**Basic Numerical Methods**

**Project**

Submitted by:

**Abhijot Singh                 19103180**

**Kshitij Shakya           19103181**

**Ayush Jaiswal          19103185**

**Vansh Sachdeva         19103194**

Under the supervision of:

**------------------Prof. Ramesh Chand Mittal -----------------**

****

**Jaypee Institute of Information Technology University, Noida**

Acknowledgement

The completion of any inter-disciplinary project depends upon cooperation, coordination, and combined efforts of several sources of knowledge. We are grateful to Dr Ramesh Chand Mittal for isr willingness to give us valuable advice and direction whenever we approached him with a problem.

We are thankful to her for providing us with immense guidance for this project. We would also like to thank our College authorities for allowing us to pursue our project in this subject.

**Methods:**

**1) Bisection Method:**

The Bisection is used to find the root of a polynomial. Separate the intervals and divide the interval where the square root of the equation exists. The principle of this method is the intermediate value theorem for continuous functions. It works by narrowing the gap between positive and negative intervals until you reach the correct answer. This technique narrows the gap by averaging the positive and negative intervals. This is an easy method and is relatively time consuming. The dichotomy method is also known as the interval half method, root-finding method, binary search method, and dichotomy method.

Let us consider a continuous function “f” which is defined on the closed interval [a, b], is given with f(a) and f(b) of different signs. Then by intermediate theorem, there exists a point x belong to (a, b) for which f(x) = 0.

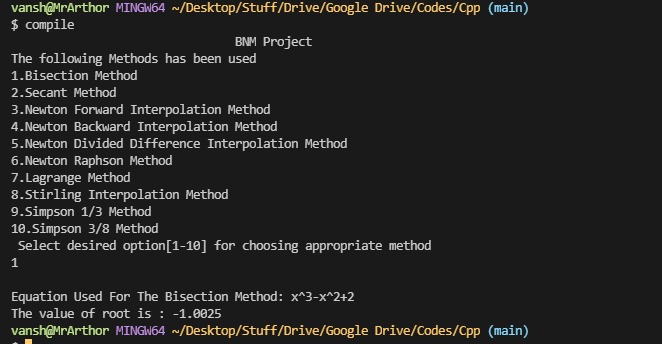
**Algorithm:**

For any continuous function f(x),

* Find two points, say a and b such that a < b and f(a)\* f(b) < 0
* Find the midpoint of a and b, say “t”
* t is the root of the given function if f(t) = 0; else follow the next step
* Divide the interval [a, b] – If f(t)\*f(a) <0, there exist a root between t and a  
  – else if f(t) \*f (b) < 0, there exist a root between t and b
* Repeat above three steps until f(t) = 0.

The bisection method is an approximation method to find the roots of the given equation by repeatedly dividing the interval. This method will divide the interval until the resulting interval is found, which is extremely small.

**Code Ouput:**



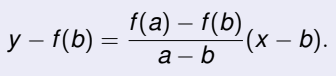
**2) Secant Method**

This method is likely to the Ragula-Falsi method but here at the two

points a and b, the curve y = f(x) need not to have opposite signs.

Now we are finding the root with the help of a straight line joining the

two points (b, f(b)) and (a, f(a)) i.e.



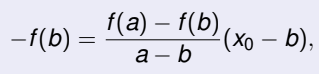
The point where this line cuts the x-axis, we assume as the initial

approximation, therefore on x-axis the y-ordinate is y = 0 and let the

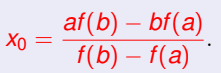
x-ordinate be x = x0.

**Algorithm:**

Now the equation of line becomes



on solving, we get



If f(x0) = 0, then x0 is the approximate root of the equation f(x) = 0.

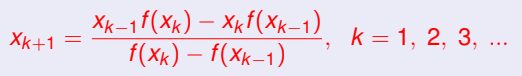
If f(x0) ≠ 0, then we may discard any of the point either a or b and will

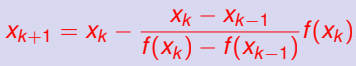
process the same with remaining two points either x0 and a or x0 and b

until we get the root with desired accuracy.

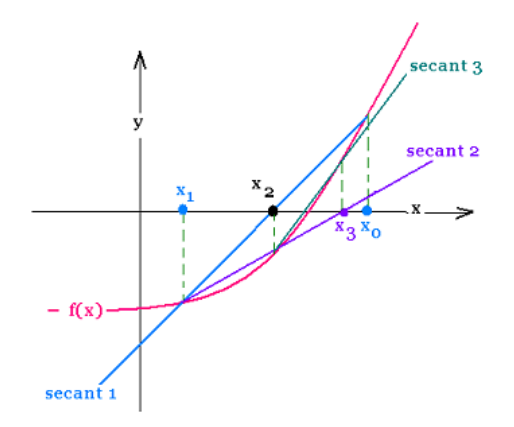
Let iterations xk−1 and xk have been computed then the (k + 1)th

iteration of Secant method is given by

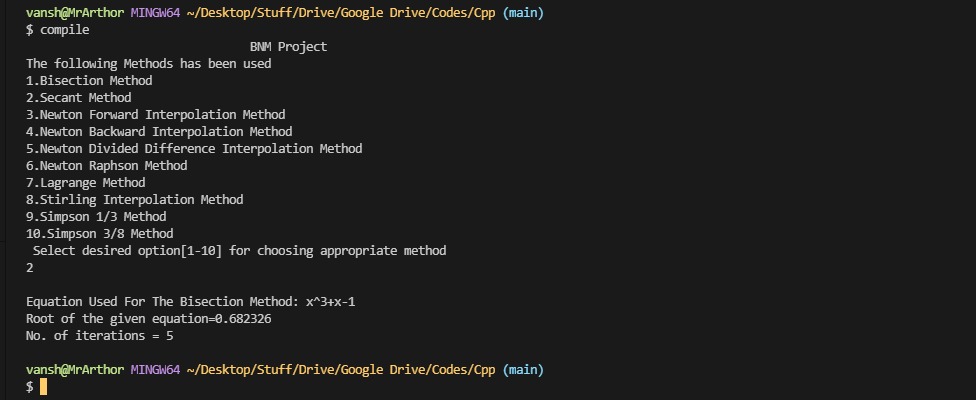


Or  

**Geometrical Interpretation:**



**Code Output:**



**3) Newton Forward Interpolation Method**

Gregory Newton’s is a forward difference formula which is applied to calculate finite difference identity. Regarding the first value f0 and the power of the forward difference Δ, Gregory Newton’s forward formula gives an interpolated value between the tabulated points.

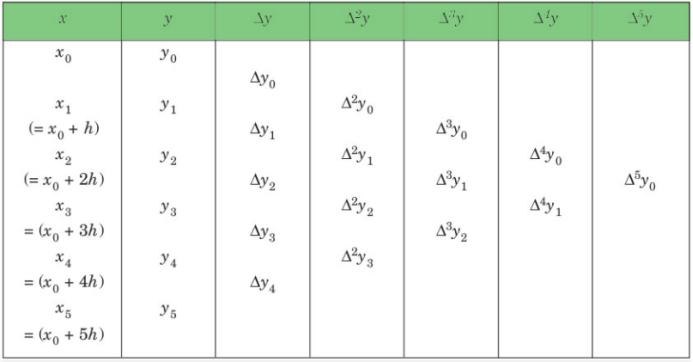
**Interpolation** is the technique of estimating the value of a function for any intermediate value of the independent variable.

**Algorithm:**

Forward Differences: The differences y1 – y0, y2 – y1, y3 – y2, ……, yn – yn–1 when denoted by dy0, dy1, dy2, ……, dyn–1 are respectively, called the first forward differences. Thus, the first forward differences are :



**Forward Difference Table:**

****

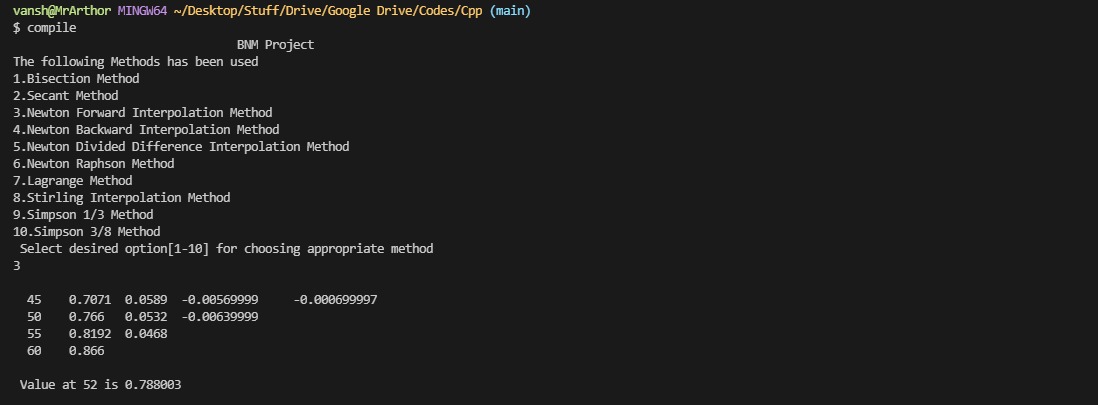
**NEWTON’S GREGORY FORWARD INTERPOLATION FORMULA** :



**Where, h = interval of difference, and**

**u = ( x – a ) / h**

**Code Output:**



**4) Newton Backward Interpolation Method**

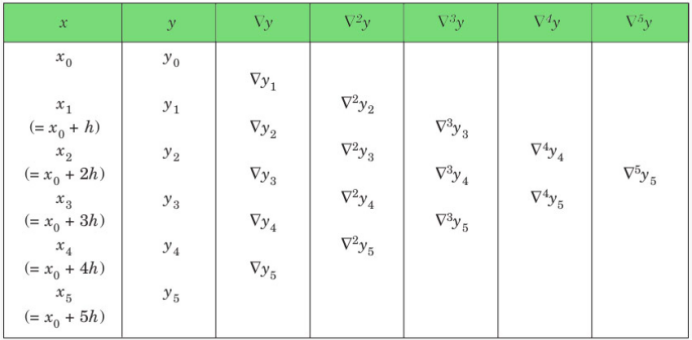
**Algorithm:**

**Backward Differences**: The differences y1 – y0, y2 – y1, ……, yn – yn–1 when denoted by dy1, dy2, ……, dyn, respectively, are called first backward difference.

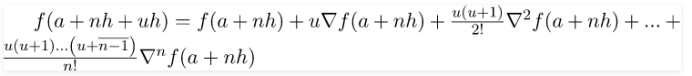
Thus, the first backward differences are :



**Backward Difference Table:**

****

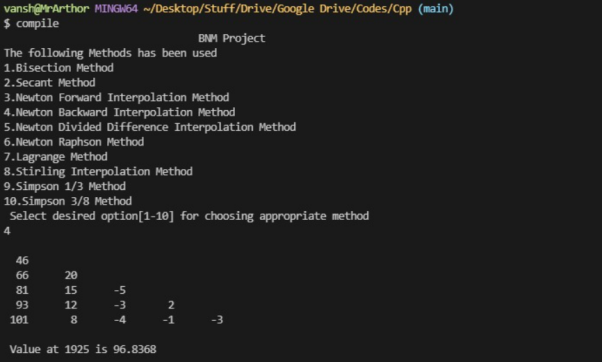
**NEWTON’S GREGORY BACKWARD INTERPOLATION FORMULA** :



**Where, h = interval of difference, and**

**u = ( x – an ) / h**

**Code Output:**

****

**5) Newton Divided Difference Interpolation**

**Interpolation** is an estimation of a value within two known values in a sequence of values.

**Newton’s divided difference interpolation formula** is a interpolation technique used when the interval difference is not same for all sequence of values.

**Algorithm:**

Suppose f(x0), f(x1), f(x2)………f(xn) be the (n+1) values of the function y=f(x) corresponding to the arguments x=x0, x1, x2…xn, where interval differences are not same

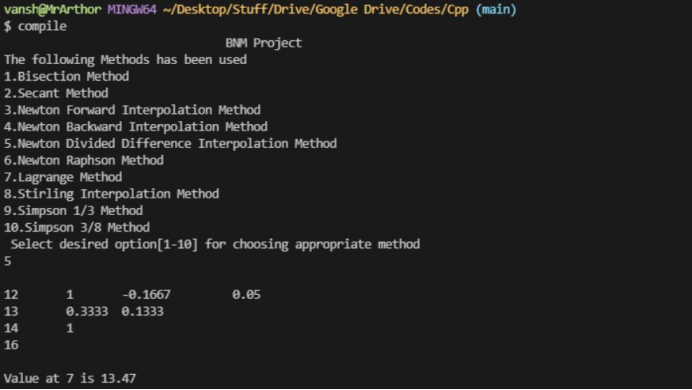
Then the first divided difference is given by:

**f[x0, x1]=f(x1) - f(x0)/(X1 - X0)**

The second divided difference is given by:

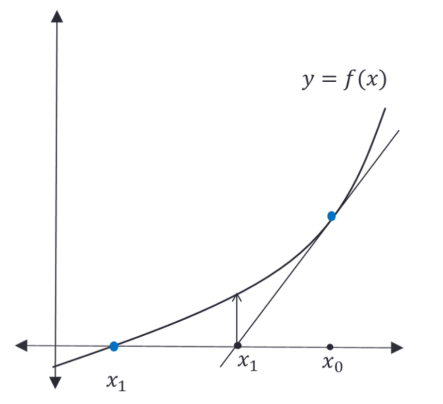
**f[x0, x1, x2]=f(x1,X2) - f(x0,X1)/(X2 - X0)**

**Code Output:**

****

**6) Newton Raphson Method**

Newton Raphson Method is yet another numerical method to approximate the root of a polynomial. Newton Raphson Method is an open method of root finding which means that it needs a single initial guess to reach the solution instead of narrowing down two initial guesses.



Newton Raphson Method uses to the slope of the function at some point to get closer to the root. Using equation of line **y =** [**m**](https://xplaind.com/869399/javascript:window.alert()**x0 + c** we can calculate the point where it meets x axis, in a hope that the original function will meet x-axis somewhere near. We can reach the original root if we repeat the same step for the new value of x.

**Algorithm:**

Suppose you need to find the root of a continuous, differentiable function f(x), and you know the root you are looking for is near the point x = x0.

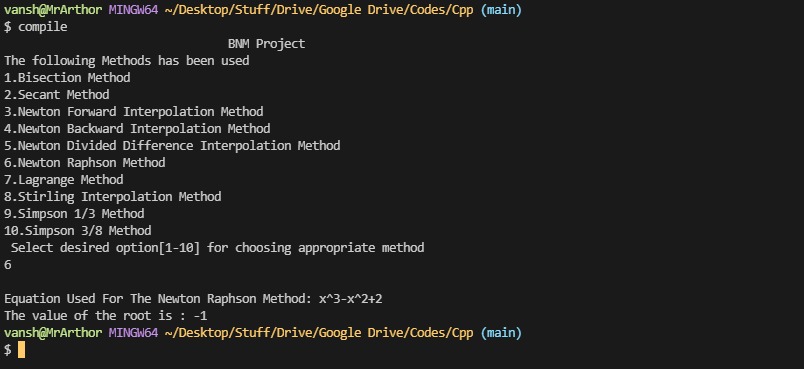
Then Newton's method tells us that a better approximation for the root is

x1 = x0 - f (x0)/f ’(x0)

Therefore, **General Term:**

Xn+1 = xn - f (xn)/f ’(xn)

**Code Output:**

****

**7) Lagrange’s Interpolation**

**Algorithm:**

Suppose, 𝑦 = 𝑓(𝑥) is a given function.

Let us consider, 𝑓(𝑥0 ), 𝑓(𝑥1 ), 𝑓(𝑥2 ), …………….., 𝑓(𝑥𝑛 ) are the values of the function 𝑦 = 𝑓(𝑥) corresponding to the arguments 𝑥0, 𝑥1, 𝑥2, ……………., 𝑥𝑛 respectively, not necessarily equally spaced.

We assume, 𝑃𝑛 (𝑥) is a polynomial in 𝑥 of degree 𝑛 such that

Pn (x) = A0 (x - x1 )(x - x2 )(x - x3 ) ……….(x - xn  )

+A1 x - x0 x - x2 x - x3  ……….x - xn

+A2 x - x0 x - x1 x - x3  ……….x - xn

+A3 x - x0 x - x1 x - x2 ……….x - xn

+ …………………………….

+An x - x0 x - x1 x - x2 ……….x - xn-1 ……… (1)

Where 𝐴0, 𝐴1, 𝐴3, … ………, 𝐴𝑛 are (𝑛 + 1) constants to be determined. To determine the constants 𝐴0,𝐴1, 𝐴3, … ………, 𝐴𝑛, we assume

𝑃𝑛 (𝑥0 ) = 𝑓(𝑥0 )

𝑃𝑛 (𝑥1 ) = 𝑓(𝑥1 )

𝑃𝑛 (𝑥2 ) = 𝑓(𝑥2 )

……………………………….

………………………………

𝑃𝑛 (𝑥𝑛 ) = 𝑓(𝑥𝑛 )

Now, putting successively 𝑥 = 𝑥0, 𝑥1, 𝑥2, …………………… , 𝑥𝑛 in (1), we get

𝑓(𝑥0 ) = 𝐴0 (𝑥0 − 𝑥1 )(𝑥0 − 𝑥2 )(𝑥0 − 𝑥3 ) ……….(𝑥0 − 𝑥𝑛 )

⇒ A0 = f(x0 )(x0-x1 )(x0-x2 )(x0-x3 ) ……….(x0-xn)

Similarly, we have

 A1 = f(x1)(x1-x0 )(x1-x2 )(x1-x3 ) ……….(x1-xn)

 A2 = f(x2)(x2-x0 )(x2-x1 )(x2-x3 ) ……….(x2-xn)

…………………………………………………………………………… ……………………………………………………………………………

 An = fxn xn-x0 xn-x1 xn-x2 ……….xn-xn-1

Now, putting these values of the constants in (1), we get

Pn x=x - x1 x - x2 x - x3 … … … . x - xn x0 - x1 x0 - x2 x0  - x3 … … … . x0 - xn   fx0

+x-x0 x-x2 x-x3 ……….x-xnx1-x0x1-x2 x1-x3 ……….x1-xn fx1

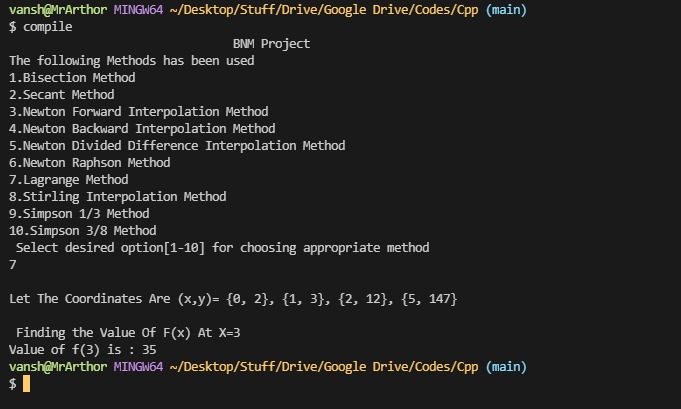
+x-x0 x-x1 x-x3 ……….x-xnx2 -x0x2 -x1 x2 -x3 ……….x2 -xn  fx2

+ ……………………………………………………..

+x-x0 x-x1 x-x2  ……….x-xn-1 (xn-x0 )(xn-x1 )(xn-x2  ) ……….(xn-xn-1 )  f(xn)…… (2)

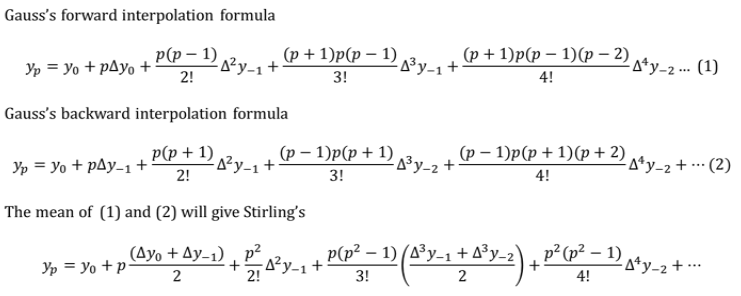
This formula (2) is known as Lagrange’s Interpolation Formula with unequal intervals.

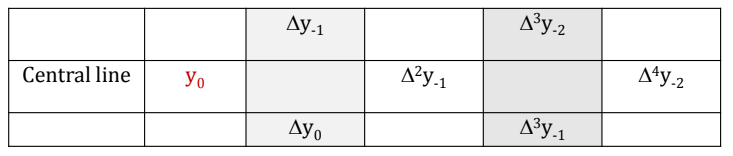
**Code Output:**

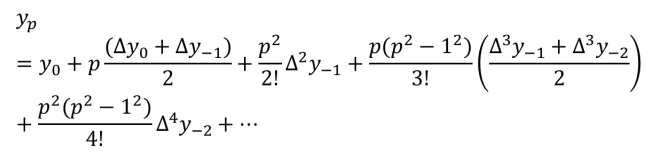


**8) Stirling’s formula**

**Algorithm:**

****

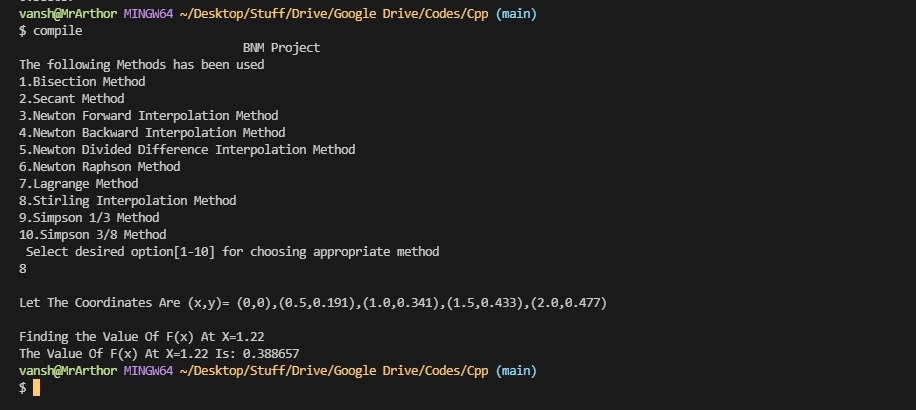
****

****

**WHEN TO APPLY:**

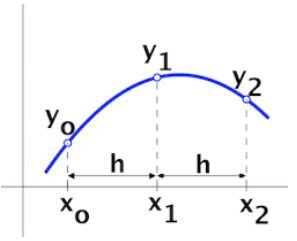
* This formula is applied for interpolation near the middle of the table for the values *p* close to zero.
* In practical application, it is applied for -0.5<p<0.5, but preferred for -0.25<p<0.25.

**Code Output:**



**9) Simpson’s 1/3 rule**

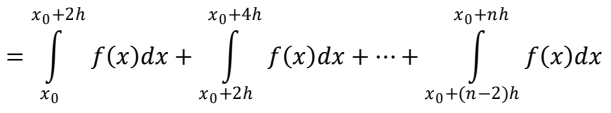
Here the function f(x) is approximated to second degree polynomial passing through three successive points (xi, yi), (xi+1, yi+1), and (xi+2, yi+2), i = 0, 3, 6, ... , n − 2 respectively. So, the area bounded by the curve y = f(x) , x-axis and the ordinates x = x0 and x = xn is then approximately equal to the sum of the areas of each n/2 segment of width 2h as shown in the figure



**Algorithm:**

Now,

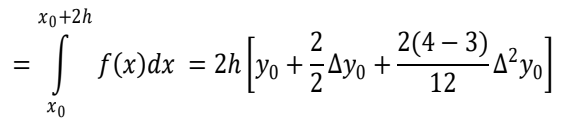


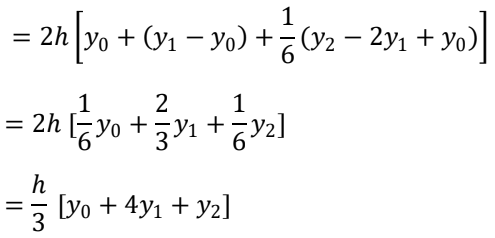


Putting n = 2 in N-C formula, all differences higher than three will become zero (since other differences do not exist if n > 2)

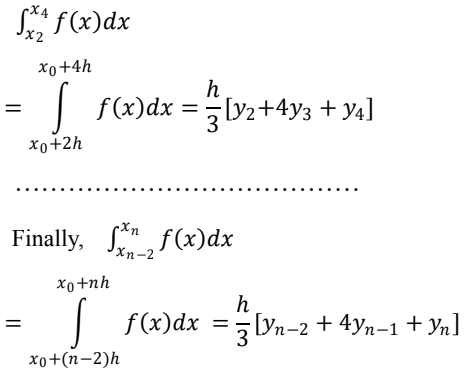
Notably, the given interval must be divided into an even number of equal sub-intervals of width h.

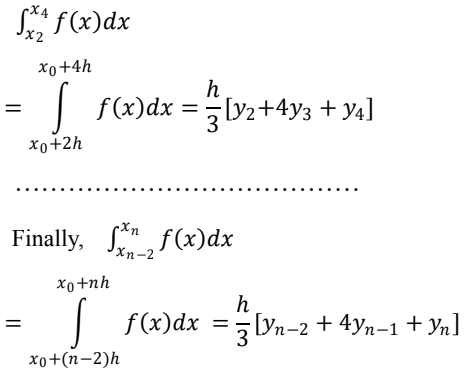
So, 

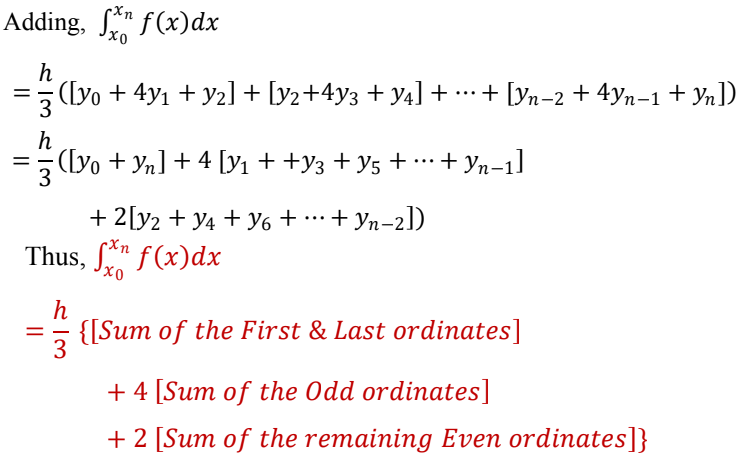




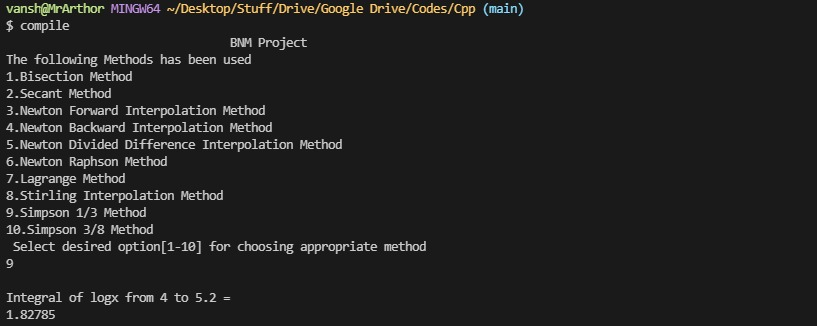
Similarly,

****

****

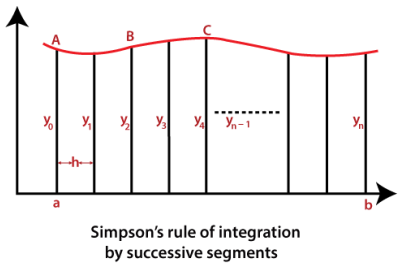
****

**Code Output:**



**10) Simpson’s 3/8 rule**

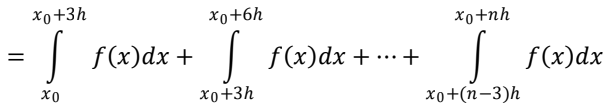
Here the function f(x) is approximated to third degree polynomial passing through four successive points (xi, yi), (xi+1, yi+1), (xi+2, yi+2), and (xi+3, yi+3), i = 0, 3, 6, ... , n − 3 respectively. So, the area bounded by the curve y = f(x) , x-axis and the ordinates x = x0 and x = xn is then approximately equal to the sum of the areas of each n/3 segment of width 3h as shown in the figure



**Algorithm:**

Now,

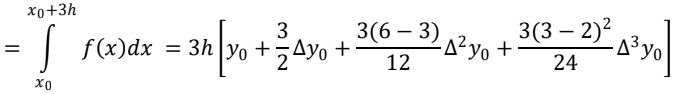


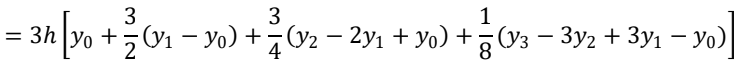


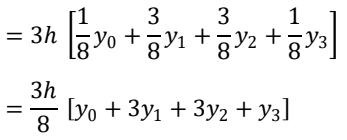
Putting n = 3 in N-C formula, all differences higher than three will become zero (since other differences do not exist if n > 3)

Notably, the given interval must be divided into a multiple of three, equal sub-intervals of width 3h.

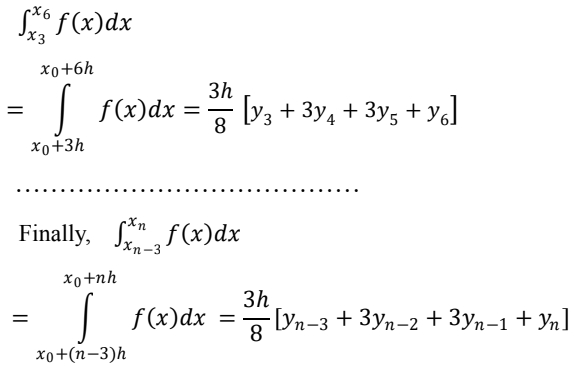
So,

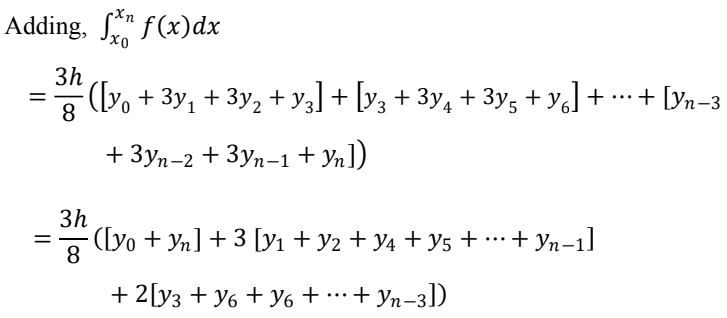




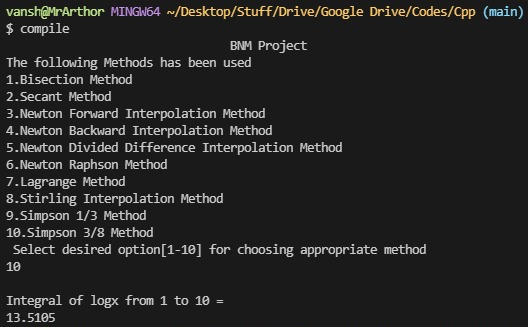


Similarly,

****

****

**Code Output:**

****

Code:-

*#include* <bits/stdc++.h>

using namespace std;

*#define* EPSILON 0.01

int BisectionMethod()

{

auto Function = [&](double x)

{

*return* x \* x \* x - x \* x + 2;

};

auto bisection = [&](double a, double b)

{

cout << "Equation Used For The Bisection Method: x^3-x^2+2 " << endl;

*if* (Function(a) \* Function(b) >= 0)

{

cout << "You have not assumed right a and b\n";

*return*;

}

double c = a;

*while* ((b - a) >= EPSILON)

{

c = (a + b) / 2;

*if* (Function(c) == 0.0)

*break*;

*else* *if* (Function(c) \* Function(a) < 0)

b = c;

*else*

a = c;

}

cout << "The value of root is : " << c;

};

double a = -200, b = 300;

bisection(a, b);

*return* 0;

}

int NRMethod()

{

cout << "Equation Used For The Newton Raphson Method: x^3-x^2+2 " << endl;

auto Function = [&](double x)

{

*return* x \* x \* x - x \* x + 2;

};

auto derivFunction = [&](double x)

{

*return* 3 \* x \* x - 2 \* x;

};

auto NewtonRaphson = [&](double x)

{

double h = Function(x) / derivFunction(x);

*while* (abs(h) >= EPSILON)

{

h = Function(x) / derivFunction(x);

x = x - h;

}

cout << "The value of the root is : " << x;

};

double x0 = -20;

NewtonRaphson(x0);

*return* 0;

}

int NewtonDividedDifferenceMethod()

{

auto proterm = [&](int i, float value, float x[])

{

float pro = 1;

*for* (int j = 0; j < i; j++)

{

pro = pro \* (value - x[j]);

}

*return* pro;

};

*// Function to evaluate the polynomial*

auto DividedDifferenceTable = [&](float x[], float y[][10], int n)

{

*for* (int i = 1; i < n; i++)

{

*for* (int j = 0; j < n - i; j++)

{

y[j][i] = (y[j][i - 1] - y[j + 1]

[i - 1]) /

(x[j] - x[i + j]);

}

}

};

auto applyFormula = [&](float value, float x[], float y[][10], int n)

{

float Sum = y[0][0];

*for* (int i = 1; i < n; i++)

{

Sum = Sum + (proterm(i, value, x) \* y[0][i]);

}

*return* Sum;

};

auto printDiffTable = [&](float y[][10], int n)

{

*for* (int i = 0; i < n; i++)

{

*for* (int j = 0; j < n - i; j++)

{

cout << setprecision(4) << y[i][j] << "\t ";

}

cout << "\n";

}

};

int n = 4;

float value, Sum, y[10][10];

float x[] = {5, 6, 9, 11};

y[0][0] = 12;

y[1][0] = 13;

y[2][0] = 14;

y[3][0] = 16;

DividedDifferenceTable(x, y, n);

printDiffTable(y, n);

value = 7;

cout << "\nValue at " << value << " is "

<< applyFormula(value, x, y, n) << endl;

*return* 0;

}

int NewtonForwardInterpolation()

{

auto u\_cal = [&](float u, int n)

{

float temp = u;

*for* (int i = 1; i < n; i++)

temp = temp \* (u - i);

*return* temp;

};

auto fact = [&](int n)

{

int f = 1;

*for* (int i = 2; i <= n; i++)

f \*= i;

*return* f;

};

int n = 4;

float x[] = {45, 50, 55, 60};

float y[n][n];

y[0][0] = 0.7071;

y[1][0] = 0.7660;

y[2][0] = 0.8192;

y[3][0] = 0.8660;

*for* (int i = 1; i < n; i++)

{

*for* (int j = 0; j < n - i; j++)

y[j][i] = y[j + 1][i - 1] - y[j][i - 1];

}

*for* (int i = 0; i < n; i++)

{

cout << setw(4) << x[i]

<< "\t";

*for* (int j = 0; j < n - i; j++)

cout << setw(4) << y[i][j]

<< "\t";

cout << endl;

}

float value = 52;

float Sum = y[0][0];

float u = (value - x[0]) / (x[1] - x[0]);

*for* (int i = 1; i < n; i++)

{

Sum = Sum + (u\_cal(u, i) \* y[0][i]) /

fact(i);

}

cout << "\n Value at " << value << " is "

<< Sum << endl;

*return* 0;

}

int NewtonBackwardInterpolationMethod()

{

auto u\_cal = [&](float u, int n)

{

float temp = u;

*for* (int i = 1; i < n; i++)

temp = temp \* (u + i);

*return* temp;

};

auto fact = [&](int n)

{

int f = 1;

*for* (int i = 2; i <= n; i++)

f \*= i;

*return* f;

};

int n = 5;

float x[] = {1891, 1901, 1911,

1921, 1931};

float y[n][n];

y[0][0] = 46;

y[1][0] = 66;

y[2][0] = 81;

y[3][0] = 93;

y[4][0] = 101;

*for* (int i = 1; i < n; i++)

{

*for* (int j = n - 1; j >= i; j--)

y[j][i] = y[j][i - 1] - y[j - 1][i - 1];

}

*for* (int i = 0; i < n; i++)

{

*for* (int j = 0; j <= i; j++)

cout << setw(4) << y[i][j]

<< "\t";

cout << endl;

}

float value = 1925;

float Sum = y[n - 1][0];

float u = (value - x[n - 1]) / (x[1] - x[0]);

*for* (int i = 1; i < n; i++)

{

Sum = Sum + (u\_cal(u, i) \* y[n - 1][i]) /

fact(i);

}

cout << "\n Value at " << value << " is "

<< Sum << endl;

*return* 0;

}

int LagrangeInterpolationMethod()

{

struct Data

{

int x, y;

};

auto interpolate = [&](Data f[], int xi, int n)

{

double result = 0;

*for* (int i = 0; i < n; i++)

{

double term = f[i].y;

*for* (int j = 0; j < n; j++)

{

*if* (j != i)

term = term \* (xi - f[j].x) / double(f[i].x - f[j].x);

}

result += term;

}

*return* result;

};

Data f[] = {{0, 2}, {1, 3}, {2, 12}, {5, 147}};

cout << "Let The Coordinates Are (x,y)= {0, 2}, {1, 3}, {2, 12}, {5, 147}\n\n";

cout<<" Finding the Value Of F(x) At X=3\n";

cout << "Value of f(3) is : " << interpolate(f, 3, 5);

*return* 0;

}

int Simpsons38Rule()

{

auto Function = [&](float x)

{

*return* log(x);

};

auto calculate = [&](float lower\_limit, float upper\_limit,

int interval\_limit)

{

float value;

float interval\_size = (upper\_limit - lower\_limit) / interval\_limit;

float Sum = Function(lower\_limit) + Function(upper\_limit);

*for* (int i = 1; i < interval\_limit; i++)

{

*if* (i % 3 == 0)

Sum = Sum + 2 \* Function(lower\_limit + i \* interval\_size);

*else*

Sum = Sum + 3 \* Function(lower\_limit + i \* interval\_size);

}

*return* (3 \* interval\_size / 8) \* Sum;

};

int interval\_limit = 10;

float lower\_limit = 1;

float upper\_limit = 10;

float integral\_res = calculate(lower\_limit, upper\_limit,

interval\_limit);

cout << integral\_res;

*return* 0;

}

int Simpson13Rule()

{

auto Function = [&](float x)

{

*return* log(x);

};

auto simpsons\_ = [&](float ll, float ul, int n)

{

float h = (ul - ll) / n;

float x[10], fx[10];

*for* (int i = 0; i <= n; i++)

{

x[i] = ll + i \* h;

fx[i] = Function(x[i]);

}

float res = 0;

*for* (int i = 0; i <= n; i++)

{

*if* (i == 0 || i == n)

res += fx[i];

*else* *if* (i % 2 != 0)

res += 4 \* fx[i];

*else*

res += 2 \* fx[i];

}

res = res \* (h / 3);

*return* res;

};

float lower\_limit = 4;

float upper\_limit = 5.2;

int n = 6;

cout << simpsons\_(lower\_limit, upper\_limit, n);

*return* 0;

}

int SecantMethod()

{

auto f = [&](float x)

{

float f = pow(x, 3) + x - 1;

*return* f;

};

cout << "Equation Used For The Bisection Method: x^3+x-1 " << endl;

auto secant = [&](float x1, float x2, float E)

{

float n = 0, xm, x0, c;

*if* (f(x1) \* f(x2) < 0)

{

*do*

{

x0 = (x1 \* f(x2) - x2 \* f(x1)) / (f(x2) - f(x1));

c = f(x1) \* f(x0);

x1 = x2;

x2 = x0;

n++;

*if* (c == 0)

*break*;

xm = (x1 \* f(x2) - x2 \* f(x1)) / (f(x2) - f(x1));

} *while* (fabs(xm - x0) >= E);

cout << "Root of the given equation=" << x0 << endl;

cout << "No. of iterations = " << n << endl;

}

*else*

cout << "Can not find a root in the given interval";

};

float x1 = 0, x2 = 1, E = 0.0001;

secant(x1, x2, E);

*return* 0;

}

int StirlingMethod()

{

auto Stirling = [&](float x[], float fx[], float x1,

int n)

{

float h, a, u, y1 = 0, N1 = 1, d = 1,

N2 = 1, d2 = 1, temp1 = 1, temp2 = 1,

k = 1, l = 1, delta[n][n];

int i, j, s;

h = x[1] - x[0];

s = floor(n / 2);

a = x[s];

u = (x1 - a) / h;

*for* (i = 0; i < n - 1; ++i)

{

delta[i][0] = fx[i + 1] - fx[i];

}

*for* (i = 1; i < n - 1; ++i)

{

*for* (j = 0; j < n - i - 1; ++j)

{

delta[j][i] = delta[j + 1][i - 1] - delta[j][i - 1];

}

}

y1 = fx[s];

*for* (i = 1; i <= n - 1; ++i)

{

*if* (i % 2 != 0)

{

*if* (k != 2)

{

temp1 \*= (pow(u, k) -

pow((k - 1), 2));

}

*else*

{

temp1 \*= (pow(u, 2) -

pow((k - 1), 2));

}

++k;

d \*= i;

s = floor((n - i) / 2);

y1 += (temp1 / (2 \* d)) \*

(delta[s][i - 1] +

delta[s - 1][i - 1]);

}

*else*

{

temp2 \*= (pow(u, 2) -

pow((l - 1), 2));

++l;

d \*= i;

s = floor((n - i) / 2);

y1 += (temp2 / (d)) \*

(delta[s][i - 1]);

}

}

cout << y1;

};

int n;

n = 5;

float x[] = {0, 0.5, 1.0, 1.5, 2.0};

float fx[] = {0, 0.191, 0.341, 0.433,

0.477};

cout << "Let The Coordinates Are (x,y)= (0,0),(0.5,0.191),(1.0,0.341),(1.5,0.433),(2.0,0.477)\n\n";

cout<<"Finding the Value Of F(x) At X=1.22\n";

float x1 = 1.22;

cout<<"The Value Of F(x) At X=1.22 Is: ";

Stirling(x, fx, x1, n);

*return* 0;

}

int main()

{

cout << "\t\t\t\tBNM Project\n";

cout << "The following Methods has been used \n";

cout << "1.Bisection Method\n";

cout << "2.Secant Method\n";

cout << "3.Newton Forward Interpolation Method\n";

cout << "4.Newton Backward Interpolation Method\n";

cout << "5.Newton Divided Difference Interpolation Method\n";

cout << "6.Newton Raphson Method\n";

cout << "7.Lagrange Method\n";

cout << "8.Stirling Interpolation Method\n";

cout << "9.Simpson 1/3 Method\n";

cout << "10.Simpson 3/8 Method\n";

cout << " Select desired option[1-10] for choosing appropriate method\n";

int choose;

cin >> choose;

*switch* (choose)

{

*case* 1:

cout << endl;

BisectionMethod();

*break*;

*case* 2:

cout << endl;

SecantMethod();

*break*;

*case* 3:

cout << endl;

NewtonForwardInterpolation();

*break*;

*case* 4:

cout << endl;

NewtonBackwardInterpolationMethod();

*break*;

*case* 5:

cout << endl;

NewtonDividedDifferenceMethod();

*break*;

*case* 6:

cout << endl;

NRMethod();

*break*;

*case* 7:

cout << endl;

LagrangeInterpolationMethod();

*break*;

*case* 8:

cout << endl;

StirlingMethod();

*break*;

*case* 9:

cout << endl;

cout << "Integral of logx from 4 to 5.2 = \n";

Simpson13Rule();

*break*;

*case* 10:

cout << endl;

cout << "Integral of logx from 1 to 10 = \n";

Simpsons38Rule();

*break*;

*default*:

cout << "Wrong number selected\n";

}

*return* 0;

}