

## Problem 1

The output:

$$a+b+c = 96$$

$$(a+b)+c = 96$$

$$a+(b+c) = 96$$

$$(a+c)+b = 95.196036565001179 \text{ if format long}$$

$$a+(c+b) = 96$$

b is around 0.8 and a is  $-1.10714946411818e+17$  in which the mantissa is 64B1 D50D 5D2A in hex and takes up 48 bits which can be stored in 53 bits in a double. When storing c into float, we get 110714946411818096 as c, with the ending 096 instead of our expected 100. This is because the floats are stored in binary and we cannot always represent float precisely using binary. When doing a+b, the result is  $1.107149464118180008.....e+17$ . The mantissa now become F5D 6292 5A07 38A8 which needs 60 bits to store. 60 bits is way more than the 53 bits that a double can hold so the extra digits that cannot be stored will be rounded off. So the actual mantissa we will be storing is still 110714946411818. So the result of a+b is still  $-1.10714946411818e+17$ .

The same applies to  $(a+b)+c$  because the operation order is the same as a+b+c. The same applies to  $a+(b+c)$  because  $1.107149464118181008.....e+17$  is also too big to store in a double. The same applies to  $a+(c+b)$  because b+c and c+b will produce the same result.

We get  $(a+c)+b = 95.196036565001179$  because we first do a+c which gives us  $9.6*10$ . Then we calculate  $9.6*10-8.039634349988262e-1$  and get 95.196036565001179 and we don't miss the 0.8..... number this time because we no longer have that lengthy mantissa.

## Problem 2

$$a. a=5.001, b=5.003$$

$$a+b \text{ after rounding} = 1.000e1, 1.000e1/2=5, 5 \text{ not in } [a,b]$$

$$b. a=-1e-4, b=-1, c=1.001$$

$$a+b \text{ after rounding} = -1.000, -1.000+c = 1e-3.$$

$$b+c = 1e-3, 1e-3+a = 1.1e-3$$

$$1e-3 \neq 1.1e-3$$

$$c. a=5.007, b=9.000e-1, c=2.000e1$$

$$ab \text{ after rounding} = 4.506, 4.506*c = 90.12$$

$$bc = 18, 18*a \text{ after rounding} = 90.13$$

90.12 != 90.13

Discussion: No, if  $a$  and  $b$  are IEEE-754 FP numbers and  $a \leq b$ ,  $(a+b)/2$  not in  $[a, b]$  cannot occur. Because in IEEE-754 numbers are stored in binary and when doing rounding with binary numbers, we don't lose as much precision as decimal numbers.

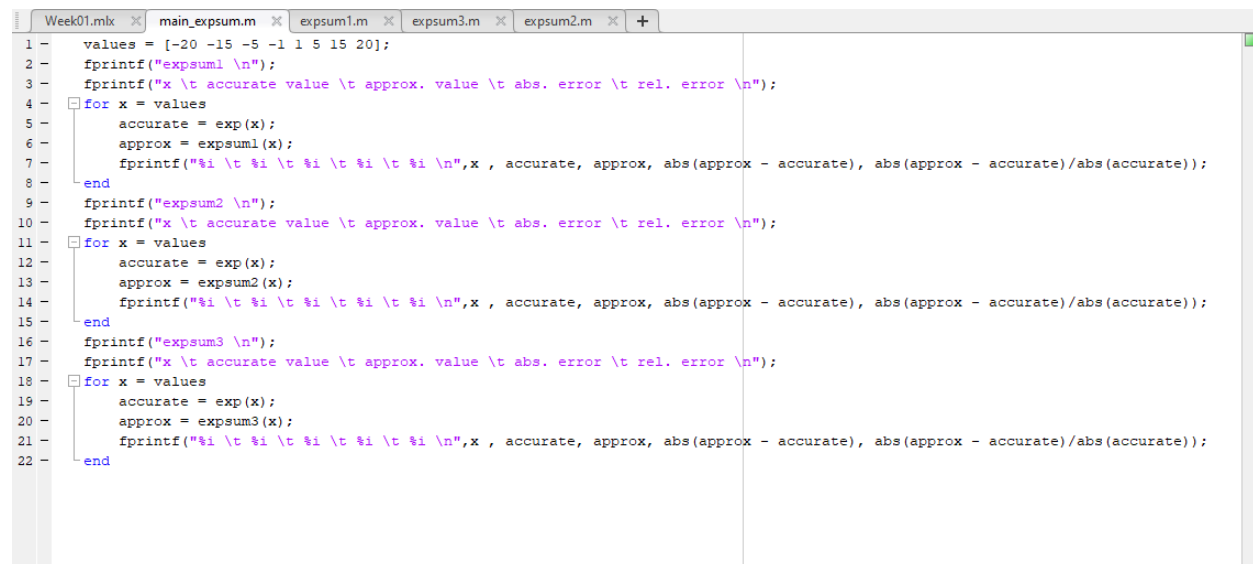
If  $\text{fl}(a) == \text{fl}(b)$ , then  $(\text{fl}(a) + \text{fl}(b))/2 == \text{fl}(a) == \text{fl}(b)$  all the time. Because either the last digit is 0 or 1 for both  $a$  and  $b$ , the last digit after the addition of  $a$  and  $b$  will always be 0, so we don't need to round in this case and therefore we don't miss any precision.

If  $\text{fl}(a) < \text{fl}(b)$ , then the worst case (the  $[a, b]$  interval is the smallest) is when  $a$  and  $b$  differs by only the last digit. But if their last digits are different, we only miss at most the last "1" when rounding. If we only miss at most the last "1" and  $a$  and  $b$  only differ by the last digit, our final result still fall into the range  $[a, b]$ . For the digits to the left of the last digit, dividing them by 2 will be compensated by the exponent. (For example,  $1.001 \cdot 2^2$  divided by 2, mantissa is still 1.001 just the exponent become  $2^1$ ).

### Problem 3

a.

main\_expsum.m code:



```
1 values = [-20 -15 -5 -1 1 5 15 20];
2 fprintf("expsum1\n");
3 fprintf("x \t accurate value \t approx. value \t abs. error \t rel. error\n");
4 for x = values
5     accurate = exp(x);
6     approx = expsum1(x);
7     fprintf("%i \t %i \t %i \t %i \t %i\n", x, accurate, approx, abs(approx - accurate), abs(approx - accurate)/abs(accurate));
8 end
9 fprintf("expsum2\n");
10 fprintf("x \t accurate value \t approx. value \t abs. error \t rel. error\n");
11 for x = values
12     accurate = exp(x);
13     approx = expsum2(x);
14     fprintf("%i \t %i \t %i \t %i \t %i\n", x, accurate, approx, abs(approx - accurate), abs(approx - accurate)/abs(accurate));
15 end
16 fprintf("expsum3\n");
17 fprintf("x \t accurate value \t approx. value \t abs. error \t rel. error\n");
18 for x = values
19     accurate = exp(x);
20     approx = expsum3(x);
21     fprintf("%i \t %i \t %i \t %i \t %i\n", x, accurate, approx, abs(approx - accurate), abs(approx - accurate)/abs(accurate));
22 end
```

expsum1(x) code:

```
Week01.mbx  main_expsum.m  expsum1.m  expsum3.m  expsum2.m  +
1  function s = expsum1(x)
2      s = 0;
3      current_term = 1;
4      i = 1;
5      while s + current_term ~= s
6          s = s + current_term;
7          current_term = x^i / factorial(i);
8          i = i + 1;
9      end
10 end
```

expsum2(x) code:

```
Week01.mbx  main_expsum.m  expsum1.m  expsum3.m  expsum2.m  +
1  function s = expsum2(x)
2      if x >= 0
3          s = expsum1(x);
4      else
5          s = 1/expsum1(-x);
6      end
7  end
```

expsum3(x) code:

```
Week01.mlx x main_expsum.m x expsum1.m x expsum3.m x expsum2.m x +
1 function s = expsum3(x)
2     pos = 0;
3     neg = 0;
4     current_term = 1;
5     i = 1;
6     while pos + current_term ~= pos && neg + current_term ~= neg
7         if current_term >= 0
8             pos = pos + current_term;
9         else
10            neg = neg + current_term;
11        end
12        current_term = x^i / factorial(i);
13        i = i + 1;
14    end
15    s = pos + neg;
16 end
```

Output:

```
>> main_expsum
expsum1
x    accurate value    approx. value    abs. error    rel. error
-20    2.061154e-09    4.173637e-09    2.112484e-09    1.024904e+00
-15    3.059023e-07    3.059055e-07    3.167241e-12    1.035376e-05
-5     6.737947e-03    6.737947e-03    1.439820e-15    2.136883e-13
-1     3.678794e-01    3.678794e-01    1.110223e-16    3.017899e-16
1      2.718282e+00    2.718282e+00    0 0
5      1.484132e+02    1.484132e+02    2.842171e-14    1.915040e-16
15     3.269017e+06    3.269017e+06    0 0
20     4.851652e+08    4.851652e+08    1.192093e-07    2.457087e-16
expsum2
x    accurate value    approx. value    abs. error    rel. error
-20    2.061154e-09    2.061154e-09    4.135903e-25    2.006596e-16
-15    3.059023e-07    3.059023e-07    0 0
-5     6.737947e-03    6.737947e-03    1.734723e-18    2.574558e-16
-1     3.678794e-01    3.678794e-01    5.551115e-17    1.508950e-16
1      2.718282e+00    2.718282e+00    0 0
5      1.484132e+02    1.484132e+02    2.842171e-14    1.915040e-16
15     3.269017e+06    3.269017e+06    0 0
20     4.851652e+08    4.851652e+08    1.192093e-07    2.457087e-16
expsum3
x    accurate value    approx. value    abs. error    rel. error
-20    2.061154e-09    5.960464e-08    5.754349e-08    2.791810e+01
-15    3.059023e-07    3.057066e-07    1.956854e-10    6.396989e-04
-5     6.737947e-03    6.737947e-03    9.889659e-15    1.467755e-12
-1     3.678794e-01    3.678794e-01    0 0
1      2.718282e+00    2.718282e+00    0 0
5      1.484132e+02    1.484132e+02    2.842171e-14    1.915040e-16
15     3.269017e+06    3.269017e+06    0 0
20     4.851652e+08    4.851652e+08    1.192093e-07    2.457087e-16
..
```

Discussion:

For positive x inputs, all three functions perform equally well, they have similar absolute errors and relative errors.

But for negative x inputs, the output shows expsum2 is the most accurate among the three functions. Expsum1 is the second most accurate. Expsum3 is the worst in terms of accuracy.

This is due to cancellations when x is negative and so we have alternating positive and negative terms in the expansion series.

As an example:

```
>> expsum3(-20)
```

```
pos = 2.425825977048952e+08, neg = -2.425825977048951e+08
```

Obviously in this case, sum of positive terms and sum of negative terms are sharing a lot of significant digits, so when we add them together (we are in fact doing subtraction), we lose a lot of precision because we are only able to carry a limited number of digits in a float and many of them are cancelled out during the subtraction. Then we only have a very limited number of digits (digits starting from 1 of 2.425825977048951 to the end of the last digit representable by a float) that are correctly subtracted, for digits after them we don't have information about them because we weren't able to store them in float. That's why we are getting a huge relative error for expsum3(-20).

Same thing happens to expsum1(-20) but since in expsum1(-20) we are having less subtraction calculations that experience cancellations than expsum3(-20) (because we are not subtracting two huge values when all the terms are added together and the pos and neg are sharing lots of digits), the accuracy is not as bad as expsum3. But the accuracy is still not satisfactory.

But when it comes to expsum2, since we are doing  $1/\text{expsum1}(-x)$  when  $x$  is negative, we won't have negative terms in our calculations, so we won't experience cancellations in expsum2. Therefore, the accuracy of expsum2 is the best among the three.

b. No. Like I discussed in part a), expsum3 will suffer a lot from cancellations because we are subtracting two numbers that share a lot of digits at the end after all the pos and neg terms are summed up separately. As  $|-x|$  gets larger and larger, the cancellation issue will become disastrous, and the relative error will become much larger.

#### Problem 4

a.

[linl20@mills ~] more benchmark.out

Memory required: 3914K.

LINPACK benchmark

Single precision

Digits: 6

Array size 1000 X 1000.

Average rolled and unrolled performance:

	Reps	Time(s)	DGEFA	DGESL	OVERHEAD	MFLOPS	GFLOPS
-----							
8	0.84	95.24%	1.19%	3.57%	1655.967	1.656	
16	1.68	95.24%	0.60%	4.17%	1666.253	1.666	

32	3.34	95.81%	0.00%	4.19%	1676.667	1.677
64	6.69	95.81%	0.90%	3.29%	1658.527	1.659
128	13.39	95.37%	0.67%	3.96%	1668.844	1.669

Memory required: 7824K.

LINPACK benchmark

Double precision

Digits: 15

Array size 1000 X 1000.

Average rolled and unrolled performance:

Reps	Time(s)	DGEFA	DGESL	OVERHEAD	MFLOPS	GFLOPS
8	0.97	95.88%	1.03%	3.09%	1426.950	1.427
16	1.94	96.39%	0.52%	3.09%	1426.950	1.427
32	3.88	96.65%	0.52%	2.84%	1423.165	1.423
64	7.76	97.04%	0.39%	2.58%	1419.400	1.419
128	15.51	96.84%	0.45%	2.71%	1422.222	1.422

Memory required: 15645K.

LINPACK benchmark

Long Double precision

Digits: 18

Array size 1000 X 1000.

Average rolled and unrolled performance:

	Reps	Time(s)	DGEFA	DGESL	OVERHEAD	MFLOPS	GFLOPS
-----							
	2	0.67	98.51%	0.00%	1.49%	508.081	0.508
	4	1.34	97.76%	1.49%	0.75%	504.261	0.504
	8	2.69	97.03%	0.74%	2.23%	510.013	0.510
	16	5.37	97.77%	0.93%	1.30%	506.164	0.506
	32	10.74	98.32%	0.65%	1.02%	504.735	0.505

Memory required: 15645K.

LINPACK benchmark

Float128 precision

Digits: 0

Array size 1000 X 1000.

Average rolled and unrolled performance:

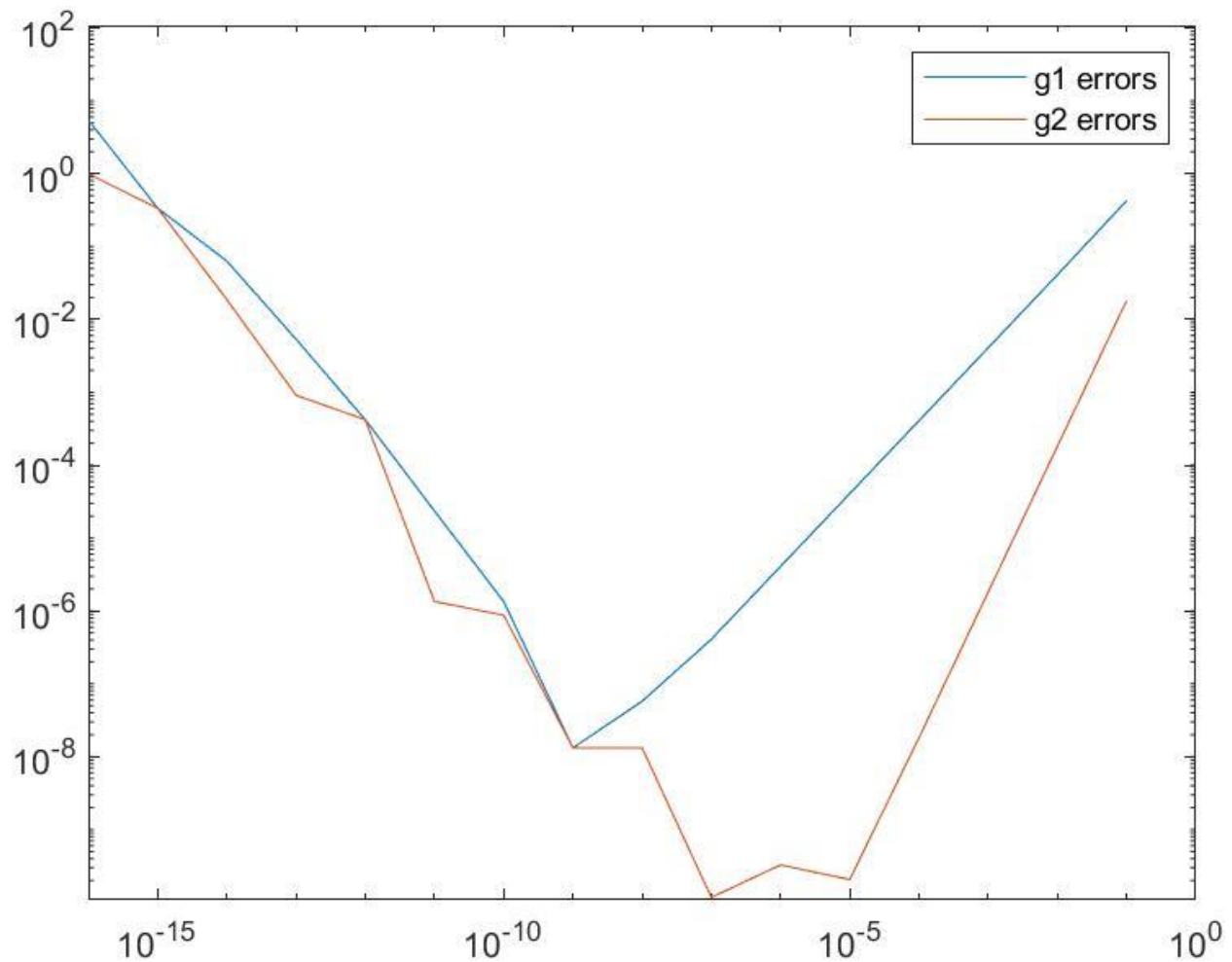
	Reps	Time(s)	DGEFA	DGESL	OVERHEAD	MFLOPS	GFLOPS
-----							
	1	22.98	99.26%	0.61%	0.13%	7.306	0.007

b. Apparently the performance measured by FLOPS decreased for the order of single, double, long double, float128. Float128 caused the performance to drop the most because its size is the largest and carry more digits so the computer needs more time to complete a floating point operation. Single precision is the most compact among the four types so we can process the most floating point operations at a given period. Double and long double are in between. This story tells us that we should consider balancing between accuracy and performance.

## Problem 5

a.





b. g1:  $h = 1e-9$ , g2:  $h = 1e-7$

c. When  $k$  is small,  $h$  is large, the error will for sure be large, because the distance between  $x_0$  and  $x_0+h$  is too large, so our approximation itself is not accurate.

When  $k$  gets larger,  $h$  becomes smaller, the error goes down as shown on the plot from  $10^0$  to the middle. It is because our approximation itself is getting more accurate.

But after the middle, we see a rise on the error. This is because when  $h$  is small enough, the difference between  $f(x+h)$  and  $f(x)$  (or  $f(x-h)$ ) is only reflected on the relatively less significant digits. But we only got a limited number of digits to store the float, so the relatively less significant digits get truncated. So we first experience truncation error. Then, because of the truncation, the digits stored in the floats are very similar, so we also experience cancellations after that. This issue becomes more severe when  $h$  gets smaller and smaller, because  $f(x+h)$  and  $f(x)$  (or  $f(x-h)$ ) are getting more and more similar digits.

For example, when  $k$  is 16,  $(1+h) * \exp(1+h) = 2.718281828459046$ ,  $1 * \exp(1) = 2.718281828459046$ ,  $(1-h) * \exp(1-h) = 2.718281828459045$ . Apparently they share a lot of common digits. This is because  $h$  is too small and the only differentiating digits cannot be stored in floats, so we experience truncation error. Then when we subtract the floats that share a lot of common digits, we experience cancellation error.

## Problem 6

Matlab code:

```
main_expsum.m x expsum1.m x expsum3.m x expsum2.m x main_sum.m x main_deriv.m x +
1 n = 10000;
2
3 decrease_sum = 0.0;
4 for i = 1:n
5     decrease_sum = decrease_sum + 1/i;
6 end
7
8 increase_sum = 0.0;
9 for i = n:-1:1
10    increase_sum = increase_sum + 1/i;
11 end
12
13 kahan_summation_sum = 0.0;
14 c = 0.0;
15 for i = 1:n
16     y = 1/i - c;
17     t = kahan_summation_sum + y;
18     c = (t - kahan_summation_sum) - y;
19     kahan_summation_sum = t;
20 end
21
22 accurate_sum = 0.0;
23 c = 0.0;
24 for i = 1:n
25     y = vpa(1/i) - c;
26     t = accurate_sum + y;
27     c = (t - accurate_sum) - y;
28     accurate_sum = (t);
29 end
30
31 fprintf("decreasing order error %i \n", double(abs(decrease_sum - accurate_sum)));
32 fprintf("increasing order error %i \n", double(abs(increase_sum-accurate_sum)));
33 fprintf("Kahan's sum error %i \n", double(abs(kahan_summation_sum-accurate_sum)));
```

Output

```
>> main_sum
decreasing order error 3.406305e-14
increasing order error 3.240446e-15
Kahan's sum error 3.122680e-16
```

Discussion: Decreasing order had the worst error because we start from the largest numbers and when largest numbers are added together, they form even larger numbers. When the sum gets too large and we add the sum to smaller numbers, the smaller numbers will become too small for the sum to add because we only get a limited number of digits that can be stored in a float. So we start missing the smaller numbers after then.

Increasing order performed better than decreasing order because we start with adding numbers that do not have that much difference so we can preserve more precision. We certainly will miss precision when the sum gets too large to store in a float but it is better than decreasing order.

Kahan summation worked the best among the three because it keeps a variable to accumulate small errors so that we won't miss as much numbers as the previous two methods.

I computed the accurate result by using vpa on kahan summation because kahan is the most accurate among the three and using vpa supports more digits so we lose not much precision.