Riccardo Cusinato - 3627460
Giacomo Barzon - 3626438
Damiano Sgarbossa - 3623971

Assignment 1

Exercise 1

1. Consider the following C/C++ code:

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

which prints out two float numbers. Explain why the numbers are not all equal.

Solution In the first case both numbers involved in the division are integers, thus a division between integers is performed and the result is another integer, i.e. 3, so y=6.0 (because y is a *float*). In the second case, 2. is a *float* and the division is performed with real numbers, giving the correct result of 3.5, so z=7.0.

2. Consider the following numbers:

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

Calculate the results for x and y. Which one is correct, if any? Explain, why the law of associativity is here broken.

Solution In the first case the result for x is 1.000000e+00, while in the second one y=0.000000e+00. The second one is wrong and this happens because the double-precision has 16 significant digits of precision, so 1 is beyond the machine precision when summed to b and thus the result of (b+c) is again just b. Obviously the first one is good because the two numbers are exactly the same but opposite and they cancel out.

3. Consider the following C/C++ code:

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

Explain what you see.

Solution In the first case the number 1.000000e+20 is displayed. While in the second case "inf" is displayed, meaning that an infinite value is associated to the operation; this happens because the program is trying to compute a number (1e40) which exceed the maximal limit for *float* numbers (in 32 bits is $3.4 \cdot 10^{38}$) so it save the value of y as infinite, then obviously y/x remains inf.

Exercise 2

Consider the following function:

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

Clearly for x = 0 this function is ill determined. However, for the limit x → 0 the function goes to a non-zero and non-infinite value.

1. Determine $\lim_{x\to 0} f(x)$

   The exponent in f(x) can be expanded up to the second order, obtaining $e^{-x} \approx 1 - x + x^2/2$, thus obtaining $\lim_{x\to 0} f(x) = 1/2$.

2. Write a computer program that asks for a value of x from the user and then prints $f(x)$.

   The code in Python is the following:

```
def f(x):
    return (x+np.exp(-x)-1.)/(x**2)

val = input("Value of x: ")
print("f(x =",float(val),")=",f(float(val)))

Value of x: 2.0
f(x = 2.0 )= 0.2838338208091532
```

3. For small (but positive non-zero) values of x this evaluation goes wrong. Determine experimentally at which values of x the formula goes wrong.

   The strange behaviour of the function $f(x)$ can be exploited by writing the following code that shows what happens for small values of x:

```
# Defining precision parameters for finding
# the incorrect range
epsilonx = 1E-6
epsilony = 0.0

# Searching for first incorrect evaluated number
lin = np.arange(0.0, 1E-4, epsilonx)
x_wrong = lin[1]
corr = np.zeros(len(lin))
for i in range(len(lin)-1,1,-1):
    if (f(lin[i])>(f(lin[i-1])+epsilony) or
                        f(lin[i])>(0.5+epsilony)):
        x_wrong = lin[i]
        print("The evaluation goes wrong at x = ", lin[i])
        break
```

The plot is the following:

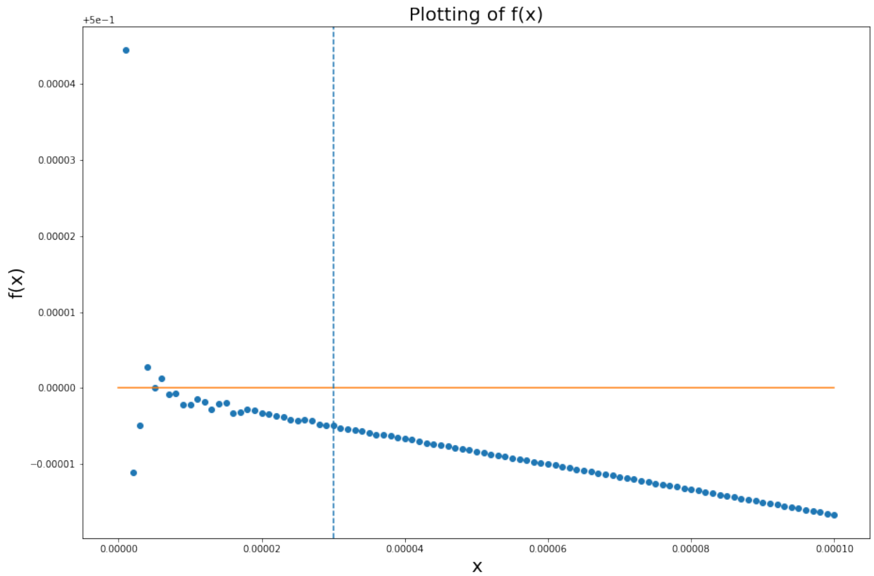

Figure 1: Plot of the behaviour of the function for small numbers.

It is possible to notice two different problematic behaviour in the function approaching 0: the first one is the clearer one and is the fact that approaching 0, some values of the function are evaluated completely differently from what one should expect; the other one, more subtle, is that, while the function is correctly approaching 1/2, it is not completely monotone as it should be; this means that in some cases for $x' < x$ the function assumes the value $f(x') < f(x)$ (instead of a bigger value). So we decided to set the value x_{wrong} when this happens, with some tolerance. In setting the tolerance on the y-axis (epsilony in the code) we noticed that, for a fixed grid spacing epsilonx, when diminishing this value after a certain point, the value of x_{wrong} didn't change anymore, so we directly set the value of epsilony at 0.0. This procedure gives a value $x_{wrong} = 3.0 \cdot 10^{-5}$.

4. Explain why this happens.

The reason why the function gives wrong (imprecise) results for small x is the rounding error; in fact, like in the 2nd part of exercise 1, there is the difference between 2 numbers that nearly cancel $x + e^{-x}$ (which for x near zero is a little less than 1) and 1. For this motivation there is a loss of precision so it's reduced the number of significant digits. Then performing the division with a very small number (x^2) emphasizes this loss of precision, so that even small inaccuracies are shown up.

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.

   The correct behaviour is restored if we define the following function:

```
def correct_f(x):
    if x > x_wrong:
        return (x+np.exp(-x)-1.)/(x**2)
    else:
        return 0.5 - x/6.

correct_f = np.vectorize(correct_f)
```

   This function is nothing but $f(x)$ in which the exponential is Taylor expanded up to the third order for small values of x. In this way we can use the new function only on values $x < x_{wrong}$, obtaining the following new behaviour:

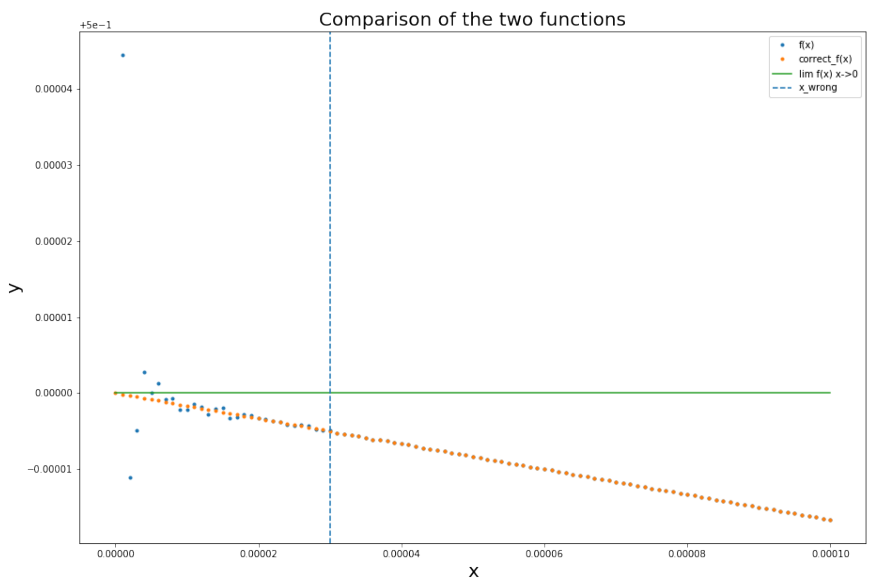

Figure 2: Plot of the behaviour of the function for small numbers after the correction for small x.