

ETH Lectures 401-0663-00L Numerical Methods for Computer Science  
401-2673-00L Numerical Methods for CSE

# Homework Problems and Projects

Prof. R. Hiptmair, SAM, ETH Zurich

Autumn Term 2021, Version of October 2, 2022  
(C) Seminar für Angewandte Mathematik, ETH Zürich

[Link](#) to the current version of this homework collection

## 0.1 General Information

### 0.1.1 Weekly Homework Assignments

All problems will be published in this single “.pdf” file. Every week, we publish a list of problem numbers on the lecture Moodle page. These are the current assignments for that week. Details are given on the lecture Moodle page.

### 0.1.2 Importance of Homework

Homework assignments are **not** mandatory. However, it is **very important** that you constantly exercise with the material you learn. “Solving” the homework assignments one week before the main exam by looking at the solutions will likely result in failure.

We provide hints and solutions from the beginning. It is your responsibility to look at the solutions only if you are stuck with a difficult problem and you tried to find a solution for a sufficient amount of time.

Most homework problems come with an estimate for the time required for its solution. These times are what we expect a bright and well-prepared student to require for that problem during an exam.

### 0.1.3 Corrections and Grading of Assignments

Homework assignments will not be graded.

However, you may submit your handwritten solutions by handing them to your tutor directly or uploading them via the Moodle upload interface, see the lecture Moodle page for instructions. The assistants will

**Problem 3-4: QR-Factorization of Tridiagonal Matrices**

In [Lecture → Rem. 3.3.3.26] we saw that a QR-factorization of a tri-diagonal matrix features an upper triangular factor that is tridiagonal again. In this exercise we exploit this fact for the design of efficient algorithms, in particular for the stable solution of tridiagonal linear systems of equations.

This exercise assume familiarity with the concept of QR-decomposition/factorization [Lecture → Section 3.3.3] and, in particular, Givens rotations [Lecture → § 3.3.3.15] and their compressed storage [Lecture → Rem. 3.3.3.21].

Throughout this problem, in order to be compatible with some software, for generic real-valued square tridiagonal matrices we are supposed to use the data structure

```
#include <Eigen/Dense>
struct TriDiagonalMatrix {
    Eigen::Index n; // Matrix size: n × n
    Eigen::VectorXd d; // n-vector of diagonal entries
    Eigen::VectorXd l; // n-1-vector, entries of first lower diagonal
    Eigen::VectorXd u; // n-1-vector, entries of first upper diagonal
    // Simple constructor
    TriDiagonalMatrix(const Eigen::VectorXd &d,
        const Eigen::VectorXd &l,
        const Eigen::VectorXd &u)
    : n(d.size()), d(d), l(l), u(u) {
        assert((n-1) == l.size() && (n-1) == u.size());
    }
};
```

Note that in full-fledged C++ implementations, this would be a C++ class with four different constructors and a destructor explicitly defined, private data members, and access member functions. The rudimentary definition above is proposed solely for the sake of simplicity.

Let us write  $\mathbf{d} \in \mathbb{R}^n$ ,  $\mathbf{l} \in \mathbb{R}^{n-1}$ , and  $\mathbf{u} \in \mathbb{R}^{n-1}$  for the vectors stored in `d`, `l`, `u`. Then the matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$  stored in an `TriDiagonalMatrix` object is defined by

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

$$\Leftrightarrow \mathbf{A} = \begin{bmatrix} d_1 & u_1 & & \\ l_1 & d_2 & \ddots & \\ & \ddots & \ddots & u_{n-1} \\ & & l_{n-1} & d_n \end{bmatrix}, \quad \begin{aligned} \mathbf{d} &= [d_1, \dots, d_n]^\top \in \mathbb{R}^n, \\ \mathbf{u} &= [u_1, \dots, u_{n-1}]^\top \in \mathbb{R}^{n-1}, \\ \mathbf{l} &= [l_1, \dots, l_{n-1}]^\top \in \mathbb{R}^{n-1}. \end{aligned}$$

**(3-4.a)**  (30 min.) Prove that for a *tridiagonal* invertible matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$  a QR-factorization  $\mathbf{A} = \mathbf{Q}\mathbf{R}$ ,  $\mathbf{Q} \in \mathbb{R}^{n,n}$  orthogonal,  $\mathbf{R} \in \mathbb{R}^{n,n}$  upper triangular with positive diagonal entries, satisfies

$$(\mathbf{R})_{i,j} = 0 \quad \text{for } j > i + 2 \quad \text{and} \quad (\mathbf{Q})_{i,j} = 0 \quad \text{for } i > j + 1. \quad (3.4.1)$$

**HINT 1** for (3-4.a): There is a direct proof just employing the rules for multiplying matrices. First establish the property of  $\mathbf{Q} = \mathbf{A}\mathbf{R}^{-1}$  and then that of  $\mathbf{R}$ . The following result is useful:

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

$$\mathbf{A}, \mathbf{B} \begin{cases} \text{diagonal} \\ \text{upper triangular} \\ \text{lower triangular} \end{cases} \Rightarrow \mathbf{AB} \text{ and } \mathbf{A}^{-1} \begin{cases} \text{diagonal} \\ \text{upper triangular} \\ \text{lower triangular} \end{cases} .$$

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

SOLUTION of (3-4.a):

The assertion (3.4.1) can be proved in different ways:

I. Based on the hint we give a direct proof based on the rules of matrix multiplication, assuming that  $\mathbf{A}$  is invertible.

First, we rely the observation made in [Lecture → Lemma 1.3.1.9] that the inverse of a regular upper triangular matrix is again upper triangular and apply it to  $\mathbf{R}$ :  $\mathbf{R}^{-1}$  is upper triangular. Thus, writing  $a_{i,j} := (\mathbf{A})_{i,j}$  ( $\Rightarrow a_{i,j} = 0$ , if  $|i - j| > 1$ ) and  $s_{i,j} := (\mathbf{R}^{-1})_{i,j}$  ( $\Rightarrow s_{i,j} = 0$  for  $i > j$ ), we find

$$\begin{aligned} (\mathbf{Q})_{j,k} &= \sum_{0 \leq l \leq n} a_{j,l} s_{l,k} = \sum_{l \in \{j-1, j, j+1\}} a_{j,l} s_{l,k} \quad , \text{ because } \mathbf{A} \text{ is tridiagonal,} \\ &= \sum_{l \in \{j-1, j, j+1\}, l \leq k} a_{j,l} s_{l,k} \quad , \text{ because } \mathbf{R}^{-1} \text{ is upper triangular,} \\ &= 0 \quad , \text{ if } j > k+1, \quad j, k \in \{1, \dots, n\} . \end{aligned} \quad (3.4.2)$$

Next,  $\mathbf{R} = \mathbf{Q}^\top \mathbf{A}$  is upper triangular by definition and ( $q_{i,j} := (\mathbf{Q})_{i,j}$ ,  $q_{i,j} = 0$ , if  $i > j+1$ )

$$\begin{aligned} (\mathbf{R})_{j,k} &= \sum_{0 \leq l \leq n} (\mathbf{Q}^\top)_{j,l} a_{l,k} = \sum_{0 \leq l \leq n} q_{l,j} a_{l,k} \\ &= \sum_{l \in \{k-1, k, k+1\}} q_{l,j} a_{l,k} \quad , \text{ because } \mathbf{A} \text{ is tridiagonal,} \\ &= \sum_{l \in \{k-1, k, k+1\}, l \leq j+1} q_{l,j} a_{l,k} \quad , \text{ because of the pattern of } \mathbf{Q}, \\ &= 0 \quad , \text{ if } j+1 < k-1 \quad , \text{ i.e., if } k > j+2, \quad j, k \in \{1, \dots, n\} . \end{aligned} \quad (3.4.3)$$

In words, if  $k > j+2$  the non-zero entries in the  $k$ -th column of  $\mathbf{A}$  meet only the trailing zeros in the  $j$ -th row of  $\mathbf{Q}^\top$ :  $\mathbf{Q}^\top \mathbf{A} =$

$$\begin{bmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ * & * & * & * & * & 0 & 0 & 0 \\ * & * & * & * & * & * & 0 & 0 \\ * & * & * & * & * & * & * & 0 \\ * & * & * & * & * & * & * & * \end{bmatrix} \begin{bmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 & 0 & 0 \\ * & * & * & * & * & 0 & 0 & 0 \\ * & * & * & * & * & * & 0 & 0 \\ * & * & * & * & * & * & * & 0 \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \end{bmatrix} .$$

Here the entries marked with  $*$  are zero by the very definition of the QR-decomposition, not by the rules of matrix multiplication.

II. We can exploit the algebraic equivalence of QR-decomposition and Gram-Schmidt orthonormalization explained in [Lecture → Section 3.3.3.1]. [Lecture → Thm. 3.3.3.2] tells us that the spans of the  $k$  leftmost columns of  $\mathbf{Q}$  and  $\mathbf{A}$  are the same,

$$\text{Span}\{(\mathbf{Q})_{:,1}, \dots, (\mathbf{Q})_{:,k}\} = \text{Span}\{(\mathbf{A})_{:,1}, \dots, (\mathbf{A})_{:,k}\}, \quad (3.4.4)$$

because  $\mathbf{A}$  has full rank  $n$ , since it is regular. Obviously, for a tridiagonal matrix  $\mathbf{A}$  we have

$$\text{Span}\{(\mathbf{A})_{:,1}, \dots, (\mathbf{A})_{:,k}\} \subset \text{Span}\{\mathbf{e}_1, \dots, \mathbf{e}_k\} = \left\{ \begin{bmatrix} \xi_1 \\ \vdots \\ \xi_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \xi_j \in \mathbb{R} \right\}. \quad (3.4.5)$$

Combining (3.4.4) and (3.4.5) gives the assertion on the pattern of  $\mathbf{Q}$ . Then proceed as in (3.4.3).

III. Alternatively, the proof that for the upper triangular factor  $\mathbf{R}$  only the zeroth, first, and second upper diagonal are occupied can be constructive and employ  $n - 1$  successive Givens rotations, see [Lecture → Rem. 3.3.3.26]. Applying these Givens rotations to an identity matrix reveals the structure of  $\mathbf{Q}^\top$ .



To store a QR-factorization  $\mathbf{A} = \mathbf{Q}\mathbf{R}$  of a real-valued tridiagonal matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$  we use the (again rudimentary) data structure

```
class TriDiagonalQR {
public:
    explicit TriDiagonalQR(const TriDiagonalMatrix &A);

    template <typename VecType>
    Eigen::VectorXd applyQT(const VecType &x) const;
    template <typename VecType>
    Eigen::VectorXd solve(const VecType &b) const;
    // For debugging purposes: extract factors as dense matrices
    std::pair<Eigen::MatrixXd, Eigen::MatrixXd> getQRFactors(void)
        const;

private:
    Eigen::Index n; // size of the square matrix
    Eigen::Matrix B; // Three non-zero upper diagonals of R
    Eigen::VectorXd rho; // Encoded Givens rotations
};
```

Here the non-zero entries of  $\mathbf{R}$  are stored in an  $n \times 3$ -matrix  $\mathbf{B}$  adopting the convention

$$(\mathbf{R})_{i,j} := \begin{cases} (\mathbf{B})_{i,1} & , \text{ if } i = j, \\ (\mathbf{R})_{i,j} = (\mathbf{B})_{i,2} & , \text{ if } j = i + 1, \\ (\mathbf{R})_{i,j} = (\mathbf{B})_{i,3} & , \text{ if } j = i + 2, \\ 0 & \text{ else,} \end{cases} \quad i, j \in \{1, \dots, n\}, \quad (3.4.6)$$

that is,  $\mathbf{R} = \begin{bmatrix} x_1 & y_1 & z_1 & & & \\ & x_2 & y_2 & z_2 & & \\ & & x_3 & y_3 & z_3 & \\ & & & \ddots & \ddots & \ddots \\ & & & & x_{n-2} & y_{n-2} & z_{n-2} \\ & & & & & x_{n-1} & y_{n-1} \\ & & & & & & x_n \end{bmatrix} \Rightarrow \mathbf{B} = \begin{bmatrix} x_1 & y_1 & z_1 \\ \vdots & \vdots & \vdots \\ x_{n-2} & y_{n-2} & z_{n-2} \\ x_{n-1} & y_{n-1} & 0 \\ x_n & 0 & 0 \end{bmatrix}.$

Note that three entries of  $\mathbf{B}$  will never be used.

The  $\mathbf{Q}$ -factor is stored in compressed format through the underlying  $n - 1$  Givens rotations whose target rows are clear a priori, see [Lecture → Rem. 3.3.3.26]. Every Givens rotation is stored through a single number  $\rho$  following the convention described in [Lecture → Rem. 3.3.3.21]. These  $n - 1$  numbers are collected in the data member `rho`. Its  $j$ -th entry corresponds to the Givens rotation acting on rows  $j$  &  $j + 1$ .

(3-4.b) 🕒 (20 min.) Implement a C++ function

```
std::tuple<double, double, double>
compGivensRotation(Eigen::Vector2d a);
```

that computes a Givens rotation mapping the vector  $\mathbf{a} \in \mathbb{R}^2$  onto a multiple of the first Cartesian coordinate vector  $\mathbf{e}_1 \in \mathbb{R}^2$ . Use the algorithm implemented in [Lecture → Code 3.3.3.17]. The function is to return the triplet  $(\rho, \gamma, \sigma)$ , where  $\mathbf{G} = \begin{bmatrix} \gamma & \sigma \\ -\sigma & \gamma \end{bmatrix}$  is the transformation matrix such that  $\mathbf{G}^T \mathbf{a} = (\pm \|\mathbf{a}\|, 0)^T$ , and  $\rho \in \mathbb{R}$  encodes it according to the rule set

$$\rho := \begin{cases} \text{sign}(\sigma) & , \text{ if } \gamma = 0, \\ \sqrt{2} \text{sign}(\gamma) \sigma & , \text{ if } |\sigma| < |\gamma|, \\ \frac{1}{2} \sqrt{2} \text{sign}(\sigma) / \gamma & , \text{ if } |\sigma| \geq |\gamma|. \end{cases} \quad [\text{Lecture} \rightarrow \text{Eq. (3.3.3.22)}]$$

SOLUTION of (3-4.b):

The first part of the implementation amounts to copying the code of [Lecture → Code 3.3.3.17]. Then use the formulas from [Lecture → Rem. 3.3.3.21] to define the code  $\rho$  depending on the entries of the rotation matrix, see [Lecture → Eq. (3.3.3.22)].

#### C++ code 3.4.7: Computing and encoding a Givens rotation

```
2 std::tuple<double, double, double> compGivensRotation(Eigen::Vector2d a) {
3     double rho, gamma, sigma;
4     // TODO (3-4.b): Compute the parameters of a rotation that when
5     // applied to a results in the vector  $(\|\mathbf{a}\|, 0)^T$  or  $(-\|\mathbf{a}\|, 0)^T$ .
6     // Hint: You can use the function sign() above when determining rho
7     // from gamma and sigma.
8     // START
9     if (a(1) != 0.0) {
10        double t;
11        // Separate cases to avoid cancellation/overflow
12        if (std::abs(a(1)) > std::abs(a(0))) {
13            t = -a(0) / a(1);
14            sigma = 1.0 / std::sqrt(1.0 + t * t);
15            gamma = sigma * t;
16        } else { // Then abs(a(0)) >= abs(a(1)) > 0
```

```

17     t = -a(1) / a(0); // a(0) != 0
18     gamma = 1.0 / std::sqrt(1.0 + t * t);
19     sigma = gamma * t;
20 }
21 } else {
22     // No rotation.
23     gamma = 1.0;
24     sigma = 0.0;
25 }
26 // Determine rho
27 if (gamma == 0.0)
28     rho = sign(sigma);
29 else if (std::abs(gamma) > std::abs(sigma))
30     rho = std::sqrt(2.0) * sign(gamma) * sigma;
31 else
32     rho = 0.5 * std::sqrt(2.0) * sign(sigma) / gamma;
33 // END
34 return std::make_tuple(rho, gamma, sigma);
35 }

```

Get it on  GitLab (tridiagqr.hpp).

The function `sign()` just returns the sign of a number:

```

template <typename T>
int sign(T val) {
    return (T(0) < val) - (val < T(0));
}

```



### (3-4.c) (45 min.) [ depends on [Lecture → Rem. 3.3.3.26] ]

Write an *efficient* code for the constructor of **TriDiagonalQR**, which initializes the data members, that is, computes the QR-factorization of the tridiagonal matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$  passed to it in `A`.

HINT 1 for (3-4.c): First copy the data from `A` to `B` and then perform in-situ (in-place) transformations on `B`. ┐

HINT 2 for (3-4.c): An implementation, which uses a different data structure, however, can be found in the first part of [Lecture → Code 3.3.4.5]. ┐

SOLUTION of (3-4.c):

We employ  $n - 1$  Givens rotations targeting the rows

$$(1, 2) \rightarrow (2, 3) \rightarrow \dots \rightarrow (n - 1, n).$$

The challenge is to figure out, which entries of **B** are affected by which Givens rotations.

Outline of the transformation:

$$\begin{bmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} \xrightarrow{G_{12}^\top} \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} \xrightarrow{G_{n-1,n}^\top \cdots G_{23}^\top} \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix}$$

\*  $\hat{=}$  entry set to zero by Givens rotation, \*  $\hat{=}$  new non-zero entry

### C++ code 3.4.8: Constructor of **TriDiagonalQR**

```

2  TriDiagonalQR::TriDiagonalQR(const TriDiagonalMatrix &A) : n(A.n) {
3      rho = Eigen::VectorXd::Zero(n - 1);
4      B = Eigen::MatrixXd::Zero(n, 3);
5      B.col(0) = A.d;
6      B.col(1).tail(n - 1) = A.l;
7      B.col(2).head(n - 1) = A.u;
8      // Currently we have for k=0,...,n-1:
9      // B(k,0) = A(k,k), B(k,1) = A(k,k-1), B(k,2) = A(k,k+1).
10
11     // A bit clumsy, but less error prone than juggling indices
12     Eigen::MatrixXd Ablock(2, 3);
13     double gamma, sigma;
14     for (unsigned int k = 0; k < n - 1; ++k) {
15         // The block of the current A that will be affected by the next
16         // rotation.
17         Ablock << B(k, 0), B(k, 2), 0, B(k + 1, 1), B(k + 1, 0), B(k + 1, 2);
18         // TODO (3-4.c): Find the k-th Givens rotation and update rho and
19         // B.
20         // START
21         std::tuple<double, double, double> params =
22             compGivensRotation(Ablock.col(0));
23         rho[k] = std::get<0>(params); // Store rotation parameter
24         gamma = std::get<1>(params);
25         sigma = std::get<2>(params);
26
27         // Update B
28         Eigen::Matrix2d G;
29         G << gamma, -sigma, sigma, gamma;
30         Ablock = G * Ablock; // Rotate to eliminate Ablock(1,0)
31         B.row(k) = Ablock.row(0);
32         B(k + 1, 0) = Ablock(1, 1);
33         B(k + 1, 2) = Ablock(1, 2);
34         // END
35     }
36 }

```

Get it on  GitLab (tridiagqr.hpp).

Note that a more compact way to handle multiple return values is presented in [Lecture → Code 0.3.4.3].



(3-4.d)  (30 min.)

Provide an *efficient* implementation of the method `applyQT()` of **TriDiagonalQR** that computes  $Q^\top x$  for a supplied vector  $x \in \mathbb{R}^n$  (passed in `x`).

The type **VecType** must describe a vector with real-valued entries supplying component access through **operator [] const** and a **size()** method.

HINT 1 for (3-4.d): Look up [Lecture  $\rightarrow$  Eq. (3.3.3.23)] to understand how to recover the rotation from the code  $\rho$ . ┘

SOLUTION of (3-4.d):

The following code reconstructs the rotation matrix from  $\rho$ .

#### C++ code 3.4.9: Decoding compact storage of Givens matrices

```

2  inline Eigen::Matrix2d Givens(double rho) {
3      double gamma, sigma;
4      if (std::abs(rho) < 1.0) {
5          sigma = 0.5 * std::sqrt(2.0) * rho;
6          gamma = std::sqrt(1.0 - sigma * sigma);
7      } else if (std::abs(rho) > 1.0) {
8          gamma = 0.5 * std::sqrt(2.0) / rho;
9          sigma = std::sqrt(1 - gamma * gamma);
10     } else {
11         gamma = 0.0;
12         sigma = sign(rho);
13     }
14     Eigen::Matrix2d G;
15     G << gamma, sigma, -sigma, gamma;
16     return G;
17 }

```

Get it on  GitLab (tridiagqr.hpp).

This code applies the  $n - 1$  rotations encoded in `rho` to consecutive entries of the vector `x`.

#### C++ code 3.4.10: Method `applyQT()` of **TriDiagonalQR**

```

2  template <typename VecType>
3  Eigen::VectorXd TriDiagonalQR::applyQT(const VecType &x) const {
4      assert(n == x.size() &&
5             "Input must have size equal to the number of rows in Q.");
6      Eigen::VectorXd y(n);
7      // TODO (3-4.d): Fill in the entries of y.
8      // START
9      for (unsigned int k = 0; k < n; ++k) {
10         y[k] = x[k];
11     }
12     Eigen::Matrix2d G;
13     for (unsigned int k = 0; k < n - 1; ++k) {
14         G = Givens(rho[k]).transpose();
15         y.segment(k, 2) = G * y.segment(k, 2);
16     }
17     // END
18     return y;
19 }

```

Get it on  GitLab (tridiagqr.hpp).



**(3-4.e)** 🕒 (30 min.) [ depends on Sub-problem (3-4.d) and [Lecture → Rem. 3.3.4.3] ]

Devise an efficient implementation of the method `solve()` of **TriDiagonalQR** that accepts a vector  $\mathbf{b} \in \mathbb{R}^n$  as arguments and solves the linear systems of equations  $\mathbf{Ax} = \mathbf{b}$ ,  $\mathbf{A} \in \mathbb{R}^{n,n}$  the tridiagonal matrix stored in the current **TriDiagonalQR** object, returning the solution  $\mathbf{x} \in \mathbb{R}^n$ . A `std::runtime_error` exception should be thrown in case the matrix  $\mathbf{A}$  is not invertible.

HINT 1 for (3-4.e): You may get some inspiration from the second part of [Lecture → Code 3.3.4.5].

SOLUTION of (3-4.e):

- First step:  $\mathbf{c} = \mathbf{Q}^T \mathbf{b}$  by `applyQT()`
- Second step: back-substitution for a tridiagonal matrix

**C++ code 3.4.11: Method `solve` of **TriDiagonalQR****

```

2  template <typename VecType>
3  Eigen::VectorXd TriDiagonalQR::solve(const VecType &b) const {
4      assert(n == b.size() &&
5              "Input must have size equal to the number of rows in Q.");
6      Eigen::VectorXd x(n);
7      // TODO (3-4.e): Solve the linear system A*x = b if A is invertible,
8      // otherwise throw an std::runtime_error.
9      // START
10     if ((B.col(0).array().abs() < 1E-10).any()) {
11         throw std::runtime_error("A is not invertible!");
12     }
13     // Multiply by Q^T on both sides of Q*R*x=b:
14     // R*x = Q^T*b = c
15     Eigen::VectorXd c = applyQT(b);
16
17     // Back-substitution
18     int k = n - 1;
19     x(k) = c(k) / B(k, 0);
20     k--;
21     x(k) = (c(k) - B(k, 1) * x(k + 1)) / B(k, 0);
22     k--;
23     while (k >= 0) {
24         x(k) = (c(k) - B(k, 1) * x(k + 1) - B(k, 2) * x(k + 2)) / B(k, 0);
25         k--;
26     }
27     // END
28     return x;
29 }

```

Get it on 🐙 [GitLab](#) (`tridiagqr.hpp`).

**(3-4.f)** 🕒 (10 min.) [ depends on Sub-problem (3-4.b), Sub-problem (3-4.d), Sub-problem (3-4.e) ]

What is the asymptotic computational effort for your implementations of the constructor of **TriDiagonalQR** and of the methods `applyQT()` and `solve()` in terms of  $n \rightarrow \infty$ ?

SOLUTION of (3-4.f):

All functions/methods enjoy the asymptotic complexity  $O(n)$  for  $n \rightarrow \infty$  as they all involve a single loop.



**(3-4.g)** (30 min.) The following templated C++ function implements (in a non-optimal way) a so-called **inverse iteration** [Lecture → Section 9.3.2].

```
template <typename MatrixType>
unsigned int invit(const MatrixType &A, Eigen::VectorXd &x,
double TOL = 1E-6, unsigned int maxit = 100) {
    x.normalize();
    Eigen::VectorXd x_old(x.size());
    int it = 0;
    do {
        x_old = x;
        x = A.lu().solve(x_old);
        x.normalize();
    }
    while (((x-x_old).norm() > TOL) && (it++ < maxit));
    return it;
}
```

Implement a **specialization** ( C++ reference) of this function for **MatrixType = TriDiagonalMatrix**.

HINT 1 for (3-4.g): Take into account [Lecture → Rem. 2.5.0.10] for the sake of efficient implementation.

SOLUTION of (3-4.g):

For the sake of efficiency it is important to compute the QR-decomposition outside the loop, recall [Lecture → Rem. 2.5.0.10]. This is done in the constructor of **TriDiagonalQR**.

#### C++ code 3.4.12: Implementation of `invit()` for **TriDiagonalMatrix**

```
2  template <>
3  unsigned int invit<TriDiagonalMatrix>(const TriDiagonalMatrix &A,
4                                         Eigen::VectorXd &x, double TOL,
5                                         unsigned int maxit) {
6      unsigned int it = 0;
7      // TODO (3-4.g): Specialize (efficiently) the template function invit()
8      // for the TriDiagonalMatrix class.
9      // START
10     x.normalize();
11     Eigen::VectorXd x_old(x.size());
12     TriDiagonalQR Aqr(A);
13     do {
14         x_old = x;
15         x = Aqr.solve(x_old);
16         x.normalize();
17     } while (((x - x_old).norm() > TOL) && (it++ < maxit));
18     // END
19     return it;
20 }
```

Get it on 🐙 [GitLab \(tridiagqr.hpp\)](https://github.com/tridiagqr).

**Remark.** Computing the QR-decomposition outside the loop does not make a difference in terms of asymptotic complexity, because it merely requires  $O(n)$  operations.

---



End Problem 3-4 , 195 min.