Math 5610 – Assignment 2

Andrew Sheridan

September 2016

Program 1

Absolute and relative error for real and complex numbers.

```
1  // The complete code can be found at https://github.com/↩
       andrewsheridan/Math5610
2
3  // Complex.h
4  class Complex
5  {
6  public:
7     double real;
8     double imaginary;
9
10    Complex(double newReal, double newImaginary)
11    {
12       real = newReal;
13       imaginary = newImaginary;
14    }
15
16    Complex operator-(Complex c) {
17       double realResult = real - c.real;
18       double imaginaryResult = imaginary - c.imaginary;
19       Complex result(realResult, imaginaryResult);
20       return result;
21    }
22
23    Complex operator/(Complex c) {
24       double realNumerator = real*c.real - imaginary*c.imaginary;
25       double imaginaryNumerator = imaginary*c.real - real*c.↩
             imaginary;
26       double denominator = (c.real*c.real) - (c.imaginary*c.↩
             imaginary);
27
28       return Complex(realNumerator / denominator, ↩
             imaginaryNumerator / denominator);
29    }
30 };
31
32 double complexAbsolute(Complex a) {
33    return std::sqrt(std::pow(a.real, 2) + std::pow(a.imaginary, ↩
          2));
```

```
}

std::cout << "Result: " << epsilon << std::endl;
```

```
// Error.h
double realAbsolute(double number, double approximation) {
    return std::abs(number - approximation);
}

double imaginaryAbsolute(Complex complex, Complex ↩
    complexApproximation) {
    return complexAbsolute(complex - complexApproximation);
}

double realRelative(double number, double approximation) {
    return std::abs(number - approximation) / std::abs(number);
}

double imaginaryRelative(Complex a, Complex b) {
    return complexAbsolute(a - b) / complexAbsolute(a);
}
```

```

int main(void) {
    testRealAbsolute(1.0, 0.99); //Outputs the result to the ↩
        console
    testRealAbsolute(1.0, 1.01);
    testRealAbsolute(100.0, 100.3);
    testRealAbsolute(-1.5, -1.2);
    testRealAbsolute(100.0, 99.99);
    testRealAbsolute(100.0, 99);

    testRealRelative(1.0, 0.99);
    testRealRelative(1.0, 1.01);
    testRealRelative(100.0, 100.3);
    testRealRelative(-1.5, -1.2);
    testRealRelative(100.0, 99.99);
    testRealRelative(100.0, 99);

    testImaginaryAbsolute(3.0, 4.0, 3.0, 4.1);
    testImaginaryAbsolute(3.0, 4.0, 3.3, 4.0);
    testImaginaryAbsolute(3.0, 4.0, 3.2, 4.3);

    testImaginaryRelative(3.0, 4.0, 3.0, 4.1);
    testImaginaryRelative(3.0, 4.0, 3.3, 4.0);
    testImaginaryRelative(3.0, 4.0, 3.2, 4.3);

    return 0;
}
```

```
|1 - 0.99|
Result: 0.01
```

```
|1 - 1.01|
Result: 0.01
|100 - 100.3|
Result: 0.3
|-1.5 - -1.2|
Result: 0.3
|100 - 99.99|
Result: 0.01
|100 - 99|
Result: 1
|1 - 0.99| / |1|
Result: 0.01
|1 - 1.01| / |1|
Result: 0.01
|100 - 100.3| / |100|
Result: 0.003
|-1.5 - -1.2| / |-1.5|
Result: 0.2
|100 - 99.99| / |100|
Result: 0.0001
|100 - 99| / |100|
Result: 0.01
|(3+4i) - (3+4.1i)|
Result: 0.1
|(3+4i) - (3.3+4i)|
Result: 0.3
|(3+4i) - (3.2+4.3i)|
Result: 0.360555
|(3+4i) - (3+4.1i)| / |3+4i|
Result: 0.02
|(3+4i) - (3.3+4i)| / |3+4i|
Result: 0.06
|(3+4i) - (3.2+4.3i)| / |3+4i|
Result: 0.072111
```

Program 2

Computing Norms

```
1 //Norm.h
2 double euclideanLength(std::vector<double> list) {
3    double sum = 0;
4    for (unsigned i = 0; i < list.size(); i++) {
5       sum += (list[i] * list[i]);
6    }
7    return std::sqrt(sum);
```

```
}

double manhattanNorm(std::vector<double> list) {
   double sum = 0;
   for (unsigned i = 0; i < list.size(); i++) {
      sum += std::abs(list[i]);
   }
   return sum;
}

double infinityNorm(std::vector<double> list) {
   double max = 0;
   for (unsigned i = 0; i < list.size(); i++) {
      if (std::abs(list[i]) > max) {
         max = std::abs(list[i]);
      }
   }
   return max;
}
```

```
//Main.cpp
int main(void) {
   std::cout << "Testing Euclidean Length" << std::endl;
   std::vector<double> test1;
   test1.push_back(3);
   test1.push_back(4);
   testEuclideanLength(test1);

   std::cout << "Testing Euclidean Length" << std::endl;
   std::vector<double> test2;
   test2.push_back(3.5);
   test2.push_back(4.1);
   test2.push_back(8.4);
   testEuclideanLength(test2);

   std::cout << "Testing manhattan norm" << std::endl;
   std::vector<double> test3;
   test3.push_back(1.1);
   test3.push_back(2.2);
   test3.push_back(-3.3);
   test3.push_back(4.4);
   test3.push_back(-5.5);
   testManhattanNorm(test3);

   std::cout << "Testing Infinity Norm" << std::endl;
   std::vector<double> test4;
   test4.push_back(1.1);
   test4.push_back(2.2);
   test4.push_back(-3.3);
   test4.push_back(4.4);
   test4.push_back(-5.5);
   testInfinityNorm(test4);

   return 0;
}
```

```
Testing Euclidean Length
Numbers in list:
3
4
Result: 5

Testing Euclidean Length
Numbers in list:
3.5
4.1
8.4
Result: 9.98098

Testing manhattan norm
Numbers in list:
1.1
2.2
-3.3
4.4
-5.5
Result: 16.5

Testing Infinity Norm
Numbers in list:
1.1
2.2
-3.3
4.4
-5.5
Result: 5.5
```

Program 3

Dot and Cross Products.

```
//Product.h
double dotProduct(std::vector<double> one, std::vector<double> two) {
   if (one.size() != two.size()) {
      std::cout << "These vectors are not the same size." << std::endl;
      return -1;
   }

   double sum = 0;

   for (int i = 0; i < one.size(); i++) {
```

```
11          sum += one[i] * two[i];
12      }
13
14      return sum;
15  }
16
17  void testDotProduct() {
18      std::vector<double> one = getVectorInput();
19      std::vector<double> two = getVectorInput();
20
21      double result = dotProduct(one, two);
22
23      std::cout << "Result: " << result << std::endl;
24  }
25
26  void testDotProduct(std::vector<double> one, std::vector<double> ↩
        two) {
27      std::cout << "Numbers in list one: " << std::endl;
28      for (int i = 0; i < one.size(); i++) {
29          std::cout << one[i] << std::endl;
30      }
31      std::cout << "Numbers in list two: " << std::endl;
32      for (int i = 0; i < two.size(); i++) {
33          std::cout << two[i] << std::endl;
34      }
35      double result = dotProduct(one, two);
36      std::cout << "Result: " << result << std::endl;
37  }
38
39  std::vector<double> crossProduct(std::vector<double> one, std::↩
        vector<double> two) {
40      std::vector<double> result;
41      if (one.size() != two.size()) {
42          std::cout << "These vectors are not the same size." << std↩
                ::endl;
43          return result;
44      }
45      if (one.size() != 3) {
46          std::cout << "These vectors are not the correct size." << ↩
                std::endl;
47          return result;
48      }
49
50      result.push_back(one[1]*two[2] - one[2]*two[1]);
51      result.push_back(one[2] * two[0] - one[0] * two[2]);
52      result.push_back(one[0] * two[1] - one[1] * two[0]);
53      return result;
54  }
55
56  void testCrossProduct() {
57      std::vector<double> one = getVectorInput(3);
58      std::vector<double> two = getVectorInput(3);
59
60      std::vector<double> result = crossProduct(one, two);
61
62      std::cout << "Result: " << std::endl;
63      std::cout << "<" << result[0] << ", " << result[1] << ", " << ↩
```

```
        result[2] << ">" << std::endl;
}

void testCrossProduct(std::vector<double> one, std::vector<double> two) {
    std::cout << "Numbers in list one: " << std::endl;
    for (int i = 0; i < one.size(); i++) {
        std::cout << one[i] << std::endl;
    }
    std::cout << "Numbers in list two: " << std::endl;
    for (int i = 0; i < two.size(); i++) {
        std::cout << two[i] << std::endl;
    }
    std::vector<double> result = crossProduct(one, two);
    std::cout << "Result: " << std::endl;
    for (int i = 0; i < result.size(); i++) {
        std::cout << result[i] << std::endl;
    }
}
```

```
//Main.cpp
std::cout << "Testing Dot Product" << std::endl;
std::vector<double> test1;
test1.push_back(3);
test1.push_back(4);
std::vector<double> test2;
test2.push_back(10);
test2.push_back(15);
testDotProduct(test1, test2);

std::cout << "Testing Dot Product" << std::endl;
std::vector<double> test3;
test3.push_back(3);
test3.push_back(4);
test3.push_back(5);
std::vector<double> test4;
test4.push_back(0.5);
test4.push_back(0.4);
test3.push_back(0.7);
testDotProduct(test3, test4);

std::cout << "Testing Cross Product" << std::endl;
std::vector<double> test5;
std::vector<double> test6;
test5.push_back(2);
test5.push_back(3);
test5.push_back(4);
test6.push_back(6);
test6.push_back(5);
test6.push_back(1);
testCrossProduct(test5, test6);

std::cout << "Testing Cross Product" << std::endl;
std::vector<double> test7;
std::vector<double> test8;
test7.push_back(1.5);
```

```
37 test7.push_back(2.3);
38 test7.push_back(4.1);
39 test8.push_back(8.1);
40 test8.push_back(2.4);
41 test8.push_back(1);
42 testCrossProduct(test7, test8);
```

```
Testing Dot Product
Numbers in list one:
3
4
Numbers in list two:
10
15
Result: 90
Testing Dot Product
Numbers in list one:
3
4
5
0.7
Numbers in list two:
0.5
0.4
These vectors are not the same size.
Result: -1
Testing Cross Product
Numbers in list one:
2
3
4
Numbers in list two:
6
5
1
Result:
-17
22
-8
Testing Cross Product
Numbers in list one:
1.5
2.3
4.1
Numbers in list two:
8.1
```

```
2.4
1
Result:
-7.54
31.71
-15.03
```

Problem 1

The fraction in a single precision word has 23 bits (alas, less *than half the length of the double precision word). Show that the corresponding rounding unit is approximately* 6×10^{-8}.

```
1 float epsilon = 1;
2 while (1 + epsilon != 1) {
3    epsilon /= 2;
4 }
5
6 std::cout << "Result: " << epsilon << std::endl;
```

```
Result: 5.96046e-08
```

Problem 2

The function $f_1(x_0, h) = sin(x_0 + h) - sin(x_0)$ *can be transformed into another form,* $f_2(x_0, h)$*, using the trigonometric formula*

$$sin(\phi) - sin(\psi) = 2cos(\frac{\phi + \psi}{2})sin(\frac{\phi - \psi}{2})$$

Thus, f_1 *and* f_2 *have the same values, in exact arithmetic, for any given argument values* x_0 *and* h.

A.

Derive $f_2(x_0, h)$.
If we set $\phi = x_0 + h$ and $\psi = x_0$, we can then plug these values into the given equation and get $2\cos(\frac{x_0+h+x_0}{2})\sin(\frac{x_0+h-x_0}{2})$. Simplified this is $2\cos(x_0 + \frac{h}{2})\sin(\frac{h}{2})$

B.

Suggest a formula that avoids cancellation errors for computing the approximation $(f(x_0 + h) - f(x_0))/h$ *to the derivative of* $f(x) = sin(x)$ *at* $x = x_0$. *Write a program that implements your formula and computes an approximation of* $f'(1.2)$*, for* $h - 1e^{-20}, 1e{-}19, ..., 1$.
Not completed.

C.

Explain the difference in accuracy between your results and the results reported in Example 1.3.
Not completed.

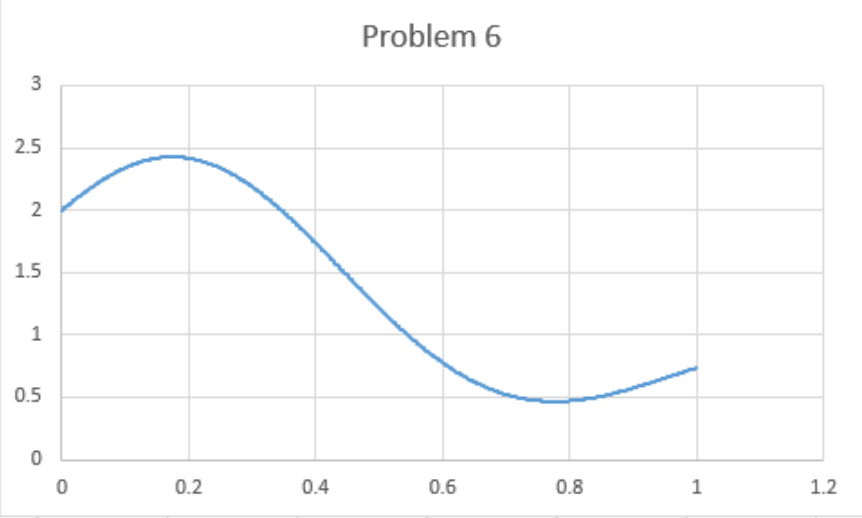
Problem 6

Write a program that receives as input a number x and a parameter n and returns x rounded to n decimal digits. Write your program so that it can handle an array as input, returning an array of the same size in this case.

```
#include <iostream>
#include <vector>
#include <cmath>
#include "VectorInput.h"

double doRound(double x, int n) {
   double result = x * pow(10, n);
   result += 0.5;
   result = std::floor(result);
   result *= pow(10, -n);
   return result;
}

std::vector<double> myRound(std::vector<double> list, int n) {
   std::vector<double> result;
   for (int i = 0; i < list.size(); i++) {
      double aThing = doRound(list[i], n);
      result.push_back(aThing);
   }
   return result;
}

int main(void) {
   int n = 0;
   while (true) {
      std::vector<double> list = getVectorInput();
      std::cout << "Please input an integer for your exponent: ";
      std::cin >> n;

      std::vector<double> result = myRound(list, n);
      for (int i = 0; i < result.size(); i++) {
         std::cout << list[i] << ": " << result[i] << std::endl;
      }
   }

   return 0;
}
```

Use your program to generate numbers for Example 2.2, demonstrating the phenomenon depicted there without use of single precision.
Using the same formula and step of 0.002, we get the graph shown below by using

double precision. I have the code which was asked for in the first portion of the problem, but the way I'm supposed to use it in relation to example 2.2 is unclear.



Problem 11

(a) Show that $\ln(x - \sqrt{x^2 - 1}) = -\ln(x + \sqrt{x^2 - 1})$

We can rewrite the equation $-\ln(x + \sqrt{x^2 - 1})$ as $\ln(\frac{1}{x+\sqrt{x^2-1}})$. If we multiply by the conjugate within the fraction, we get $\ln(\frac{1}{x+\sqrt{x^2-1}} \frac{x-\sqrt{x^2-1}}{x-\sqrt{x^2-1}})$. This becomes $\ln(\frac{x-\sqrt{x^2+1}}{x^2-x^2+1})$, which simplifies to $\ln(x - \sqrt{x^2 + 1})$, which is our desired result.

(b) Which of the two formulas is more suitable for numerical computation? Explain why, and provide a numerical example in which the difference in accuracy is evident.

I don't know which of the two formulas is better for numerical computation.

Problem 13

Did not understand the question.

Problem 14

Consider the approximation to the first derivative

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

The truncation *(or* discretization*) error for this formula is $O(h)$. Suppose that the absolute error in evaluating the function f is bounded by ϵ and let us ignore the errors generated in basic arithmetic operations.*

(a) Show that the total computational error (truncation and rounding combined) is bounded by $\frac{Mh}{2} + \frac{2\epsilon}{h}$, where M is a bound on $|f''(x)|$.

We can show that $\frac{Mh}{2}$ is the truncation error by examining the formula in Example 1.2.

$$f'(x_0) = \frac{f(x_0+h) - f(x_0)}{h} - (\frac{h}{2}f''(x_0) + \frac{h^2}{6}f'''(x_0) + ...)$$

We are told that the error for this formula is order h, therefore we can end the error portion of the above formula at $(\frac{h}{2}f''(x_0))$. We are also told that M is a bound on $|f''(x)|$, so we can rewrite the error as $\frac{Mh}{2}$.

Because the error is order h, the rounding error on each $f(x)$ can be represented by $\frac{\epsilon}{h}$. There are two functions $f(x)$ and $f(x+h)$, therefore the rounding error is $\frac{2\epsilon}{h}$.

(b) What is the value of h for which the above bound is minimized? We can find the minimization of h by taking the derivative of our error representation. If we set $g(h) = \frac{Mh}{2} + \frac{2\epsilon}{h}$, we can then find $g'(h)$ which is $\frac{M}{2} + \frac{-2\epsilon}{h^2}$. If we then set $g'(h) = 0$, we can find the maxes or mins of $g(h)$. We simplify and find that $h = \pm 2\sqrt{\frac{\epsilon}{M}}$ The second derivative tells us the positive value of h is our min.

(c) The rounding unit we employ is approximately equal to 10^{-16}. Use this to explain the behavior of the graph in Example 1.3. Make sure to explain the shape of the graph as well as the value where the apparent minimum is attained.

The error in the graph in example 1.3 starts increasing in magnitude a great deal once h hits about 10^-8. The is due to the roundoff error. Once h hits 10^{-16} the error stops increasing because the rounding unit is also approximately 10^-16. The apparent minimum is around 10^-8, before the roundoff error kicks in.

(d) It is not difficult to show, using Taylor expansions, that $f'(x)$ can be approximated more accurately (in terms of truncation error) by

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

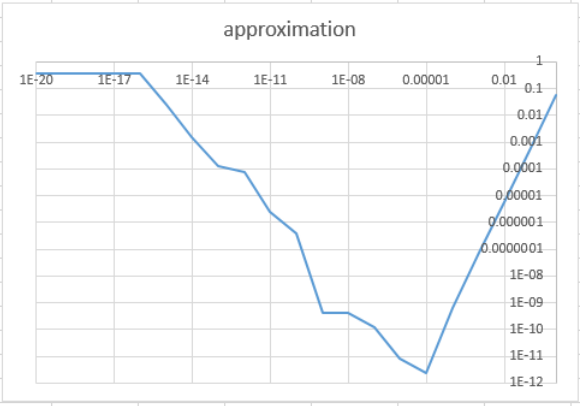
For this approximation, the truncation error is Oh^2. Generate a graph similar to Figure 1.3 (please generate only the solid line) for the same function and the same value of x, namely, for $sin(1.2)$, and compare the two graphs. Explain the meaning of your results.

```
1 double findApproximation(double x, double h) {
2     return std::abs(cos(x) - (sin(x + h) - sin(x - h)) / (2 * h));
```

```
}

int main(void) {
   std::ofstream myfile;
   myfile.open("problem14.txt");
   double stoppingPoint = pow(10, -20);

   myfile << "h \t" << "approximation \t" << std::endl;
   for (double h = 1.0; h > stoppingPoint; h *= 0.1) {
      double approximation = findApproximation(1.2, h);

      std::cout << h << "\t" << approximation << std::endl;
      myfile << h << "\t" << approximation << std::endl;
   }
   myfile.close();

   return 0;
}
```

I used the little program above to output some data points, which I then imported to Excel. My graph has a very similar shape to the one shown in figure 1.3. Mine does not have the dips that the graph in 1.3 shows, however.

h	approximation
1	0.057444
0.1	0.000604
0.01	6.04E-06
0.001	6.04E-08
0.0001	6.04E-10
1.00E-05	2.43E-12
1.00E-06	7.98E-12
1.00E-07	1.19E-10
1.00E-08	4.36E-10
1.00E-09	4.36E-10
1.00E-10	3.88E-07
1.00E-11	2.39E-06
1.00E-12	7.46E-05
1.00E-13	0.00013
1.00E-14	0.001535
1.00E-15	0.02622
1.00E-16	0.362358
1.00E-17	0.362358
1.00E-18	0.362358
1.00E-19	0.362358
1.00E-20	0.362358



Problem 15

Suppose a machine with a floating point system $(\beta, t, L, U) = (10, 8, -50, 50)$ *is used to calculate the roots of the quadratic equation*

$$ax^2 + bx + c = 0,$$

where a*,* b*, and* c *are given, real coefficients. For each of the following, state the numerical difficulties that arise if one uses the standard formula for computing*

the roots. Explain how to overcome these difficulties (when possible).

(a) $a = 1; b = -10^5; c = 1$

In this scenario, if we plug the values into the quadratic formula, the problem we encounter is that we have an imaginary number in the solution. To resolve this, we would need to include support for complex numbers in our program.

(b) $a = 6 \times 10^{30}; b = 5 \times 10^{30}; c = -4 \times 10^{30}$

The problem with this scenario is that the exponent is too large for our system to be able to represent it. A way we can work around this is scale all the exponents down at the beginning of the execution because we can scale a, b, and c by the same amount. Then, after the computation is done, we can scale the solution back up.

(c) $a = 10^{-30}; b = -10^5; c = 10^{30}$

This time we come across the issue of accuracy in our precision. Because the values of b and c have very large exponents, and a has a small exponent, when we subtract 1 from -10^{30^2} we will have a long string of nines, most of which will be truncated. To fix this, we'd have to add extra precision.