COMPSCI 4X03

Assignment 1

Mingzhe Wang

McMaster University

October 25, 2021

Problems 1

q1

```
clc
a = -1.10714946411818e+17;
b = -8.039634349988262e-01;
c = 1.107149464118181e+17;
fprintf('%e\n', a+b+c)
fprintf('%e\n', (a+b)+c)
fprintf('%e\n', a+(b+c))
fprintf('%e\n', (a+c)+b)
fprintf('%e\n', a+(c+b))
```

output

```
        9.600000e+01
        9.600000e+01
        9.600000e+01
        9.519604e+01
        9.600000e+01
```

The accurate result should be $vpa(a)+vpa(b)+vpa(c) = vpa(a)+vpa(c)+vpa(b) = 9.519604e+01$. And only the $(a+c)+b$ equation calculate to this most accurate result. The reason is that by default, MATLB uses 16 digits of precision, which means when we calculate $a+b$ or $b+c$, due to the large difference between them (beyond 16 digit precision), the smaller number's precision is lost, which induces inaccuracy. However, by doing $(a+c)+b$, the $a+c$ part cancel a lot of digits because their digits are the same except the last one. Then the result $(a+c)$ now becomes a small number, whose magnitude does not have a large difference with the b's. Therefore, this method $(a+c)+b$ gives us the most accuracy result in this case.

Problems 2

a. If $a = 9.999 \times 10^1$, $b = 9.999 \times 10^1$, then $(a+b)/2 = 1.000 \times 10^2 \notin \{9.999 \times 10^1\}$.

b. If $a = 1.234 \times 10^3$, $b = 4.547 \times 10^1$, $c = 0.004 \times 10^{-1}$, then $a+(b+c) = 1.280 \times 10^3 \neq 1.279 \times 10^3 = (a+b)+c$.

c. If $a = 1.234 \times 10^3$, $b = 4.547 \times 10^1$, $c = 0.004 \times 10^{-1}$, then $a*(b*c) = 2.245 \times 10^1 \neq 2.244 \times 10^1 = (a*b)*c$.

No. By assumption $a < b$, we have $(a+b)/2 < (b+b)/2 = b$. In IEEE-754 FP system, for $x \leq y$, $fl(x) \leq fl(y)$, which is also the hint, so we have $fl((a+b)/2) \leq fl(b)$. Because b is an IEEE-754 FP number, $fl(b) = b$. Therefore, $fl((a+b)/2) \leq b$. Similarly, we can show $a \leq fl((a+b)/2)$. Therefore $(a+b)/2 \notin [a, b]$ cannot occur.

Problems 3

expsum1

```
function s = expsum1(x)
    s = 0;
    i = 0;
    while (1)
        term = x ^ i / factorial(i);
        if (s == s + term)
            break;
        end
        s = s + term;
        i = i + 1;
    end
end
```

expsum2

```
function s = expsum2(x)
    if (x >= 0)
        s = expsum1(x);
    else
        s = 1 / expsum1(-x);
    end
end
```

expsum3

```
function s = expsum3(x)
    pos_s = 0;
    neg_s = 0;
    i = 0;
    while (1)
        term = x ^ i / factorial(i);
        if (term > 0)
            if (pos_s == pos_s + term)
                break;
            end
            pos_s = pos_s + term;
        else
            if (neg_s == neg_s + term)
                break;
            end
            neg_s = neg_s + term;
        end
        i = i + 1;
    end
    % reason why cancellation cannot be avoid.
    % fprintf("pos_s: %e\n", pos_s);
    % fprintf("neg_s: %e\n", neg_s);
    s = pos_s + neg_s;
end
```

main_expsum

```
xs = [-20,-15,-5,-1,1,5,15,20];

% expsum1 results
fprintf('expsum1 result\n');
fprintf('   x     accurate value        approx.value      abs.error rel. error\n');
for i = 1:length(xs)
    x = xs(i);
    accurate_value = exp(x);
    approximate_value = expsum1(x);
    abs_error = abs(approximate_value - accurate_value);
    rel_error = abs_error / abs(accurate_value);
    fprintf("%5.1f  %.12e  %.12e  %.2e  %.2e \n", x, accurate_value, approximate_value, abs_error, rel_error);
end
fprintf("\n");

% expsum2 results
fprintf('expsum2 result\n');
fprintf('   x     accurate value        approx.value      abs.error rel. error\n');
```

```
for i = 1:length(xs)
    x = xs(i);
    accurate_value = exp(x);
    approximate_value = expsum2(x);
    abs_error = abs(approximate_value - accurate_value);
    rel_error = abs_error / abs(accurate_value);
    fprintf(”%5.1f  %.12e  %.12e  %.2e  %.2e \n”, x, accurate_value,
       approximate_value, abs_error, rel_error);

end
fprintf(”\n”);

% expsum3 results
fprintf('expsum3 result\n');
fprintf('   x      accurate value         approx.value      abs.error rel.
   error\n');
for i = 1:length(xs)
    x = xs(i);
    accurate_value = exp(x);
    approximate_value = expsum3(x);
    abs_error = abs(approximate_value - accurate_value);
    rel_error = abs_error / abs(accurate_value);
    fprintf(”%5.1f  %.12e  %.12e  %.2e  %.2e \n”, x, accurate_value,
       approximate_value, abs_error, rel_error);

end
fprintf(”\n”);
```

output

```
>> main_expsum
expsum1 result
   x      accurate value         approx.value      abs.error rel.error
-20.0  2.061153622439e-09  4.173637499438e-09  2.11e-09  1.02e+00
-15.0  3.059023205018e-07  3.059054877425e-07  3.17e-12  1.04e-05
 -5.0  6.737946999085e-03  6.737946999087e-03  1.44e-15  2.14e-13
 -1.0  3.678794411714e-01  3.678794411714e-01  1.11e-16  3.02e-16
  1.0  2.718281828459e+00  2.718281828459e+00  0.00e+00  0.00e+00
  5.0  1.484131591026e+02  1.484131591026e+02  2.84e-14  1.92e-16
 15.0  3.269017372472e+06  3.269017372472e+06  0.00e+00  0.00e+00
 20.0  4.851651954098e+08  4.851651954098e+08  1.19e-07  2.46e-16

expsum2 result
   x      accurate value         approx.value      abs.error rel.error
```

```
-20.0  2.061153622439e-09  2.061153622439e-09  4.14e-25  2.01e-16
-15.0  3.059023205018e-07  3.059023205018e-07  0.00e+00  0.00e+00
 -5.0  6.737946999085e-03  6.737946999085e-03  1.73e-18  2.57e-16
 -1.0  3.678794411714e-01  3.678794411714e-01  5.55e-17  1.51e-16
  1.0  2.718281828459e+00  2.718281828459e+00  0.00e+00  0.00e+00
  5.0  1.484131591026e+02  1.484131591026e+02  2.84e-14  1.92e-16
 15.0  3.269017372472e+06  3.269017372472e+06  0.00e+00  0.00e+00
 20.0  4.851651954098e+08  4.851651954098e+08  1.19e-07  2.46e-16

expsum3 result
   x       accurate value         approx.value       abs.error rel.error
-20.0  2.061153622439e-09  5.960464477539e-08  5.75e-08  2.79e+01
-15.0  3.059023205018e-07  3.057066351175e-07  1.96e-10  6.40e-04
 -5.0  6.737946999085e-03  6.737946999095e-03  9.89e-15  1.47e-12
 -1.0  3.678794411714e-01  3.678794411714e-01  0.00e+00  0.00e+00
  1.0  2.718281828459e+00  2.718281828459e+00  0.00e+00  0.00e+00
  5.0  1.484131591026e+02  1.484131591026e+02  2.84e-14  1.92e-16
 15.0  3.269017372472e+06  3.269017372472e+06  0.00e+00  0.00e+00
 20.0  4.851651954098e+08  4.851651954098e+08  1.19e-07  2.46e-16
```

a. As showed above, the accuracy (measured by relative error) of expsum1, expsum2, expsum3 for positive input is the same. While for negative input, expsum2 has the highest accuracy, expsum3 has the lowest accuracy, with expsum1 in the middle.

Note the methods are corresponding to the following equations:

$$\begin{aligned}
&e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... \\
&e^x = \frac{1}{e^{-x}} = \frac{1}{1 + (-x) + \frac{x^2}{2!} + (-\frac{x^3}{3!}) + ...} \\
&e^x = \mathsf{pos\_terms} + \mathsf{neg\_terms} = (1 + \frac{x^2}{2!} + ...) + (x + \frac{x^3}{3!} + ...)
\end{aligned}$$

For postive input, there is no cancellation error for all of this methods, because all the terms are positive. So the accuracy should be the same.

However, for negtive input, note the second method doesn't have any cancellation error because all terms are positive, it has the highest accuracy. While the first method has cancellation error between each pair of even and odd terms, that's why its accuracy is the lowest. Finally, it seems the third equation doesn't have cancellation error either in pos_terms or neg_terms, when the two sub-summations finally are added up, they also have cancellation error, because when the n is large enough, the two sub-sum should approaches the same abs value.

b. No. As the above reasoning, even there is no cancellation error either in pos_terms or neg_terms, when the two sub-summations are finally added up, they also have cancellation error, because when the n is large enough, the two sub-sum should approaches the same abs value with different signs.

Problems 4

a.

```
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    0.83  96.39%   0.00%    3.61%   1676.667       1.677
      16    1.68  95.83%   0.00%    4.17%   1666.253       1.666
      32    3.34  95.81%   0.30%    3.89%   1671.443       1.671
      64    6.69  95.37%   0.60%    4.04%   1671.443       1.671
     128   13.39  95.89%   0.22%    3.88%   1667.547       1.668

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    0.98  95.92%   0.00%    4.08%   1426.950       1.427
      16    1.94  97.42%   0.00%    2.58%   1419.400       1.419
      32    3.88  96.91%   0.52%    2.58%   1419.400       1.419
      64    7.77  97.04%   0.51%    2.45%   1415.655       1.416
     128   15.54  96.78%   0.51%    2.70%   1419.400       1.419

Memory required:  15645K.


LINPACK benchmark
```

```
LongDouble 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%   1.49%   0.00%    500.498      0.500
       4   1.35  97.04%   0.74%   2.22%    508.081      0.508
       8   2.69  98.88%   0.37%   0.74%    502.372      0.502
      16   5.38  97.58%   0.56%   1.86%    508.081      0.508
      32  10.77  98.24%   0.46%   1.30%    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  23.00  99.26%   0.61%   0.13%      7.299      0.007
```

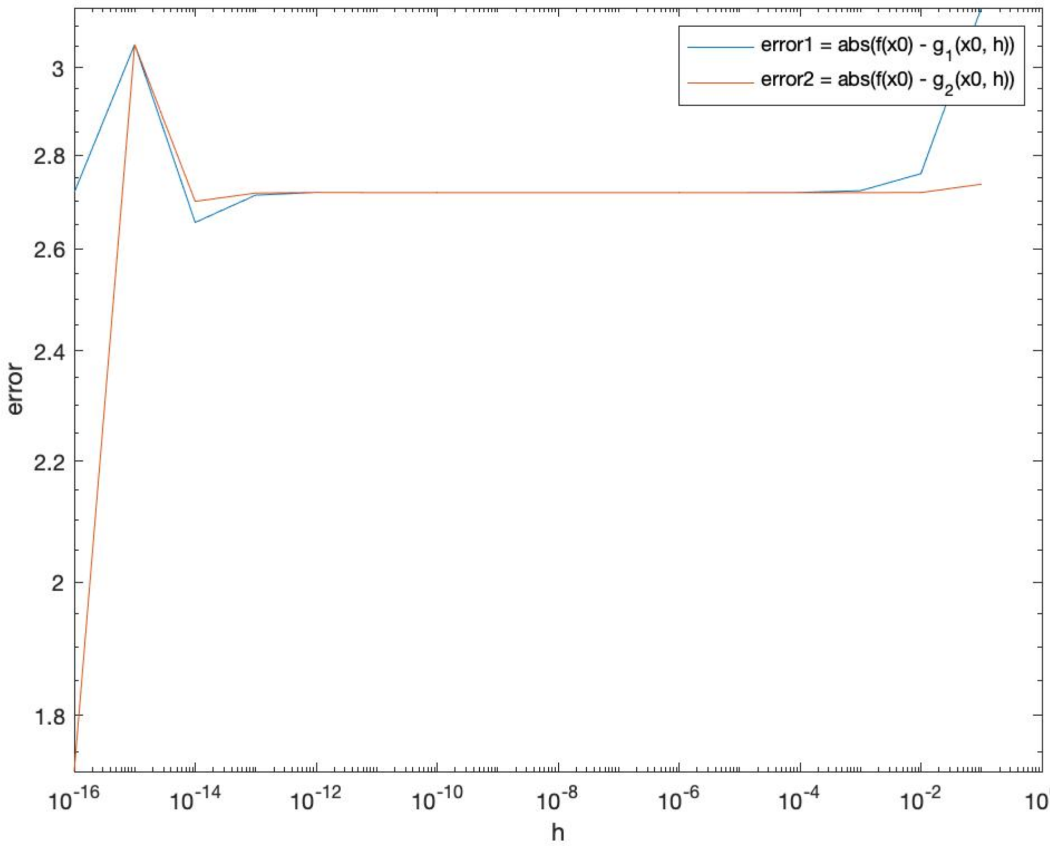
b.

As the above result shows, for respond speed, these data types has the following order (from the most fast to the most slower): single precision, double precision, long double precision, float128. In particular, for 8 times' respond, the comsumed time is 0.83s, 0.98s, 2.69s; and the giga float point operations performed are 1.677, 1.427, 0.502, separately. It should be noted that the float128 could only respond 1 time in 23.00 seconds, which causes only one line result is output because it is beyond the time limit.

The above result shows that for longer precision data types, the float point operations on them will take longer time.

Problems 5

a.



b.

The minimum error1 is 2.655198e+00, which can be achieved by h = 1.000000e-14.
The minimum error2 is 1.722610e+00, which can be achieved by h = 1.000000e-16.

c.

For error2, its truncation error is $O(h^2)$ and the original approximation can be simplified to the following:

$$f'(x) \approx \frac{(1+h)e^{1+h} - (1-h)e^{1-h}}{2h}.$$

Note when $h \approx 0$, the cancellation error could appear. But also its truncation error will approaches 0, which could be able compensate the increasing of cancellation error. It should also be noted that the denominator part, compared with the error1's, is also has a coefficient of 2, which can also compensate the increasing of cancellation error.

The plot shows when h is $1.000000e^{-16}$, which is the most nearest value of h for all input k, the error2 is the smallest. Based on the above reasoning, this plot result is reasonable.

For error1, its truncation error is $O(h)$ and the original approximation can be simplified to the following:

$$f'(x) \approx \frac{(1+h)e^{1+h} - e}{h}.$$

Similarly, when $h \approx 0$, the cancellation error appears. However, compared to the error1: First, the domination's coefficient is 1, which has a lower contribution for compensating the increasing of cancellation error. Second, the because the truncation error is $O(h)$, which has a lower power than error1, when the $h \approx 0$, the decreasing truncation error also has a lower contribution for compensating the increasing of cancellation error.
The plot shows when h is $1.000000e^{-14}$, which is the some value of h for all input k, the error1 is the smallest. This plot result is also reasonable based on the above reasoning, the point where $h = 1.000000e^{-14}$ should be somewhere the h value contributes the most to decreasing both the truncation and cancellation error.

Problems 6

decSum

```
function sum = decSum(input)
     des_input = input;
     des_input = sort(des_input, 'descend');
     sum = 0;
     for i = 1:1:size(input, 2)
          sum = sum + des_input(i);
     end
end
```

incSum

```
function sum = incSum(input)
     inc_input = input;
     inc_input = sort(inc_input, 'ascend');
     sum = 0;
     for i = 1:1:size(input, 2)
          sum = sum + inc_input(i);
     end
end
```

kahanSum

```
function sum = kahanSum(input)
    sum = 0;
    c = 0;
    for i = 1:1:size(input, 2)
        y = input(i) - c;
        t = sum + y;
        c = (t - sum) - y;
        sum = t;
    end
end
```

accSum

```
function sum = accSum(input)
    vpa_input = input;
    vpa_input = vpa(vpa_input);
    sum = 0;
    for i = 1:1:size(input, 2)
        sum = sum + vpa_input(i);
    end
end
```

main_Sum

```
i = 1:1:10000;
input = 1 ./ i;

acc_value = accSum(input);

dec_value = decSum(input);
dec_error = abs(dec_value - acc_value);

inc_value = incSum(input);
inc_error = abs(inc_value - acc_value);

kahan_value = kahanSum(input);
kahan_error = abs(kahan_value - acc_value);


fprintf('decreasing order      %.4e\n', dec_error);
fprintf('increasing order      %.4e\n', inc_error);
fprintf('Kahan''s sum           %.4e\n', kahan_error);
```

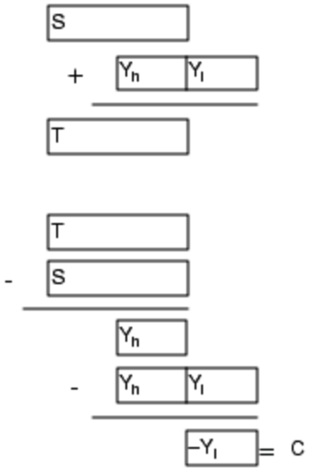
output

```
>> main_sum
decreasing order        3.4063e-14
increasing order        3.2404e-15
Kahan's sum             3.1227e-16
```

Discussion

As the result shows, the Kahan's summation algorithm is the most accurate way to compute this sum. The error in the summation is raised by the fact that when adding a small number (each term) to a large number (sum result), we loss information (the lower bit of small number), which induces the error. So decreasing order summation has the worst accuracy because when as the index grows, the difference between sum result and each term also grows, and the last terms could even be omitted.

Kahan's summation algorithm is the most accuracy way by using a correction factor c to automatically correct the sum result during the summation process. It's process can be illustrated by the following diagram, which I found from the Oracle's document. (What Every Computer Scientist Should Know About Floating-Point Arithmeti.)



Each time a summand is added, there is a correction factor *C* which will be applied on the next loop. So first subtract the correction *C* computed in the previous loop from X_j, giving the corrected summand *Y*. Then add this summand to the running sum *S*. The low order bits of *Y* (namely Y_l) are lost in the sum. Next compute the high order bits of *Y* by computing *T* - *S*. When *Y* is subtracted from this, the low order bits of *Y* will be recovered. These are the bits that were lost in the first sum in the diagram. They become the correction factor for the next loop. A formal proof of Theorem 8, taken from Knuth [1981] page 572, appears in the section Theorem 14 and Theorem 8."