ECE 4960 Assignment 1 Report
Yazhi Fan(yf92)
Yijia Chen(yc2366)

## I. Platform

windows 10, visual studio 2015
MacOS, code block
Language: c++

## II. File structure

The file is divided into head files and source files. Each source file represents different function and contains several modules. Header files includes: “defs.h”, “floatOperations.h” “gradualUnderflows.h” “signedZero.h”. Source file includes ”dividedByZero.cpp” “floatOperations.cpp” “gradualUnderflows.cpp” “integerOverflows” “main.cpp” “signedZero.cpp”

## III. Result

Through performing each task, we have the result as following

### 1. Integer overflows

We use Fibonacci to create integer overflows. At the exact 47th element, overflow occurs. It will return -1323752223 when IntegerOverflows() called; we use the previous element is greater than the current element to verify that it is overflow, since all the element in the fibonacci are positive.

On macOS' CodeBlock, overflow occurs at fib(47) and return value is a negative number.

### 2. Float overflows

We use Fibonacci to create integer overflows. At the exact 187th element, overflow occurs. It will return when floatOverflows() called; we use the previous element is greater than the current element to verify that it is overflow, since all the element in the fibonacci are positive.

On macOS' CodeBlock, overflow occurs at fib(1477) and return value is inf.

### 3. Integer divided by 0

On windows platform, the PC will throw an Arithmetic Exception and the program will break.

On macOS' CodeBlock, the PC will throw Floating point exception: 8 and the program will break.

### 4. Floating-point operations of INF, NINF and NaN

1) Generation and varification

We use 1/x to generate INF and NINF, log(x) to generate NaN. then verify them by calling isINF, isNINF, isNaN

Windows platform:

2) Propagation and interaction

Windows platform:

| | INF | NINF | NAN |
|---|---|---|---|
| 1/x | 0 | -0 | --- |
| sin(x) | -nan(ind) | -nan(ind) | --- |
| exp(x) | Inf | 0 | --- |
| *zero | -nan(ind) | -nan(ind) | -nan(ind) |

macOS platform:

| | INF | NINF | NAN |
|---|---|---|---|
| 1/x | 0 | -0 | --- |
| sin(x) | nan | nan | --- |
| exp(x) | Inf | 0 | --- |
| *zero | nan | nan | nan |

5. Signed Zero

Negative 0 can be generate by float 0.0, or -1.0*0 etc, which is working on both of the platform.

| | Windows | MacOS |
|---|---|---|
| log(+0) | -inf | -inf |
| log(-0.0f) | -inf | -inf |
| sin(0) / +0 | nan | nan |
| sin(-0.0f) / (-0.0f) | nan | nan |
| sin(-1.0 * 0) / abs(-1.0 * 0) | nan | nan |

6. Floating point gradual underflow

x-y:

Windows platform:(0 represents false here)

```
 1.18576e-322 - 9.88131e-323 = 1.97626e-323       x==y?0
 1.18576e-322 - 9.88131e-323 = 1.97626e-323       x==y?0
 1.18576e-322 - 9.88131e-323 = 1.97626e-323        x==y?0
 1.18576e-322 - 9.88131e-323 = 1.97626e-323       x==y?0
 1.13635e-322 - 9.88131e-323 = 1.4822e-323        x==y?0
 1.13635e-322 - 9.88131e-323 = 1.4822e-323        x==y?0
 1.13635e-322 - 9.88131e-323 = 1.4822e-323        x==y?0
 1.13635e-322 - 9.88131e-323 = 1.4822e-323        x==y?0
 1.13635e-322 - 9.88131e-323 = 1.4822e-323        x==y?0
 1.08694e-322 - 9.88131e-323 = 9.88131e-324       x==y?0
 1.08694e-322 - 9.88131e-323 = 9.88131e-324       x==y?0
 1.08694e-322 - 9.88131e-323 = 9.88131e-324       x==y?0
 1.08694e-322 - 9.88131e-323 = 9.88131e-324       x==y?0
 1.08694e-322 - 9.88131e-323 = 9.88131e-324       x==y?0
 1.03754e-322 - 9.88131e-323 = 4.94066e-324       x==y?0
 1.03754e-322 - 9.88131e-323 = 4.94066e-324       x==y?0
 1.03754e-322 - 9.88131e-323 = 4.94066e-324       x==y?0
 1.03754e-322 - 9.88131e-323 = 4.94066e-324       x==y?0
 1.03754e-322 - 9.88131e-323 = 4.94066e-324       x==y?0
```

The above operations are within 4.9407*10^-324.
The next operation shows the difference of the smallest
valid x and y above divided by two, which results in underflow.

```
 4.94066e-323 - 4.94066e-323 = 0                   x==y?0
```

x-y

macOS

```
 1.18576e-322 - 9.88131e-323 = 1.97626e-323 x==y?0
 1.18576e-322 - 9.88131e-323 = 1.97626e-323 x==y?0
 1.18576e-322 - 9.88131e-323 = 1.97626e-323 x==y?0
 1.18576e-322 - 9.88131e-323 = 1.97626e-323 x==y?0
 1.13635e-322 - 9.88131e-323 = 1.4822e-323 x==y?0
 1.13635e-322 - 9.88131e-323 = 1.4822e-323 x==y?0
 1.13635e-322 - 9.88131e-323 = 1.4822e-323 x==y?0
 1.13635e-322 - 9.88131e-323 = 1.4822e-323 x==y?0
 1.13635e-322 - 9.88131e-323 = 1.4822e-323 x==y?0
 1.08694e-322 - 9.88131e-323 = 9.88131e-324 x==y?0
 1.08694e-322 - 9.88131e-323 = 9.88131e-324 x==y?0
 1.08694e-322 - 9.88131e-323 = 9.88131e-324 x==y?0
 1.08694e-322 - 9.88131e-323 = 9.88131e-324 x==y?0
 1.08694e-322 - 9.88131e-323 = 9.88131e-324 x==y?0
 1.03754e-322 - 9.88131e-323 = 4.94066e-324 x==y?0
 1.03754e-322 - 9.88131e-323 = 4.94066e-324 x==y?0
 1.03754e-322 - 9.88131e-323 = 4.94066e-324 x==y?0
 1.03754e-322 - 9.88131e-323 = 4.94066e-324 x==y?0
 1.03754e-322 - 9.88131e-323 = 4.94066e-324 x==y?0
```

The above operations are within 4.9407*10^-324.
The next operation shows the difference of the smallest
valid x and y above divided by two, which results in underflow.

```
 4.94066e-323 - 4.94066e-323 = 0 x==y?0
```

x/y
Same result for both platforms. After constant divisions by 10 of a small number, precision was lost. After e-324, return value become zero as shown below:

```
1.23457e-308
1.23457e-309
1.23457e-310
1.23457e-311
1.23457e-312
1.23457e-313
1.23457e-314
1.23457e-315
1.23457e-316
1.23457e-317
1.23457e-318
1.23457e-319
1.23467e-320
1.23516e-321
1.23516e-322
9.88131e-324
0
0
0
```

sinc(1.23456789012345x)/x
Same result for both platforms.

```
Nan
Nan
Nan
Nan
```

Then system throw an arithmetic exception

## IV. Applications of floating-point precision

1. Description: A quad precision addition/subtraction method was implemented in file addBig.cpp. The function takes in two double precision floating point numbers and add them (subtraction can be done by simply adding a negative number). User must specify all 16 digits in their input including zeros at the end, otherwise results are not precise. The outputs are also two double precision floating point numbers whose exponents are sequential and have no overlap. For example, addBig(1.234567890123455e10, 1.7890123456789012e-5) returns 12345678901.23456 and 7.890123456789002e-06.
2. Implementation: This method is implemented by first finding the exponent and mantissa components of the two input numbers. Then an integer array of length 617 is made, so that each index corresponds to an exponent, covering range of all doubles from e308 to e-308. Mantissa components of the two inputs are added to the integer array, where each digit corresponds to an exponent. If the two numbers have overlapping exponent places, their corresponding mantissa will be added. Non-overlapping exponent places will be simply stored. In the end, all the non-zero elements in the array are added to the first double result until it reaches 16 digits. Then the rest digits are added to the second double result.
3. Exception handling: If one input is infinity and the other is larger than zero, result is infinity. If one input is negative infinity and the other is smaller than zero, result is negative infinity. If the sum of the two numbers are infinity, meaning that the first result is infinity, although second number can be stored regularly, both results will return infinity. If one is NAN, both results are NAN. Sum of negative infinity and positive infinity will be 0.
4. Special Cases: This method is valid even when the two numbers have large differences, such as adding two numbers that are many orders of magnitude apart. For example, addBig(1.234567890123456e10, 6.7890123456789012e-50) will return 12345678901.23456 and 6.789012345678901e-50, reserving all digits. The method can also be expanded easily to add more than two numbers, or exceed quad precision by outputting more double precision numbers.