION of (0-3.b):

1. FALSE, because $\mathbf{U}$ need not be a square matrix. Only for the full SVD the U-factor would always be orthogonal.
2. FALSE, because if $m \geq n$, $\mathrm{rank}(\mathbf{A}) < n$, then the columns of $\mathbf{U}$ will span a space of dimension n, while $\mathcal{R}(\mathbf{A})$ has a smaller dimension.
3. FALSE, because only the columns of $\mathbf{V}$ are orthogonal, if $m < n$, $\mathbf{V}\mathbf{V}^\top = \mathbf{I}$.
4. TRUE, because $\mathbf{V}$ will always have orthonormal columns.
5. TRUE, because $\boldsymbol{\Sigma}$ is a *diagonal matrix*.
6. FALSE, because this matrix product may no even be defined. Even if it is, the rows of $\mathbf{U}$ and $\mathbf{V}$ are not related in any respect.

Also refer to [Lecture → § 3.4.1.4].

▲

(0-3.c) (3 pts.) What is the asymptotic computational effort for computing the economical SVD of a dense matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $m,n \in \mathbb{N}$?

$$\mathrm{cost}(\text{economical SVD of } \mathbf{A} \in \mathbb{R}^{m,n}) = O(\qquad\qquad)$$

for $m,n \to \infty$.

SOLUTION of (0-3.c):

According to [Lecture → § 3.4.2.2] we have

$$\mathrm{cost}(\text{economical SVD of } \mathbf{A} \in \mathbb{R}^{m,n}) = O(\min\{m,n\}^2 \max\{m,n\}) \quad \text{for} \quad m,n \to \infty\,.$$

▲

End Problem 0-3, 12 pts.