

ETH Lecture 401-0663-00L Numerical Methods for CSE

Mid-Term Exam

Autumn Term 2019

Oct 18, 2019, 13:15, HG F 1 (A-L) & HG E 3 (M-Z)

Don't panic!

		%
Family Name		
First Name		
Department		
Legi Nr.		
Date	Oct 18, 2019	

Points:

	1	2	3	Total
max	20	16	20	56
achvd				

- This is a **closed-book exam.**
- Keep only writing material and your ETH ID card on the table.
- Keep mobile phones, tablets, smartwatches, etc. turned off in your bag.
- Fill in this cover sheet first.
- Turn the cover sheet only when instructed to do so.
- Then write your name and ETH ID number on every page.
- **Write your answers in the appropriate fields on these problem sheets.**
- **Wrong ticks in multiple-choice boxes will lead to points being subtracted.** Hence, mere guessing is really dangerous! If you have no clue, leave all tickboxes empty.
- If your chnage your mind about an answer to an MC-question, write a clear NO next to the old answer, draw fresh 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.
- Make sure to hand in every sheet.
- Two blank pages handed out with the exam: space for notes
- **Duration: 30 minutes.**

Throughout the exam use the notations introduced in [Lecture $\rightarrow$ 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-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.

By default, vectors are regarded as column vectors.

Problem 0-1: Rank-1 Modification

The problem addresses various aspects of the rank-1 modification of a matrix, as introduced in [Lecture → § 2.6.0.12].

Theoretical problem also asking for supplementing C++ code. Wrong ticks in multiple-choice parts incur a point penalty.

We recall the concept of a rank-1 modification:

Definition 0.1.1. Rank-1 Modification of a Matrix

A matrix $\widetilde{\mathbf{A}}$ is called a **rank-1 modification** of another matrix $\mathbf{A} \in \mathbb{K}^{m,n}$, if there exist two vectors $\mathbf{u} \in \mathbb{K}^m$ and $\mathbf{v} \in \mathbb{K}^n$ such that

$$\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{u}\mathbf{v}^{\mathrm{H}} . \tag{0.1.2}$$

(0-1.a) (8 pts.) Assess the correctness of the following statements about rank-1 modifications of a matrix:

(i) A rank-1 modification of $\mathbf{A} \in \mathbb{R}^{n,n}$ affects at most $2n-1$ entries of the matrix.

○ true ○ false

(ii) If $\widetilde{\mathbf{A}}$ is a rank-1 modification of $\mathbf{A} \in \mathbb{R}^{n,n}$, then

$$\operatorname{rank}(\mathbf{A}) - 1 \leq \operatorname{rank}(\widetilde{\mathbf{A}}) \leq \operatorname{rank}(\mathbf{A}) + 1 .$$

○ true ○ false

(iii) For every matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ there is an *invertible* $\widetilde{\mathbf{A}}$ arising from a rank-1 modification of $\mathbf{A}$.

○ true ○ false

(iv) By rank-1 modification every matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ can be converted into a *singular* (non-invertible) matrix.

○ true ○ false

SOLUTION of (0-1.a):

(i) This statement is false, because choosing $\mathbf{u} = \mathbf{1}$ and $\mathbf{v} = \mathbf{1}$, $\mathbf{1}$ the column vector with entries all $= 1$, in (0.1.2) will add 1 to *every* entry of $\mathbf{A}$.

(ii) This is true, because

- $\operatorname{rank}(\mathbf{u}\mathbf{v}^\top) \leq 1$ and, in general, $\operatorname{rank}(\mathbf{A}+\mathbf{B}) \leq \operatorname{rank}(\mathbf{A}) + \operatorname{rank}(B)$ for all matrices,
- the same argument can be applied "in reverse":

$$\operatorname{rank}(\mathbf{A}) = \operatorname{rank}(\widetilde{\mathbf{A}} - \mathbf{u}\mathbf{v}^{\mathrm{H}}) \leq \operatorname{rank}(\widetilde{\mathbf{A}}) + 1 .$$

(iii) The statement is false, because the zero matrix $\mathbf{O} \in \mathbb{R}^{2,2}$ provides a counterexample.

(iv) The statement is true: take $\mathbf{u} := -(\mathbf{A})_{:,1}$, $\mathbf{v} := \mathbf{e}_1$. Then the first column of $\widetilde{\mathbf{A}}$ will vanish.

(0-1.b) (4 pts.) Let $\widetilde{\mathbf{A}}$ be the matrix arising from $\mathbf{A} \in \mathbb{R}^{m,n}$, $m,n \in \mathbb{N}$, by replacing its k-th row $(\mathbf{A})_{k,:}$, $k \in \{1,\ldots,m\}$ with $\mathbf{w}^\top$, where $\mathbf{w} \in \mathbb{R}^n$ is a given vector. What rank-1 modification of $\mathbf{A}$ spawns $\widetilde{\mathbf{A}}$?

$$\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{u}\mathbf{v}^\top \quad \text{with} \quad \mathbf{u} = \boxed{}\,, \quad \mathbf{v} = \boxed{}\,.$$

SOLUTION of (0-1.b):

We may choose $\mathbf{u} = \mathbf{e}_k$ and $\mathbf{v} := \mathbf{w} - \left((\mathbf{A})_{k,:}\right)^\top$. Other choices, shuffling a scalar factor between $\mathbf{u}$ and $\mathbf{v}$ are also possible.

(0-1.c) (8 pts.) For an invertible/regular matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ and $\mathbf{u},\mathbf{v},\mathbf{b} \in \mathbb{R}^n$ the following C++ function should return $\mathbf{x} := (\mathbf{A} + \mathbf{u}\mathbf{v}^\top)^{-1}\mathbf{b}$ (if it exists) based on the Sherman-Morrison-Woodbury formula.

Lemma 0.1.3. Sherman-Morrison-Woodbury formula

For regular $\mathbf{A} \in \mathbb{K}^{n,n}$, *and* $\mathbf{U},\mathbf{V} \in \mathbb{K}^{n,k}$, $n,k \in \mathbb{N}$, $k \leq n$, *holds*

$$(\mathbf{A} + \mathbf{U}\mathbf{V}^H)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I} + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^H\mathbf{A}^{-1}\,,$$

if $\mathbf{I} + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U}$ *is regular/invertible.*

C++ code 0.1.4: Compute $\mathbf{x} := (\mathbf{A} + \mathbf{u}\mathbf{v}^\top)^{-1}\mathbf{b}$ based on Lemma 0.1.3.

```
template <class LUDec>
Eigen::VectorXd smw(const LUDec &lu, const Eigen::VectorXd &u,
                    const Eigen::VectorXd &v, const Eigen::VectorXd &b) {
  const Eigen::VectorXd z = lu.solve([    ]);
  const Eigen::VectorXd w = lu.solve([    ]);
  double q = 1.0 + [    ].dot([    ]);
  double p = [    ].dot([    ]);
  if (std::abs([    ]) < std::numeric_limits<double>::epsilon() * std::abs(p))
    throw std::runtime_error("Modified matrix nearly singular");
  else
    return ([    ] - w * p / q);
}
```

The argument `lu` passes the matrix $\mathbf{A}$ encapsulated in an object that provides a method

```
VectorXd solve(const VectorXd &y) const;
```

that computes the solution of the linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{y}$. The other arguments supply the vectors $\mathbf{u}, \mathbf{v}, \mathbf{b} \in \mathbb{R}^n$. Supplement the missing code in the boxes.

SOLUTION of (0-1.c):

Use the formula of the lemma for $k = 1$, see [Lecture → Eq. (2.6.0.22)]

$$\widetilde{\mathbf{x}} = \mathbf{A}^{-1}\mathbf{b} - \frac{\mathbf{A}^{-1}\mathbf{u}(\mathbf{v}^H(\mathbf{A}^{-1}\mathbf{b}))}{1 + \mathbf{v}^H(\mathbf{A}^{-1}\mathbf{u})} . \tag{0.1.5}$$

C++ code 0.1.6: Compute $\mathbf{x} := (\mathbf{A} + \mathbf{u}\mathbf{v}^\top)^{-1}\mathbf{b}$, see [Lecture → Code 2.7.3.3].

```
template <class LUDec>
Eigen::VectorXd smw(const LUDec &lu, const Eigen::VectorXd &u,
                    const Eigen::VectorXd &v, const Eigen::VectorXd &b) {
  const Eigen::VectorXd z = lu.solve(b); // z = A^{-1}b
  const Eigen::VectorXd w = lu.solve(u); // w = A^{-1}u
  double q = 1.0 + v.dot(w); // Compute denominator of (0.1.5)
  double p = v.dot(z);       // Factor for numerator of (0.1.5)
  if (std::abs(q) < std::numeric_limits<double>::epsilon() * std::abs(p))
    throw std::runtime_error("Modified matrix nearly singular");
  else
    return (z - w * p / q); // see (0.1.5)
}
```

The order of the vectors in the inner products can be swapped.

▲

End Problem 0-1, 20 pts.

Problem 0-2: Computational cost of numerical linear algebra operations

In the problem we face undocumented EIGEN based snippets of C++ codes that perform some operations on dense matrices and vectors. You will be asked to determine the asymptotic computational cost of these operations.

Purely theoretical problem related to [Lecture $\to$ Section 1.4].

The listings below display four EIGEN-based C++ functions that take a dense square matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ and a vector $\mathbf{b} \in \mathbb{R}^n$ as arguments `A` and `b`. In every case determine, in leading order, their asymptotic complexity for $n \to \infty$.

(0-2.a) (3 pts.)

C++ code 0.2.1: A function computing a scalar quantity.

```cpp
double sumtrv1(const Eigen::MatrixXd &A, const Eigen::VectorXd &b) {
  const int n = A.cols();
  assert((A.rows() == n) && (b.size() == n));

  return b.transpose() *
         A.triangularView<Eigen::Upper>().solve(
             Eigen::MatrixXd::Identity(n, n)) *
         b;
}
```

Asymptotic complexity for $n \to \infty$

○ $O(n)$ ○ $O(n^2)$ ○ $O(n^3)$ ○ $O(n^4)$

SOLUTION of (0-2.a):

The asymptotic complexity is $O(n^3)$, because the code is solving n linear systems of equations with an $n \times n$ upper triangular system matrix. This amounts to n backward substitutions, each of which costs $O(n^2)$ operations.

▲

(0-2.b) (3 pts.)

C++ code 0.2.2: Another function computing a scalar quantity.

```cpp
double sumtrv2(const Eigen::MatrixXd &A, const Eigen::VectorXd &b) {
  const int n = A.cols();
  assert((A.rows() == n) && (b.size() == n));

  return b.transpose() * A.triangularView<Eigen::Upper>().solve(b);
}
```

Asymptotic complexity for $n \to \infty$

○ $O(n)$ ○ $O(n^2)$ ○ $O(n^3)$ ○ $O(n^4)$

SOLUTION of (0-2.b):

The asymptotic complexity is $O(n^2)$, because we solve a single $n \times n$ upper triangular linear system of equations. This amounts to a backward substitutions, with asymptotic cost $O(n^2)$ operations. The vector operations do not matter, because they all require an effort $O(n)$.

(0-2.c) (5 pts.)

C++ code 0.2.3: A function computing a vector

```
Eigen::VectorXd diagmodsolve1(Eigen::MatrixXd A, const Eigen::VectorXd &b) {
  const int n = A.cols();
  assert((A.rows() == n) && (b.size() == n));
  Eigen::VectorXd x{Eigen::VectorXd::Zero(n)};
  double tmp = A(0, 0);
  for (int i = 0; i < n; ++i) {
    if (i > 0)
      A(i - 1, i - 1) = tmp;
    tmp = A(i, i);
    A(i, i) *= 2.0;
    x += A.lu().solve(b);
  }
  return x;
}
```

Asymptotic complexity for $n \to \infty$

○ $O(n)$ ○ $O(n^2)$ ○ $O(n^3)$ ○ $O(n^4)$

SOLUTION of (0-2.c):

The asymptotic complexity is $O(n^4)$, because we solve a dense $n \times n$ linear system of equations n times.

(0-2.d) (5 pts.)

C++ code 0.2.4: Another function computing a vector

```
Eigen::VectorXd diagmodsolve2(const Eigen::MatrixXd &A,
                              const Eigen::VectorXd &b) {
  const int n = A.cols();
  assert((A.rows() == n) && (b.size() == n));
  const auto Alu = A.lu(); //
  const auto z = Alu.solve(b);
  const auto W = Alu.solve(Eigen::MatrixXd::Identity(n, n)); //
  const Eigen::VectorXd alpha = Eigen::VectorXd::Constant(n, 1.0) +
                                  A.diagonal().cwiseProduct(W.diagonal());
  if ((alpha.cwiseAbs().array() < 1E-12).any())
    throw std::runtime_error("Tiny pivot!");
```

```
13     return n * z - W * z.cwiseProduct(A.diagonal().cwiseQuotient(alpha));
14 }
```

Asymptotic complexity for $n \to \infty$

○ $O(n)$ ○ $O(n^2)$ ○ $O(n^3)$ ○ $O(n^4)$

SOLUTION of (0-2.d):

The asymptotic complexity is $O(n^3)$, due to both, the LU-decomposition of an $n \times n$ densely populated matrix in Code 0.2.4, Line 6 and the n backsubstitutions to be carried out in Code 0.2.4, Line 8. The remaining operations incur cost of merely $O(n^2)$.

▲

End Problem 0-2, 16 pts.

Problem 0-3: Cancellation in Function Evaluations

In this exercise we study C++ functions that might be vulnerable to perilous amplification of round-off errors due to cancellation, if their argument lies within certain “critical ranges”. You will be asked to propose a mathematically equivalent implementation that avoids cancellation

A practical exercise connected with [Lecture → Section 1.5.4]

For the listed C++ functions, decide whether cancellation might make them return results with a large relative error for some arguments in their domains. If you conclude that this can occur, specify the “dangerous” range of arguments in the form

- $x \approx a$: cancellation for valid arguments close to a,
- $x \approx a_1, a_2, \ldots, a_n$: cancellation for valid arguments close any in a comma-separated sequence of values,
- $x \approx +\infty$: cancellation for large arguments $x \to \infty$,
- $x \approx -\infty$: cancellation for small arguments $x \to -\infty$.

Then propose an equivalent *cancellation-free* implementation.

(0-3.a) ⚁ (5 pts.)

C++ code 0.3.1: Function $f_1(x) := \log(\sqrt{x^2+1} - x)$

```
double f1(double x) { return std::log(std::sqrt(x * x + 1) - x); }
```

○ No cancellation ○ Cancellation for $x \approx$ ____________.

In case of cancellation an alternative mathematically equivalent implementation is (leave blank, if no cancellation for any valid argument)

C++ code 0.3.2: Cancellation-free implementation of f_1

```
double f1(double x) {
  return ____________
}
```

SOLUTION of (0-3.a):

We provide a detailed roundoff error analysis based on the “Axiom of Roundoff Analysis” [Lecture → Ass. 1.5.3.11]. More details can be found in [SB02, Ch. 1].

Lemma 0.3.3. Bound for accumulated relative errors

If $|\delta_\ell| \leq \mathtt{EPS}$ for some $0 \leq \mathtt{EPS} \ll 1$ and all $\ell \in \{1,\dots,n\}$, $n \in \mathbb{N}$, then

$$\prod_{\ell=1}^{n}(1+\delta_\ell)^{\pm 1} = 1+\delta \quad \textit{for some} \quad \delta \in \mathbb{R} \quad \textit{with} \quad |\delta| \leq \frac{n\mathtt{EPS}}{1-n\mathtt{EPS}} \,. \tag{0.3.4}$$

Proof. By simple induction w.r.t. n. □

Lemma 0.3.5. Bound for root of relative error

If $|\varepsilon| < 1$, then

$$\sqrt{1+\varepsilon} = 1+\delta \quad \textit{for some} \quad \delta \in \mathbb{R} \quad \textit{with} \quad |\delta| \leq \frac{\frac{1}{2}|\varepsilon|}{1-\frac{1}{2}|\varepsilon|} \,. \tag{0.3.6}$$

Proof. By the concavity of the function $x \mapsto \sqrt{x}$ we conclude $\sqrt{1+\varepsilon} \leq 1+\frac{1}{2}\varepsilon$, which also implies

$$\sqrt{\frac{1}{1+\varepsilon}} = \sqrt{1-\frac{\varepsilon}{1+\varepsilon}} \leq 1-\frac{\frac{1}{2}\varepsilon}{1+\varepsilon} = \frac{1+\frac{1}{2}\varepsilon}{1+\varepsilon} \,,$$

$$\sqrt{1+\varepsilon} \geq \frac{1+\varepsilon}{1+\frac{1}{2}\varepsilon} = 1+\frac{\frac{1}{2}\varepsilon}{1+\frac{1}{2}\varepsilon} \,,$$

which means for $\delta := \sqrt{1+\varepsilon}-1$,

$$\delta \leq \tfrac{1}{2}\varepsilon \quad \text{and} \quad \delta \geq \frac{\frac{1}{2}\varepsilon}{1+\frac{1}{2}\varepsilon} \,.$$

This yields the assertion of the lemma. □

We write $\widetilde{*}, \widetilde{+} \widetilde{-}$ for the elementary binary operations as realized in machine arithmetic. By [Lecture → Ass. 1.5.3.11] they satisfy

$$x \widetilde{\star} y = (x \star y)(1+\delta) \quad \text{with} \quad |\delta| \leq \mathtt{EPS} \,, \quad \star \in \{*,+,-\} \,. \tag{0.3.7}$$

where $0 < \mathtt{EPS} \ll 1$ is the machine precision. Moreover, we can also take for granted:

$$\begin{aligned} \texttt{std::sqrt}(x) &= \sqrt{x}(1+\delta) \,, \\ \texttt{std::log}(x) &= \log(x)(1+\delta) \,, \end{aligned} \quad \text{with} \quad |\delta| \leq \mathtt{EPS} \,. \tag{0.3.8}$$

Under these assumptions we find, thanks to (0.3.7),

$$\begin{aligned} x \widetilde{*} x \widetilde{+} 1 = \big(x^2(1+\delta_1)+1\big)(1+\delta_2) &= (x^2+1)\big(1+\frac{x^2}{x^2+1}\delta_1\big)(1+\delta_2) \\ &= (x^2+1)(1+\delta_3) \quad \text{with} \quad |\delta_3| \leq \frac{2\mathtt{EPS}}{1-2\mathtt{EPS}} \,, \end{aligned}$$

by Lemma 0.3.3. Here and in the sequel all so-called modifiers δ_ℓ are bounded (in modulus) by $\mathtt{EPS}$, unless specified otherwise. We continue

$$\texttt{std::sqrt}\,(x\tilde{*}x\tilde{+}1)\,\tilde{-}x \overset{(0.3.8)\,\&\,(0.3.7)}{=} \left(\sqrt{(x^2+1)(1+\delta_3)}(1+\delta_4)-x\right)(1+\delta_5)$$
$$= \left(\sqrt{x^2+1}\,\underbrace{(1+\delta_6)(1+\delta_4)}_{=1+\delta_7}-x\right)(1+\delta_5)$$
$$= \left(\sqrt{x^2+1}-x\right)\left(1+\frac{\sqrt{x^2+1}}{\sqrt{x^2+1}-x}\,\delta_7\right)(1+\delta_5)$$
$$= \left(\sqrt{x^2+1}-x\right)\left(1+\left(x^2+1+\sqrt{x^2+1}x\right)\delta_7\right)(1+\delta_5)$$
$$= \left(\sqrt{x^2+1}-x\right)(1+\Gamma(x))\,,\quad |\Gamma(x)| = O(x^2)\mathtt{EPS} \quad \text{for} \quad x\to\infty\,..$$

Roundoff error analysis is a *worst-case analysis*: we have to face the (unlikely) directed accumlation of roundoff errors. Hence, now we assume

$$\texttt{std::sqrt}\,(x\tilde{*}x\tilde{+}1)\,\tilde{-}x = \left(\sqrt{x^2+1}-x\right)(1+\Gamma(x)) \quad \text{with} \quad \Gamma(x)\geq Cx^2 \quad \text{for some} \quad C>0\,. \tag{0.3.9}$$

We infer

$$\texttt{std::log}\,(\texttt{std::sqrt}\,(x\tilde{*}x\tilde{+}1)\,\tilde{-}x) = \log\left(\left(\sqrt{x^2+1}-x\right)(1+\Gamma(x))\right)(1+\delta_8)$$
$$= \left[\log\left(\sqrt{x^2+1}-x\right)\left(1+\frac{\log(1+\Gamma(x))}{\log\left(\sqrt{x^2+1}-x\right)}\right)\right](1+\delta_8)$$
$$= \left[\log\left(\sqrt{x^2+1}-x\right)\left(1-\frac{\log(1+\Gamma(x))}{\underbrace{\log\left(\sqrt{x^2+1}+x\right)}_{\to\infty \text{ for } x\to\infty}}\right)\right](1+\delta_8)\,.$$

We conclude that in the worst case

$$\texttt{std::log}\,(\texttt{std::sqrt}\,(x\tilde{*}x\tilde{+}1)\,\tilde{-}x) = \log\left(\sqrt{x^2+1}-x\right)(1+\Gamma^*(x))\,,$$

with $|\Gamma^*(x)|\to\infty$ for $x\to\infty$. Hence, the result of f_1 can be marred by ever larger relative errors due to roundoff as $x\to\infty$. In other word, cancellation will hit Code 0.3.1 for $x\approx+\infty$. A remedy is the usual expansion trick using the binomial formula $a-b=\frac{a^2-b^2}{a+b}$, which yields the stable formula

$$\log\left(\sqrt{x^2+1}-x\right) = \begin{cases} \log\left(\dfrac{1}{\sqrt{x^2+1}+x}\right) = -\log\left(\sqrt{x^2+1}+x\right) & \text{for} \quad x>0\,, \\ \log\left(\sqrt{x^2+1}-x\right) & \text{for} \quad x<0\,. \end{cases}$$

Its stability can be proved by exactly the same estimates as elaborated above, swapping a single "-" for a "+". Also note that there is no cancellation for $x\to-\infty$: The term $\Gamma(x)$ in the estimates above remains $\sim\mathtt{EPS}$ in this case.

C++ code 0.3.10: Cancellation-free implementation of f_1

```
double f1c(double x) {
  return (x > 0.0) ? -std::log((std::sqrt(x * x + 1) + x))
                   : std::log(std::sqrt(x * x + 1) - x);
}
```

▲

(0-3.b) (5 pts.)

C++ code 0.3.11: Function $f_2(x) := \log(x^2+1) - 2\log x$

```cpp
double f2(double x) {
  assert(x > 0);
  return std::log(x * x + 1) - 2 * std::log(x);
}
```

○ No cancellation ○ Cancellation for $x \approx$ ______.

In case of cancellation an alternative mathematically equivalent implementation is (leave blank, if no cancellation for any valid argument)

C++ code 0.3.12: Cancellation-free implementation of f_2

```cpp
double f2(double x) {
  assert(x > 0);
  return 
}
```

SOLUTION of (0-3.b):

Cancellation will hit us for $x \approx +\infty$. We use the functional equation $\log a - \log b = \log \frac{a}{b}$ to prevent it.

C++ code 0.3.13: Cancellation-free implementation of f_2

```cpp
double f2c(double x) {
  assert(x > 0);
  const double y = 1.0 / x;
  return std::log(y * y + 1);
}
```

(0-3.c) (5 pts.)

C++ code 0.3.14: Function $f_3(x) := 1 - \sqrt{1-x^2}$

```cpp
double f3(double x) {
  assert((x >= -1) && (x <= 1));
  return 1 - std::sqrt(1 - x * x);
}
```

○ No cancellation ○ Cancellation for $x \approx$ ______.

In case of cancellation an alternative mathematically equivalent implementation is (leave blank, if no cancellation for any valid argument)

C++ code 0.3.15: Cancellation-free implementation of f_3

```
double f3(double x) {
  assert((x >= -1) && (x <= 1));
  return
}
```

SOLUTION of (0-3.c):

Cancellation will hit us for $x \approx 0$! No serious cancellation effects can be observed for $x \approx 1$, because the small result of $\sqrt{1-x^2}$, which may have a large relative error, will subsequently subtracted from 1 and the relative error will become small again. We apply the usual expansion trick using the binomial formula $a - b = \frac{a^2-b^2}{a+b}$.

C++ code 0.3.16: Cancellation-free implementation of f_3

```
double f3c(double x) {
  assert((x >= -1) && (x <= 1));
  const double s = x * x;
  return s / (1 + std::sqrt(1 - s));
}
```

▲

(0-3.d) (5 pts.)

C++ code 0.3.17: Function $f_4(x) := \sqrt{1-\cos^2 x}$

```
double f4(double x) {
  const double s = std::cos(x);
  return std::sqrt(1 - s * s);
}
```

○ No cancellation ○ Cancellation for $x \approx$ ______.

In case of cancellation an alternative mathematically equivalent implementation is (leave blank, if no cancellation for any valid argument)

C++ code 0.3.18: Cancellation-free implementation of f_4

```
double f4(double x) {
  return
}
```

SOLUTION of (0-3.d):

Cancellation will hit us for $x \approx \ldots, -\pi, 0, \pi, 2\pi, \ldots 0$! We use the trigonometric identity $\cos^2\xi + \sin^2\xi = 1$ to avoid it. Do not forget the modulus!

C++ code 0.3.19: Cancellation-free implementation of f_4

```
double f4c(double x) { return std::abs(std::sin(x)); }
```

▲

References

[SB02] J. Stoer and R. Bulirsch. *Introduction to numerical analysis*. Third. Vol. 12. Texts in Applied Mathematics. Springer-Verlag, New York, 2002, pp. xvi+744. DOI: 10.1007/978-0-387-21738-3 (cit. on p. 10).

End Problem 0-3, 20 pts.