Solutions to Problem Set 1

Fundamentals of Simulation Methods 8 ECTS
Heidelberg University WiSe 20/21

Elias Olofsson
ub253@stud.uni-heidelberg.de

November 11, 2020

1 Packing of numbers (2pt)

Estimate how many numbers there are in the interval between 1.0 and 2.0, and in between the interval of 255.0 to 256.0, for IEEE-754 single and double precision.

When a number $f \in [1.0, 2.0)$ is to be represented in IEEE-754, the exponent e of the representation has to be equal to 0, due to the convention of normalization. Since the exponent is the same throughout this interval, the spacing between each representable number is constant with a distance of

$$\Delta f = \frac{1}{2^p} \cdot 2^0, \tag{1}$$

and the upper and lower representations of this interval is

$$f_{lower} = (1 + 0) \cdot 2^0 = 1, \tag{2}$$

$$f_{upper} = \left(1 + \frac{2^p - 1}{2^p}\right) \cdot 2^0 < 2 \tag{3}$$

Thus, the amount of representable numbers within this range is determined by the precision p of the representation, and how many permutations the mantissa has. For single and double precision, where p is 23 and 52 respectively, the quantity of numbers in the interval (1.0, 2.0) are thus

$$2^p - 1 = 2^{23} - 1 = 8388607, \quad \text{(single precision)} \tag{4}$$

$$2^p - 1 = 2^{52} - 1 = 4.503599627 \cdot 10^{15}. \quad \text{(double precision)} \tag{5}$$

Note that since the question specified the numbers *between* 1.0 and 2.0, the single point 1.0 has been excluded from the interval.

To estimate the amount of representable numbers in the interval between 255.0 and 256.0, we first can note that the exponent e has to be equal to 7, due to the convention of normalization. In the interval $f \in [2^7, 2^8) = [128, 256)$ we have constant spacing between the representable numbers, and

also the same amount of numbers as in the interval $[1.0, 2.0)$. Since the interval [255.0, 256.0) is a 1/128 part of the interval $[2^7, 2^8)$, we can estimate the amount of representable numbers between 255.0 and 256.0 to

$$\frac{2^p}{128} - 1 = \frac{2^{23}}{128} - 1 = 65535, \qquad \text{(single precision)} \tag{6}$$

$$\frac{2^p}{128} - 1 = \frac{2^{52}}{128} - 1 = 3.518437209 \cdot 10^{13}, \qquad \text{(double precision)} \tag{7}$$

for single and double precision respectively. Similar to before, we deduct a single value in order to not count the the point exactly at 255.0, since we are interested in the amount of points *between* 255.0 and 256.0.

2 Pitfalls of integer & floating point arithmetic (6pt)

2.1 Part 1

Consider the given C/C++ code, which prints out two float numbers. Explain why the two numbers are not equal.

In this C/C++ example, the code evaluates the variables to `y = 6.0` and `z = 7.0` respectively. The reason why the two numbers are not equal can be explained by looking at what point an implicit type-conversion occurs in the two calculations. In the first case

```
float y = 2*(i/2);
```
(8)

the integer 7 is divided by the integer 2, and thus all decimals are truncated and lost due to the integer division. On the other hand, for the second calculation

```
float z = 2*(i/2.);
```
(9)

the point behind the 2 in the denominator implies that the integer 7 is divided by a floating point number. Thus an implicit type conversion is performed here and the resulting decimals from the division are kept intact.

2.2 Part 2

Consider the following numbers in the given code. Calculate the results for x and y. Which one is correct, if any? Explain why the law of associativity is broken here.

The code in the example evaluates to `x = 1.0` and `y = 0.0`, with the first alternative giving the correct answer. The reason for the discrepancy in the evaluations is due to the large relative magnitude of the numbers and in which order they are added. Since the numbers `a` and `b` are of the same

magnitude, they correctly cancel out when added together as in the first calculation. But when `c` which is 16 orders of magnitude smaller gets added to `b`, a rounding inevitably has to occur. Since 1.0 is smaller than the distance to the closest representable number neighbouring `a` or `b`, the addition of `c` to either `a` or `b` is effectively ignored. Thus the law of associativity is broken and we have

$$\texttt{a + c = a} \qquad \text{for } |\texttt{a}| \gg |\texttt{c}|, \tag{10}$$

$$\texttt{b + c = b} \qquad \text{for } |\texttt{b}| \gg |\texttt{c}|. \tag{11}$$

Per my quick calculations, `c` would have to be larger than or equal to 16.0 in order to not get rounded away when added to `a` or `b` in this case, when using normal double precision.

2.3 Part 3

Consider the following C/C++ code and explain what you see.

In this code, we overreach the representable range of numbers for the IEEE-754 single precision standard for storing floating point numbers. Since single precision has the upper limit of $f_{max} \simeq 3.4 \cdot 10^{38}$, any numbers larger in magnitude than this value will be handled as $\pm$`inf`. Thus when the square of 10^{20} is calculated, the limit of the data type `float` is exceeded and the code incorrectly evaluates

$$\texttt{y/x = x*x/x = inf} \qquad \text{when} \quad \texttt{x = 1e20}. \tag{12}$$

3 Machine epsilon (5pt)

Write a computer program in C, C++ or Python that experimentally determines the machine epsilon ϵ_m , i.e. the smallest number ϵ_m such that $1+\epsilon_m$ still evaluates to something different from 1, for the data types float, double and long double. Evaluate and print out $1+\epsilon_m$. Do you see something strange?

See the code `fsm_ex1_task3.c` submitted with this pdf. Here, we start with an ansatz of 1.0 for a machine epsilon and divide this ansatz by a factor of 2 until the condition

$$\epsilon_m + 1 = 1, \tag{13}$$

is satisfied. The last value that ϵ_m had before this condition was fulfilled is then the machine epsilon for the given floating point data type. We use 1.0 as the initial ansatz for a specific reason, namely because the mantissa is 0 for this number, giving the smallest possible value within the current range of representable numbers, determined by the exponent e.

Having the mantissa be zero ensures that when the condition in eq.(13) is eventually satisfied, the last number ϵ_m in the previous iteration was the smallest possible number where $\epsilon_m + 1 \neq 1$, i.e. where the digit in the last position of the mantissa for the floating point number 1 flipped due to the addition of ϵ_m. This is the definition of machine epsilon, and experimentally I have then determined the machine epsilons for the given data types to

$$\epsilon_m \simeq 1.19209 \cdot 10^{-7}, \quad \text{(float)} \tag{14}$$
$$\epsilon_m \simeq 2.22045 \cdot 10^{-16}, \quad \text{(double)} \tag{15}$$
$$\epsilon_m \simeq 1.08420 \cdot 10^{-19}, \quad \text{(long double)} \tag{16}$$

for the standard C/C++ data types `float`, `double` and `long double`. As a remark, since this code was compiled with the GNU Compiler Collection on a Linux machine, the last result for the data type `long double` I believe have an 80-bit extended precision in my specific case [1].

5/5

4 Near-cancellation of numbers (7pt)

Consider the function

$$f(x) = \frac{x + \exp(-x) - 1}{x^2}, \tag{17}$$

and note that $f(x)|_{x=0}$ is ill-defined. However, for the limit $x \to 0$ there exists an non-zero and non-infinite value of $f(x)$.

1. *Determine $\lim_{x\to 0} f(x)$.*
2. *Write a computer program that asks for a value of x from the user and then prints $f(x)$.*
3. *For small (but positive non-zero) values of x this evaluation goes wrong. Determine experimentally at which values of x the evaluation breaks down.*
4. *Explain why this happens.*
5. *Add an if-clause to the program such that for small values the function is evaluated in another way that does not break down, so that for all positive values of x the program produces a reasonable result.*

First, we determine the limit of function f as x goes to zero. Using

L'Hôpital's rule, we evaluate the limit to

$$\begin{aligned}\lim_{x\to 0} f(x) &= \lim_{x\to 0} \frac{x + \exp(-x) - 1}{x^2} \\ &= \lim_{x\to 0} \frac{1 - \exp(-x)}{2x} \\ &= \lim_{x\to 0} \frac{\exp(-x)}{2} \\ &= \frac{1}{2}.\end{aligned} \tag{18}$$

Following this, a computer program which asks the user for a value x and prints the evaluation of $f(x)$ can be seen in the attached code `fsm_ex1_task4.c`.

Using the standard floating point data type `double` for values of x and slowly approaching $x = 0$ from the positive side, we can see that the evaluation $f(x)$ starts behaving badly already at $x \simeq 10^{-6}$. Since the derivative of $f(x)$ is negative for all x, and the limit $\lim_{x\to 0} f(x) = 1/2$, we should have constantly increasing numbers, all strictly smaller than 0.5, when approaching $x = 0$ from the positive side. However, evaluating the function with $x = 10^{-6}$, we get the impossible result $f(x)|_{x=10^{-6}} \simeq 0.50004$.

This peculiar result can be explained by the near cancellation or numbers in the numerator of the function f when x is close to zero. Since $\exp(-x) \simeq 1$ when $x \simeq 0$, we here lose precision in the evaluation of `exp(-x)-1`, and decimals that should have been included in the real result, instead gets truncated in the numerical evaluation. Since the result of `exp(-x)-1` is negative and also truncated, i.e. smaller in magnitude, the evaluation of the entire numerator `x+exp(-x)-1` results in a larger value than the real one. Thus, the code yields a numerical result that should be mathematically impossible.

If we decrease the value of x even more, we get even more dramatic numerical errors. When $x \simeq 10^{-8}$ the evaluation of `x+exp(-x)` gets rounded to exactly 1. When this happens, the numerator becomes zero and the entire expression evaluates to zero. Considering the analytical limit for $x = 0$ which we previously determined, this is obviously wildly incorrect.

If we for fun take this decrease of x to its extreme limit, when $x \simeq 10^{-165}$ we overreach the representable range of `double` when the square of x is evaluated in the denominator. Thus, we we will get division by zero and the entire expression will evaluate to `NaN`.

In order to correct the code and account for the ill-defined range of $f(x)$ for values x close to zero, we can modify the function expression with an if-clause which goes into effect when x is closer to zero than a certain threshold. In my solution I chose to represent the function f within this interval by utilizing a Taylor expansion of f around $x = 0$

$$f(x) = \frac{1}{2} - \frac{x}{6} + \frac{x^2}{24} - \frac{x^3}{120} + \mathcal{O}(x^4), \tag{19}$$

which has no ill-defined behaviour when $x \simeq 0$. I estimate that a reasonable threshold for where expansion should go into effect could be $|x| > 0.001$, since that leaves a good margin to the first point $x \simeq 10^{-6}$ where we know for sure that the evaluation is seriously starting to break down. Since the range $|x| > 0.001$ is still quite narrow, an expansion to the 3^{rd} order should work well enough for us to produce accurate results for most purposes, since the error in a third order expansion is at will scale as $\mathcal{O}(x^4)$.

References

[1] Wikipedia, "long double". [Online]. Available: https://en.wikipedia.org/wiki/Long_double.