

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 4-4: Solving triangular Toeplitz systems

In [Lecture → Section 4.5] we learned about **Toeplitz matrices**, the class of matrices with constant diagonals [Lecture → Def. 4.5.1.7]. Obviously, $m \times n$ Toeplitz matrices are **data-sparse** in the sense that it takes only $m + n - 1$ steps to encode them completely. Therefore, operations with Toeplitz matrices can often be done with asymptotic computational cost significantly lower than that of the same operations for a generic matrix of the same size. In [Lecture → Section 4.5.2] we have seen this for FFT-based matrix $\times$ vector multiplication for Toeplitz matrices.

In this problem we study an efficient FFT-based algorithm for solving triangular linear systems of equations whose coefficient matrix is Toeplitz. Such linear systems are faced, for instance, when inverting a finite, linear, time-invariant, causal channel (LT-FIR) as introduced in [Lecture → Section 4.1].

Template: Get it on  **GitLab**.

In [Lecture → Section 4.1.2] we found that the output operator of a causal finite linear time-invariant filter with impulse response $\mathbf{h} = [h_0, \dots, h_{n-1}]^T \in \mathbb{R}^n$ can be obtained by discrete convolution [Lecture → Eq. (4.1.2.4)]; see also [Lecture → Def. 4.1.3.3]. Given an input signal $\mathbf{x} := [x_0, \dots, x_{n-1}]^T \in \mathbb{R}^n$, the first n components $y_0, \dots, y_{n-1}$ of the output are obtained by:

$$\begin{bmatrix} y_0 \\ \vdots \\ y_{n-1} \end{bmatrix} = \underbrace{\begin{bmatrix} h_0 & 0 & \cdots & \cdots & 0 \\ h_1 & h_0 & 0 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & 0 \\ h_{n-1} & \cdots & \cdots & h_1 & h_0 \end{bmatrix}}_{=:\mathbf{H}_n \in \mathbb{R}^{n,n}} \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix} \quad (4.4.1)$$

Note that the matrix $\mathbf{H}_n \in \mathbb{R}^{n,n}$ is a lower triangular matrix with constant diagonals (linear algebra indexing)

$$(\mathbf{H})_{l,j} = \begin{cases} h_{l-j} & \text{if } l \geq j, \\ 0 & \text{else,} \end{cases} \quad l, j \in \{1, \dots, n\}$$

This matrix is clearly not circulant: see [Lecture → Def. 4.1.4.12]. At the same time, it still belongs to the class of *Toeplitz matrices*, see [Lecture → Def. 4.5.1.7], which generalizes the class of circulant matrices. (Recall [Lecture → Def. 4.5.1.7]: $\mathbf{T} \in \mathbb{K}^{m,n}$ is a Toeplitz matrix if there is a sequence/array $\mathbf{u} = (u_{-m+1}, \dots, u_{n-1}) \in \mathbb{K}^{m+n-1}$ such that $t_{ij} = u_{j-i}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$.)

If you want to recover the input signal from the output (*deconvolution*), you will face the following crucial task for digital signal processing: *solve a lower triangular LSE with a Toeplitz system matrix*. Of course, forward elimination, see [Lecture → Section 2.3], can achieve this with asymptotic complexity $\mathcal{O}(n^2)$: see [Lecture → Eq. (2.3.1.6)]. However, since a Toeplitz matrix merely has an information content of $\mathcal{O}(n)$ numbers, there might be a more efficient way – which ought to be explored by you in this problem.

(4-4.a)  (15 min.) Given a Toeplitz matrix $\mathbf{T} \in \mathbb{R}^{n,n}$, find another matrix $\mathbf{S} \in \mathbb{R}^{n,n}$ such that the following block matrix is a **circulant** matrix:

$$\mathbf{C} = \begin{bmatrix} \mathbf{T} & \mathbf{S} \\ \mathbf{S} & \mathbf{T} \end{bmatrix} \in \mathbb{R}^{2n,2n}.$$

HINT 1 for (4-4.a): Refer to [Lecture → Section 4.5.2].

SOLUTION of (4-4.a):

We start from the Toeplitz matrix

$$\mathbf{T} = \begin{bmatrix} t_0 & t_1 & \cdots & t_{n-1} \\ t_{-1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_1 \\ t_{-n+1} & \cdots & t_{-1} & t_0 \end{bmatrix}.$$

If we define

$$\mathbf{S} = \begin{bmatrix} 0 & t_{-n+1} & \cdots & t_{-1} \\ t_{n-1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_{-n+1} \\ t_1 & \cdots & t_{n-1} & 0 \end{bmatrix}$$

then, the $2n \times 2n$ -matrix

$$\mathbf{C} := \begin{bmatrix} \mathbf{T} & \mathbf{S} \\ \mathbf{S} & \mathbf{T} \end{bmatrix} = \begin{bmatrix} t_0 & t_1 & \cdots & t_{n-1} & 0 & t_{-n+1} & \cdots & t_{-1} \\ t_{-1} & \ddots & \ddots & \vdots & t_{n-1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_1 & \vdots & \ddots & \ddots & t_{-n+1} \\ t_{-n+1} & \cdots & t_{-1} & t_0 & t_1 & \cdots & t_{n-1} & 0 \\ 0 & t_{-n+1} & \cdots & t_{-1} & t_0 & t_1 & \cdots & t_{n-1} \\ t_{n-1} & \ddots & \ddots & \vdots & t_{-1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_{-n+1} & \vdots & \ddots & \ddots & t_1 \\ t_1 & \cdots & t_{n-1} & 0 & t_{-n+1} & \cdots & t_{-1} & t_0 \end{bmatrix}$$

will be circulant, because its columns are generated by repeated cyclic permutations.



(4-4.b) ☐ (15 min.) [depends on Sub-problem (4-4.a)]

Write the following C++ function:

```
MatrixXd toeplitz(const VectorXd &c, const VectorXd &r)
```

This function should take as input column vectors $\mathbf{c} \in \mathbb{R}^m$ and $\mathbf{r} \in \mathbb{R}^n$ and return a **Toeplitz matrix** $\mathbf{T} \in \mathbb{R}^{m,n}$ [Lecture → Def. 4.5.1.7] such that (linear-algebra indexing)

$$(\mathbf{T})_{i,j} = \begin{cases} (c)_{i-j+1} & , \text{ if } i \geq j, \\ (r)_{j-i+1} & , \text{ if } j > i, \end{cases} \quad 1 \leq i \leq m, 1 \leq j \leq n.$$

SOLUTION of (4-4.b):

C++11-code 4.4.2: Solution of (4-4.b)

```
2 Eigen::MatrixXd toeplitz(const Eigen::VectorXd &c, const Eigen::VectorXd &r) {
3     // Find size of Toeplitz matrix
```

```

4  const int m = c.size();
5  const int n = r.size();
6  Eigen::MatrixXd T(m, n);
7
8  // TODO: build Toeplitz matrix T
9  // START
10 for (int i = 0; i < n; ++i) {
11     T.col(i).tail(m - i) = c.head(m - i);
12 }
13 for (int i = 0; i < m; ++i) {
14     T.row(i).tail(n - i - 1) = r.segment(1, n - i - 1);
15 } // Do not reassign the diagonal!
16 // END
17
18 return T;
19 }

```

Get it on  GitLab (toeplitz.hpp).



(4-4.c) (20 min.) [depends on Sub-problem (4-4.a)]

Show that the following two C++ functions realize the same linear mapping $\mathbf{x} \mapsto \mathbf{y}$, when supplied with the same arguments.

C++11-code 4.4.3: Function `toepmatmult()`

```

2  Eigen::VectorXd toepmatmult(const Eigen::VectorXd &c, const Eigen::VectorXd &r,
3                             const Eigen::VectorXd &x) {
4      assert(c.size() == r.size() && c.size() == x.size() &&
5             "c, r, x have different lengths!");
6      return toeplitz(c, r) * x;
7  }

```

Get it on  GitLab (toeplitz.hpp).

C++11-code 4.4.4: Function `toepmult()`

```

2  Eigen::VectorXd toepmult(const Eigen::VectorXd &c, const Eigen::VectorXd &r,
3                           const Eigen::VectorXd &x) {
4      assert(c.size() == r.size() && c.size() == x.size() &&
5             "c, r, x have different lengths!");
6      int n = c.size();
7
8      // Complex arithmetic required here
9      Eigen::VectorXcd cr_tmp = c.cast<std::complex<double>>();
10     Eigen::VectorXcd x_tmp = x.cast<std::complex<double>>();
11
12     // Prepare vector encoding circulant matrix C
13     cr_tmp.conservativeResize(2 * n);
14     cr_tmp.tail(n) = Eigen::VectorXcd::Zero(n);
15     cr_tmp.tail(n - 1).real() = r.tail(n - 1).reverse();
16
17     // Zero padding
18     x_tmp.conservativeResize(2 * n);
19     x_tmp.tail(n) = Eigen::VectorXcd::Zero(n);
20 }

```

```

21 // Periodic discrete convolution from [Lecture → Code 4.2.2.4]
22 Eigen::VectorXcd y = pconvfft(cr_tmp, x_tmp).real();
23 y.conservativeResize(n);
24
25 return y;
26 }

```

Get it on  GitLab (toeplitz.hpp).

The function `pconvfft` is defined in [Lecture → Section 4.2.2], [Lecture → Code 4.2.2.4], and `toeplitz()` is the simple C++ function from Sub-problem (4-4.b).

HINT 1 for (4-4.c): When analyzing function `toeplmult()`, recall (4-4.a) and the properties of the product between a circulant matrix and a vector [Lecture → Rem. 4.1.4.15].

SOLUTION of (4-4.c):

First, we consider function `toeplmatmult()`. It takes two vectors c and r of the same size n as arguments. These vectors determine the first column and the first row of a Toeplitz matrix according to

$$T = \begin{bmatrix} (c)_1 & (r)_2 & \cdots & (r)_n \\ (c)_2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & (r)_2 \\ (c)_n & \cdots & (c)_2 & (c)_1 \end{bmatrix}.$$

Thus, the first C++ code `toeplmatmult()` multiplies this matrix with the vector x using the standard matrix-vector product.

Now we look `toeplmult`. This function uses the function `pconvfft` given in [Lecture → Code 4.2.2.4], which implements the periodic convolution of two vectors of the same size, which is equivalent to multiplying one vector with the circulant matrix defined by the other, see [Lecture → Section 4.1.4].

C++11-code 4.4.5: Function `pconvfft()` for the periodic convolution of two vectors

```

2 Eigen::VectorXcd pconvfft(const Eigen::VectorXcd &u,
3                           const Eigen::VectorXcd &x) {
4     Eigen::FFT<double> fft;
5     Eigen::VectorXcd tmp = (fft.fwd(u)).cwiseProduct(fft.fwd(x));
6
7     return fft.inv(tmp);
8 }

```

Get it on  GitLab (pconvfft.hpp).

By using the result from Sub-problem (4-4.a), we extend the Toeplitz matrix $T \in \mathbb{R}^{n,n}$ to a circulant matrix $C \in \mathbb{R}^{2n,2n}$. For circulant matrices the product with a vector is a periodic convolution, which can be computed via the discrete Fourier transform.

Since the matrix is now of a bigger size, we have to extend the vector x . This is done by **zero padding**.

$$C \begin{bmatrix} x \\ 0 \end{bmatrix} = \begin{bmatrix} T & S \\ S & T \end{bmatrix} \begin{bmatrix} x \\ 0 \end{bmatrix} = \begin{bmatrix} Tx + S0 \\ Sx + T0 \end{bmatrix} = \begin{bmatrix} Tx \\ Sx \end{bmatrix}$$

Using this result, we find the result of Tx in the first half of the vector that we get from periodic convolution.



(4-4.d) 🕒 (10 min.) What is the asymptotic complexity of `toepmatmult()` (→ Code 4.4.3) and `toepmult()` (→ Code 4.4.4)?

SOLUTION of (4-4.d):

- `toepmatmult()` uses standard matrix-vector multiplication. Thus, the complexity is $\mathcal{O}(n^2)$ for $n \rightarrow \infty$.
- The complexity of `toepmult` is mainly determined by that of the discrete Fourier transforms. At the same time, since we extended the matrix and the vector, we have to compute three Fourier transforms of length $2n$. Thus, the asymptotic complexity is $\mathcal{O}(n \log(n))$ for $n \rightarrow \infty$.



(4-4.e) 🧠 Explain in detail what the following C++ functions are meant for and why they are algebraically equivalent (i.e. equivalent when ignoring roundoff errors), provided that `h.size() = 2^l`.

C++11-code 4.4.6: Function `ttmatsolve`

```

2  Eigen::VectorXd ttmatsolve(const Eigen::VectorXd &h, const Eigen::VectorXd &y) {
3      assert(h.size() == y.size() && "h and y have different lengths!");
4      int n = h.size();
5
6      Eigen::VectorXd h_tmp = Eigen::VectorXd::Zero(n);
7      h_tmp(0) = h(0);
8
9      Eigen::MatrixXd T = toeplitz(h, h_tmp);
10
11     Eigen::VectorXd x = T.fullPivLu().solve(y);
12
13     return x;
14 }
```

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

C++11-code 4.4.7: Function `ttrecsolve`

```

2  Eigen::VectorXd ttrecsolve(const Eigen::VectorXd &h, const Eigen::VectorXd &y,
3                             int l) {
4      assert(h.size() == y.size() && "h and y have different lengths!");
5      // Result vector
6      Eigen::VectorXd x;
7      // Trivial case of as n 1x1 LSE
8      if (l == 0) {
9          x.resize(1);
10         x(0) = y(0) / h(0);
11     } else {
12         int n = std::pow(2, l);
13         int m = n / 2;
14
15         // Check matching length of vectors
16         assert(h.size() == n && y.size() == n &&
17              "h and y have length different from 2^l!");
18     }
```

```

18 Eigen::VectorXd x1 = ttrecsolve(h.head(m), y.head(m), l - 1);
19 Eigen::VectorXd y2 =
20   y.segment(m, m) -
21   toepmult(h.segment(m, m), h.segment(1, m).reverse(), x1);
22 Eigen::VectorXd x2 = ttrecsolve(h.head(m), y2, l - 1);
23
24
25 x.resize(n);
26 x.head(m) = x1;
27 x.tail(m) = x2;
28 }
29
30 return x;
31 }

```

Get it on  GitLab (toeplitz.hpp).

HINT 1 for (4-4.e): Consider the following 2×2 block partitioning of the linear system of equations in the case of even system size n :

$$\mathbf{H}_n \mathbf{x} = \mathbf{y} \Leftrightarrow \begin{bmatrix} \mathbf{H}_m & 0 \\ \mathbf{T}_m & \mathbf{H}_m \end{bmatrix} \begin{bmatrix} \mathbf{x}^1 \\ \mathbf{x}^2 \end{bmatrix} = \begin{bmatrix} \mathbf{y}^1 \\ \mathbf{y}^2 \end{bmatrix}, \quad \mathbf{T}_m \in \mathbb{R}^{m,m}, \quad m := n/2$$

SOLUTION of (4-4.e):

In `ttmatsolve()` we first build the following triangular Toeplitz matrix:

$$\mathbf{H}_n = \begin{bmatrix} h_1 & 0 & \cdots & 0 \\ h_2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ h_n & \cdots & h_2 & h_1 \end{bmatrix}.$$

Then, we solve the triangular system. Thus, $\mathbf{x} = \mathbf{H}_n^{-1} \mathbf{y}$.

To understand the purpose of `ttrecsolve()`, we first take into account the hint. For $n = 2m$, we state the linear system of equations as a 2×2 block system:

$$\mathbf{H}_n \mathbf{x} = \mathbf{y} \Leftrightarrow \begin{bmatrix} \mathbf{H}_m & 0 \\ \mathbf{T}_m & \mathbf{H}_m \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix}.$$

The block matrices are defined as follows:

$$\mathbf{H}_m = \begin{bmatrix} h_1 & 0 & \cdots & 0 \\ h_2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ h_{n/2} & \cdots & h_2 & h_1 \end{bmatrix}, \quad \mathbf{T}_m = \begin{bmatrix} h_{n/2+1} & h_{n/2} & \cdots & h_2 \\ h_{n/2+2} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & h_{n/2} \\ h_n & \cdots & h_{n/2+2} & h_{n/2+1} \end{bmatrix}.$$

The solution of the block-triangular system is given by

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{H}_m^{-1} \mathbf{y}_1 \\ \mathbf{x}_2 &= \mathbf{H}_m^{-1} (\mathbf{y}_2 - \mathbf{T}_m \mathbf{x}_1) \end{aligned}$$

Note that we only have to invert $\mathbf{H}_m$, which is also a triangular Toeplitz matrix. This suggests a *recursive approach*.

The function `ttrecsolve()` first finds out whether the vectors are of length 1 or not. If they are of length 1, the solution is trivial and is calculated by just dividing y by h . If the length is larger than 1, the system is divided in the two subsystems as shown above. The first subsystem is solved by applying the `ttrecsolve()` again. Then, the second subsystem is set up by calculating the right hand side of the second system. Note that $\mathbf{T}_m$ is a Toeplitz matrix. Thus, the multiplication is performed by using the `toeplmult()` function from Sub-problem (4-4.c). Then, the second system is again solved by applying `ttrecsolve()`.

Since the system is always divided in half, this algorithm will not work if the size of the original system is not a power of 2.



(4-4.f) (5 min.) Why is the algorithm implemented in `ttrecsolve()` called a “divide & conquer” method?

SOLUTION of (4-4.f):

The algorithm is called “divide & conquer” because we divide the problem into two problems of the same kind, both of only half the original size, until we end up with a trivial one-dimensional problem. The solutions of the two sub-problems are then assembled into the solution of the full-size problem.



(4-4.g) (30 min.) Perform a runtime comparison of `toeplmatmult` vs `toeplmult` and `ttmatsolve` vs `ttrecsolve` for

```
h = VectorXd::LinSpaced(n, 1, n).cwiseInverse();
```

and $n = 2^l$, $l = 3, \dots, 9$. Plot the timing results for both functions in doubly logarithmic scale. Describe your observations.

HINT 1 for (4-4.g): Refer to § 1.1.6 for detailed instructions on how to measure runtimes in a C++ code using `std::chrono`. Alternatively, use the provided `timer.h`.

SOLUTION of (4-4.g):

C++11-code 4.4.8: Solution of (4-4.g)

```
2 void runtime_toeplitz() {
3     // memory allocation for plot
4     std::vector<double> vec_size;
5     std::vector<double> elap_time_matmult, elap_time_mult, elap_time_ttmatsolve,
6         elap_time_ttrecrecsolve;
7
8     // header for the results to print out
9     std::cout << std::setw(8) << "n" << std::setw(15) << "toeplmatmult"
10        << std::setw(15) << "toeplmult" << std::setw(20) << "ttmatsolve"
11        << std::setw(15) << "ttrecsolve" << std::endl;
12 }
```

```

13 for (unsigned int l = 3; l <= 9; l += 1) {
14     // vector size
15     unsigned int n = std::pow(2, l);
16     // save vector size n
17     vec_size.push_back(n);
18
19     // number of repetitions
20     unsigned int repeats = 3;
21     // TODO: (4-4.g) Perform a runtime comparison by repeating time
        computation
22     // 'repeats' times
23     // START
24     Timer tm_matmult, tm_mult, tm_ttmult, tm_ttrec;
25     // repeat test 'repeats' times
26     for (unsigned int rr = 0; rr < repeats; ++rr) {
27         // create runtime test using random vectors and given vector h
28         Eigen::VectorXd h = Eigen::VectorXd::LinSpaced(n, 1, n).cwiseInverse();
29         Eigen::VectorXd c = Eigen::VectorXd::Random(n);
30         Eigen::VectorXd r = Eigen::VectorXd::Random(n);
31         Eigen::VectorXd x = Eigen::VectorXd::Random(n);
32         Eigen::VectorXd y = Eigen::VectorXd::Random(n);
33         r(0) = c(0);
34
35         // compute times for toepmatmult implementation
36         tm_matmult.start();
37         toepmatmult(c, r, x);
38         tm_matmult.stop();
39
40         // compute times for toepmult implementation
41         tm_mult.start();
42         toepmult(c, r, x);
43         tm_mult.stop();
44
45         // compute times for ttmatsolve implementation
46         tm_ttmult.start();
47         ttmatsolve(h, y);
48         tm_ttmult.stop();
49         // compute times
50         tm_ttrec.start();
51         ttrecsolve(h, y, l);
52         tm_ttrec.stop();
53     }
54     // END
55
56     // print the results: toepmult vs toepmatmult
57     std::cout << std::setw(8) << n << std::scientific << std::setprecision(3)
58         << std::setw(15) << tm_matmult.min() << std::setw(15)
59         << tm_mult.min() << std::setw(20) << tm_ttmult.min()
60         << std::setw(15) << tm_ttrec.min() << std::endl;
61
62     // save elapsed time for plot: toepmatmult vs toepmult
63     elap_time_matmult.push_back(tm_matmult.min());
64     elap_time_mult.push_back(tm_mult.min());
65     // save elapsed time for plot: ttmatsolve vs ttrecsolve
66     elap_time_ttmult.push_back(tm_ttmult.min());
67     elap_time_ttrec.push_back(tm_ttrec.min());
68 }
69 /* DO NOT CHANGE */
70 // create plot
71 plot(vec_size, elap_time_mult, elap_time_matmult, "./cx_out/fig1.png",

```

```

72     "toepmult", "toepmatmult");
73     plot(vec_size, elap_time_ttrech, elap_time_ttmult, "./cx_out/fig2.png",
74         "ttrecsolve", "ttmatsolve");
75 }

```

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

This function produces the following pictures.

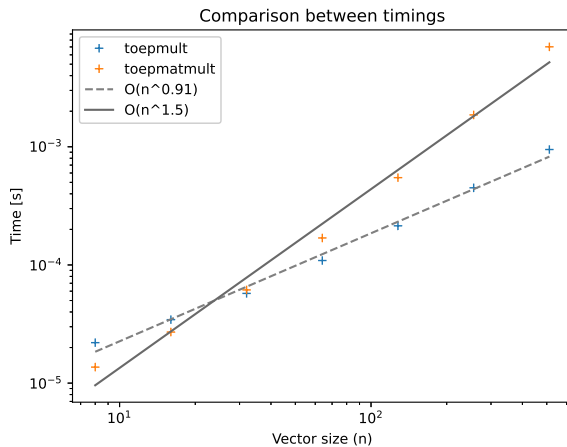


Fig. 28

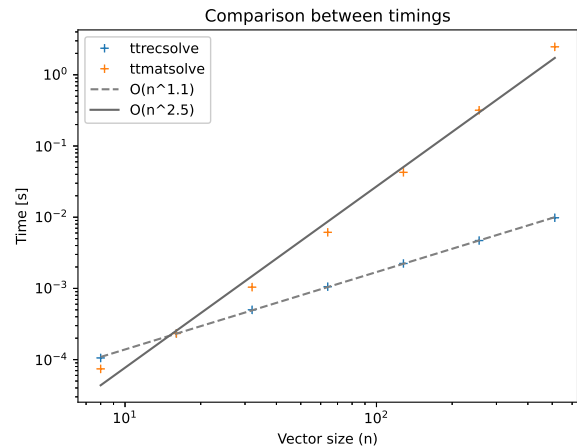


Fig. 29

For large problem sizes n , `ttrecsolve()` is much faster than `ttmatsolve()`. The runtime of the latter seems to grow like $O(n^3)$, whereas the former seems to scale (almost) linearly in n .



(4-4.h)  (20 min.) Derive the asymptotic complexity of both functions `ttmatsolve()` and `ttrecsolve()` in terms of the problem size parameter $n \rightarrow \infty$.

SOLUTION of (4-4.h):

For `ttmatsolve()`, the complexity is determined by the complexity of the solver employed, which is generally $O(n^3)$. Note that the implementation of Code 4.4.3 fails to communicate to EIGEN that the matrix is actually triangular. Thus EIGEN will blindly carry out full Gaussian elimination, cf. [Lecture → Rem. 2.5.0.5].

For `ttrecsolve()`, the number of operations in each step is given by the multiplication with the Toeplitz matrix and the vector subtraction in $\mathbf{x}_2 = \mathbf{H}_m^{-1}(\mathbf{y}_2 - \mathbf{T}_m \mathbf{x}_1)$. Thus, if we have a problem of size $\bar{n} = 2^{\bar{l}}$, we have

$$\approx \bar{n} \log(\bar{n}) + \bar{n} \approx 2^{\bar{l}} \bar{l} + 2^{\bar{l}} = 2^{\bar{l}} (\bar{l} + 1)$$

operations in this step. The first term comes from the complexity found in (4-4.d).

Now, we have to add these terms for all the steps. If the original problem is of size $n = 2^l$, recursion enters into l levels, and on each level $\bar{l} = 1, \dots, l$ we have $2^{\bar{l}}$ subproblems, each of size $2^{l-\bar{l}}$.

Thus, the total number of operations is:

$$\begin{aligned} \sum_{\bar{l}=1}^l 2^{l-\bar{l}} (l - \bar{l} + 1) 2^{\bar{l}} &= 2^l \sum_{\bar{l}=1}^l (l - \bar{l} + 1) \\ &= n \left(l^2 - \frac{l(l+1)}{2} + l \right) = n \frac{l(l+1)}{2} = O(n \log^2(n)) \quad \text{for } n \rightarrow \infty. \end{aligned}$$

In the last step we exploited the equality $l = \log_2(n) = \frac{\log(n)}{\log(2)}$.



(4-4.i) 🕒 (30 min.) For the case that $n = h.size()$ is **not** a power of 2, implement a *wrapper function*

```
VectorXd ttsolve(const VectorXd &h, const VectorXd &y);
```

for `ttrecsolve` that, in the absence of roundoff errors, would behave exactly like `ttmatsolve`, i.e. `ttsolve(h,y) == ttmatsolve(h,y)`. The term “wrapper function” indicates that `ttsolve()` is supposed to call `ttrecsolve()` after preparing suitable input vectors.

HINT 1 for (4-4.i): Just note that a lower triangular linear system of equations can be extended to a larger lower triangular LSE by adding additional rows/columns at the bottom/right of the coefficient matrix without affecting the first components of the solution vector. Thus, a triangular Toeplitz LSE can always be enlarged to a triangular Toeplitz LSE of a size that is a power of 2. ┘

SOLUTION of (4-4.i):

The wrapper function has to provide a solution $\mathbf{H}_n \mathbf{x} = \mathbf{y}$ for the problem sizes n which are not powers of 2. If we have a problem of size m and $n = 2^l$ is the next larger power of 2, we can choose the following extension:

$$\mathbf{H} \mathbf{x}_{\text{long}} = \begin{bmatrix} \mathbf{H}_m & \mathbf{0} \\ \mathbf{X}_{n-m} & \mathbf{H}_{n-m} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \tilde{\mathbf{x}} \end{bmatrix} = \begin{bmatrix} \mathbf{y} \\ \tilde{\mathbf{y}} \end{bmatrix} = \mathbf{y}.$$

Here the matrix $\mathbf{X} \in \mathbb{R}^{n-m,m}$ has to be chosen such that $\mathbf{H} \in \mathbb{R}^{n,n}$ becomes a lower triangular Toeplitz matrix. The solution of this linear system, which can efficiently be computed with `ttrecsolve()` is:

$$\begin{aligned} \mathbf{x} &= \mathbf{H}_m^{-1} \mathbf{y}, \\ \tilde{\mathbf{x}} &= \mathbf{H}_{n-m}^{-1} (\tilde{\mathbf{y}} - \mathbf{T}_{n-m} \mathbf{y}), \end{aligned}$$

and $\mathbf{x}$ is what we are looking for.

Thus, we must find a suitable extension $\mathbf{H}_{n-m}$ and $\mathbf{X}$ such that $\mathbf{H}$ is a lower triangular Toeplitz matrix of the right size n . The solution of the original system of equations would then be available as the first m entries of $\mathbf{x}_{\text{long}}$.

$\mathbf{H}$ is therefore the $n \times n$ triangular Toeplitz matrix whose vector $\mathbf{h}$ of elements along the diagonal is determined by $\mathbf{h}_m$ extended with $n - m$ zeros.

For example, in the case $m = 3$ and given

$$\mathbf{H}_m = \begin{bmatrix} h_1 & 0 & 0 \\ h_2 & h_1 & 0 \\ h_3 & h_2 & h_1 \end{bmatrix},$$

we have that $n = 4$ and

$$\mathbf{H} = \begin{bmatrix} h_1 & 0 & 0 & 0 \\ h_2 & h_1 & 0 & 0 \\ h_3 & h_2 & h_1 & 0 \\ 0 & h_3 & h_2 & h_1 \end{bmatrix}.$$

C++ code 4.4.9: Solution of (4-4.i)

```

2  Eigen::VectorXd ttsolve(const Eigen::VectorXd &h, const Eigen::VectorXd &y) {
3      assert(h.size() == y.size() && "h and y have different lengths!");
4      int n = h.size();
5
6      Eigen::VectorXd x;
7
8      // TODO: wrap 'ttrecsolve' for any size n
9      // START
10     int l = std::ceil(std::log2(n));
11     int m = std::pow(2, l);
12
13     Eigen::VectorXd h_tmp = h;
14     h_tmp.conservativeResize(m);
15     h_tmp.bottomRows(m - n).setZero();
16
17     Eigen::VectorXd y_tmp = y;
18     y_tmp.conservativeResize(m);
19     y_tmp.bottomRows(m - n).setZero();
20
21     x = ttrecsolve(h_tmp, y_tmp, l);
22     x.conservativeResize(n);
23     // END
24
25     return x;
26 }

```

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



End Problem 4-4, 145 min.