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

Chapter 0

Introduction

Problem 0-1: Simple operations with vectors and matrices in EIGEN

The problem encourages first steps in C++ coding using the EIGEN template library. The purpose of this exercise is to learn to declare and initialize **Eigen::Matrix** objects, and to get familiar with some handy typedefs and methods.

This problem involves coding in C++ and is related to [Lecture → Section 1.2.1].

The central data type in EIGEN is the matrix, of which vectors are just a special version. Matrix data type fall into three fundamental categories

(I) Variable-size and resizable dense matrices declared with the template argument `Eigen::Dynamic`,

(II) Fixed-size (small) matrices, whose size must be known at compile time,

(III) Variable size sparse matrices that are stored in special formats and which will be treated in [Lecture → Section 2.7.2].

Remark. For all codes in this problem, there are many ways how to meet the specification. Please think about it and discuss alternatives.

(0-1.a) ⚀ (10 min.) Which header file has to be included so that a code can use the (templated) type **Eigen::Matrix**?

SOLUTION of (0-1.a):

You have to add **#include** `<Eigen/Dense>` in the beginning of every file that uses EIGEN tools. Of course the compiler must be told the directory, where EIGEN's header files reside.

▲

(0-1.b) ⚂ (15 min.) Write a C++ function

```
Eigen::Matrix<double, 2, 2>
smallTriangular( double a, double b, double c );
```

that returns the 2×2-matrix $\begin{bmatrix} a & b \\ 0 & c \end{bmatrix}$.

HINT 1 for (0-1.b): The test case implemented in `main.cpp` is `smallTriangular(1,2,3)` and returns

$$\begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix}.$$

⌟

SOLUTION of (0-1.b):

Matrix initialization with the << operator [Lecture → § 1.2.1.3] is convenient. However also the entry access operator **double &operator (int, int)** can be used directly.

C++ code 0.1.1: Initialization of fixed size 2×2-matrix

```
2  Eigen::Matrix<double, 2, 2> smallTriangular(double a, double b, double c) {
3    // We know in advance that we want to create a matrix of doubles with a fixed
4    // size of 2 by 2.  Therefore, we pass the parameters <double, 2, 2> to the
5    // template class Eigen::Matrix.
6    Eigen::Matrix<double, 2, 2> A;
7
8    // TODO: (0-1.b) Initialize the matrix A.
9    // START
10   // We have declared the variable A of the type Eigen::Matrix<double,2,2>,
11   // but we have not initialized its entries.
12   // We can do this using comma-initialization:
13   A << a, b, 0, c;
14   // END
15
16   // Question:  Is A now an upper triangular matrix, or a lower triangular
17   // matrix?
18
19   return A;
20 }
```

Get it on GitLab (MatrixClass.hpp).

▲

(0-1.c) ⚀ (15 min.) Now, again in the file `MatrixClass.hpp`, implement the C++ function

```
Eigen::MatrixXd constantTriangular( int n, double val );
```

that initializes an $n \times n$ upper triangular matrix [Lecture → Def. 1.1.2.3], whose non-zero entries all contain the value passed in the `val` argument.

HINT 1 for (0-1.c): The test call `constantTriangular(3,20)` from `main.cpp` should yield the matrix

$$\begin{bmatrix} 20 & 20 & 20 \\ 0 & 20 & 20 \\ 0 & 0 & 20 \end{bmatrix}.$$

⌟

SOLUTION of (0-1.c):

Of course, one can simply set the matrix entries using the direct access to matrix entries and two nested loops.

C++ code 0.1.2: Initialization of upper triangular part of a matrix

```
2  Eigen::MatrixXd constantTriangular(unsigned int n, double val) {
3    // Now we do not know the size of our matrix at compile time.
4    // Hence, we use the special value Eigen::Dynamic to set the size of A.
5    Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> A;
6
7    // Eigen::Matrix has a method Zero(n_rows,n_cols) which returns the n_rows by
8    // n_cols matrix whose entries are all equal to zero.
9    A = Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>::Zero(n, n);
10
11   // To get a triangular matrix, we must set the values above the diagonal to
12   // val. This we can do by a nested for-loop.
13   for (unsigned int row = 0; row < n; ++row) {
14     // TODO: (0-1.c) Write the inner for-loop to initialize A.
15     // Hint: We can access and change the entries of A using parentheses, i.e.
16     // A(row,col) = val; Note that indexing starts at the value 0 (as usual),
17     // and not 1 (as in Matlab).
18
19     // START
20     for (unsigned int col = row; col < n; ++col) {
21       A(row, col) = val;
22     }
23     // END
24   }
25
26   // instead of the for-loop initialization, triangularView can be used
27   // A.triangularView<Eigen::Upper>() = Eigen::MatrixXd::Constant(n, n, val);
28
29   return A;
30 }
```

Get it on GitLab (MatrixClass.hpp).

Alternative: One can also use the `triangularView()` member function for **Eigen::Matrix**, see [Lecture → Code 1.2.1.6], EIGEN documentation.

▲

(0-1.d) ⚀ (15 min.) Vectors and matrices in EIGEN can also have integer and complex entries. The corresponding types have to be tagged with the `i` and `cd` suffix, respectively, see [Lecture → § 1.2.1.1]. Operations involving matrices/vectors with different types of entries usually entail type casting.

In the file `MatrixClass.hpp` supplement the missing parts of the function `casting()` such that it returns the real part of the Euclidean (standard) scalar product of the vectors `u` and `v`. To that end use EIGEN's dot product accessible via the method `dot()` of **Eigen::Vector**.

SOLUTION of (0-1.d):

If in EIGEN you want to mix matrices/vectors with different scalar types in arithmetic expressions, you have to do explicit typecasts using the `cast()` methods.

C++ code 0.1.3: Casting the scalar type of a matrix

```
double casting() {
  // Because the syntax Eigen::Matrix<type, n_rows, n_cols> is very
  // cumbersome, Eigen provides convenience classes that work as shorthands.
  // For example, Eigen::Matrix2d is shorthand for Eigen::Matrix<double, 2, 2>.

  // Vectors are just a special type of matrices that have only one column.
  // Thus, VectorXi is a shorthand for Eigen::Matrix<int, Eigen::Dynamic, 1>.
  // Constant(n,x) creates a vector of size n and initializes the entries with
  // the value x.
  Eigen::VectorXi u = Eigen::VectorXi::Constant(2, 2);

  // std::complex is a template class for complex numbers.
  // Here we declare two complex numbers, with real and imaginary parts
  // represented by doubles. z0 = 1 - i z1 = 5 + i
  std::complex<double> z0(1, -1), z1(5, 1);

  // Declare and initialize a size 2 vector of std::complex<double>.
  Eigen::Vector2cd v;
  v(0) = z0;
  v(1) = z1;

  double x = 0.0;
  // TODO: (0-1.d) Calculate the inner product of u and v, and store the result
  // in x. Hint: The inner product of two vectors is given by u.dot(v), but
  // first we need to cast the "int" vector u to a "std::complex<double>"
  // vector. Use u.cast< NEW TYPE >() to achieve this. The result of the inner
  // product will be 1*(1-i) + 1*(5+i) = 6 + 0i, a real number. You can get the
  // real part of an std::complex<double> using the method "real()".
  // START
  std::complex<double> z = u.cast<std::complex<double>>().dot(v);
  x = z.real();
  // END
  return x;
}
```

Get it on GitLab (MatrixClass.hpp).

▲

(0-1.e) (20 min.) Given $n \in \mathbb{N}$ the matrices $\mathbf{A} \in \mathbb{R}^{n,n}$ and $\mathbf{B} \in \mathbb{C}^{n,n}$ are defined as follows

$$(\mathbf{A})_{k,\ell} := \begin{cases} 5 & \text{, if } k \geq \ell \text{ ,} \\ 0 & \text{, else} \end{cases} \quad , \quad (\mathbf{B})_{k,\ell} := \frac{k + \imath\ell}{k - \imath\ell} \quad , \quad 1 \leq k, \ell \leq n \text{ ,}$$

where $\imath \in \mathbb{C}$ is the imaginary unit, $\imath^2 = -1$. In the file `MatrixClass.hpp` realize complete the C++ function

```
Eigen::VectorXcd arithmetics(int n);
```

that evaluates the expression $\mathbf{B}(\mathbf{A} - 5\mathbf{I})[1, 2, \ldots, n]^\top \in \mathbb{C}^n$.

HINT 1 for (0-1.e): The test call in `main.cpp` is `arithmetics(4)` and should return the vector

$$\begin{bmatrix} -41.4706 + 27.1176\imath \\ -23.7692 + 42.8462\imath \\ -6.47692 + 48.4154\imath \\ 7.2 + 48.4\imath \end{bmatrix} .$$

SOLUTION of (0-1.e):

Refer to the comments in the code and notice the use of the `cast()` method.

C++ code 0.1.4: Evaluating a simple matrix-vector expression

```
2  Eigen::VectorXcd arithmetics(unsigned int n) {
3    // This declares dynamic size (signified by the letter 'X') matrices of
4    // complex numbers (signified by the letter c) with real and imaginary
5    // parts represented by doubles (signified by 'd').
6    Eigen::MatrixXcd A, C, I;
7
8    // We initialize the matrices arbitrarily:
9    // an n by n lower triangular matrix,
10   A = constantTriangular(n, 5.0).transpose().cast<std::complex<double>>();
11   // The n by n identity matrix,
12   I = Eigen::MatrixXcd::Identity(n, n);
13
14   // Declare the n by n matrix B.
15   Eigen::MatrixXcd B(n, n);
16
17   // TODO: (0-1.e) Fill in the matrix B such that B(k-1,l-1) = (k+l*i)/(k-l*i),
18   // where i is the imaginary unity, i^2 = -1 and k,l=1,...,n.
19   // START
20   for (unsigned int k = 0; k < n; ++k) {
21     for (unsigned int l = 0; l < n; ++l) {
22       std::complex<double> tmp(k + 1, l + 1);
23       B(k, l) = tmp / std::conj(tmp);
24     }
25   }
26   // END
27
28   // We can perform arithmetics on matrices: +, -, *
29   // Note that for + and -, the left hand matrix and the right hand matrix have
30   // to be the same size, and that matrix multiplication * is only defined if
31   // the number of columns in the left hand matrix is the same as the number of
32   // rows in the right hand matrix.
33   C = B * (A - 5.0 * I);
34
35   // Eigen::VectorXcd is shorthand for
36   // Eigen::Matrix< std::complex<double>, Eigen::Dynamic, 1>.
37   // Hence, all the same arithmetics work for vectors.
38   Eigen::VectorXcd u, v;
39
40   // TODO: (0-1.e) Initialize the entries of u as the integers from 1 to n,
41   // multiply u by the matrix C, and store the result in v.
42   // START
43   u = Eigen::VectorXcd::LinSpaced(n, 1, n);  // Or use a for-loop.
44   v = C * u;
45   // END
46
47   return v;
48 }
```

Get it on GitLab (MatrixClass.hpp).

▲

End Problem 0-1 , 75 min.