

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 2-2: Structured linear systems with pivoting

This problem deals with a block structured system and ways to efficiently implement the solution of such system. For this problem, you should have understood Gauss elimination with partial pivoting for the solution of a linear system, see [Lecture → Section 2.3.3].

This problem is connected with [Lecture → Section 2.3.2], [Lecture → Section 2.5], and involves C++ implementation using EIGEN.

We consider a block partitioned linear system of equations

$$\mathbf{Ax} = \mathbf{b}, \quad \mathbf{A} = \begin{bmatrix} \mathbf{D}_1 & \mathbf{C} \\ \mathbf{C} & \mathbf{D}_2 \end{bmatrix} \in \mathbb{R}^{2n, 2n}, \quad (2.2.1)$$

where all the $n \times n$ -matrices $\mathbf{D}_1$, $\mathbf{D}_2$ and $\mathbf{C}$ are *diagonal*. Hence, the matrix $\mathbf{A}$ can be described through three n -vectors $\mathbf{d}_1$, $\mathbf{c}$ and $\mathbf{d}_2$, which provide the diagonals of the matrix blocks. These vectors will be passed as arguments `d1`, `c`, and `d2` to the C++ codes below.

(2-2.a) ☐ (20 min.) Which permutation of rows and columns converts the matrix into a **tridiagonal matrix**?

HINT 1 for (2-2.a): Draw the “pattern” of the matrix, that is, the positions of its non-zero entries, e.g., for $n = 3$

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

Which rows and columns are “coupled” through non-zero entries? ┘

SOLUTION of (2-2.a):

Let us first look at the case $n = 2$:

$$\mathbf{A} = \begin{bmatrix} d_1 & 0 & c_1 & 0 \\ 0 & d_2 & 0 & c_2 \\ c_1 & 0 & d_3 & 0 \\ 0 & c_2 & 0 & d_4 \end{bmatrix}.$$

We employ the following permutation of rows and columns: $1 \leftarrow 1, 2 \leftarrow 3, 3 \leftarrow 2, 4 \leftarrow 4$ and end up with the rearranged matrix

$$\begin{bmatrix} d_1 & 0 & c_1 & 0 \\ 0 & d_2 & 0 & c_2 \\ c_1 & 0 & d_3 & 0 \\ 0 & c_2 & 0 & d_4 \end{bmatrix} \xrightarrow{\text{row pmt.}} \begin{bmatrix} d_1 & 0 & c_1 & 0 \\ c_1 & 0 & d_3 & 0 \\ 0 & d_2 & 0 & c_2 \\ 0 & c_2 & 0 & d_4 \end{bmatrix} \xrightarrow{\text{col pmt.}} \begin{bmatrix} d_1 & c_1 & 0 & 0 \\ c_1 & d_3 & 0 & 0 \\ 0 & 0 & d_2 & c_2 \\ 0 & 0 & c_2 & d_4 \end{bmatrix}.$$

Obviously, we need to “interlace” the blocks of the matrix.

We start by moving the $n + k$ -th row ($k > 1$) to $(2k - 1)$ -th position, and the k -th row ($k \leq n$) to $2k$ -th row.

We do the same for the columns: moving the $n + k$ -th column ($k > 1$) to $(2k - 1)$ -th position, and the k -th column ($k \leq n$) to $2k$ -th column.

In fact, we observe that the resulting tridiagonal matrix is rather special: It is a **block-diagonal** matrix, whose diagonal blocks are of size 2×2 !



(2-2.b) 🕒 (20 min.) In the file `blocklsepiv.hpp` write an efficient C++ function

```
Eigen::VectorXd multA(
    const Eigen::VectorXd & d1, const Eigen::VectorXd & d2,
    const Eigen::VectorXd & c, const Eigen::VectorXd & x);
```

that returns $y := Ax$. The argument x passes a column vector $x \in \mathbb{R}^{2n}$ and the other arguments the vectors defining the matrix as in (2.2.1).

SOLUTION of (2-2.b):

Let us decompose $x := [x_1; x_2]$, $x_i \in \mathbb{R}^n$. Performing a block matrix vector multiplication, we obtain:

$$Ax = \begin{bmatrix} d_1 \odot x_1 + c \odot x_2 \\ c \odot x_1 + d_2 \odot x_2 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} \odot x + \begin{bmatrix} c \\ c \end{bmatrix} \odot \begin{bmatrix} x_2 \\ x_1 \end{bmatrix}.$$

Here, “ $\odot$ ” denotes the component wise product of two vectors of equal length, and d_1, d_2, c are vectors containing the diagonal entries of the matrices D_1, D_2 and C .

This can efficiently be implemented in C++.

C++ code 2.2.2: Implementation of `multA()`

```
2 Eigen::VectorXd multA(const Eigen::VectorXd& d1, const Eigen::VectorXd& d2,
3                       const Eigen::VectorXd& c, const Eigen::VectorXd& x) {
4     const unsigned int n = d1.size();
5     assert(n == d2.size() && n == c.size() && 2 * n == x.size() &&
6            "Size mismatch!");
7     Eigen::ArrayXd y = Eigen::VectorXd::Zero(2 * n);
8
9     // TODO: (2-2.b) Efficiently calculate A*x. Eigen's array
10    // arithmetic is
11    // helpful here.
12    // START
13    auto& x1 = x.head(n).array();
14    auto& x2 = x.tail(n).array();
15
16    y << d1.array() * x1 + c.array() * x2, d2.array() * x2 + c.array() * x1;
17    // END
18    return y;
19 }
```

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



(2-2.c) 🕒 (15 min.) Compute the LU-factors of A as given in (2.2.1), where you may assume that they exist.

HINT 1 for (2-2.c): Take the cue from the block LU-factorization discussed in [Lecture → Rem. 2.3.2.19], in particular [Lecture → Eq. (2.3.2.20)].

SOLUTION of (2-2.c):

We compute the LU-factors using the block Gaussian elimination algorithm:

$$\mathbf{A} = \begin{bmatrix} \mathbf{D}_1 & \mathbf{C} \\ \mathbf{C} & \mathbf{D}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{D}_1^{-1}\mathbf{C} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{D}_1 & \mathbf{C} \\ \mathbf{0} & \mathbf{D}_2 - \mathbf{C}\mathbf{D}_1^{-1}\mathbf{C} \end{bmatrix} = \mathbf{L}\mathbf{U}, \quad (2.2.3)$$

where $\mathbf{I}$ denotes $n \times n$ identity matrix and $\mathbf{0}$ indicates $n \times n$ zero matrix. Since all matrices are diagonal, the factor $\mathbf{L}$ is lower triangular with unit diagonal and the factor $\mathbf{U}$ is upper triangular.



(2-2.d) 🕒 (15 min.) [depends on Sub-problem (2-2.c)]

When does an LU-decomposition of $\mathbf{A}$ from (2.2.1) according to [Lecture → Def. 2.3.2.3] exist?

SOLUTION of (2-2.d):

As we see from the explicit formula (2.2.3), an LU-decomposition exists, **if and only if** $\mathbf{D}_1$ is regular, that is, if and only if the diagonal entries of $\mathbf{D}_1$ are all different from 0.



(2-2.e) 🕒 (15 min.) [depends on Sub-problem (2-2.c)]

Give an example of a matrix $\mathbf{A}$ with the structure as in (2.2.1)

- that possesses an LU-decomposition exists, but fails to be regular,
- for which an LU-decomposition does not exist, though it is invertible.

HINT 1 for (2-2.e): Examples can already be found for $n = 2$



SOLUTION of (2-2.e):

Consider the matrices

$$\mathbf{A}_1 := \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}, \quad \mathbf{A}_2 := \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}.$$

- Matrix $\mathbf{A}_1$ is obviously singular, because its columns/rows are linearly dependent.
- Matrix $\mathbf{A}_2$ is invertible, because it has linearly independent rows.



(2-2.f) 🕒 (45 min.) [depends on Sub-problem (2-2.c) and Sub-problem (2-2.a)]

Write an *efficient* C++ function

```
Eigen::VectorXd solveA(
    const Eigen::VectorXd & d1, const Eigen::VectorXd & d2,
    const Eigen::VectorXd & c, const Eigen::VectorXd & b);
```

that solves $\mathbf{Ax} = \mathbf{b}$ with Gaussian elimination. The argument $\mathbf{b}$ passes the right-hand-side vector $\mathbf{b}$, whereas the other arguments provide information about the matrix $\mathbf{A}$ as defined in (2.2.1).

- You must not use EIGEN's built-in linear solvers for the full matrix.
- Test for “near singularity” of the matrix.

HINT 1 for (2-2.f): Recall the observation made in Sub-problem (2-2.a), which permits you to reduce everything to the solution of $n \times 2$ linear systems of equations. ┘

SOLUTION of (2-2.f):

Drawing on what we learned in Sub-problem (2-2.a), namely that only rows/column k and $n+k$ of $\mathbf{A}$, $k = 1, \dots, n$ are coupled through non-zero entries, we see that the linear system of equations $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ has the structure as in (2.2.1), is equivalent to

$$\begin{bmatrix} (\mathbf{d}_1)_k & (\mathbf{c})_k \\ (\mathbf{c})_k & (\mathbf{d}_2)_k \end{bmatrix} \begin{bmatrix} (\mathbf{x})_k \\ (\mathbf{x})_{n+k} \end{bmatrix} = \begin{bmatrix} (\mathbf{b})_k \\ (\mathbf{b})_{n+k} \end{bmatrix}, \quad k = 1, \dots, n. \quad (2.2.4)$$

To solve the 2×2 linear systems (2.2.4) you may rely on EIGEN's direct elimination solvers as discussed in [Lecture $\rightarrow$ § 2.5.0.4].

Remark. In light of the advice given in the beginning of [Lecture $\rightarrow$ Section 2.5] it is even advisable to use library routines even for 2×2 linear systems, also for the sake of numerical stability.

To test for near singularity of a 2×2 -matrix we can rely on the angle enclosed by its columns. For a $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2] \in \mathbb{R}^{2,2}$ we check

$$\left| \frac{\mathbf{x}_1^\top \mathbf{x}_2}{\|\mathbf{x}_1\|_2 \|\mathbf{x}_2\|_2} \right| \leq 1 - \text{EPS}.$$

If that fraction attains a value ± 1 , the columns $\mathbf{x}_1, \mathbf{x}_2$ of the matrix $\mathbf{X}$ are linearly dependent. Alternatively, you may check for a relatively small determinant

$$|\det \mathbf{X}| = |(\mathbf{X})_{11}(\mathbf{X})_{22} - (\mathbf{X})_{12}(\mathbf{X})_{21}| \leq \text{EPS} \cdot \|\mathbf{x}_1\|_2 \|\mathbf{x}_2\|_2.$$

Here, inspecting the relative size of the determinant is crucial, recall [Lecture $\rightarrow$ Rem. 1.5.3.15].

C++ code 2.2.5: Implementation of `solveA()`

```
2 Eigen::VectorXd solveA(const Eigen::VectorXd& d1, const Eigen::VectorXd& d2,
3                       const Eigen::VectorXd& c, const Eigen::VectorXd& b) {
4     const unsigned int n = d1.size();
5     assert(n == d2.size() && n == c.size() && 2 * n == b.size() &&
6           "Size mismatch!");
7
8     Eigen::ArrayXd c1_arr = c, c2_arr = c, d1_arr = d1, d2_arr = d2, b_arr = b;
9
10    constexpr double eps = std::numeric_limits<double>::epsilon();
11
12    // TODO: (2-2.d) Solve Ax = b for x using Gaussian elimination
13    // without using
14    // Eigen's built-in solvers
```

```

14 // START
15
16 // for forward elimination and pivoting we are only required to loop
17 // the first
18 // half of the matrix; loop over diagonal
19 for (unsigned int k = 0; k < n; ++k) {
20     // check if need to pivot (i.e. swap two rows)
21     // luckily we only need to check two rows
22     const double maxk = std::max(std::abs(d1_arr(k)), std::abs(c1_arr(k)));
23     const double maxnk = std::max(std::abs(c2_arr(k)), std::abs(d2_arr(k)));
24     if (std::abs(c1_arr(k)) / maxk > std::abs(d1_arr(k)) / maxnk) {
25         // matrix
26         std::swap(d1_arr(k), c2_arr(k));
27         std::swap(c1_arr(k), d2_arr(k));
28         // r.h.s.
29         std::swap(b_arr(n), b_arr(n + k));
30     }
31
32     // check if matrix is almost singular
33     const double piv = d1_arr(k);
34     // norm of the block from k, k to n - 1, n - 1
35     const double norm = std::abs(d1_arr(k)) + std::abs(d2_arr(k)) +
36                         std::abs(c1_arr(k)) + std::abs(c2_arr(k));
37     if (piv < eps * norm) {
38         std::cout << "Warning: matrix nearly singular!" << std::endl;
39     }
40
41     // Multiplication factor
42     const double fac = c2_arr(k) / piv;
43
44     // Actually perform substitution
45     // bottom right portion changes
46     d2_arr(k) -= c1_arr(k) * fac;
47     // r.h.s.
48     b_arr(n + k) -= b_arr(k) * fac;
49 }
50
51 // Now the system has the form:
52 // | d1 | c | | | |
53 // | 0 | d2 | * | x | = | b |
54 // with d1, d2, c diagonal
55
56 // Backward substitution
57 // lower portion
58 b_arr.tail(n) /= d2_arr;
59 // upper portion
60 b_arr.head(n) = (b_arr.head(n) - c1_arr * b_arr.tail(n)) / d1_arr;
61 // END
62
63 return b_arr;
64 }

```

Get it on  GitLab (blocksepiv.hpp).

Remark. Probably the use of EIGEN's *sparse elimination solvers* as presented in [Lecture → Section 2.7.3] also yields an efficient implementation, because that solver should detect the special decoupling of unknowns encoded in [A](#). ▲

(2-2.g)  (10 min.) State the **asymptotic complexity** of your implementation of `solveA` in term of

the problem size parameter $n \rightarrow \infty$.

SOLUTION of (2-2.g):

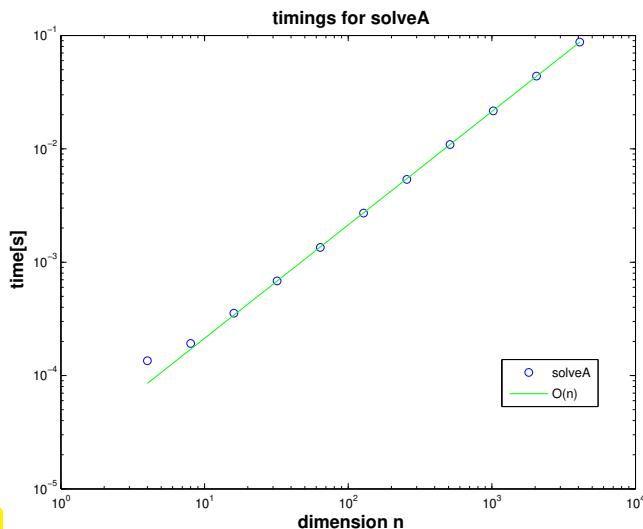
In the implementation of `solveA()` we sequentially solve n small linear systems of size 2×2 , which amounts to spending n times a constant effort. Thus the asymptotic complexity is $O(n)$ for $n \rightarrow \infty$.



(2-2.h) Determine the asymptotic complexity of `solveA` in a numerical experiment through measuring runtimes.

As test case use $\mathbf{d}_1 := \mathbf{1}^\top = [1, \dots, n]^\top = -\mathbf{d}_2$, $\mathbf{c} = \mathbf{1}$ and $\mathbf{b} = \begin{bmatrix} \mathbf{d}_1 \\ \mathbf{d}_1 \end{bmatrix}$. Tabulate the runtimes (in seconds, 3 decimal digit, scientific notation) for $n = 2^\ell$, $\ell = 2, \dots, 20$.

SOLUTION of (2-2.h):



As a clear indication for linear complexity the data points in a problem size vs. runtime plot should be approximately located on a straight line, as in Fig. 5.

Fig. 5

C++11-code 2.2.6: Solution of (2-2.h)

```

2 void numericalExperiment() {
3     constexpr unsigned int repeats = 10;
4     Eigen::VectorXd d1, d2, c, b, y;
5     plt::figure();
6
7     // TODO: (2-2.f) Tabulate runtimes of solveA and plot them using
8     // matplotlibcpp. START
9     std::vector<double> n(7), times(7), o_n(7);
10
11     // Table header
12     std::cout << std::setw(10) << "k" << std::scientific << std::setprecision(3)
13               << std::setw(15) << "runtime [s]" << std::endl;
14
15     // Loop from 2 to 2^7
16     for (unsigned int k = 2, i = 0; k <= (1 << 7); k <= 1, ++i) {
17         d1 = Eigen::VectorXd::LinSpaced(k, 1, k);
18         d2 = -d1;
19         c = Eigen::VectorXd::Ones(k);
20         b.resize(2 * k);
    
```

```

21     b << d1, d1;
22
23     // Repeat measurements
24     Timer tm;
25     for (unsigned int r = 0; r < repeats; ++r) {
26         // Test run
27         tm.start();
28         y = solveA(d1, d2, c, b);
29         tm.stop();
30     }
31
32     // Keep track of data for plotting
33     n[i] = k;
34     times[i] = tm.duration();
35     o_n[i] = k * 1e-5; // scale to beautify plot
36
37     // Print measurements
38     std::cout << std::setw(10) << k << std::scientific << std::setprecision(3)
39               << std::setw(15) << tm.duration() << std::endl;
40 }
41
42 // Plotting
43 plt::loglog(n, times, "o", {"label", "solveA"});
44 plt::loglog(n, o_n, {"label", "O(n)"});
45 plt::title("timings for solveA accumulated over 10 runs");
46 plt::xlabel("dimension n");
47 plt::ylabel("time [s]");
48 plt::legend();
49 // END
50 plt::savefig("cx_out/complexity.png");
51 }

```

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



End Problem 2-2 , 140 min.