NumCSE exercise sheet 1
Cancellation errors, manipulating matrices

alexander.dabrowski@sam.math.ethz.ch
oliver.rietmann@sam.math.ethz.ch

October 9, 2018

Exercise 1.1. *Summing floating point numbers.*

Suppose you have a `vector<float> v` which contains some unknown numbers with very different magnitudes, and you want to compute its total sum.

(a) Suppose in this subpoint that `v` has size 10^7, and that it has a single element equal to `1` and the rest are all in the range `[1e-8, 2e-8]`. Consider the following simple code to sum all the numbers in `v`:

```
float sum = 0;
for (float x : v) sum += x;
```

What is the final value of `sum` if `1` is in the first position of `v`? What can you say if `1` is in the last position of `v`?

Solution: In the first case it is `1`, because at each iteration we add something smaller than `2e-8` to `1`, which with `float` precision remains `1`. In the second case we add small numbers first, so no catastrophic cancellation occurs, so we would expect the final sum to lie approximately in the range `[1.1, 1.2]`.

(b) Implement a function `sum` which returns the sum of `v` with good accuracy (if `v` is as in subpoint (a), the sum should have an error ~~smaller than `1e-3`~~ (**Correction**: approximately `1e-2`) for any possible position of `1`).

Hint: add up first the numbers which have small magnitude.

Solution:

```
13 float sum(vector<float> v) {
14     float sum = 0, prev_sum = 0;
15     sort(v.begin(), v.end(), [] (auto x, auto y) { return abs(x) < abs(y); } );
16     for(int i = 0; i < v.size(); i++) {
17         prev_sum = sum;
18         sum += v[i];
19     }
20     return sum;
```

kahan_sum.cpp

Notice that adding up from smallest to largest doesn't guarantee best accuracy. For instance, just by replacing line 15 with

```
swap(*max_element(v.begin(), v.end()), *(v.end()-1));
```

which puts the `1` in last position, gives an error of approximately `1e-3` for our test vector.

(c) What if instead `v` has size 10^8, with all elements in the range `[1e-8, 2e-8]` except for a single element `1`?

Solution: Simply sorting and adding up in order the numbers will give $\simeq$`1.5`, which is incorrect. This is due to the fact that `0.5+2e-8` is still `0.5` for a `float`. A possible solution to this issue would be to take the two smallest numbers of `v`, delete them from `v`, re-insert their sum, and so on. This strategy would require modifying (or copying) `v` and additional time for finding the minimum (or for keeping `v` sorted). A better solution is given in the next subpoint.

(d) Implement a function `acc_sum` which returns the sum of `v` by adapting the simple code of Point (a) to keep track in an accumulator variable of cancellation errors which arise in each iteration of the loop.

Hint: There are many possible solutions to this problem. The following hint might be helpful to derive one of the "best" strategies. At each loop: add `x` to the accumulator, then try to add the accumulator to the running sum, then add to the accumulator the cancellation error.

Solution:

This strategy is also called Kahan algorithm[1].

```
22
23 float acc_sum(vector<float> &v) {
24     float sum = 0, e = 0;
25     for (float x : v) {
26         e += x;
27         float tmp = sum + e;
28         e += sum - tmp;
29         sum = tmp;
30     }
31     return sum;
```

kahan_sum.cpp

[1]William Kahan, "Further remarks on reducing truncation errors", Communications of the ACM, vol. 8, 1965.

Exercise 1.2. *Structured matrix–vector product.*

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

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

The matrix-vector product $y = Ax$ can be implemented as

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

Code 1: structured_matrix_vector.cpp

(a) What is the asymptotic complexity of Code 1 w.r.t. the problem size 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 a C++ function

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

which computes $\mathbf{y} = \mathbf{Ax}$ with complexity $O(n)$. You can test your implementation by comparing with the output from Code 1.

Hint: Make use of partial sums $\sum_{i=j}^{n} x_i$.

Solution: We set $v_n = x_n$ and define for all $i = 1, \ldots, n-1$

$$v_i = v_{i+1} + x_i.$$

Then our solution vector is given by $y_1 = v_1$ and

$$y_i = y_{i-1} + v_i \tag{2}$$

for all $i = 2, \ldots, n$.

```
55 void multAmin(const VectorXd & x, VectorXd & y) {
56     unsigned int n = x.size();
57
58     if (n == 0) return;
59
60     y = VectorXd::Zero(n);
61     VectorXd v = VectorXd::Zero(n);
62
63     v(n-1) = x(n-1);
64     for (unsigned int i = n-1; 0 < i; --i)
65         v(i-1) = v(i) + x(i-1);
66
67     y(0) = v(0);
68     for (unsigned int i = 1; i < n; ++i)
69         y(i) = y(i-1) + v(i);
70 }
```

structured_matrix_vector.cpp

(c) Measure and compare the runtimes of your `multAmin` and Code 1 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 Code 1 has complexity $O(n^2)$. The faster implementation has complexity $O(n)$. The time measurements are shown in Figure 1.

```
94      unsigned int nruns = 10;
95
96      std::cout << "--> Timings:" << std::endl;
97      // Header, see iomanip documentation
98      std::cout << std::setw(15)
99                << "N"
100               << std::scientific << std::setprecision(3)
101               << std::setw(15) << "multAminSlown"
102               << std::setw(15) << "multAminLoops"
103               << std::setw(15) << "multAmin"
104               << std::endl;
105     // From $2^4$ to $2^{13}$
106     for(unsigned int i = 0; i<nLevels; i++) {
107         // Compute runtime many times
108         double min_slow = std::numeric_limits<double>::infinity();
109         double min_slow_loops = std::numeric_limits<double>::infinity();
110         double min_fast = std::numeric_limits<double>::infinity();
111
112         for(unsigned int r = 0; r < nruns; ++r) {
113             VectorXd x = VectorXd::Random(n[i]);
114             VectorXd y;
115
116             std::chrono::time_point<std::chrono::high_resolution_clock> start, end;
117             std::chrono::duration<double, std::ratio<1>> duration;
118
119             // Runtime of slow method
120             start = std::chrono::high_resolution_clock::now();
121             multAminSlow(x, y);
122             end = std::chrono::high_resolution_clock::now();
123             duration = end - start;
124             min_slow = std::min(min_slow, (double)duration.count());
125
126             // Runtime of slow method with loops
127             start = std::chrono::high_resolution_clock::now();
128             multAminLoops(x, y);
129             end = std::chrono::high_resolution_clock::now();
130             duration = end - start;
131             min_slow_loops = std::min(min_slow_loops, (double)duration.count());
132
133             // Runtime of fast method
134             start = std::chrono::high_resolution_clock::now();
135             multAmin(x, y);
136             end = std::chrono::high_resolution_clock::now();
137             duration = end - start;
138             min_fast = std::min(min_fast, (double)duration.count());
139         }
140
141         minTime[i] = min_slow;
142         minTimeLoops[i] = min_slow_loops;
143         minTimeEff[i] = min_fast;
144
```

```
145         std::cout << std::setw(15)
146                   << n[i]
147                   << std::scientific << std::setprecision(3)
148                   << std::setw(15) << minTime[i]
149                   << std::setw(15) << minTimeLoops[i]
150                   << std::setw(15) << minTimeEff[i]
151                   << std::endl;
152     }
```

structured_matrix_vector.cpp

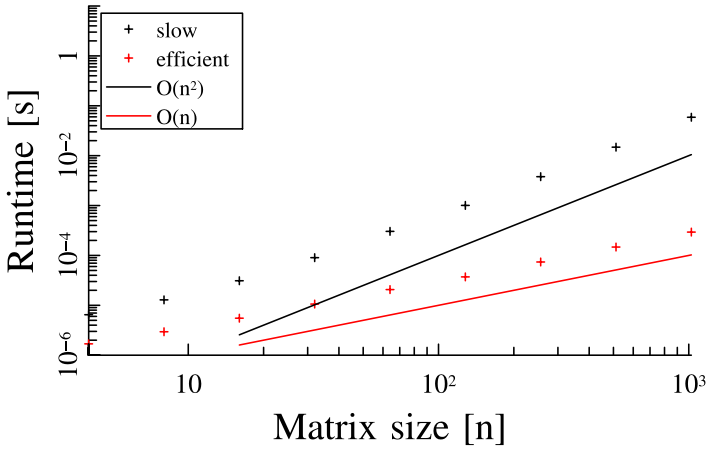

Figure 1: Runtime comparison.

(d) Sketch the matrix B created by the following C++ snippet:

```
190     MatrixXd B = MatrixXd::Zero(nn,nn);
191     for(unsigned int i = 0; i < nn; ++i) {
192         B(i,i) = 2;
193         if(i < nn-1) B(i+1,i) = -1;
194         if(i > 0) B(i-1,i) = -1;
195     }
196     B(nn-1,nn-1) = 1;
```

Code 2: structured_matrix_vector.cpp

Solution: The matrix B is the following:

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

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

(e) Write a short Eigen-based C++ program, which computes ABe_j using A from Equation (1) and B from Code 2. Here 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 ABe_j, can you predict the relation between A and B?

Solution: You should have observed that you recover the same unit vector that you have multiplied with AB. Hence, the matrix B is the (unique) inverse of A.

Exercise 1.3. *Strassen algorithm*

Note: In this exercise focus on understanding the algorithms presented and calculating their complexity. The coding parts are optional, and their aim is only to allow you to check your understanding of the definitions and practice `C++`/`Eigen`. We recommend to not spend your time optimizing code in this exercise (for instance, don't worry about excessive copies of matrices in recursive calls).

Let $N = 2^n$ and let A, B be two matrices of size $N \times N$. Let $C = AB$. By definition of matrix product

$$c_{i,j} = \sum_k a_{i,k} b_{k,j}. \tag{3}$$

Equivalently, partitioning the matrices in equally sized sub-blocks

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right), B = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right), C = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right),$$

we have

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} \quad \text{for } i, j \in \{1, 2\}. \tag{4}$$

Strassen discovered[2] that defining the matrices

$$\begin{cases} M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 = (A_{21} + A_{22})B_{11} \\ M_3 = A_{11}(B_{12} - B_{22}) \\ M_4 = A_{22}(B_{21} - B_{11}) \\ M_5 = (A_{11} + A_{12})B_{22} \\ M_6 = (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 = (A_{12} - A_{22})(B_{21} + B_{22}) \end{cases}$$

one can compute C as

$$\begin{cases} C_{11} = M_1 + M_4 - M_5 + M_7 \\ C_{12} = M_3 + M_5 \\ C_{21} = M_2 + M_4 \\ C_{22} = M_1 - M_2 + M_3 + M_6 \end{cases} \tag{5}$$

(a) Implement a function `mult` which computes the matrix product AB by applying directly the definition of (3).

Solution:

```
7  MatrixXf mult(const MatrixXf &A, const MatrixXf &B) {
8      int N = A.rows();
9      MatrixXf C = MatrixXf::Zero(N, N);
10     for (int i = 0; i < N; i++) {
11         for (int j = 0; j < N; j++) {
12             for (int k = 0; k < N; k++) {
13                 C(i, j) += A(i, k) * B(k, j);
14             }
15         }
16     }
```

[2]Volker Strassen, "Gaussian elimination is not optimal", Numerische Mathematik, vol. 13, 1969.

```
    return C;
}
```

strassen.cpp

(b) What is the asymptotic complexity of `mult`?

Solution: Since there are three nested loops of length N, the algorithm is $\Theta(N^3)$.

(c) Implement a function `mult_rec` which computes the matrix product recursively using (4) (the base case is when the blocks have size 1×1).

Solution:

```
// unorthodox but handy utility macros
#define b11 .block(0, 0, N/2, N/2)
#define b12 .block(0, N/2, N/2, N/2)
#define b21 .block(N/2, 0, N/2, N/2)
#define b22 .block(N/2, N/2, N/2, N/2)

MatrixXf mult_rec(const MatrixXf &A, const MatrixXf &B) {
    int N = A.rows();
    if (N <= 1) return A*B;

    MatrixXf C(N, N);
    C << mult_rec(A b11, B b11) + mult_rec(A b12, B b21),
         mult_rec(A b11, B b12) + mult_rec(A b12, B b22),
         mult_rec(A b21, B b11) + mult_rec(A b22, B b21),
         mult_rec(A b21, B b12) + mult_rec(A b22, B b22);

    return C;
}
```

strassen.cpp

(d) What is the asymptotic complexity of `mult_rec`?

Hint: Indicate $f(n)$ as the number of elementary operations required to multiply two $2^n \times 2^n$ matrices, find a recurrence relation for f, and solve it.

Solution: To compute (4) for a single pair of indices (i, j) we need to compute twice the product of two $2^{n-1} \times 2^{n-1}$ matrices, which takes $2f(n-1)$ operations, and add the results together, which takes $2^{2(n-1)}$ operations. Since we have to do it for 4 pair of indices ($(i, j) = (1, 1), (1, 2), (2, 1), (2, 2)$), we have that

$$f(n) = 8f(n-1) + 2^{2n}.$$

Since $f(0) = 1$, solving this recurrence relation[3] we have $f(n) = 2(2^{3n}) - 2^{2n}$, and thus $f(n) = \Theta(2^{3n}) = \Theta(N^3)$ as for the iterative algorithm.

(e) Implement a function `strassen` which computes the matrix product recursively using (5).

Solution:

[3]For the purpose of the exercise it would be enough to rewrite informally $f(n) = 8f(n-1)+\cdots = 8(8f(n-2))+\cdots = \cdots = C8^n + \ldots$. The formula in the solution can be obtained formally from the theory of difference equations (for an overview see `/en.wikipedia.org/wiki/Recurrence_relation`).

```
40 MatrixXf strassen(const MatrixXf &A, const MatrixXf &B) {
41     int N = A.rows();
42     if (N <= 1) return A*B;
43
44     MatrixXf M1 = strassen(A b11 + A b22, B b11 + B b22);
45     MatrixXf M2 = strassen(A b21 + A b22, B b11);
46     MatrixXf M3 = strassen(A b11, B b12 - B b22);
47     MatrixXf M4 = strassen(A b22, B b21 - B b11);
48     MatrixXf M5 = strassen(A b11 + A b12, B b22);
49     MatrixXf M6 = strassen(A b21 - A b11, B b11 + B b12);
50     MatrixXf M7 = strassen(A b12 - A b22, B b21 + B b22);
51     MatrixXf C(N, N);
52     C << M1 + M4 - M5 + M7, M3 + M5,
53          M2 + M4, M1 - M2 + M3 + M6;
54     return C;
55 }
```

strassen.cpp

(f) What is the asymptotic complexity of `strassen`?

Hint: adapt the approach of Point (d).

Solution: At each step of (5) we need to compute 7 multiplications and 18 additions between $2^{n-1} \times 2^{n-1}$ matrices, thus

$$f(n) = 7f(n-1) + 18(2^{2(n-1)}).$$

Since $f(0) = 1$, solving this recurrence relation we have $f(n) = 7(7^n) - 6(2^{2n})$, thus

$$f(n) = \Theta(7^n) = \Theta(N^{\log_2 7}).$$

Therefore Strassen algorithm is asymptotically faster than naive matrix multiplication.

Figure 2: Log-plot of number of elementary operations as a function of n