

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 0-3: EIGEN's reduction operations on matrices

EIGEN's matrices have a range of methods that reduce them to a vector or even a single number, see  [EIGEN documentation](#). We will also learn about the class template **Eigen::Array**, see  [EIGEN documentation](#).

The problem practises the use of EIGEN and can be regarded as a supplement to [Lecture → Section 1.2.1].

Remark. All the coding tasks can be tackled in many different ways. The codes provided as master solution just demonstrate one way.

All functions you will be asked to complete reside in the file `MatrixReduce.hpp`.

(0-3.a)  (15 min.) Implement the C++ function

```
double average(const MatrixXd &A);
```

that computes the average

$$\frac{1}{mn} \sum_{i=1}^n \sum_{j=1}^m (A)_{i,j}$$

of the entries of the matrix $A \in \mathbb{R}^{n,m}$, $n, m \in \mathbb{N}$. That matrix is passed as argument `A`.

HINT 1 for (0-3.a): Read about the `sum()` method available for matrices in EIGEN, see  [EIGEN documentation](#). ┘

HINT 2 for (0-3.a): From `main.cpp` we call `average(Matrix3d::Identity())` and should get `0.333333`. ┘

SOLUTION of (0-3.a):

C++ code 0.3.1: Computing the average of matrix entries

```
2 double average(const Eigen::MatrixXd& A) {
3   double m = 0;
4
5   // TODO: (0-3.a) Calculate the average of the entries of A.
6   // START
7   m = A.sum() / A.size();
8   // END
9
10  return m;
11 }
```

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



(0-3.b)  (20 min.) Supply the missing parts of the C++ function

```
double percent_zero(const MatrixXd &A);
```

the computes the percentage of (exact) zeros among the entries of the matrix object `A`.

HINT 1 for (0-3.b): Gather information about **Boolean reductions** from  **EIGEN documentation**. ┘

HINT 2 for (0-3.b): The test call `percent_zero(Matrix3d::Identity())` in `main.cpp` should result in **66.6667**. ┘

SOLUTION of (0-3.b):

Note the expression `(Arr == 0).count()` and that the `size()` method returns the total number of entries for a dense matrix.

C++ code 0.3.2: Determines the percentage of (exactly) zero entries

```

2  double percent_zero(const Eigen::MatrixXd& A) {
3      // Eigen provides an Array class, that is similar to Matrix,
4      // but has operations that are entry-wise rather than linear algebra
5      // operations. For example, multiplying two Arrays is entry-wise
6      // multiplication, and not matrix multiplication.
7      // The benefit of using an Array here is that we can do entry-wise
8      // comparison, which is not available for matrices.
9      Eigen::ArrayXXd Arr = A.array();
10
11     double ratio = 0.;
12     // TODO: (0-3.b) Calculate the ratio of zero entries in A.
13     // START
14     const unsigned int zeros = (Arr == 0).count();
15     ratio = (static_cast<double>(zeros)) / A.size();
16     // END
17
18     return ratio * 100;
19 }

```

Get it on  **GitLab** (`MatrixReduce.hpp`).

(0-3.c)  (20 min.) Complete the implementation of the function

```
bool has_zero_column(const MatrixXd &A);
```

that tests whether the matrix passed in `A` has a column that is exactly zero.

HINT 1 for (0-3.c): You can use **partial reductions** for your code, see  **EIGEN documentation**. ┘

HINT 2 for (0-3.c): For `B=MatrixXd::Identity(4, 5)` `has_zero_column(B)` should yield `true`, while `has_zero_column(B.transpose())` must return `false`. This is the test implemented in `main.cpp`. ┘

SOLUTION of (0-3.c):

EIGEN's `any()` method checks whether an expression evaluates to **true** for a single entry of a matrix.

C++ code 0.3.3: Checking for an exactly vanishing column

```

2  bool has_zero_column(const Eigen::MatrixXd& A) {
3      bool result;
4

```

```

5 | // A vector is the zero vector if and only if it has norm 0.
6 | // The following vector contains the squared norms of the columns of
7 | // A.
8 | Eigen::VectorXd norms = A.colwise().squaredNorm();
9 | // The norm is 0 if and only if the norm squared is zero.
10 | // We use squaredNorm() instead of norm(), because norm() =
11 | // sqrt(squaredNorm()) calculates a square root that is not
12 | // necessary for our
13 | // purposes.
14 | // TODO: (0-3.c) Check if any one of the norms is equal to zero.
15 | // Hint: Use an array to perform entry-wise comparison.
16 | // START
17 | result = (norms.array() == 0).any();
18 | // END
19 |
20 | return result;

```

Get it on  GitLab (MatrixReduce.hpp).

(0-3.d)  (25 min.) Finish the implementation of the C++ function

```
Eigen::MatrixXd columns_sum_to_zero(const Eigen::MatrixXd &A);
```

that returns a matrix that agrees with the *square* matrix passed in A except for the diagonal. Its diagonal entries are set in way that makes all column sums vanish.

HINT 1 for (0-3.d): The test case implemented in `main.cpp` calls `columns_sum_to_zero(C)` where `C=MatrixXd::Random()+MatrixXd::Constant(1)`. Note that the matrix **C** is not really random, because the seed for the “random” number generator is set to a particular value. Thus we always get the matrix

$$\begin{bmatrix} -0.721194 & 0.160061 & 0.561133 \\ 0.0929355 & -0.989846 & 0.896911 \\ 1.98551 & 0.159289 & -2.1448 \end{bmatrix}$$

SOLUTION of (0-3.d):

The solution gives a very compact implementation using sub-matrix access via `diagonal()` and the `rowwise` modifier for an ensuing reduction operation.

C++ code 0.3.4: Setting diagonals to make row sums vanish

```

2 | Eigen::MatrixXd columns_sum_to_zero(const Eigen::MatrixXd& A) {
3 |     Eigen::MatrixXd B(A);
4 |
5 |     // TODO: (0-3.d) Replace the diagonal of B with values such that the
6 |     // of B sum up to zero. Hint: Use diagonal(), rowwise(), and
7 |     // sum().
8 |     // START
9 |     unsigned int p = std::min(B.rows(), B.cols());
10 |    B.diagonal() = Eigen::VectorXd::Zero(p);

```

```
10 | B.diagonal() = -B.rowwise().sum();  
11 | // END  
12 |  
13 | return B;  
14 | }
```

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



End Problem 0-3, 80 min.