NumCSE

Autumn Semester 2017

Prof. Rima Alaifari

Exercise sheet 1
Computing with Matrices and Vectors

P. Bansal

b:arrowmatvec

Problem 1.1: Arrow matrix×vector multiplication

Innocent looking linear algebra operation can burn considerable CPU power when implemented carelessly. In this problem we study operations involving the so-called arrow matrices, i.e. matrices for which only a few rows/columns and the diagonal are populated.
Refer the matrix class in EIGEN.

Let $n \in \mathbb{N}, n > 0$. An "arrow matrix" $\mathbf{A} \in \mathbb{R}^{n\times n}$ is constructed given two vectors $\mathbf{a} \in \mathbb{R}^n$ and $\mathbf{d} \in \mathbb{R}^n$. The matrix is then squared and multiplied with a vector $\mathbf{x} \in \mathbb{R}^n$. This is implemented in the following C++ function:

tor_arrowmatvec

C++11-code 1.1: Computing $\mathbf{A}^2\mathbf{x}$ for an arrow matrix $\mathbf{A}$

```
void arrow_matrix_2_times_x(const VectorXd &d, const VectorXd &a,
                            const VectorXd &x, VectorXd &y) {
    assert(d.size() == a.size() && a.size() == x.size() &&
           "Vector size must be the same!");
    int n = d.size();

    VectorXd d_head = d.head(n-1);
    VectorXd a_head = a.head(n-1);
    MatrixXd d_diag = d_head.asDiagonal();

    MatrixXd A(n,n);

    A << d_diag,             a_head,
         a_head.transpose(), d(n-1);

    y = A*A*x;
}
```

(a) For general vectors $\mathbf{d} = [d_1, \dots, d_n]^\top$ and $\mathbf{a} = [a_1, \dots, a_n]^\top$ sketch the pattern of the matrix $\mathbf{A}$ created in the function `arrow_matrix_2_times_x` in Code 1.1.

SOLUTION:

From code `arrowmatvec.cpp` we deduce that the argument vectors are copied into the diagonal, the bottom row and rightmost column of the matrix $\mathbf{A}$, which yields

$$\mathbf{A} = \begin{bmatrix} d_1 & & & & a_1 \\ & d_2 & & & a_2 \\ & & \ddots & & \vdots \\ & & & d_{n-1} & a_{n-1} \\ a_1 & a_2 & \dots & a_{n-1} & d_n \end{bmatrix} \tag{1.1}$$

(b) What is the asymptotic complexity of computing $\mathbf{A}^2\mathbf{x}$ in Code 1.1, w.r.t. the matrix size n?

SOLUTION:

The standard matrix-matrix multiplication has complexity $O(n^3)$ and the standard matrix-vector multiplication has complexity $O(n^2)$. Hence, the overall computational complexity is dominated by $O(n^3)$.

In this case, the line affecting the complexity is `y = A*A*x`.

Remember that most of C++ operators have a **left-to-right** precedence. This means that, even if, mathematically, $(\mathbf{AA})\mathbf{y} = \mathbf{A}(\mathbf{Ay})$, when implementing the expression in C++, the *complexity* of the code will be very different. The left hand side has complexity $O(n^3)$, whilst the right hand size has complexity $O(n^2)$.

Remark: Even EIGEN's own internal template expression mechanism is not able to efficiently exploit this fact.

(c) Write an *efficient* C++ function:

```
void efficient_arrow_matrix_2_times_x(const VectorXd &d,
                                      const VectorXd &a,
                                      const VectorXd &x,
                                      VectorXd &y);
```

cpp:ArrowMatrixVector_arrowmatvec

which computes $\mathbf{A}^2\mathbf{x}$ as in Code 1.1, but with optimal asymptotic complexity with respect to n. Here `d` passes the vector $[d_1, \dots, d_n]^\top$ and `a` passes the vector $[a_1, \dots, a_n]^\top$. Why do you expect your implementation to be better? What is the asymptotic complexity of your algorithm?

SOLUTION:

Due to the special structure of the matrix, it is possible to write a function that is more efficient than the standard matrix-vector multiplication.

Define $\mathbf{a}_- := (a_{(1)}, \dots, a_{(n-1)}, 0)$. We can rewrite $\mathbf{A}$ as follows:

$$\mathbf{A} = \begin{bmatrix} d_1 & & & & a_1 \\ & d_2 & & & a_2 \\ & & \ddots & & \vdots \\ & & & d_{n-1} & a_{n-1} \\ a_1 & a_2 & \dots & a_{n-1} & d_n \end{bmatrix} \tag{1.2}$$

$$= \operatorname{diag}(\mathbf{d}) + [\mathbf{0}, \dots, \mathbf{0}, \mathbf{a}_-] + [\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{a}_-]^\top =: \mathbf{D} + \mathbf{V} + \mathbf{H}. \tag{1.3}$$

and $\mathbf{Ax} = \mathbf{Dx} + \mathbf{Vx} + \mathbf{Hx}$. Each of these multiplications can obviously be done in $O(n)$ operations, see Code 1.2.

xVector_arrowmatvec2

C++11-code 1.2:

```
void efficient_arrow_matrix_2_times_x(const VectorXd &d,
                                      const VectorXd &a,
                                      const VectorXd &x,
                                      VectorXd &y) {
    assert(d.size() == a.size() && a.size() == x.size() &&
        "Vector size must be the same!");
    int n = d.size();

    // Notice that we can compute (A*A)*x more efficiently using
    // A*(A*x).  This is, in fact, performing two matrix vector
    // multiplications
    // instead of a more expensive matrix-matrix multiplication.
    // Therefore, as first step, we need a way to efficiently
    // compute A*x

    // This function computes A*x.  you can use it
    // by calling A_times_x(x).
    // This is the syntax for lambda functions:  notice the extra
    // [variables] code.  Each variable written within [] brackets
    // will be captured (i.e.  seen) inside the lambda function.
    // Without &, a copy of the variable will be performed.
    // Notice that A = D + H + V, s.t.  A*x = D*x + H*x + V*x
    // D*x can be rewritten as d*x componentwise
    // H*x is zero, except at the last component
    // V*x is only affected by the last component of x
    auto A_times_x = [&a, &d, n] (const VectorXd & x) {
        // This takes care of the diagonal (D*x)
        // Notice:  we use d.array() to tell Eigen to treat
        // a vector as an array.  As a result:  each operation
        // is performed componentwise.
        VectorXd Ax = ( d.array() * x.array() ).matrix();

        // H*x only affects the last component of A*x
        // This is a dot product between a and x with the last
        // component removed
        Ax(n-1) += a.head(n - 1).dot(x.head(n-1));

        // V*x is equal to the vector
        // (a(0)*x(n-1), ..., a(n-2)*x(n-1), 0)
        Ax.head(n-1) +=  x(n-1) * a.head(n-1);

        return Ax;
    };

    // <=> y = A*(A*x)
    y = A_times_x(A_times_x(x));
}
```

Note the use of C++() **lambda functions** in order to achieve a streamlined implementation. Also note the use of the method **head** to access the top components of a vector.

The efficient implementation only needs two vector-vector element-wise multiplications, a dot-product, and two vector-scalar multiplication. Each of them has complexity $O(n)$. Therefore the complexity is $O(n)$.

(d) Report the runtime of your *efficient* version and the implementation given in Code 1.1 for $n = 2^4, \dots, 2^{10}$. Output the time measurements in seconds, using 3 decimal digits in scientific notation.

Beware: The computations may take a long time for large n ($n > 2048$).

Remark: Run your code using optimization flags (`-O3`) and "-march=native".

SOLUTION:

fig:arrowmatvec-timings-comparison

The standard matrix multiplication has runtime growing with $O(n^3)$. The complexity of the more efficient implementation is $O(n)$. See Code 1.3 and Fig. 1, with data points almost lying on straight lines in doubly logarithmic plot. The slope of these lines indicates the power of n.

Comparison of runtimes of the two implementaitons in `arrowmatvec.cpp` ▷

(The file `arrowmatvec.cpp` also contains an implementation using `chrono`.)

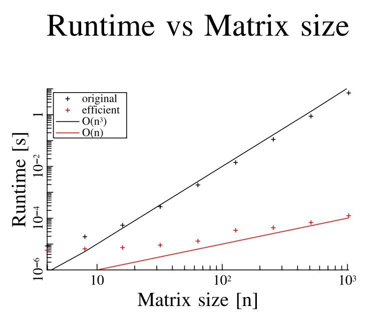

Fig. 1

cpp:arrowmat_timing

C++11-code 1.3: Computation of timings of arrow matrix multiplication

```cpp

    std::cout << std::setw(8) << "n"
              << std::setw(15) << "original"
              << std::setw(15) << "efficient"
              << std::endl;
    for (unsigned n = 4; n <= 512; n *= 2) {
        // Number of repetitions
        unsigned int repeats = 10;

        Timer timer, timer_eff;
        // Repeat test many times
        for (unsigned int r = 0; r < repeats; ++r) {
            // Create test input using random vectors
            Eigen::VectorXd a = Eigen::VectorXd::Random(n),
                            d = Eigen::VectorXd::Random(n),
                            x = Eigen::VectorXd::Random(n),
                            y;

            // Compute times
            timer.start();
            arrow_matrix_2_times_x(d, a, x, y);
            timer.stop();

            // Compute times for efficient implementation
            timer_eff.start();
            efficient_arrow_matrix_2_times_x(d, a, x, y);
            timer_eff.stop();
        }

        // Print results
        std::cout << std::setw(8) << n
                  << std::scientific << std::setprecision(3)
                  << std::setw(15) << timer.min()
                  << std::setw(15) << timer_eff.min()
                  << std::endl;
    }
```

Important comments:

- You can use `std::setw(int)`, `std::precision(int)` and `std::scientific` from the standard library to output formatted text (include `iomanip`). For example-

  ```
  std::cout << std::scientific << std::setprecision(3)
            << std::setw(15) << 1./3e-9;
  ```

- Multiple runs of an implementation should be timed and the minimum time measurement should be reported. For example - run the function `arrow_matrix_2_times_x` 10 times, record time for each run and compute the minimum.
- To measure run times in a C++ code, either use `std::chrono` or `Timer` class. For the latter, include `timer.h` and create a new `Timer` object, as demonstrated in the following code. All times will be outputted in seconds. `Timer` is simply a wrapper to `std::chrono`.

cpp:timer-usage

C++11-code 1.4: Usage of `Timer`

```
Timer t;
t.start();
// HERE CODE TO TIME
t.stop();

// Now you can get the time passed between start and stop using
t.duration();
// You can start() and stop() again, a number of times
// Ideally:  repeat experiment many times and use min() to obtain
    the
// fastest run
t.min();
// You can also obtain the mean():
t.mean();
```

:cancellation

Problem 1.2: Avoiding cancellation

The so-called *cancellation phenomenon* is a major cause of numerical instability. Cancellation is the massive amplification of *relative errors* when subtracting two real numbers in floating point representation of about the same value.
Fortunately, expressions vulnerable to cancellation can often be recast in a mathematically equivalent form which is no longer affected by cancellation.

- Consider the function

$$f_1(x_0, h) := \sin(x_0 + h) - \sin(x_0)\ . \tag{1.4}$$

{\cpl :eq:cancel:1}

(a) Derive an equivalent function $f_2(x_0, h) = f_1(x_0, h)$, which does not involve the difference of return values of trigonometric functions.

(b) Suggest a formula that avoids cancellation errors for computing the approximation

$$f'(x) \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

of the derivative of $f(x) := \sin(x)$ at $x = x_0$.

(c) Write a C++ program that implements your formula and computes an approximation of $f'(1.2)$, for $h = 1 \cdot 10^{-20}, 1 \cdot 10^{-19}, \dots, 1$. Tabulate the relative error of the result using $\cos(1.2)$ as exact value. Plot the error of the approximation of $f'(x)$ at $x = 1.2$.

(d) Explain the observed behaviour of the error.

Hint: Use the trigonometric identity

$$\sin(\varphi) - \sin(\psi) = 2\cos\left(\frac{\varphi + \psi}{2}\right)\sin\left(\frac{\varphi - \psi}{2}\right).$$

SOLUTION:

Check the implementation in Code 1.5 and the plot in Fig. 2. We can observe that the computation using f_1 does not converge as $h \to 0$. This is due to the cancellation error given by the subtraction of two number of approximately same magnitude. The second implementation, using f_2, is very stable and does not display cancellation issues.

fig:error:cancellation

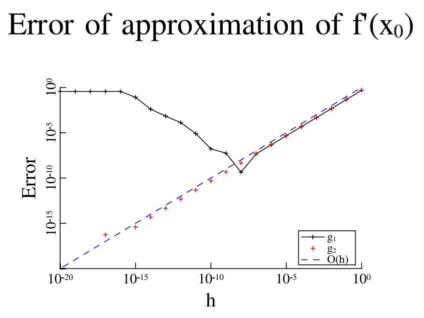

Fig. 2

llation:cpp:solution

C++-code 1.5: C++ implementation for Problem 1.2

```
    // Array of values of h
    ArrayXd h = ArrayXd::LinSpaced(21, -20, 0.)
        .unaryExpr([] (double i) {
            return std::pow(10., i);
        });
    // Dummy array where to evaluate the derivative (1.2)
    ArrayXd x = ArrayXd::Constant(h.size(), 1.2);

    // Derivative
    ArrayXd g1 = (sin(x +h) - sin(x)) / h; // naive
    ArrayXd g2 = 2 * cos(x+0.5*h) * sin(0.5 * h) / h; // better
    ArrayXd ex = cos(x); // exact

    //// Print error

    // Table header
    std::cout << std::setw(15) << "h"
              << std::setw(15) << "exact"
              << std::setw(15) << "cancellation"
              << std::setw(15) << "error"
              << std::setw(15) << "improved"
              << std::setw(15) << "error" << std::endl;
    for(unsigned int i = 0; i < h.size(); ++i) {
        // Table entries
        std::cout << std::setw(15) << h(i)
                  << std::setw(15) << ex(i)
                  << std::setw(15) << g1(i)
                  << std::setw(15) << std::abs(g1(i) - ex(i))
                  << std::setw(15) << g2(i)
                  << std::setw(15) << std::abs(g2(i) - ex(i)) <<
                     std::endl;
    }
```

- Rewrite function $f(x) :- \ln(x-\sqrt{x^2-1})$, $x > 1$, into a mathematically equivalent expression that is more suitable for numerical evaluation for any $x > 1$. Explain why, and provide a numerical example, which highlights the superiority of your new formula.

SOLUTION:

We have

$$\ln(x-\sqrt{x^2-1}) = -\ln(x+\sqrt{x^2-1})\,.$$

We immediately derive $\ln(x-\sqrt{x^2-1}) + \ln(x+\sqrt{x^2-1}) = \log(x^2-(x^2-1)) = 0$.

As $x \to \infty$ the left $\log$ consists of subtraction of two numbers of almost equal value, whilst the right $\log$ consists of the addition of two numbers of approximately the same value (which does not incur in cancellation errors). Therefore, in the l.h.s. there may be cancellation for large values of x, making it worse for numerical computation. You can see the difference of the two expressions with $x = 10^8$.

dMatrixVector

Problem 1.3: Structured matrix–vector product

In Problem 1.1, we saw how the particular structure of a matrix can be exploited to compute a matrix-vector product with substantially reduced computational effort. This problem presents a similar example.

Let $n \in \mathbb{N}$ and $\mathbf{A}$ be a real $n \times n$ matrix defined as:

$$(\mathbf{A})_{i,j} = a_{i,j} = \min\{i,j\},\ i,j = 1,\ldots,n. \tag{1.5}$$

{smv:1}

The matrix-vector product $\mathbf{y} = \mathbf{A}\mathbf{x}$ can be implemented as

pp:multAminSlow

C++11-code 1.6: Computing $\mathbf{A}\mathbf{x}$ for $\mathbf{A}$ from Eq. (1.5)

```
    VectorXd one = VectorXd::Ones(n);
    VectorXd linsp = VectorXd::LinSpaced(n,1,n);
    y = ( ( one * linsp.transpose() )
          .cwiseMin( linsp * one.transpose()) ) * x;
```

(a) What is the asymptotic complexity of the evaluation of the C++ code displayed above w.r.t. the problem size parameter n?

SOLUTION:

Notice the following:

- the outer product of two vectors has complexity $O(n^2)$;
- the Matrix–vector multiplication has quadratic complexity $O(n^2)$;
- component-wise minimum has complexity $O(n^2)$.

Therefore, the overall complexity is $O(n^2)$.

(b) Write an *efficient* C++ function

```
    void multAmin(const VectorXd &x, VectorXd &y);
```

which computes $\mathbf{y} = \mathbf{A}\mathbf{x}$ with a lower asymptotic complexity with respect to n. You can test your implementation by comparing with the output from Code 1.6.

SOLUTION:

For every $j = 1, \cdots, n$ we have

$$y_j = \sum_{k=1}^{j} k x_k + j \sum_{k=j+1}^{n} x_k.$$

Recursively define for $j = 2, \ldots, n$:

$$v_j := v_{j-1} + x_{n-j+1} = \sum_{k=n-j+1}^{n} x_k$$

and

$$w_j := w_{j-1} + jx_j = \sum_{k=1}^{j} kx_k$$

with $v_1 = x_n$ and $w_1 = x_1$. You can easily show that:

$$v_j = \sum_{k=n-j+1}^{n} x_k, w_j = \sum_{k=1}^{j} kx_k$$

Then

$$y_j = \sum_{k=1}^{j} kx_k + j \sum_{k=j+1}^{n} x_k = w_j + jv_{n-j}. \tag{1.6}$$

{eq:recursive-comput-y

We can compute w_j and v_j for $j = 1, \ldots, n$ with a for loop of size n. Once we computed w_j and v_j, we can use (1.6) to compute y_j for each $j = 1 \ldots, n$ using another for loop.

eq:recursive-comput-y

Particular care has to be taken when implementing EIGEN code, since indices start at 0.

cpp:multAminSlow_sol

C++11-code 1.7: Efficiently computing $\mathbf{Ax}$

```
void multAmin(const VectorXd & x, VectorXd & y) {
    unsigned int n = x.size();
    y = VectorXd::Zero(n);
    VectorXd v = VectorXd::Zero(n);
    VectorXd w = VectorXd::Zero(n);

    v(0) = x(n-1);
    w(0) = x(0);

    for(unsigned int j = 1; j < n; ++j) {
        v(j) = v(j-1) + x(n-j-1);
        w(j) = w(j-1) + (j+1)*x(j);
    }
    for(unsigned int j = 0; j < n-1; ++j) {
        y(j) = w(j) + v(n-j-2)*(j+1);
    }
    y(n-1) = w(n-1);
}
```

(c) What is the asymptotic complexity of your function `multAmin`?

SOLUTION:

Only simple loops of size n are performed. Therefore we have linear complexity: $O(n)$.

(d) Measure and compare the runtimes of your `multAmin` and Code 1.6 for $n = 2^4, \ldots, 2^{10}$. Report the minimum runtime in seconds with scientific notation using 3 decimal digits.

SOLUTION:

The matrix multiplication in (1.6) has complexity $O(n^2)$. The faster implementation has complexity $O(n)$. The time measurements are shown in Fig. 3.

cpp:multAminSlow

cpp:multAmin

C++11-code 1.8: Runtime test for `multAmin`

```cpp
    // Timing from 2^4 to 2^13 repeating "nruns" times
    unsigned int nruns = 10;

    std::cout << "--> Timings:" << std::endl;
    // Header, see iomanip documentation
    std::cout << std::setw(15)
              << "N"
              << std::scientific << std::setprecision(3)
              << std::setw(15) << "multAminSlown"
              << std::setw(15) << "multAminLoops"
              << std::setw(15) << "multAmin"
              << std::endl;
    // From 2^4 to 2^13
    for(unsigned int N = (1 << 4); N <= (1 << 10); N = N << 1) {
        Timer tm_slow, tm_slow_loops, tm_fast;
        // Compute runtime many times
        for(unsigned int r = 0; r < nruns; ++r) {
            VectorXd x = VectorXd::Random(N);
            VectorXd y;

            // Runtime of slow method
            tm_slow.start();
            multAminSlow(x, y);
            tm_slow.stop();

            // Runtime of slow method with loops
            tm_slow_loops.start();
            multAminLoops(x, y);
            tm_slow_loops.stop();

            // Runtime of fast method
            tm_fast.start();
            multAmin(x, y);
            tm_fast.stop();
        }

        std::cout << std::setw(15)
                  << N
                  << std::scientific << std::setprecision(3)
                  << std::setw(15) << tm_slow.min()
                  << std::setw(15) << tm_slow_loops.min()
```

```
                          << std::setw(15) << tm_fast.min()
                          << std::endl;
     }
```

gs-comparison

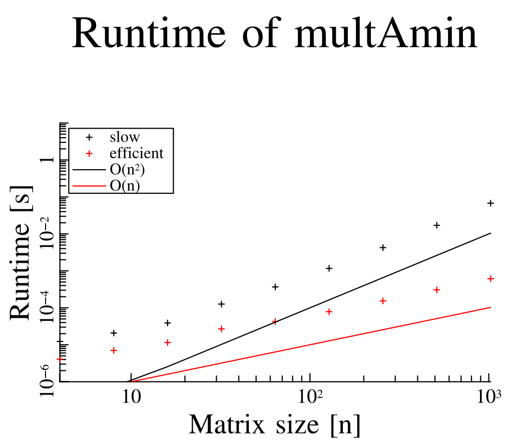

Fig. 3

(e) Sketch the matrix $\mathbf{B}$ created by the following C++ snippet:

cpp:multAminSlow_6

C++11-code 1.9: Initializing $\mathbf{B}$

```
     MatrixXd B = MatrixXd::Zero(n,n);
     for(unsigned int i = 0; i < n; ++i) {
         B(i,i) = 2;
         if(i < n-1) B(i+1,i) = -1;
         if(i > 0) B(i-1,i) = -1;
     }
     B(n-1,n-1) = 1;
```

SOLUTION:

The matrix $\mathbf{B}$ is the following:

$$\mathbf{B} := \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 1 \end{bmatrix}$$

Notice the value 1 in the entry (n, n).

(f) Write a short EIGEN-based C++ program, which computes $\mathbf{ABe}_j$ using $\mathbf{A}$ from Eq. (1.5) and $\mathbf{B}$ from Code 1.9. Here $\mathbf{e}_j$ is the j-th unit vector in $\mathbb{R}^n$, $j = 1, \dots, n$ and $n = 10$. Based on the result of the computation $\mathbf{ABe}_j$, can you predict the relation between $\mathbf{A}$ and $\mathbf{B}$?

SOLUTION:

You should have observed that you recover the same unit vector that you have multiplied with $\mathbf{AB}$. Hence, the matrix $\mathbf{B}$ is the (unique) inverse of $\mathbf{A}$.

mschmidteigen

Problem 1.4: Gram-Schmidt orthonormalization with EIGEN

In this problem, we look at the famous **Gram-Schmidt orthonormalization** algorithm.

(a) Implement an EIGEN-based C++ function for Gram-Schmidt orthonormalization, which returns the output vectors as the columns of a matrix:

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

SOLUTION:

eigen:cpp:solution1

C++-code 1.10: Gram-Schmidt orthonormalization with EIGEN

```cpp
MatrixXd gram_schmidt(const MatrixXd & A) {
    // We create a matrix Q with the same size and data of A
    MatrixXd Q(A);

    // The first vector just gets normalized
    Q.col(0).normalize();

    // Iterate over all other columns of A
    for(unsigned int j = 1; j < A.cols(); ++j) {
        // See eigen documentation for usage of col and leftCols
        Q.col(j) -= Q.leftCols(j) * (Q.leftCols(j).transpose() *
            A.col(j));

        // Normalize vector, if possible
        // (otherwise it means columns of A are
        // almost linear dependant)
        double eps = std::numeric_limits<double>::denorm_min();
        if( Q.col(j).norm() <= eps * A.col(j).norm() ) {
            std::cerr << "Gram-Schmidt failed because "
                      << "A has (almost) linear dependant "
                      << "columns. Bye." << std::endl;
            break;
        } else {
            Q.col(j).normalize();
        }
    }

    return Q;
}
```

(b) Test your implementation by applying the function `gram_schmidt` to a small random matrix and checking the orthonormality of the columns of the output matrix.

Hint: A simple test of how close a matrix is to the zero matrix can rely on computing a suitable norm of the matrix. You may use the **norm** method of the **Matrix** class provided by EIGEN.

SOLUTION:

We compute the matrix norm $\|\mathbf{Q}^\top\mathbf{Q}-\mathbf{I}\|_F$ (Frobenius norm- root of the sum of the squares of the matrix entries). This matrix norm is returned by EIGEN's **norm**() method.

$\|\mathbf{Q}^\top\mathbf{Q}-\mathbf{I}\|_F$ is zero when $\mathbf{Q}$ has orthonormal columns (**orthogonal** matrix), and is a measure of how far the matrix $\mathbf{Q}$ is from being orthogonal.

eigen:cpp:solution2

C++-code 1.11: Test of Gram-Schmidt orthonormalization with EIGEN

```
int main(void) {
    // Orthonormality test
    unsigned int n = 9;
    MatrixXd A = MatrixXd::Random(n,n);
    MatrixXd Q = gram_schmidt( A );

    // Compute how far is Q^T * Q from the identity
    // i.e.  "How far is Q from being orthonormal?"
    double err = (Q.transpose()*Q - MatrixXd::Identity(n,n))
        .norm();

    // Error has to be small, but not zero (why?)
    std::cout << "Error is: "
              << err
              << std::endl;

    // If error is too big, we exit with error
    double eps = std::numeric_limits<double>::denorm_min();
    exit(err < eps);
}
```