Assignment 1

Jame Tran, tranj52, 400144141

Question 1

1. (a+c)+b evaluates to 95.1960, while all the other variants evaluate to 96. This is because of the truncation errors introduced when adding very small numbers to very large numbers. For example, in (a+b)+c, a very small b (b = -8.039634349988262e-01) is added to a very large a (a = -1.10714946411818e+17). Due to truncation and rounding errors, this evaluates out to be effectively a. (a + b = a). When added with c (c = 1.107149464118181e+17), this produces 96. However, with (a+c)+ b, we get 96 + b. 96 is a much smaller number than what a and c were originally, and the errors introduced by adding very small numbers to very large numbers is eliminated. With these errors minimized, we get 95.1960.

Question 2

a. a = 7.004, b = 7.005
   a+b = 14.009 ≈ 14.00
   a+b/2 ≈ 7.00, ∈/ [a, b]

b. a = 1.00E+30, b = -1.00E+30, c= 0.001
   (a+b) = 0
   (a+b)+c = 0 + 0.001 = 0.001
   b+c = -1.00E+30 + 0.001 ≈ -1.00E+30
   a+(b+c) = 1.00E+30 + (b+c) ≈ a + b = 0, hence (a+b)+c ≠ a+(b+c)

c. a = 5.000, b = 7.001, c = 2.000
   a*(b*c) ≠ (a*b)*c

d. Yes it can. If both a and b are floating points, fl(a) and fl(b) (the normalized floating point numbers closest to a and b) are subject to some rounding, such that fl(a) = a (1+ ε). From this, we know that fl(a)+fl(b) /2 is not in the range of [a, b].
   This evaluates to (a(1+ ε) + b(1+ ε))/2.
   ε is the minimum value to increase 1 to the next floating point number. Since it is so negligible, it is effectively 1.
   Thus, we can evaluate it to ((a+b)/2) *(1+ ε) ≈ (a+b)/2.
   Hence, (a+b)/2 is not in the range of [a,b].

Question 3

Question 3 Code

```
function expsum
 exp_nums = [-20, -15, -5, -1, 1, 5, 15, 20];
    %Running expsum1(x)
    fprintf("expsum1(x) results:\n")
    fprintf("x \t \t accurate value \t \t approx. value \t \t abs. error \t \t \t
rel. error \n");
    for i = 1 : length(exp_nums)
        y_accurate = exp(exp_nums(i));
        y_approx = expsum1(exp_nums(i));
        abs_error = abs(y_accurate - y_approx);
```

```
        rel_error = abs(y_accurate - y_approx)/abs(y_accurate);
        fprintf("%d \t %5.12e \t %.12e \t %e \t \t %e \n", exp_nums(i), y_accurate,
y_approx, abs_error, rel_error);
    end
    fprintf("\n");
    %Running expsum2(x)
    fprintf("expsum2(x) results:\n")
    fprintf("x \t \t accurate value \t \t approx. value \t \t abs. error \t \t \t
rel. error \n");
    for i = 1 : length(exp_nums)
        y_accurate = exp(exp_nums(i));
        y_approx = expsum2(exp_nums(i));
        abs_error = abs(y_accurate - y_approx);
        rel_error = abs(y_accurate - y_approx)/abs(y_accurate);
        fprintf("%d \t %5.12e \t %.12e \t %e \t \t %e \n", exp_nums(i), y_accurate,
y_approx, abs_error, rel_error);
    end
    fprintf("\n");
    %Running expsum3(x)
    fprintf("expsum3(x) results:\n")
    fprintf("x \t \t accurate value \t \t approx. value \t \t abs. error \t \t \t
rel. error \n");
    for i = 1 : length(exp_nums)
        y_accurate = exp(exp_nums(i));
        y_approx = expsum3(exp_nums(i));
        abs_error = abs(y_accurate - y_approx);
        rel_error = abs(y_accurate - y_approx)/abs(y_accurate);
        fprintf("%d \t %5.12e \t %.12e \t %e \t \t %e \n", exp_nums(i), y_accurate,
y_approx, abs_error, rel_error);
    end
end

function s = expsum1(x)
i = 0;
sum = 0;
while 1
    y = (x.^i)/(factorial(i));
    i = i + 1;
    if sum == sum + y
        break
    end
    sum = sum + y;

    end
s = sum;
end

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

```
function s = expsum3(x)
i = 0;
sum_pos = 0;
sum_neg = 0;

if x < 0
    while 1
        y = (x.^i)/(factorial(i));
        i = i+1;
        if mod(i, 2) ~= 0 %if it's odd, and therefore negative
            if sum_neg == sum_neg + y %if the sum won't change
                break
            end
            sum_neg = sum_neg + y;
        else %if it's even
            if sum_pos == sum_pos + y %if the sum won't change
                break
            end
        sum_pos = sum_pos + y;
        end
    end

    else
        sum_pos = expsum1(x);
end
s = sum_pos + sum_neg;

end
```

Results

expsum1(x) results:

x	accurate value	approx. value	abs. error	rel. error
-20	2.061153622439e-09	4.173637499438e-09	2.112484e-09	1.024904e+00
-15	3.059023205018e-07	3.059054877425e-07	3.167241e-12	1.035376e-05
-5	6.737946999085e-03	6.737946999087e-03	1.439820e-15	2.136883e-13
-1	3.678794411714e-01	3.678794411714e-01	1.110223e-16	3.017899e-16
1	2.718281828459e+00	2.718281828459e+00	0.000000e+00	0.000000e+00
5	1.484131591026e+02	1.484131591026e+02	2.842171e-14	1.915040e-16
15	3.269017372472e+06	3.269017372472e+06	0.000000e+00	0.000000e+00
20	4.851651954098e+08	4.851651954098e+08	1.192093e-07	2.457087e-16

expsum2(x) results:

x	accurate value	approx. value	abs. error	rel. error
-20	2.061153622439e-09	2.061153622439e-09	4.135903e-25	2.006596e-16
-15	3.059023205018e-07	3.059023205018e-07	0.000000e+00	0.000000e+00
-5	6.737946999085e-03	6.737946999085e-03	1.734723e-18	2.574558e-16
-1	3.678794411714e-01	3.678794411714e-01	5.551115e-17	1.508950e-16
1	2.718281828459e+00	2.718281828459e+00	0.000000e+00	0.000000e+00
5	1.484131591026e+02	1.484131591026e+02	2.842171e-14	1.915040e-16
15	3.269017372472e+06	3.269017372472e+06	0.000000e+00	0.000000e+00
20	4.851651954098e+08	4.851651954098e+08	1.192093e-07	2.457087e-16

expsum3(x) results:

x	accurate value	approx. value	abs. error	rel. error
-20	2.061153622439e-09	5.960464477539e-08	5.754349e-08	2.791810e+01
-15	3.059023205018e-07	3.057066351175e-07	1.956854e-10	6.396989e-04
-5	6.737946999085e-03	6.737946999095e-03	9.889659e-15	1.467755e-12
-1	3.678794411714e-01	3.678794411714e-01	0.000000e+00	0.000000e+00
1	2.718281828459e+00	2.718281828459e+00	0.000000e+00	0.000000e+00
5	1.484131591026e+02	1.484131591026e+02	2.842171e-14	1.915040e-16
15	3.269017372472e+06	3.269017372472e+06	0.000000e+00	0.000000e+00
20	4.851651954098e+08	4.851651954098e+08	1.192093e-07	2.457087e-16

a. Expsum2 is the most accurate, because it avoids cancellation errors. This is because expsum2 returns $\frac{1}{\text{expsum1}(-x)}$ for all values where x is negative. This forces the values to be positive when summed, removing the cancellation errors from subtraction. Expsum3 accumulates the positive and the negative values separate from each other. When they are added together, we add two large sums of differing signs (positive and negative). This causes one very large cancellation error
b. No. When expsum3 is less than zero, it splits its sums into positive and negative sums based on the terms. As stated as above, at the end this causes a large cancellation error.

Question 4

1.

```
/nfs/u50/tranj52$ cat 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  94.05%   3.57%   2.38%   1635.772      1.636
      16   1.68  95.83%   0.00%   4.17%   1666.253      1.666
      32   3.35  95.22%   1.19%   3.58%   1661.094      1.661
      64   6.71  94.49%   1.49%   4.02%   1666.253      1.666
     128  13.42  96.05%   0.22%   3.73%   1661.094      1.661

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  94.90%   1.02%   4.08%   1426.950      1.427
      16   1.96  97.45%   0.51%   2.04%   1397.222      1.397
      32   3.91  96.68%   1.28%   2.05%   1400.870      1.401
      64   7.85  97.20%   0.51%   2.29%   1399.044      1.399
     128  15.68  96.81%   0.38%   2.81%   1408.224      1.408

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   0.67 100.00%   0.00%   0.00%    500.498      0.500
       4   1.35  97.04%   0.74%   2.22%    508.081      0.508
```

```
       8    2.70   97.41%    1.48%    1.11%     502.372       0.502
      16    5.38   98.33%    0.37%    1.30%     505.210       0.505
      32   10.77   98.05%    0.93%    1.02%     503.315       0.503

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   22.99   99.26%    0.61%    0.13%       7.303       0.007
```

2. The more you increase the precision, the slower the speed gets (we can see this in MFLOPs, or millions of floating point operations). This is because the larger the precision, the more space in memory is needed to store the added precision. With this added storage comes increased read write operations, reducing the speed of floating point operations.

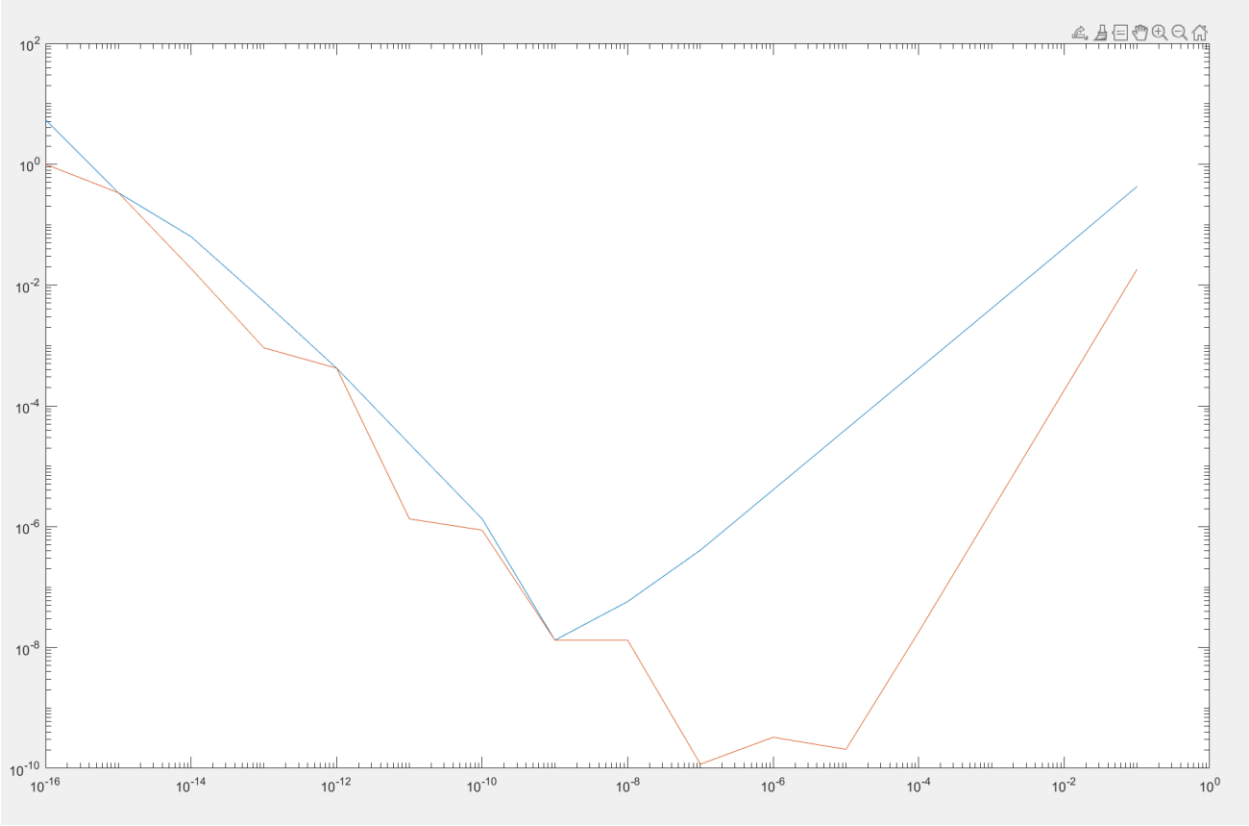
Question 5

Code:

```
function main_deriv

x = fprime(1);
i_values = []
g1_errors = []
g2_errors = []

for i = 1 : 16
    a = g1(1, 10.^-i);
    b = g2(1, 10.^-i);

    g1_error = abs(x - a);
    g2_error = abs(x - b);
    i_values(i) = 10.^-i;
    g1_errors(i) = g1_error;
    g2_errors(i) = g2_error;
    loglog(i_values, g1_errors);
    hold on;
    loglog(i_values, g2_errors);
    hold off;



end


end
```

```
function g = g1(x, h)
g = (f(x+h) - f(x))/h
end

function g = g2(x, h)
g = (f(x+h) - f(x-h))/(2*h)
end

function x = f(x)
x= x*exp(x)
end
function x = fprime(x)
x = (x+1)*exp(x);
end
```

a.



b. g1 = 10^{-9} g2= 10^{-7}

c. As we can see in the graph, there is a “sweet spot” value of h where the errors are the lowest. This is because of the relationship between truncation and cancellation errors. At smaller values of k, cancellation errors become very significant. As we tend towards larger values of h, truncation errors gain prominence until they become the dominant source of error.

Question 6

Code:

```
function main_sum
format short
[x, y] = kahan_sum();
ks_error = abs(y-x);
[a, b] = decreasing();
d_error = abs(b-a);
[c, d] = increasing();
i_error = abs(d-c);
fprintf("The error of the decreasing order is %e \n", d_error)
fprintf("The error of the increasing sum is %e \n", i_error)
fprintf("The error of the Kahan Sum is %e \n", ks_error)


end

function [apr_sum, acc_sum] = decreasing()
n = 10000;
apr_sum = 0;
acc_sum = 0;
for i = 1 : n
    apr_sum = apr_sum + 1/i;
    acc_sum = vpa(acc_sum + 1/i);
end
end

function [apr_sum, acc_sum] = increasing()
    n = 10000;
    apr_sum = 0;
    acc_sum = 0;
    for i = n : -1 : 1
        apr_sum = apr_sum + 1/i;
        acc_sum = vpa(acc_sum + 1/i);
    end
end

%Kahan
function [apr, acc] = kahan_sum()
n = 10000;
apr = 0;
acc = 0;
c_apr = 0;
c_acc = 0;
for i = 1 : n
    y_apr = 1/i - c_apr;
    y_acc = vpa(1/i - c_acc);

    t_apr = apr + y_apr;
    t_acc = vpa(acc + y_acc);

    c_apr = (t_apr - apr) - y_apr;
    c_acc = vpa((t_acc - acc) - y_acc);
```

```
    apr = t_apr;
    acc = t_acc;

end
end
```

```
The error of the decreasing order is 3.406305e-14
The error of the increasing sum is 3.730349e-14
The error of the Kahan Sum is 3.122680e-16
```

1. The decreasing order is the least accurate, compared to the increasing sum and the Kahan sum. This is because there are truncation errors when it comes to adding very small numbers to very large numbers. For decreasing order, you would have increasingly small numbers being added to a sum that is many magnitudes larger. Due to truncation errors, some of the digits added are lost. Increasing sum minimizes this problem by adding the smaller terms in the sum early on when the summation is still small. The Kahan sum has the least error because it uses a running compensator which helps recover lost digits when adding a small number to large sum.