Homework 4

# Section 2.4

## Problem 1.

Use Newton’s Method to find solutions accurate to within to the following problem.

#include <iostream>

#include <cmath>

using namespace std;

//This is where the function will be in.

double function(double x)

{

long double fx = 0;

fx = pow(x, 3) - 3 \* pow(x, 2) \* pow(2, -x) + 3 \* x \* pow(4, -x) - pow(8, -x);

return fx;

}

//This is where the derivative will be in.

double derfunction(double x)

{

double dfx = 0;

dfx = 3 \* pow(x, 2) + 3 \* pow(x, 2) \* pow(2, -x) \* log(2) - 6 \* x \* pow(2, -x)

+ 3 \* pow(4, -x) - 3 \* x \* pow(4, -x) \* log(2) + pow(8, -x) \* log(8);

return dfx;

}

//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 pnext = 0;

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

pnext = p - (function(p) / derfunction(p));

return pnext;

}

//This is the main function.

int main()

{

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

double p = 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 steps = 0;

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

tol = 1 / pow(10, 5);

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

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

cin >> p;

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

do

{

if (abs(function(p)) <= tol)

{

flag = true;

}

else

{

p = NewtonMethod(p);

steps += 1;

}

} while (flag == false);

//This is the solution.

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

cout << "The function evaluated at this root is: " << function(p) << endl;

return 0;

}

Using this program, we get that after 418 iterations with the initial guess of 1, we get the value to be .

## Problem 2.

Use Newton’s method to find solutions accurate to within to the following problems.

Using the following function change:

//This is where the function will be in.

double function(double x)

{

long double fx = 0, f1 = 0, f2 = 0, f3 = 0, f4 = 0;

f1 = sin(3 \* x);

f2 = 3 \* exp(-2 \* x)\* sin(x);

f3 = -3 \* exp(-x)\* sin(2 \* x);

f4 = -exp(-3 \* x);

fx = f1 + f2 + f3 + f4;

return fx;

}

//This is where the derivative will be in.

double derfunction(double x)

{

double dfx = 0, df1 = 0, df2 = 0, df3 = 0, df4 = 0;

df1 = 3 \* cos(3 \* x);

df2 = 3 \* exp(-2 \* x)\* cos(x) - 6 \* exp(-2 \* x)\* sin(x);

df3 = -6 \* exp(-x)\* cos(2 \* x) + 3 \* exp(-x)\* sin(2 \* x);

df4 = 3 \* exp(-3 \* x);

dfx = df1 + df2 + df3 + df4;

return dfx;

}

By choosing as our guess, we get that the program converges to in steps.

## Problem 9.

1. Construct a sequence that converges to of order .

Given any sequence , the convergence is as follows:

Since we want this sequence to converge to :

From here we see that the sequence has to follow a form:

Meaning that we need to pick a sequence that has the property that every term will be to the the value cubed. We also want this value to decrease so we can consider a negative exponential.

Take :

Thus we know that converges with order .

1. Suppose . Construct a sequence that converges to of order .

If we generalize the steps is part (a), we can get that the sequence should be:

# Section 2.6

## Problem 9.

Use each of the following methods to find a solution in accurate to within for

1. Bisection Method

#include <iostream>

#include <cmath>

#include <math.h>

using namespace std;

//Global Constants, if any.

//This is where the function is at.

long double function(long double x)

{

long double fx = 0;

fx = 600 \* pow(x, 4) - 550 \* pow(x, 3) + 200 \* pow(x, 2) - 20 \* x - 1;

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 = 0.1, b = 1, xn = 0, tol = 0;

bool flag1 = false;

tol = pow(10, -4);

//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)

{

flag1 = true;

}

else //In case there are any errors.

{

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

flag1 = true;

}

//This is the step counter.

steps = steps + 1;

//This is for any step limitations.

/\*

if (steps == 30)

{

flag1 = true;

}

\*/

//This is for the tolerance.

if (abs(function(xn)) <= tol)

{

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 function evaluated at this root is: " << function(xn) << endl;

return 0;

}

Using the Newton Method, I got that it took steps to get the value .

1. Newton Method

#include <iostream>

#include <cmath>

using namespace std;

//This is where the function will be in.

double function(double x)

{

long double fx = 0;

fx = 600 \* pow(x, 4) - 550 \* pow(x, 3) + 200 \* pow(x, 2) - 20 \* x - 1;

return fx;

}

//This is where the derivative will be in.

double derfunction(double x)

{

double dfx = 0;

dfx = 2400 \* pow(x, 3) - 1650 \* pow(x, 2) + 400 \* x - 20;

return dfx;

}

//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 pnext = 0;

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

pnext = p - (function(p) / derfunction(p));

return pnext;

}

//This is the main function.

int main()

{

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

double p = 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 steps = 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 >> p;

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

do

{

if (abs(function(p)) <= tol)

{

flag = true;

}

else

{

p = NewtonMethod(p);

steps += 1;

}

} while (flag == false);

//This is the solution.

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

cout << "The function evaluated at this root is: " << function(p) << endl;

return 0;

}

Using the endpoints as our initial guess, we see that we get the following:

|  |  |  |
| --- | --- | --- |
|  | – Number of steps |  |
|  |  |  |
|  |  |  |

1. Secant Method

#include <iostream>

#include <cmath>

#include <math.h>

using namespace std;

//Global Constants, if any.

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

double Function(double x)

{

double fx = 0;

fx = 600 \* pow(x, 4) - 550 \* pow(x, 3) + 200 \* pow(x, 2) - 20 \* x - 1;

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 = .1, p\_2 = 1, tol = 0;

int steps = 0;

bool flag = true;

tol = pow(10, -4);

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

do

{

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

if (abs(Function(p)) <= tol)

{

flag = false;

}

else

{

p\_2 = p\_1;

p\_1 = p;

}

steps += 1;

//Step Limit, if any.

/\*

if (steps == 10)

{

flag = false;

}

\*/

} while (flag == true);

//This is the solution.

//This is the solution.

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

cout << "The function evaluated at this root is: " << Function(p) << endl;

return 0;

}

After iterations, we got that the value for was .

1. Method of False Position

#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 = 600 \* pow(x, 4) - 550 \* pow(x, 3) + 200 \* pow(x, 2) - 20 \* x - 1;

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 = 0.1, p\_2 = 1, tol = 0;

int steps = 0;

bool flag = true;

tol = pow(10, -4);

//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 (abs(Function(p)) < tol)

{

flag = false;

}

else 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;

}

//In case of errors.

else

{

cout << "Error" << endl;

flag = false;

}

steps += 1;

//Step limit, if any.

/\*

if (steps == 10)

{

flag = false;

}

\*/

} while (flag == true);

//This is the solution.

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

cout << "The function evaluated at this root is: " << Function(p) << endl;

return 0;

}

Using this method, we get that where . The reason for this high number is because this thing has a huge jump from the points to .

1. Muller’s Method

To find the complex roots of a polynomial function, we can use the method of Muller’s Method. The process for this is to consider a 2nd order polynomial which allow us to catch all the complex roots of our function.

To do this, first consider the following polynomial:

Since we know that this first fact, using the quadratic equation:

Now we can use these formulas to solve for a root which is approximatly . To do this we do:

We can iterate this program as much as needed to make sure that the solution is as accurate as needed.

#include <iostream>

#include <cmath>

#include <math.h>

#include <complex>

using namespace std;

//Global Constants, if any.

//This is where the function is at.

double function(double x)

{

double fx = 0;

fx = 600 \* pow(x, 4) - 550 \* pow(x, 3) + 200 \* pow(x, 2) - 20 \* x - 1;

return fx;

}

//This returns C.

double returnC(double p2)

{

double C = 0;

C = function(p2);

return C;

}

//This returns B.

double returnB(double p0,double p1, double p2)

{

double B = 0, Top = 0, Bottom = 0;

Top = pow(p0 - p2, 2)\*(function(p1) - function(p2)) - pow(p1 - p2, 2)\*(function(p0) - function(p2));

Bottom = (p0 - p2)\*(p1 - p2)\*(p0 - p1);

B = Top / Bottom;

return B;

}

//This returns A.

double returnA(double p0, double p1, double p2)

{

double A = 0, Top = 0, Bottom = 0;

Top = pow(p1 - p2, 2)\*(function(p0) - function(p2)) - pow(p0 - p2, 2)\*(function(p1) - function(p2));

Bottom = (p0 - p2)\*(p1 - p2)\*(p0 - p1);

A = Top / Bottom;

return A;

}

//This returns P3.

double ReturnP3(double p2, double A, double B, double C)

{

double P3 = 0, disc = 0, sign = 0;

disc = pow(B, 2) - 4 \* A \* C;

sign = B / abs(B);

if (disc >= 0)

{

P3 = p2 - (2 \* C) / (B + sign \* pow(disc, .5));

}

else if (disc < 0)

{

P3 = p2 - -B/A;

}

return P3;

}

//This is where the main program is at.

int main()

{

//The starting points are here.

long double p = 0, p0 = 0.1, p1 = .5, p2 = 1, tol = 0;

bool flag1 = false;

tol = pow(10, -4);

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

int steps = 0;

//Doint the first iteration.

p = ReturnP3(p2, returnA(p0, p1, p2), returnB(p0, p1, p2), returnC(p2));

steps += 1;

//This loop will perform the process as many times as needed.

do

{

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

if (abs(function(p)) <= tol)

{

flag1 = true;

}

else if (abs(function(p)) > tol) // Reassigns the values.

{

p0 = p1;

p1 = p2;

p2 = p;

p = ReturnP3(p2, returnA(p0, p1, p2), returnB(p0, p1, p2), returnC(p2));

}

else //In case of an error.

{

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

flag1 = true;

}

//This is the step counter.

steps = steps + 1;

//This is for any step limitations.

/\*

if (steps == 30)

{

flag1 = true;

}

\*/

} while (flag1 == false);

//This is the solution.

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

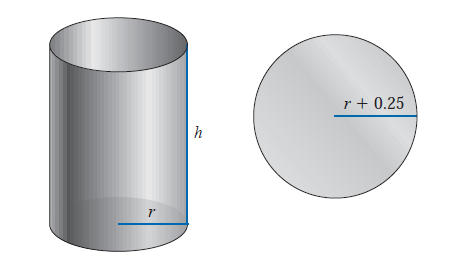
cout << "The function evaluated at this root is: " << function(p) << endl;

return 0;

}

## Problem 11.

A can is the shape of a right circular cylinder is to be constructed to contain . The circular top and bottom of the can must have a radius of more than the radius of the can so that the excess can be used to form a seal with the side. The sheet of material being formed into the side of the can must also be longer than the circumference of the can so that a seal can be formed. Find, to within , the minimal amount of material needed to construct the can.



Volume of a Cylinder:

Surface Area of a Cylinder:

Therefore the equation for the Cylinder as a function of radius is:

Now the material itself has to have the conditions that the radius of the top has to have a radius of more than that of the cylinder. The sheet has to be longer than the circumference of the can. Thus we can find the Surface Area of the material.

To minimize this, we will use the derivative:

Using the Bisection Method from 1 to 10.

long double function(long double x)

{

long double fx = 0;

fx = -(2000 \* x + (500 / PI)) / pow((x), 3) + 4 \* PI \* (x + .25);

return fx;

}

Minimizing this to find the that will give us our solution, we get that .

Plugging this back into our equation, we get that .