Ruprecht-Karls University of Heidelberg
Institut für Theoretische Astrophysik
Prof. Dr. Ralf S. Klessen
Tutor: Marcelo Barraza
Due Date: 23/10/2019

Teammates:
H.Lüdemann (3589643),
D.Wolf (3548138)
F.Wallberg (3548199)

1 Pitfalls of integer and floating point arithmetic

The aim of the first exercise is to illustrate some pitfalls one can encounter when dealing with in integer and floating point arithmetic in C and C++.
For this purpos some code samples are provided which should be executed and the results explained.

1. Sample

The first code sample shows the difference between integer and floating point devision.

```
int   i = 7;
float y = 2*(i/2);
float z = 2*(i/2.);
printf("%e␣%e␣\n", y,z);
```

If executed this code returns the following numbers to the console for x and y.

```
y = 6.000000e+00
x = 7.000000e+00
```

Clearly the first number is not the expected result. The reason for this is that the calculation of i/2 is done as an integer devision. This devision cuts of the floating part of the number. So the calculation 7/2 gives the result 3. This is then multiplied by 2 and cast into a floating point number. But if one of the numbers of the devision is marked as a floating point number like 7/2. then the correct result is calculated.

2. Sample

```
double a = 1.0e17;
double b = -1.0e17;
double c = 1.0;
double x = (a + b) + c;
double y = a + (b + c);
```

If this second code sample is executed the following result is printed to the console for x and y.

```
x = 1.000000e+00
y = 0.000000e+00
```

Again it is clear that only the result for x is correct. The problem with the second part is the summation of b and c which are numbers of very different magnitude. When adding the numbers the small value will be converted into a representation with the same exponent as the larg number. But the mantissa of the floating point number can only cover a certain range and 1.0 falls beneath the machine precision.
As a result c is not significant in the addition of the small and the large number and therefor only the large number is returned which leads to an incorrect final result. This problem does not occure if the two large numbers are substracted first.
So the reason why the law of associativity is broken in this example is that computers can only

add numbers with the same exponent and that the mantissa of a floating point number can only cover numbers in a certain range.

3. Sample

```
float x = 1e20;
float y;
y = x * x;
printf("%e␣%e\n", x,y/x);
```

The result of the last sample code is:

```
x = 1.000000e+20
y/x = inf
```

The problem in this case is that the maximum number that can be represented with a float is approximately $3,4\text{e}38$ and therefore $y = 1\text{e}40$ will actually be represented as infinity. This is the reason why the devision y/x also gives infinity as a result.

2 Near-cancelation of numbers

In this section the computation of the function

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

should be analysed.

2.1 Calculating the lim.

First the limit $\lim_{x\to 0} f(x)$ should be calculated. This can easily be done with L'Hôpital's rule.

$$\lim_{x\to 0} \frac{x + e^{-x} - 1}{x^2} = \lim_{x\to 0} \frac{1 - e^{-x}}{2x} = \lim_{x\to 0} \frac{1 + e^{-x}}{2} = \frac{1}{2} \tag{2}$$

2.2 Interactive program to compute the function.

The interactive program for the computation of this function is lokated in the file *part2.cpp*. For easy compilation a *CMakeLists.txt* file is included to use with *cmake*. The different parts of the program can be used by specifying different command line arguments. For the interactive program choose 0 and for the evaluation of the function for small values choose 1 as a command line argument.

2.3 Analyse critical small values.

If the function is evaluated for small values the formula goes wrong. To determine the value of x for which this happens, the function is evaluated for a range of small values and the results are plotted in figure 1.
It can be easily seen that for values of x arount $5 \cdot 10^{-6}$ the formula fails.

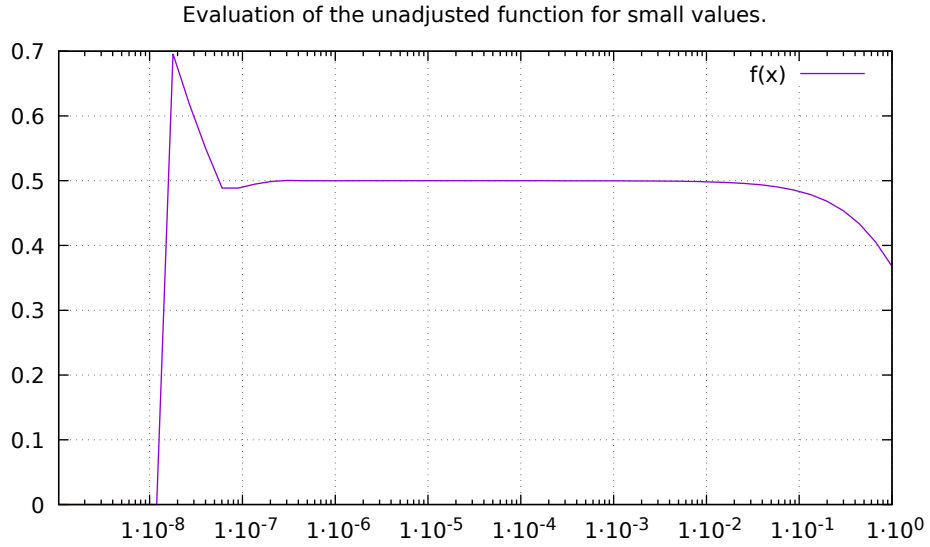

Figure 1: Evaluation of the function for small values.

2.4 Explanation for the failure of the formula for small values

From the taylor expansion it can be gathered that the second order term is the relevant term for small values of x. As such, given a mantissa of length m and a precision p, the evaluation fails if

$$x^2 < 2^{-(m-p)} \tag{3}$$

$$\Rightarrow x < 2^{-\frac{m-p}{2}} \tag{4}$$

since in those cases, x^2 cannot be reepresented to the given precision within the exponential function.
for a double ($m = 53$) with a precision of $p = 3$, this turns out to be

$$x_{min} = 2^{-\frac{53-3}{2}} = 2^{-25} \approx 2.98 \cdot 10^{-8} \tag{5}$$

to get good values for the function, the third order term of the taylor expansion should also be included, which similarily breaks down at

$$x_{min,err} = 2^{-\frac{m-p}{3}} \approx 9.61 * 10^{-6} \tag{6}$$

2.5 Adjustment for the calculation of the function.

To get around the problem with the small values an if-statement can be added to the execution of the function. This conditional case will only be executed if the values of x are beneath the threshhold determined previously.
In order to get correct results for values beneath this threshhold a different formular is needed. This can be done with the taylor expansion of the function (1) which is given by:

$$f(x) = \sum_{n=0}^{\inf} \frac{(-x)^n}{(n+2)!} \tag{7}$$

Using the first three terms of this formular as an adustment to the function the overall results become reasonable. This is illustrated in figure 2.

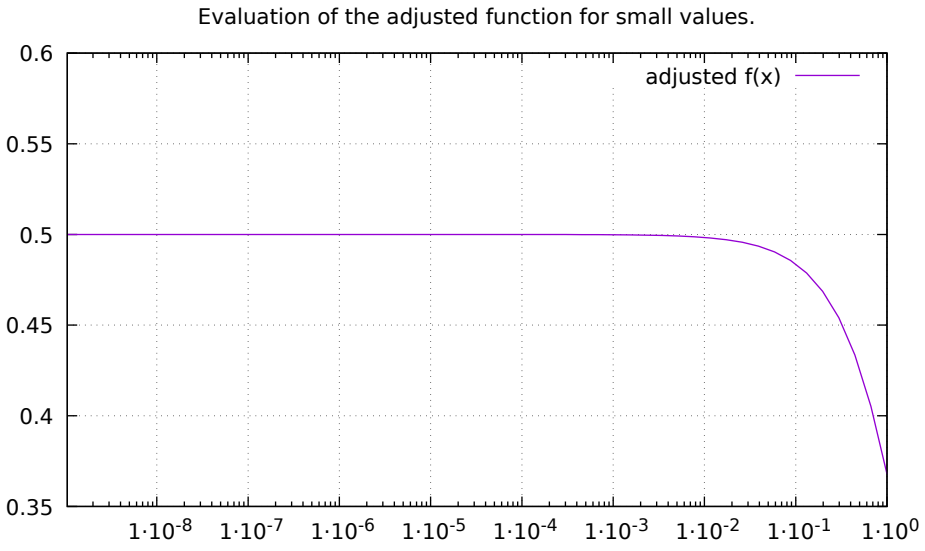

Figure 2: Evaluation of the adjusted function for small values.