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

Final Examination

Autumn Term 2022

Feb 8, 2022, 9:30, ETH HG G1

Don't panic!

Family Name		**Grade**
First Name		
Department		
Legi Nr.		
Date	Feb 8, 2022	

Points:

Prb. No.	1	2	3	Total
max	21	25	25	71
achvd				

(100% = 65 pts. , $\approx$40% (passed) = 26 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.

  Only the contents of solution boxes will be taken into account when grading the exam. Thus, **bring glue and scissors** in case you want to replace one of your solutions. No scrap paper must be handed in along with the exam paper.
- 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.
- Wrong ticks in multiple-choice-type problems may lead to points being deducted.

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. Put all your written solution in the exam envelopes and leave those on the desk along with the computer form.
4. 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.

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-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 $\leq n$),
- and superscript indices in brackets to denote iterates: $\mathbf{x}^{(k)}$, etc.

By default, vectors are regarded as column vectors.

Problem 0-1: Least-Squares Solution of Low-Rank Linear Systems

For any linear system of equations (LSE) a generalized solution can be defined, see [Lecture → Def. 3.1.3.1]. In this problem we study efficient algorithms for computing the generalized solution of an LSE whose system matrix has known low rank.

This problem depends on [Lecture → Section 3.1.3], [Lecture → Section 3.4.1], [Lecture → Section 3.4.2], and, crucially, on [Lecture → Section 3.4.3].

Every matrix $\mathbf{M} \in \mathbb{R}^{m,n}$, $m,n \in \mathbb{N}$, with $\text{rank}(\mathbf{M}) = p$, $0 \leq p \leq \min\{m,n\}$, can be represented as a matrix product $\mathbf{M} = \mathbf{A}\mathbf{B}^\top$ with $\mathbf{A} \in \mathbb{R}^{m,p}$ and $\mathbf{B} \in \mathbb{R}^{n,p}$. This is of particular interest if $p \ll m,n$.

The following data type stores a low-rank matrix in the mentioned factored representation:

```
struct LowRankMatrix {
  LowRankMatrix(const Eigen::MatrixXd &A, const Eigen::MatrixXd &B)
      : A_(A), B_(B) {
    if (A.cols() != B.cols()) throw std::runtime_error("A,B size
       mismatch");
  }
  operator Eigen::MatrixXd() const { return A_ * (B_.transpose()); }
  Eigen::MatrixXd A_;  // First matrix factor
  Eigen::MatrixXd B_;  // Transpose of second matrix factor
};
```

(0-1.a) ⚂ (5 pts.) Based on EIGEN the numerical rank of a matrix can be computed by the following function, see also [Lecture → Code 3.4.2.5]:

C++ code 0.1.1: Computation of numerical rank of a dense matrix in EIGEN

```
unsigned int rank(const Eigen::MatrixXd &A,
                  double tol = std::numeric_limits<double>::epsilon()) {
  if (A.norm() == 0) return 0;
  Eigen::JacobiSVD<Eigen::MatrixXd> svd_object(A);
  const Eigen::VectorXd sv = svd_object.singularValues();
  const unsigned int n = sv.size();
  unsigned int r = 0;
  while ((r < n) && (sv(r) >= sv(0) * A.rows() * A.cols() * tol)) r++;
  return r;
}
```

The overloaded C++ function

```
unsigned int rank(const LowRankMatrix &M,
    double tol = std::numeric_limits<double>::epsilon());
```

is meant to efficiently compute the numerical rank of a matrix stored in a **LowRankMatrix** object using a singular value decomposition, as done in `rank()` from Code 0.1.1 for a generic dense matrix. Fill in the blanks in the following undocumented listing so that the function serves this purpose.

C++ code 0.1.2: Incomplete code for `rank()`

```cpp
unsigned int rank(const LowRankMatrix &M,
                  double tol = std::numeric_limits<double>::epsilon()) {
  unsigned int p = M.A_.cols();

  if ((M.A_.norm() == 0.0) || (M.B_.norm() == 0.0)) return 0;
  Eigen::HouseholderQR<Eigen::MatrixXd> qrA(M.A_);
  Eigen::HouseholderQR<Eigen::MatrixXd> qrB(M.B_);
  const Eigen::MatrixXd RA =

      qrA.matrixQR().block(                    
          ).template triangularView<Eigen::Upper>();
  const Eigen::MatrixXd RB =

      qrB.matrixQR().block(                    
          ).template triangularView<Eigen::Upper>();
  Eigen::JacobiSVD<Eigen::MatrixXd> svd_object(                          );

  const Eigen::VectorXd sv = svd_object.singularValues();
  const unsigned int n = sv.size();
  unsigned int r = 0;
  while ((r < n) && (sv(r) >= sv(0) *                              )) r++;

  return r;
}
```

SOLUTION of (0-1.a):

We first explain how to obtain a singular-value decomposition of $\mathbf{M} = \mathbf{A}\mathbf{B}^\top$, $\mathbf{A} \in \mathbb{R}^{m,p}$, $\mathbf{B} \in \mathbb{R}^{n,p}$, efficiently in the case $p \ll m, n$. The idea is to compute an economical (thin) QR-decomposition [Lecture → Thm. 3.3.3.4] of both $\mathbf{A}$ and $\mathbf{B}$ first:

$$\begin{aligned}
\mathbf{A} &= \mathbf{Q}_A\mathbf{R}_A\,, \quad \mathbf{Q}_A \in \mathbb{R}^{m,p}\,, \quad \mathbf{R}_A \in \mathbb{R}^{p,p}\,, \quad \mathbf{Q}_A^\top\mathbf{Q}_A = \mathbf{I}_p\,,\\
\mathbf{B} &= \mathbf{Q}_B\mathbf{R}_B\,, \quad \mathbf{Q}_B \in \mathbb{R}^{n,p}\,, \quad \mathbf{R}_B \in \mathbb{R}^{p,p}\,, \quad \mathbf{Q}_B^\top\mathbf{Q}_B = \mathbf{I}_p\,.
\end{aligned}$$

Next we compute the singular-value decomposition of the small matrix $\mathbf{R}_A\mathbf{R}_B^\top \in \mathbb{R}^{p,p}$:

$$\mathbf{R}_A\mathbf{R}_B^\top = \mathbf{U}_p\,\mathrm{diag}(\sigma_1,\dots,\sigma_p)\mathbf{V}_p^\top\,, \quad \begin{aligned}\mathbf{U}_p^\top\mathbf{U}_p &= \mathbf{I}_p\,,\\ \mathbf{V}_p^\top\mathbf{V}_p &= \mathbf{I}_p\,,\end{aligned} \quad \sigma_\ell \geq 0\,. \tag{0.1.3}$$

The singular values σ_ℓ are sorted: $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$.

Based on (0.1.3) we obtain a singular-value decomposition $\mathbf{M} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ of $\mathbf{M}$

$$\mathbf{M} = \mathbf{Q}_A\mathbf{R}_A\mathbf{R}_B^\top\mathbf{Q}_B^\top = \underbrace{(\mathbf{Q}_A\mathbf{U}_p)}_{=:\mathbf{U}}\,\underbrace{\mathrm{diag}(\sigma_1,\dots,\sigma_p)}_{=:\mathbf{\Sigma}}\,\underbrace{(\mathbf{Q}_B\mathbf{V}_p)^\top}_{=:\mathbf{V}^\top}\,. \tag{0.1.4}$$

We easily verify

$$\mathbf{U}^\top\mathbf{U} = \mathbf{U}_p^\top\underbrace{\mathbf{Q}_A^\top\mathbf{Q}_A}_{=\mathbf{I}_p}\mathbf{U}_p = \mathbf{U}_p^\top\mathbf{U}_p = \mathbf{I}_p \quad , \quad \mathbf{V}^\top\mathbf{V} = \mathbf{V}_p^\top\underbrace{\mathbf{Q}_B^\top\mathbf{Q}_B}_{=\mathbf{I}_p}\mathbf{V}_p = \mathbf{V}_p^\top\mathbf{V}_p = \mathbf{I}_p\,.$$

The SVD (0.1.4) can be used as replacement for the SVD in Code 0.1.1. In particular, the singular values are available already after the (cheap, $p \ll m, n$!) computation of the SVD of $\mathbf{R}_A \mathbf{R}_B^\top$.

C++ code 0.1.5: Implementation of `rank()` for LowRankMatrix objects

```cpp
unsigned int rank(const LowRankMatrix &M,
                  double tol = std::numeric_limits<double>::epsilon()) {
  // Maximal rank of matrix M
  unsigned int p = M.A_.cols();
  if ((M.A_.norm() == 0.0) || (M.B_.norm() == 0.0)) return 0;
  // Compute singular values efficiently
  // QR decompositions of matrix factors
  Eigen::HouseholderQR<Eigen::MatrixXd> qrA(M.A_);
  Eigen::HouseholderQR<Eigen::MatrixXd> qrB(M.B_);
  // Extract R factors and form their product
  const Eigen::MatrixXd RA =
      qrA.matrixQR().block(0, 0, p, p).template triangularView<Eigen::Upper>();
  const Eigen::MatrixXd RB =
      qrB.matrixQR().block(0, 0, p, p).template triangularView<Eigen::Upper>();
  // Compute SVD of R_A R_B^T, singular values only
  Eigen::JacobiSVD<Eigen::MatrixXd> svd_object(RA * (RB.transpose()));
  const Eigen::VectorXd sv = svd_object.singularValues();
  const unsigned int n = sv.size();
  unsigned int r = 0;
  while ((r < n) && (sv(r) >= sv(0) * M.A_.rows() * M.B_.rows() * tol)) r++;
  return r;
}
```

▲

(0-1.b) ⚃ (13 pts.) [depends on Sub-problem (0-1.a)]

In the file `lowranklsqsol.hpp` provide an *efficient* implementation of the C++ function

```cpp
Eigen::VectorXd lowRankLsqSol(
    const LowRankMatrix &M, const Eigen::VectorXd &b,
    double tol = std::numeric_limits<double>::epsilon());
```

that computes the generalized solution of the linear system of equations $\mathbf{M}\mathbf{x} = \mathbf{b}$ ($\mathbf{M}$ supplied through `M`, $\mathbf{b}$ through `b`) in the sense of

Definition [Lecture → Def. 3.1.3.1]. Generalized solution of a linear system of equations

The generalized solution $\mathbf{x}^\dagger \in \mathbb{R}^n$ of a linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, is defined as

$$\mathbf{x}^\dagger := \operatorname{argmin}\{\|\mathbf{x}\|_2 \colon \mathbf{x} \in \operatorname{lsq}(\mathbf{A}, \mathbf{b})\} \,. \tag{0.1.6}$$

Here $\operatorname{lsq}(\mathbf{A}, \mathbf{b})$ is the set of least-squares solutions of $\mathbf{A}\mathbf{x} = \mathbf{b}$, *cf.* [Lecture → Eq. (3.1.1.2)].

Moreover, the implementation should take the result of `rank(M)` (`rank()` as in Code 0.1.2) as the true rank of $\mathbf{M}$.

HINT 1 for (0-1.b): [Lecture → Code 3.4.3.13] demonstrates the SVD-based computation of a generalized solution of a LSE. ⌟

HINT 2 for (0-1.b): Almost all of Code 0.1.2 can be reused. ⌟

SOLUTION of (0-1.b):

Refer to the solution of Sub-problem (0-1.a) and, in particular, (0.1.4), which gives you the (thin) SVD factors $\mathbf{U}$ and $\mathbf{V}$ of $\mathbf{M}$. Once the SVD is available, [Lecture → Code 3.4.3.13] shows how to compute the generalized solution of $\mathbf{M}\mathbf{x} = \mathbf{b}$.

C++ code 0.1.7: Efficient implementation of `lowRankLsqSol()`

```cpp
Eigen::VectorXd lowRankLsqSol(
    const LowRankMatrix &M, const Eigen::VectorXd &b,
    double tol = std::numeric_limits<double>::epsilon()) {
  assert(M.B_.rows() == b.size());
  // Maximal rank of matrix M
  unsigned int p = M.A_.cols();
  // Economical (!) QR decompositions of matrix factors
  Eigen::HouseholderQR<Eigen::MatrixXd> qrA(M.A_);  //
  Eigen::HouseholderQR<Eigen::MatrixXd> qrB(M.B_);  //
  // Extract R factors and form their product
  const Eigen::MatrixXd RA =  //
      qrA.matrixQR().block(0, 0, p, p).template triangularView<Eigen::Upper>();
  const Eigen::MatrixXd RB =  //
      qrB.matrixQR().block(0, 0, p, p).template triangularView<Eigen::Upper>();
  // Compute SVD of $\mathbf{R}_A\mathbf{R}_B^\top$
  Eigen::JacobiSVD<Eigen::MatrixXd> svd_object(
      RA * (RB.transpose()),
      Eigen::ComputeThinU | Eigen::ComputeThinV);  //
  // Determine numerical rank
  const Eigen::VectorXd sv = svd_object.singularValues();
  const unsigned int n = sv.size();
  unsigned int r = 0;
  while ((r < n) && (sv(r) >= sv(0) * M.A_.rows() * M.B_.rows() * tol)) r++;
  // Compute SVD factors
  const Eigen::MatrixXd U =
      qrA.householderQ() *
    (Eigen::MatrixXd::Identity(M.A_.rows(), p) * svd_object.matrixU());  //

  const Eigen::MatrixXd V =
      qrB.householderQ() *
      (Eigen::MatrixXd::Identity(M.B_.rows(), p) * svd_object.matrixV());  //
  return V.leftCols(r) *
         (sv.head(r).cwiseInverse().asDiagonal() *  //
          (U.leftCols(r).transpose() * b));
}
```

▲

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

The function `lowRankLsqSol()` from Sub-problem (0-1.b) is called with an **LowRankMatrix** object `M`, which represents an $m \times n$-matrix of rank atmost p.

Assuming $p \ll m, n$ small and fixed, what is the asymptotic complexity of your implementation of

`LowRankLsqSol()` for $m, n \to \infty$.

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

SOLUTION of (0-1.c):

The considerations refer to Code 0.1.7.

1. Line 9: Economical QR-decomposition of $\mathbf{A} \in \mathbb{R}^{m,p}$: $\mathrm{cost} = O(m)$ for $m \to \infty$, p fixed.
2. Line 10: Economical QR-decomposition of $\mathbf{B} \in \mathbb{R}^{n,p}$: $\mathrm{cost} = O(n)$ for $n \to \infty$, p fixed
3. Line 12, Line 14: Copying of $p \times p$-matrices: fixed cost $O(1)$
4. Line 19, Computation of the SVD of a $p \times p$-matrix: fixed cost $O(1)$
5. Line 28, extraction of $\mathbf{U}$-factor: application of p Householder reflections (stored in compressed format [Lecture $\to$ Rem. 3.3.3.21]) to a $m \times p$-matrix: $\mathrm{cost} = O(m)$ for $m \to \infty$, p fixed.
6. Line 32, extraction of $\mathbf{V}$-factor: application of p Householder reflections (stored in compressed format [Lecture $\to$ Rem. 3.3.3.21]) to a $n \times p$-matrix: $\mathrm{cost} = O(n)$ for $n \to \infty$, p fixed.
7. Line 34:
   (a) Multiplication of an $r \times m$-matrix with a vector
   (b) Multiplication of an $r \times r$-matrix with a vector
   (c) Multiplication of an $n \times r$-matrix with a vector

   The total cost is $O(m+n)$ for $m, n \to \infty$ and $r \leq p$ fixed.

Summing up the computation efforts we find

$$\mathrm{cost}(\texttt{LowRankLsqSol()}) = O(m+n) \quad \text{for} \quad m, n \to \infty\,.$$

▲

End Problem 0-1 , 21 pts.

Problem 0-2: Energy Minimization for a Discrete System

In physics so-called discrete models with a finite number of state variables are popular, because they often capture key properties of more complicated continuous-state models. A typical task is to determine the so-called ground state of a discrete model characterized as a (local) minimum of some energy functional on state space.

This problem assumes familiarity with [Lecture → Section 8.5].

The state of a discrete physical system is described by $n \in \mathbb{N}$ variables $u_1, \dots, u_n$, which can be collected into a vector $\mathbf{u} = [u_1, \dots, u_n]^\top \in \mathbb{R}^n$. For instance, the u_i may be related to a physical quantity at different points in space. A physicist tells us that the free energy of the system is given by the functional

$$J(\mathbf{u}) := \frac{1}{2h}\Big(\sum_{j=1}^{n}\sum_{k=1}^{n} e^{h|j-k|}(u_j+u_k)^2\Big) + h\sum_{j=1}^{n} e^{u_j}\,, \quad \mathbf{u} \in \mathbb{R}^n\,. \tag{0.2.1}$$

where $h := \frac{1}{n+1}$. What he is interested in are (local) minima of $J : \mathbb{R}^n \to \mathbb{R}$, that is, he wants us to find solutions of the non-linear system of equations $\mathbf{grad}\, J(u) = \mathbf{0}$.

(0-2.a) (5 pts.) For any $\ell \in \{1, \dots, n\}$ and $\mathbf{u} \in \mathbb{R}^n$ compute the partial derivative of the energy functional

$$\frac{\partial J}{\partial u_\ell}(\mathbf{u}) = \boxed{}\,.$$

HINT 1 for (0-2.a): Trivially for any term T depending on two integer indices

$$\sum_{j=1}^{n}\sum_{k=1}^{n} T(j,k) = \sum_{\substack{j=1\\ j\neq\ell}}^{n}\sum_{\substack{k=1\\ k\neq\ell}}^{n} T(j,k) + \sum_{\substack{j=1\\ j\neq\ell}}^{n} T(j,\ell) + \sum_{\substack{k=1\\ k\neq\ell}}^{n} T(\ell,k) + T(\ell,\ell)\,. \tag{0.2.2}$$

┘

SOLUTION of (0-2.a):

I. Fix $\ell \in \{1, \dots, n\}$. Using (0.2.2) we can split the double sum in the definition of $J(\mathbf{u})$ as

$$\sum_{j=1}^{n}\sum_{k=1}^{n} e^{h|j-k|}(u_j+u_k)^2 = \sum_{\substack{j=1\\ j\neq\ell}}^{n}\sum_{\substack{k=1\\ k\neq\ell}}^{n} e^{h|j-k|}(u_j+u_k)^2 + \sum_{\substack{j=1\\ j\neq\ell}}^{n} e^{h|j-\ell|}(u_j+u_\ell)^2 + \sum_{\substack{k=1\\ k\neq\ell}}^{n} e^{h|\ell-k|}(u_\ell+u_k)^2 + 4u_\ell^2\,.$$

Thus the terms of $J(\mathbf{u})$ that directly depend on u_ℓ are

$$J(\mathbf{u}) = \frac{1}{2h}\sum_{\substack{j=1\\ j\neq\ell}}^{n} e^{h|j-\ell|}(u_j+u_\ell)^2 + \frac{1}{2h}\sum_{\substack{k=1\\ k\neq\ell}}^{n} e^{h|\ell-k|}(u_\ell+u_k)^2 + \frac{2}{h}u_\ell^2 + he^{u_\ell} + \dots\,, \quad k = 1, \dots, n\,.$$

Differentiation with respect to u_ℓ is straightforward and yields

$$\frac{\partial J}{\partial u_\ell}(\mathbf{u}) = \frac{1}{h}\sum_{\substack{j=1\\ j\neq\ell}}^{n} e^{h|j-\ell|}(u_\ell+u_j) + \frac{1}{h}\sum_{\substack{k=1\\ k\neq\ell}}^{n} e^{h|\ell-k|}(u_\ell+u_k) + \frac{4}{h}u_\ell + he^{u_\ell}\,,$$

$$\frac{\partial J}{\partial u_\ell}(\mathbf{u}) = \Big(\frac{2}{h}\sum_{\substack{j=1\\j\neq\ell}}^{n} e^{h|j-\ell|}(u_\ell + u_j)\Big) + \frac{4}{h}u_\ell + he^{u_\ell} \quad , \quad k = 1,\dots,n\;. \tag{0.2.3}$$

II. There is another perspective. Applying the binomial formula, we find

$$J(\mathbf{u}) := \frac{1}{2h}\Big(\sum_{j=1}^{n}\sum_{k=1}^{n} e^{h|j-k|}(u_j^2 + 2u_ju_k + u_k^2)\Big) + h\sum_{j=1}^{n} e^{u_j}\;, \quad \mathbf{u}\in\mathbb{R}^n\;. \tag{0.2.4}$$

Next, remember the elementary linear algebra formulas

$$\mathbf{u}^\top\mathbf{M}\mathbf{u} = \sum_{j=1}^{n}\sum_{k=1}^{n}(\mathbf{M})_{j,k}u_ju_k \quad \text{for any matrix} \quad \mathbf{M}\in\mathbb{R}^n\;, \tag{0.2.5}$$

$$\mathbf{u}^\top \operatorname{diag}(d_1,\dots,d_n)\mathbf{u} = \sum_{j=1}^{n} d_ju_j^2 \quad \forall d_j\in\mathbb{R}\;. \tag{0.2.6}$$

Combining (0.2.4) and (0.2.5), (0.2.6) allows to rewrite J as

$$J(\mathbf{u}) = \frac{1}{h}\left(\mathbf{u}^\top \operatorname{diag}\left(\left[\sum_{k=1}^{n} e^{h|j-k|}\right]_{j=1}^{n}\right)\mathbf{u} + \mathbf{u}^\top\left[e^{h|j-k|}\right]_{j,k=1}^{n}\mathbf{u}\right) + h\sum_{j=1}^{n} e^{u_j}\;,$$

$$\blacktriangleright \quad J(\mathbf{u}) = \tfrac{1}{2}\mathbf{u}^\top\mathbf{A}\mathbf{u} + h\sum_{j=1}^{n} e^{u_j}\;, \tag{0.2.7}$$

with a symmetric $n\times n$-matrix $\mathbf{A}$, whose entries are

$$(\mathbf{A})_{k,j} := \frac{2}{h}\cdot\begin{cases} \sum_{m=1}^{n} e^{h|j-m|} + 1 & \text{for } k = j\;,\\ e^{h|j-k|} & \text{for } k\neq j\;,\end{cases} \quad k,j\in\{1,\dots,n\}\;. \tag{0.2.8}$$

From [Lecture → Ex. 8.5.1.19] remember that for any matrix $\mathbf{M}\in\mathbb{R}^{n,n}$

$$\mathbf{grad}\left\{\mathbf{x}\mapsto \tfrac{1}{2}\mathbf{x}^\top\mathbf{M}\mathbf{x}\right\} = \tfrac{1}{2}(\mathbf{M}+\mathbf{M}^\top)\mathbf{x}\;,$$

which yields, when applied to (0.2.7) ($\mathbf{A}^\top = \mathbf{A}$!)

$$\mathbf{grad}\, J(\mathbf{u}) = \mathbf{A}\mathbf{u} + [he^{u_j}]_{j=1}^{n} \in\mathbb{R}^n\;, \quad \mathbf{u}\in\mathbb{R}^n\;. \tag{0.2.9}$$

The partial derivative $\frac{\partial J}{\partial u_\ell}(\mathbf{u})$ is the ℓ-th component of the gradient:

$$\blacktriangleright \quad \frac{\partial J}{\partial u_\ell}(\mathbf{u}) = \left(\mathbf{A}\mathbf{u} + [he^{u_j}]_{j=1}^{n}\right)_\ell = \sum_{j=1}^{n}(\mathbf{A})_{\ell,j}u_j + he^{u_\ell}$$

$$= \frac{2}{h}\Big(\sum_{\substack{j=1\\j\neq\ell}}^{n} e^{h|j-\ell|}u_j\Big) + \frac{2}{h}\Big(\sum_{\substack{j=1\\j\neq\ell}}^{n} e^{h|j-\ell|}\Big)u_\ell + \frac{4}{h}u_\ell + he^{u_\ell}\;.$$

▲

(0-2.b) ⚁ (5 pts.) [depends on Sub-problem (0-2.a)]

The non-linear system of equations $\mathbf{grad}\, J(\mathbf{u}) = \mathbf{0}$ can be written in the form

$$\mathbf{A}\mathbf{u} + \mathbf{b}(\mathbf{u}) = \mathbf{0} \quad \text{with} \quad \mathbf{A} = \mathbf{A}^\top \in \mathbb{R}^{n,n} \quad \mathbf{b} : \mathbb{R}^n \to \mathbb{R}^n \;. \tag{0.2.10}$$

Give explicit formulas for the symmetric matrix $\mathbf{A}$ and the vector-valued function $\mathbf{u} \mapsto \mathbf{b}(\mathbf{u})$.

$$\mathbf{b}(\mathbf{u}) = \boxed{}\;, \mathbf{u} = [u_1, \dots, u_n]^\top \in \mathbb{R}^n,$$

$$(\mathbf{A})_{\ell,j} = \boxed{}\;, \quad \ell, j \in \{1, \dots, n\} \;.$$

SOLUTION of (0-2.b):

We start from the formula for the partial derivatives of J,

$$\begin{array}{rcl} \dfrac{\partial J}{\partial u_\ell}(\mathbf{u}) & = & \frac{2}{h}\Big(\sum\limits_{\substack{j=1\\ j\neq\ell}}^{n} e^{h|j-\ell|}u_j\Big) + \frac{2}{h}\big(\sum\limits_{\substack{j=1\\ j\neq\ell}}^{n} e^{h|j-\ell|}\big)u_\ell + \frac{4}{h}u_\ell \quad + \quad he^{u_\ell} \\ & & \qquad\qquad\qquad \updownarrow \qquad\qquad\qquad\qquad\qquad \updownarrow \\ \mathbf{grad}\, J(\mathbf{u}) & = & \qquad\qquad\qquad \mathbf{A}\mathbf{u} \qquad\qquad\qquad\quad + \;\; \mathbf{b}(\mathbf{u}) \;, \end{array}$$

where we have already hinted at the natural correspondences. First, we conclude that

$$\boxed{\mathbf{b}(\mathbf{u}) = [he^{u_j}]_{j=1}^{n} \in \mathbb{R}^n} \;, \quad \mathbf{u} = [u_1, \dots, u_n]^\top \in \mathbb{R}^n \;. \tag{0.2.11}$$

Secondly, the ℓ-th component of $\mathbf{A}\mathbf{u}$ must be

$$(\mathbf{A}\mathbf{u})_\ell = \frac{2}{h}\Big(\sum_{\substack{j=1\\ j\neq\ell}}^{n} e^{h|j-\ell|}u_j\Big) + \frac{2}{h}\big(\sum_{\substack{j=1\\ j\neq\ell}}^{n} e^{h|j-\ell|}\big)u_\ell + \frac{4}{h}u_\ell \;,$$

which means that

$$\boxed{(\mathbf{A})_{\ell,j} = \begin{cases} \dfrac{2}{h}\sum\limits_{\substack{j=1\\ j\neq\ell}}^{n} e^{h|j-\ell|} + \dfrac{4}{h} & \text{for} \quad j = \ell \;, \\ \dfrac{2}{h}e^{h|j-\ell|} & \text{for} \quad j \neq \ell \;, \end{cases}} \quad \ell, j \in \{1, \dots, n\} \;. \tag{0.2.12}$$

Of course, this is the same result as (0.2.9) and (0.2.8).

▲

A popular approach to the iterative solution of the non-linear system of equations $\mathbf{A}\mathbf{u} + \mathbf{b}(\mathbf{u}) = \mathbf{0}$ from (0.2.10) is the **fixed-point iteration** with the recursive definition

$$\mathbf{u}^{(k+1)} = -\mathbf{A}^{-1}\mathbf{b}(\mathbf{u}^{(k)}) \,, \quad k \in \mathbb{N}_0 \,. \tag{0.2.13}$$

(0-2.c) ⚁ (10 pts.) [depends on Sub-problem (0-2.b)]

Implement an *efficient* C++ function

```
Eigen::VectorXd groundState_fpit(unsigned int n,
  double rtol = 1.0E-8, double atol = 1.0E-10,
  unsigned int maxit = 20);
```

that employs the fixed-point iteration (0.2.13) to solve $\mathbf{grad}\, J(\mathbf{u}) = \mathbf{0}$. The argument `n` passes n. As initial guess use $\mathbf{u}^{(0)} := \mathbf{0}$.

Employ a correction based termination criterion with relative tolerance `rtol` and absolute tolerance `atol`. Stop after at most `maxit` iteration steps.

SOLUTION of (0-2.c):

For the sake of efficiency we have to initialize according to (0.2.12) and LU-decompose the dense matrix $\mathbf{A}$ outside the main iteration loop, whereas the $\mathbf{u}$-dependent vector $\mathbf{b}(\mathbf{u})$ has to be set anew according to (0.2.11) in every step of the iteration.

C++ code 0.2.14: Efficient implementation of `groundState_fpit()`

```
Eigen::VectorXd groundState_fpit(unsigned int n, double rtol = 1.0E-8,
                                 double atol = 1.0E-10,
                                 unsigned int maxit = 20) {
  const double h = 1.0 / (1.0 + n);
  // Vector for storing the iterates
  Eigen::VectorXd u(n);
  // Initialize the matrix A
  Eigen::MatrixXd A(n, n);
  // Abuse the vector u as temporary storage to avoid an excessive
  // number of calls to the exponential function.
  for (unsigned int j = 0; j < n; ++j) {
    u[j] = 2.0 / h * std::exp(h * j);
  }
  for (unsigned int l = 0; l < n; ++l) {
    A(l, l) = 4 / h;
    for (unsigned int j = 0; j < l; ++j) {
      A(l, l) += u[l - j];
    }
    for (unsigned int j = l + 1; j < n; ++j) {
      A(l, l) += u[j - l];
    }
    for (unsigned int j = 0; j < l; ++j) {
      A(l, j) = A(j, l) = u[l - j];
    }
  }
  // Precompute LU factorization of the matrix A, see
  // [Lecture → Code 2.5.0.12]
  auto A_ludec = A.lu();
  // Initial guess zero
  u.setZero();
  // Main loop for fixed-point iteration
  unsigned int it_cnt = 0;  // iteration count
  double corr_norm;          // Norm of correction
  Eigen::VectorXd b(n);      // Stores the vector b(u)
  do {
    // Initialize vector b(u)
    for (unsigned int j = 0; j < n; ++j) {
      b[j] = -h * std::exp(u[j]);
    }
    // Backward and forward substitution
    Eigen::VectorXd u_new = A_ludec.solve(b); //
    corr_norm = (u - u_new).norm();
    // Next iterate
    u = u_new;
  } while ((corr_norm > rtol * u.norm()) && (corr_norm > atol) &&
           (++it_cnt < maxit));
  return u;
}
```

Important is the precomputation of the LU-factorization of the dense matrix $\mathbf{A}$ prior to entering the iteration loop, *cf.* [Lecture $\to$ Rem. 2.5.0.10].

▲

(0-2.d) ⚄ (5 pts.) [depends on Sub-problem (0-2.c)]

We make the following assumption

Assumption 0.2.15. Linear convergence of fixed-point iteration

The fixed point iteration (0.2.13) *converges linearly* [Lecture $\to$ Def. 8.2.2.1] with a rate $L \leq 1 - \frac{c}{n} < 1$ with some constant $c > 0$.

For very large $n \gg 1$ what is the worst-case n-asymptotics of the *additional* computational effort your implementation of `groundState_fpit()` requires in order to reduce the iteration error by a factor of $\rho > 0$?

$$\text{Additional computational effort} = O\left(\boxed{}\right) \quad \text{for} \quad n \to \infty\,.$$

HINT 1 for (0-2.d): For $\delta \approx 0$ in your derivation you may replace $\log(1+\delta)$ with δ. ⌟

SOLUTION of (0-2.d):

❶ Write $\epsilon_k := \left\|\mathbf{u}^{(k)} - \mathbf{u}^*\right\|$, $\mathbf{u}^* := \lim\limits_{k\to\infty} u^{(k)}$, for the iteration error after k steps. For an iteration that converges linearly with rate $L \leq 1 - \frac{c}{n} < 1$ in each step the norm of the iteration error is multiplied with L: $\epsilon_{k+1} \leq L\epsilon_k$. Hence in order to reduce the norm of the iteration error by a factor of $\rho > 1$ we need at most K additional steps, with K given by

$$L^K = \rho^{-1} \quad \Leftrightarrow \quad K = -\frac{\log\rho}{\log L} = -\frac{\log\rho}{\log(1-\frac{c}{n})}\,.$$

Since $n \gg 1$, we have $\frac{c}{n} \ll 1$ so that we can linearize the logarithm around 1: $\log(1-\frac{c}{n}) \approx -\frac{c}{n}$. Plugging this into the formula for K yields for $n \gg 1$

$$K = n\frac{\log\rho}{c} = O(n) \quad \text{for} \quad n \to \infty$$

Hence the number of additional iteration steps will grow linearly with n.

❷ According to [Lecture → Thm. 2.5.0.3] the computational effort for a single iteration step in Code 0.2.14 is $O(n^2)$, dominated by the forward and backward elimination involved in the `solve()` member function of an EIGEN object representing an LU-decomposed matrix.

Hence the total additional computational effort for the reduction of the norm of the iteration error by a fixed factor is $O(n^3)$.

▲

End Problem 0-2, 25 pts.

Problem 0-3: ROW Method for Simulating the van der Pol Oscillator

The so-called van der Pol oscillator is a well-known model for a stable oscillator with non-linear damping, see Wikipedia page. In this problem we study how to apply a special semi-implicit single-step methods to simulate it.

This problem is related to [Lecture → Section 12.4] and[Lecture → Section 12.1].

The van der Pol equation is the scalar second-order ODE

$$\ddot{y} - \mu(1-y^2)\dot{y} + y = 0\,, \quad \mu > 0\,. \tag{0.3.1}$$

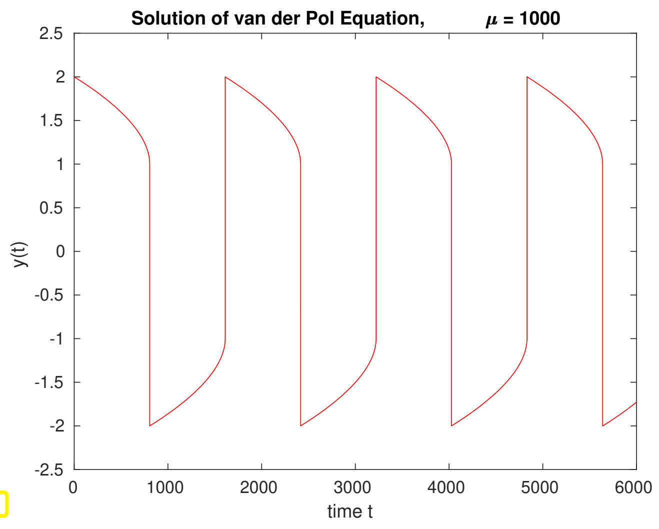

Fig. 1

◁ Plot of the solution for the special initial-value problem for (0.3.1):

- $\mu = 1000$,
- $t_0 = 0$,
- $y(0) = 2$, $\dot{y}(0) = 0$

For $t \to \infty$ the solution will be periodic.

(0-3.a) ⚄ (10 pts.) Give rigorous arguments, why the initial-value problem

$$\ddot{y} - \mu(1-y^2)\dot{y} + y = 0\,, \quad \mu = 10^4\,, \quad y(0) = 2\,,\ \dot{y}(0) = 0\,, \tag{0.3.2}$$

is stiff, at least in the beginning of a simulation.

SOLUTION of (0-3.a):

Stiffness was discussed for autonomous first-order initial value problems $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$ in [Lecture → Section 12.2].

How to distinguish stiff initial value problems

An initial value problem for an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ will probably be stiff, if, for substantial periods of time,

$$\min\{\operatorname{Re}\lambda : \lambda \in \sigma(\mathrm{D}\,\mathbf{f}(\mathbf{y}(t)))\} \ll 0\,, \tag{0.3.4}$$

and

$$\max\{\operatorname{Re}\lambda : \lambda \in \sigma(\mathrm{D}\,\mathbf{f}(\mathbf{y}(t)))\} \lesssim 0\,, \tag{0.3.5}$$

where $t \mapsto \mathbf{y}(t)$ is the solution trajectory and $\sigma(\mathbf{M})$ is the spectrum of the matrix $\mathbf{M}$, see [Lecture → Def. 9.1.0.1].

In order to verify these criteria we first have to transform the van der Pol equation (0.3.1) into first-order form by introducing $v := \dot{y}$ as a new state variable:

$$\ddot{y} - \mu(1-y^2)\dot{y} + y = 0 \quad \blacktriangleright \quad \left\{ \begin{array}{rcl} \dot{y} & = & v\,, \\ \dot{v} & = & \mu(1-y^2)v - y\,. \end{array} \right. \tag{0.3.6}$$

Lumping the state variables into the state vector $\mathbf{z} := \left[\begin{smallmatrix} y \\ v \end{smallmatrix}\right]$, we end up with the first-order ODE in standard form

$$\dot{\mathbf{z}} = \begin{bmatrix} \dot{z}_1 \\ \dot{z}_2 \end{bmatrix} = \mathbf{f}(\mathbf{z}) \quad \text{with} \quad \mathbf{f}(\mathbf{z}) = \begin{bmatrix} z_2 \\ \mu(1-z_1^2)z_2 - z_1 \end{bmatrix}, \tag{0.3.7}$$

We have to investigate the spectrum of the Jacobian $\mathrm{D}\,\mathbf{f}(\mathbf{z})$ for $\mathbf{z} = \left[\begin{smallmatrix} 2 \\ 0 \end{smallmatrix}\right]$, which is the initial condition. We find

$$\mathrm{D}\,\mathbf{f}(\mathbf{z}) = \begin{bmatrix} 0 & 1 \\ -2\mu z_1 z_2 - 1 & \mu(1-z_1^2) \end{bmatrix} \quad \blacktriangleright \quad \mathrm{D}\,\mathbf{f}(\left[\begin{smallmatrix} 2 \\ 0 \end{smallmatrix}\right]) = \begin{bmatrix} 0 & 1 \\ -1 & -3\mu \end{bmatrix}. \tag{0.3.8}$$

The characteristic polynomial of this matrix and its roots are

$$\chi(t) = t^2 + 3\mu t + 1 \quad \blacktriangleright \quad t_{\pm} = \tfrac{1}{2}\left(-3\mu \pm \sqrt{9\mu^2 - 4}\right).$$

For $\mu \gg 1$, the spectrum is approximately

$$\sigma(\mathrm{D}\,\mathbf{f}(\left[\begin{smallmatrix} 2 \\ 0 \end{smallmatrix}\right])) \approx \{-3\mu, 0\},$$

and the stiffness criteria are satisfied for states close to $\left[\begin{smallmatrix} 2 \\ 0 \end{smallmatrix}\right]$.

Alternatively, you may simply compute the eigenvalues of $\mathrm{D}\,\mathbf{f}(\left[\begin{smallmatrix} 2 \\ 0 \end{smallmatrix}\right])$ numerically.

C++ code 0.3.9: Computing the eigenvalues of the van der Pol Jacobian

```cpp
Eigen::Vector2d vanDerPolJacobianEVs(Eigen::Vector2d z, double mu) {
  Eigen::Matrix2d J(2, 2);
  J << 0, 1, -2 * mu * z[0] * z[1] - 1.0, mu * (1 - z[0] * z[0]);
  Eigen::EigenSolver<Eigen::Matrix2d> eig(J);
  auto ev = eig.eigenvalues().real();
  return ev.head(2);
}
```

C++ code 0.3.10: Calling `vanDerPolJacobianEVs()`

```cpp
struct ROW {
  explicit ROW(unsigned int s)
      : s_(s),
        alpha_(Eigen::MatrixXd::Zero(s, s)),
        gamma_(Eigen::MatrixXd::Zero(s, s)),
        b_(Eigen::VectorXd::Zero(s)) {}
  unsigned int s_;          // Number of stages
  Eigen::MatrixXd alpha_;   // Coefficients α_{i,j}
  double d_gamma_;          // Diagonal of γ-matrix
  Eigen::MatrixXd gamma_;   // Coefficients γ_{i,j}
  Eigen::VectorXd b_;       // Weights b_i
};
```

Remark. The initial value problem behind Fig. 1 is stiff almost all the time during the simulation, as can be seen from a plot of the two eigenvalues of $t \mapsto \mathrm{D}\,\mathbf{f}(\mathbf{y}(t))$. ▷

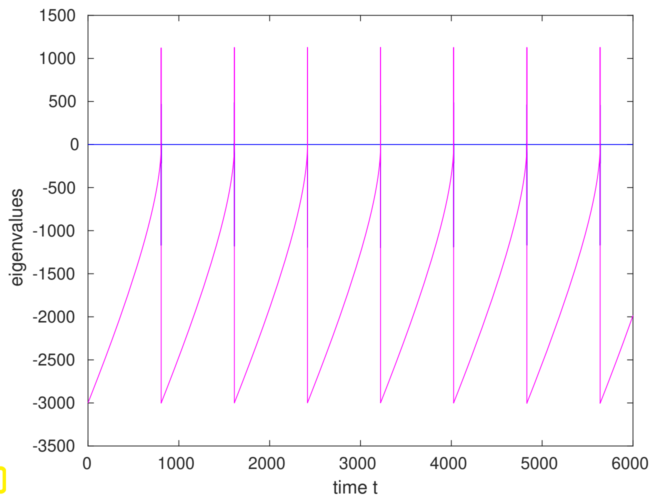

Fig. 2

▲

In the research article [**RAN15**] new so-called **Rosenbrock-Wanner (ROW)** semi-implicit single-step methods for the numerical integration of general first-order ODEs $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$, $\mathbf{f} : I \times D \to \mathbb{R}^N$, $I \subset \mathbb{R}$, $D \subset \mathbb{R}^N$, are presented. The author introduces a single step of an s-stage ROW method with timestep size $h > 0$ as follows:

$$\mathbf{k}_i = \mathbf{f}(t_0 + \alpha_i h, \mathbf{z}_i) + h\mathbf{J} \sum_{j=1}^{i} \gamma_{i,j}\mathbf{k}_j + h\gamma_i \frac{\partial \mathbf{f}}{\partial t}(t_0, \mathbf{y}_0) , \quad i = 1, \dots, s , \tag{0.3.11a}$$

$$\mathbf{z}_i = \mathbf{y}_0 + h \sum_{j=1}^{i-1} \alpha_{i,j}\mathbf{k}_j , \quad i = 1, \dots, s \tag{0.3.11b}$$

$$\mathbf{y}_1 = \mathbf{y}_0 + h \sum_{i=1}^{s} b_i \mathbf{k}_i , \tag{0.3.11c}$$

where $\mathbf{J} := \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_0, \mathbf{y}_0) \in \mathbb{R}^{N,N}$ is a Jacobian, and $\alpha_i, \gamma_i, \alpha_{i,j}, \gamma_{i,j}, b_i \in \mathbb{R}$ are suitably chosen parameters, which must satisfy

$$\alpha_i = \sum_{j=1}^{i-1} \alpha_{i,j} \quad , \quad \gamma_i = \sum_{j=1}^{i-1} \gamma_{i,j} , \quad i = 1, \dots, s .$$

Moreover, all $\gamma_{i,i}$, $i = 1, \dots, s$, are equal and positive: $\gamma_{i,i} = \gamma > 0$. The coefficients are stored in a special data structure based in EIGEN's datatype, explanations in the comments.

```
struct ROW {
  explicit ROW(unsigned int s)
  : s_(s),
  alpha_(Eigen::MatrixXd::Zero(s, s)),
  gamma_(Eigen::MatrixXd::Zero(s, s)),
  b_(Eigen::VectorXd::Zero(s)) {}
  unsigned int s_;          // Number of stages
  Eigen::MatrixXd alpha_;   // Coefficients α_{i,j}, i,j = 1,...,s
  double d_gamma_;          // Value γ > 0 for γ_{i,i}, i = 1,...,s
  Eigen::MatrixXd gamma_;   // Coefficients γ_{i,j}, i,j = 1,...,s, j < i
  Eigen::VectorXd b_;       // Weights b_i, i = 1,...,s
};
```

From [**RAN15**] we get appropriate values for the coefficients for the 3-stage ROW method called ROSxPR. Note that the $\alpha_{i,j}$ and $\gamma_{i,j}$ are needed only for $i > j$.

C++ code 0.3.12: Initialization of coefficient for 3-stage ROW method ROSxPR

```
ROW init_ROSxPR(void) {
  ROW row(3);                                  // s = 3
  row.alpha_(1, 0) = 2.3660254037844388;       // α_{2,1}
  row.alpha_(2, 0) = 0.0;                      // α_{3,1}
  row.alpha_(2, 1) = 1.0;                      // α_{3,2}
  row.gamma_(1, 0) = -2.3660254037844388;      // γ_{2,1}
  row.gamma_(2, 0) = -0.28468642516567449;     // γ_{3,1}
  row.gamma_(2, 1) = -1.0813389786187642;      // γ_{3,2}
  row.d_gamma_ = 0.78867513459481287;   // γ_{1,1} = γ_{2,2} = γ_{3,3}
  row.b_[0] = 0.29266384402395124;      // b_1
  row.b_[1] = -0.081338978618764143;    // b_2
  row.b_[2] = 0.78867513459481287;      // b_3
  return row;
}
```

(0-3.b) ⚄ (5 pts.) The templated C++ function

```
template <typename STATETYPE, typename RHSTYPE, typename JACTYPE>
std::vector<STATETYPE>
odeROW(RHSTYPE &&rhs, JACTYPE &&Jacobian, const STATETYPE &y0,
double T, unsigned int M, const ROW &row);
```

is supposed to realize `M` uniform steps of the ROW method specified in `row` for the *autonomous* IVP

$$\dot{\mathbf{y}} = \mathtt{rhs}(\mathbf{y}) \quad , \quad \mathbf{y}(0) = \mathtt{y0} \ .$$

until final time `T`. The argument `Jacobian` should contain a functor providing the Jacobi matrix $\mathbf{y} \mapsto \mathsf{D}\,\mathtt{rhs}(\mathbf{y})$. The following requirements are imposed on the template argument types:

- **STATETYPE** should be compatible with a vector type of EIGEN,
- **RHSTYPE** must be `std::function<STATETYPE(const STATETYPE &)>`,
- **JACTYPE** must be a functor taking a **STATETYPE** argument and returning an object with the capabilities of an EIGEN matrix.

Fill in the blank in the following listing so that it properly implements `odeROW()`.

C++ code 0.3.13: Incomplete code for `odeROW()`

```
template <typename STATETYPE, typename RHSTYPE, typename JACTYPE>
std::vector<STATETYPE> odeROW(RHSTYPE &&rhs, JACTYPE &&Jacobian,
const STATETYPE &y0, double T, unsigned int M,
const ROW &row) {
  using MATTYPE = decltype(Jacobian(y0)); // Type for Jacobi matrix
  unsigned int N = y0.size();
  std::vector<STATETYPE> y(M + 1); // Vector of computed states
  y[0] = y0;
  std::vector<STATETYPE> k(row.s_);  // Increments k_i
  std::vector<STATETYPE> z(row.s_);   // Auxiliary states z_i
  const double h = T / M;  // timestep size h
  for (unsigned int m = 0; m < M; ++m) {   // Main timestepping loop
    const MATTYPE J = Jacobian(y[m]);
    const MATTYPE IgJ = MATTYPE::Identity(N, N) - h *                 ;

    auto IgJ_lu = IgJ.lu();
    z[0] = y[m];
    k[0] = IgJ_lu.solve(                                  );
    y[m + 1] =                                          ;
    for (unsigned int i = 1; i < row.s_; ++i) {
      z[i] = y[m];
      for (unsigned int j = 0; j < i; ++j) {
        z[i] += h * row.                      * k[          ]; }
      STATETYPE tmp = row.gamma_(i, 0) * k[          ];
      for (unsigned int j = 1; j < i; ++j) {
        tmp +=                                  ; }

      k[i] = IgJ_lu.solve(                                         );
      y[m + 1] +=                                          ;
    }
  }
  return y;
}
```

HINT 1 for (0-3.b): Rewrite (0.3.11a) as a linear system of equations for the increment $\mathbf{k}_i$. ⌟

SOLUTION of (0-3.b):

Note that, using the fact that $\gamma = \gamma_{i,i}$ for all $i = 1, \dots, s$, we can rewrite the equations defining the ROW method

$$\begin{aligned}
(\mathbf{I} - h\gamma\mathbf{J})\mathbf{k}_i &= \mathbf{f}(t_0 + \alpha_i h, \mathbf{z}_i) + h\mathbf{J}\sum_{j=1}^{i-1}\gamma_{i,j}\mathbf{k}_j + h\gamma_i\frac{\partial \mathbf{f}}{\partial t}(t_0, \mathbf{y}_0)\;, \quad i = 1,\dots,s\;,\\
\mathbf{z}_i &= \mathbf{y}_0 + h\sum_{j=1}^{i-1}\alpha_{i,j}\mathbf{k}_j\;, \quad i = 1,\dots,s\\
\mathbf{y}_1 &= \mathbf{y}_0 + h\sum_{i=1}^{s} b_i\mathbf{k}_i\;,
\end{aligned}$$

So all the increments can be computed by solving an $N \times N$ linear system of equations with the same system matrix $\mathbf{I} - h\gamma\mathbf{J} \in \mathbb{R}^{N,N}$. Of course, the LU-decomposition of that matrix should be precomputed for the sake of efficiency.

C++ code 0.3.14: Implementation of `odeROW()`

```cpp
template <typename STATETYPE, typename RHSTYPE, typename JACTYPE>
std::vector<STATETYPE> odeROW(RHSTYPE &&rhs, JACTYPE &&Jacobian,
                              const STATETYPE &y0, double T, unsigned int M,
                              const ROW &row) {
  using MATTYPE = decltype(Jacobian(y0)); // Type for Jacobi matrix
  unsigned int N = y0.size();            // Dimension of state space
  std::vector<STATETYPE> y(M + 1);       // Vector of computed states
  y[0] = y0;
  std::vector<STATETYPE> k(row.s_);  // Increments k_i
  std::vector<STATETYPE> z(row.s_);  // Auxiliary states z_i
  const double h = T / M;            // timestep size
  // Main timestepping loop, y[m] contains the current iterate
  for (unsigned int m = 0; m < M; ++m) {
    const MATTYPE J = Jacobian(y[m]);
    const MATTYPE IgJ = MATTYPE::Identity(N, N) - h * row.d_gamma_ * J;
    auto IgJ_lu = IgJ.lu(); // Precompte LU decomposition
    z[0] = y[m];
    k[0] = IgJ_lu.solve(rhs(z[0]));            // Just fwd./bkw. substitution
    y[m + 1] = y[m] + h * row.b_[0] * k[0];    // Next state
    for (unsigned int i = 1; i < row.s_; ++i) {
      // Compute auxiliary states
      z[i] = y[m];
      for (unsigned int j = 0; j < i; ++j) {
        z[i] += h * row.alpha_(i, j) * k[j];
      }
      // Compute increments
      STATETYPE tmp = row.gamma_(i, 0) * k[0];
      for (unsigned int j = 1; j < i; ++j) {
        tmp += row.gamma_(i, j) * k[j];
      }
      k[i] =
          IgJ_lu.solve(rhs(z[i]) + h * J * tmp); // Just fwd./bkw.
             substitution
      // Update next state
      y[m + 1] += h * row.b_[i] * k[i];
    }
  }
  return y;
}
```

▲

(0-3.c) ⚁ (5 pts.) [depends on Sub-problem (0-3.b)]

In the file `vanderpol.hpp` complete the code of the C++ function

```
std::vector<double> solveROWVanDerPol(double mu, double T,
    unsigned int M);
```

that solves an IVP for the van der Pol equation (0.3.1) with initial data $y(0) = 2$, $\dot{y}(0) = 0$, using the ROSxPR method provided by `init_ROSxPR()` in Code 0.3.12. The arguments `mu`, `T`, and `M` pass the model parameter $\mu > 0$, the final time $T > 0$, and the number of uniform timesteps. Of course, your implementation can call the templated function `odeROW()`.

SOLUTION of (0-3.c):

The numerical integrator `odeROW` can only deal with first-order autonomous ODEs. Therefore we have to convert the second-order van der Pol equation to an equivalent first-order system, see (0.3.6) and (0.3.7):

$$\dot{\mathbf{z}} = \begin{bmatrix} \dot{z}_1 \\ \dot{z}_2 \end{bmatrix} = \mathbf{f}(\mathbf{z}) \quad \text{with} \quad \mathbf{f}(\mathbf{z}) = \begin{bmatrix} z_2 \\ \mu(1 - z_1^2)z_2 - z_1 \end{bmatrix} . \tag{0.3.7}$$

We also have to provide the Jacobian

$$\mathsf{D}\,\mathbf{f}(\mathbf{z}) = \begin{bmatrix} 0 & 1 \\ -2\mu z_1 z_2 - 1 & \mu(1 - z_1^2) \end{bmatrix} . \tag{0.3.8}$$

Both $\mathbf{f} : \mathbb{R}^2 \to \mathbb{R}^2$ and $\mathsf{D}\,\mathbf{f} : \mathbb{R}^2 \to \mathbb{R}^{2,2}$ can be incarnated as lambda functions. The states $\in \mathbb{R}^2$ can be represented by the type **Eigen::Vector2d**, the Jacobian as **Eigen::Matrix2d**.

C++ code 0.3.15: Implementation of `solveROWVanDerPol()`

```cpp
std::vector<double> solveROWVanDerPol(double mu, double T, unsigned int M) {
  // Initialize ROW method
  const ROW row = init_ROSxPR();
  // Define autonomous initial-value problem
  auto f = [mu](Eigen::Vector2d z) -> Eigen::Vector2d {
    return Eigen::Vector2d(z[1], mu * (1.0 - z[0] * z[0]) * z[1] - z[0]);
  };
  auto Df = [mu](Eigen::Vector2d z) -> Eigen::Matrix2d {
    Eigen::Matrix2d J;
    J << 0, 1, -2 * mu * z[0] * z[1] - 1.0, mu * (1 - z[0] * z[0]);
    return J;
  };
  Eigen::Vector2d y0(2.0, 0.0);  // Initial state
  // Launch timestepping
  auto z_vec = odeROW(f, Df, y0, T, M, row);
  // The first component of the state vectors provides the y state
  std::vector<double> y_vec(z_vec.size());
  for (unsigned int k = 0; k < y_vec.size(); ++k) {
    y_vec[k] = z_vec[k][0];
  }
  return y_vec;
}
```

▲

(0-3.d) (5 pts.) Use the C++ function `solveROWVanDerPol()` implemented in Sub-problem (0-3.c) to create (in the file `vanderpol.hpp`) a C++ function

```
void testCvgROWVanDerPol(void);
```

that prints to `stdout` a *table* of values, from which one can immediately see the rate of algebraic convergence of the error at final time of the ROW method applied to solve the IVP for the van der Pol equation detailed in Sub-problem (0-3.c). Use the parameter value $\mu = 1$ and final time $T = 6$.

The function `testCvgROWVanDerPol()` is called from the `main()` program. Run the code and conclude the order of the ROW method ROSxPR from the output.

order of ROSxPR $= \boxed{}$.

HINT 1 for (0-3.d): You are advised to use the provided implementation of `solveROWVanDerPol` available as `solveROWVanDerPol_ref()`. ┘

SOLUTION of (0-3.d):

Meaningful tabular output to deduce algebraic convergence of a timestepping method should follow these rules:

- The timestepping scheme is run with different numbers of uniform timesteps increasing in geometric progression:

$$M_{\ell+1} = \rho \cdot M_\ell \quad \text{for} \quad \rho \in \mathbb{N}\,, \quad \rho > 1 \quad .$$

  Most conveniently, one chooses $\rho = 2$, but other factors are possible as well.

- The ratio of the error norms recorded for two successive runs with numbers M_ℓ and $M_{\ell+1}$ of timesteps are displayed.

Writing ϵ_ℓ for the error norm observed for M_ℓ timesteps, a timestepping scheme of order $p \in \mathbb{N}$ will yield

$$\epsilon_\ell \approx C M_\ell^{-p}\,,$$

which means, under the assumption of sharpness of the estimate,

$$\frac{\epsilon_{\ell+1}}{\epsilon_\ell} \approx \frac{M_{\ell+1}^p}{M_\ell^p} \approx \rho^p \in \mathbb{N}\,.$$

This cardinal number ρ^p is easily discerned in a table.

A table output might look like this

```
M= 128: |yM-y(T)| = 0.00146655
M= 256: |yM-y(T)| = 0.000158326, ratio = 9.26288
M= 512: |yM-y(T)| = 1.79114e-05, ratio = 8.83941
M= 1024: |yM-y(T)| = 2.10996e-06, ratio = 8.48898
M= 2048: |yM-y(T)| = 2.55302e-07, ratio = 8.26457
M= 4096: |yM-y(T)| = 3.13726e-08, ratio = 8.13772
M= 8192: |yM-y(T)| = 3.8873e-09, ratio = 8.07053
M= 16384: |yM-y(T)| = 4.83667e-10, ratio = 8.03714
```

We see that doubling the number of timesteps $h \to h/2$ leads to a reduction of the error by a factor $\approx 8 = 2^3$. This is empiric evidence that the ROSxPR method is of order 3 .

C++ code 0.3.16: Function `testCvgROWVanDerPol()` for empiric study of convergence

```
void testCvgROWVanDerPol(void) {
  unsigned int M_min = 128;   // Minimal number of timesteps
  unsigned int no_runs = 8;   // Number of times the timestep is doubled
  unsigned int fac_ref = 10;  // That many more timesteps for reference
  // Vectors for storing final approximate states
  std::vector<double> yT;
  unsigned int M = M_min;
  for (unsigned int l = 0; l < no_runs; ++l, M *= 2) {
    yT.push_back(solveROWVanDerPol(1.0, 6.0, M).back());
  }
  // Finally run solver with very many timesteps to obtain reference
     solution.
  const double y_ref = solveROWVanDerPol(1.0, 6.0, fac_ref * M).back();
  // Tabulate the absolute values of the errors at final time
  M = M_min;
  double y_errp;
  for (unsigned int l = 0; l < yT.size(); ++l, M *= 2) {
    const double y_errn = std::abs(y_ref - yT[l]);
    std::cout << "M= " << M << ": |yM-y(T)| = " << y_errn;
    if (l > 0) {
      std::cout << ", ratio = " << y_errp / y_errn;
    }
    std::cout << std::endl;
    y_errp = y_errn;
  }
}
```

▲

End Problem 0-3 , 25 pts.