List 1 report

Albert Kołodziejski

October 22, 2023

Exercise 1

Description of problem:

The main task in this exercise is to make some known floating point constants iteratively to get better understanding of theirs meaning.

Results:

	Float16	Float32	Float64
eps()	0.000977	1.1920929e-7	2.220446049250313e-16
my_eps()	0.000977	1.1920929e-7	2.220446049250313e-16
float.h	-	1.19209e-07	2.22045e-16
nextfloat()	6.0e-8	1.0e-45	5.0e-324
my_eta()	6.0e-8	1.0e-45	5.0e-324
floatmax()	6.55e4	3.4028235e38	1.7976931348623157e308
my_max()	6.55e4	3.4028235e38	1.7976931348623157e308
float.h	-	3.40282e+38	1.79769e+308

QA:

How macheps relate to precision of arithmetic?

Precision of arithmetic is an upper bound of relative error,
eps = 2^{-t}.
Macheps is distance to next bigger number represented in that arithmetic.
macheps = $2^{-(t - 1)}$

$$\text{macheps} = 2^{-(t - 1)} = 2^{-t + 1} = 2^{-t} * 2 = \text{eps} * 2$$

What is the relationship between the number eta and the number MIN_{sub}?

MIN_{sub} is the smallest subnormal number that can be represented. Subnormal means that it mantissa starts with 0 instead of 1. Eta is the next number after zero. Both are the same. In my results it is hard to see because Julia rounds those numbers, if we would take bits of both number we would see that they are the same.

What does the function floatmin() return and what is the relationship of with $\mathrm{MIN_{nor}}$?

floatmin() returns a minimal normalized number, so it is equal to $\mathrm{MIN_{nor}}$.

Interpretation and conclusions:

Naming conventions are here a little tricky, machine epsilon is not epsilon and there are 2 completely different minimal values, so better check it before blindly running floatmin() in any language.

Exercise 2

Description of problem:

Test Khan formula that may compute machine epsilon.

Results:

	Float16	Float32	Float64
experiment()	-0.000977	1.1920929e-7	-2.220446049250313e-16
eps()	0.000977	1.1920929e-7	2.220446049250313e-16

Interpretation and conclusions:

We can get epsilon from this formula, but if the numbers of bit used for mantissa is odd, we will get negative value. There are tricky ways you can play with float arithmetic to get what you want.

Exercise 3

Description of problem:

Test how numbers are distributed in float arithmetic.

Results:

1 + 0step	0011111111110000000000000000000000000000000000000000000000000000
1 + 1step	0011111111110000000000000000000000000000000000000000000000000001
1 + 2step	0011111111110000000000000000000000000000000000000000000000000010
1 + 3step	0011111111110000000000000000000000000000000000000000000000000011
1 + (2^{52}-2)step	0011111111111111111111111111111111111111111111111111111111111110
1 + (2^{52}-1)step	0011111111111111111111111111111111111111111111111111111111111111
1 + 2^{52}step	0100000000000000000000000000000000000000000000000000000000000000

Adding 1 to the end of mantissa should create all numbers between 1 and 2

QA:

How numbers are distributed in [0.5, 1] and how can be represented?

They are distributed evenly with step = 2^{-53}, they can be represented as
x = 0.5 + k*step, where k = 1, 2, ..., 2^{51} - 1

How numbers are distributed in [2, 4] and how can be represented?

They are distributed evenly with step = 2^{-51}, they can be represented as
x = 2 + k*step, where k = 1, 2, ..., 2^{51} - 1

Interpretation and conclusions:

Normalized numbers are distributed evenly between two neighbor powers of 2, the farther from range [1, 2] you are, the smaller amount of numbers distributed.

Exercise 4

Description of problem:

Find the smallest number bigger than that, breaks 1 * (1/x) = 1 formula in Float64 arithmetic.

Results:

1.000000057228997

Interpretation and conclusions:

Never use equal sign when dealing with float arithmetic, because it can be counter-intuitive.

Exercise 5

Description of problem:

Find the best way to compute scalar product when vectors are almost perpendicular.

Results:

	result	error	relative error
forward Float32	-0.4999443	0.49994429944939167	4.967e10
backward Float32	-0.4543457	0.4543457031149343	4.514e10
ascending on Float32	-0.5	0.4999999999899343	4.967e10
descending Float32	-0.5	0.4999999999899343	4.968e10
forward Float64	1.0251881368296672e-10	1.1258452438296671e-10	11.18
backward Float64	-1.5643308870494366e-10	1.4636737800494365e-10	14.54
ascending on Float64	0.0	1.0065710699999998e-11	1.0
descending Float64	0.0	1.0065710699999998e-11	1.0

Interpretation and conclusions:

There are no good solutions, one can only pray to god that they are not perpendicular.

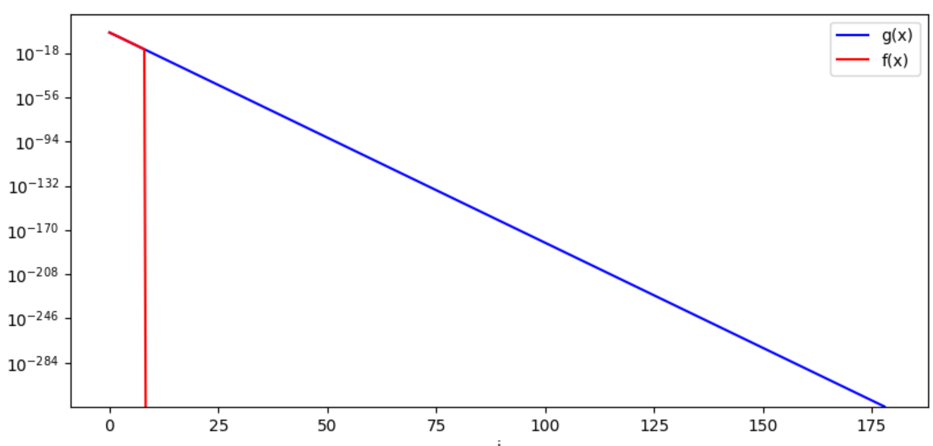
Exercise 6

Description of problem:

Check what effect avoiding subtracting 2 almost the same numbers can have on results.

Results:

This formula should never reach zero for any x bigger than zero. A good check would be finding biggest i, that for x = 8^{-i} function is still bigger than zero.



	f(x)	g(x)
i = 1	0.0077822185373186414	0.0077822185373187065
i = 8	1.7763568394002505e-15	1.7763568394002489e-15
i = 9	0.0	2.7755575615628914e-17
i = 178	0.0	1.6e-322
i = 179	0.0	0.0

QA:

Although f() and g() give different results, which we should trust more?

As we can see, g() stays non-zero for much longer, so we should trust it more.

Interpretation and conclusions:

We should try to avoid subtracting 2 numbers that are close to each other by reformulating.

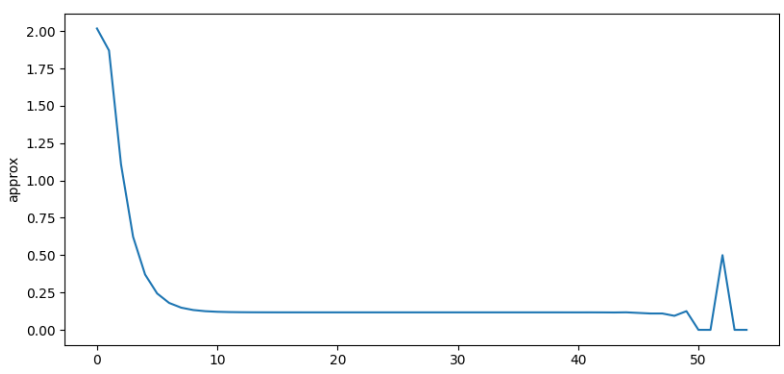
Exercise 7

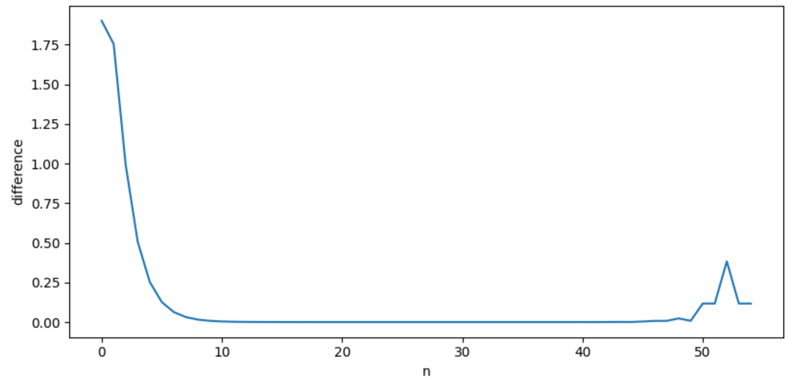
Description of problem:

Test behavior of a function that is supposed to approximate derivative at the point in Float64 arithmetic.

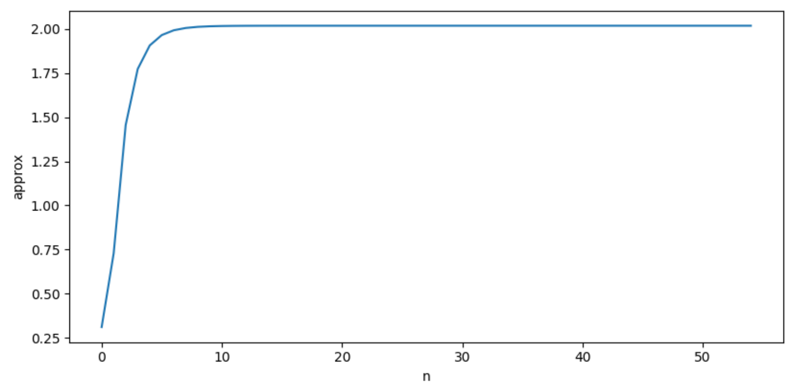
Results:

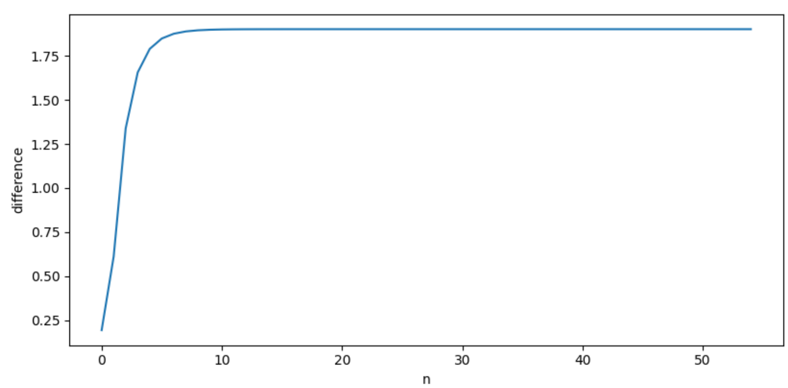
For h:




difference
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
10
20
30
40
50
n

For 1 + h:


approx
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0
10
20
30
40
50
n


difference
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0
10
20
30
40
50
n

QA:

How to explain that, from a certain point onwards, decreasing the value of h does not improve the approximation of the of the derivative?

Take a look at those results:

	n = 27	n = 28	n = 29
f(x)	-0.1485215117925489	-0.1485215117925489	-0.1485215117925489
f(x + h)	-0.148521510**92126076**	-0.148521511**35690494**	-0.148521511**57472704**
difference	3.46e-8	4.80e-9	5.48e-8
error	3.74e-8	7.49e-8	1.50e-7

The most correct approximation is for n = 28, any bigger and lesser produce worse results. A possible reason why it is a case is that we are here subtracting two almost identical numbers, so the error is getting bigger and bigger.

How does 1 + h behave?

It is good at the begin, and it is getting worse with bigger n. But don't have weird behavior with large n.

Interpretation and conclusions:

We need to be aware of error and consequences that it can bring to the table.