

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 4-6: Asymptotic Cost of EIGEN-Based Functions

This short problem is about reading off the asymptotic computational cost of some C++ functions performing numerical linear algebra tasks based on EIGEN. It relies on information provided in [Lecture → Section 1.4.2], [Lecture → § 2.5.0.4], [Lecture → § 3.3.3.37], and [Lecture → Section 4.3].

Just reading of C++ code is required.

(4-6.a) 🕒 (8 min.) The function `slvtriag()` listed in Code 4.6.1 expects two vector arguments of equal length $n \in \mathbb{N}$. What is its asymptotic computational complexity for $n \rightarrow \infty$?

$$\text{cost}(\text{slvtriag}) = O(\boxed{}) \quad \text{for } n \rightarrow \infty.$$

C++ code 4.6.1: Function `slvtriag()`

```

2 Eigen::VectorXd slvtriag(const Eigen::VectorXd &u, const Eigen::VectorXd &v) {
3   assert((u.size() == v.size()) && "Size mismatch");
4   const Eigen::MatrixXd A{u * v.transpose()};
5   return A.triangularView<Eigen::Upper>().solve(u);
6 }

```

SOLUTION of (4-6.a):

- The function involves forming the tensor product of two vectors of length n , which created a densely populated $n \times n$ -matrix at asymptotic cost of $O(n^2)$ for $n \rightarrow \infty$
- The solution of a *triangular* linear system of size n amounts to a simple elimination step with asymptotic cost $O(n^2)$ for $n \rightarrow \infty$, see [Lecture → § 2.3.2.15].

Hence the overall asymptotic complexity of `slvtriag()` is $O(n^2)$ for $n \rightarrow \infty$.

Remark. The function computes

$$\text{slvtriag}(\mathbf{u}, \mathbf{v}) = \text{triu}(\mathbf{u}\mathbf{v}^\top)^{-1}\mathbf{u}.$$



(4-6.b) 🕒 (8 min.) What is the asymptotic complexity of the C++ function `slvsymrom()` listed as Code 4.6.2 in terms of the length n of its argument vector $\mathbf{u}$?

$$\text{cost}(\text{slvsymrom}) = O(\boxed{}) \quad \text{for } n \rightarrow \infty.$$

A sharp bound is expected.

C++ code 4.6.2: Function `slvsymrom()`

```

2 Eigen::VectorXd slvsymrom(const Eigen::VectorXd &u) {
3   const unsigned int n = u.size();
4   return (Eigen::MatrixXd::Identity(n, n) + u * u.transpose()).lu().solve(u);

```

5 }

SOLUTION of (4-6.b):

A call to `slvsymrom()` involves the solution of a dense $n \times n$ linear system of equations by means of LU-decomposition and subsequent eliminations. The most costly step is the computation of the LU-decomposition with cost $O(n^3)$ for $n \rightarrow \infty$ [Lecture $\rightarrow$ Eq. (2.3.2.10)]. This step determines the asymptotic computational complexity of the function.

Remark. The result of the function is

$$\text{slvsymrom}(\mathbf{u}) = (\mathbf{I} + \mathbf{u}\mathbf{u}^\top)^{-1}\mathbf{u}.$$



(4-6.c) 🕒 (12 min.) Code 4.6.3 lists the C++ function `getcompbas()`, whose argument $\mathbf{A}$ is a densely populated matrix $\mathbf{A} \in \mathbb{R}^{n,k}$, $k < n$. Give a sharp asymptotic bound for the asymptotic computational cost of `getcompbas()` for $n \rightarrow \infty$ assuming k to be small and fixed.

$$\text{cost}(\text{getcompbas}) = O(\boxed{}) \quad \text{for } n \rightarrow \infty.$$

C++ code 4.6.3: Function `getcompbas()`

```

2  Eigen::MatrixXd getcompbas(const Eigen::MatrixXd &A) {
3      const int n = A.rows();
4      const int k = A.cols();
5      Eigen::HouseholderQR<Eigen::MatrixXd> qr(A);
6      return qr.householderQ() *
7          (Eigen::MatrixXd(n, n - k) << Eigen::MatrixXd::Zero(k, n - k),
8           Eigen::MatrixXd::Identity(n - k, n - k))
9           .finished();
10 }
```

HINT 1 for (4-6.c): Note that the constructor of `Eigen::HouseholderQR<>` computes an economical QR-factorization stored in compressed format. The call `qr.householderQ() *` just applies Householder transformations.

SOLUTION of (4-6.c):

According to [Lecture $\rightarrow$ § 3.3.3.37] the computation of the (thin, which is the default in EIGEN) QR-decomposition of $\mathbf{A} \in \mathbb{R}^{n,k}$ incurs an asymptotic computational effort of $O(nk^2)$ for $n, k \rightarrow \infty$. Since k is small and fixed, this means $O(n)$ computational cost. Note that the Q-factor is never computed as a dense $n \times n$ -matrix, but stored in compressed format as discussed in [Lecture $\rightarrow$ Rem. 3.3.3.21].

In the last step the Q-factor is multiplied with a matrix $\mathbf{M} := \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix}$ of size $n \times (n - k)$, resulting in a matrix of the same size. What is going on internally is that k Householder transformations are applied to $\mathbf{M}$, which creates a dense $n \times (n - k)$ -matrix with $O(n^2)$ entries. This will take $O(n^2)$ operations for $n \rightarrow \infty$ and this determines the overall asymptotic cost of the function.

Remark. If $\mathbf{A}$ has full rank, the function `getcompbas()` returns an orthonormal basis of the orthogonal complement of $\mathcal{R}(\mathbf{A})$ in the columns of the result matrix.



(4-6.d) 🕒 (12 min.) Code 4.6.4 shows the implementation of the C++ function `fft2_square()`, which requires a complex $n \times n$ -matrix $\mathbf{Y} \in \mathbb{C}^{n,n}$ as argument $\mathbf{Y}$. What is a sharp bound on the asymptotic complexity of `fft2_square()` as $n \rightarrow \infty$?

$$\text{cost}(\text{fft2_square}) = O(\boxed{}) \quad \text{for } n \rightarrow \infty.$$

C++ code 4.6.4: Function `fft2_square()`

```

2  Eigen::MatrixXcd fft2_square(const Eigen::MatrixXcd &Y) {
3      const unsigned int n = Y.rows();
4      assert((n == Y.cols()) && "Matrix Y must be square");
5      Eigen::MatrixXcd tmp(n, n);
6      Eigen::FFT<double> fft;
7      for (int k = 0; k < n; ++k) {
8          const Eigen::VectorXcd tv(Y.row(k));
9          tmp.row(k) = fft.fwd(tv).transpose();
10     }
11     for (int k = 0; k < n; ++k) {
12         const Eigen::VectorXcd tv(tmp.col(k));
13         tmp.col(k) = fft.fwd(tv);
14     }
15     return tmp;
16 }
```

SOLUTION of (4-6.d):

The function executes $2n$ discrete Fourier transforms of length n . Using the FFT algorithm the cost of each will be $O(n \log n)$ [Lecture $\rightarrow$ § 4.3.0.3], which leads to a total computational effort of $O(n^2 \log n)$ for $n \rightarrow \infty$

Remark. The code performs the 2D DFT discussed in [Lecture $\rightarrow$ Section 4.2.5].



End Problem 4-6, 40 min.