

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 1-2: Gram-Schmidt orthonormalization with EIGEN

In this exercise we rely on EIGEN to implement a fundamental algorithm from linear algebra, Gram-Schmidt orthonormalization, see [NIS02] or [Lecture → § 1.5.1.1], in order to practice the use of EIGEN's basic vector and matrix operations.

This problem relies on [Lecture → Ex. 0.3.5.29], involves C++ implementation with EIGEN.

In [Lecture → Ex. 0.3.5.29] and [Lecture → Code 0.3.5.30] you can find an implementation of the famous Gram-Schmidt orthonormalization algorithm from linear algebra. That code makes use of a rather simple (and inefficient) implementation of vector and matrix classes. In this exercise we want to use EIGEN to implement Gram-Schmidt orthonormalization.

(1-2.a)  (15 min.) Determine the asymptotic complexity of the function `gramschmidt()` from [Lecture → Code 0.3.5.30] in terms of the matrix dimension n , if the argument matrix A has size $n \times n$ and no premature termination occurs.

SOLUTION of (1-2.a):

C++ code [Lecture → Code 0.3.5.30]: Templated function for Gram-Schmidt orthonormalization → [GITLAB](#)

```

1  template <typename Vec>
2  std::vector<Vec> gramscmidt(const std::vector<Vec> &A, double eps=1E-14) {
3      const int k = A.size(); // no. of vectors to orthogonalize
4      const int n = A[0].size(); // length of vectors
5      cout << "gramschmidt orthogonalization for " << k << ' ' << n << "-vectors" <<
6          endl;
7      std::vector<Vec> Q({A[0]/A[0].norm()}); // output vectors
8      for(int j=1; (j<k) && (j<n); ++j) {
9          Q.push_back(A[j]);
10         for(int l=0; l<j; ++l) Q.back() -= (A[j]*Q[l]) * Q[l];
11         if (Q.back().norm() < eps * A[j].norm()) { // premature termination ?
12             Q.pop_back(); break;
13         }
14         Q.back() /= Q.back().norm(); // normalization
15     }
16     return (Q); // return at end of local scope

```

If the algorithm does not terminate prematurely, the two nested loops with loop indices i and j trigger execution of the innermost loop body $\frac{1}{2}n(n-1)$ times. This many times an inner product of two vectors of length n has to be computed, which leads to an overall cost of n^3 for $n \rightarrow \infty$.



(1-2.b)  (40 min.) Based on the C++ linear algebra library EIGEN [Lecture → Section 1.2.1], implement a function that performs the same computations as [Lecture → Code 0.3.5.30]:

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

The output vectors should be returned as the columns of a matrix.

HINT 1 for (1-2.b): Use EIGENS block operations see  [EIGEN documentation](#) and [Lecture → Code 1.2.1.6].

SOLUTION of (1-2.b):

There are many ways to code the function `gram_schmidt()`. The following code makes heavy use of EIGEN's block operations.

C++-code 1.2.1: Gram-Schmidt orthonormalization with EIGEN

```

2  Eigen::MatrixX<double> gram_schmidt(const Eigen::MatrixX<double> &A) {
3      // We create a matrix Q with the same size and data of A
4      Eigen::MatrixX<double> Q(A);
5
6      // The first vector just gets normalized
7      Q.col(0).normalize();
8
9      // TODO: (1-2.b) Implement the gram_schmidt procedure by iterating
10     // over all
11     // other columns of A.
12     // START
13     for (unsigned int j = 1; j < A.cols(); ++j) {
14         // See eigen documentation for usage of col and leftCols
15         Q.col(j) -= Q.leftCols(j).transpose() * A.col(j);
16
17         // Normalize vector, if possible
18         // (otherwise it means columns of A are
19         // almost linearly dependent)
20         constexpr double eps = std::numeric_limits<double>::denorm_min();
21         if (Q.col(j).norm() <= eps * A.col(j).norm()) {
22             std::cerr << "Gram-Schmidt failed because "
23             << "A has (almost) linearly dependent "
24             << "columns." << std::endl;
25             break;
26         } else {
27             Q.col(j).normalize();
28         }
29     }
30     // END
31     return Q;
32 }

```

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

(1-2.c)  (25 min.) [depends on Sub-problem (1-2.b)]

Implement a function

```
bool testGramSchmidt(unsigned int n);
```

that tests your implementation by applying the function `gram_schmidt` to the matrix

$$\mathbf{A} \in \mathbb{R}^{n,n}, \quad (\mathbf{A})_{i,j} := i + 2j, \quad 1 \leq i, j \leq n.$$

then checks the orthonormality of the columns of the output matrix, returning **true**, if orthonormality is confirmed.

HINT 1 for (1-2.c): A simple test of how close a matrix is to the zero matrix can rely on computing a suitable norm of the matrix. In EIGEN you may use the `norm` method of the `Eigen::Matrix` class. ┘

HINT 2 for (1-2.c): Remember the warning given in [Lecture → Rem. 1.5.3.15] and design your test accordingly: Never compare a matrix with floating-point entries which is the result of a computation to the zero matrix, when you want to test, whether it is “equal to zero”. ┘

HINT 3 for (1-2.c): If a matrix Q has orthonormal columns, then $Q^T Q = I$ (identity matrix). Such matrices are called **orthogonal**. ┘

SOLUTION of (1-2.c):

We compute the matrix norm $\|Q^T Q - I\|_F$ (Frobenius norm, Euclidean norm of the vectorized matrix, root of the sum of the squares of the matrix entries [Lecture → Def. 3.4.4.17]). This matrix norm is returned by EIGEN's `norm()` method.

This quantity is zero when Q has orthonormal columns (**orthogonal** matrix), and is a measure of how far the matrix Q is from being orthogonal.

C++-code 1.2.2: Test of Gram-Schmidt orthonormalization with EIGEN

```

2  bool testGramSchmidt(unsigned int n) {
3      // Orthonormality test
4      bool orthonormal = false;
5      constexpr double eps = 1e-9;
6      Eigen::MatrixXd A, Q;
7
8      // TODO: (1-2.c) Create A, use gram_schmidt() to compute an
9      // orthonormalization of A, call it Q, and return whether it is
10     // orthonormal.
11     // START
12     Eigen::VectorXd linspace = Eigen::VectorXd::LinSpaced(n, 1., n);
13     A = Eigen::MatrixXd::Zero(n, n);
14     A.rowwise() += 2. * linspace.transpose();
15     A.colwise() += linspace;
16     Q = gram_schmidt(A);
17     const double err =
18         (Q.transpose() * Q - Eigen::MatrixXd::Identity(n, n)).norm();
19     orthonormal = err < eps;
20     // END
21     return orthonormal;
22 }

```

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



End Problem 1-2 , 80 min.