NumCSE

Autumn Semester 2017

Prof. Rima Alaifari

Exercise sheet 1
Computing with Matrices and Vectors

P. Bansal

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:

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.

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

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

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

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?

(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".

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`.

C++11-code 1.2: 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();
```

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

$$f_1(x_0, h) := \sin(x_0 + h) - \sin(x_0)\ . \tag{1.1}$$

  (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 $h = 1 \cdot 10^{-20}, 1 \cdot 10^{-19}, \ldots, 1$. 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

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

- 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.

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:

$$(\mathbf{A})_{i,j} = a_{i,j} = \min\{i,j\}, \; i,j = 1,\ldots,n. \tag{1.2}$$

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

C++11-code 1.3: Computing $\mathbf{A}\mathbf{x}$ for $\mathbf{A}$ from Eq. (1.2)

```
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?

(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 1.3.

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

(d) Measure and compare the runtimes of your `multAmin` and Code 1.3 for $n = 2^4, \ldots, 2^{10}$. Report the minimum runtime in seconds with scientific notation using 3 decimal digits.

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

C++11-code 1.4: 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;
```

(f) Write a short EIGEN-based C++ program, which computes $\mathbf{A}\mathbf{B}\mathbf{e}_j$ using $\mathbf{A}$ from Eq. (1.2) and $\mathbf{B}$ from Code 1.4. Here $\mathbf{e}_j$ is the j-th unit vector in $\mathbb{R}^n$, $j = 1,\ldots,n$ and $n = 10$. Based on the result of the computation $\mathbf{A}\mathbf{B}\mathbf{e}_j$, can you predict the relation between $\mathbf{A}$ and $\mathbf{B}$?

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);
```

(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.