

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 2-6: Banded matrix

Banded matrices are an important class of structured matrices; see [Lecture → Section 2.7.5] and [Lecture → Def. 2.7.5.1]. We will study ways to exploit during computations the knowledge that only some (sub)diagonals of a matrix are nonzero.

Connected with [Lecture → Section 1.4.3] and [Lecture → Section 2.7.5]

For $n \in \mathbb{N}$, consider the following matrix ($a_i, b_i \in \mathbb{R}$):

$$\mathbf{A} := \begin{bmatrix} 2 & a_1 & 0 & \dots & \dots & \dots & 0 \\ 0 & 2 & a_2 & 0 & \dots & \dots & 0 \\ b_1 & 0 & \ddots & \ddots & \ddots & & \vdots \\ 0 & b_2 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & a_{n-1} \\ 0 & 0 & \dots & 0 & b_{n-2} & 0 & 2 \end{bmatrix} \in \mathbb{R}^{n,n} \quad (2.6.1)$$

This matrix is an instance of a **banded matrix**.

(2-6.a)  (15 min.) In the file `efficientbandmult.hpp` implement an *efficient* C++ function `multAx` for the computation of $\mathbf{y} = \mathbf{Ax}$:

```
Eigen::VectorXd multAx(
    const Eigen::VectorXd & a, const VectorXd & b,
    const Eigen::VectorXd & x);
```

`Eigen::VectorXd` `a` and `Eigen::VectorXd` `b` contain the nonzero entries along the subdiagonals of matrix $\mathbf{A}$ from (2.6.1). The function is supposed to return $\mathbf{y} = \mathbf{Ax}$, when `Eigen::VectorXd` `x` supplies the vector $\mathbf{x} \in \mathbb{R}^n$.

SOLUTION of (2-6.a):

C++ code 2.6.2: Solution of (2-6.a)

```
2  template <class Vector>
3  void multAx(const Vector& a, const Vector& b, const Vector& x, Vector& y) {
4      const unsigned int n = x.size();
5      if (a.size() < n - 1 || b.size() < n - 2) {
6          std::cerr << "Error: size mismatch!" << std::endl;
7          return;
8      }
9      y.resize(n);
10
11     // TODO: (2-6.a) Multiply A*x and exploit the diagonal structure of
12     // the matrix. START Handle first two rows
13     if (n > 1) {
14         y(0) = 2 * x(0) + a(0) * x(1);
15     } else {
16         y(0) = 2 * x(0);
17         return;
18     }
19     if (n > 2) {
20         y(1) = 2 * x(1) + a(1) * x(2);
```

```

21 } else {
22     y(1) = 2 * x(1);
23     return;
24 }
25
26 // Row if n > 3 and without last row
27 for (unsigned int i = 2; i < n - 1; ++i) {
28     y(i) = 2 * x(i) + b(i - 2) * x(i - 2) + a(i) * x(i + 1);
29 }
30
31 // Last row special case
32 if (n > 2) y(n - 1) = 2 * x(n - 1) + b(n - 3) * x(n - 3);
33 // END
34 }

```

Get it on  GitLab (efficientbandmult.hpp).



(2-6.b)  (30 min.) Show that $\mathbf{A}$ is invertible if $a_i, b_i \in [0, 1]$.

HINT 1 for (2-6.b): Trick: Give a proof by contradiction of $\ker \mathbf{A} = \{\mathbf{0}\}$ by looking at the largest (in modulus) component of a vector $\mathbf{x} \in \ker \mathbf{A}$.

SOLUTION of (2-6.b):

Assume by contradiction that $\ker \mathbf{A} \neq \{\mathbf{0}\}$.

Pick $\mathbf{x} \in \ker \mathbf{A}$ and consider its largest (in modulus) component $i = \operatorname{argmax} |x_i|$, with $x_i \neq 0$.

Since $\mathbf{A}\mathbf{x} = \{\mathbf{0}\}$, we have that $2x_i + a_i x_{i+1} + b_{i-2} x_{i-2} = 0 \Rightarrow 2 = -\left(\frac{x_{i+1}}{x_i} a_i + \frac{x_{i-2}}{x_i} b_{i-2}\right) \Rightarrow 2 \leq \left|\frac{x_{i+1}}{x_i} a_i + \frac{x_{i-2}}{x_i} b_{i-2}\right|$ (the last step is always true).

Then, given that $a_i, b_i \in [0, 1]$ by hypothesis and $\frac{x_{i+1}}{x_i}, \frac{x_{i-2}}{x_i} \in [-1, 1]$ from x_i being the largest (in modulus) component, we can say that $2 \leq \left|\frac{x_{i+1}}{x_i} a_i + \frac{x_{i-2}}{x_i} b_{i-2}\right| \leq a_i + b_{i-2} \leq 2$, unless all elements of $\mathbf{x}$ are equal. (But in this case, with $\mathbf{x} \neq \mathbf{0}$, we have another contradiction, as $2x + a_i x + b_{i-2} x = 0 \Rightarrow 2 + a_i + b_{i-2} = 0$ cannot be true.)

By contradiction, $\ker \mathbf{A} = \{\mathbf{0}\}$.

Notice that $\ker \mathbf{A} = \{\mathbf{0}\}$ can also be proven in a more straightforward way from the general fact that kernel vectors of irreducible and diagonally dominant matrices (see [Lecture $\rightarrow$ Def. 2.8.0.8]), like $\mathbf{A}$ in our example, must be multiples of $[1, 1, \dots, 1]^\top$ (i.e. all their elements must be equal). Given this fact, we can immediately jump to the step where we see that $2x + a_i x + b_{i-2} x = 0$ cannot hold.



(2-6.c)  (20 min.) Assume that $b_i = 0$ for all $i = 1, \dots, n - 2$. Implement an efficient C++ function solving $\mathbf{A}\mathbf{x} = \mathbf{r}$:

```

Eigen::VectorXd solvelseUpper(const Eigen::VectorXd & a, const
    Eigen::VectorXd & r);

```

`Eigen::VectorXd` `a` contains the nonzero entries along the upper subdiagonal of matrix `A` as defined in (2.6.1). `r` provides the right-hand-side vector of the linear system. The function should return the solution vector.

HINT 1 for (2-6.c): What special linear system do we face in the case $b_i = 0$? ┘

SOLUTION of (2-6.c):

If $b_i = 0$ we face a **upper triangular** linear system of equations, which can be solved by means of simple backward substitution.

The implementation involves only elementary operations on vector components.

C++11-code 2.6.3: Solution of (2-6.c)

```

2  template <class Vector>
3  void solveUpper(const Vector& a, const Vector& r, Vector& x) {
4      // Set up dimensions
5      const unsigned int n = r.size();
6      if (a.size() < n - 1) {
7          std::cerr << "Error: size mismatch!" << std::endl;
8          return;
9      }
10     x.resize(n);
11
12     // TODO: (2-6.c) Solve the LSE  $Ax = r$  for  $x$  by exploiting the
        diagonal
13     // structure of  $A$  and the fact that  $b$  is zero.
14     // START
15     // Backward substitution
16     x(n - 1) = 0.5 * r(n - 1);
17     for (int j = n - 2; j >= 0; --j) {
18         x(j) = 0.5 * (r(j) - a(j) * x(j + 1));
19     }
20     // END
21 }
```

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



(2-6.d)  (60 min.) For general $a_i, b_i \in [0, 1]$, implement an *efficient* C++ function that computes the solution of the linear system of equations $Ax = r$ by Gaussian elimination:

```

Eigen::VectorXd solveA(
    const Eigen::VectorXd & a, const Eigen::VectorXd & b,
    const Eigen::VectorXd & r);
```

The vectors `a` and `b` pass information about the matrix `A` as in Sub-problem (2-6.c), whereas `r` supplies the right-hand-side vector `R`. The return value is the solution `x`.

You must not use any built-in linear solvers of EIGEN.

HINT 1 for (2-6.d):

- Thanks to the constraint $a_i, b_i \in [0, 1]$, pivoting is not required in order to ensure the stability of Gaussian elimination, see [Lecture → Lemma 2.8.0.9].

- Note that Gaussian elimination fills in zero entries of **A** only along one lower off-diagonal, which can be stored as vector `Eigen::VectorXd c` (similarly to what was done in Sub-problem (2-6.a), Sub-problem (2-6.c)). If you store the diagonal (initially only made of 2s) as `Eigen::VectorXd d`, you can perform a straightforward Gaussian elimination only operating on vectors `Eigen::VectorXd c`, `Eigen::VectorXd d` and `Eigen::VectorXd r`.

┘

SOLUTION of (2-6.d):

Try to perform Gaussian elimination on matrix **A** in **??**, augmented by the known vector **r**.

Start from cell $(3,1)$, where b_1 is. By subtracting the 1st row from the 3rd row, you fill in the zero cell $(3,2)$ with $-\frac{b_1}{2}a_1$: call this entry c_2 (c_1 is the zero cell $(2,1)$). On the other hand, r_3 (3rd element of the known vector **r**, in position $(3, n+1)$) becomes $= r_3 - \frac{b_1}{2}r_1$.

Now in the 2nd column you have nonzero cells c_2 and b_2 , in position $(3,2)$ and $(4,2)$ respectively. To make them zero, subtract the 2nd row from the 3rd and 4th row. You then make cell $(3,3)$, which we can call d_3 as it is the 3rd cell along the diagonal, $= 2 - \frac{c_2}{2}a_2$, while cell the zero cell $(4,3)$, which we can call c_3 , becomes $= -\frac{b_2}{2}a_2$. On the other hand, r_3 becomes $= r_3 - \frac{c_2}{2}r_2$ while r_4 becomes $= r_4 - \frac{b_2}{2}r_2$.

Given these first two steps, you can derive the general algorithm in the solution code below.

C++ code 2.6.4: Solution of (2-6.d)

```

2  template <class Vector>
3  void solvelseA(const Vector& a, const Vector& b, const Vector& r, Vector& x) {
4      // Set up dimensions
5      const unsigned int n = r.size();
6      if (a.size() < n - 1 || b.size() < n - 2) {
7          std::cerr << "Error: size mismatch!" << std::endl;
8          return;
9      }
10
11     // TODO: (2-6.d) Compute the solution to A*x=r by Gaussian
12     // elimination and
13     // without the help of Eigen's solvers.
14     // START
15     Vector c, d, y;
16     c.resize(n - 1);
17     d.resize(n);
18     for (unsigned int i = 0; i < n - 1; ++i) {
19         c(i) = 0;
20         d(i) = 2;
21     }
22     d(n - 1) = 2;
23     x = r;
24     y = r;
25
26     // Plain vectors are enough
27     // Fill-in is confined to a single lower off-diagonal, which can be
28     // held in
29     // vector c
30     for (unsigned int i = 0; i < n - 2; ++i) {
31         c(i + 1) = -b(i) / d(i) * a(i);
32         d(i + 1) -= c(i) / d(i) * a(i);
33         y(i + 1) -= c(i) / d(i) * y(i);
34         y(i + 2) -= b(i) / d(i) * y(i);
35     }

```

```

34 |
35 | x(n - 1) = y(n - 1) / d(n - 1);
36 | for (int i = n - 2; i >= 0; --i) {
37 |     x(i) = (y(i) - a(i) * x(i + 1)) / d(i);
38 | }
39 | // END
40 | }

```

Get it on  [GitLab \(efficientbandmult.hpp\)](#).



(2-6.e) (10 min.) [depends on Sub-problem (2-6.d)]

What is the **asymptotic complexity** of your implementation of `solveA` for $n \rightarrow \infty$?

HINT 1 for (2-6.e): Figure out the computational cost of each step of the algorithm you coded and “sum them up”.

SOLUTION of (2-6.e):

In the code Code 2.6.4 we use three for-loops: one loop of size n and two loops of size, at most, 3 each (exploiting the banded structure of A), thus $O(n \cdot 3 \cdot 2)$ operations. Thirdly, for the backward substitution we use two loops, one of size n and the other of size, at most, 3, thus $O(n \cdot 3)$ operations. Therefore, the total complexity amounts to $O(12n)$ operations, which results in $O(n)$ asymptotic complexity.



(2-6.f) Implement a C++ function

```

Eigen::VectorXd solveASparse (
    const Eigen::VectorXd & a, const Eigen::VectorXd & b,
    const Eigen::VectorXd & r);

```

with exactly the same specification as `solveA()` in Sub-problem (2-6.d), but this time using EIGEN's sparse elimination solver as explained in [Lecture → Section 2.7.3].

HINT 1 for (2-6.f): The standard way of initialising a sparse EIGEN matrix efficiently is via the triplet format discussed in [Lecture → Section 2.7.2]. You may also use directly initialise a sparse matrix, provided that you `reserve()` (EIGEN built-in function) enough space for the nonzero entries of each column (see the  [EIGEN documentation](#)).

SOLUTION of (2-6.f):

C++ code 2.6.5: Solution of (2-6.f)

```

2 | template <class Vector>
3 | void solveAEigen(const Vector& a, const Vector& b, const Vector& r,
4 |                 Vector& x) {
5 |     // Set up dimensions
6 |     typedef typename Vector::Scalar Scalar;
7 |     const unsigned int n = r.size();

```

```

8  if (a.size() < n - 1 || b.size() < n - 2) {
9      std::cerr << "Error: size mismatch!" << std::endl;
10     return;
11 }
12
13 // TODO: (2-6.f) Solve A*x=r using Eigen's sparse solver.
14 // START
15 // Fill in matrix:
16 // We reserve three nonzero entries per row for Gaussian fill-in
17 Eigen::SparseMatrix<Scalar> A(n, n);
18 A.reserve(3);
19 for (unsigned int i = 0; i < n; ++i) {
20     A.insert(i, i) = 2;
21     if (i < n - 1) A.insert(i, i + 1) = a(i);
22     if (i >= 2) A.insert(i, i - 2) = b(i - 2);
23 }
24 A.makeCompressed();
25
26 // Call SparseLU
27 Eigen::SparseLU<Eigen::SparseMatrix<Scalar>> solver;
28 solver.analyzePattern(A);
29 solver.compute(A);
30 x = solver.solve(r);
31 // END
32 }

```

Get it on  GitLab (efficientbandmult.hpp).



End Problem 2-6 , 135 min.