

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 1-3: Kronecker product

In [Lecture → Def. 1.4.3.7] we learned about the so-called **Kronecker product**. In this problem we revisit the discussion of [Lecture → Ex. 1.4.3.8].

This problem is connected with [Lecture → Section 1.4.3].

Definition [Lecture → Def. 1.4.3.7]. Kronecker product

The **Kronecker product** $\mathbf{A} \otimes \mathbf{B}$ of two matrices $\mathbf{A} \in \mathbb{K}^{m,n}$ and $\mathbf{B} \in \mathbb{K}^{l,k}$, $m, n, l, k \in \mathbb{N}$, is the $(ml) \times (nk)$ -matrix

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} (\mathbf{A})_{11}\mathbf{B} & (\mathbf{A})_{12}\mathbf{B} & \dots & \dots & (\mathbf{A})_{1,n}\mathbf{B} \\ (\mathbf{A})_{2,1}\mathbf{B} & (\mathbf{A})_{2,2}\mathbf{B} & & & \vdots \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ (\mathbf{A})_{m,1}\mathbf{B} & (\mathbf{A})_{m,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{m,n}\mathbf{B} \end{bmatrix} \in \mathbb{K}^{ml,nk}.$$

(1-3.a)  (10 min.) Compute the Kronecker product $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ of the 2×2 matrices

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}. \quad (1.3.1)$$

SOLUTION of (1-3.a):

The matrix $\mathbf{C}$ is:

$$\mathbf{C} = \begin{bmatrix} 5 & 6 & 10 & 12 \\ 7 & 8 & 14 & 16 \\ 15 & 18 & 20 & 24 \\ 21 & 24 & 28 & 32 \end{bmatrix}.$$



(1-3.b)  (20 min.)

Based on EIGEN, in the file `kron.hpp` implement a C++ function

```
Eigen::MatrixXd kron(const Eigen::MatrixXd & A, const
Eigen::MatrixXd & B);
```

that computes the Kronecker product of the argument matrices $\mathbf{A}$ and $\mathbf{B}$ and returns the resulting matrix.

HINT 1 for (1-3.b): You can use Eigen “block” operations, see the 🐘 [EIGEN documentation](#). 

SOLUTION of (1-3.b):

The Kronecker product has the structure:

$$\mathbf{C} = \begin{bmatrix} a_{1,1}\mathbf{B} & \dots & a_{1,n}\mathbf{B} \\ \vdots & & \vdots \\ a_{n,1}\mathbf{B} & \dots & a_{n,n}\mathbf{B} \end{bmatrix}.$$

Here, each $a_{(i,j)} \mathbf{B} \in \mathbb{R}^{n \times n}$ (for $0 < i, j \leq n$) is a Matrix (we will call this matrix a block). The proposed code loops over each block and stores the result of the product withing the matrix.

C++-code 1.3.2: Kronecker product

```

2  Eigen::MatrixXd kron(const Eigen::MatrixXd &A, const Eigen::MatrixXd &B) {
3      // Allocate enough space for the matrix
4      Eigen::MatrixXd C = Eigen::MatrixXd(A.rows() * B.rows(), A.cols() * B.cols());
5      // TODO: (1-3.b) Fill in the entries of C.
6      // Hint: Use a nested for-loop and C.block().
7      // START
8      for (unsigned int i = 0; i < A.rows(); ++i) {
9          for (unsigned int j = 0; j < A.cols(); ++j) {
10             // We use eigen block operations to set the values of
11             // each  $n \times n$  block.
12             C.block(i * B.rows(), j * B.cols(), B.rows(), B.cols()) =
13                 A(i, j) * B; //  $\in \mathbb{R}^{(n \times n)}$ 
14         }
15     }
16     // END
17
18     return C;
19 }

```

Get it on  GitLab (kron.hpp).

In fact, EIGEN itself provides an implementation of the Kronecker product, see  [EIGEN documentation](#).

(1-3.c) (10 min.) [depends on Sub-problem (1-3.b)]

What is the **asymptotic complexity** ([Lecture $\rightarrow$ Def. 1.4.1.1]) in terms of the problem size parameter $n \rightarrow \infty$ of your implementation of `kron` from (1-3.b), when we pass $n \times n$ square matrices as arguments.

You may use the *Landau* symbol from [Lecture $\rightarrow$ Def. 1.4.1.2] to state your answer.

SOLUTION of (1-3.c):

The code loops two times from 0 to n , i.e. is performing the inner portion of the code n^2 times. Within the for loop, the code is multiplying a $n \times n$ matrix by a scalar, this operation has complexity $O(n^2)$. In total, the complexity is $O(n^2 \cdot n^2) = O(n^4)$.

(1-3.d) (30 min.)

In general, computing the entire matrix is unnecessary. Devise an *efficient* implementation of an EIGEN-based C++ function (in the file `kron.hpp`)

```

Eigen::VectorXd kron_mult(
    const Eigen::MatrixXd & A, const Eigen::MatrixXd & B,
    const Eigen::VectorXd & x);

```

for the computation of $\mathbf{y} = (\mathbf{A} \otimes \mathbf{B})\mathbf{x}$ for *square matrices* $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,n}$. Do not use reshaping through the `Map` method of EIGEN's matrix classes; use access to sub-vectors by the `segment` method instead. The meaning of the arguments of `kron_mult` should be self-explanatory.

HINT 1 for (1-3.d): You can exploit the structure of the matrix and reuse computations as much as possible. The tricks how to do this are explained in detail in [Lecture → Ex. 1.4.3.8].

SOLUTION of (1-3.d):

Define $\tilde{\mathbf{x}}_j := (x_{j \cdot n + 1}, \dots, x_{(j+1) \cdot n})^\top \in \mathbb{R}^n$ (a “segment” of $\mathbf{x}$) for $j = 0, \dots, n-1$. Notice the following:

$$\mathbf{C}\mathbf{x} = \begin{bmatrix} a_{1,1}\mathbf{B} & \dots & a_{1,n}\mathbf{B} \\ \vdots & & \vdots \\ a_{n,1}\mathbf{B} & \dots & a_{n,n}\mathbf{B} \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{x}}_0 \\ \vdots \\ \tilde{\mathbf{x}}_{n-1} \end{bmatrix} = \begin{bmatrix} a_{1,1}\mathbf{B}\tilde{\mathbf{x}}_0 + \dots + a_{1,n}\mathbf{B}\tilde{\mathbf{x}}_{n-1} \\ \vdots \\ a_{n,1}\mathbf{B}\tilde{\mathbf{x}}_0 + \dots + a_{n,n}\mathbf{B}\tilde{\mathbf{x}}_{n-1} \end{bmatrix}.$$

The computation of $\mathbf{z}_j := \mathbf{B}\tilde{\mathbf{x}}_j$ needs to be carried out only once and the vectors $\mathbf{z}_j$ can be reused.

C++-code 1.3.3: Kronecker product times vector

```

2  Eigen::VectorXd kron_mult(const Eigen::MatrixXd &A, const Eigen::MatrixXd &B,
3                          const Eigen::VectorXd &x) {
4      assert(A.rows() == A.cols() && A.rows() == B.rows() && B.rows() == B.cols() &&
5             "Matrices A and B must be square matrices with same size!");
6      assert(x.size() == A.cols() * A.cols() &&
7             "Vector x must have length A.cols()^2");
8      const unsigned int n = A.rows();
9
10     // Allocate space for output
11     Eigen::VectorXd y = Eigen::VectorXd::Zero(n * n);
12
13     // TODO: (1-3.d) Fill in the entries of y.
14     // Hint: Use a nested for-loop, x.segment(), and y.segment().
15     // In the outer loop, you can perform a computation based on
16     // B and x, and save the result in a variable that is reused in
17     // each iteration of the inner loop.
18     // START
19
20     // Note: this is like doing a matrix-vector multiplication
21     // where the entries of the matrix are smaller matrices
22     // and entries of the vector are smaller vectors
23
24     // Loop over all segments of x (x̃)
25     for (unsigned int j = 0; j < n; ++j) {
26         // Reuse computation of z
27         Eigen::VectorXd z = B * x.segment(j * n, n);
28         // Loop over all segments of y
29         for (unsigned int i = 0; i < n; ++i) {
30             y.segment(i * n, n) += A(i, j) * z;
31         }
32     }
33     // END
34
35     return y;
36 }
```

Get it on  GitLab (kron.hpp).



(1-3.e) 🕒 (30 min.) Devise another efficient implementation of a C++ code for the computation of $y = (A \otimes B)x$, $A, B \in \mathbb{R}^{n,n}$. This time use EIGEN's "matrix reshaping" functions. The function to implement in the file `kron.hpp` is

```
Eigen::VectorXd kron_reshape (
    const Eigen::MatrixXd &A, const Eigen::MatrixXd &B,
    const Eigen::VectorXd &x);
```

As its counterpart `kron_mult()` from Sub-problem (1-3.d) it is supposed to compute $y = (A \otimes B)x$.

HINT 1 for (1-3.e): Study [Lecture → Rem. 1.2.3.6] about "reshaping" matrices in EIGEN. See also the 🐘 [EIGEN documentation](#). └

HINT 2 for (1-3.e): In EIGEN, reshaping a matrix is done using the `MatrixXd::Map` class, see and 🐘 [EIGEN documentation](#) and 🐘 [EIGEN documentation](#). A constructor for the `Map` class takes three arguments. The first is a pointer to the data, and the second and third are the shape of the matrix represented by `Map`. Important: since `Map` is accessing a raw-pointer to the data, it is very important that the data is stored in column-major format. This is because the `Map` class has no idea of the underlying structure of the matrix. └

SOLUTION of (1-3.e):

See also [Lecture → Code 1.4.3.9]. More explanations are given there.

C++-code 1.3.4: Kronecker product times vector using Map ()

```
2 Eigen::VectorXd kron_reshape(const Eigen::MatrixXd &A, const Eigen::MatrixXd &B,
3                             const Eigen::VectorXd &x) {
4     assert(A.rows() == A.cols() && A.rows() == B.rows() && B.rows() == B.cols() &&
5           "Matrices A and B must be square matrices with same size!");
6     const unsigned int n = A.rows();
7
8     // TODO: (1-3.e) Fill in the entires of y.
9     // Hint: Use Eigen::MatrixXd::Map() to reshape x into a n by n matrix.
10    // Then y is obtained by simple matrix multiplications and
11    // another reshape.
12    // START
13    Eigen::MatrixXd t = B * Eigen::MatrixXd::Map(x.data(), n, n) * A.transpose();
14    Eigen::VectorXd y = Eigen::MatrixXd::Map(t.data(), n * n, 1);
15    // END
16
17    return y;
18 }
```

Get it on 🐘 [GitLab](#) (`kron.hpp`).



(1-3.f) 🕒 (30 min.) Compare the runtimes of your implementations in sub-problems (1-3.b), (1-3.d) and (1-3.e). To this end, measure the runtime of the functions `kron`, `kron_mult`, `kron_reshape`, with $n = 2^1, \dots, 2^8$. Repeat the experiments 10 times and consider only the smallest runtime (skip the computations with `kron` for $n > 2^6$). You can use the class `Timer` or the STL class `std::chrono` [C++ reference](#).

Report the measurements in seconds with scientific notation using 3 decimal digits.

SOLUTION of (1-3.f):

The following code is an example of runtime computation:

C++-code 1.3.5: Runtime comparison

```

2 void kron_runtime() {
3     Eigen::MatrixXd A, B, C;
4     Eigen::VectorXd x, y;
5     // We repeat each runtime measurement 10 times
6     // (this is done in order to remove outliers).
7     constexpr unsigned int repeats = 10;
8
9     std::cout << "Runtime for each implementation." << std::endl;
10    std::cout << std::setw(5) << "n" << std::setw(15) << "kron" << std::setw(15)
11        << "kron_mult" << std::setw(15) << "kron_reshape" << std::endl;
12    // Loop from M = 2, ..., 2^8
13    for (unsigned int M = 2; M <= (1 << 8); M = M << 1) {
14        Timer tm_kron, tm_kron_mult, tm_kron_map;
15        // Run experiments "repeats" times
16        for (unsigned int r = 0; r < repeats; ++r) {
17            // Random matrices for testing
18            A = Eigen::MatrixXd::Random(M, M);
19            B = Eigen::MatrixXd::Random(M, M);
20            x = Eigen::VectorXd::Random(M * M);
21
22            // Do not want to use kron for large values of M
23            if (M < (1 << 6)) {
24                // Kron using direct implementation
25                tm_kron.start();
26                C = kron(A, B);
27                y = C * x;
28                tm_kron.stop();
29            }
30
31            // TODO: (1-3.f) Measure the runtime of kron_mult() and kron_reshape().
32            // START
33
34            // Kron matrix-vector multiplication
35            tm_kron_mult.start();
36            y = kron_mult(A, B, x);
37            tm_kron_mult.stop();
38
39            // Kron using reshape
40            tm_kron_map.start();
41            y = kron_reshape(A, B, x);
42            tm_kron_map.stop();
43
44            // END
45        }
46
47        double kron_time = (M < (1 << 6)) ? tm_kron.min() : std::nan("");
48        std::cout << std::setw(5) << M << std::scientific << std::setprecision(3)
49            << std::setw(15) << kron_time << std::setw(15)
50            << tm_kron_mult.min() << std::setw(15) << tm_kron_map.min()
51            << std::endl;
52    }

```

53 | }

Get it on  GitLab (kron.hpp).

n	kron	kron_mult	kron_reshape
2	6.485e-06	6.431e-06	3.950e-06
4	3.063e-05	1.913e-05	6.748e-06
8	3.000e-04	8.111e-05	2.869e-05
16	5.489e-03	4.229e-04	1.973e-04
32	9.057e-02	2.993e-03	1.590e-03
64	nan	2.474e-02	1.368e-02
128	nan	1.847e-01	1.243e-01
256	nan	1.390e+00	9.089e-01

The runtime measurements were run on a Huawei notebook with a Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz compiled without any optimization flags using the default NumCSE build system.

On your system, the runtimes might vary but the general trend should be similar.

Observations

- From Sub-problem (1-3.c) we know that `kron()` grows as $O(n^4)$ for $n \rightarrow \infty$. Thus it is not surprising that the runtimes `kron()` grow fast with increasing problem size.
- The runtime of `kron_mult()` increases much more moderately, because we took advantage of trick presented in [Lecture $\rightarrow$ Ex. 1.4.3.8]. We observe a growth of the runtimes as $O(n^3)$.
- `kron_reshape()` uses corresponding EIGEN methods which are highly optimized, explaining the slightly lower runtimes compared to `kron_mult()`. Also note the absence of loops. Its runtimes increase like $O(n^3)$ for $n \rightarrow \infty$.

It is good practice to check your results: ([Lecture $\rightarrow$ Def. 1.4.1.1]).

For `kron()` we can confirm $T(32)/T(16) = 9.057e-02/5.489e-03 = 16.5 \sim 2^4$.

For `kron_mult()` we can confirm $T(256)/T(128) = 1.390e+00/1.847e-01 = 7.5 < 2^3$.

For `kron_reshape()` we can confirm $T(256)/T(128) = 9.089e-01/1.243e-01 = 7.3 < 2^3$



End Problem 1-3, 130 min.