Homework 3

Section 2.2

# Problem 3.

The following four methods proposed to compute . Rank them in order, based on their apparent speed of convergence, assuming .

Using the following program:

#include <iostream>

#include <cmath>

using namespace std;

//This is the iteration that we will use.

double Iteration(double p)

{

double pn = 0;

pn = ((20 \* p) + (21 / (p \* p))) / 21;

return pn;

}

//This is the main function.

int main()

{

//Here are the initilizations.

double root = 2.75892417638, p = 1, tol = 0;

int steps = 0;

bool flag = false;

//The tolerance will be 10^-4.

tol = 1 / (pow(10,4));

//Here the loop will be done until we are withing the tolerance.

while (flag == false)

{

p = Iteration(p);

steps += 1;

//Here are the tests to end the loops.

if (abs(p - root) < tol)

{

flag = true;

}

if (steps >= 100)

{

flag = true;

}

}

//Displaying the information.

cout << "The number of steps it took was: ";

cout << steps << endl;

}

Using this method, we got within of the correct value in iterations.

Changing the Iteration:

double Iteration(double p)

{

double pn = 0;

pn = p - ((p \* p \* p) - 21) / (3 \* (p \* p));

return pn;

}

Using this method, we got within of the correct value in iterations.

Changing the Iteration:

double Iteration(double p)

{

double pn = 0;

pn = p - ((p \* p \* p \* p) - 21 \* p) / ((p \* p) - 21);

return pn;

}

This does not converge.

Changing the Iteration:

double Iteration(double p)

{

double pn = 0;

pn = pow((21 / p), .5);

return pn;

}

Using this method, we got within of the correct value in iterations.

The order of the convergence from fasted to slowest is: (b), (d), and (a) but (c) does not converge.

# Problem 11.

For each of the following equations, determine an interval on which fixed-point iteration will converge. Estimate the number of iterations necessary to obtain approximations accurate to within , and perform the calculations.

Using a program:

#include <iostream>

#include <cmath>

using namespace std;

//This is the iteration that we will use.

double Iteration(double p)

{

double pn = 0;

pn = (2 - exp(p) + (p \* p)) / 3;

return pn;

}

//This is the main function.

int main()

{

//Here are the initilizations.

double root = 0, p = 0, tol = 0, test = 0;

int steps = 0;

bool flag = false;

//The tolerance will be 10^-4.

tol = 1 / (pow(10,5));

//Here the loop will be done until we are withing the tolerance.

while (flag == false)

{

test = p - Iteration(p);

p = Iteration(p);

steps += 1;

//Here are the tests to end the loops.

if (abs((test - root)) < tol)

{

flag = true;

}

if (steps >= 100)

{

flag = true;

}

}

//Displaying the information.

cout << "The root is: " << p << endl;

cout << "The number of steps it took was: ";

cout << steps << endl;

}

The root was approximated to be and the number of iterations it took was .

Section 2.3

# Problem 5.

Use Newton’s Method to find solution accurate to within for the following problem:

Newton’s method is the following:

If we iterate this, we can get , thus iterating this, we can get the following:

The only requirement is that we need to have to be close to the root .

For the function , we first need to derive this formula:

One guess we can use is to say that

Using a program:

#include <iostream>

#include <cmath>

using namespace std;

//This is the function which will take out p\_n and give us p\_(n+1)

double NewtonMethod(double p)

{

//These are the parts of the Newton method.

double fp = 0, dfp = 0, pnext = 0;

//The function and it's derivative.

fp = p - cos(p);

dfp = 1 + sin(p);

//This will give us p\_(n+1)

pnext = p - (fp / dfp);

return pnext;

}

//This is the main function.

int main()

{

//These are the values that we will be using.

double pguess = 0, prepeat = 0, fx = 0, tol = 0;

bool flag = false;

//We will use a counter to see how many steps it took to solve this problem.

int counter = 0;

//This is where we will be calculating the tolerance.

tol = 1 / pow(10, 4);

//This is where we will get the user to guess a value.

cout << "Input your guess for the root." << endl;

cin >> pguess;

//We first check the guess to see if they got lucky and got an answer in the tolerance zone.

//If the flag changes to true, then we are done with the process.

fx = pguess - cos(pguess);

if (fx < tol && fx >(-tol))

{

cout << "The root to f(x) = x - cos(x) is:" << fx << endl;

cout << endl;

flag = true;

}

//Here we will take p\_0 and get p\_1

prepeat = NewtonMethod(pguess);

//Since we ran the program once, we need to increase the counter.

counter += 1;

//THis is where the method will be iterated as needed if needed.

while (flag == false)

{

if (fx < tol && fx >(-tol))

{

cout << "The root to f(x) = x - cos(x) is: " << prepeat << endl;

cout << endl;

flag = true;

}

//Here we will check out answer to see if it is correct.

if (flag == false)

{

prepeat = NewtonMethod(prepeat);

fx = prepeat - cos(prepeat);

counter += 1;

}

}

//We will now display all that we have on the function.

cout << "The amount of iterations it took was " << counter << endl;

cout << "The error in the calculation is " << abs(fx) << endl;

if (abs(fx) < tol )

{

cout << "This answer is less than the tolerance by " << abs(fx) - tol << endl;

cout << endl;

}

return 0;

}

We find that the value will be with the initial guess of

# Problem 18.

The function has a zero at . Let and and use ten iterations of each of the following methods to approximate this root. Which method is most successful and why?

1. Bisection method
2. Method of False Position
3. Secant Method

Bisection method:

#include <iostream>

#include <cmath>

#include <math.h>

using namespace std;

//Global Constant

const long double Pi = 3.14159265359;

//This is where the function is at.

long double function(long double x)

{

long double fx = 0;

fx = tan(Pi \* x) - 6;

return fx;

}

//This is where the main program is at.

int main()

{

//The starting points are here where it goes from 0 to 0.48.

long double a = 0, b = .48, xn = 0;

bool flag1 = false;

//This is just to know how many iterations had to be done.

int steps = 0;

//This do loop will perform the bisection method for 10 iterations.

do

{

//This is where the half way value is at.

xn = (a + b) / 2;

//This is where our values will be tested and adjusted as needed.

if (function(xn) < 0)

{

a = xn;

}

else if (function(xn) > 0)

{

b = xn;

}

else if (function(xn) == 0)

{

cout << "The value x = " << xn << " is a root to the polynomial e^x - x^2 + 3\*x - 2 = 0." << endl;

flag1 = true;

}

else

{

cout << "There is a problem" << endl;

flag1 = true;

}

//This is the step counter.

steps = steps + 1;

if (steps == 10)

{

flag1 = true;

}

} while (flag1 == false);

//This is the solution.

cout << "The value for the root via the bisection method for " << steps << " iterations is " << xn << endl;

cout << "The absolute error is " << abs(((1 / Pi) \* atan(6)) - xn) << endl;

cout << "The relative error is " << (abs(((1 / Pi) \* atan(6)) - xn)) / ((1 / Pi) \* atan(6)) << endl;

return 0;

}

Using this method, we get that the root is approximately , the absolute error being and the relative error being .

Secant method:

Using the Newton Method:

And using the definition of the derivative:

If , then:

Thus we can turn the Newton method into the Secant Method:

Using a program:

#include <iostream>

#include <cmath>

#include <math.h>

using namespace std;

const long double Pi = 3.14159265359;

//This is the function that we are trying to find the root for.

double Function(double x)

{

double fx = 0;

fx = tan(Pi \* x) - 6;

return fx;

}

//This is the secant method.

double Secant(double p\_1, double p\_2)

{

double p = 0;

p = p\_1 - (Function(p\_1) \* (p\_1 - p\_2)) / (Function(p\_1) - Function(p\_2));

return p;

}

//This is the main program.

int main()

{

//Initialise all our variables.

double p = 0, p\_1 = 0.48, p\_2 = 0;

int steps = 0;

bool flag = true;

//do loop that will perform the secant method 10 times.

do

{

p = Secant(p\_1, p\_2);

p\_2 = p\_1;

p\_1 = p;

steps += 1;

if (steps == 10)

{

flag = false;

}

} while (flag == true);

//This is the solution.

cout << "The value for the root via the Secant method for " << steps << " iterations is " << p\_1 << endl;

cout << "The absolute error is " << abs(((1 / Pi) \* atan(6)) - p\_1) << endl;

cout << "The relative error is " << (abs(((1 / Pi) \* atan(6)) - p\_1)) / ((1 / Pi) \* atan(6)) << endl;

return 0;

}

We get that the root is , the absolute error , and the relative error .

Method of False position:

This method is almost like that of the secant method only that we use the tool in the bisection method where we check the two points each iteration to see if we are going in the right direction. The method is as follows:

Using a program:

#include <iostream>

#include <cmath>

#include <math.h>

using namespace std;

const long double Pi = 3.14159265359;

//This is the function that we are trying to find the root for.

double Function(double x)

{

double fx = 0;

fx = tan(Pi \* x) - 6;

return fx;

}

//This is the secant method.

double Secant(double p\_1, double p\_2)

{

double p = 0;

p = p\_1 - (Function(p\_1) \* (p\_1 - p\_2)) / (Function(p\_1) - Function(p\_2));

return p;

}

//This is the main program.

int main()

{

//Initialise all our variables.

double p = 0, p\_1 = 0.48, p\_2 = 0;

int steps = 0;

bool flag = true;

//do loop that will perform the secant method 10 times.

do

{

p = Secant(p\_1, p\_2);

//This is where the points will be tested to see which way the iteration continues.

if (Function(p) \* Function(p\_1) < 0)

{

p\_2 = p\_1;

p\_1 = p;

}

else if (Function(p) \* Function(p\_2) < 0)

{

p\_1 = p\_2;

p\_2 = p;

}

else

{

cout << "Error" << endl;

flag = false;

}

steps += 1;

if (steps == 10)

{

flag = false;

}

} while (flag == true);

//This is the solution.

cout << "The value for the root via the Secant method for " << steps << " iterations is " << p << endl;

cout << "The absolute error is " << abs(((1 / Pi) \* atan(6)) - p) << endl;

cout << "The relative error is " << (abs(((1 / Pi) \* atan(6)) - p)) / ((1 / Pi) \* atan(6)) << endl;

return 0;

}

The root ended up being , absolute error , and the relative error .

The Bisection method ended up being the most accurate of all three of these methods. The reason why the Secant method was not very accurate was because and have to be very close and the values we were given weren’t which lead us to get out of range quickly. The Method of False position was good because it constantly was checking if we were in a good range but since the range was not great to begin with, this method was not the most efficient.