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

Final Examination

Autumn Term 2020

Tuesday, February 2, 2021, 9:30 - 13:30, ONA E7, ONA E25, HG G1, HG G26.1

Don't panic!

		Grade
Family Name		
First Name		
Department		
Legi Nr.		
Date	Tuesday, February 2, 2021	

Points:

Prb. No.	1	2	3	Total
max	24	27	40	91
achvd				

(100% = 91 pts. , ≈40% (passed) = 35 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 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.
- 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.
- 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.

Special Covid-19 Safety Measures

- Protective masks covering nose and mouth have to be worn all the time.
- Assistants are not supposed to get close to you. So you cannot expect any assistance with computer problems during the exam.
- In case your computer crashes irreversibly, raise your hand, and then you will be guided to one of the designated spare machines.
- Since the exam is distributed over several locations, no questions will be answered and you are not supposed to say anything during the exam.

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. Do not write or type before the actual start of the exam is announced.
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.
- Code that has been commented out will not be considered as part of your solution.
- 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: Smooth integrand by transformation

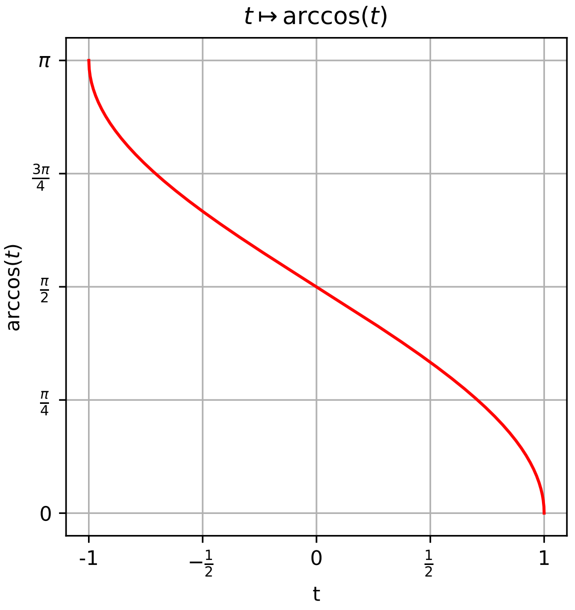
Integrals with non-smooth integrands can *sometimes* be treated by a suitable transformation, which converts the integrand into a smooth function, see [Lecture → Rem. 7.4.3.10]. In this problem we study an example.

You should be familiar with the transformation of quadrature rules [Lecture → Rem. 7.2.0.4] and Gaussian-Legendre quadrature formulas [Lecture → Section 7.4], [Lecture → Section 7.4.2]. You must be able to perform integration by substitution.

Given a smooth function $f : [-1, 1] \to \mathbb{R}$ our task is to come up with a numerical approximation of

$$I(f) := \int_{-1}^{1} \arccos(t)\, f(t)\, \mathrm{d}t \,. \qquad (0.1.1)$$

We want to approximate this integral using global Gauss-Legendre quadrature as introduced in [Lecture → Section 7.4.2].



The function $t \mapsto \arccos t$ ▷
(≙ the inverse of $\cos$ on $[0, \pi]$)

Fig. 1

The nodes and the weights of the n-point Gauss-Legendre quadrature rule on $[-1, 1]$ can be computed for $n \le 256$ using the provided C++ function

```
QuadRule gaussquad(const unsigned n);
```

which initializes a data structure describing a quadrature rule

```
struct QuadRule {
  QuadRule() = default;
  explicit QuadRule(unsigned int n) : nodes_(n),
    weights_(n) {}
  Eigen::VectorXd nodes_;
  Eigen::VectorXd weights_;
};
```

The names of the data members tell their function. The function `gaussquad()` is declared in `gaussquad.hpp`.

(0-1.a) ⚄ (7 pts.) In the file `arccosquad.hpp` implement a C++ function

```
void testConvGaussQuad();
```

that prints a table that allows you to predict *qualitatively and quantitatively* the *asymptotic behavior* (w.r.t. $n \to \infty$) of the quadrature error of n-point Gauss-Legendre numerical quadrature, when directly applied to (0.1.1) for $f(t) = \frac{1}{1+e^t}$.

Based on that table describe the observed empiric convergence of the quadrature error as a function of the number of quadrature nodes qualitatively and quantitatively. Explain, how you arrive at your conclusions.

HINT 1 for (0-1.a): The function $t \mapsto \arccos t$ is available in C++ as **std::**`acos()`, if the header `cmath` is included. ┘

HINT 2 for (0-1.a): The integral cannot be evaluated in closed form. The idea is to use an approximate value provided by “overkill numerical quadrature” with many more quadrature nodes than used for the smallest error taken into account for gauging the convergence behavior.

Such a *reference value* is $I = 1.7576137811123187$. ┘

SOLUTION of (0-1.a):

Types of convergence have been introduced in [Lecture → Def. 6.2.2.7] for approximation errors. They apply to quadrature errors as well.

Due to the singularity of $t \mapsto \arccos t$ at $t = \pm 1$, we cannot expect exponential convergence in the number of quadrature nodes. So, taking the cue from [Lecture → § 6.2.2.9], we first try to detect algebraic convergence:

$$\epsilon_i \approx C n_i^{-p} \quad \Leftrightarrow \quad \log \epsilon_i \approx \log C - p \log n_i \,, \tag{0.1.2}$$

where $(\epsilon_i)_{i=1}^{L}$ is a sequence of quadrature errors, $(n_i)_{i=1}^{L} \in \mathbb{N}$ is the corresponding sequence of numbers of quadrature nodes and $p > 0$ is the rate of algebraic convergence. It is convenient to opt for a geometric growth of n_i, for instance, $n_i = m_0 2^i$, $m_0 \in \mathbb{N}$, because in this case

$$\log \epsilon_i \approx \log C - p \log m_0 - pi \log 2 \,, \quad i = 1, \dots, L \,.$$

Then an estimate for the rate p can be obtained as

$$p \approx \frac{\log \epsilon_{i-1} - \log \epsilon_i}{\log 2} \,. \tag{0.1.3}$$

This is another quantity, one should tabulate.

Alternatively, using $n_i = m_0 2^i$, $m_0 \in \mathbb{N}$, the error behavior like (0.1.2) also manifests itself through

$$\frac{\epsilon_i}{\epsilon_{i+1}} \approx \frac{C n_i^{-p}}{C n_{i+1}^{-p}} = 2^p \,.$$

Hence, one may increase the number of quadrature points by a constant factor in a sequence of tests and then watch the ratio of two successive errors. If that ratio attains about the same value throughout the sequence, we can also conclude algebraic convergence. Thus the values $\epsilon_{i-1} : \epsilon_i$ could also be included in the table.

C++ code 0.1.4: Convenience struct `cvgdata`

```cpp
struct cvgdata {
  long n_{0};           // number of quadrature points
  double val_{0.0};     // approximate value
  double err_{-1.0};    // quadrature error
  double rate_{-1.0};   // estimate for rate of algebraic convergence
};
```

C++ code 0.1.5: Code for `testConvGaussQuad()`

```cpp
void testConvGaussQuad() {
  // Table header
  std::cout << std::setw(5) << "n" << std::setw(20) << " I_n " << std::setw(30)
            << " Error " << std::setw(10) << (" rate\n " + std::string(65, '-'))
            << std::endl;

  // Value obtained by overkill quadrature as reference value
  constexpr double ref_val = 1.7576137811123187;

  // TO DO: (0-1.a) Print a table that allows you to predict the
      asymptotic
  // behaviour of Gauss-Legendre numerical quadrature when
      approximating I(f).
  // START
  // The integrand as lambda funtion
  auto integrand = [](double t) { return std::acos(t) * 1.0 / (1.0 +
      std::exp(t)); };

  // For recording errors etc.
  std::vector<cvgdata> data;
  // We suspect algebraic convergence. Accordingly use doubling of
      number of
  // quadrature nodes
  constexpr int L = 8;  // Use at most 2^L quadrature nodes
  int n = 2;
  for (int l = 1; l <= L; ++l, n *= 2) {
    const QuadRule qr{gaussquad(n)};
    // No transformation required: the integral is on [-1,1] anyway.
    double I = 0.0;  // Summation variable for quadrature formula
    for (int j = 0; j < n; ++j) {
      I += qr.weights_[j] * integrand(qr.nodes_[j]);
    }
    data.push_back({qr.weights_.size(), I, 0.0, 0.0});
  }
  for (int l = 0; l < data.size(); ++l) {
    data[l].err_ = std::abs(data[l].val_ - ref_val);
    if (l > 0) {
      // Estimate for rate of algebraic convergence
      data[l].rate_ =
        (std::log(data[l - 1].err_) - std::log(data[l].err_)) / std::log(2.0);
    }
  }

  // Output table
  for (int l = 0; l < data.size(); ++l) {
    const cvgdata& item{data[l]};
    std::cout << std::setw(5) << item.n_ << std::setw(20) << std::fixed
              << std::setprecision(16) << item.val_ << std::setw(30)
              << std::scientific << std::setprecision(16) << item.err_
              << std::setw(10) << std::fixed << std::setprecision(2)
              << item.rate_ << std::endl;
  }
  // END
}
```

The code produces the following output:

```
|   n          I_n                    Error              rate
 ________________________________________________________________
    2   1.7436937792198774   1.3920001892441247e-02      0.00
    4   1.7553382175123289   2.2755635999898161e-03      2.61
    8   1.7572962271476178   3.1755396470090069e-04      2.84
   16   1.7575711396897615   4.2641422557165853e-05      2.90
   32   1.7576082279627416   5.5531495770644312e-06      2.94
   64   1.7576130715900262   7.0952229247467358e-07      2.97
  128   1.7576136914136691   8.9698649619052162e-08      2.98
  256   1.7576137698376213   1.1274697397922750e-08      2.99
  512   1.7576137797013049   1.4110137502854059e-09      3.00
 1024   1.7576137809381109   1.7420775932919241e-10      3.02
 2048   1.7576137810929557   1.9362955683277505e-11      3.17
```

The data provide clear evidence for algebraic convergence with rate ≈ 3 .

▲

(0-1.b) ⚃ (5 pts.) With a suitable change of variables transform the integral (0.1.1) into another integral whose integrand will be C^∞-smooth on the *closed* interval of integration, if $f \in C^\infty([-1,1])$.

SOLUTION of (0-1.b):

We use the transformation

$$s = \arccos t \quad \Leftrightarrow \quad t = \cos s \,,$$

for which

$$dt = -\sin s \, ds \,.$$

The new integration bounds will be $\pi = \arccos(-1)$ and $0 = \arccos(1)$. Thus, we arrive at

$$I(f) = -\int_{\pi}^{0} s\, f(\cos s) \sin s \, \mathrm{d}s \ . \tag{0.1.6}$$

Since $s \mapsto \cos s$, $s \mapsto \sin s$, are entire functions, they do not affect the smoothness of the integrand:

$$f \in C^{\infty}([-1,1]) \quad \Rightarrow \quad s \mapsto s\, f(\cos s) \sin s \in C^{\infty}([0,\pi]) \ .$$

▲

(0-1.c) ⚁ (5 pts.) [depends on Sub-problem (0-1.b)]

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

```
template <typename FUNCTION>
double arccosWeightedQuad(FUNCTION &&f,
                          unsigned int n);
```

that

- expects the functor argument f to pass a function $f : [-1,1] \to \mathbb{R}$ and to provide an evaluation operator **operator double** `()` `(`**double**`)` **const**,
- uses n evaluations of the function f to compute an approximation $I_n(f)$ of $I(f)$ from (0.1.1),
- that achieves exponential asymptotic convergence $I_n(f) \to I(f)$ for $n \to \infty$, if the function f possesses an analytic extension beyond $[-1,1]$.

SOLUTION of (0-1.c):

Of course, we now apply Gauss-Legendre quadrature to the transformed integral (0.1.6). If f is analytic in a neigborhood of $[-1,1]$, then $s \mapsto sf(\cos s)\sin s$ will be analytic in a neighborhood of $[0,\pi]$, because $s \mapsto \cos s$, $s \mapsto \sin s$, are entire functions, that is, analytic on all of $\mathbb{C}$.

Note that the weights and nodes of the Gauss-Legendre quadrature rules have to be transformed to the interval $[0,\pi]$ as explained in [Lecture → Rem. 7.2.0.4].

C++ code 0.1.7: Code for `arccosWeightedQuad()`

```
template <typename FUNCTION>
double arccosWeightedQuad(FUNCTION &&f, unsigned int n) {
  double I = 0.0; // For accumulating quadrature result
  // TO DO: (0-1.c) Approximate I(f) with exponential convergence in
     n.
  // START
  // Obtain Gauss-Legendre quadrature rule
  QuadRule qr{gaussquad(n)};
  // The transformed integral covers the interval [0,π], which entails
  // a transformation of the quadrature nodes and weights, see
     [Lecture → Rem. 7.2.0.4]
  qr.weights_ *= 0.5*M_PI;
  qr.nodes_ = 0.5*M_PI*(1.0+qr.nodes_.array());
  // Straightforward implementation of quadrature formula
  for (int j = 0; j < n; ++j) {
    const double c = qr.nodes_[j];
    const double fval = f(std::cos(c));
    I += qr.weights_[j] * fval * std::sin(c) * c;
  }
  // END
  return I;
}
```

▲

(0-1.d) ⚄ (7 pts.) [depends on Sub-problem (0-1.a) and Sub-problem (0-1.c)]

Analoguous to Sub-problem (0-1.a) in the file `arccosquad.hpp` write a C++ function

```
void testConvTrfGaussQuad();
```

that outputs (in a suitable table) information that allows you to predict *qualitatively and quantitatively* the *asymptotic behavior* (w.r.t. $n \to \infty$) of the approximation error of your implementation of `arccosWeightedQuad()` for $f(t) = \frac{1}{1+e^t}$.

Characterize the observed empiric convergence of the quadrature error as a function of n qualitatively and quantitatively. Justify your conclusions.

SOLUTION of (0-1.d):

Note that $t \mapsto \frac{1}{1+\exp(t)}$ is analytic in a neighborhood of $[-1,1]$, because its domain of analyticity is

$$\mathbb{C} \setminus (2\mathbb{Z}+1)\imath \; .$$

Thus, this time we expect exponential convergence, that is, using the notation from the solution of Sub-problem (0-1.a) we expect

$$\epsilon_i \approx C\exp(-qn_i) \quad \text{for some} \quad q > 0 \; . \tag{0.1.8}$$

In this case it is advantageous to let n_i increase linearly, for instance, $n_i = i$, because this implies

$$\frac{\epsilon_{i+1}}{\epsilon_i} \approx \exp(-q) < 1 \, .$$

In the case of exponential convergence we should observe a decay of the approximation error by a constant factor when raising n by 1. Thus tabulating the ratio of consecutive errors is highly recommended.

C++ code 0.1.9: Code for `testConvTrfGaussQuad()`

```cpp
void testConvTrfGaussQuad() {
  // Table header
  std::cout << std::setw(5) << "n" << std::setw(20) << " I_n " << std::setw(30)
            << " Error " << std::setw(10)
            << (" decay ratio\n " + std::string(65, '-')) << std::endl;

  // Value obtained by overkill quadrature as reference value
  const double ref_val = 1.7576137811123187;

  // TO DO: (0-1.d) Print a table that allows you to predict the
     asymptotic
  // behaviour of arccosWeightedQuad().
  // START
  // The integrand (/arccos(t)) as lambda funtion
  auto f = [](double t) { return 1.0 / (1.0 + std::exp(t)); };

  // For recording errors etc.
  std::vector<cvgdata> data;
  // We expect exponential convergence. Increment number of quadrature
     nodes by
  // one in each step
  constexpr int L = 12;  // Use a most L quadrature nodes
  for (int n = 2; n <= L; ++n) {
    data.push_back({n, arccosWeightedQuad(f, n), 0.0, 0.0});
  }
  for (int l = 0; l < data.size(); ++l) {
    data[l].err_ = std::abs(data[l].val_ - ref_val);
    if (l > 0) {
      // Quotient of successive error values
      data[l].rate_ = data[l].err_ / data[l - 1].err_;
    }
  }

  // Output table
  for (int l = 0; l < data.size(); ++l) {
    const cvgdata& item{data[l]};
    std::cout << std::setw(5) << item.n_ << std::setw(20) << std::fixed
              << std::setprecision(16) << item.val_ << std::setw(30)
              << std::scientific << std::setprecision(16) << item.err_
              << std::setw(10) << std::fixed << std::setprecision(2)
              << item.rate_ << std::endl;
  }
  // END
}
```

The code produces the following output:

```
n           I\_n                  Error              decay  ratio
-----------------------------------------------------------------
2   1.8492385346839764   9.1624753571657669e-02    0.00
3   1.7329033484157947   2.4710432696523954e-02    0.27
4   1.7605214393475945   2.9076582352758340e-03    0.12
5   1.7572524700920542   3.6131102026448758e-04    0.12
6   1.7576617761797750   4.7995067456341189e-05    0.13
7   1.7576074242880906   6.3568242281153431e-06    0.13
8   1.7576146283917027   8.4727938398643232e-07    0.13
9   1.7576136678083938   1.1330392490904728e-07    0.13
10  1.7576137962618741   1.5149555387949931e-08    0.13
11  1.7576137790901427   2.0221759822192098e-09    0.13
12  1.7576137813858141   2.7349544851063001e-10    0.14
```

The ratio of successive approximation errors clearly stabilizes at a value $\approx 0.14 < 1$, which hints at exponential convergence :

$$|I(f) - \texttt{arccosWeightedQuad(f,n)}| = O(0.14^n) \quad \text{for} \quad n \to \infty \,.$$

▲

End Problem 0-1 , 24 pts.

Problem 0-2: Implementing Discrete Convolution

[Lecture → Section 4.2.2] discussed in detail the implementation of discrete convolutions of signals stored in vectors by means of the discrete Fourier transform (DFT). This problem revisits these techniques and adds a new twist to implementation.

This problem assumes solid knowledge of [Lecture → Section 4.2.2].

From [Lecture → Section 4.1.3] we recall the definitions

Definition [Lecture → Def. 4.1.3.3]. Discrete convolution

Given $\mathbf{x} = [x_0, \dots, x_{m-1}]^\top \in \mathbb{K}^m$, $\mathbf{h} = [h_0, \dots, h_{n-1}]^\top \in \mathbb{K}^n$ their **discrete convolution** (DCONV) $\mathbf{h} * \mathbf{x}$ is the vector $\mathbf{y} = [y_0, \dots, y_{m+n-2}]^\top \in \mathbb{K}^{m+n-1}$ with components

$$y_k = \sum_{j=0}^{m-1} h_{k-j} x_j \;, \quad k = 0, \dots, m+n-2 \;, \qquad \text{[Lecture → Eq. (4.1.3.4)]}$$

where we have adopted the convention $h_j := 0$ for $j < 0$ or $j \geq n$.

and

Definition *cf.* [Lecture → Def. 4.1.4.7]. Discrete periodic convolution (of vectors)

The **discrete periodic convolution** $\mathbf{p} *_n \mathbf{x}$ of two vectors $\mathbf{p}, \mathbf{x} \in \mathbb{K}^n$ is the vector $\mathbf{y} \in \mathbb{K}^n$ with components (C++ indexing)

$$(\mathbf{y})_k := \sum_{j=0}^{n-1} (\mathbf{p})_{(k-j) \bmod n} (\mathbf{x})_j \;, \quad k \in \{0, \dots, n-1\} \;.$$

($(k-j) \bmod n \in \{0, \dots, n-1\} \;\hat{=}\;$ `(k-j)%n` in C++ syntax.)

(0-2.a) ⚃ (7 pts.) For given $\mathbf{h} \in \mathbb{K}^n$, $\mathbf{x} \in \mathbb{K}^m$, specify the vectors $\widetilde{\mathbf{h}}, \widetilde{\mathbf{x}}$, the *minimal* vector length $N \in \mathbb{N}$, and the missing index range $?:?$ in the formula

$$\mathbf{h} * \mathbf{x} = \left(\widetilde{\mathbf{h}} *_N \widetilde{\mathbf{x}}\right)_{?:?} \;.$$

C++ indexing has to be used and the notational convention is $(\mathbf{v})_{r:s} = [(\mathbf{v})_r, \dots, (\mathbf{v})_s] \in \mathbb{K}^{s-r+1}$ for $r, s \in \mathbb{N}$, $r \leq s$.

SOLUTION of (0-2.a):

We can rewrite the discrete convolution of two vectors [Lecture → Eq. (4.1.3.4)] as (C++ indexing used)

$$(\mathbf{h} * \mathbf{x})_k = \sum_{j=\max\{0,k-n+1\}}^{\min\{m-1,k\}} (\mathbf{x})_j (\mathbf{h})_{k-j}\;, \quad k = 0, \dots, m+n-2\,. \tag{0.2.1}$$

We take the cue from [Lecture → Rem. 4.1.4.15] and borrow the crucial trick of zero padding. For $N \geq \max\{m,n\}$ we introduce the zero-padded vectors

$$\widetilde{\mathbf{x}} := \begin{bmatrix} \mathbf{x} \\ \mathbf{0} \end{bmatrix} \in \mathbb{K}^N \quad , \quad \widetilde{\mathbf{h}} := \begin{bmatrix} \mathbf{h} \\ \mathbf{0} \end{bmatrix} \in \mathbb{K}^N\,.$$

Next we closely examine the periodic convolution of these two vectors, exploiting our knowledge about zero components of $\widetilde{\mathbf{x}}$ and $\widetilde{\mathbf{h}}$:

$$\begin{aligned}\left(\widetilde{\mathbf{h}} *_N \widetilde{\mathbf{x}}\right)_k &= \sum_{j=0}^{N-1} (\widetilde{\mathbf{x}})_j \left(\widetilde{\mathbf{h}}\right)_{(k-j) \bmod N} = \sum_{j=0}^{m-1} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{(k-j) \bmod N} \\ &= \sum_{j=0}^{\min\{m-1,k\}} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{k-j} + \sum_{j=\min\{m,k+1\}}^{m-1} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{k+N-j}\,.\end{aligned}$$

❶ Case $k \in \{m-1, \dots, N-1\}$: In this case the second sum becomes void and we find

$$\left(\widetilde{\mathbf{h}} *_N \widetilde{\mathbf{x}}\right)_k = \sum_{j=0}^{m-1} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{k-j} = \sum_{j=\max\{0,k-n+1\}}^{\min\{m-1,k\}} (\mathbf{x})_j (\mathbf{h})_{k-j} = (\mathbf{h} * \mathbf{x})_k\,,$$

that is, for these vector components both discrete convolutions produce the same result for any $N \geq \max\{m,n\}$.

❷ Case $k \in \{0, \dots, m-2\}$: The corresponding components of the discrete periodic convolution are

$$\begin{aligned}\left(\widetilde{\mathbf{h}} *_N \widetilde{\mathbf{x}}\right)_k &= \sum_{j=0}^{k} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{k-j} + \sum_{j=k+1}^{m-1} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{k+N-j} \\ &= \sum_{j=\max\{0,k-n+1\}}^{k} (\mathbf{x})_j (\mathbf{h})_{k-j} + \sum_{j=k+1}^{m-1} (\mathbf{x})_j \left(\widetilde{\mathbf{h}}\right)_{k+N-j}\,.\end{aligned}$$

Clearly, the second sum should evaluate to zero if both discrete convolutions are to yield the same vector components. Thus, we have to demand

$$\begin{aligned} & \left(\widetilde{\mathbf{h}}\right)_{k+N-j} = 0 && \forall j \in \{k+1, \dots, m-1\}\,, \quad \forall k \in \{0, \dots, m-2\} \\ \Leftrightarrow\quad & k + N - j \geq n && \forall j \in \{k+1, \dots, m-1\}\,, \quad \forall k \in \{0, \dots, m-2\} \\ \Leftrightarrow\quad & N - (m-1) \geq n && \\ \Leftrightarrow\quad & N \geq n + m - 1\,. && \end{aligned}$$

Hence, $N \geq n+m-1$ is a necessary and sufficient condition for the agreement of the vector components with indices $k \in \{0,\dots,m-2\}$.

Summing up we have found $N = n+m-1$ and

$$\boxed{\mathbf{h} * \mathbf{x} = \left(\widetilde{\mathbf{h}} *_{m+n-1} \widetilde{\mathbf{x}}\right)_{0:n+m-2}} \; .$$

In fact, the range specification for the vector components could be dropped, because for $N = n+m-1$ all components of the vector produced by the discrete periodic convolution $*_{m+n-1}$ are relevant.

The following code demonstrates the implementation of this idea:

C++ code 0.2.2: Discrete convolution by periodic convolution

```cpp
Eigen::VectorXd dconv_p(const Eigen::VectorXd &h, const Eigen::VectorXd &x) {
  const int n = h.size();   // length of vector h
  const int m = x.size();   // length of vector x
  const int N = m + n - 1;  // Minimal length of periodic convolution
  // Zero-padded vectors of length m+n-1
  Eigen::VectorXd ht = Eigen::VectorXd::Zero(N);
  Eigen::VectorXd xt = Eigen::VectorXd::Zero(N);
  ht.head(n) = h;
  xt.head(m) = x;
  // Discrete periodic convolution, see [Lecture → Code 4.2.2.1]
  return pconv(ht, xt);
}
```

A basic loop-based implementation of `pconv()` is shown in [Lecture → Code 4.2.2.1].

▲

The discrete Fourier transform in EIGEN relies on the library KISSFFT, which offers a very efficient implementation of DFT with asymptotic computational cost $O(n \log n)$ for vectors of length $n := 2^\ell$, $\ell \in \mathbb{N}$, but may not be fast for vectors of other lengths. To avoid degraded performance of a numerical code, the following rule is imposed throughout this problem:

EIGEN's built-in DFT implementations may be invoked only for vectors of length 2^ℓ for some $\ell \in \mathbb{N}$!

(0-2.b) ⚃ (20 pts.) Based on EIGEN, in the file `discreteconvolution.hpp` implement an *efficient* function

```cpp
Eigen::VectorXd pconv_fast(const Eigen::VectorXd &p,
                           const Eigen::VectorXd &x);
```

that computes the discrete periodic convolution $\mathbf{p} *_n \mathbf{x}$ as defined in [Lecture $\to$ Def. 4.1.4.7] of two real vectors $\mathbf{p}, \mathbf{x} \in \mathbb{R}^n$ of the same length.

HINT 1 for (0-2.b): Please note that EIGEN's FFT implementation can only deal with vectors with complex-valued entries, that is, the type **Eigen::VectorXcd**. In EIGEN you have to use the method `cast<std::complex<double>>()` when using both real-valued and complex-valued vectors in an expression. ⌟

SOLUTION of (0-2.b):

We elaborate how to realize a general periodic discrete convolution $\mathbf{p} *_n \mathbf{x}$, $\mathbf{p}, \mathbf{x} \in \mathbb{K}^n$, $n \in \mathbb{N}$, by means of a discrete periodic convolution of vectors of length $N > n$. We assume $N \geq 2n$!

We first rewrite the formula for the discrete periodic convolution of $\mathbf{p} = [p_0, \dots, p_{n-1}]^\top \in \mathbb{K}^n$, $\mathbf{x} = [x_0, \dots, x_{n-1}]^\top \in \mathbb{K}^n$ from [Lecture $\to$ Def. 4.1.4.7], see also [Lecture $\to$ Code 4.2.2.1]:

$$(\mathbf{p} *_n \mathbf{x})_k = \sum_{j=0}^{n-1} x_j p_{(k-j) \bmod n} = \sum_{j=0}^{k} x_j p_{k-j} + \sum_{j=k+1}^{n-1} x_j p_{n+k-j} \,, \quad k = 0, \dots, n-1 \,. \tag{0.2.3}$$

Then we introduce auxiliary vectors by zero padding and periodic padding, respectively:

$$\widetilde{\mathbf{x}} = \begin{bmatrix} \widetilde{x}_0 \\ \vdots \\ \widetilde{x}_{N-1} \end{bmatrix} := \begin{bmatrix} \mathbf{x} \\ \mathbf{0} \end{bmatrix} \in \mathbb{K}^N \quad , \quad \widetilde{\mathbf{p}} = \begin{bmatrix} \widetilde{p}_0 \\ \vdots \\ \widetilde{p}_{N-1} \end{bmatrix} := \begin{bmatrix} \mathbf{p} \\ \mathbf{0} \\ \mathbf{p} \end{bmatrix} \in \mathbb{K}^N \,.$$

In particular, this means

$$\widetilde{p}_j = p_j \quad \text{for} \quad j = 0, \dots, n-1 \quad , \quad \widetilde{p}_{N-\ell} = p_{n-\ell} \quad \text{for} \quad \ell = 1, \dots, n \,. \tag{0.2.4}$$

Based on (0.2.3), we examine the size-N discrete periodic convolution of $\widetilde{\mathbf{p}}$ and $\widetilde{\mathbf{x}}$:

$$(\widetilde{\mathbf{p}} *_N \widetilde{\mathbf{x}})_k = \sum_{j=0}^{k} \widetilde{x}_j \widetilde{p}_{k-j} + \sum_{k=k+1}^{N-1} \widetilde{x}_j \widetilde{p}_{N+k-j} \,.$$

First we take into account that $\widetilde{x}_j = 0$ for $j \geq n$:

$$\blacktriangleright \quad (\widetilde{\mathbf{p}} *_N \widetilde{\mathbf{x}})_k = \sum_{j=0}^{k} x_j \widetilde{p}_{k-j} + \sum_{j=k+1}^{n-1} x_j \widetilde{p}_{N+k-j} \,. \tag{0.2.5}$$

Sums whose lower bound is larger than the upper bound are meant to be $= 0$.

Now we restrict ourselves to $k \in \{0, \dots, n-1\}$, which implies

- that in the first sum of (0.2.5) that $k - j \in \{0, \dots, n-1\}$,
- and that $j - k \in \{1, \dots, n-1\}$ in the second sum of (0.2.5), which means $\widetilde{p}_{N+k-j} = p_{n+k-j}$ thanks to (0.2.4).

$$(\widetilde{\mathbf{p}} *_N \widetilde{\mathbf{x}})_k = \sum_{j=0}^{k} x_j p_{k-j} + \sum_{k=k+1}^{n-1} x_j p_{n+k-j} = (\mathbf{p} *_n \mathbf{x})_k \,, \quad k \in \{0, \dots, n-1\} \,.$$

This is exploited in the following C++ function, to which the parameter `N` passes N:

C++ code 0.2.6: Resizing periodic convolution

```cpp
Eigen::VectorXd pconvN(const Eigen::VectorXd &p, const Eigen::VectorXd &x,
                       unsigned int N) {
  const int n = x.size();
  assert(n == p.size());
  // Requirement on N to avoid overlap in periodic padding
  assert(N >= 2 * n);
  // Periodic padding of vector p
  Eigen::VectorXd pt{Eigen::VectorXd::Zero(N)};
  pt.head(n) = p;
  pt.tail(n) = p;
  // Zero padding of vector x
  Eigen::VectorXd xt{Eigen::VectorXd::Zero(N)};
  xt.head(n) = x;
  // Periodic convolution of length N
  const Eigen::VectorXd y = pconv(pt, xt);
  return y.head(n);
}
```

Of course, any efficient implementation must be based on DFTs by the FFT algorithm. Thus, we have to replace the call to `pconv()` in Code 0.2.6 by the DFT-based implementation given in [Lecture → Code 4.2.2.1]. We have to comply with the requirement that only DFTs of length $N = 2^\ell$, $\ell \in \mathbb{N}$ are admissible, and also meet the condition $N \geq 2n$. Thus, N has to be chosen as the smallest power of 2 which is $\geq 2n$. The following code chooses that N and merges Code 0.2.6 and [Lecture → Code 4.2.2.1].

C++ code 0.2.7: Fast discrete periodic convolution implemented by FFT-based DFT of length $N = 2^\ell$

```cpp
Eigen::VectorXd pconv_fast(const Eigen::VectorXd &p, const Eigen::VectorXd &x) {
  const int n = x.size();
  assert(n == p.size());
  Eigen::VectorXd z = Eigen::VectorXd::Zero(n);

  // TO DO: (0-2.b) Compute the discrete periodic convolution of p
      with x
  // in an efficient manner.
  // START

  // N >= 2*n is required to avoid overlap in periodic padding
  int N = 1;
  while (N < 2 * n) {
    N *= 2;
  }
  // Note that Eigen's FFT routines expect complex vectors
  // Periodic padding of vector p
  Eigen::VectorXcd pt{Eigen::VectorXcd::Zero(N)};
  pt.head(n) = p.template cast<std::complex<double>>();
  pt.tail(n) = p.template cast<std::complex<double>>();
  // Zero padding of vector x
  Eigen::VectorXcd xt{Eigen::VectorXcd::Zero(N)};
  xt.head(n) = x.template cast<std::complex<double>>();
  // Periodic convolution of length N = 2^l realized by FFT
  Eigen::FFT<double> fft;
  z = (fft.inv(((fft.fwd(pt)).cwiseProduct(fft.fwd(xt))).eval()))
          .real()
          .head(n);

  // END
  return z;
}
```

Note that this choice of N is important for efficiency, because choosing a larger power of 2 will incur significant extra work for large n.

Remark. An even more efficient implementation could also exploit the fact that $\mathbf{p},\mathbf{x}$ are *real* vectors, see [Lecture $\rightarrow$ Section 4.2.4]. However, this further optimization is not expected here.

▲

End Problem 0-2 , 27 pts.

Problem 0-3: QR-Iteration

We have learned about the QR-decomposition/factorization as a tool for solving linear systems of equations [Lecture → Rem. 3.3.4.3] and linear least-squares problems [Lecture → Section 3.3.4]. Yet, this matrix factorization has many more important applications. For instance, it is a crucial ingredient for the so-called QR-algorithm, an iteration for solving algebraic eigenvalue problems [GV13, Sections 7.5 & 8.3]. This problem will examine some algorithmic aspects of that algorithm.

You are supposed to know the contents of [Lecture → Section 3.3.3] and implementation in EIGEN.

We recall the QR-decomposition of matrices:

Theorem [Lecture → Thm. 3.3.3.4]. QR-decomposition

For any matrix $\mathbf{A} \in \mathbb{K}^{n,k}$ *with* $\mathrm{rank}(\mathbf{A}) = k$ *there exists*

(i) *a unique Matrix* $\mathbf{Q}_0 \in \mathbb{R}^{n,k}$ *that satisfies* $\mathbf{Q}_0^{\mathrm{H}}\mathbf{Q}_0 = \mathbf{I}_k$, *and a unique upper triangular Matrix* $\mathbf{R}_0 \in \mathbb{K}^{k,k}$ *with* $(\mathbf{R})_{i,i} > 0$, $i \in \{1,\dots,k\}$, *such that*

$$\mathbf{A} = \mathbf{Q}_0 \cdot \mathbf{R}_0 \qquad (\text{“economical” QR-decomposition})\,,$$

(ii) *a unitary Matrix* $\mathbf{Q} \in \mathbb{K}^{n,n}$ *and a unique upper triangular* $\mathbf{R} \in \mathbb{K}^{n,k}$ *with* $(\mathbf{R})_{i,i} > 0$, $i \in \{1,\dots,n\}$, *such that*

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \qquad (\text{full QR-decomposition})\,.$$

If $\mathbb{K} = \mathbb{R}$ *all matrices will be real and* $\mathbf{Q}$ *is then orthogonal.*

(0-3.a) ⚃ (5 pts.) Prove the following fact:

Lemma 0.3.1. Lower bandwidth of Q-factor

Let $\mathbf{T} = \mathbf{QR}$ *be a QR-decomposition of a regular tridiagonal matrix* $\mathbf{T} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$. *Then* $\mathbf{Q}$ *has lower bandwidth 1, that is,*

$$i,j \in \{1,\dots,n\}\,,\quad i > j+1 \quad \Rightarrow \quad (\mathbf{Q})_{i,j} = 0\,.$$

HINT 1 for (0-3.a): Remember the connection of QR-decomposition and Gram-Schmidt orthogonalization and [Lecture → Thm. 3.3.3.2]. ⌟

SOLUTION of (0-3.a):

Two possible arguments:

❶ The leftmost m columns of the Q-factor of a QR-decomposition of a regular matrix span the same space as the corresponding columns of that matrix. Since $\mathbf{T} \in \mathbb{R}^{n,n}$ is tridiagonal, we have

$$\operatorname{Span}\{(\mathbf{Q})_{:,1},\dots,(\mathbf{Q})_{:,m}\} = \operatorname{Span}\{(\mathbf{T})_{:,1},\dots,(\mathbf{T})_{:,m}\} \subset \operatorname{Span}\{\mathbf{e}_1,\dots,\mathbf{e}_{m+1}\},$$

$m \in \{1,\dots,n\}$, where $\mathbf{e}_\ell \in \mathbb{R}^n$ stands for the ℓ-th coordinate vector. Hence,

$$(\mathbf{Q})_{:,m} \in \operatorname{Span}\{\mathbf{e}_1,\dots,\mathbf{e}_{m+1}\}$$
$$\Updownarrow$$
$$(\mathbf{Q})_{:,m} = \xi_1\mathbf{e}_1 + \xi_2\mathbf{e}_2 + \dots + \xi_m\mathbf{e}_m + \xi_{m+1}\mathbf{e}_{m+1}, \quad \xi_1,\dots,\xi_{m+1} \in \mathbb{R},$$

which immediately implies the assertion of the lemma.

❷ Since $\mathbf{T} \in \mathbb{R}^{n,n}$ is supposed to be regular, so will be the upper triangular matrix $\mathbf{R}$. Thus we can write $\mathbf{Q} = \mathbf{T}\mathbf{R}^{-1}$. Then appeal to [Lecture → Lemma 1.3.1.9].

Lemma [Lecture → Lemma 1.3.1.9]. Group of regular diagonal/triangular matrices

$$\mathbf{A},\mathbf{B} \left\{\begin{array}{c} \textit{diagonal} \\ \textit{upper triangular} \\ \textit{lower triangular} \end{array}\right. \Rightarrow \quad \mathbf{AB} \quad \textit{and} \quad \mathbf{A}^{-1} \textit{ are} \left\{\begin{array}{c} \textit{diagonal} \\ \textit{upper triangular} \\ \textit{lower triangular} \end{array}\right. .$$

(assumes that $\mathbf{A}$ is regular)

This ensures that $\mathbf{R}^{-1}$ is upper triangular, too:

$$i > j \quad \Rightarrow \quad (\mathbf{R}^{-1})_{i,j} = 0 \,. \tag{0.3.2}$$

Thus, if $i > j+1$ we get

$$(\mathbf{Q})_{i,j} = (\mathbf{T}\mathbf{R}^{-1})_{i,j} = \sum_{\ell=1}^{n} (\mathbf{T})_{i,\ell}(\mathbf{R}^{-1})_{\ell,j} \overset{(0.3.2)}{=} \sum_{\ell=1}^{j} \underbrace{(\mathbf{T})_{i,\ell}}_{=0}(\mathbf{R}^{-1})_{\ell,j},$$

because $(\mathbf{T})_{i,\ell} = 0$ for $\ell \in \{1,\dots,j\}$, since $\mathbf{T}$ is tridiagonal.

▲

(0-3.b) ⚂ (5 pts.) Give a proof of the following result:

Lemma 0.3.3. QR-based similarity transform

If $\mathbf{T} = \mathbf{QR}$ is the QR-decomposition of a regular, symmetric, and tridiagonal matrix $\mathbf{T} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, then $\mathbf{T}' := \mathbf{RQ}$ is also symmetric and tridiagonal.

HINT 1 for (0-3.b): Lemma 0.3.1 may be useful for the proof and can be used even if you have not succeeded in proving it. ⌟

SOLUTION of (0-3.b):

Since $\mathbf{R}$ is upper triangular and $\mathbf{Q}$ has lower bandwidth ≤ 1 by Lemma 0.3.1,

$$\begin{aligned} i,j \in \{1,\dots,n\} \quad &\text{and} \quad i > j \quad \Rightarrow \quad (\mathbf{R})_{i,j} = 0\,, \\ i,j \in \{1,\dots,n\} \quad &\text{and} \quad i > j+1 \quad \Rightarrow \quad (\mathbf{Q})_{i,j} = 0\,, \end{aligned}$$

we find that for $i,j \in \{1,\dots,n\}$, $i > j+1$,

$$(\mathbf{T}')_{i,j} = \sum_{\ell=1}^{n} (\mathbf{R})_{i,\ell}(\mathbf{Q})_{\ell,j} = \sum_{\ell=1}^{i-1} \underbrace{(\mathbf{R})_{i,\ell}}_{=0}(\mathbf{Q})_{\ell,j} + \sum_{\ell=i}^{n} (\mathbf{R})_{i,\ell}\underbrace{(\mathbf{Q})_{\ell,j}}_{=0} = 0\,.$$

In words, $\mathbf{T}'$ has lower bandwidth ≤ 1.

Since $\mathbf{Q}$ is *orthogonal*, $\mathbf{Q}^{-1} = \mathbf{Q}^{\top}$, we deduce the identity

$$\mathbf{T}' = \mathbf{R}\mathbf{Q} = \mathbf{Q}^{\top}\mathbf{Q}\mathbf{R}\mathbf{Q} = \mathbf{Q}^{\top}\mathbf{T}\mathbf{Q}\,,$$

which implies that $\mathbf{T}'$ is symmetric:

$$(\mathbf{T}')^{\top} = \left(\mathbf{Q}^{\top}\mathbf{T}\mathbf{Q}\right)^{\top} = \mathbf{Q}^{\top}\mathbf{T}\mathbf{Q} = \mathbf{T}'\,.$$

Finally, a symmetric matrix with lower bandwidth ≤ 1 must be tridiagonal

$$i,j \in \{1,\dots,n\}\colon \begin{array}{c} i > j+1 \quad \Rightarrow \quad (\mathbf{M})_{i,j} = 0\,, \\ (\mathbf{M})_{i,j} = (\mathbf{M})_{j,i} \end{array} \quad \blacktriangleright \quad (\mathbf{M})_{i,j} = 0 \quad \text{for} \quad |i-j| > 1\,.$$

▲

(0-3.c) ⚄ (30 pts.) A *symmetric* and *tridiagonal* matrix $\mathbf{T} \in \mathbb{R}^{n,n}$ can be encoded by two vectors $\mathbf{d} = \mathbf{d}(\mathbf{T}) \in \mathbb{R}^n$ and $\mathbf{u} = \mathbf{u}(\mathbf{T}) \in \mathbb{R}^{n-1}$ when setting

$$(\mathbf{T})_{i,j} = \begin{cases} (\mathbf{d})_i & \text{, if} \quad i = j\,, \\ (\mathbf{u})_i & \text{, if} \quad j = i+1\,, \\ (\mathbf{u})_j & \text{, if} \quad i = j+1\,, \\ 0 & \text{else.} \end{cases} \qquad i,j \in \{1,\dots,n\}:$$

► $$\mathbf{T} = \begin{bmatrix} (\mathbf{d})_1 & (\mathbf{u})_1 & 0 & \dots & & & \dots & 0 \\ (\mathbf{u})_1 & (\mathbf{d})_2 & (\mathbf{u})_2 & 0 & & & & 0 \\ 0 & (\mathbf{u})_2 & (\mathbf{d})_3 & (\mathbf{u})_3 & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & & \\ & & & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & 0 & (\mathbf{u})_{n-2} & (\mathbf{d})_{n-1} & (\mathbf{u})_{n-1} \\ 0 & \dots & & & & 0 & (\mathbf{u})_{n-1} & (\mathbf{d})_n \end{bmatrix} .$$

In the file `qriteration.hpp` implement a C++ function

```
std::pair<Eigen::VectorXd, Eigen::VectorXd>
  qrStep(const Eigen::VectorXd &dT,
         const Eigen::VectorXd &uT);
```

that takes the vectors $\mathbf{d}(\mathbf{T}) \in \mathbb{R}^n$ and $\mathbf{u}(\mathbf{T}) \in \mathbb{R}^{n-1}$ describing a symmetric, tridiagonal matrix $\mathbf{T} \in \mathbb{R}^{n,n}$ as arguments, and

- returns the tuple $(\mathbf{d}(\mathbf{T}'), \mathbf{u}(\mathbf{T}'))$ of the defining vectors $\mathbf{d}(\mathbf{T}')$ and $\mathbf{u}(\mathbf{T}')$ of the symmetric, tridiagonal matrix $\mathbf{T}' := \mathbf{Q}^\top \mathbf{T} \mathbf{Q}$, where $\mathbf{Q}$ is the Q-factor of the QR-decomposition $\mathbf{T} = \mathbf{QR}$ of $\mathbf{T}$, *cf.* Lemma 0.3.1,
- and features an asymptotic complexity of $O(n)$ for $n \to \infty$.

HINT 1 for (0-3.c): In the file `qriteration.cpp` you have at your disposal a function

```
Eigen::Vector2d givens(Eigen::Vector2d a);
```

that, when supplied with any vector $\mathbf{a} \in \mathbb{R}^2$ returns a vector $\begin{bmatrix} c \\ s \end{bmatrix} \in \mathbb{R}^2$ such that

$$s^2 + c^2 = 1 \quad \text{and} \quad \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \mathbf{a} = \begin{bmatrix} \pm\|\mathbf{a}\| \\ 0 \end{bmatrix} .$$

┘

HINT 2 for (0-3.c): Lemma 0.3.3 is crucial for the design of the algorithm! ┘

SOLUTION of (0-3.c):

We write $\mathbf{G}_{i,j}^{\ell}(\mathbf{M}) \in \mathbb{R}^{n,n}$ for the orthogonal matrix describing a **Givens rotation** [Lecture → § 3.3.3.15] that through left-multiplication acts on rows i and j of $\mathbf{M} \in \mathbb{R}^{n,n}$, $i, j \in \{1, \dots, n\}$, $i \neq j$, such that a particular entry of $\mathbf{M}$ is annihilated:

$$\left(\mathbf{G}_{i,j}^{\ell}(\mathbf{M}) \mathbf{M} \right)_{j,\ell} = 0 \,, \quad \ell = 1, \dots, n \,.$$

From [Lecture → Rem. 3.3.3.26] we know that a tridiagonal matrix $\mathbf{T} \in \mathbb{R}^{n,n}$ can be transformed into an upper triangular matrix $\mathbf{R}$ by means of $n-1$ successive Givens rotations:

$$\mathbf{T}^{(1)} := \mathbf{T}\,,\quad \mathbf{T}^{(k)} := \mathbf{G}^{k-1}_{k-1,k}(\mathbf{T}^{(k-1)})\mathbf{T}^{(k-1)}\,,\quad k = 2,\dots,n\,,\quad \mathbf{R} = \mathbf{T}^{(n)}: \tag{0.3.4}$$

$$\begin{bmatrix} \ast & \ast & 0 & 0 & 0 & 0 & 0 & 0 \\ \ast & \ast & \ast & 0 & 0 & 0 & 0 & 0 \\ 0 & \ast & \ast & \ast & 0 & 0 & 0 & 0 \\ 0 & 0 & \ast & \ast & \ast & 0 & 0 & 0 \\ 0 & 0 & 0 & \ast & \ast & \ast & 0 & 0 \\ 0 & 0 & 0 & 0 & \ast & \ast & \ast & 0 \\ 0 & 0 & 0 & 0 & 0 & \ast & \ast & \ast \\ 0 & 0 & 0 & 0 & 0 & 0 & \ast & \ast \end{bmatrix} \xrightarrow{\mathbf{G}^1_{1,2}} \begin{bmatrix} \ast & \ast & \ast & 0 & 0 & 0 & 0 & 0 \\ 0 & \ast & \ast & 0 & 0 & 0 & 0 & 0 \\ 0 & \ast & \ast & \ast & 0 & 0 & 0 & 0 \\ 0 & 0 & \ast & \ast & \ast & 0 & 0 & 0 \\ 0 & 0 & 0 & \ast & \ast & \ast & 0 & 0 \\ 0 & 0 & 0 & 0 & \ast & \ast & \ast & 0 \\ 0 & 0 & 0 & 0 & 0 & \ast & \ast & \ast \\ 0 & 0 & 0 & 0 & 0 & 0 & \ast & \ast \end{bmatrix} \xrightarrow{\mathbf{G}^2_{2,3}\cdots\mathbf{G}^{n-1}_{n-1,n}} \begin{bmatrix} \ast & \ast & \ast & 0 & 0 & 0 & 0 & 0 \\ 0 & \ast & \ast & \ast & 0 & 0 & 0 & 0 \\ 0 & 0 & \ast & \ast & \ast & 0 & 0 & 0 \\ 0 & 0 & 0 & \ast & \ast & \ast & 0 & 0 \\ 0 & 0 & 0 & 0 & \ast & \ast & \ast & 0 \\ 0 & 0 & 0 & 0 & 0 & \ast & \ast & \ast \\ 0 & 0 & 0 & 0 & 0 & 0 & \ast & \ast \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ast \end{bmatrix}$$

Abbreviating the Givens rotations, we have

$$\mathbf{G}^{n-1}_{n-1,n} \cdot \dots \cdot \mathbf{G}^2_{2,3}\mathbf{G}^1_{1,2}\mathbf{T} = \mathbf{R}\,,$$

which supplies a representation of the Q-factor $\mathbf{Q}$ of the QR-decomposition of $\mathbf{T}$ as a product of Givens rotations:

$$\mathbf{Q} = \left(\mathbf{G}^1_{1,2}\right)^\top \left(\mathbf{G}^2_{2,3}\right)^\top \cdot \dots \cdot \left(\mathbf{G}^{n-1}_{n-1,1}\right)^\top .$$

We conclude

$$\mathbf{T}' = \mathbf{G}^{n-1}_{n-1,1} \cdot \dots \cdot \mathbf{G}^2_{2,3}\mathbf{G}^1_{1,2}\,\mathbf{T}\,\left(\mathbf{G}^1_{1,2}\right)^\top \left(\mathbf{G}^2_{2,3}\right)^\top \cdot \dots \cdot \left(\mathbf{G}^{n-1}_{n-1,1}\right)^\top .$$

This suggests a two-stage algorithm for computing $\mathbf{T}'$:

(I) Compute $\mathbf{R}$ according to (0.3.4) by applying $n-1$ Givens rotations to intermediate matrices.

(II) Successively right-multiply $\mathbf{R}$ with the transposed Givens rotations from step (I), in the same order.

In order to achieve an optimal computational complexity $O(n)$ for $n \to \infty$ we must not operate on densely populated square matrices, but we have to store non-zero entries only, guided by the observation that all intermediate matrices in (0.3.4) possess at most four bands with non-zero entries, see also [Lecture → Thm. 3.3.3.27]:

$$\left(\mathbf{T}^{(k)}\right)_{i,j} = 0 \quad \text{, if} \quad i - j > 1 \quad \text{or} \quad j - i > 2\,. \tag{0.3.5}$$

Thus we can store the bands of $\mathbf{T}^{(k)}$ in the rows of a $4 \times n$ auxiliary matrix $\mathbf{U}$:

$$(\mathbf{U})_{i-j+3,j} := \left(\mathbf{T}^{(k)}\right)_{i,j} \quad \text{for} \quad i - j \in \{-2,-1,0,1\}\,,\quad j = 1,\dots,n:$$

▶ $$\mathbf{U} = \begin{bmatrix} 0 & 0 & t_{1,3} & t_{2,4} & t_{3,5} & \dots & t_{n-3,n-1} & t_{n-2,n} \\ 0 & t_{1,2} & t_{2,3} & t_{3,4} & t_{4,5} & \dots & t_{n-2,n-2} & t_{n-1,n} \\ t_{1,1} & t_{2,2} & t_{3,3} & t_{4,4} & t_{5,5} & \dots & t_{n-1,n-1} & t_{n,n} \\ t_{2,1} & t_{3,2} & t_{4,3} & t_{5,4} & t_{6,5} & \dots & t_{n,n-1} & 0 \end{bmatrix}, \quad t_{i,j} := \left(\mathbf{T}^{(k)}\right)_{i,j}.$$

The third row contains the diagonal of the matrix, the bottom row the first sub-diagonal. In the first step of (0.3.4) we apply $\mathbf{G}^1_{1,2}(\mathbf{T}^{(1)})$ to $\mathbf{T}^{(1)}$. This affects six entries of the matrix

$$\mathbf{G}^1_{1,2}(\mathbf{T}^{(1)}) \cdot \begin{bmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix},$$

which correspond to the following entries of $\mathbf{U}$, highlighted with color:

$$\begin{bmatrix} 0 & 0 & t_{1,3} & t_{2,4} & t_{3,5} & \dots & t_{n-3,n-1} & t_{n-2,n} \\ 0 & t_{1,2} & t_{2,3} & t_{3,4} & t_{4,5} & \dots & t_{n-2,n-2} & t_{n-1,n} \\ t_{1,1} & t_{2,2} & t_{3,3} & t_{4,4} & t_{5,5} & \dots & t_{n-1,n-1} & t_{n,n} \\ t_{2,1} & t_{3,2} & t_{4,3} & t_{5,4} & t_{6,5} & \dots & t_{n,n-1} & 0 \end{bmatrix}.$$

We write `givens` for a function computing the relevant parameters $c := \cos\varphi$ and $s := \sin\varphi$ of the Givens rotation. Then

$$\mathbf{G}^1_{1,2}(\mathbf{T}^{(1)}) = \begin{bmatrix} c & -s & 0 & \dots & 0 \\ s & c & & & \vdots \\ 0 & & 1 & & \\ \vdots & & & \ddots & \vdots \\ 0 & \dots & & \dots & 1 \end{bmatrix}, \quad \begin{bmatrix} c \\ s \end{bmatrix} = \mathtt{givens}\left(\begin{bmatrix} t_{11} \\ t_{21} \end{bmatrix}\right),$$

is the matrix effecting the Givens transformation in the first step, which amounts to the following changes in $\mathbf{U}$:

$$\begin{bmatrix} (\mathbf{U})_{3,1} \\ (\mathbf{U})_{4,1} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{3,1} \\ (\mathbf{U})_{4,1} \end{bmatrix}, \quad \begin{bmatrix} (\mathbf{U})_{2,2} \\ (\mathbf{U})_{3,2} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{2,2} \\ (\mathbf{U})_{3,2} \end{bmatrix}, \quad \begin{bmatrix} (\mathbf{U})_{1,3} \\ (\mathbf{U})_{2,3} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{1,3} \\ (\mathbf{U})_{2,3} \end{bmatrix}.$$

In the second step of (0.3.4) we multiply $\mathbf{T}^{(2)}$ from the left with $\mathbf{G}^2_{2,3}(\mathbf{T}^{(2)})$.

$$\mathbf{G}^1_{2,3}(\mathbf{T}^{(2)}) \cdot \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} .$$

Again, we have to operate on six entries, and for $\mathbf{U}$ the transformation affects the following entries

$$\begin{bmatrix} 0 & 0 & t_{1,3} & t_{2,4} & t_{3,5} & \dots & t_{n-3,n-1} & t_{n-2,n} \\ 0 & t_{1,2} & t_{2,3} & t_{3,4} & t_{4,5} & \dots & t_{n-2,n-2} & t_{n-1,n} \\ t_{1,1} & t_{2,2} & t_{3,3} & t_{4,4} & t_{5,5} & \dots & t_{n-1,n-1} & t_{n,n} \\ t_{2,1} & t_{3,2} & t_{4,3} & t_{5,4} & t_{6,5} & \dots & t_{n,n-1} & 0 \end{bmatrix},$$

and reads

$$\begin{bmatrix} (\mathbf{U})_{3,2} \\ (\mathbf{U})_{4,2} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{3,2} \\ (\mathbf{U})_{4,2} \end{bmatrix}, \quad \begin{bmatrix} (\mathbf{U})_{2,3} \\ (\mathbf{U})_{3,3} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{2,3} \\ (\mathbf{U})_{3,3} \end{bmatrix}, \quad \begin{bmatrix} (\mathbf{U})_{1,4} \\ (\mathbf{U})_{2,4} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{1,4} \\ (\mathbf{U})_{2,4} \end{bmatrix}.$$

Now a general pattern should have emerged: In the k-th step, $\mathbf{T}^{(k)} \to \mathbf{T}^{(k+1)}$, $k = 1, \dots, n-1$, we have to perform the following transformation of $\mathbf{U}$: For $k = 1, 2, \dots, n-1$

$$\begin{bmatrix} c \\ s \end{bmatrix} := \texttt{givens}\left(\begin{bmatrix} (\mathbf{U})_{3,k} \\ (\mathbf{U})_{4,k} \end{bmatrix} \right),$$

$$\begin{bmatrix} (\mathbf{U})_{3,k} \\ (\mathbf{U})_{4,k} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{3,k} \\ (\mathbf{U})_{4,k} \end{bmatrix}, \quad \begin{bmatrix} (\mathbf{U})_{2,k+1} \\ (\mathbf{U})_{3,k+1} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{2,k+1} \\ (\mathbf{U})_{3,k+1} \end{bmatrix},$$

$$\begin{bmatrix} (\mathbf{U})_{1,k+2} \\ (\mathbf{U})_{2,k+2} \end{bmatrix} \leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{1,k+2} \\ (\mathbf{U})_{2,k+2} \end{bmatrix} \text{, if } k < n-1 .$$

Eventually, $\mathbf{U}$ will encode the $\mathbf{R}$-factor of the QR-decomposition of $\mathbf{T}$:

$$\mathbf{U} = \begin{bmatrix} 0 & 0 & r_{1,3} & r_{2,4} & r_{3,5} & \dots & r_{n-3,n-1} & r_{n-2,n} \\ 0 & r_{1,2} & r_{2,3} & r_{3,4} & r_{4,5} & \dots & r_{n-2,n-2} & r_{n-1,n} \\ r_{1,1} & r_{2,2} & r_{3,3} & r_{4,4} & r_{5,5} & \dots & r_{n-1,n-1} & r_{n,n} \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix}, \quad r_{i,j} := (\mathbf{R})_{i,j} .$$

In stage (II) of the algorithm we have to compute

$$\mathbf{T}' := \mathbf{R} \left(\mathbf{G}^1_{1,2}\right)^\top \left(\mathbf{G}^2_{2,3}\right)^\top \cdot \dots \cdot \left(\mathbf{G}^{n-1}_{n-1,1}\right)^\top .$$

We find that right-multiplication of a matrix $\mathbf{M} \in \mathbb{R}^{n,n}$ with

$$\left(\mathbf{G}^{\ell}_{i,j}\right)^{\top} = \begin{bmatrix} 1 & 0 & \dots & & & & & & \dots & 0 \\ 0 & \ddots & \ddots & & & & & & & \vdots \\ \vdots & \ddots & 1 & 0 & & & & & & \\ & & 0 & c & 0 & \dots & 0 & s & & \\ & & & 0 & 1 & & & 0 & & \\ & & & \vdots & & \ddots & & \vdots & & \\ & & & 0 & & & 1 & 0 & & \\ & & & -s & 0 & \dots & 0 & c & 0 & \\ & & & & & & & 0 & 1 & \ddots & \vdots \\ \vdots & & & & & & & & \ddots & \ddots & 0 \\ 0 & \dots & & & & & & & \dots & 0 & 1 \end{bmatrix}, \quad i,j \in \{1,\dots,n\},\ i \neq j\,,$$

boils down to combining the i-th and j-th column of $\mathbf{M}$:

$$\left[(\mathbf{M})_{:,i} \quad (\mathbf{M})_{:,j}\right] \leftarrow \left[c(\mathbf{M})_{:,i} - s(\mathbf{M})_{:,j} \quad s(\mathbf{M})_{:,i} + c(\mathbf{M})_{:,j}\right].$$

So the first step of stage (II), right multiplication with the transpose of $\mathbf{G}^1_{1,2}(\mathbf{T}^{(1)})$, involves the two leftmost columns of $\mathbf{R}$:

$$\begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix} \left(\mathbf{G}^1_{1,2}\right)^{\top} = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix}.$$

The following entries of the matrix $\mathbf{U}$ are affected:

$$\begin{bmatrix} 0 & 0 & r_{1,3} & r_{2,4} & r_{3,5} & \dots & r_{n-3,n-1} & r_{n-2,n} \\ 0 & r_{1,2} & r_{2,3} & r_{3,4} & r_{4,5} & \dots & r_{n-2,n-2} & r_{n-1,n} \\ r_{1,1} & r_{2,2} & r_{3,3} & r_{4,4} & r_{5,5} & \dots & r_{n-1,n-1} & r_{n,n} \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix}$$

The next step combines the second and third column of the matrix:

$$\begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix} \left(\mathbf{G}^2_{2,3}\right)^{\top} = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ 0 & * & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix},$$

which impacts the following entries of $\mathbf{U}$:

$$\begin{bmatrix} 0 & 0 & r_{1,3} & r_{2,4} & r_{3,5} & \dots & r_{n-3,n-1} & r_{n-2,n} \\ 0 & r_{1,2} & r_{2,3} & r_{3,4} & r_{4,5} & \dots & r_{n-2,n-2} & r_{n-1,n} \\ r_{1,1} & r_{2,2} & r_{3,3} & r_{4,4} & r_{5,5} & \dots & r_{n-1,n-1} & r_{n,n} \\ r_{1,2} & 0 & 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix}.$$

The following step targets the third and fourth column:

$$\begin{bmatrix} \ast & \ast & \ast & 0 & 0 & 0 & 0 & 0 \\ \ast & \ast & \ast & \ast & 0 & 0 & 0 & 0 \\ 0 & \ast & \ast & \ast & \ast & 0 & 0 & 0 \\ 0 & 0 & 0 & \ast & \ast & \ast & 0 & 0 \\ 0 & 0 & 0 & 0 & \ast & \ast & \ast & 0 \\ 0 & 0 & 0 & 0 & 0 & \ast & \ast & \ast \\ 0 & 0 & 0 & 0 & 0 & 0 & \ast & \ast \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ast \end{bmatrix} \left(\mathbf{G}^2_{2,3}\right)^\top = \begin{bmatrix} \ast & \ast & \ast & \star & 0 & 0 & 0 & 0 \\ \ast & \ast & \ast & \ast & 0 & 0 & 0 & 0 \\ 0 & \ast & \ast & \ast & \ast & 0 & 0 & 0 \\ 0 & 0 & \ast & \ast & \ast & \ast & 0 & 0 \\ 0 & 0 & 0 & 0 & \ast & \ast & \ast & 0 \\ 0 & 0 & 0 & 0 & 0 & \ast & \ast & \ast \\ 0 & 0 & 0 & 0 & 0 & 0 & \ast & \ast \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ast \end{bmatrix}.$$

We make that bewildering observation that the transformation introduces an entry outside the four bands stored in $\mathbf{U}$! This is where Lemma 0.3.3 comes to the rescue. It tells us that the entry $\star$ must be zero (in exact arithmetic) and, thus, safely can be ignored. So in the next step, it is again only six entries that we have to deal with:

$$\begin{bmatrix} \ast & \ast & \ast & \mathbf{0} & 0 & 0 & 0 & 0 \\ \ast & \ast & \ast & \ast & 0 & 0 & 0 & 0 \\ 0 & \ast & \ast & \ast & \ast & 0 & 0 & 0 \\ 0 & 0 & \ast & \ast & \ast & \ast & 0 & 0 \\ 0 & 0 & 0 & 0 & \ast & \ast & \ast & 0 \\ 0 & 0 & 0 & 0 & 0 & \ast & \ast & \ast \\ 0 & 0 & 0 & 0 & 0 & 0 & \ast & \ast \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ast \end{bmatrix} \left(\mathbf{G}^2_{2,3}\right)^\top = \begin{bmatrix} \ast & \ast & \ast & 0 & 0 & 0 & 0 & 0 \\ \ast & \ast & \ast & \ast & \star & 0 & 0 & 0 \\ 0 & \ast & \ast & \ast & \ast & 0 & 0 & 0 \\ 0 & 0 & \ast & \ast & \ast & \ast & 0 & 0 \\ 0 & 0 & 0 & \ast & \ast & \ast & \ast & 0 \\ 0 & 0 & 0 & 0 & 0 & \ast & \ast & \ast \\ 0 & 0 & 0 & 0 & 0 & 0 & \ast & \ast \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ast \end{bmatrix}.$$

A pattern has emerged and suggests the following operations on $\mathbf{U}$: For $k = 1, 2, \dots, n-1$

$$\begin{aligned} \begin{bmatrix} (\mathbf{U})_{4,k-1} \\ (\mathbf{U})_{3,k} \end{bmatrix} &\leftarrow \begin{bmatrix} c^{(k)} & -s^{(k)} \\ s^{(k)} & c^{(k)} \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{4,k-1} \\ (\mathbf{U})_{3,k} \end{bmatrix}, \\ \begin{bmatrix} (\mathbf{U})_{3,k-1} \\ (\mathbf{U})_{2,k} \end{bmatrix} &\leftarrow \begin{bmatrix} c^{(k)} & -s^{(k)} \\ s^{(k)} & c^{(k)} \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{3,k-1} \\ (\mathbf{U})_{2,k} \end{bmatrix}, \\ \begin{bmatrix} (\mathbf{U})_{2,k-1} \\ (\mathbf{U})_{1,k} \end{bmatrix} &\leftarrow \begin{bmatrix} c^{(k)} & -s^{(k)} \\ s^{(k)} & c^{(k)} \end{bmatrix} \begin{bmatrix} (\mathbf{U})_{2,k-1} \\ (\mathbf{U})_{1,k} \end{bmatrix}, \end{aligned} \tag{0.3.6}$$

where $c^{(k)}$ and $s^{(k)}$ are the parameters of the Givens rotation used in the k-th step of (0.3.4). In the end, $(\mathbf{U})_{3,:}$ contains the diagonal entries of $\mathbf{T}'$ and $(\mathbf{U})_{4,1:n-1}$ provides the super- and sub-diagonal.

C++ code 0.3.7: Implementation of `qrStep()`

```cpp
std::pair<Eigen::VectorXd, Eigen::VectorXd> qrStep(const Eigen::VectorXd &dT,
                                                   const Eigen::VectorXd &uT) {
  const long n = dT.size();
  assert(uT.size() == n - 1);
  // Defining vectors for T' (Written as T_p)
  Eigen::VectorXd dT_p(n);
  Eigen::VectorXd uT_p(n - 1);

  // TO DO : Compute the defining vectors d(T') and u(T') of T':= RQ
      with O(n)
  // asymptotic complexity, given the defining vectors d(T) and U(T)
      of T = QR
  // START
  // Auxiliary 4xn matrix U ∈ R^{4,n}
  Eigen::MatrixXd U{Eigen::MatrixXd::Zero(4, n)};
  U.block(1, 1, 1, n - 1) = uT.transpose();
  U.block(2, 0, 1, n) = dT.transpose();
  U.block(3, 0, 1, n - 1) = uT.transpose();
  // Stage I: QR decomposition by n-1 Givens rotations
  std::vector<Eigen::Matrix2d> givensrots;  // Store rotations!
  for (long i = 0; i < n - 1; ++i) {
    const Eigen::Vector2d g = givens(Eigen::Vector2d(U(2, i), U(3, i)));
    const Eigen::Matrix2d G{
        (Eigen::Matrix2d(2, 2) << g[0], -g[1], g[1], g[0]).finished()};
    U.block(2, i, 2, 1).applyOnTheLeft(G);
    U.block(1, i + 1, 2, 1).applyOnTheLeft(G);
    if (i < n - 2) {
      U.block(0, i + 2, 2, 1).applyOnTheLeft(G);
    }
    givensrots.push_back(G);
  }
  // At this point U encodes the R-factor of the QR-decomposition of
  // T. In particular, the bottom row of U should contain
  // only entries that are "numerically zero".

  // Stage II: Apply transposed rotations from right to R according to
  // formulas (0.3.6)
  for (long i = 1; i < n; ++i) {
    const Eigen::Matrix2d &G{givensrots[i - 1]};  // An alias
    Eigen::Vector2d tmp;
    tmp = G * Eigen::Vector2d(U(3, i - 1), U(2, i));
    U(3, i - 1) = tmp[0];
    U(2, i) = tmp[1];
    tmp = G * Eigen::Vector2d(U(2, i - 1), U(1, i));
    U(2, i - 1) = tmp[0];
    U(1, i) = tmp[1];
    tmp = G * Eigen::Vector2d(U(1, i - 1), U(0, i));
    U(1, i - 1) = tmp[0];
    U(0, i) = tmp[1];
  }
  // Extract diagonal and sub-/super-diagonal of T'.'
  dT_p = U.block(2, 0, 1, n).transpose();
  uT_p = U.block(1, 1, 1, n - 1).transpose();
  // END

  return {dT_p, uT_p};
}
```

The EIGEN method `applyOnTheLeft()` performs in-situ left-multiplication with a matrix. Of course, the code uses C++ indexing!

Obviously, for the implementation of Code 0.3.7 a call to `qrStep()` for n-vectors can be processed with $O(n)$ elementary operations for $n \to \infty$.

▲

References

[GV13] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Fourth. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, 2013, pp. xiv+756 (cit. on p. 20).

End Problem 0-3 , 40 pts.