

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 2-10: Rank-one perturbations

This exercise centers around an application of the Sherman-Morrison-Woodbury formula (see [Lecture → Lemma 2.6.0.21]). Please carefully revise [Lecture → § 2.6.0.12] in the lecture notes.

Linked to [Lecture → Rem. 2.5.0.10] and [Lecture → § 2.6.0.12], related problem Problem 2-5

Consider the following pseudocode:

Algorithm 2.10.1: Code `rankoneinvit`

Require: $\mathbf{d} \in \mathbb{R}^n, \text{tol} \in \mathbb{R}, \text{tol} > 0$

Ensure: $l_{\min}$

$\mathbf{ev} \leftarrow \mathbf{d}; l_{\min} \leftarrow 0; l_{\text{new}} \leftarrow \min|\mathbf{d}|;$

while $|l_{\text{new}} - l_{\min}| > \text{tol} \cdot l_{\min}$ **do**

$l_{\min} \leftarrow l_{\text{new}};$

$\mathbf{M} \leftarrow \text{diag}(\mathbf{d}) + \mathbf{ev} \cdot \mathbf{ev}^\top;$

$\mathbf{ev} \leftarrow \mathbf{M}^{-1}\mathbf{ev};$

$\mathbf{ev} \leftarrow \mathbf{ev}/|\mathbf{ev}|;$

$l_{\text{new}} \leftarrow \mathbf{ev}^\top \mathbf{M} \mathbf{ev};$

end while

return l_{new}

Here `diag` designates a mapping $\mathbb{R}^n \mapsto \mathbb{R}^{n,n}$ that converts a vector into a diagonal matrix with the vector components as diagonal entries.

We assume that this code does something meaningful in a larger context that we do not care about. It might be taken from an old code that we have to upgrade.

(2-10.a) 🕒 (30 min.) Directly port the function `rankoneinvit` (Algorithm 2.10.1) to C++ using EIGEN. To that end create a function

```
double rankoneinvit(const Eigen::VectorXd &d, const double &tol);
```

The C++ code should perform exactly the same computations. Recall that in EIGEN the `asDiagonal()` method converts a vector into a diagonal matrix, see 🐼 EIGEN documentation. EIGEN also offers the `normalize()` method for vectors.

HINT 1 for (2-10.a): Heed the advice of [Lecture → Rem. 2.2.1.6]! Instead of computing the inverse of a matrix, solve a linear system of equations. ┘

SOLUTION of (2-10.a):

C++-code 2.10.2: Solution of (2-10.a)

```
2 double rankoneinvit(const Eigen::VectorXd& d, const double& tol) {
3   // Initialization
4   double lmin = 0, lnew = 0;
5   // TODO: (2-10.a) Port the pseudocode in algorithm (3.10.1) to C++-code.
6   // START
7   Eigen::VectorXd ev = d;
8   lnew = d.cwiseAbs().minCoeff();
9
10  while (std::abs(lnew - lmin) > tol * lmin) {
11    lmin = lnew;
12    Eigen::MatrixX<double> M = d.asDiagonal();
```

```

13  M += ev * ev.transpose();
14  // Instead of calculating the inverse of M directly
15  // to get ev_new = M^{-1} * ev_old,
16  // we solve the LSE M * ev_new = ev_old.
17  ev = M.lu().solve(ev);
18  // Equivalent to ev <- ev / |ev|:
19  ev.normalize();
20  lnew = ev.dot(M * ev);
21  }
22  // END
23  return lnew;
24  }

```

Get it on  GitLab ([rankoneinvit.hpp](https://gitlab.com/rankoneinvit)).

(2-10.b)  (10 min.) What is the asymptotic complexity of the body of the loop in function `rankoneinvit`?

HINT 1 for (2-10.b): Find which passage mostly contributes to the overall complexity of the code. What is the corresponding asymptotic complexity?

SOLUTION of (2-10.b):

Solving the linear system with matrix M (line 10) is the most relevant contributor to the asymptotic complexity. The complexity of solving a linear system is generally $O(n^3)$.

(2-10.c)  (45 min.) [depends on Sub-problem (2-10.a)]

Implement an EIGEN-based C++ function

```
double rankoneinvit_fast(const Eigen::VectorXd &d, const double
    &tol);
```

that is *algebraically equivalent* to `rankoneinvit()` from, but enjoys a way better asymptotic complexity by avoiding any invocation of Gaussian elimination/LU-factorization.

HINT 1 for (2-10.c): Exploit the Sherman-Morrison-Woodbury formula (see [Lecture → § 2.6.0.12]).

SOLUTION of (2-10.c):

Since we want to solve $\mathbf{ev}_{new} = \mathbf{M}^{-1}\mathbf{ev}$ with $\mathbf{M} = \text{diag}(\mathbf{d}) + \mathbf{ev} \cdot \mathbf{ev}^T$, we can apply the Sherman-Morrison-Woodbury formula:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} - \frac{\mathbf{A}^{-1}\mathbf{u}(\mathbf{v}^H(\mathbf{A}^{-1}\mathbf{b}))}{1 + \mathbf{v}^H(\mathbf{A}^{-1}\mathbf{u})}$$

In our case, assuming that $\mathbf{ev}$ is real, the formula becomes:

$$\begin{aligned}\mathbf{ev}_{new} &= \text{diag}(\mathbf{d})^{-1} \mathbf{ev} - \frac{\text{diag}(\mathbf{d})^{-1} \mathbf{ev} (\mathbf{ev}^\top (\text{diag}(\mathbf{d})^{-1} \mathbf{ev}))}{1 + \mathbf{ev}^\top (\text{diag}(\mathbf{d})^{-1} \mathbf{ev})} \\ &= \text{diag}(\mathbf{d})^{-1} \mathbf{ev} \cdot \frac{1}{1 + \mathbf{ev}^\top (\text{diag}(\mathbf{d})^{-1} \mathbf{ev})}\end{aligned}$$

As $\text{diag}(\mathbf{d})^{-1} = \text{diag}(d_1^{-1}, d_2^{-1}, \dots)$, this implementation is faster. More precisely, we simply need to perform a component-wise product between vectors $\text{diag}(d_1^{-1}, d_2^{-1}, \dots)$ and $\mathbf{ev}$ and another one between $\mathbf{ev}^\top$ and $\text{diag}(\mathbf{d})^{-1} \mathbf{ev}$. The asymptotic complexity is therefore $O(n)$.

C++-code 2.10.3: Solution of (2-10.c)

```

2  double rankoneinvit_fast(const Eigen::VectorXd& d, const double& tol) {
3      // Initialization
4      double lmin = 0, lnew = 0;
5      // TODO: (2-10.c) Rewrite rankoneinvit() to have much better asymptotic
6      // complexity.
7      // START
8      Eigen::VectorXd ev = d;
9      lnew = d.cwiseAbs().minCoeff();
10
11     Eigen::VectorXd dinv = (1. / d.array()).matrix();
12     while (std::abs(lnew - lmin) > tol * lmin) {
13         lmin = lnew;
14         Eigen::VectorXd ev0 = ev;
15
16         // Here we solve the linear system
17         // with the Sherman-Morrison-Woodbury formula
18         // in the case of rank-1 perturbations.
19         // This holds from  $M = \text{diag}(d) + ev * ev^t$ 
20         Eigen::VectorXd Aib = dinv.cwiseProduct(ev);
21         const double temp = ev.transpose() * Aib;
22         ev = Aib / (1 + temp);
23         ev.normalize();
24         // Better than the corresponding naive implementation.
25         // This holds from  $M = \text{diag}(d) + ev * ev^t$ , too
26         lnew = ev.transpose() * d.cwiseProduct(ev) + pow(ev.transpose() * ev0, 2);
27     }
28     // END
29     return lnew;
30 }

```

Get it on  GitLab ([rankoneinvit.hpp](#)).



(2-10.d)  (10 min.) [depends on Sub-problem (2-10.c)]

What is the asymptotic complexity of the execution of the loop body in your implementation of `rankoneinvit_fast()` from Sub-problem (2-10.c)?

SOLUTION of (2-10.d):

The loop body of the C++ function `rankoneinvit_fast` only consists in vector-vector multiplications. The new asymptotic complexity is therefore $O(n)$.



(2-10.e) 📄 (30 min.) Tabulate the runtimes of the two C++ implementations with different vector sizes $n = 2^p$, with $p = 2, 3, \dots, 8$. As test vector `d` use `Eigen::VectorXd::LinSpaced(n, 1, 2)`. Estimate the laws of the different asymptotic complexities of the two implementations using the measured runtimes.

SOLUTION of (2-10.e):

C++-code 2.10.4: Solution of (2-10.e)

```

2  void rankoneinvit_runtime() {
3      constexpr unsigned int repeats = 3;
4      constexpr double tol = 1e-3;
5      // TODO: (2-10.e) Tabulate the runtimes of both implementations according to
6      // the problem description. The choice of the "tol" argument affects the
7      // runtime.
8      // START
9      double lmin;
10     Eigen::VectorXd d;
11     Timer tm_slow, tm_fast;
12
13     std::cout << std::endl
14               << std::setw(15) << "n" << std::setw(15) << "Slow" << std::setw(15)
15               << "Fast" << std::endl;
16
17     for (unsigned int n = 2; n <= 256; n <= 1) {
18         tm_slow.reset();
19         tm_fast.reset();
20         for (unsigned int r = 0; r < repeats; ++r) {
21             d = Eigen::VectorXd::LinSpaced(n, 1, 2);
22
23             tm_slow.start();
24             lmin = rankoneinvit(d, tol);
25             tm_slow.stop();
26
27             tm_fast.start();
28             lmin = rankoneinvit_fast(d, tol);
29             tm_fast.stop();
30         }
31         std::cout << std::setw(15) << n << std::scientific << std::setprecision(3)
32               << std::setw(15) << tm_slow.min() << std::setw(15)
33               << tm_fast.min() << std::endl;
34     }
35     // END
36 }

```

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



End Problem 2-10 , 125 min.