HW3 – Math5610

Andrew Sheridan

September 2016

Problem 1.

Implement a code that will find roots of a function of one variable using the Bisection method. Make sure your code tests the input to the algorithm for errors that might occur.

I used the Matlab code given in Section 3.2, and translated to C++. I then used the functions and input from Examples 3.3 to assure that my function worked properly. The results are shown as console output at the end of the code below.

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++

//Bisection.h
double bisection(double (*f)(double), double a, double b, double atol) {
  if (a > b) { // If a is greater than b, switch them.
    double temp = b;
    b = a;
    a = temp;
  }
  double fa = f(a);
  double fb = f(b);
  if (a >= b || (fa * fb >= 0) || atol <= 0) // Validate input
  {
    std::cout << "This input is not valid." << std::endl;
    return 0;
  }

  int n = ceil(log2(b - a) - log2(2 * atol)); //Compute number of iterations.

  for (int k = 0; k < n; k++) { //Iterate n times, each iteration bisecting once.
    double p = (a + b) / 2;
    double fp = f(p);
    if (fa * fp < 0) {
      b = p;
```

```
      fb = fp;
    }
    else {
      a = p;
      fa = fp;
    }
  }
  return (a + b) / 2;
}
```

```
//main.cpp
#include "Bisection.h"
#include <cmath>
#include <iostream>

//First function from example 3.3
double func(double x) {
  return pow(x, 3) - (30 * x * x) + 2552;
}

//Second function from example 3.3
double func2(double x) {
  return 2.5*sinh(x / 4) - 1;
}

int main() {
  cout << "Testing our bisect method with formulas in Example 3.3." <<
       std::endl;
  double result = bisection(func, 0.0, 20.0, 1 * pow(10, -8));
  cout << "Result 1: " << result << std::endl;
  double result2 = bisection(func2, -10.0, 10.0, 1 * pow(10, -10));
  cout << "Result 2: " << result2 << std::endl;
  cin >> result;

  return 0;
}
```

```
Testing our bisect method with formulas in Example 3.3.
Result 1: 11.8615
Result 2: 1.56014
```

Problem 2.

Implement a code that will find roots of a function of one variable using Functional Iteration. Make sure that the code checks for any possible errors in the input to the algorithm.

Below is my implementation of Function Iteration, run with two functions found in an example at the site linked within the code.

```
//Andrew Sheridan
//Math 5610
//Written in C++
//Fixed_Point.h
#include <cmath>
#include <iostream>

double fixed_point(double(*g)(double), double x0, unsigned max_iterations, double atol) {

  // Check input. Other inputs will work properly because of their types.
  if (atol < 0) {
    std::cout << "The tolerance must be positive." << std::endl;
  }

  double x_i = 99999999;//x sub i
  double x_ip1 = x0; //x sub i+1

  // computes x of i+1 until the max interations are reached or we are within the tolerance.
  for (int i = 0; i < max_iterations || std::abs(x_ip1 - g(x_i)) > atol; i++) {
    x_i = x_ip1;
    x_ip1 = g(x_i);
  }
  return x_ip1;
}
```

```
//Main.cpp
#include "Fixed_Point.h"
#include <iostream>
#include <cmath>

//Test functions found at https://mat.iitm.ac.in/home/sryedida/public_html/caimna/transcendental/iteration%20methods/fixed-point/iteration.html

double g(double x) { //First test function.
  return std::pow(x + 10, 0.25);
}


double g2(double x) { //Second test function.
```

```cpp
  return std::pow(x + 10, 0.5) / x;
}

int main(void) {
  double result = fixed_point(g, 1.0, 10, std::pow(10, -8));
  std::cout << "Result: " << result << std::endl;
  double result2 = fixed_point(g2, 1.8, 100, std::pow(10, -8));
  std::cout << "Result2: " << result2 << std::endl;

  std::cin >> result;
  return 0;
}
```

Problem 3.

Implement a code that will find roots of a function of one variable using Newton's method. Make sure that the code checks for any possible errors in the input to the algorithm.

Included with my implementation are two test cases. One is a simple case, the other is Example 3.7 found from the book.

```
//Andrew Sheridan
//Math 5610
//Written in C++
//Newton_Method.h
#pragma once
#include <cmath>

double newton(double (*f)(double), double (*df)(double), double x0, double tol, unsigned max_iterations) {
  double xk = 99999999; //x sub k
  double xkp1 = x0; //x sub k+1
  //Loops until we reach the maximum iterations or we are within the input tolerance.
  for (unsigned k = 0; k < max_iterations && std::abs(xkp1 - xk) > tol; k++) {
    xk = xkp1;
    xkp1 = xk - (f(xk) / df(xk));
  }

  return xkp1;
}
```

```
//Main.cpp
#include <iostream>
#include "Newton_Method.h"
#include <cmath>

//A test function, f(x)
double f(double x) {
  return std::pow(x, 2) - 2;
}

// A test function, f'(x), to be paired with f(x)
double df(double x) {
  return 2 * x;
}

// Functions from Example 3.7
double g(double x) {
  return 2 * cosh(x / 4) - x;
}

double dg(double x) {
  return 0.5 * sinh(x / 4) - 1;
```

```
}

int main(void) {
  double result = newton(f, df, 1, 0.0000001, 10);
  std::cout << "result: " << result << std::endl;

  double result2 = newton(g, dg, 2, pow(10, -8), 10);
  std::cout << "result2: " << result2 << std::endl;

  return 0;
}
```

```
result: 1.41421
result2: 2.35755
```

Problem 4.

Implement a code that will find roots of a function of one variable using the Secant method. Make sure that the code checks for any possible errors in the input to the algorithm.

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++
#include<cmath>

// Secant_Method.h
double secant_method(double (*f)(double), double x0, double x1, double tol, unsigned max_iterations) {
  double xkm1; //x sub k-1
  double xk = x0; //x sub k
  double xkp1 = x1; //x sub k+1

  //Loops until we reach the maximum iterations or we are within the input tolerance.
  for (unsigned k = 0; k < max_iterations && std::abs(xkp1 - xk) > tol; k++) {
    xkm1 = xk;
    xk = xkp1;
    xkp1 = xk - (f(xk) * (xk - xkm1)) / (f(xk) - f(xkm1));
  }

  return xkp1;
}
```

```cpp
// Main.cpp
#include <iostream>
#include "Secant_Method.h"
#include <cmath>

//A test function, f(x), as given in Example 3.8
double f(double x) {
  return 2 * std::cosh(x/4) - x;
}

int main(void) {
  double result = secant_method(f, 2, 1, std::pow(10, -8), 10);
  std::cout << "result: " << result << std::endl;
  return 0;
}
```

```
result: 2.35755
```

Problem 5.

Implement a code that will find roots of a function of one variable using hybrid method that starts using the Bisection method and then switches to Newton's method. Make sure that the code checks for any possible errors in the input to the algorithm.

This program has a dependency on *Newton_Method.h*, which is found in Problem 3. This method takes as input parameters a function F(x), a function F'(x), an initial guess of a and b, the number of iterations of bisection we take before attempting Newton's Method, the maximum number of total iterations, and a tolerance.

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++
#pragma once
//Hybrid_Method.h
#include <math.h>
#include <iostream>
#include "Newton_Method.h"

double hybrid(double(*f)(double), double(*df)(double), double a, double b, unsigned int bisection_iterations, unsigned int max_iterations, double tol) {

  double fa = f(a);
  double fb = f(b);
  // Validate input
  if (a >= b || (fa * fb >= 0) || tol <= 0 || bisection_iterations < 1 || max_iterations < 1)
  {
    std::cout << "This input is not valid." << std::endl;
    return 0;
  }

  unsigned int totalIterationCount = 0;
  unsigned int bisectionIterationCount = 0;
  bool useNewton = false;
  double p;
  while (totalIterationCount < max_iterations && !useNewton && (b-a) > tol) {
    p = (a + b) / 2;
    double fp = f(p);
    if (fa * fp < 0) {
      b = p;
      fb = fp;
    }
    else {
      a = p;
      fa = fp;
    }
```

```cpp

    bisectionIterationCount++;
    totalIterationCount++;

    // If we've done the set number of bisections, try newton's method
    if(bisectionIterationCount == bisection_iterations){
      double newtonResult = newton(f, df, p, tol, 1);
      // If newton's method was more efficient than bisection, start using newton's method.
      if (std::abs(newtonResult - p) < std::abs(b - a)){
        useNewton = true;
        p = newtonResult;
        totalIterationCount++;
      }
      //Else, start bisection again.
      else {
        bisectionIterationCount = 0;
      }
    }
  }
  // If we have begun using newton's method, iterate through newton's method until we've reached the max number of iterations or tolerance is met.
  if (useNewton) {
    double previous = 999999;
    while (totalIterationCount < max_iterations && std::abs(p - previous) > tol) {
      previous = p;
      p = newton(f, df, p, tol, max_iterations - totalIterationCount);
      totalIterationCount++;
    }
  }

  return p;
}
```

```cpp
//Main.cpp
#include <iostream>
#include "Hybrid_Method.h"
#include <cmath>

//A test function, f(x)
double f(double x) {
  return std::pow(x, 2) - 2;
}

// A test function, f'(x), to be paired with f(x)
double df(double x) {
  return 2 * x;
}

int main(void) {
  double result = hybrid(f, df, 1, 10, 2, 10, std::pow(10, -8));
  std::cout << "result: " << result << std::endl;
  return 0;
}
```

Problem 6.

Complete Problem 1. at the end of Chapter 3 in the textbook.

(a) The function is $f(x) = \sqrt{x} - 1.1$, and I applied the bisection method on the interval $[0, 2]$ with the tolerance of 1.e-8. I used the Bisection code found in Problem 1 of this assignment, and the rest of the code is below.

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++
double problem6(double x) {
  return std::sqrt(x) - 1.1;
}

int main(void){
    double result3 = bisection(problem6, 0, 2, 1 * std::pow(10, -8));
    std::cout << "Result 3: " << result3 << std::endl;
    return 0;
}
```

```
Iterations: 27
Result 3: 1.21
```

Page 43 of the textbook gives a formula for finding the number of required iterations required for the bisection method.

$$n = \lceil \log_2 \frac{b-a}{2atol} \rceil$$

With the input values of $b = 2$, $a = 0$, and $atol = 1.e - 8$, we get the result of $n = 27$. Because this formula is hard coded into my function, as it is in the textbook, this is the result we obtain.

(b) The actual value of the root is 1.21, and the result is 1.2100000008940697, so our absolute error is 8.940687×10^{-10}. The formula for finding this difference is $|x^* - x_n| \leq \frac{b-a}{2} \times 2^{-n}$, which comes out to 2^{-27}, or about 7.45×10^{-9}. Our absolute error conforms with this result.

Problem 7.

Complete Problem 2. at the end of Chapter 3 in the textbook.

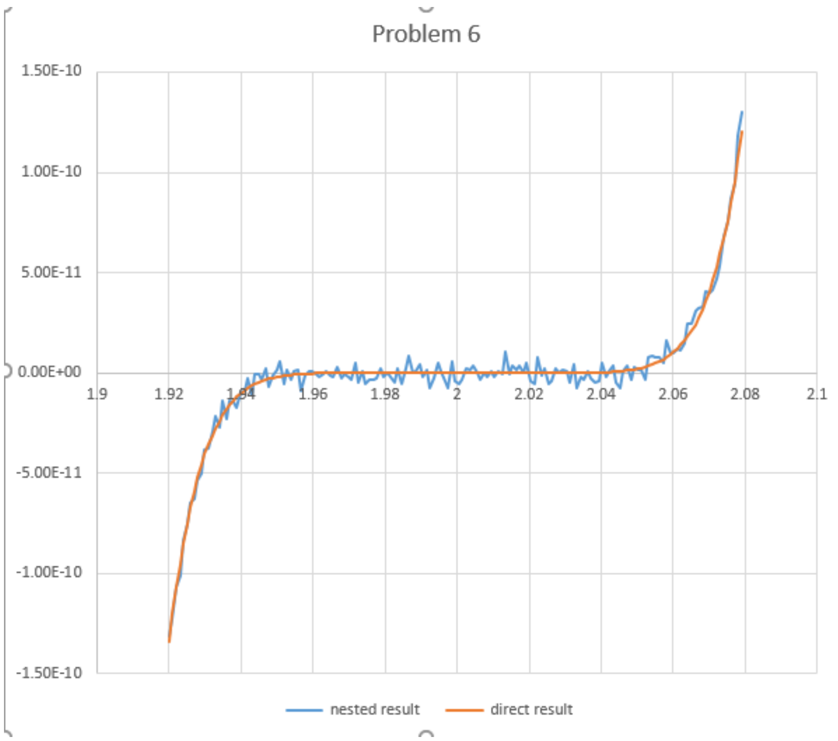
(a) I implemented both part *i* and *ii* in the same program, and outputted my results to a text file, which was then imported to Excel.

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++
//Nested.h
#pragma once
#include <cmath>

double nested(double data[], unsigned length, double x) {
  double p = data[0];
  for (int i = 1; i < length; i++) {
    p = p*x + data[i];
  }
  return p;
}
```

```cpp
//main.cpp
#include <iostream>
#include <fstream>
#include "Nested.h"

int main() {
  std::ofstream file;
  file.open("output.txt");
  double list[10] = { 1, -18, 144, -672, 2016, -4032, 5376, -4608, 2304, -512 };
  double step = (2.08 - 1.92) / 161;
  double nestedResults[161];
  unsigned i = 0;
  file << "x \t nested result \t direct result" << std::endl;
  for (double x = 1.92; i < 161; x += step, i++) {
    double nestedResult = nested(list, 10, x);
    nestedResults[i] = nestedResult;

    double directResult = std::pow(x - 2, 9);

    file << x << "\t" << nestedResult << "\t"  << directResult << std::endl;
  }
  file.close();
  return 0;
}
```

(b) The direct results make a smooth curve, as they should. The nested results are less smooth, but still follow the curve of the real results pretty well.

(c) The function will find a root which satisfies $|p-2| \leq 10^-6$. The error of each iteration is always smaller than 1e-12, and even after iterating 161 times, the total error will not be greater than 10^-6.



Problem 8.

Complete Problem 4. at the end of Chapter 3 in the textbook.

(a) The two fixed points of the function $g(x) = x^2 + \frac{3}{16}$ are $\frac{1}{4}$ and $\frac{3}{4}$.

(b) The fixed point $\frac{1}{4}$ will converge, but $\frac{3}{4}$ will not. We know this because of the fixed point theorem found on page 47:

$$|g'(x)| \leq \rho$$

For our point $\frac{1}{4}$, the result of this formula is $\frac{11}{16}$, which is less than 1, so we know it will converge, but the rate of convergence is poor. Bisection would be better, and it's more robust. For the point $\frac{3}{4}$ the result is $\frac{27}{16}$, which is greater than one and it doesn't converge.

(c) It will take the fixed point $\frac{1}{4}$ approximately seven iterations to reduce the convergence error by a factor of 10, because six iterations $\frac{11}{16}^6$ only equals 0.10559, while seven gives $\frac{11}{16}^7 = 0.07259$.

Problem 9.

Complete Problem 7. at the end of Chapter 3 in the textbook. Our objective is to show that Steffensen's method

$$x_{k+1} = x_k - \frac{f(x_k)}{g(x_k)}$$

converges quadratically. We are given that $g(x_k) = \frac{f(x+f(x))-f(x)}{f(x)}$. If we first clear the fractions in Steffensen's method, we have

$$g(x_k)(x_{k+1} - x_k) = -f(x_k)$$

If we then write out the formula for $g(x)$ we have

$$(f(x_k + f(x_k)) - f(x_k))(x_{k+1} - x_k) = -f(x_k)^2$$

We can then find the taylor series expansion for the term $f(x + f(x))$.

$$f(x + f(x)) = f(x*) + f'(x*)(f(x) + x - x*) + \frac{1}{2}f''(\xi)(f(x) + x - x*)^2$$

The term $f(x*)$ cancels out because it equals zero. If we plug this into our formula where we left it, we get

$$[f'(x*)(f(x_k)+x_k-x*)+\frac{1}{2}f''(\xi_1)(f(x_k)+x_k-x*)^2-f(x_k)](x_{k+1}-x_k) = -f(x_k)^2$$

We can now do another taylor series expansion for the term $f(x_k)$.

$$f(x_k) = f(x*) + f'(x*)(x_k - x*) + \frac{1}{2}f''(\xi_2)(x_k - x*)^2$$

Once again, $f(x*) = 0$, which leaves us with

$$f(x_k) = f'(x*)(x_k - x*) + \frac{1}{2}f''(\xi_2)(x_k - x*)^2$$

We can substitute this into the main formula, which gives

$$\begin{aligned}
&[f'(x*)(f'(x*)(x_k - x*) + \frac{1}{2}f''(\xi_2)(x_k - x*)^2 + x_k - x*) \\
&+\frac{1}{2}f''(\xi_1)(f'(x*)(x_k - x*) + \frac{1}{2}f''(\xi_2)(x_k - x*)^2 + x_k - x*)^2 \\
&-f(x*) + f'(x*)(x_k - x*) + \frac{1}{2}f''(\xi_2)(x_k - x*)^2](x_{k+1} - x_k) \\
&= -(f'(x*)(x_k - x*) + \frac{1}{2}f''(\xi_2)(x_k - x*)^2)^2
\end{aligned}$$

We can then begin substituting the x_k and $x*$ terms for error terms as follows.

$$
\begin{aligned}
&[f'(x*)(f'(x*)(-e_k) + \frac{1}{2}f''(\xi_2)(-e_k)^2 - e_k) \\
&+\frac{1}{2}f''(\xi_1)(f'(x*)(-e_k) + \frac{1}{2}f''(\xi_2)(-e_k)^2 - e_k)^2 \\
&-f(x*) + f'(x*)(-e_k) + \frac{1}{2}f''(\xi_2)(-e_k)^2](e_{k+1} + e_k) \\
&= -(f'(x*)(-e_k) + \frac{1}{2}f''(\xi_2)(-e_k)^2)^2
\end{aligned}
$$

From here I would hope to simplify the equation to the form of $e_{k+1} = O(e_k^2)$ to show that the formula converges quadratically. However, after eight hours of attempting this, I've accepted defeat, for I have other responsibilities which also demand my time.

Problem 10.

Complete Problem 10. at the end of Chapter 3 in the textbook. **(a)** Newton's iteration for the function $f(x) = (x-1)^2 e^x$ can be found with the following steps.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

$$\begin{aligned} f'(x) &= 2(x-1)e^x + (x-1)^2 e^x \\ &= e^x(2x - 2 + x^2 - 2x + 1) \\ &= e^x(x^2 - 1) \\ &= e^x(x-1)(x+1) \end{aligned}$$

$$x_{k+1} = x_k - \frac{e^x(x_k - 1)^2}{e^x(x_k-1)(x_k+1)}$$

$$x_{k+1} = x_k - \frac{x_k - 1}{x_k + 1}$$

So long as $x_k \neq -1$, the formula will converge. In part (b) we will observe how the convergence rate is similar to that of bisection.

(b) Using the code from problem 3, I added print statements to each iteration of newton's method so that we could see the performance of each iteration. Below is the code and the results.

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++

//Newton_Method.h
#pragma once
#include <cmath>

double newton(double (*f)(double), double (*df)(double), double x0, double tol, unsigned max_iterations) {
  double xk = 99999999; //x sub k
  double xkp1 = x0; //x sub k+1
  std::cout << "x_kp1 \t x_k" << std::endl;
  //Loops until we reach the maximum iterations or we are within the input tolerance.
  for (unsigned k = 0; k < max_iterations && std::abs(xkp1 - xk) > tol; k++) {
    xk = xkp1;
    xkp1 = xk - (f(xk) / df(xk));
    std::cout << xkp1 << "\t" << xk << std::endl;
  }

  return xkp1;
}
```

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++

//Main.cpp
#include <iostream>
#include "Newton_Method.h"
#include <cmath>

double pten(double x) {
  return (x - 1)*(x - 1)*std::exp(x);
}

double ptenprime(double x) {
  return std::exp(x) * (x - 1) * (x + 1);
}

int main(void) {
  double problem10 = newton(pten, ptenprime, 2, pow(10, -8), 15);
  std::cout << problem10 << std::endl;

  return 0;
}
```

```
x_kp1    x_k
1.66667 2
1.41667 1.66667
1.24425 1.41667
1.13542 1.24425
1.072   1.13542
1.03725 1.072
1.01897 1.03725
1.00957 1.01897
1.00481 1.00957
1.00241 1.00481
1.00121 1.00241
1.0006  1.00121
1.0003  1.0006
1.00015 1.0003
1.00008 1.00015
problem10: 1.00008
```

We can see from the results that the performance is very close to that of bisection, especially in the later iterations.

(c) Bisection would not work so well on this equation because rather than crossing the x-axis, the root only touches the x-axis at x=1. The function I've implemented would simply not work, for it checks to make sure that f(a)*f(b) is negative, which would not be true for this equation. Newton's method is more reliable in this circumstance.