Name: ETH ID No.:

ETH Lecture 401-2673-00L Numerical Methods for **CSE**

# Repeat Examination

Autumn Term 2024

Wednesday, Aug 6, 2025, 10:00, HG D 12

**Don't panic!**

| | | **Grade** |
|---|---|---|
| Family Name | | |
| First Name | | |
| Department | | |
| Legi Nr. | | |
| Date | Wednesday, Aug 6, 2025 | |

Points:

| Prb. No. | 1 | 2 | 3 | Total |
|---|---|---|---|---|
| max | 35 | 23 | 32 | 90 |
| achvd | | | | |

(100% = 85 pts. , ≈40% (passed) = 34 pts.)

- **Duration: 180 minutes + 30 minutes advance reading**
- Cell phones and other communication devices are not allowed. Make sure that they are turned off and stowed away in your bag.
- No own notes or other aids are permitted during the exam!
- You may cite theorems, lemmas, and equations from the lecture document by specifying their precise number in the supplied PDF.
- *Write clearly with a non-erasable pen. Do not use red pen or green pen.* No more than one solution can be handed in per problem. Invalid attempts should be clearly crossed out.
- Write your name on every problem sheet in the blank line at the top of the pages.
- Write clean solutions of theoretical tasks in the boxes with the green frame.
- If a tasks is to be answered in a free-text box, include relevant preparatory considerations, auxiliary computations, etc. in your solution.
- Get a general idea of the problems. Pay attention to the number of points awarded for each subtask. It is roughly correlated with the amount of work the task will require.
- **If you have failed to solve a sub-problem, do not give up on the entire problem**, but try the next one. Dependencies between sub-problems are indicated where relevant.
- For coding tasks, for which no solution boxes are provided, only the code files will be examined for grading.
- Codes will be graded like theoretical problems: Partial solutions and correct approaches and ideas will be rewarded as long as they are evident from the code. Hence, it is advisable to write tidy and well-structured codes and add some comments.

- In coding tasks, even if you could not fully implement a function, you may still call it in a following coding task.
- Syntax errors in coding tasks lead to at most one point being deducted per task.

**At the beginning of the exam**

Follow the steps listed below. This is not counted as exam time.

*You are not allowed to start the exam until you have a clear indication from an assistant or the Professor.*

1. Upon entering the examination room, starting from the far end of the room sit down at any available desk.
2. Put your ETH ID card ("legi") on the table.
3. Fill in the blanks on the cover page of the problem sheet. Do not turn pages yet!
4. On your desk you will find scrap paper.
5. The computer screen should show an exam selection page. You can change the language of the page (upper right). In case you have requested a US keyboard, you can change the keyboard layout (bottom right corner).
6. Please provide your information (last name, first name, legi-nr.), select the correct exam from the drop-down menu, and continue.
7. Log into Moodle as a member of ETH Zürich using your NETHZ username and password.
8. You should now see the start page of the Moodle exam. You can not start the exam, but you can open the available resources (C++ Reference, Eigen documentation, Lecture document). Navigate between windows using alt+tab. You can also arrange the windows side by side.
9. Wait until everybody is finished with the previous steps.
10. Only when you are asked so, turn the pages of the problem sheets. Then, you may read through all the problems and also click through the provided documents. However, *during the advance reading time you must not touch the keyboard or a pen*!
11. You are allowed to remove the staples, but you must not forget to *write your name on every page*.
12. After the 30 minutes of advance reading, the exam password will be communicated and you can begin the exam.

**Instructions concerning CodeExpert**

- When you enter the password and start the Moodle exam, a timer of 180 minutes is started. Once the timer expires, your attempt is submitted and you can no longer make changes to your answers. Your answers are automatically saved at regular intervals.
- You will see the Code Expert environment (CE) embedded in a small window. It may take a moment to load. This is the coding part of the first problem.
- In the top right corner of the CE, you see a blue icon that allows you to view the CE in full screen mode.
- At the bottom of the CE, you can click 'Console' to get access to the 'Run' and 'Test' buttons. At the right side, you can click 'Task' to view a short task description. At the left side, you can click 'Project file system' to see the available files for this problem.

- Under ‘Task’ on the right hand side, there is a ‘message’ icon. If it shows a notification symbol, please check the announcement sent by the examiners.
- To navigate to the next problem, first minimize the CE (click the blue icon in the top right corner), and then click ‘Next page’. You can return to a previous problem by pressing ‘Previous page’.
- At the final problem of the exam, you will see a ‘Finish attempt...’. If you click this button, you will see a summary of your attempt. Then you can decide to submit your solution, or to return to your attempt.
- Concerning implementation in C++ you will only be asked to complete classes or (member) functions in existing header files. The sections of the code you have to supply will be marked by

  ```
  // TO DO: ...
  // START

  // END
  ```

  Insert your code between these two markers. Some variables may already have been defined earlier in the template code.
- Note that codes rejected by the compiler will incur a point penalty.
- *C++-code* that has been commented out will not be considered as part of your solution. *Text* comments may be taken into account, however.
- You are free to edit the file ‘main.cpp’, but its contents will not be considered as part of your solution.
- Each problem has test cases in ‘tests.cpp’. These tests do not determine your grade.

## At the end of the examination

1. **Do not log out and do not turn off the computer!**
2. Make sure that you have written you name on the top of all pages of the exam paper.
3. Wait until your row will be called to leave the room.

## Problems

- If you have problems with the computer, please raise your hand to get support, right in the beginning of the exam, if possible.
- In case your computer fails irrevocably (“freezes”) you will be assigned another one.

## Notations

Throughout the exam use the notations introduced in class, in particular $\left[\text{Lecture} \rightarrow \text{Section 1.1.1}\right]$:

- $(\mathbf{A})_{i,j}$ 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 $\mathbf{A}$,
- $\mathbf{A}^\top$ for the transposed matrix,
- $\otimes$ to denote the Kronecker product,
- “$\cdot$” as an alternative way to write the Euclidean inner product of two vectors,

- $(\mathbf{x})_k$ 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,
- $\mathbf{I}$ ($\mathbf{I}_n$) to denote the identity matrix (of size $n \times n$),
- $\mathbf{O}$ to write a zero matrix,
- $\mathcal{P}_n$ for the space of (univariate polynomials of degree $\leq n$),
- superscript indices in brackets to denote iterates: $\mathbf{x}^{(k)}$, etc,
- vectors regarded as column vectors by default.

## Problem 0-1: Computing with Kronecker Products

In [Lecture → Ex. 1.4.3.8] we could catch a glimpse of the fact that matrix operations can be carried out with significantly reduced cost, if (some of) the involved matrices have Kronecker product structure.

This problem is connected to [Lecture → Ex. 1.4.3.8] and [Lecture → Section 3.4.4.2]. It requests implementation in C++ based on EIGEN.

We recall the definition of the Kronecker product

**Definition [Lecture → Def. 1.4.3.7]. Kronecker product**

The **Kronecker product** $\mathbf{A} \otimes \mathbf{B}$ of two matrices $\mathbf{A} \in \mathbb{K}^{m,n}$ and $\mathbf{B} \in \mathbb{K}^{\ell,k}$, $m, n, \ell, k \in \mathbb{N}$, is the $(m\ell) \times (nk)$-matrix

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} (\mathbf{A})_{1,1}\mathbf{B} & (\mathbf{A})_{1,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{1,n}\mathbf{B} \\ (\mathbf{A})_{2,1}\mathbf{B} & (\mathbf{A})_{2,2}\mathbf{B} & & & \vdots \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ (\mathbf{A})_{m,1}\mathbf{B} & (\mathbf{A})_{m,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{m,n}\mathbf{B} \end{bmatrix} \in \mathbb{K}^{m\ell,nk} .$$

and the concept of the **vectorization** of a matrix by stacking its columns on top of each other:

$$\mathrm{vec} : \mathbb{K}^{n,m} \to \mathbb{K}^{n \cdot m} \quad , \quad \mathrm{vec}(\mathbf{A}) := \begin{bmatrix} (\mathbf{A})_{:,1} \\ (\mathbf{A})_{:,2} \\ \vdots \\ (\mathbf{A})_{:,m} \end{bmatrix} \in \mathbb{R}^{n \cdot m} . \qquad \text{[Lecture} \to \text{(1.2.3.5)]}$$

When solving the sub-problems you can rely on the following properties of the Kronecker product

**Lemma 0.1.1. Properties of matrix Kronecker products**

- *The Kronecker product according to [Lecture → Def. 1.4.3.7] is a bilinear mapping* $\mathbb{R}^{m,n} \times \mathbb{R}^{\ell,k} \to \mathbb{R}^{m\ell,nk}$.
- *If* $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ *are matrices of such size that the matrix products* $\mathbf{AC}$ *and* $\mathbf{BD}$ *can be formed, then*

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = (\mathbf{AC}) \otimes (\mathbf{BD}) . \tag{0.1.2}$$

- *For all* $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{\ell,k}$, $m, n, \ell, k \in \mathbb{N}$, $\mathbf{X} \in \mathbb{R}^{k,n}$,

$$(\mathbf{A} \otimes \mathbf{B})\,\mathrm{vec}(\mathbf{X}) = \mathrm{vec}\left(\mathbf{B}\mathbf{X}\mathbf{A}^\top\right) . \tag{0.1.3}$$

- *For all* $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{\ell,k}$, $m, n, \ell, k \in \mathbb{N}$,

$$(\mathbf{A} \otimes \mathbf{B})^\top = \mathbf{A}^\top \otimes \mathbf{B}^\top . \tag{0.1.4}$$

**(0-1.a)** ⚁ (1 pts.) Let $\mathbf{A} \in \mathbb{R}^{m,m}$ and $\mathbf{B} \in \mathbb{R}^{n,n}$ be regular/invertible matrices. Then

$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \boxed{\phantom{XXXX}} \otimes \boxed{\phantom{XXXX}} .$$

SOLUTION of (0-1.a):

We exploit the mixed product formula (0.1.2) and find

$$(\mathbf{A}\otimes\mathbf{B})\left(\mathbf{A}^{-1}\otimes\mathbf{B}^{-1}\right) = \left(\mathbf{A}\,\mathbf{A}^{-1}\right)\otimes\left(\mathbf{B}\,\mathbf{B}^{-1}\right) = \mathbf{I}_m\otimes\mathbf{I}_n = \mathbf{I}_{mn}\;,$$

which immediately implies

$$\boxed{(\mathbf{A}\otimes\mathbf{B})^{-1} = \mathbf{A}^{-1}\otimes\mathbf{B}^{-1}}\;. \tag{0.1.5}$$

▲

**(0-1.b)** (4 pts.) Given $\mathbf{A}\in\mathbb{R}^{m,n}$ and $\mathbf{B}\in\mathbb{R}^{\ell,k}$, $m,n,\ell,k\in\mathbb{N}$, with $m\geq n$ and $\ell\geq k$, let

$$\mathbf{A} = \mathbf{Q}_A\mathbf{R}_A\;,\quad \begin{array}{l}\mathbf{Q}_A^\top\mathbf{Q}_A = \mathbf{I}_n\\ \mathbf{R}_A\in\mathbb{R}^{n,n}\text{ upper triangular}\end{array}\;,\quad \mathbf{B} = \mathbf{Q}_B\mathbf{R}_B\;,\quad \begin{array}{l}\mathbf{Q}_B^\top\mathbf{Q}_B = \mathbf{I}_k\\ \mathbf{R}_B\in\mathbb{R}^{k,k}\text{ upper triangular,}\end{array}$$

be economical/thin **QR-factorizations** of $\mathbf{A}$ and $\mathbf{B}$, respectively. Then an economical QR-factorization of the Kronecker product $\mathbf{A}\otimes\mathbf{B}$ is

$$\mathbf{A}\otimes\mathbf{B} = \underbrace{\boxed{\phantom{\mathbf{Q}_A\otimes\mathbf{Q}_B\qquad}}}_{\mathbf{Q}\text{-factor}}\cdot\underbrace{\boxed{\phantom{\mathbf{R}_A\otimes\mathbf{R}_B\qquad}}}_{\mathbf{R}\text{-factor}}\;.$$

SOLUTION of (0-1.b):

Again, the argument relies on the mixed product property (0.1.2),

$$\boxed{\mathbf{A}\otimes\mathbf{B}} = (\mathbf{Q}_A\mathbf{R}_B)\otimes(\mathbf{Q}_B\mathbf{R}_B) = \boxed{\underbrace{(\mathbf{Q}_A\otimes\mathbf{Q}_B)}_{\in\mathbb{R}^{m\ell,nk}}\cdot\underbrace{(\mathbf{R}_A\otimes\mathbf{R}_B)}_{\in\mathbb{R}^{nk,nk}}}\;. \tag{0.1.6}$$

- Thanks to (0.1.2) and (0.1.4) we have

  $$(\mathbf{Q}_A\otimes\mathbf{Q}_B)^\top(\mathbf{Q}_A\otimes\mathbf{Q}_B) = \left(\mathbf{Q}_A^\top\mathbf{Q}_A\right)\otimes\left(\mathbf{Q}_B^\top\mathbf{Q}_B\right) = \mathbf{I}_n\otimes\mathbf{I}_k = \mathbf{I}_{nk}\;.$$

  Hence, the Q-factor satisfies the essential orthogonality properties.
- $\mathbf{R}_A\otimes\mathbf{R}_B$ is upper triangular directly by the definition of the Kronecker product. It is easy to see that its diagonal entries are all positive.

▲

**(0-1.c)** (12 pts.) [ depends on Sub-problem (0-1.a) ]

In the file `kronprodmisc.hpp` implement an *efficient* C++ function

```
Eigen::VectorXd solveAkpBSys(
   const Eigen::MatrixXd &A, const Eigen::MatrixXd &B,
   const Eigen::VectorXd &y);
```

which solves the linear system of equations

$$(\mathbf{A} \otimes \mathbf{B})\mathbf{x} = \mathbf{y}$$

for given regular matrices $\mathbf{A} \in \mathbb{R}^{m,m}$, $\mathbf{B} \in \mathbb{R}^{n,n}$, and given right-hand side vector $\mathbf{y} \in \mathbb{R}^{m \cdot n}$, and returns the solution vector $\mathbf{x} \in \mathbb{R}^{m \cdot n}$.

HINT 1 for (0-1.c): Use the result of Sub-problem (0-1.a) and then (0.1.3). ⌟

HINT 2 for (0-1.c): Make use of reshaping of matrices and vectors by means of EIGEN's **Map** class, see [Lecture → Code 1.4.3.9]. ⌟

---

SOLUTION of (0-1.c):

Using (0.1.5) we have, with $\mathbf{Y} \in \mathbb{R}^{n,m}$ such that $\mathrm{vec}(\mathbf{Y}) = \mathbf{y}$,

$$\mathbf{x} = \left(\mathbf{A}^{-1} \otimes \mathbf{B}^{-1}\right)\mathbf{y} \overset{(0.1.3)}{=} \mathrm{vec}\left(\mathbf{B}^{-1}\mathbf{Y}\mathbf{A}^{-\top}\right) = \mathrm{vec}\left(\left(\mathbf{A}^{-1}\left(\mathbf{B}^{-1}\mathbf{Y}\right)^{\top}\right)^{\top}\right). \quad (0.1.7)$$

The left matrix products with $\mathbf{A}^{-1}$ and $\mathbf{B}^{-1}$ must be realized by means of Gaussian elimination with multiple right-hand sides, see [Lecture → § 2.5.0.4].

The following code directly implements (0.1.7) using EIGEN's **Map** class for realizing $\mathrm{vec}$.

**C++ code 0.1.8: Solving a Kronecker product linear system**

```cpp
Eigen::VectorXd solveAkpBSys(const Eigen::MatrixXd &A, const Eigen::MatrixXd &B,
                             const Eigen::VectorXd &y) {
  const Eigen::Index m = A.rows();
  const Eigen::Index n = B.rows();
  assertm(A.cols() == m, "Matrix A must be square!");
  assertm(B.cols() == n, "Matrix B must be squares!");
  assertm(y.size() == m * n, "Right-hand side vecvtor must have size n*m");
#if SOLUTION
  Eigen::MatrixXd T =
      (A.lu().solve(
           (B.lu().solve(Eigen::Map<const Eigen::MatrixXd>(y.data(), n, m)))
               .transpose()))
          .transpose();
  return Eigen::Map<Eigen::VectorXd>(T.data(), m * n, 1);
#else
  // **********************************************
  // Your code here
  // **********************************************
#endif
}
```

Of course, an implementation with several temporary vectors is possible, too. One may also first perform LU-decompositions of $\mathbf{A}$ and $\mathbf{B}$ and then invoke `solve()` vector by vector.

---

▲

**(0-1.d)** ⚁ (3 pts.) [ depends on Sub-problem (0-1.c) ]

What is the asymptotic complexity of your implementation of `solveAkpBSys()` from Subproblem (0-1.c) for $m, n \to \infty$. Give a sharp bound for general densely populated matrices.

$$\mathrm{cost}(\texttt{solveAkpBSys()}) = O\left(\boxed{\phantom{xxxxxxxxxxxxxxxx}}\right) \quad \text{for} \quad m, n \to \infty\,.$$

SOLUTION of (0-1.d):

- We have to compute the LU-decompositions of both $\mathbf{A}$ and $\mathbf{B}$, which incurs asymptotic cost of $O(n^3 + m^3)$ for $m, n \to \infty$.
- Then we perform backward/forward elimination for $\mathbf{B}$ and $m$ right-hand side vectors. This involves a computational effort of $O(n^2 \cdot m)$ for $m, n \to \infty$.
- Afterwards we carry out backward/forward elimination for $\mathbf{A}$ and $n$ right-hand side vectors, which will cost $O(m^2 \cdot n)$ for $m, n \to \infty$.

So the total asymptotic cost is

$$\mathrm{cost}(\texttt{solveAkpBSys()}) = O(m^3 + m^2 n + mn^2 + n^3) = O(n^3 + m^3) \quad \text{for} \quad m, n \to \infty\,.$$

▲

Following [VP93] we study the best approximation of matrices by Kronecker products of smaller matrices. The key idea in [VP93] relies on the **rearrangement operator**

$$\mathrm{rar}_{\ell,k} : \mathbb{R}^{m\ell,nk} \to \mathbb{R}^{mn,\ell k}\,, \quad \mathrm{rar}_{\ell,k}(\mathbf{A}) := \begin{bmatrix} \widehat{\mathbf{A}}_1 \\ \vdots \\ \widehat{\mathbf{A}}_n \end{bmatrix}, \quad \widehat{\mathbf{A}}_j = \begin{bmatrix} \mathrm{vec}(\mathbf{B}_{1,j})^\top \\ \vdots \\ \mathrm{vec}(\mathbf{B}_{m,j})^\top \end{bmatrix}, \tag{0.1.9}$$

based on the block decomposition

$$\mathbf{A} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} & \dots & \mathbf{B}_{1,n} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} & \dots & \mathbf{B}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{B}_{m,1} & \mathbf{B}_{m,2} & \dots & \mathbf{B}_{m,n} \end{bmatrix}, \quad \mathbf{B}_{i,j} \in \mathbb{R}^{\ell,k}\,. \tag{0.1.10}$$

The authors of [VP93] establish the following remarkable identity.

**Lemma 0.1.11. Frobenius distance to Kronecker product**

*Let $\mathbf{A} \in \mathbb{R}^{m\cdot\ell,b\cdot k}$ be given. If $\mathbf{X} \in \mathbb{R}^{m,n}$ and $\mathbf{Y} \in \mathbb{R}^{\ell,k}$, then*

$$\|\mathbf{A} - \mathbf{X} \otimes \mathbf{Y}\|_F = \left\| \mathrm{rar}_{\ell,k}(\mathbf{A}) - \mathrm{vec}(\mathbf{X})\ \mathrm{vec}(\mathbf{Y})^\top \right\|_F\,,$$

*where $\|\cdot\|_F$ stands for the **Frobenius norm** [Lecture → Def. 3.4.4.17] of a matrix.*

**(0-1.e)** ⚃ (15 pts.) Based on Lemma 0.1.11 implement (in the file `kronprodmisc.hpp`) a C++ function

```
std::pair<Eigen::MatrixXd, Eigen::MatrixXd>
  kronprodBestApprox(const Eigen::MatrixXd &A,
                     Eigen::Index l, Eigen::Index k);
```

that solves the following best approximation problem:

> Given $\mathbf{A} \in \mathbb{R}^{m\cdot\ell,n\cdot k}$ find $\mathbf{X} \in \mathbb{R}^{m,n}$ and $\mathbf{Y} \in \mathbb{R}^{\ell,k}$ such that
>
> $$\|\mathbf{A} - \mathbf{X} \otimes \mathbf{Y}\|_F = \min\left\{\|\mathbf{A} - \mathbf{X}' \otimes \mathbf{Y}'\|_F : \mathbf{X}' \in \mathbb{R}^{m,n},\ \mathbf{Y}' \in \mathbb{R}^{\ell,k}\right\} .$$

The argument `A` passes the matrix $\mathbf{A}$ and `l` and `k` give the size of the second Kronecker factor. They must divide the numbers of rows and columns, respectively, of $\mathbf{A}$.

HINT 1 for (0-1.e): [Lecture → Thm. 3.4.4.19] tells you how to compute a rank-1 best approximation.

┘

---

SOLUTION of (0-1.e):

The message of Lemma 0.1.11 is that the best approximation of $\mathbf{A}$ by means of a Kronecker product can be obtained through a **rank-1 best approximation** of the rearrangement $\mathrm{rar}_{\ell,k}(\mathbf{A})$; we have to find $\mathbf{x} \in \mathbb{R}^{m\cdot n}$ and $\mathbf{y} \in \mathbb{R}^{\ell\cdot k}$ such that, writing $\mathbf{R} := \mathrm{rar}_{\ell,k}(\mathbf{A})$,

$$\begin{aligned}\left\|\mathbf{R} - \mathbf{x}\mathbf{y}^\top\right\|_F &= \min\left\{\left\|\mathbf{R} - \widetilde{\mathbf{x}}\widetilde{\mathbf{y}}^\top\right\|_F : \widetilde{\mathbf{x}} \in \mathbb{R}^{mn},\ \widetilde{\mathbf{y}} \in \mathbb{R}^{\ell k}\right\} \\ &= \min\{\|\mathbf{R} - \mathbf{M}\|_F : \mathbf{M} \in \mathcal{R}_1(mn, \ell k)\} ,\end{aligned} \tag{0.1.12}$$

where $\mathcal{R}_1(mn, \ell k)$ is the set of $mn \times \ell k$-matrices with $\mathrm{rank} = 1$. This is the notation used in [Lecture → Thm. 3.4.4.19].

Owing to [Lecture → Thm. 3.4.4.19] a solution of this best approximation problem is

$$\mathbf{x} = (\mathbf{\Sigma})_{1,1} \cdot (\mathbf{U})_{:,1} \quad \text{and} \quad \mathbf{y} = (\mathbf{V})_{:,1} ,$$

where

$$\mathbf{R} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top , \quad \mathbf{U} \in \mathbb{R}^{mn,mn},\ \mathbf{V} \in \mathbb{R}^{\ell k,\ell k},\ \mathbf{\Sigma} \in \mathbb{R}^{mn,\ell k}$$

is a **singular value decomposition (SVD)** [Lecture → Def. 3.4.1.3] of $\mathbf{R}$. Its computation by means of EIGEN's built-in linear algebra utilities is explained in [Lecture → Section 3.4.2], see [Lecture → Code 3.4.2.1]. In particular [Lecture → Code 3.4.4.20] invoked with `k=1` realizes the solution of (0.1.12).

The computation of the rearrangement of $\mathbf{R}$ of $\mathbf{A}$ can be done using four nested loops. The outer two traverse the block decomposition (0.1.10) in column-major order, the inner two sweep the blocks in column major order.

Finally, we have to convert the vectors $\mathbf{x}$ and $\mathbf{y}$ into matrices by "inverse vectorization". This can easily be done using EIGEN's **Map** utility.

**C++ code 0.1.13: Computation of best approximation by means of a Kronecker factorization**

```
std::pair<Eigen::MatrixXd, Eigen::MatrixXd> kronprodBestApprox(
    const Eigen::MatrixXd &A, Eigen::Index l, Eigen::Index k) {
  Eigen::Index m = A.rows() / l;  // rows() of 1st Kronecker factor
  Eigen::Index n = A.cols() / k;  // cols() of 1st Kronecker factor
  assertm(A.rows() == m * l, "l does not divide A.rows()!");
  assertm(A.cols() == n * k, "k does not divide A.cols()!");
#if SOLUTION
  // Compute rearrangments
  Eigen::MatrixXd R(m * n, l * k);
  Eigen::Index r_row = 0;
  for (Eigen::Index j = 0; j < n; ++j) {
    for (Eigen::Index i = 0; i < m; ++i, ++r_row) {
      Eigen::Index r_col = 0;
      for (Eigen::Index j_in = 0; j_in < k; ++j_in) {
        for (Eigen::Index i_in = 0; i_in < l; ++i_in, ++r_col) {
          R(r_row, r_col) = A(i * l + i_in, j * k + j_in);
        }
      }
    }
  }
  // Compute rank-1 best approximation of R
  const Eigen::JacobiSVD<Eigen::MatrixXd> svdR(
      R, Eigen::ComputeThinU | Eigen::ComputeThinV);
  const double sigma_max = svdR.singularValues()[0];
  const Eigen::VectorXd vec_X = sigma_max * svdR.matrixU().col(0);
  const Eigen::VectorXd vec_Y = svdR.matrixV().col(0);
  return {Eigen::Map<const Eigen::MatrixXd>(vec_X.data(), m, n),
          Eigen::Map<const Eigen::MatrixXd>(vec_Y.data(), l, k)};
#else
  // **************************************************
  // Your code here
  // **************************************************
#endif
}
```

Of course, all EIGEN block operations can be replaced by simple loops and vice versa.

▲

## References

[VP93] C. F. Van Loan and N. Pitsianis. "Approximation with Kronecker products". In: *Linear algebra for large scale and real-time applications (Leuven, 1992)*. Vol. 232. NATO Adv. Sci. Inst. Ser. E: Appl. Sci. Kluwer Acad. Publ., Dordrecht, 1993, pp. 293–314 (cit. on p. 10).

**End Problem 0-1** , 35 pts.

## Problem 0-2: Rational Functions of Type $(m-1, m)$

In this problem we study a particular class of rational functions for which the degree of the numerator polynomial is smaller than the degree of the denominator polynomial. The focus is on algorithms for the converting different representations of these functions into each other.

This problem requires familiarity with [Lecture → Section 8.4].

The following terminology is borrowed from approximation theory [Tre13, Chapter 23]:

**Definition 0.2.1. Rational function of type $(m, n)$**

Given $m, n \in \mathbb{N}_0$ and two polynomials $p \in \mathcal{P}_m$ and $q \in \mathcal{P}_n$ with non-vanishing leading coefficient the quotient function

$$x \mapsto \frac{p(x)}{q(x)}, \quad x \in \mathbb{R} \setminus \{z \in \mathbb{R} : q(z) = 0\},$$

is a **rational function of type** $(m, n)$.

In this problem we consider a particular class of rational functions. For $m \in \mathbb{N}$ we define

$$\mathcal{X}_m := \left\{ x \mapsto c\frac{(x-z_1)(x-z_2)\cdot\dots\cdot(x-z_{m-1})}{(x-a_1)(x-a_2)\cdot\dots\cdot(x-a_m)} : \begin{array}{l} [z_1,\dots,z_{m-1}] \in \mathbb{R}^{m-1}, \\ [a_1,\dots,a_m] \in \mathbb{R}^m,\ c \in \mathbb{R}, \\ z_i \neq a_j \quad \begin{array}{l}\forall\, i = 1,\dots,m-1, \\ \forall j = 1,\dots,m\end{array} \end{array} \right\}.$$

This set contains special rational functions of type $(m-1, m)$.

The following data type is used to represent rational functions in $\mathcal{X}_m$.

**C++ code 0.2.2: Struct describing $r \in \mathcal{X}_m$**

```cpp
struct RatFuncX {
  RatFuncX() = delete;
  RatFuncX(const RatFuncX &) = default;
  RatFuncX(RatFuncX &&) = default;
  RatFuncX &operator=(const RatFuncX &) = default;
  ~RatFuncX() = default;

  template <typename VEC1, typename VEC2>
  RatFuncX(const VEC1 &z, const VEC2 &a, double c) :
      a_(a.size()), z_(z.size()), c_(c), m_(a.size()) {
    assertm(z.size() == m_ - 1, "Size misatch for z!");
    for (size_t i = 0; i < z.size(); ++i) z_[i] = z[i];
    for (size_t i = 0; i < a.size(); ++i) a_[i] = a[i];
  }
  std::vector<double> a_;  // zeros of denominator
  std::vector<double> z_;  // zeros of numerator
  double c_;               // coefficient c
  size_t m_;               // rationa function of type (m-1,m)
};
```

**(0-2.a)** ⚁ (1 pts.) Give an example of a rational function $r$ of type $(1, 2)$, which does not belong to

$\mathcal{X}_2$.

$$r(x) = \boxed{\phantom{xxxxxxxxxxxxxxxxxxxx}} .$$

SOLUTION of (0-2.a):

The requirement $r \in \mathcal{X}_2$ implies that the denominator of $r$ must have at least one real zero. This is not the case, for instance, for

$$r(x) = \frac{x}{1+x^2} .$$

▲

**(0-2.b)** (10 pts.) In the file `zerosratfunc.hpp` implement the C++ function

```
std::vector<double> partialFractionExpansion(const RatFuncX &r);
```

that computes the weights $w_j \in \mathbb{R}$. $j = 1, \dots, m$ for the **partial fraction expansion**

$$c\frac{(x-z_1)(x-z_2)\cdot\dots\cdot(x-z_{m-1})}{(x-a_1)(x-a_2)\cdot\dots\cdot(x-a_m)} = \sum_{j=1}^{m} \frac{w_j}{x-a_j} , \quad x \notin \{a_1, \dots, a_m\} , \tag{0.2.3}$$

and returns $(w_1, \dots, w_m)$.

The **RatFuncX**-type argument `r` contains all information on $m$, $z_j$ and $a_j$.

We assume that the zeros of the denominator are all distinct: $a_i \neq a_j$, if $i \neq j$.

SOLUTION of (0-2.b):

Multiply both sides of (0.2.3) with $(x-a_1)(x-a_2)\cdot\dots\cdot(x-a_m)$!

$$c(x-z_1)(x-z_2)\cdot\dots\cdot(x-z_{m-1}) = \sum_{j=1}^{m} w_j \prod_{\substack{\ell=1\\ \ell\neq j}}^{m} (x-a_\ell) , \quad x \in \mathbb{R} . \tag{0.2.4}$$

we observe that

- the left-hand side of (0.2.4) is a polynomial of degree $m-1$, and
- that $\{x \mapsto \prod_{\substack{\ell=1\\ \ell\neq j}}^{m}(x-a_\ell)\}_{j=1}^{m}$ is a basis of $\mathcal{P}_{m-1}$, because $\dim \mathcal{P}_{m-1} = m$ and linear independence of these functions is straightforward.

Hence, the basis expansion coefficients $w_j$ are uniquely determined.

Plug $x = a_j$, $j = 1, \dots, m$, into (0.2.4).

$$c(a_j - z_1)(a_j - z_2) \cdot \dots \cdot (a_j - z_{m-1}) = w_j \underbrace{\prod_{\substack{\ell=1 \\ \ell \neq j}}^{m} (a_j - a_\ell)}_{\neq 0} .$$

$$w_j = \frac{c \prod_{\ell=1}^{m-1} (a_j - z_\ell)}{\prod_{\substack{\ell=1 \\ \ell \neq j}}^{m} (a_j - a_\ell)} , \quad j = 1, \dots, m . \qquad (0.2.5)$$

This formula is implemented in the following code.

**C++ code 0.2.6: Computation of weights of partial fraction expansion**

```cpp
std::vector<double> partialFractionExpansion(const RatFuncX &r) {
  std::vector<double> w(r.m_);
  if (r.m_ == 1) {
    w[0] = r.c_;
  } else {
    // Check validity of data
    for (unsigned int i = 1; i < r.m_; ++i) {
      for (unsigned int j = 0; j < i; ++j) {
        assertm((r.a_[i] != r.a_[j]), "Zeros of denominator must be distinct!");
      }
    }
#if SOLUTION
    for (unsigned int j = 0; j < r.m_; ++j) {
      w[j] = r.c_;
      unsigned int k = 0;
      for (unsigned int l = 0; l < r.m_ - 1; ++l, ++k) {
        if (k == j) {
          ++k;
        }
        w[j] *= (r.a_[j] - r.z_[l]) / (r.a_[j] - r.a_[k]);
      }
    }
#else
// **************************************************
// Your code here
// **************************************************
#endif
  }
  return w;
}
```

**Remark.** The algorithm implemented in Code 0.2.6 is the most efficient way to compute the weights $w_j$. However, they can also be obtained by a **collocation approach**: find $w_j \in \mathbb{R}$, $j = 1, \dots, m$, such that

$$c\frac{(x_\ell - z_1)(x - z_2) \cdot \dots \cdot (x_\ell - z_{m-1})}{(x_\ell - a_1)(x_\ell - a_2) \cdot \dots \cdot (x_\ell - a_m)} = \sum_{j=1}^{m} \frac{w_j}{x_\ell - a_j} \quad \forall \ell \in \{1, \dots, m\} , \qquad (0.2.7)$$

and for a set of **collocation points**

$$\{x_1,\dots,x_m\} \in \mathbb{R} \setminus \{a_1,\dots,a_m\} . \tag{0.2.8}$$

At second glance, (0.2.7) is an $m \times m$ linear system of equations for the unknown $w_j$

$$\begin{bmatrix} \frac{1}{x_1-a_1} & \cdots & \frac{1}{x_1-a_m} \\ \vdots & & \vdots \\ \frac{1}{x_m-a_1} & \cdots & \frac{1}{x_m-a_m} \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ \vdots \\ w_m \end{bmatrix} = \begin{bmatrix} c\frac{(x_1-z_1)(x_1-z_2)\cdot\cdots\cdot(x_1-z_{m-1})}{(x_1-a_1)(x-a_2)\cdot\cdots\cdot(x_1-a_m)} \\ \vdots \\ c\frac{(x_m-z_1)(x_m-z_2)\cdot\cdots\cdot(x_m-z_{m-1})}{(x_m-a_1)(x-a_2)\cdot\cdots\cdot(x_m-a_m)} \end{bmatrix} .$$

How to choose the collocation points? Actually *any* choice complying (0.2.8) is suitable (in theory). A reasonable choice is the midpoints of the intervals $]a_j, a_{j+1}[$ plus some other point and this choice is adopted in Code 0.2.9.

**C++ code 0.2.9: Computation of weights of partial fraction expansion by collocation approach**

```cpp
std::vector<double> partialFractionExpansion_alt(const RatFuncX &r) {
  std::vector<double> w(r.m_);
  if (r.m_ == 1) {
    w[0] = r.c_;
  } else {
    // Check validity of data
    for (unsigned int i = 1; i < r.m_; ++i) {
      for (unsigned int j = 1; j < i; ++j) {
        assertm((r.a_[i] != r.a_[j]), "Zeros of denominator must be distinct!");
      }
    }
#if SOLUTION
    // Collocation approach: determine unknown coefficients w_j by
       solving a
    // linear system of equations Find a suitable set of collocation
       points:
    // Here take the midpoints of the intervals defined by the pole
       positions.
    Eigen::VectorXd x(r.m_);
    std::vector<unsigned int> idx(r.a_.size());  // Sorting indices
    for (unsigned int i = 0; i < idx.size(); ++i) idx[i] = i;
    std::sort(idx.begin(), idx.end(),
              [&](const unsigned int &i, const unsigned int &j) {
                return (r.a_[i] < r.a_[j]);
              });
    // Check validity of data
    for (unsigned int i = 1; i < r.m_; ++i) {
      assertm((r.a_[idx[i]] - r.a_[idx[i - 1]] > 0.0), "Sorting failed!");
    }
    x[0] = r.a_[idx[0]] - 0.5 * (r.a_[idx[1]] - r.a_[idx[0]]);
    for (unsigned int i = 1; i < r.m_; ++i) {
      x[i] = 0.5 * (r.a_[idx[i]] + r.a_[idx[i - 1]]);
    }
    // System matrix for collocation LSE
    Eigen::MatrixXd C(r.m_, r.m_);
    for (unsigned int i = 0; i < r.m_; ++i) {
      for (unsigned int j = 0; j < r.m_; ++j) {
        C(i, j) = 1.0 / (x[i] - r.a_[j]);
      }
    }
    // Right-hand side vector for collocation LSE
    Eigen::VectorXd rhs(r.m_);
    for (unsigned int i = 0; i < r.m_; ++i) {
      double val = 1.0;
      for (unsigned int j = 0; j < r.m_; ++j) {
        val /= (x[i] - r.a_[j]);
      }
      for (unsigned int j = 0; j < r.m_ - 1; ++j) {
        val *= (x[i] - r.z_[j]);
      }
      rhs[i] = r.c_ * val;
    }
    // Solve linear system and write result into vector w
    Eigen::Map<Eigen::VectorXd>(w.data(), r.m_) = C.lu().solve(rhs);
#else
// **************************************************
// Your code here
// **************************************************
#endif
  }
  return w;
}
```

Of course, this way of computing the $w_j$ is much more expensive, asymptotic cost $O(m^3)$ vs. $O(m^2)$ for $m \to \infty$, than the simple formula (0.2.5).

▲

**(0-2.c)** ⚅ (12 pts.) In the file `zerosratfunc.hpp` complete the code of the C++ function

```
std::vector<double>
  zerosPFE(const std::vector<double> &w,
    const std::vector<double> &a,
    double reltol, double abstol)
```

that relies on **Newton's method** to compute the $m-1$ distinct zeros of the rational function

$$r(x) = \sum_{j=1}^{m} \frac{w_j}{x - a_j}, \quad x \notin \{a_1, \ldots, a_m\},$$

$$\text{with} \quad w_j > 0 \quad \text{and} \quad a_j \in \mathbb{R}, \quad a_1 < a_2 < \cdots < a_m .$$

The arguments `w` and `a` pass the coefficient sequence $(w_1, \ldots, w_m)$ and the pole position sequence $(a_1, \ldots, a_m)$, respectively, the latter of which is assumed to be sorted.

The parameters `reltol` and `abstol` provide the tolerances for the **correction-based termination** of the Newton iterations.

HINT 1 for (0-2.c):

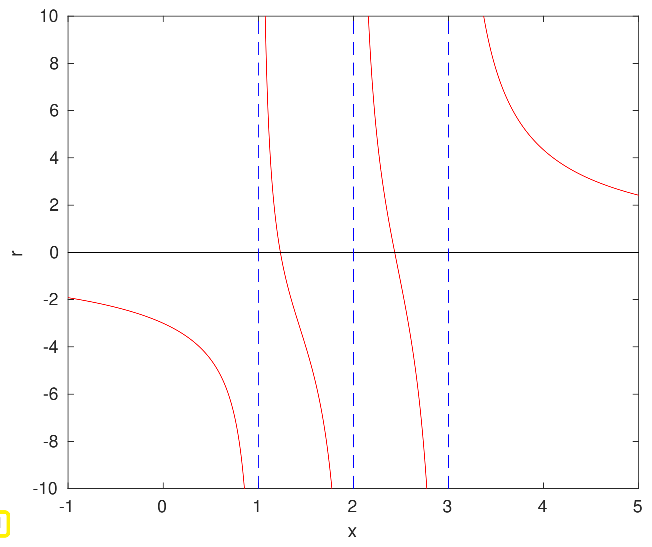

Fig. 1

◁ Graph of $r \in \mathcal{X}$ for

- $m = 3$,
- $a_1 = 1$, $a_2 = 2$, $a_3 = 3$,
- $w_1 = 1$, $w_2 = 2$, $w_3 = 3$.

┘

SOLUTION of (0-2.c):

**Remark.** The numerator of $r \in \mathcal{X}$ is a polynomial of degree $m-1$, which means that $r$ can have $m-1$ zeros at most. Thanks to $w_j > 0$ the function $r$ varies from $+\infty$ to $-\infty$ in each of the $m-1$ intervals $]a_j, a_{j+1}[$ and is decreasing. This guarantees the existence of exactly one zero of $x \mapsto r(x)$ in each interval $]a_j, a_{j+1}[$! There are no zeros outside $]a_1, a_m[$.

Hence, for each interval $]a_j, a_{j+1}[$, $j = 1, \dots, m-1$, we run the Newton iteration [Lecture → (8.4.2.1)]

$$z_j^{(k+1)} := z_j^{(k)} + s \,, \quad s := -\frac{r(z_j^{(k)})}{r'(z_j^{(k)})} \,, \quad k = 0, 1, 2, \dots$$

until [Lecture → Code 8.4.2.2]

$$|s| < \texttt{reltol} \cdot |z| \quad \text{or} \quad |s| < \texttt{abstol} \,.$$

The derivative of $x \mapsto r(x)$ is

$$r'(x) = -\sum_{j=1}^{m} \frac{w_j}{(x - a_j)^2} \,, \quad x \neq a_j \,.$$

We see that as a consequence of $w_j > 0$ we can conclude $r'(x) > 0$ for all $x \neq a_j$. The confirms that $x \mapsto r(x)$ is decreasing on its entire domain of definition.

As initial value we may choose a “suitable” $z_j^{(0)} \in ]a_j, a_{j+1}[$, for instance

- the midpoint of the interval $z_j^{(0)} = \frac{1}{2}(a_j + a_{j+1})$, or
- the zero of the function

$$x \mapsto \frac{w_j}{x - a_j} + \frac{w_{j+1}}{x - a_{j+1}} \quad \blacktriangleright \quad z_j^{(0)} = \frac{w_{j+1} a_j + w_j a_{j+1}}{w_j + w_{j+1}} \,,$$

which is the choice made in the code below.

⚠ There does not seem to be a rigorous guarantee that the iterates $z_j^{(k)}$ remain in the interval $]a_j, a_{j+1}[$!

**C++ code 0.2.10: Finding all zeros of $r$**

```cpp
std::vector<double> zerosPFE(const std::vector<double> &w,
                             const std::vector<double> &a, double reltol,
                             double abstol) {
  const size_t m = a.size();
  assertm(w.size() == m, "Sizes of arrays a and w have to agree!");
  // Check sorting and signs of data
  for (size_t j = 1; j < m; ++j) {
    assertm((a[j] - a[j - 1] > 0.0), "Pole positions have to be sorted!");
  }
  for (size_t j = 0; j < m; ++j) {
    assertm((w[j] > 0.0), "Weights have to be positive!");
  }
  // Vector of zeros
  std::vector<double> zj(m - 1);
#if SOLUTION
  // Run through the intervals between poles
  for (size_t j = 0; j < m - 1; ++j) {
    // Initial guess
    double z = (w[j] * a[j + 1] + w[j + 1] * a[j]) / (w[j] + w[j + 1]);
    double dz;
    size_t it_cnt = 0;
    const size_t it_max = 10;
    do {
      it_cnt++;
      // Evaluate r and the derivative of r at z
      double r_val = 0.0;
      double dr_val = 0.0;
      for (size_t k = 0; k < m; ++k) {
        const double t = z - a[k];
        const double s = w[k] / t;
        r_val += s;
        dr_val -= s / t;
      }
      dz = r_val / dr_val;  // Newton correction
      z -= dz;              // Update
      assertm((z > a[j]) && (z < a[j + 1]), "Iteration out of bounds!");
    } while ((std::abs(dz) > reltol * std::abs(z)) && (std::abs(dz) > abstol) &&
             (it_cnt < it_max));
    zj[j] = z;
  }
#else
  // **************************************************
  // Your code here
  // **************************************************
#endif
  return zj;
}
```

▲

## References

[Tre13] Lloyd N. Trefethen. *Approximation theory and approximation practice*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2013, viii+305 pp.+back matter (cit. on p. 13).

**End Problem 0-2** , 23 pts.

## Problem 0-3: Implicit Runge-Kutta Single-Step Method

In [Lecture → Chapter 12] we learned that the use of *implicit* Runge-Kutta single step methods (IRK-SSMs) is mandatory for the efficient numerical integration of *stiff* initial value problems for ordinary differential equations. In this problem we examine the details of an implementation of a generic IRK-SSM.

This problem assumes familiarity with [Lecture → Section 12.3.3] and [Lecture → Section 8.5.4]. Small-scale coding in C++ is part of it.

Recall that a general $s$-stage, $s \in \mathbb{N}$, Runge-Kutta single-step method according to [Lecture → Def. 12.3.3.1] is fully characterized by its **Butcher scheme**,

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array} , \qquad \text{[Lecture → (12.3.3.3)]}$$

with the Butcher matrix $\mathfrak{A} = [a_{i,j}]_{i,j=1}^s$, $\mathbf{c} \in \mathbb{R}^s$, and weight vector $\mathbf{b} = [b_i]_{i=1}^s \in \mathbb{R}^s$.

**(0-3.a)** ⚁ (5 pts.) We apply the general $s$-stage RK-SSM given by the Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$, to the *non-autonomous* linear ODE $\mathbf{M}\dot{\mathbf{y}} + \mathbf{T}\mathbf{y} = \mathbf{q}(t)$ with state space $\mathbb{R}^N$, $N \in \mathbb{N}$, and a function $\mathbf{q} : \mathbb{R} \to \mathbb{R}^N$. Here $\mathbf{M}, \mathbf{T} \in \mathbb{R}^{N,N}$ are symmetric positive definite (s.p.d.) matrices.

Complete the following formulas describing one step $t_0 \to t_1$, $\mathbf{y}_0 \to \mathbf{y}_1$ of the RK-SSM for that ODE:

$$\mathbf{k}_i \in \mathbb{R}^N: \quad \begin{bmatrix} \mathbf{X}_{1,1} & \mathbf{X}_{1,2} & \cdots & \cdots & \mathbf{X}_{1,2} \\ \mathbf{X}_{2.1} & \mathbf{X}_{2,2} & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ \mathbf{X}_{s,1} & \mathbf{X}_{s,2} & \cdots & \cdots & \mathbf{X}_{s,s} \end{bmatrix} \begin{bmatrix} \mathbf{k}_1 \\ \mathbf{k}_2 \\ \vdots \\ \vdots \\ \mathbf{k}_s \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \vdots \\ \mathbf{b}_s \end{bmatrix} ,$$

$$\mathbf{y}_1 := \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{k}_i , \quad h := t_1 - t_0 ,$$

with $\mathbf{X}_{i,j} = \boxed{\phantom{xxxxxxxxxxxxxxxx}} \in \mathbb{R}^{N,N}$ , $i,j \in \{1,\ldots,s\}$ ,

$\mathbf{b}_i = \boxed{\phantom{xxxxxxxxxxxxxxxx}} \in \mathbb{R}^N$ , $i \in \{1,\ldots,s\}$ .

⚠ No inverses of matrices must occur in the formulas!

SOLUTION of (0-3.a):

We start from the increment form of the defining equations of an $s$-stage RK-SSM:

**Definition [Lecture → Def. 12.3.3.1]. General Runge-Kutta single step method**

For $b_i, a_{ij} \in \mathbb{R}$, $c_i := \sum_{j=1}^{s} a_{ij}$, $i,j = 1,\dots,s$, $s \in \mathbb{N}$, a single step of size $h > 0$ of an **$s$-stage Runge-Kutta single step method** (RK-SSM) for the IVP [Lecture → (11.1.3.2)] is defined by

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + h\sum_{j=1}^{s} a_{i,j}\mathbf{k}_j) \;, \quad i = 1,\dots,s \quad , \quad \mathbf{y}_1 := \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{k}_i \;.$$

The vectors $\mathbf{k}_i \in \mathbb{R}^N$ are called **increments**.

We temporarily rewrite the ODE in the standard form by multiplying with $\mathbf{M}^{-1}$, which exists, because an s.p.d. matrix is regular,

$$\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t)) := -\mathbf{M}^{-1}(\mathbf{T}\mathbf{y}(t) + \mathbf{q}(t)) \;.$$

Hence, the increment equations become

$$\begin{aligned}
\mathbf{k}_i = -\mathbf{M}^{-1}\mathbf{T}\left(\mathbf{y}_0 + h\sum_{j=1}^{s} a_{i,j}\mathbf{k}_j\right) + \mathbf{M}^{-1}\mathbf{q}(t_0 + c_i h) \\
\Leftrightarrow \quad \mathbf{M}\mathbf{k}_i = -\mathbf{T}\left(\mathbf{y}_0 + h\sum_{j=1}^{s} a_{i,j}\mathbf{k}_j\right) + \mathbf{q}(t_0 + c_i h) \\
\Leftrightarrow \quad \mathbf{M}\mathbf{k}_i + h\sum_{j=1}^{s} a_{i,j}\mathbf{T}\mathbf{k}_j = -\mathbf{T}\mathbf{y}_0 + \mathbf{q}(t_0 + c_i h) \;, \quad i = 1,\dots,s \;.
\end{aligned}$$

This amounts to the linear system of equations

$$\begin{bmatrix} \mathbf{X}_{1,1} & \mathbf{X}_{1,2} & \dots & \dots & \mathbf{X}_{1,2} \\ \mathbf{X}_{2.1} & \mathbf{X}_{2,2} & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ \mathbf{X}_{s,1} & \mathbf{X}_{s,2} & \dots & \dots & \mathbf{X}_{s,s} \end{bmatrix} \begin{bmatrix} \mathbf{k}_1 \\ \mathbf{k}_2 \\ \vdots \\ \vdots \\ \mathbf{k}_s \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \vdots \\ \mathbf{b}_s \end{bmatrix} ,$$

with $\mathbf{X}_{i,j} = \delta_{i,j}\mathbf{M} + h a_{i,j}\mathbf{T}$ , $i,j \in \{1,\dots,s\}$ , $\delta_{i,j} := \begin{cases} 1 & \text{, if } i = j \,, \\ 0 & \text{, if } i \neq j \end{cases}$ ,

$\mathbf{b}_i = -\mathbf{T}\mathbf{y}_0 + \mathbf{q}(t_0 + c_i h) \in \mathbb{R}^N$ , $i \in \{1,\dots,s\}$ .

▲

The following data type has been introduced to store the coefficients of any Runge-Kutta single step method according to [Lecture → Def. 12.3.3.1].

**C++ code 0.3.1: Data type meant for storing coefficients of a Runge-Kutta method**

```cpp
struct RKData {
  RKData() = delete;
  RKData(const RKData &) = default;
  RKData &operator=(const RKData &) = default;
  virtual ~RKData() = default;

  RKData(Eigen::MatrixXd A, Eigen::VectorXd b) : A_(A), b_(b) {
    s_ = A_.rows();
    assertm(A_.cols() == s_, "Butcher matrix must be square");
    assertm(b.size() == s_, "Size mismatch for weight vector");
  }
  unsigned int s_;     // Number of stages
  Eigen::MatrixXd A_;  // Butcher matrix
  Eigen::VectorXd b_;  // Weight vector
};
```

The coefficients $c_i$ are not stored, but assumed to be computed by the formula $c_i = \sum_{j=1}^{s} a_{i,j}$, $i \in \{1,\dots,s\}$.

**(0-3.b)** ⚃ (15 pts.) The C++ function `IRKSSMSolver()` in the file `irkssm.hpp`,

```
template <typename RECORDER = std::function<
                    void(double, const Eigen::VectorXd &, unsigned int)>>
Eigen::VectorXd IRKSSMSolver(
    const RKData &RKSSM_BS,
    std::function<Eigen::VectorXd(const Eigen::VectorXd &)> f_rhs,
    std::function<Eigen::MatrixXd(const Eigen::VectorXd &)> f_Jac,
    const Eigen::VectorXd &y0, double T_final, unsigned int M,
    double newt_reltol, double newt_abstol,
    RECORDER rec = [](double, const Eigen::VectorXd &, unsigned
       int) {});
```

is supposed to implement a numerical integrator for an initial-value problem for the *autonomous* ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ on a finite time interval $[0, T]$. It relies on an implicit Runge-Kutta single step method whose coefficients are passed through the `RKSSM_BS` argument. The other parameters are

- `f_rhs`: a functor object providing $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$,
- `f_Jac`: a functor encoding the Jacobian $\mathsf{D}\,\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^{N,N}$,
- `y0`: the initial state vector $\mathbf{y}_0 \in \mathbb{R}^N$,
- `T_final`: the final time $T > 0$,
- `M`: the number $M \in \mathbb{N}$ of *uniform* timesteps,
- `newt_reltol`, `newt_abstol`: tolerances for the termination of Newton's method,
- `rec`: an object for recording the sequence of states generated by the numerical integrator.

The function returns an approximation of the final state at time $t = T$.

In each time step, a nonlinear system of equations $F(x) = 0$ needs to be solved. The associated lambda function `F()` and `DF()` have not be implemented yet and it is your task to do this.

---

SOLUTION of (0-3.b):

We first have to find out what the lambda functions `F()` and `DF()` are meant for.

**C++ code 0.3.2: Implementation of implicit RK-SSM numerical integrator with fixed timestep**

```
template <typename RECORDER = std::function<
```

```
              void(double, const Eigen::VectorXd &, unsigned int)>>
Eigen::VectorXd IRKSSMSolver(
    const RKData &RKSSM_BS,
    std::function<Eigen::VectorXd(const Eigen::VectorXd &)> f_rhs,
    std::function<Eigen::MatrixXd(const Eigen::VectorXd &)> f_Jac,
    const Eigen::VectorXd &y0, double T_final, unsigned int M,
    double newt_reltol, double newt_abstol,
    RECORDER rec = [](double, const Eigen::VectorXd &, unsigned int) {}) {
  const size_t N = y0.size();    // Dimension of state space
  double t = 0.0;                // Initial time t_0 = 0
  const double h = T_final / M;  // Fixed stepsize
  Eigen::VectorXd y = y0;        // State vector

  auto F = [&RKSSM_BS, &f_rhs, &y, h,
            N](const Eigen::VectorXd &g_vec) -> Eigen::VectorXd {
    Eigen::VectorXd res = g_vec;
#if SOLUTION
    const unsigned int s = RKSSM_BS.s_;
    assertm(static_cast<unsigned int>(g_vec.size()) == s * N,
            "Size mismatch for F arg");
    for (unsigned int k = 0; k < s; ++k) {
      res.segment(k * N, N) -= y;
    }
    for (unsigned int j = 0; j < s; ++j) {
      const Eigen::VectorXd fgj = f_rhs(g_vec.segment(j * N, N));
      for (unsigned int i = 0; i < s; ++i) {
        res.segment(i * N, N) -= h * RKSSM_BS.A_(i, j) * fgj;
      }
    }
#else
    // **************************************************
    // Your code here
    // **************************************************
#endif
    return res;
  };
  auto DF = [&RKSSM_BS, &f_Jac, h,
             N](const Eigen::VectorXd &g_vec) -> Eigen::MatrixXd {
    const unsigned int s = RKSSM_BS.s_;
    Eigen::MatrixXd res(N * s, N * s);
#if SOLUTION
    for (unsigned int j = 0; j < s; ++j) {
      const Eigen::MatrixXd Dfgj = f_Jac(g_vec.segment(j * N, N));
      for (unsigned int i = 0; i < s; ++i) {
        res.block(i * N, j * N, N, N) = -h * RKSSM_BS.A_(i, j) * Dfgj;
      }
      res.block(j * N, j * N, N, N) += Eigen::MatrixXd::Identity(N, N);
    }
#else
    // **************************************************
    // Your code here
    // **************************************************
#endif
    return res;
  };

  const unsigned int s = RKSSM_BS.s_;
  constexpr double l_min = 1.0E-3;
  // Main timestepping loop
  for (unsigned int k = 0; k < M; ++k, t += h) {
```

```
63     double lambda = 1.0;  //
64     Eigen::VectorXd g_vec(N * s);
65     for (unsigned int k = 0; k < s; ++k) {
66       g_vec.segment(k * N, N) = y;
67     }
68     Eigen::VectorXd nc(N * s);     // Newton correction
69     Eigen::VectorXd nct(N * s);    // Simplified Newton correction
70     Eigen::VectorXd g_new(N * s);  // Tentative next iterate
71     double nc_norm;                // Norm of Newton correction
72     double nct_norm;               // Norm of simplified N.c.
73     unsigned int n_ns_loc = 0;     // Number of Newton steps
74     do {
75       auto Df_LU = DF(g_vec).lu();
76       nc = Df_LU.solve(F(g_vec));
77       nc_norm = nc.norm();
78       lambda *= 2;
79       do {
80         lambda /= 2;
81         if (lambda < l_min) {
82           throw std::runtime_error("Damped Newton failed to converge");
83         }
84         g_new = g_vec - lambda * nc;
85         nct = Df_LU.solve(F(g_new));
86         nct_norm = nct.norm();
87         n_ns_loc++;
88       } while (nct_norm > (1 - lambda / 2) * nc_norm);
89       g_vec = g_new;
90       lambda = std::min(2.0 * lambda, 1.0);
91     } while ((nct_norm > newt_reltol * g_vec.norm()) &&
92              (nct_norm > newt_abstol));  //
93     for (unsigned int i = 0; i < s; ++i) {  //
94       y += h * RKSSM_BS.b_[i] * f_rhs(g_vec.segment(i * N, N));
95     }  //
96     rec(t, y, n_ns_loc);
97   }
98   return y;
99 }
```

Comparing with [Lecture → Code 8.5.4.5], we see that Line 63-Line 92 implement a **damped Newton method** for solving the non-linear system of equations `F(x)=0`. It goes without saying that the lambda function `DF` has to provide the **Jacobian** of $F$.

Which non-linear system of equations is faced in this implementation of an implicit Runge-Kutta numerical integrator? We look at the state update done in Line 93-Line 95,

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\sum_{i=1}^{s} b_i \mathbf{f}(\mathbf{g}_i) \ , \quad \mathbf{g}_i := (\vec{\mathbf{g}})_{(i-1)N+1:iN} \in \mathbb{R}^N \ ,$$

where $\vec{\mathbf{g}} \in \mathbb{R}^{sN}$ is the solution of the non-linear system of equations computed by the Newton method. Match this with the update formula from [Lecture → Def. 12.3.3.1],

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\sum_{i=1}^{s} b_i \mathbf{k}_i \quad \text{with increments} \quad \mathbf{k}_i \in \mathbb{R}^N \ ,$$

which reveals $\mathbf{f}(\mathbf{g}_i) = \mathbf{k}_i$. Does this ring a bell? What the damped Newton method computes approximately is the **stages**

$$\mathbf{g}_i = \mathbf{y}_k + h\sum_{j=1}^{s} a_{i,j}\mathbf{k}_j \in \mathbb{R}^N \ , \quad i = 1,\dots,s \ ,$$

already introduced in [Lecture → Rem. 12.3.3.6]. So Newton's method tackles the stage equations

$$\mathbf{g}_i = \mathbf{y}_k + h\sum_{j=1}^{s} a_{i,j}\mathbf{f}(\mathbf{g}_j)\;, \quad i = 1,\dots,s\;. \tag{0.3.3}$$

We rewrite them into a problem of finding the zeros of a function $F : \mathbb{R}^{sN} \to \mathbb{R}^{sN}$.

$$(0.3.3) \quad \Leftrightarrow \quad F\left(\begin{bmatrix}\mathbf{g}_1\\ \vdots \\ \mathbf{g}_s\end{bmatrix}\right) = \mathbf{0} \quad \text{with} \quad F\left(\begin{bmatrix}\mathbf{g}_1\\ \vdots \\ \mathbf{g}_s\end{bmatrix}\right) := \begin{bmatrix}\mathbf{g}_1 - \mathbf{y}_k - h\sum_{j=1}^{s} a_{1,j}\mathbf{f}(\mathbf{g}_j)\\ \vdots \\ \mathbf{g}_s - \mathbf{y}_k - h\sum_{j=1}^{s} a_{s,j}\mathbf{f}(\mathbf{g}_j)\end{bmatrix}, \quad \mathbf{g}_i \in \mathbb{R}^N\;.$$

Assuming that $\mathbf{f}$ is differentiable with derivative $\mathsf{D}\,\mathbf{f}(\mathbf{y}) \in \mathbb{R}^{N,N}$ we find that the Jacobian of $F$ is the $s \times s$ block matrix with blocks of size $N \times N$,

$$\mathsf{D}\,F\left(\begin{bmatrix}\mathbf{g}_1\\ \vdots \\ \mathbf{g}_s\end{bmatrix}\right) = \mathbf{I}_{sN} - h\begin{bmatrix} a_{1,1}\,\mathsf{D}\,\mathbf{f}(\mathbf{g}_1) & a_{1,2}\,\mathsf{D}\,\mathbf{f}(\mathbf{g}_2) & \dots & \dots & a_{1,s}\,\mathsf{D}\,\mathbf{f}(\mathbf{g}_s)\\ a_{2,1}\,\mathsf{D}\,\mathbf{f}(\mathbf{g}_1) & a_{2,2}\,\mathsf{D}\,\mathbf{f}(\mathbf{g}_2) & \dots & \dots & a_{2,s}\,\mathsf{D}\,\mathbf{f}(\mathbf{g}_s)\\ \vdots & & & & \vdots \\ a_{s,1}\,\mathsf{D}\,\mathbf{f}(\mathbf{g}_1) & a_{s,2}\,\mathsf{D}\,\mathbf{f}(\mathbf{g}_2) & \dots & \dots & a_{s,s}\,\mathsf{D}\,\mathbf{f}(\mathbf{g}_s)\end{bmatrix} \tag{0.3.4}$$

This block matrix can be built column by column. You may also use EIGEN's `KroneckerProduct()` function, see EIGEN documentation.

---

▲

**(0-3.c)** (5 pts.) The following C++ function (not implemented!) prints a single line

```
Error ratio = ...
```

to the terminal.

**C++ code 0.3.5: Output ratio or error norms**

```cpp
void test_Output(const RKData &RKSSM) {
  // Smooth right-hand side for autonomous ODE
  auto f_rhs = [](Eigen::VectorXd y) -> Eigen::VectorXd {
    assertm(y.size() == 3, "3D state space!");
    return ((Eigen::VectorXd(3) << (-y[0] - y[1] + std::cos(y[2])),
             (y[0] - y[1] + std::sin(y[2])), 1)
                .finished());
  };
  // Jacobian of right-hand side function
  auto f_Jac = [](Eigen::VectorXd y) -> Eigen::MatrixXd {
    assertm(y.size() == 3, "3D state space!");
    return ((Eigen::MatrixXd(3, 3) << -1.0, -1.0, -std::sin(y[2]), 1.0, -1.0,
             std::cos(y[2]), 0.0, 0.0, 0.0)
                .finished());
  };
  // Initial value
  Eigen::VectorXd y0 = (Eigen::VectorXd(3) << 1.0, 0.0, 0.0).finished();
  // Known exact solution for given initial value
  auto y_exact = [](double t) -> Eigen::Vector3d {
    return Eigen::Vector3d(std::cos(t), std::sin(t), t);
  };
  // Tolerances for Newton's method
  const double reltol = 1.0E-10;
  const double abstol = reltol / 100.0;
  Eigen::VectorXd y_H =
      IRKSSMSolver(RKSSM, f_rhs, f_Jac, y0, 1.0, 100, reltol, abstol);
  Eigen::VectorXd y_h =
      IRKSSMSolver(RKSSM, f_rhs, f_Jac, y0, 1.0, 200, reltol, abstol);
  std::cout << "Error ratio = "
            << (y_H - y_exact(1.0)).norm() / (y_h - y_exact(1.0)).norm()
            << std::endl;
}
```

With *two-digit* precision predict the output for different implicit Runge-Kutta single step methods:

- **Implicit Euler method**:

  `Error ratio =` $\boxed{\phantom{00}}$ ,

- 2-stage **Radau RK-SSM** [Lecture → Ex. 12.3.4.21]:

  `Error ratio =` $\boxed{\phantom{00}}$ ,

- 2-stage **Gauss collocation RK-SSM** [Lecture → Section 12.3.2]:

  `Error ratio =` $\boxed{\phantom{00}}$ .

---

SOLUTION of (0-3.c):

In [Lecture → Section 11.3.2] we have learned that for a single-step method of **order** $p$ when applied to an IVP with smooth solution and run with uniform timestep $h$ over $M$ steps we can expect the following asymptotic behavior of the integration error at final time $T$:

$$\|\mathbf{y}_M - \mathbf{y}(T)\| = O(h^p) \quad \text{for} \quad h \to 0\,. \tag{0.3.6}$$

In the function `test_Output()` we run the same RK-SSM once with $100$ steps and then with $200$ steps, that is, with timestep sizes $H := 1/100$ and $h = 1/200 \Rightarrow h = H/2$. Taking for granted that (0.3.6) is sharp also preasymptotically we conclude

$$\frac{\|\mathbf{y}_{H,100} - \mathbf{y}(1)\|}{\|\mathbf{y}_{h,200} - \mathbf{y}(1)\|} \approx \left(\frac{H}{h}\right)^p = 2^p\,.$$

- **Implicit Euler method** has order $1$

  ▶ `Error ratio =` 2.0 ,

- 2-stage **Radau RK-SSM** has order $3$

  ▶ `Error ratio =` 8.0 ,

- 2-stage **Gauss collocation RK-SSM** has order $4$ [Lecture → Thm. 12.3.2.12]

  ▶ `Error ratio =` 16 .

This is the actual output of the C++ code:

```
Project: Embedded implicit RK-SSM
Test output for implicit Euler
Error ratio = 1.99503
Test output for 2-stage Radau
Error ratio = 7.99362
Test output for 2-stage Gauss collocation method
Error ratio = 16.0051
```

▲

**(0-3.d)** ⚃ (7 pts.) The C++ function

```
double quadByIntegration(double T, unsigned int M);
```

that you can find in the file `irkssm.hpp` calls `IRKSSMSolver()` to approximately evaluate the integral

$$\int_0^T e^{\sin(t)}\,\mathrm{d}t\,.$$

The parameter `T` supplies the integration bound $T > 0$, whereas `M` is the number of uniform timesteps.

You are asked to supply the missing function bodies for the two lambda functions supplying the `f_rhs` and `f_Jac` parameters of `IRKSSMSolver()` and to specify the initial value.

HINT 1 for (0-3.d): The function `make_2StageGauss()` fills an **RKData** object with the coefficients for the 2-stage Gauss collocation Runge-Kutta single-step method. ⌟

HINT 2 for (0-3.d): Notice that `IRKSSMSolver()` can deal with *autonomous* ODEs only. It cannot be applied for numerical quadrature straightforwardly. ⌟

SOLUTION of (0-3.d):

We can do numerical quadrature of the integrand $t \mapsto \varphi(t)$ over $[0, T]$ by solving the scalar initial value problem

$$\dot{y}(t) = \varphi(t) \quad , \quad y(0) = T\,, \tag{0.3.7}$$

until final time $T$ by numerical integration.

Unfortunately, (0.3.7) is *not autonomous* and, therefore, we cannot use `IRKSSMSolver()` directly to tackle it. We have to resort to **autonomization** as explained in [Lecture → § 11.1.3.5] to convert (0.3.7) to an IVP for an autonomous ODE. Autonomization entails switching to a two-dimensional state space and results in the IVP

$$\dot{\mathbf{z}}(t) = \mathbf{f}(\mathbf{z}(t)) := \begin{bmatrix}\varphi(z_2)\\ 1\end{bmatrix}, \quad \mathbf{z}(t) = \begin{bmatrix}z_1(t)\\ z_2(t)\end{bmatrix} = \begin{bmatrix}y(t)\\ t\end{bmatrix} \quad , \quad \mathbf{z}(0) = \mathbf{0}\,.$$

Then we have

$$\int_0^T \varphi(t)\,\mathrm{d}t = z_1(T)\,.$$

The Jacobian of the right-hand side function $\mathbf{f}$ is

$$\mathrm{D}\,\mathbf{f}(\mathbf{z}) = \begin{bmatrix}0 & \varphi'(z_2)\\ 0 & 0\end{bmatrix}.$$

Concretely for $\varphi(t) = e^{\sin(t)}$ this means

$$\mathbf{f}(\mathbf{z}) := \begin{bmatrix} e^{\sin(z_2)} \\ 1 \end{bmatrix} \quad , \quad \mathrm{D}\,\mathbf{f}(\mathbf{z}) = \begin{bmatrix} 0 & e^{\sin(z_2)}\cos(z_2) \\ 0 & 0 \end{bmatrix} , \quad \mathbf{z} \in \mathbb{R}^2 .$$

**C++ code 0.3.8: Using `IRKSSMSolver()` for numerical quadrature**

```cpp
double quadByIntegration(double T, unsigned int M) {
  const Eigen::VectorXd y0
#if SOLUTION
      = Eigen::VectorXd::Zero(2)
#else
  // **************************************************
  // Your code here
  // **************************************************
#endif
      ;
  return IRKSSMSolver(
      // Get coefficients for 2-stage Gauss collocation RK-SSM
      make_2StageGauss(),
      [](const Eigen::VectorXd &y) -> Eigen::VectorXd {
#if SOLUTION
        assertm(y.size() == 2, "2D state space required!");
        return (Eigen::VectorXd(2) << std::exp(std::sin(y[1])), 1).finished();
#else
  // **************************************************
  // Your code here
  // **************************************************
#endif
      },
      [](const Eigen::VectorXd &y) -> Eigen::MatrixXd {
#if SOLUTION
        assertm(y.size() == 2, "2D state space required!");
        return (Eigen::MatrixXd(2, 2) << 0.0,
                std::exp(std::sin(y[1])) * std::cos(y[1]), 0.0, 0.0)
            .finished();
#else
  // **************************************************
  // Your code here
  // **************************************************
#endif
      },
      y0, T, M, 1.0E-8, 1.0E-10)[0];
}
```

▲

**End Problem 0-3** , 32 pts.