

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

Autumn Term 2022

Friday, Nov 4, 2022, 14:15, HG F 1

**Don't  
panic!**

Family Name		<b>Grade</b>
First Name		
Department		
Legi Nr.		
Date	Friday, Nov 4, 2022	

Points:

Prb. No.	1	2	3	Total
max				
achvd				

(100% = pts. ,  $\approx 40\%$  (passed) = 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.
- You will have **10 minutes advance-reading time**, during which you should read the exam problem, but you *must not write anything*.
- 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.

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  $i$ -th-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: Avoiding Cancellation by Rewriting Formulas

This problem studies two situations where careful implementation is necessary to avoid severe error amplification due to cancellation:

1. Blindly applying the well-known analytic formula for the roots of a quadratic polynomial can result in an implementation vulnerable to cancellation.
2. A code unstable for certain arguments will result from straightforwardly implementing the simple formula for the inverse function of the hyperbolic sine.

This problem revisits [Lecture → Ex. 1.5.4.17].

(0-1.a)  (4 pts.)

The task is to compute all real roots of the real quadratic polynomial

$$t \mapsto t^2 + \alpha t + \beta \quad \text{for given } \alpha, \beta \in \mathbb{R}.$$

The following C++ code is meant to implement a function

```
Eigen::Vector2d zerosquadpol(double a, double b);
```

that computes the zeros of  $t \mapsto t^2 + \alpha t + \beta$  in a stable way. The arguments  $a$  and  $b$  pass the coefficients  $\alpha$  and  $\beta$  of the polynomial. Supplement the missing parts.

#### C++ code 0.1.1: Stable computation of real roots of a quadratic polynomial

```
1 Eigen::Vector2d zerosquadpol(double a, double b) {
2   Eigen::Vector2d z(2);
3   double D = a * a - 4 * b;
4   if (D < 0) { throw "no real zeros"; }
5   else {
6     double wD = std::sqrt(D);
7     if (  >= 0) {
8       double t = ;
9       z << , ;
10    } else {
11      double t = ;
12      z << , ;
13    }
14  }
15  return z;
16 }
```

HINT 1 for (0-1.a): The well-known formula for the roots of the quadratic polynomial  $t \mapsto t^2 + \alpha t + \beta$  is

$$z_1 = \frac{1}{2}(-\alpha + \sqrt{D}) \quad , \quad z_2 = \frac{1}{2}(-\alpha - \sqrt{D}) \quad , \quad (0.1.2)$$

valid when the discriminant  $D := \alpha^2 - 4\beta$  is not negative. ┘

HINT 2 for (0-1.a): If the polynomial  $t \mapsto t^2 + \alpha t + \beta$  has two real zeros  $z_1$  and  $z_2$ , then

$$\beta = z_1 z_2 \quad \text{and} \quad \alpha = -(z_1 + z_2) \quad . \quad (0.1.3)$$

HINT 3 for (0-1.a): In EIGEN the `<<` can be used to initialize a vector by supplying a comma-separated list of values for its components.

SOLUTION of (0-1.a):

Refer to the C++ code listed as [Lecture → Code 1.5.4.18] for a complete solution and to [Lecture → Ex. 1.5.4.17] for additional explanations.

(0-1.b) (3 pts.) The inverse function for the hyperbolic sine  $\sinh(x) = \frac{1}{2}(e^x - e^{-x})$  is

$$\operatorname{arsinh}(x) := \log(x + \sqrt{x^2 + 1}), \quad x \in \mathbb{R}. \quad (0.1.4)$$

The following C++ code is to supply a *cancellation-free* implementation of  $\operatorname{arsinh}(x)$  for all  $x \in \mathbb{R}$ :

#### C++ code 0.1.5: Stable evaluation of $\operatorname{arsinh}(x)$

```

1 double arsinh(double x) {
2     if (x > 0) {
3         return           ;
4     } else {
5         return           ;
6     }
7 }
```

HINT 1 for (0-1.b): Remember the “ $(a - b)(a + b) = a^2 - b^2$ ”-trick.

SOLUTION of (0-1.b):

The direct implementation

```
double arsinh(double x) { return log(std::sqrt(1 + x * x) + x); }
```

will suffer from severe amplification of roundoff errors through cancellation in the case  $x < 0$  and  $|x| \gg 1$ . Fortunately, there is an equivalent formula

$$\begin{aligned} \log(\sqrt{x^2 + 1} + x) &= \log\left(\frac{(\sqrt{x^2 + 1} + x)(\sqrt{x^2 + 1} - x)}{\sqrt{x^2 + 1} - x}\right) \\ &= \log\left(\frac{1}{\sqrt{x^2 + 1} - x}\right) = -\log(\sqrt{x^2 + 1} - x). \end{aligned}$$

In formula cancellation will occur for  $x \gg 0$ , but not for  $x \ll 0$ . Therefore, one should use it for  $x < 0$ , whereas (0.1.4) has not problems for  $x > 0$ . This suggests the following implementation:

#### C++ code 0.1.6: Stable implementation of $\operatorname{arsinh}()$

```

2 double arsinh(double x) {
3     if (x > 0) {
```

```
4 |   return std::log(std::sqrt(1 + x * x) + x);  
5 | } else {  
6 |   return -std::log(std::sqrt(1 + x * x) - x);  
7 | }  
8 | }
```



**End Problem 0-1 , 7 pts.**

### Problem 0-2: Asymptotic Complexity of Numerical Linear Algebra Operations

Most numerical algorithms rely on basic operations from numerical linear algebra. In order to gauge the computational cost of these algorithm precise knowledge about the asymptotic computational effort involved in these operations is required. This problem asks you to analyze a given algorithm from in this respect.

This problem draws on [Lecture → Section 1.4.2] and [Lecture → Section 2.5]. No C++ implementation is requested.

The function

```
std::pair<double, Eigen::VectorXd>
  invit(Eigen::MatrixXd& A, double tol);
```

implements a so-called inverse power iteration. It takes a general (invertible) matrix  $A \in \mathbb{R}^{n,n}$ ,  $n \in \mathbb{N}$ , as argument A.

The C++ code for `invit()` is listed in Code 0.2.1.

#### C++ code 0.2.1: Function implementing inverse power iteration

```
2  std::pair<double, Eigen::VectorXd> invit(Eigen::MatrixXd& A,
3                                           double tol = 1.0E-6) {
4      assert((A.cols() == A.rows()) && "A must be square!");
5      const unsigned int n = A.cols();
6      Eigen::VectorXd y_new = Eigen::VectorXd::Random(n);
7      Eigen::VectorXd y_old(n);
8      double l_new = y_new.transpose() * A * y_new;
9      double l_old;
10     const auto A_lu_dec = A.lu(); //
11     y_new = y_new / y_new.norm();
12     do {
13         l_old = l_new;
14         y_old = y_new; //
15         y_new = A_lu_dec.solve(y_old); //
16         y_new /= y_new.norm(); //
17         l_new = y_new.transpose() * A * y_new; //
18     } while (std::abs(l_new - l_old) > tol * std::abs(l_new));
19     return {l_new, y_new};
20 }
```

(0-2.a) (1 pts.) What is the asymptotic complexity of the operations in Line 10,

```
const auto A_lu_dec = A.lu();
```

of Code 0.2.1?

$$\text{Cost}(\text{Line 10 of Code 0.2.1}) = O\left(\boxed{\phantom{000000}}\right) \text{ for } n \rightarrow \infty.$$

SOLUTION of (0-2.a):

This line of code calls an EIGEN built-in function for the computation of the LU-decomposition of the dense  $n \times n$ -matrix A. According to [Lecture → Thm. 2.5.0.2] this involves an asymptotic effort of

$O(n^3)$  for  $n \rightarrow \infty$ .



(0-2.b) (1 pts.) What is the asymptotic complexity of the operations in Line 14,

```
y_old = y_new;
```

of Code 0.2.1?

$$\text{Cost}(\text{Line 14 of Code 0.2.1}) = O(\boxed{\phantom{00}}) \text{ for } n \rightarrow \infty.$$

SOLUTION of (0-2.b):

This line of code just copies a vector of length  $n$ , which obviously takes  $O(n)$  machine operations. However, in the strict sense of

**Definition [Lecture → Def. 1.4.0.1]. Computational effort**

The **computational effort/computational cost** required/incurred by a numerical code amounts to the number of **elementary operations** (additions, subtractions, multiplications, divisions, square roots) executed in a run.

there is no (numerical) computational cost. Hence  $O(1)$  for  $n \rightarrow \infty$  would also be an acceptable answer.



(0-2.c) (1 pts.) What is the asymptotic complexity of the operations in Line 15,

```
y_new = A_lu_dec.solve(y_old);
```

of Code 0.2.1?

$$\text{Cost}(\text{Line 15 of Code 0.2.1}) = O(\boxed{\phantom{00}}) \text{ for } n \rightarrow \infty.$$

SOLUTION of (0-2.c):

The variable `A_lu_dec` contains the already computed LU-factors of **A**. Therefore the above line of code executes two triangular solves (backward and forward elimination). According to [Lecture → Thm. 2.5.0.3] the cost is  $O(n^2)$  for  $n \rightarrow \infty$ .



(0-2.d) (1 pts.) What is the asymptotic complexity of the operations in Line 16,

```
y_new /= y_new.norm();
```

of Code 0.2.1?

$$\text{Cost}(\text{Line 16 of Code 0.2.1}) = O\left(\boxed{\phantom{000000}}\right) \text{ for } n \rightarrow \infty.$$

SOLUTION of (0-2.d):

The eigen built-in member function `norm()` for a (real) vector computes its Euclidean norm by summing the squares of its entries. Subsequently each vector component is divided by the same number. All this takes  $O(n)$  elementary operations for  $n \rightarrow \infty$



**(0-2.e)** (1 pts.) What is the asymptotic complexity of the operations in Line 17,

```
l_new = y_new.transpose() * A * y_new;
```

of Code 0.2.1?

$$\text{Cost}(\text{Line 17 of Code 0.2.1}) = O\left(\boxed{\phantom{000000}}\right) \text{ for } n \rightarrow \infty.$$

SOLUTION of (0-2.e):

In linear-algebra notation Line 17 of Code 0.2.1) computes

$$\lambda_{\text{new}} = \mathbf{y}_{\text{new}}^{\top} \mathbf{A} \mathbf{y}_{\text{new}}.$$

Obviously this involves

- a matrix-vector product of an  $n \times n$ -matrix with an  $n$ -vector,
- and a dot product of two  $n$ -vectors.

The matrix-vector product dominates the total cost which boils down to  $O(n^2)$  for  $n \rightarrow \infty$ .



**End Problem 0-2 , 5 pts.**




### Problem 0-3: Extended Normal Equations

In [Lecture → § 3.2.0.7] we learned that sparsity of the system matrix of a large overdetermined linear system of equations, which is lost when forming the regular normal equations, can be preserved by switching to the so-called extended normal equations. This problem reviews those and examines the initialization of the corresponding system matrix.

The problem is based on [Lecture  $\rightarrow$  Section 3.1.2], [Lecture  $\rightarrow$  § 3.2.0.7], and requires familiarity with [Lecture  $\rightarrow$  Section 2.7.2].

Throughout this problem we consider a large *sparse* overdetermined linear system of equations  $\mathbf{Ax} = \mathbf{b}$ , with  $\mathbf{A} \in \mathbb{R}^{m,n}$ ,  $\mathbf{b} \in \mathbb{R}^m$  given,  $m \gg n \gg 1$ . The matrix  $\mathbf{A}$  is assumed to have full rank.

**(0-3.a)**  (3 pts.) By introducing the residual  $\mathbf{r} := \mathbf{Ax} - \mathbf{b}$  as additional auxiliary variable the **normal equations** [Lecture  $\rightarrow$  Eq. (3.1.2.2)] for  $\mathbf{Ax} = \mathbf{b}$  can be converted into the **extended normal equations**

$$\begin{bmatrix} \mathbf{r}_e \\ \mathbf{x}_e \end{bmatrix} = \begin{bmatrix} \mathbf{r}_e \\ \mathbf{x}_e \end{bmatrix}. \quad (0.3.1)$$

Complete the blocks of the system matrix and of the right-hand-side vector such that the solution of (0.3.1) provides the (unique) least-squares solution of  $\mathbf{Ax} = \mathbf{b}$  in  $\mathbf{x}_e$  and the residual  $\mathbf{r} := \mathbf{Ax} - \mathbf{b}$  in  $\mathbf{r}_e$ .

SOLUTION of (0-3.a):

As in [Lecture → § 3.2.0.7] we rewrite the normal equations for  $\mathbf{Ax} = \mathbf{b}$

$$\mathbf{A}^\top \mathbf{A} \mathbf{x} = \mathbf{a}^\top \mathbf{b} \Leftrightarrow \mathbf{A}^\top (\mathbf{A} \mathbf{x} - \mathbf{b}) = \mathbf{0} \Leftrightarrow \mathbf{A}^\top \mathbf{r} = \mathbf{0}.$$

This equation is combined with the definition of the residual

$$\begin{array}{l} -\mathbf{r} + \mathbf{A}\mathbf{x} = \mathbf{b} \\ \mathbf{A}^\top \mathbf{r} = \mathbf{0} \end{array} \Leftrightarrow \begin{bmatrix} -\mathbf{I} & \mathbf{A} \\ \mathbf{A}^\top & \mathbf{O} \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix}. \quad (0.3.2)$$

Here  $\mathbf{I} = \mathbf{I}_m$  is the  $m \times m$  identity matrix, and  $\mathbf{O} = \mathbf{O}_{n,n}$  is the  $n \times n$  zero matrix.

▲

**(0-3.b)**  (1 pts.) [ depends on Sub-problem (0-3.a) ]

What is the number of non-zero entries of the system matrix  $\mathbf{A}_e$  of the extended normal equation (0.3.1) in terms of the number  $\text{nnz}(\mathbf{A})$  of non-zero entries of  $\mathbf{A}$ ?

$$\text{nnz}(\mathbf{A}_e) =$$

SOLUTION of (0-3.b):

From the formula for  $\mathbf{A}_e$  from (??) we immediately see

$$\text{nnz}(\mathbf{A}_e) = 2 \text{nnz}(\mathbf{A}) + m.$$



(0-3.c) (3 pts.) [ depends on Sub-problem (0-3.a) ]

The function

```
TripletMatrix
extNeqSysMatCOO (
    unsigned int m, unsigned int n,
    const TripletMatrix &A_coo);
```

creates the matrix  $\mathbf{A}_e$  of the extended normal equations (0.3.1) in triplet (COO) format when given the matrix  $\mathbf{A} \in \mathbb{R}^{m,n}$  in triplet format through the argument  $\mathbf{A}$ , and its dimensions through  $m$  and  $n$ . The following typedefs have been used:

```
using Triplet = Eigen::Triplet<double>;
using TripletMatrix = std::vector<Triplet>;
```

Complete the implementation of `extNeqSysMatCOO()` given in Code 0.3.3.

#### C++ code 0.3.3: Initialization of $\mathbf{A}_e$ in triplet format

```
1 using Triplet = Eigen::Triplet<double>;
2 using TripletMatrix = std::vector<Triplet>;
3 TripletMatrix extNeqSysMatCOO(unsigned int m, unsigned int /*n*/,
4                               const TripletMatrix &A_coo) {
5     TripletMatrix A_ext_coo{};
6     for (int j = 0; j < (int)m; ++j) {
7         A_ext_coo.push_back(
8             Triplet{ , , });
9     }
10    for (const auto &t : A_coo) {
11        A_ext_coo.push_back(
12            Triplet{ , , });
13        A_ext_coo.push_back(
14            Triplet{ , , });
15    }
16    return A_ext_coo;
17 }
```

HINT 1 for (0-3.c): This is the definition of EIGEN's triplet class:

```

template<typename Scalar, typename StorageIndex=typename
    SparseMatrix<Scalar>::StorageIndex >
class Triplet
{
public:
    Triplet() : m_row(0), m_col(0), m_value(0) {}
    Triplet(const StorageIndex& i, const StorageIndex& j, const
        Scalar& v = Scalar(0))
        : m_row(i), m_col(j), m_value(v) {}
    const StorageIndex& row() const { return m_row; }
    const StorageIndex& col() const { return m_col; }
    const Scalar& value() const { return m_value; }
protected:
    StorageIndex m_row, m_col;
    Scalar m_value;
};

```

The type **StorageIndex** can be read as **unsigned int**.

SOLUTION of (0-3.c):

**C++ code 0.3.4: Initialization of system matrix of extended normal equations in triplet format.**

```

2  using Triplet = Eigen::Triplet<double>;
3  using TripletMatrix = std::vector<Triplet>;
4  TripletMatrix extNeqSysMatCOO(unsigned int m, unsigned int /*n*/,
5                               const TripletMatrix &A_coo) {
6      TripletMatrix A_ext_coo{};
7      // Set up the m x m identity matrix block
8      for (int j = 0; j < (int)m; ++j) {
9          A_ext_coo.push_back(Triplet{j, j, -1.0});
10     }
11     for (const auto &t : A_coo) {
12         // Right upper block A
13         A_ext_coo.push_back(Triplet{t.row(), t.col() + (int)m, t.value()});
14         // Left lower block AT
15         A_ext_coo.push_back(Triplet{t.col() + (int)m, t.row(), t.value()});
16     }
17     return A_ext_coo;
18 }

```

Note that the typecast **(int)** is included in order to suppress annoying compiler warnings about a “narrowing conversion”.

**End Problem 0-3 , 7 pts.**