Alexandria University

Numerical Analysis

**Assignment 1**

**Ashraf Saleh**

**Antoine Diaa Morcos**

**Mohamed Mahmoud Adel ElBelbesy**

**Ahmed Magdy**

**Ahmed Gamaleldin**

**General structure:**

**abstract Class Function:**  
    double power;  
  
    ***Function getDerivative(){***  
    //    returns the derivative of the function  
    ***Pseudo code:***  
        if(power is zero){  
            return ZeroFunction  
        }  
        if(power is 1)  
            return function derivative  
        return power \* function with decreased power \* function derivative.

}

***abstract differentiateFunction()***

***double abstract getValue(double a)***

// return the value of the function at x = a.

***Subclasses of Function are:***

* ***Class CosFun:***

//    to represent cos(f(x))

***Function differentiateFunction()***

return -1\*sin(f(x))\*f(x).getDerivative.

***double getValue(double a)***

return cos(f(x).getValue(a)).

* ***Class ExpFun:***

//    to represent e^(f(x))

***Function differentiateFunction()***

return ExpFun(f(x))\*f(x).getDerivative().

***double getValue(double a)***

return e^(f(x).getDerivative(a)).

* ***Class NumberFun:***

      double number;

// to represent constant numbers

***Function differentiateFunction()***

return ZeroFunction.

***double getValue(double a)***

return number

* ***Class Polynomial:***

    // to represent the sum of functions.  
    List<Function> functions

***Function differentiateFunction()***

return sum of derivative of functions in the list.

***double getValue(double a)***

return sum of values of functions in the list at x = a.

* ***Class Product:***

    //  to represent the product of functions.  
     List<Function> functions

***Function differentiateFunction()***

term1 = f0\*derivative of (product/f0), where f0 is the first function in the list.

// Recursion is needed to find derivative of product/f0

term2 = derivative of f0\*(product/f0).

derivative = term1 + term2.

return sum of derivative of functions in the list.

***double getValue(double a)***

return product of values of functions in the list at x = a.

* ***Class SinFun:***

//  to represent sin(f(x))

***Function differentiateFunction()***

return cos(f(x))\*f(x).getDerivative.

***double getValue(double a)***

return sin(f(x).getValue(a)).

* ***Class XFun:***

// to represent coefficient\*(x^power)

***Function differentiateFunction()***

return new Number Function(1).

***double getValue(double a)***

return new NumberFunction(coefficient\*(a^power))

* ***Class ZeroFunction:***

// to represent zero.

***Function differentiateFunction()***

return new ZeroFunction.

***double getValue(double a)***

return new ZeroFunction..

Interpolation Methods:

**Lagrange:**

***Polynomial solve():***  
 returns the interpolating polynomial  
Pseudo Code:  
   Polynomial result

for(i=0; i<num. of points; i++){

currentX = points[i].x

currentY = points[i].y

Product term.

    for(j=0; j<size; j++)

        if(j not = i){

            tempX = points[ j ].x

            term.multiply((x-tempX)/(currentX-tempX))

        }

    term.multiply(currentY)

    result.add( term).

}

return result.

***double getError(Function f, double a, double startInterval, double endInterval){***

    double result

    maxDerivative = max value for fsize(c),

( such that c lies in the interval [startInterval, endInterval])

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

result = result \* (a - points[ i ].x)

}

return  result\*maxDerivative/(size)!

}

# Divided difference :

## Data structure used*:*

**double**[] X : Contains the X points

**double**[][] table : Contains the table of the method

**double**[] Y: Contains the Y points

Function F: Function that is going to be interpolate

Vector<Double> C:the Coefficients

Polynomial[] terms: brackets of the X point eg(x-x1)

## Constructors:

***public Divide(Function f, double[] x)*** : take the function compute Y points and fill the table

***public Divide(double[] x, double[] y)*** : just fill the table with Y and initialize the rest

## Methods:

***private void buildTerms():*** build the terms eg (x-x1) and save them in terms Array

***public Polynomial solve():*** Solve the table by applying the Divide difference method

by applying relation :

I 1:n

J i:n

table[j][i] = (table[j][i - 1] - table[j - 1][i - 1]) / (X[j] - X[j - (1 \* i)]);

then get the coefficient in the C array

then call buildTerms

last return the polynomial

***private Polynomial buildPol()*** : build the polynomial as known

***private double fact(int i)*** : get the factorial of a number

***Bisection Method(int a, int b):***

Select midpoint c=(a+b)/2

.Check the cases

–f(c) =0 then the root is at c

// base case

–f(a) and f(c) have opposite signs,

//then the zero in the interval [a,c]

bisection (a,c)

–f(b) and f(c) have opposite signs,

// then the zero in the interval [c,b]

bisection (c,b)

***FalsePostion(int a,int b):***

Select midpoint c=b – f(b)(b-a)/f(b)-f(a)

.Check the cases

–f(c) =0 then the root is at c

// base case

–f(a) and f(c) have opposite signs,

//then the zero in the interval [a,c]

bisection (a,c)

–f(b) and f(c) have opposite signs,

// then the zero in the interval [c,b]

bisection (c,b)

**Fixed point Iteration()**

for (iterations = 1; iterations < max\_iterations; iterations++)

points.add(f.getValue(points.get(iterations - 1)));

double absoluteError = Math.abs(points.get(iterations) - points.get(iterations - 1));

error = absoluteError / (points.get(iterations) + epson);

if (error <= tolerance || absoluteError <= tolerance)

stopping\_reason |= TOLERANCE\_REACHED;

break;

if (iterations == max\_iterations)

stopping\_reason |= MAX\_ITERATIONS\_REACHED;

if (points.isEmpty())

throw new NoSolutionException("No Points Were Generated");

return new fixedPointIteration(points.get(points.size() - 1))

**NewtonRaphson():**

// used for solving the equation *f*(*x*) = 0

double p0 = start\_point, p1 = 0, y = 0;

for (iterations = 0; iterations < max\_iterations; iterations++)

p1 = p0 - (f.getValue(p0) / f.getDerivative().getValue(p0));

points.add(p0);

points.add(p1);

double absolute\_error = abs(p1 - p0);

error = 2 \* absolute\_error / (abs(p1 + tolerance));

p0 = p1;

y = f.getValue(p0);

if (error < tolerance | absolute\_error < tolerance | abs(y) < epson)

stopping\_reason |= TOLERANCE\_REACHED;

break;

if (iterations == max\_iterations)

stopping\_reason |= MAX\_ITERATIONS\_REACHED;

return new Solution(p0)

**SecantMethod():**

double p0 = start\_point, p1 = end\_point, p2 = 0,y=0;

for (iterations = 0; iterations < max\_iterations; iterations++) {

points.add(p0);

points.add(p1);

p2 = p1 - (f.getValue(p1) \* (p1 - p0)) / (f.getValue(p1f.getValue(p0));

double absolute\_error = abs(p2-p1);

error = 2\*absolute\_error/(abs(p2)-tolerance);

p0=p1;

p1=p2;

y=f.getValue(p1);

if(error<tolerance | absolute\_error<tolerance | abs(y) <epson){

stopping\_reason |= TOLERANCE\_REACHED;

break;

}

}

if(iterations==max\_iterations)

stopping\_reason |= MAX\_ITERATIONS\_REACHED;

return new secantMethod(p0);

**Analysis:**

In the fixed point iteration as p tends to infinity the iterations converge to approximately equal values.

Example: 𝑃0 =.5 and 𝑝𝑘+1= 𝑒−𝑝𝑘 for k = 0,1 , 2,..

As k increases converge to approximately equal values.

if 𝑔′(𝑥)>1 For all 𝑥∈[𝑎,𝑏], then 𝑝𝑛=𝑔𝑝𝑛−1 will not converge to 𝑃 in [𝑎,𝑏]. 𝑃 is said to be a repelling fixed point and the iteration exhibits local divergence.

if 𝑔′(𝑥) ≤𝐾<1 For all 𝑥∈[𝑎,𝑏], then 𝑝𝑛=𝑔𝑝𝑛−1 will converge to the unique fixed point 𝑃 in [𝑎,𝑏].

In bisection method must start with interval [𝑎,] where (𝑎) and (𝑏) have opposite signs.

Systematically moves the end points of the interval towards the root.

In false position method when the number of iterations exceeds certain value, the iterations result doesn’t change

Example: ℎ(𝑥) = 𝑥 𝑠𝑖𝑛(𝑥)−1 in the interval [0,2].

In Newton Raphson method as p chosen is closer to the root, the number of iterations required to reach the root becomes less.

But it has many problems as:

1-Division by Zero when 𝑓′(𝑥)≈0

2-Having double roots will slow the convergence

3-Far away initial approximation would fail the method

It has advantages as:

1 – Fast convergence.

2 - Requires one guess only.

3 – Has the fastest rate of convergence.

In Lagarange polynomials passes through each of the points used in its construction

pN(x)=y0(x-x1)(x-x2)…(x-xN)/(x0-x1)(x0-x2)(x0-xN)

+ y1(x-x0)(x-x2)…(x-xN)/(x1-x0)(x1-x2)(x1-xN)

+ yN(x-x0)(x-x1)…(x-xN)/(xN-x0)(xN-x2)(xN-xN-1)

Advantages:

1-Uniqueness The difference polynomial has degree ≤𝑁

𝑇(𝑥)=𝑃𝑁𝑥−𝑄𝑁𝑥

For 𝑥=𝑥0 to 𝑥𝑁 we have 𝑃𝑁𝑥−𝑄𝑁𝑥=0

Thus (𝑥) must be 0

2-Also, the data are not required to be specified with x in ascending or descending order

Disadvantages:

1-Although the computation of PN(x) is simple, the method is still not particularly efficient for large values of N.

2-When N is large and the data for x is ordered, some improvement in efficiency can be obtained by considering only the data pairs in the vicinity of the x value for which PN(x) is sought.

3-The price of this improved efficiency is the possibility of a poorer approximation to PN(x).

4-It involves more arithmetic operations than does the divided differences.

5- If we desire to add or subtract a point from the set to construct the polynomial, we essentially have to start over in the computations.

The divided difference avoids this.