Introduction to Supercomputing

TMA4280 · Problem set 1

Exercise 1. Exercise 1.1 in the lecture notes.

Solution. We found that single precision numbers have about 7 significant (decimal) digits, since they use a 23-bit mantissa, and $2^{-23} \approx 10^{-7}$.

Exercise 2. Exercise 1.2 in the lecture notes.

Solution.

$$\begin{aligned} 4.25 &= 4 + \frac{1}{4} \\ &= 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} \\ &= 100.01_2 \\ &= 1.0001_2 \cdot 2^2 \end{aligned}$$

Exercise 3. Exercise 1.3 in the lecture notes.

Solution. This was computed in the lecture. A double precision floating point number has a 52 binary digit mantiassa, so its accuracy is $2^{-52} \approx 10^{-16}$. Therefore, about 16 decimal digits.

Exercise 4. Exercise 1.4 in the lecture notes.

Solution. Some options:

- Use nested loops, where none of the loops go higher than about 10^9 iterations.
- Use a larger datatype. E.g. in C, `long` (at least 32 bits) or `long long` (at least 64). Note that a C `int` may be as low as 16 bits.
- Use an unsigned datatype. This gives us twice the range since the sign bit isn't needed.

Exercise 5. Exercise 1.5 in the lecture notes.

Solution. The first case needs n additions and multiplications, so $2n = \mathcal{O}(n)$ operations total.

For the second case, each element of y requires n multiplications and $n-1$ additions to form. Therefore the total number of operations should be $n(n+n-1) = n(2n-1) = \mathcal{O}(n^2)$.

Exercise 6. Exercise 1.6 in the lecture notes.

Solution. The matrix requires n^2 numbers to store, and each of the vectors requires n. We assume double precision floating point numbers (8 B per number). Let us also assume that n is large, so that the total memory requirement will be approximately n^2 numbers.

$$8n^2 = 10^9 \quad \Longrightarrow \quad n \approx 11180.$$

Therefore, only about 10^4 unknowns can fit in this memory.

Exercise 7. In the lecture we found that adding a small number to a large number can cause problems when the relative difference between the numbers exceed the accuracy of the floating point representation.

With this in mind, suggest an algorithm for summing a list of numbers that is more accurate than doing it "naively".

Solution. The simplest solution is to sort the numbers before summing them in order from smallest to largest. However, this approach has complexity $O(n \log n)$, which is worse than naive summation.

The *Kahan* algorithm keeps track of the lost bits and tries to add them on the following iterations. For each element s_i in the input array, the sum is updated as

$$S_i = S_{i-1} + \underbrace{s_i + c_{i-1}}_{y_i},$$

where S_i is the sum of the first i elements, and c_{i-1} is the compensation from the first $i-1$ elements. However, since the sum S_{i-1} is large, and y_i is small, the lower bits may be lost. These lower bits are computed and used as compensation for the next iteration:

$$c_i = y_i - \underbrace{(S_i - S_{i-1})}_{\text{high bits of } y_i}.$$

Subtracting the high bits of y_i from y_i should leave just the low bits, which were exactly what went missing.

Note that algebraically (in infinite precision arithmetic), c_i should always be zero.

Exercise 8. Implement a C or Fortran program that calculates $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$.

$$\boldsymbol{A} = \begin{pmatrix} 0.3 & 0.4 & 0.3 \\ 0.7 & 0.1 & 0.2 \\ 0.5 & 0.5 & 0.0 \end{pmatrix}, \qquad \boldsymbol{x} = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \end{pmatrix}.$$

Solution. In C:

```c
#include <stdio.h>

const double A[][3] = {{0.3, 0.4, 0.3},
                       {0.7, 0.1, 0.2},
                       {0.5, 0.5, 0.0}};
const double x[3] = {1.0, 1.0, 1.0};

int main(int argc, char **argv)
{
  double y[3];
  for (int i = 0; i < 3; i++) {
    y[i] = 0.0;
    for (int j = 0; j < 3; j++)
      y[i] += A[i][j] * x[j];
  }
  printf("y = %f %f %f\n", y[0], y[1], y[2]);
  return 0;
}
```