Numerical Methods

Table of Contents

Solution of Linear Equations

Gauss Elimination Method

Gauss Elimination Method Theory

Add theory here

Gauss Elimination Method Code

```
// add your code here
```

Gauss Elimination Method Input

```
add input here
```

Gauss Elimination Method Output

```
add output here
```

Gauss Jordan Elimination Method

Gauss Jordan Elimination Method Theory

Add theory here

Gauss Jordan Elimination Method Code

```
// add your code here
```

Gauss Jordan Elimination Method Input

```
add input here
```

Gauss Jordan Elimination Method Output

```
add output here
```

LU Decomposition Method

LU Decomposition Method Theory

The LU decomposition method is a numerical technique used to solve a system of linear equations by factorizing the coefficient matrix into a lower triangular matrix and an upper triangular matrix. Instead of solving the system AX = B directly, the matrix A is decomposed into two simpler matrices, which makes the solution process more efficient.

The given system usually involves a square coefficient matrix with non-zero pivot elements. Such a matrix can be decomposed into the product of a lower triangular matrix L and an upper triangular matrix U.

Formula

Matrix Decomposition:

```
A = LU
```

Notation

- `A` : coefficient matrix
- $L = [l_{ij}]$: lower triangular matrix
- $U = [u_{ij}]$: upper triangular matrix
- `X` : solution vector
- `B` : constant vector
- `Y` : intermediate vector

Process

Step 1: Forward Substitution

Solve:

```
LY = B
```

using forward substitution

$$y_i = b_i - \Sigma (l_{ij} y_j); j = 1 \text{ to } i-1$$

Step 2: Backward Substitution

Solve:

```
UX = Y
```

using backward substitution

$$x_i = (1 / u_{ii}) [y_i - \Sigma (u_{ij} x_j)]; j = i+1 \text{ to } n$$

Steps to Apply

1. Decompose the coefficient matrix A into L and U.
2. Solve LY = B using forward substitution.
3. Solve UX = Y using backward substitution.
4. Obtain the solution vector X.

Conditions of Applicability

- The coefficient matrix must be square.
- Pivot elements must be non-zero.
- Pivoting may be required for numerical stability.

Advantages

- More efficient than repeated Gauss elimination.
- Suitable for solving multiple systems with the same coefficient matrix.
- Reduces computational effort after decomposition.

Limitations

- Decomposition fails for singular matrices.
- Numerical errors may occur without pivoting.
- Additional storage is required for L and U matrices.

LU Decomposition Method Code

```
 #include<bits/stdc++.h>
using namespace std;
void print(vector<vector<double>>a)
{
    for(int i=0; i<a.size(); i++)
    {
        for(int j=0; j<a.size(); j++)
        {
            cout<<a[i][j]<<" ";

        }
        cout<<endl;

    }
    cout<<endl;
    return ;
}
int main()
{
//cout<<"ok"<<endl;
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
   int t;
   cin>>t;
   for(int test=1;test<=t;test++){

   cout<<"Test case : "<<test<<endl;
    int n;
   //cout<<"enter the no of eqaution:";
    cin>>n;
    //cout<<"enter the augmented matrix:"<<endl;
    vector<vector<double>>a(n,vector<double>(n)),L(n,vector<double>(n,0)),U(n,vector<double>(n,0));
    vector<double>b(n,0),x(n,0),y(n,0);
    int r=0;
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<n+1; j++)
        {
            if(j==n)
            {
                cin>>b[r];
                r++;
            }
            else cin>>a[i][j];

        }
    }
    bool f=false;
    for(int i=0; i<n; i++)
    {
        for(int j=i; j<n; j++)
        {
            double sum=0;
            for(int k=0; k<i; k++)
            {
                sum+=L[i][k]*U[k][j];
            }
            U[i][j]=a[i][j]-sum;
        }
        for(int j=i; j<n; j++)
        {
            double sum=0;
            for(int k=0; k<i; k++)
            {
                sum+=L[j][k]*U[k][i];
            }
            if(U[i][i]!=0)
            {
                L[j][i]=(a[j][i]-sum)/U[i][i];

            }
            else
            {
                f=true;
            }

        }

    }
    cout<<"L matrix is : "<<endl;
    print(L);
    cout<<"U matrix is :"<<endl;
    print(U);
    for(int i=0; i<n; i++)
    {
        double sum=0;
        for(int j=0; j<i; j++)
        {
            sum+=L[i][j]*y[j];
        }
        y[i]=b[i]-sum;
    }
    cout<<"Y matrix is :"<<endl;
    for(int i=0; i<n; i++)
    {
        cout<<y[i]<<endl;
    }
    cout<<endl;
    if(f)
    {

        if(y[n-1]==0)
        {
            cout<<"INFINITE SOLUTION "<<endl;
        }
        else cout<<"NO solution"<<endl;

    }
    else
    {
        cout<<"Unique solution "<<endl;
        for(int i=n-1; i>=0; i--)
        {
            double sum=0;
            double sum=0;
            for(int j=i+1; j<n; j++)
            {
                sum+=U[i][j]*x[j];
            }
            x[i]=(y[i]-sum)/U[i][i];
        }
        cout<<"The solution is (x1,x2,x3....) :";
        for(int i=0; i<n; i++)cout<<x[i]<<" ";
        cout<<endl;
    }
    cout<<endl<<endl;
   }
    return 0;
}
```

LU Decomposition Method Input

```
 4
5
2 1 -1 3 2 9
1 3 2 -1 1 8
3 2 4 1 -2 20
2 1 3 2 1 17
1 -1 2 3 4 15

3
2 3 -1 5
4 1 2 6
-2 5 3 12

2
1 1 2
2 2 4

2
1 1 2
2 2 5
```

LU Decomposition Method Output

```
 Test case : 1
L matrix is :
1 0 0 0 0
0.5 1 0 0 0
1.5 0.2 1 0 0
1 0 0.8 1 0
0.5 -0.6 0.8 1.71429 1

U matrix is :
2 1 -1 3 2
0 2.5 2.5 -2.5 0
0 0 5 -3 -5
0 0 0 1.4 3
0 0 0 0 1.85714

Y matrix is :
9
3.5
5.8
3.36
2.2

Unique solution
```

```
The solution is (x1,x2,x3....) :5.15385 -1 2.26154 -0.138462 1.18462


Test case : 2
L matrix is :
1 0 0
2 1 0
-1 -1.6 1

U matrix is :
2 3 -1
0 -5 4
0 0 8.4

Y matrix is :
5
-4
10.6

Unique solution
The solution is (x1,x2,x3....) :0.416667 1.80952 1.2619


Test case : 3
L matrix is :
1 0
2 0

U matrix is :
1 1
0 0

Y matrix is :
2
0

INFINITE SOLUTION


Test case : 4
L matrix is :
1 0
2 0

U matrix is :
1 1
0 0

Y matrix is :
2
1

NO solution
```

Matrix Inversion

Matrix Inversion Theory

Add theory here

Matrix Inversion Code

```
// add your code here
```

Matrix Inversion Input

```
add input here
```

Matrix Inversion Output

```
add output here
```

Solution of Non-Linear Equations

Bisection Method

Bisection Method Theory

The bisection method is a numerical technique used to find a real root of a nonlinear equation f(x) = 0. It is based on the property that a continuous function changes sign over an interval containing a root. The method begins with two initial points that lie on opposite sides of the root and repeatedly reduces the interval to improve the approximation.

The given equation is usually a polynomial of the form
$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 = 0$,
which is continuous over the chosen interval. At each iteration, the interval is divided into two equal parts, and the subinterval that contains the sign change is selected. This process is continued until the root is obtained within the desired accuracy.

Basic Formula

Midpoint Calculation

```
c = (a + b) / 2
```

Notation

- `a,b` : endpoints of the interval
- `c` : midpoint of the interval
- `f(x)` : given nonlinear function
-
    - `aₙ, aₙ₋₁, … , a₀` : coefficients of the polynomial
- `n` : number of iterations

Convergence Behavior

With each iteration, the interval length is reduced by half. After n iterations, the maximum possible error becomes `(b − a) / 2ⁿ`. Hence, the bisection method shows linear convergence.

Steps to Apply

1. Select two values a and b such that `f(a)*f(b) < 0`.
2. Calculate the midpoint `c = (a + b) / 2`.
3. Evaluate `f(c)`.
4. Replace a or b based on the sign of `f(c)`.
5. Repeat until the required accuracy is achieved.

Conditions of Applicability

- The function must be continuous over the interval.
- The initial interval must contain a sign change.
- Only real roots can be determined.

Advantages

- Simple and easy to understand.
- Guaranteed convergence under proper conditions.
- Stable and reliable for a wide range of problems.

Limitations

- Converges slowly compared to other methods.
- Not suitable when very high accuracy is required quickly.
- Cannot be used if the initial interval does not contain a sign change.

Bisection Method Code

```cpp
#include <bits/stdc++.h>
using namespace std;

int degree;
vector<double> coeff;

double f(double x)
{
    double val = 0;
    double d = degree;
    int n = coeff.size();
    for (int i = 0; i < n; i++)
    {
        val += coeff[i] * pow(x, d);
        d--;
    }
    return val;
}

int main()
{
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);

    int t;
    cin >> t;

    for (int test = 1; test <= t; test++)
    {
        cout << "Test case : " << test << endl;

        cin >> degree;
        coeff.resize(degree + 1);
        for (int i = 0; i <= degree; i++)
            cin >> coeff[i];

        double xmax = sqrt((coeff[1] / coeff[0]) * (coeff[1] / coeff[0])
                           - 2 * (coeff[2] / coeff[0]));

        double root = 0;
        double e = 0.0001;

        for (double i = -xmax; i <= xmax; i += 0.5)
        {
            double a = i;
            double b = i + 0.5;

            if (f(a) * f(b) < 0)
            {
                root++;
                int it = 0;
                double c;

                do
                {
                    it++;
                    c = (a + b) / 2.0;
                    if (f(a) * f(c) < 0)
                        b = c;
                    else
                        a = c;

                } while (fabs(b - a) > e);

                cout << "the root " << root << " is: " << c << endl;
                cout << "search interval [" << i << "," << i + 0.5 << "]" << endl;
                cout << "iteration is:" << it << endl << endl;
            }
        }
        cout << endl;
    }
    return 0;
}
```

Bisection Method Input

```
 3
4
1 0 -5 0 4
3
1 -6 11 -6
2
1 5 6
```

Bisection Method Output

```
 Test case : 1
the root 1 is: -1.99999
search interval [-2.16228,-1.66228]
iteration is:13

the root 2 is: -0.999985
search interval [-1.16228,-0.662278]
iteration is:13

the root 3 is: 1.00001
search interval [0.837722,1.33772]
iteration is:13

the root 4 is: 2.00001
search interval [1.83772,2.33772]
iteration is:13


Test case : 2
the root 1 is: 0.999981
search interval [0.758343,1.25834]
iteration is:13

the root 2 is: 1.99998
search interval [1.75834,2.25834]
iteration is:13

the root 3 is: 2.99998
search interval [2.75834,3.25834]
iteration is:13


Test case : 3
the root 1 is: -3.00002
search interval [-3.10555,-2.60555]
iteration is:13

the root 2 is: -2.00002
search interval [-2.10555,-1.60555]
iteration is:13
```

False Position Method

False Position Method Theory

The false position method, also known as the regula falsi method, is a numerical technique used to find a real root of a nonlinear equation f(x) = 0. It is based on the fact that a continuous function changes sign over an interval containing a root. The method starts with two initial points that lie on opposite sides of the root and improves the approximation using linear interpolation.

The given equation is usually a polynomial of the form
$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 = 0$,
which is continuous over the chosen interval. Instead of dividing the interval equally, the false position method estimates the root by finding the point where the straight line joining the function values at the endpoints intersects the x-axis. The interval is then updated while keeping the root bracketed.

Basic Formula

Root Approximation

```
c = (a*f(b) - b*f(a)) / (f(b) - f(a))
```

Notation

- `a, b` : endpoints of the interval

- `c` : approximate root
- `f(x)` : given nonlinear function
- $a_n, a_{n-1}, \dots, a_0$: coefficients of the polynomial
- `f(a), f(b)` : function values at the endpoints
- `n` : number of iterations

Convergence Behavior

The false position method generally converges faster than the bisection method because it uses function values to estimate the root. However, the convergence is linear and may slow down if one endpoint remains unchanged over several iterations.

Steps to Apply

1. Select two values a and b such that `f(a)*f(b) < 0`.
2. Compute the approximation
   `c = (a*f(b) - b*f(a)) / (f(b) - f(a))` .
3. Evaluate `f(c)`.
4. Replace a or b based on the sign of `f(c)`.
5. Repeat until the required accuracy is achieved.

Conditions of Applicability

- The function must be continuous over the interval.
- The initial interval must contain a sign change.
- Only real roots can be determined.

Advantages

- Faster convergence than the bisection method in many cases.
- Guaranteed convergence when the root is bracketed.
- Simple to implement.

Limitations

- Convergence may slow down if one endpoint remains fixed.
- Still slower compared to Newton–Raphson type methods.
- Requires a valid initial bracketing interval.

False Position Method Code

```
 #include<bits/stdc++.h>
using namespace std;
int degree;
vector<double>coeff(degree+1);
 double f(double x){
        double val=0;
     double d=degree;
     int n=coeff.size();
     for(int i=0;i<n;i++){
           //  cout<<degree<<endl;
        double t=coeff[i]*(pow(x,d));
        val+=t;
        d--;
     }

   return val;
 }
int main(){
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
   int t;
   cin>>t;
   for(int test=1;test<=t;test++){

   cout<<"Test case : "<<test<<endl;
  // cout<<"please enter the degree of polynomial equation :";
    cin>>degree;
 coeff.resize(degree + 1);
    //cout<<"please enter the coefficient of polynomial :";
    for(int i=0;i<degree+1;i++)cin>>coeff[i];
    double xmax=sqrt((coeff[1]/coeff[0])*(coeff[1]/coeff[0])-2*(coeff[2]/coeff[0]));
  double a,b,c,root=0;
  for(double i=-xmax;i<=xmax;i+=0.5){
        a=i;
      b=i+0.5;
        double fa=f(a),fb=f(b);
   // cout<<a<<" "<<b<<" "<<fa*fb<<endl;
    if(fa*fb<0){
       double it=0;
       root++;
   double e=0.0001;
  do{
        it++;

     c=a-f(a)*((b-a)/(f(b)-f(a)));
   // cout<<"a="<<a<<" f(a)="<<f(a)<<" b="<<b<<" f(b)="<<f(b)<<" c="<<c<<" f(c)="<<f(c)<<endl;
    if(fabs(c)<=e) break;
    if(f(c)*f(a)<0)b=c;
    else a=c;
  }while(fabs(f(c))> e && fabs(b-a)>e);
   cout<<"the root "<< root<<" is: "<<c<<endl;
    cout<<"search interval ["<<i<<","<<i+0.5<<"]"<<endl;
    cout<<"iteration is:"<<it<<endl<<endl;
    }
  }
  cout<<endl;
   }
 return 0;


}
```

False Position Method Input

```
 3
4
1 0 -5 0 4
3
1 -6 11 -6
2
1 5 6
```

False Position Method Output

```
 Test case : 1
the root 1 is: -2
search interval [-2.16228,-1.66228]
iteration is:8

the root 2 is: -1
search interval [-1.16228,-0.662278]
iteration is:3

the root 3 is: 1
search interval [0.837722,1.33772]
iteration is:3

the root 4 is: 2
search interval [1.83772,2.33772]
iteration is:11


Test case : 2
the root 1 is: 1.00001
search interval [0.758343,1.25834]
iteration is:8

the root 2 is: 1.99993
search interval [1.75834,2.25834]
iteration is:2

the root 3 is: 2.99998
search interval [2.75834,3.25834]
iteration is:8


Test case : 3
the root 1 is: -2.99995
search interval [-3.10555,-2.60555]
iteration is:4

the root 2 is: -2.00006
search interval [-2.10555,-1.60555]
iteration is:6
```

Secant Method

Secant Method Theory

The secant method is a numerical technique used to find a real root of a nonlinear equation f(x) = 0. It is similar to the Newton–Raphson method but does not require the computation of derivatives. Instead of using a tangent, it approximates the derivative using a secant line formed by two previous points.

The given equation is usually a polynomial of the form
$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 = 0$.

This method is not a bracketing method and requires two initial guesses close to the root.

Basic Formula

Iterative Formula

$$x_{n+1} = x_n - f(x_n) * (x_n - x_{n-1}) / [f(x_n) - f(x_{n-1})]$$

Notation

- x_n, x_{n-1} : previous approximations of the root
- x_{n+1} : next approximation of the root
- $f(x)$: given nonlinear function
- $a_n, a_{n-1}, \dots, a_0$: coefficients of the polynomial
- n : number of iterations

Convergence Behavior

The secant method converges faster than the bisection and false position methods but slower than the Newton–Raphson method. Its convergence is superlinear but not quadratic.

Steps to Apply

1. Choose two initial guesses x_0 and x_1.
2. Evaluate $f(x_0)$ and $f(x_1)$.
3. Compute the next approximation using the secant formula.
4. Replace x_0 and x_1 with the latest values.
5. Repeat until the required accuracy is achieved.

Conditions of Applicability

- The function must be continuous near the root.
- Two initial guesses are required.
- The difference $f(x_n) - f(x_{n-1})$ must not be zero.
- Only real roots can be determined.

Advantages

- Does not require derivative calculation.
- Faster convergence than Newton–Raphson in some cases where derivatives are costly.
- More efficient than bracketing methods.

Limitations

- Not a bracketing method; convergence is not guaranteed.
- Slower and less stable than Newton–Raphson.
- Sensitive to the choice of initial guesses.

Secant Method Code

```
#include<bits/stdc++.h>
using namespace std;
vector<double>coeff;
void print(vector<double>coeff)
{
    int power=coeff.size()-1;
    bool m=false;
    for(int i=0; i<coeff.size(); i++)
    {
        if(coeff[i]==0)
        {
            power--;
            continue;
        }
        if(power==0)
        {
            if(coeff[i]>0)cout<<"+";
            //else cout<<"-";
            cout<<coeff[i];
            continue;
        }
        if(!m)
        {
            m=true;
            cout<<coeff[i]<<"X^"<<power;
        }
        else
        {
            if(coeff[i]>0)cout<<"+";
            //else cout<<"-";
            cout<<coeff[i]<<"X^"<<power;
        }
        power--;
    }
    cout<<"=0"<<endl;

}
double f(double x)
{
    double val=0;
    double power=coeff.size()-1;
    for(int i=0; i<coeff.size(); i++)
    {
        val+= coeff[i]* pow(x,power);
        power--;
    }
    return val;

}
int main()
{
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
    int t;
    cin>>t;
    for(int test=1; test<=t; test++)
    {

        cout<<"Test case : "<<test<<endl;
        int degree;
//cout<<"enter the degree: ";
        cin>>degree;
//cout<<"enter the coefficient :";

        for(int i=0; i<=degree; i++)
        {
            int x;
            cin>>x;
            coeff.push_back(x);
        }
        cout<<"equation is: ";
        print(coeff);
        cout<<endl;
        double xmax=0,e=0.001;
        for(int i=0; i<coeff.size(); i++)
        {
            double temp=coeff[i]/coeff[0];
            xmax=max(xmax,temp);
        }
        xmax++;
        double c=-xmax;
        while(c<=xmax)
        {
            double x0=c;
            double x1=c+0.45;
            double fx0=f(x0),fx1=f(x1);
            if(fx0*fx1<0)
            {
                cout<<"search interval is: ["<<x0<<","<<x1<<"]"<<endl;
                int it=0;
                do
                {
                    double x2=x1-f(x1)*((x1-x0)/(fx1-fx0));
                    it++;
                    double fx2=f(x2);
                    if(abs(x1-x2)<e && fx2<e)
                    {
                        cout<<"root is ="<<x2<<endl<<"iteration is = "<<it<<endl<<endl;
                        break;
                    }
                    x0=x1;
                    x1=x2;
                    fx0=fx1;
                    fx1=fx2;

                }
                while(1);
            }
            c=c+0.45;
        }
        coeff.clear();
       cout<<endl<<endl;

    }
        return 0;
    }
```

Secant Method Input

```
 3
4
1 0 -5 0 4
4
1 -8 18 -8 -15
5
1 0 -7 0 10 -4
```

Secant Method Output

```
 Test case : 1
equation is: 1X^4-5X^2+4=0

search interval is: [-2.3,-1.85]
root is =-2
iteration is = 5

search interval is: [-1.4,-0.95]
root is =-1
iteration is = 3

search interval is: [0.85,1.3]
root is =1
iteration is = 4

search interval is: [1.75,2.2]
root is =2
iteration is = 5



Test case : 2
equation is: 1X^4-8X^3+18X^2-8X^1-15=0

search interval is: [-1,-0.55]
root is =-0.645751
iteration is = 4

search interval is: [4.4,4.85]
root is =4.64575
iteration is = 4



Test case : 3
equation is: 1X^5-7X^3+10X^1-4=0

search interval is: [0.25,0.7]
root is =0.470683
iteration is = 6

search interval is: [0.7,1.15]
root is =0.99999
iteration is = 5

search interval is: [2.05,2.5]
root is =2.34292
iteration is = 5
```

Newton Raphson Method

Newton Raphson Method Theory

The Newton–Raphson method is a numerical technique used to find a real root of a nonlinear equation f(x) = 0. Unlike bracketing methods, it starts from a single initial guess and uses the tangent to the curve to obtain successive approximations of the root. The method is based on the idea that the tangent at a point near the root will intersect the x-axis close to the actual root.

The given equation is usually a polynomial of the form
$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = 0$.
This method is not a bracketing method, so convergence depends strongly on the choice of the initial guess.

Basic Formula

Iterative Formula

```
$x_{n+1} = x_n - f(x_n) / f'(x_n)$
```

Notation

- x_n : current approximation
- x_{n+1} : next approximation
- $f(x)$: given nonlinear function
- $f'(x)$: derivative of the function
- $a_n, a_{n-1}, \dots, a_0$: coefficients of the polynomial
- n : number of iterations

Convergence Behavior

When the initial guess is chosen close to the root, the Newton–Raphson method converges very rapidly and exhibits quadratic convergence. However, poor initial guesses may lead to slow convergence or divergence.

Steps to Apply

1. Choose an initial guess x_0.
2. Compute the derivative f'(x).
3. Calculate the next approximation using $x_{n+1} = x_n - f(x_n) / f'(x_n)$.
4. Repeat until the required accuracy is achieved.

Conditions of Applicability

- The function must be differentiable near the root.
- The derivative should not be zero at the root.
- A good initial guess is required for convergence.
- Only real roots can be determined.

Advantages

- Very fast convergence near the root.
- Requires only one initial guess.
- Highly efficient compared to bracketing methods.

Limitations

- Not a bracketing method; convergence is not guaranteed.
- Fails if the derivative becomes zero or very small.
- Sensitive to the choice of the initial guess.

Newton Raphson Method Code

```
 #include<bits/stdc++.h>
using namespace std;
int degree;
vector<double>coeff;
void print()
{
    int d=degree;
    bool m=true;
    for(int i=0; i<degree+1; i++)
    {
        if(m)
        {
            cout<<coeff[i]<<"X^"<<d;
            m=false;
            d--;
            continue;

        }
        if(coeff[i]==0)
        {
            d--;
            continue;
        }
        if(coeff[i]>0)cout<<"+";
        if(i!=degree)cout<<coeff[i]<<"X^"<<d;
        else cout<<coeff[i];
        d--;
    }
    cout<<"=0"<<endl;
}
double f(double x)
{

    double val=0;
    double deg=coeff.size()-1;
    for(int i=0; i<coeff.size(); i++)
    {
        val+= coeff[i]* pow(x,deg);
        deg--;
    }
    return val;
}
double df(double x)
{
    double val=0;
    double deg=coeff.size()-1;
    for(int i=0; i<coeff.size()-1; i++)
    {
        val+=coeff[i] *deg* pow(x,deg-1);
        deg--;
    }
    return val;

}
int main()
{
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
    int t;
    cin>>t;
    for(int test=1; test<=t; test++)
    {

        cout<<"Test case : "<<test<<endl;
        //cout<<"enter degree:";
        cin>>degree;
       // cout<<"enter the coefficient:";
        coeff.resize(degree+1);
        for(int i=0; i<degree+1; i++)
        {
            cin>>coeff[i];
        }
        cout<<"equation is: ";
        print();
        double xmax=sqrt((coeff[1]/coeff[0])*(coeff[1]/coeff[0])-2*(coeff[2]/coeff[0]));
        double c=-xmax;
        double num=0,e=0.00001;
        while(c<=xmax)
        {
            double x0=c,x1=c+0.65;
            double fx0=f(x0),fx1=f(x1);
            if(fx0*fx1<0)
            {
                num++;
                cout<<"search interval is ["<<x0<<","<<x1<<"]"<<endl;
                int it=0;
                do
                {
                    double df1=df(x1);
                    double x2=x1-(fx1/df1);
                    double fx2=f(x2);
                    it++;
                    if(abs(x2-x1)<e|| abs(fx2-fx1)<e)
                    {
                        cout<<"the root "<< num<<" is: "<<x2<<endl;
                        cout<<"iteration is:"<<it<<endl<<endl;
                        break;
                    }
                    x1=x2;
                    fx1=fx2;

                }
                while(1);
            }
            c=c+0.65;
        }
        cout<<endl<<endl;
        }

        return 0;
    }
```

Newton Raphson Method Input

```
 3
4
1 0 -5 0 4
3
1 -6 11 -6
2
1 5 6
```

Newton Raphson Method Output

```
 Test case : 1
equation is: 1X^4-5X^2+4=0
search interval is [-2.51228,-1.86228]
the root 1 is: -2
iteration is:5

search interval is [-1.21228,-0.562278]
the root 2 is: -1
iteration is:4

search interval is [0.737722,1.38772]
the root 3 is: 1
iteration is:4

search interval is [1.38772,2.03772]
the root 4 is: 2
iteration is:3



Test case : 2
equation is: 1X^3-6X^2+11X^1-6=0
search interval is [0.808343,1.45834]
the root 1 is: 3
iteration is:8

search interval is [1.45834,2.10834]
the root 2 is: 2
iteration is:3

search interval is [2.75834,3.40834]
the root 3 is: 3
iteration is:5



Test case : 3
equation is: 1X^2+5X^1+6=0
search interval is [-3.60555,-2.95555]
the root 1 is: -3
iteration is:3

search interval is [-2.30555,-1.65555]
the root 2 is: -2
iteration is:5
```

Solution of Interpolation

Newton's Forward Interpolation Method

Newton's Forward Interpolation Method Theory

Newton’s Forward Interpolation Method is a numerical technique used to estimate unknown values of a function from equally spaced tabulated data. It constructs an interpolation polynomial using finite forward differences of the function values. It is most accurate when the value to be interpolated lies near the beginning of the data.

Basic Idea

Given a set of equally spaced data points:

$x_0, x_1, x_2, \ldots, x_n$ and corresponding function values:

$f(x_0), f(x_1), f(x_2), ..., f(x_n)$

We calculate **forward differences** ($\Delta y, \Delta^2 y, \Delta^3 y, ...$) and use them to form the interpolation polynomial.

Forward Difference Table

The forward difference table organizes the differences systematically:

x	f(x)	Δf(x)	Δ²f(x)	Δ³f(x)	...
x_0	f_0	Δf_0	$\Delta^2 f_0$	$\Delta^3 f_0$	...
x_1	f_1	Δf_1	$\Delta^2 f_1$	...	
x_2	f_2	Δf_2	...		
x_3	f_3	...			
...	...				

Where:

$\Delta f_0 = f_1 - f_0$
$\Delta^2 f_0 = \Delta f_1 - \Delta f_0$
$\Delta^3 f_0 = \Delta^2 f_1 - \Delta^2 f_0$
and so on.

Formula

Let $h = x_1 - x_0$ (equal spacing) Let $u = (x - x_0) / h$

Newton Forward Interpolation Polynomial:

$P(x) = f_0 + u\Delta f_0 + u(u-1)/2!\ \Delta^2 f_0 + u(u-1)(u-2)/3!\ \Delta^3 f_0 + ... + u(u-1)...(u-n+1)/n!\ \Delta^n f_0$

Steps to Apply

1. Construct the forward difference table from the given data.
2. Compute $\Delta f_0, \Delta^2 f_0, ..., \Delta^n f_0$.
3. Calculate $u = (x - x_0)/h$ for the required x.
4. Substitute into the interpolation polynomial to find P(x).

Conditions of Applicability

- Data points must be equally spaced.
- The value to be interpolated should lie near the beginning of the table.
- Function should be continuous over the interval.

Advantages

- Simple and systematic for equally spaced data.
- Forward difference table reduces repeated calculations.
- Good accuracy near the beginning of the data set.

Limitations

- Not suitable for unequally spaced data.
- Accuracy decreases for values far from x_0.
- Higher-order differences may introduce rounding errors.

Newton's Forward Interpolation Method Code

```
 #include <bits/stdc++.h>
using namespace std;

double forward_interpolation(vector<double> &x, vector<double> &y, int n, double xn)
{
    vector<vector<double>> dif(n, vector<double>(n, 0));
    for (int i = 0; i < n; i++)
        dif[i][0] = y[i];

    for (int j = 1; j < n; j++)
        for (int i = 0; i < n - j; i++)
            dif[i][j] = dif[i + 1][j - 1] - dif[i][j - 1];
    cout << "Forward difference table:" << endl;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - i; j++)
            cout << setw(12) << dif[i][j];
        cout << endl;
    }
    cout << endl;

    double h = x[1] - x[0];
    double u = (xn - x[0]) / h;

    double res = y[0];
    double term = 1.0;
    double fact = 1.0;

    for (int i = 1; i < n; i++)
    {
        term *= (u - (i - 1));
        fact *= i;
        res += (term * dif[0][i]) / fact;
    }
    return res;
}

int main()
{
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);

    cout << fixed << setprecision(3);

    int t;
    cin >> t;
    for (int test = 1; test <= t; test++)
    {
        cout << "Test case : " << test << endl;

        int n;
        cin >> n;
        vector<double> x(n), y(n);

        double sum = 0;
        for (int i = 0; i < n; i++)
        {
            int x1, x2, yf;
            cin >> x1 >> x2 >> yf;
            x[i] = x2;
            sum += yf;
            y[i] = sum;
        }

        int d1;
        double xn;
        cin >> d1 >> xn;

        double res = forward_interpolation(x, y, n, xn);
        cout << "the value of Y at " << d1 << "-" << xn << " : " << res << endl;

        double ex1, ex2, ey;
        cin >> ex1 >> ex2 >> ey;
        sum += ey;
        x.push_back(ex2);
        y.push_back(sum);
        double res2 = forward_interpolation(x, y, n + 1, xn);
        cout << "error is:" << fabs(res2 - res) << endl
             << endl;
    }
    return 0;
}
```

Newton's Forward Interpolation Method Input

```
 3
5
30 40 31
40 50 42
50 60 51
60 70 55
70 80 31
40 45
80 90 25

4
10 20 15
20 30 20
30 40 18
40 50 22
20 25
50 60 30

6
0 10 5
10 20 9
20 30 14
30 40 20
40 50 27
50 60 35
10 15
60 70 45
```

Newton's Forward Interpolation Method Output

```
 Test case : 1
Forward difference table:
      31.000      42.000       9.000      -5.000     -23.000
      73.000      51.000       4.000     -28.000
     124.000      55.000     -24.000
     179.000      31.000
     210.000

the value of Y at 40-45.000 : 51.461
Forward difference table:
      31.000      42.000       9.000      -5.000     -23.000      69.000
      73.000      51.000       4.000     -28.000      46.000
     124.000      55.000     -24.000      18.000
     179.000      31.000      -6.000
     210.000      25.000
     235.000

error is:1.887

Test case : 2
Forward difference table:
      15.000      20.000      -2.000       6.000
      35.000      18.000       4.000
      53.000      22.000
      75.000

the value of Y at 20-25.000 : 25.625
Forward difference table:
      15.000      20.000      -2.000       6.000      -2.000
      35.000      18.000       4.000       4.000
      53.000      22.000       8.000
      75.000      30.000
     105.000

error is:0.078

Test case : 3
Forward difference table:
       5.000       9.000       5.000       1.000       0.000       0.000
      14.000      14.000       6.000       1.000       0.000
      28.000      20.000       7.000       1.000
      48.000      27.000       8.000
      75.000      35.000
     110.000

the value of Y at 10-15.000 : 8.938
Forward difference table:
       5.000       9.000       5.000       1.000       0.000       0.000       1.000
      14.000      14.000       6.000       1.000       0.000       1.000
      28.000      20.000       7.000       1.000       1.000
      48.000      27.000       8.000       2.000
      75.000      35.000      10.000
     110.000      45.000
     155.000

error is:0.021
```

Newton's Backward Interpolation Method

Newton's Backward Interpolation Method Theory

Newton’s Backward Interpolation Method is a numerical technique used to estimate unknown values of a function from equally spaced tabulated data. It constructs an interpolation polynomial using finite backward differences of the function values. It is most accurate when the value to be interpolated lies near the end of the data.

Basic Idea

Given a set of equally spaced data points:

$x_0, x_1, x_2, ..., x_n$ and corresponding function values:

$f(x_0), f(x_1), f(x_2), ..., f(x_n)$

We calculate backward differences ($\nabla y, \nabla^2 y, \nabla^3 y, ...$) and use them to form the interpolation polynomial.

Backward Difference Table

The backward difference table organizes the differences systematically:

x	f(x)	∇f(x)	∇²f(x)	∇³f(x)	...
x_0	f_0				
x_1	f_1	∇f_1			
x_2	f_2	∇f_2	$\nabla^2 f_2$		
x_3	f_3	∇f_3	$\nabla^2 f_3$	$\nabla^3 f_3$	
...	...	...	...	...	...
x_n	f_n	∇f_n	$\nabla^2 f_n$	$\nabla^3 f_n$	...

Where:

$\nabla f_n = f_n - f_{n-1}$
$\nabla^2 f_n = \nabla f_n - \nabla f_{n-1}$
$\nabla^3 f_n = \nabla^2 f_n - \nabla^2 f_{n-1}$
and so on.

Formula

Let $h = x_1 - x_0$ (equal spacing) Let $u = (x - x_n) / h$

Newton Backward Interpolation Polynomial:

$P(x) = f_n + u\nabla f_n + u(u+1)/2!\ \nabla^2 f_n + u(u+1)(u+2)/3!\ \nabla^3 f_n + ... + u(u+1)...(u+n-1)/n!\ \nabla^n f_n$

Steps to Apply

1. Construct the backward difference table from the given data.
2. Compute $\nabla f_n, \nabla^2 f_n, ..., \nabla^n f_n$.
3. Calculate $u = (x - x_n)/h$ for the required x.
4. Substitute into the interpolation polynomial to find P(x).

Conditions of Applicability

- Data points must be equally spaced.
- The value to be interpolated should lie near the end of the table.
- Function should be continuous over the interval.

Advantages

- Simple and systematic for equally spaced data.
- Backward difference table reduces repeated calculations.
- Good accuracy near the end of the data set.

Limitations

- Not suitable for unequally spaced data.
- Accuracy decreases for values far from x_n.
- Higher-order differences may introduce rounding errors.

Newton's Backward Interpolation Method Code

```
 #include<bits/stdc++.h>
using namespace std;
```

```
double error(vector<double>&x,vector<double>&y,double val)
{
    int n=x.size();
    vector<vector<double>>dif(n,vector<double>(n));
    for(int i=0;i<n;i++) dif[i][0]=y[i];
    for(int j=1;j<n;j++)
        for(int i=0;i<n-j;i++)
            dif[i][j]=(dif[i+1][j-1]-dif[i][j-1])/(x[i+j]-x[i]);

    double del=1.0;
    for(int i=1;i<n;i++)
        del*= (val-x[i-1]);

    return dif[0][n-1]*del;
}

int main()
{
    freopen("input.txt","r",stdin);
    freopen("output.txt","w",stdout);

    cout<<fixed<<setprecision(3);

    int t;
    cin>>t;
    for(int test=1;test<=t;test++)
    {
        cout<<"Test case : "<<test<<endl;

        int n;
        cin>>n;
        vector<double>x(n),y(n);
        for(int i=0;i<n;i++)
            cin>>x[i]>>y[i];

        vector<vector<double>>dif(n,vector<double>(n,0));
        for(int i=0;i<n;i++)
            dif[i][0]=y[i];

        for(int j=1;j<n;j++)
            for(int i=n-1;i>=j;i--)
                dif[i][j]=dif[i][j-1]-dif[i-1][j-1];

        cout<<"Backward difference table:"<<endl;
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<=i;j++)
                cout<<setw(12)<<dif[i][j];
            cout<<endl;
        }
        cout<<endl;

        double xx;
        cin>>xx;

        double h=x[n-1]-x[n-2];
        double v=(xx-x[n-1])/h;

        double res=y[n-1];
        double term=1.0;
        double fact=1.0;

        for(int i=1;i<n;i++)
        {
            term*=(v+i-1);
            fact*=i;
            res+=(term*dif[n-1][i])/fact;
        }

        cout<<"answer is : "<<res<<endl;

        double nx,ny;
        cin>>nx>>ny;
        x.push_back(nx);
        y.push_back(ny);

        cout<<"error is : "<<fabs(error(x,y,xx))<<endl<<endl;
    }
    return 0;
}
```

Newton's Backward Interpolation Method Input

```
 3
5
10 5
20 9
30 14
40 20
50 27
45
60 35

4
1 2
2 4
3 9
4 16
3
5 25

6
0 1
1 1
2 2
3 6
4 24
5 120
4
6 720
```

Newton's Backward Interpolation Method Output

```
 Test case : 1
Backward difference table:
       5.000
       9.000       4.000
      14.000       5.000       1.000
      20.000       6.000       1.000       0.000
      27.000       7.000       1.000       0.000       0.000

answer is : 23.375
error is : 0.000

Test case : 2
Backward difference table:
       2.000
       4.000       2.000
       9.000       5.000       3.000
      16.000       7.000       2.000      -1.000

answer is : 9.000
error is : 0.000

Test case : 3
Backward difference table:
       1.000
       1.000       0.000
       2.000       1.000       1.000
       6.000       4.000       3.000       2.000
      24.000      18.000      14.000      11.000       9.000
     120.000      96.000      78.000      64.000      53.000      44.000

answer is : 24.000
error is : 0.000
```

Newton's Divided Difference Method

Newton's Divided Difference Method Theory

Newton’s Divided Difference Interpolation Method is a numerical technique used to estimate unknown values of a function from a set of **unequally spaced data points**. It constructs an interpolation polynomial using **divided differences** of the function values. This method generalizes Newton's forward and backward methods and works for both equally and unequally spaced data.

Basic Idea

Given a set of data points:

$x_0, x_1, x_2, ..., x_n$ and corresponding function values:

$f(x_0), f(x_1), f(x_2), ..., f(x_n)$

We calculate **divided differences** ($f[x_i, x_j], f[x_i, x_j, x_k], ...$) recursively and use them to form the interpolation polynomial. Divided differences generalize forward/backward differences to unequally spaced points.

Divided Difference Table

The divided difference table organizes the differences systematically:

x	f(x)	1st Divided Difference	2nd Divided Difference	3rd Divided Difference	...
x_0	f_0	$f[x_0,x_1]$	$f[x_0,x_1,x_2]$	$f[x_0,x_1,x_2,x_3]$	...
x_1	f_1	$f[x_1,x_2]$	$f[x_1,x_2,x_3]$	...	
x_2	f_2	$f[x_2,x_3]$	...		

x	f(x)	1st Divided Difference	2nd Divided Difference	3rd Divided Difference	...
x_3	f_3	...			
...	...				...

Where:

1st Divided Difference: $f[x_i, x_{i+1}] = (f(x_{i+1}) - f(x_i)) / (x_{i+1} - x_i)$
2nd Divided Difference: $f[x_i, x_{i+1}, x_{i+2}] = (f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]) / (x_{i+2} - x_i)$
3rd Divided Difference: $f[x_i, x_{i+1}, x_{i+2}, x_{i+3}] = (f[x_{i+1}, x_{i+2}, x_{i+3}] - f[x_i, x_{i+1}, x_{i+2}]) / (x_{i+3} - x_i)$
and so on.

Formula

Newton Divided Difference Polynomial:

$P(x) = f(x_0) + (x - x_0)f[x_0,x_1] + (x - x_0)(x - x_1)f[x_0,x_1,x_2] + (x - x_0)(x - x_1)(x - x_2)f[x_0,x_1,x_2,x_3] + ... + (x - x_0)(x - x_1)...(x - x_{n-1})f[x_0,x_1,...,x_n]$

Steps to Apply

1. Arrange the given data points in a table.
2. Construct the divided difference table recursively.
3. Use the top row of divided differences to construct the interpolation polynomial.
4. Substitute the required value of x into the polynomial to find P(x).

Conditions of Applicability

- Data points can be equally or unequally spaced.
- Function should be continuous over the interval.

Advantages

- Works for unequally spaced data points.
- Systematic and can be extended to higher orders easily.
- Provides an explicit polynomial for interpolation.

Limitations

- Computationally more intensive for large datasets.
- Accuracy may decrease for very high-order polynomials due to rounding errors.

Error in Divided Difference Interpolation

In Newton's Divided Difference interpolation, the interpolation polynomial of degree (n-1) is:

P_{n-1}(x) = $y_0 + (x - x_0)\Delta_1 + (x - x_0)(x - x_1)\Delta_2 + ...$

The theoretical interpolation error at a point x = val is:

$E(x) = f[x_0, x_1, ..., x_{n-1}] * (x - x_0)(x - x_1)...(x - x_{n-2})$

Where:

- $f[x_0, x_1, ..., x_{n-1}]$ is the highest-order divided difference.
- The product term $(x - x_0)(x - x_1)...(x - x_{n-2})$ increases if val is far from known data points, which increases the error.

Newton's Divided Difference Method Code

```
 #include<bits/stdc++.h>
using namespace std;
double error(vector<double>&x,vector<double>&y,double val)
{
    int n=x.size();
    vector<vector<double>>dif(n,vector<double>(n));
    for(int i=0; i<n; i++) dif[i][0]=y[i];
    for(int j=1; j<n; j++)
    {
        for(int i=0; i<n-j; i++)
        {
            dif[i][j]=(dif[i+1][j-1]-dif[i][j-1])/(x[i+j]-x[i]);
        }
    }
    double del=1.0;
    for(int i=1; i<n; i++)
    {
        del=del*(val-x[i-1]);
```

```cpp
    }
    double e=dif[0][n-1]*del;
    return e;

}
double ddi(vector<double>x,vector<double>y,double val)
{
    int n=x.size();
    vector<vector<double>>tb(n,vector<double>(n,0));
    for(int i=0; i<n; i++) tb[i][0]=y[i];
    for(int j=1; j<n; j++)
    {
        for(int i=0; i<n-j; i++)
        {
            tb[i][j]=(tb[i+1][j-1]-tb[i][j-1])/(x[i+j]-x[i]);
        }
    }
    cout<<"divided difference table is :"<<endl;
    for(int j=0; j<n; j++)
    {
        for(int i=0; i<n-j; i++)
        {
          if(i<n-j-1)  cout<<tb[i][j]<<setw(12);
          else cout<<tb[i][j];
        }
        cout<<endl;
    }
    cout<<endl;
    double res=y[0],del=1.0;

    for(int i=1; i<n; i++)
    {
        del=del*(val-x[i-1]);
        res+=del*tb[0][i];
    }
    return res;
}
int main()
{
//cout<<"ok"<<endl;
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
cout << fixed << setprecision(3);
    int t;
    cin>>t;
    for(int test=1; test<=t; test++)
    {


        cout<<"Test case : "<<test<<endl;
        int n;
        cin>>n;
        vector<double>x(n),y(n);
        for(int i=0; i<n; i++)
        {
            cin>>x[i]>>y[i];
        }
//cout<<"enter the x:";
        double val;
        cin>>val;
        double res1=ddi(x,y,val);
        cout<<"value of y at x= "<<val<<" is :"<<res1<<endl;
        cout<<"error is :"<<error(x,y,val)<<"%"<<endl<<endl;
    }
    return 0;
}
```

Newton's Divided Difference Method Input

```
 3
5
1 1
2.5 2
4 4
6 5
7.5 7
3.3

4
0 0
1 2
3 10
4.5 20
2.7

6
0 1
0.5 2
1.7 5
3 6
4.2 10
6 20
3.5
```

Newton's Divided Difference Method Output

```
 Test case : 1
divided difference table is :
1.000       2.000       4.000       5.000       7.000
0.667       1.333       0.500       1.333
0.222      -0.238       0.238
-0.092       0.095
0.029

value of y at x= 3.300 is :3.161
error is :0.100%

Test case : 2
divided difference table is :
0.000       2.000      10.000      20.000
2.000       4.000       6.667
0.667       0.762
0.021

value of y at x= 2.700 is :8.431
error is :-0.029%

Test case : 3
divided difference table is :
1.000       2.000       5.000       6.000      10.000      20.000
2.000       2.500       0.769       3.333       5.556
0.294      -0.692       1.026       0.741
-0.329       0.464      -0.066
0.189      -0.096
-0.048

value of y at x= 3.500 is :6.973
error is :0.315%
```

Solution of Numerical Differentiation

Numerical Differentiation by Forward Interpolation Method

Numerical Differentiation by Forward Interpolation Method Theory

Numerical Differentiation using Forward Interpolation is a technique to approximate derivatives of a function from equally spaced tabulated data. It uses finite forward differences to construct formulas for the first, second, or higher-order derivatives. This method is most accurate when the derivative is evaluated near the beginning of the data.

Basic Idea

Given a set of equally spaced data points:

$x_0, x_1, x_2, ..., x_n$ and corresponding function values:

$f(x_0), f(x_1), f(x_2), ..., f(x_n)$

We use forward differences ($\Delta y, \Delta^2 y, \Delta^3 y, ...$) to approximate derivatives at points near x_0. The first derivative at x_0 can be approximated as:

$f'(x_0) \approx (\Delta f_0)/h - (\Delta^2 f_0)/(2h) + (\Delta^3 f_0)/(3h) - ...$

The second derivative at x_0 can be approximated as:

$f''(x_0) \approx (\Delta^2 f_0)/h^2 - (\Delta^3 f_0)/h^2 + (11\Delta^4 f_0)/(12h^2) - ...$

Formula

First derivative: $f'(x_0) \approx \Delta f_0/h - \Delta^2 f_0/(2h) + \Delta^3 f_0/(3h) - ...$
Second derivative: $f''(x_0) \approx \Delta^2 f_0/h^2 - \Delta^3 f_0/h^2 + 11\Delta^4 f_0/(12h^2) - ...$

Steps to Apply

1. Compute forward differences $\Delta f_0, \Delta^2 f_0, ..., \Delta^n f_0$ from the given data.
2. Choose the point near the beginning of the data for differentiation.
3. Substitute the forward differences into the formulas for the first or second derivative.
4. Compute the derivative using the step size h.

Conditions of Applicability

- Data points must be equally spaced.
- The value for differentiation should lie near the beginning of the table.
- Function should be continuous over the interval.

Advantages

- Simple and systematic for equally spaced data.
- Forward differences reduce repeated calculations.
- Good accuracy near the beginning of the data set.

Limitations

- Not suitable for unequally spaced data.
- Accuracy decreases for points far from x_0.
- Higher-order differences may introduce rounding errors.

Numerical Differentiation by Forward Interpolation Method Code

```
#include <bits/stdc++.h>
using namespace std;

double f(double x)
{
    return pow(x, 5) + 4 * pow(x, 4) + 1;
}

double f_prime(double x)
double f_prime(double x)
{
    return 5 * pow(x, 4) + 16 * pow(x, 3);
}

double f_double_prime(double x)
{
    return 20 * pow(x, 3) + 48 * pow(x, 2);
}

int main()
{
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
    int test;
    cin >> test;

    for (int t = 1; t <= test; t++)
    {
        cout << "Testcase: " << t << endl;

        int n;
        double a, b, X;

        cin >> n;
        cin >> a >> b;
        cin >> X;

        double h = (b - a) / (n - 1);

        vector<double> x(n), y(n);
        for (int i = 0; i < n; i++)
        {
            x[i] = a + i * h;
            y[i] = f(x[i]);
        }

        vector<vector<double>> diff(n);
        diff[0] = y;

        for (int i = 1; i < n; i++)
        {
            diff[i].resize(n - i);
            for (int j = 0; j < n - i; j++)
                diff[i][j] = diff[i - 1][j + 1] - diff[i - 1][j];
        }

        cout << fixed << setprecision(6);
        cout << "Forward Difference Table\n\n";

        for (int col = 0; col < n; col++)
        {
            for (int row = 0; row <= col; row++)
                cout << setw(14) << diff[row][col - row];
            cout << "\n";
        }

        double u = (X - x[0]) / h;

        double dy_dx = 0.0;
        dy_dx += diff[1][0];
        if (n >= 3)
            dy_dx += ((2 * u - 1) / 2.0) * diff[2][0];
        if (n >= 4)
            dy_dx += ((3 * u * u - 6 * u + 2) / 6.0) * diff[3][0];
        if (n >= 5)
```

```
            dy_dx += ((4 * u * u * u - 18 * u * u + 22 * u - 6) / 24.0) * diff[4][0];
        dy_dx /= h;

        double d2y_dx2 = 0.0;
        if (n >= 3)
            d2y_dx2 += diff[2][0];
        if (n >= 4)
            d2y_dx2 += (u - 1) * diff[3][0];
        if (n >= 5)
            d2y_dx2 += ((6 * u * u - 6 * u - 1) / 12.0) * diff[4][0];
        d2y_dx2 /= (h * h);

        double exact1 = f_prime(X);
        double exact2 = f_double_prime(X);

        cout << "\nNumerical f'(x)  = " << dy_dx << "\n";
        cout << "Exact f'(x)      = " << exact1 << "\n";
        cout << "Numerical f''(x) = " << d2y_dx2 << "\n";
        cout << "Exact f''(x)     = " << exact2 << "\n";

        cout << "Error f'(x)  (%) = " << fabs((exact1 - dy_dx) / exact1) * 100 << "\n";
        cout << "Error f''(x) (%) = " << fabs((exact2 - d2y_dx2) / exact2) * 100 << "\n";
    }

    return 0;
}
```

Numerical Differentiation by Forward Interpolation Method Input

```
2
6
0.5 1.0
0.65
7
1.0 1.6
1.20
```

Numerical Differentiation by Forward Interpolation Method Output

```
Testcase: 1
Forward Difference Table

      1.281250
      1.596160      0.314910
      2.128470      0.532310      0.217400
      2.966080      0.837610      0.305300      0.087900
      4.214890      1.248810      0.411200      0.105900      0.018000
      6.000000      1.785110      0.536300      0.125100      0.019200      0.001200

Numerical f'(x)  = 5.286475
Exact f'(x)      = 5.286531
Numerical f''(x) = 26.660000
Exact f''(x)     = 25.772500
Error f'(x)  (%) = 0.001064
Error f''(x) (%) = 3.443593
Testcase: 2
Forward Difference Table

      6.000000
      8.466910      2.466910
     11.782720      3.315810      0.848900
     16.137330      4.354610      1.038800      0.189900
     21.744640      5.607310      1.252700      0.213900      0.024000
     28.843750      7.099110      1.491800      0.239100      0.025200      0.001200
     37.700160      8.856410      1.757300      0.265500      0.026400      0.001200      0.000000

Numerical f'(x)  = 38.015600
Exact f'(x)      = 38.016000
Numerical f''(x) = 106.080000
Exact f''(x)     = 103.680000
Error f'(x)  (%) = 0.001052
Error f''(x) (%) = 2.314815
```

Numerical Differentiation by Backward Interpolation Method

Numerical Differentiation by Backward Interpolation Method Theory

Numerical Differentiation using Backward Interpolation is a technique to approximate derivatives of a function from equally spaced tabulated data. It uses finite backward differences to construct formulas for the first, second, or higher-order derivatives. This method is most accurate when the derivative is evaluated near the end of the data.

Basic Idea

Given a set of equally spaced data points:

$x_0, x_1, x_2, ..., x_n$ and corresponding function values:

$f(x_0), f(x_1), f(x_2), ..., f(x_n)$

We use backward differences ($\nabla y, \nabla^2 y, \nabla^3 y, ...$) to approximate derivatives at points near x_n. The first derivative at x_n can be approximated as:

$f'(x_n) \approx (\nabla f_n)/h + (\nabla^2 f_n)/(2h) + (\nabla^3 f_n)/(3h) + ...$

The second derivative at x_n can be approximated as:

$f''(x_n) \approx (\nabla^2 f_n)/h^2 + (\nabla^3 f_n)/h^2 + (11\nabla^4 f_n)/(12h^2) - ...$

Formula

First derivative: $f'(x_n) \approx \nabla f_n/h + \nabla^2 f_n/(2h) + \nabla^3 f_n/(3h) + ...$
Second derivative: $f''(x_n) \approx \nabla^2 f_n/h^2 + \nabla^3 f_n/h^2 + 11\nabla^4 f_n/(12h^2) - ...$

Steps to Apply

1. Compute backward differences $\nabla f_n, \nabla^2 f_n, \ldots, \nabla^n f_n$ from the given data.
2. Choose the point near the end of the data for differentiation.
3. Substitute the backward differences into the formulas for the first or second derivative.
4. Compute the derivative using the step size h.

Conditions of Applicability

- Data points must be equally spaced.
- The value for differentiation should lie near the end of the table.
- Function should be continuous over the interval.

Advantages

- Simple and systematic for equally spaced data.
- Backward differences reduce repeated calculations.
- Good accuracy near the end of the data set.

Limitations

- Not suitable for unequally spaced data.
- Accuracy decreases for points far from x_n.
- Higher-order differences may introduce rounding errors.

Numerical Differentiation by Backward Interpolation Method Code

```
#include <bits/stdc++.h>
using namespace std;

double f(double x)
{
    return pow(x, 5) + 4 * pow(x, 4) + 1;
}

double f_prime(double x)
{
    return 5 * pow(x, 4) + 16 * pow(x, 3);
}

double f_double_prime(double x)
{
    return 20 * pow(x, 3) + 48 * pow(x, 2);
}

int main()
{
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);

    int test;
    cin >> test;

    for (int t = 1; t <= test; t++)
    {
        cout << "Testcase " << t << " : \n";

        int n;
        double a, b, X;
        cin >> n >> a >> b >> X;

        double h = (b - a) / (n - 1);

        vector<double> x(n), y(n);
        for (int i = 0; i < n; i++)
        {
            x[i] = a + i * h;
            y[i] = f(x[i]);
        }
```

```cpp
        vector<vector<double>> diff(n, vector<double>(n, 0.0));
        for (int i = 0; i < n; i++)
            diff[i][0] = y[i];

        for (int j = 1; j < n; j++)
            for (int i = n - 1; i >= j; i--)
                diff[i][j] = diff[i][j - 1] - diff[i - 1][j - 1];

        cout << "\nDifference Table:\n\n";
        cout << fixed << setprecision(6);

        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n - i; j++)
                cout << setw(14) << diff[i][j];
            cout << "\n";
        }

        double u = (X - x[n - 1]) / h;

        double dy_dx = diff[n - 1][1];
        if (n >= 3)
            dy_dx += (2 * u + 1) * diff[n - 1][2] / 2.0;
        if (n >= 4)
            dy_dx += (3 * u * u + 6 * u + 2) * diff[n - 1][3] / 6.0;

        dy_dx /= h;

        double d2y_dx2 = 0.0;
        if (n >= 3)
            d2y_dx2 += diff[n - 1][2];
        if (n >= 4)
            d2y_dx2 += (u + 1) * diff[n - 1][3];

        d2y_dx2 /= (h * h);

        double exact1 = f_prime(X);
        double exact2 = f_double_prime(X);

        cout << "\nNumerical f'(x)  = " << dy_dx << "\n";
        cout << "Exact f'(x)      = " << exact1 << "\n";
        cout << "Numerical f''(x) = " << d2y_dx2 << "\n";
        cout << "Exact f''(x)     = " << exact2 << "\n";

        cout << "Error f'(x)  (%) = " << fabs((exact1 - dy_dx) / exact1) * 100 << "\n";
        cout << "Error f''(x) (%) = " << fabs((exact2 - d2y_dx2) / exact2) * 100 << "\n";
    }

    return 0;
}
```

Numerical Differentiation by Backward Interpolation Method Input

```
3
5
1.0 1.6
1.5
6
1.0 1.5
1.4
7
0.5 1.7
1.6
```

Numerical Differentiation by Backward Interpolation Method Output

```
Testcase 1 :

Difference Table:

      6.000000      0.000000      0.000000      0.000000      0.000000
     10.007382      4.007382      0.000000      0.000000
     16.137330      6.129948      2.122566
     25.091759      8.954429
     37.700160

Numerical f'(x)  = 79.381600
Exact f'(x)      = 79.312500
Numerical f''(x) = 174.687500
Exact f''(x)     = 175.500000
Error f'(x)  (%) = 0.087124
Error f''(x) (%) = 0.462963
Testcase 2 :

Difference Table:

      6.000000      0.000000      0.000000      0.000000      0.000000      0.000000
      8.466910      2.466910      0.000000      0.000000      0.000000
     11.782720      3.315810      0.848900      0.000000
     16.137330      4.354610      1.038800
     21.744640      5.607310
     28.843750

Numerical f'(x)  = 63.133600
Exact f'(x)      = 63.112000
Numerical f''(x) = 149.180000
Exact f''(x)     = 148.960000
Error f'(x)  (%) = 0.034225
Error f''(x) (%) = 0.147691
Testcase 3 :

Difference Table:

      1.281250      0.000000      0.000000      0.000000      0.000000      0.000000      0.000000
      2.128470      0.847220      0.000000      0.000000      0.000000      0.000000
      4.214890      2.086420      1.239200      0.000000      0.000000
      8.466910      4.252020      2.165600      0.926400
     16.137330      7.670420      3.418400
     28.843750     12.706420
     48.606970

Numerical f'(x)  = 98.395100
Exact f'(x)      = 98.304000
Numerical f''(x) = 201.680000
Exact f''(x)     = 204.800000
Error f'(x)  (%) = 0.092672
Error f''(x) (%) = 1.523438
```

Solution of Ordinary Differential Equations (ODE)

Runge Kutta Method

Runge Kutta Method Theory

The Runge–Kutta fourth order method is a numerical technique used to approximate the solution of first-order ordinary differential equations of the form dy/dx = f(x, y). It improves accuracy by combining weighted slopes evaluated at different points within each step.

The method provides a good balance between accuracy and computational effort and is widely used for solving initial value problems.

Formula

For step size h:

$$k_1 = h*f(x_n, y_n)$$
$$k_2 = h*f(x_n + h/2, y_n + k_1/2)$$
$$k_3 = h*f(x_n + h/2, y_n + k_2/2)$$
$$k_4 = h*f(x_n + h, y_n + k_3)$$

$$y_{n+1} = y_n + (1/6)*(k_1 + 2k_2 + 2k_3 + k_4)$$

Notation

- x_n : current values independent variable
- y_n : current approximate value of the solution y at x_n
- y_{n+1} : next approximate value of the solution
- `h` : step size
- `f(x, y)` : given differential equation

Steps to Apply

1. Choose the initial values x_0 and y_0.
2. Select a suitable step size h.
3. Compute k_1, k_2, k_3, and k_4 using the given formulas.
4. Calculate y_{n+1}.
5. Repeat for the required interval.

Conditions of Applicability

- The differential equation must be of first order.
- The function f(x, y) should be continuous.
- Initial conditions must be given.

Advantages

- High accuracy compared to lower-order methods.
- Does not require higher derivatives.
- Widely used in practical applications.

Limitations

- Requires more computations per step.
- Fixed step size may reduce efficiency.
- Not suitable for stiff differential equations.

Runge Kutta Method Code

```
#include <bits/stdc++.h>
using namespace std;

double f(double x, double y)
{
    return (x - y) / 2.0;
}

int main()
{
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);

    double x0, y0, xn, h;
    cin >> x0 >> y0;
    cin >> xn;
    cin >> h;

    double x = x0;
    double y = y0;

    int n = (xn - x0) / h;

    for (int i = 0; i < n; i++)
    {
        double k1 = h * f(x, y);
        double k2 = h * f(x + h / 2.0, y + k1 / 2.0);
        double k3 = h * f(x + h / 2.0, y + k2 / 2.0);
        double k4 = h * f(x + h, y + k3);

        y = y + (k1 + 2*k2 + 2*k3 + k4) / 6.0;
        x = x + h;
    }

    cout << fixed << setprecision(5);
    cout << "Initial x0: " << x0 << endl;
    cout << "Initial y0: " << y0 << endl;
    cout << "Final x: " << xn << endl;
    cout << "Step h: " << h << endl;
    cout << "The value of y at x is: " << y << endl;

    return 0;
}
```

Runge Kutta Method Input

```
0
1
2
0.1
```

Runge Kutta Method Output

```
Initial x0: 0.00000
Initial y0: 1.00000
Final x: 2.00000
Step h: 0.10000
The value of y at x is: 1.10364
```

Solution of Numerical Integrations

Simpson's One-Third Rule

Simpson's One-Third Rule Theory

Simpson’s 1/3 Rule is a numerical integration method used to approximate the definite integral of a function when an exact analytical solution is difficult or impossible to obtain. It provides higher accuracy than the Trapezoidal Rule by approximating the integrand using parabolic arcs instead of straight lines.

Basic Idea

In Simpson’s 1/3 Rule, the interval [a, b] is divided into an even number of equal sub-intervals of width:

$h = (b - a) / n$, where n is even a is the lower limit and b is the upper limit.

The function values at these equally spaced points are used to construct quadratic polynomials over pairs of intervals. The area under each parabola is then calculated to approximate the total area under the curve.

Mathematical Formula

Let:

$x_0 = a$
$x_1 = a + h$
$x_2 = a + 2h$
...
$x_n = b$

and

$y_i = f(x_i)$

Then Simpson’s 1/3 Rule is given by:

$\int_a^b f(x)\,dx \approx (h / 3)\,[\, y_0 + y_n + 4(y_1 + y_3 + \dots + y_{n-1}) + 2(y_2 + y_4 + \dots + y_{n-2})\,]$

Conditions of Applicability

- The number of sub-intervals must be even
- The data points must be equally spaced
- The function should be smooth and continuous over the interval

Advantages

- Higher accuracy than the Trapezoidal Rule
- Simple and easy to apply
- Requires fewer intervals for good accuracy

Limitations

- Cannot be applied when the number of intervals is odd
- Not suitable for unequally spaced data
- Accuracy decreases for highly oscillatory functions

Simpson's One-Third Rule Code

```
 #include <bits/stdc++.h>
using namespace std;

void print(vector<double> &coeff)
{
    int n = coeff.size();
    int pow = n - 1;
    bool m = true;
    for (int i = 0; i < n; i++)
    {
        if (coeff[i] == 0)
        {
            pow--;
            continue;
        }
        if (i == n - 1)
        {
            if (coeff[i] < 0)
```

```
                cout << coeff[i] << "=0";
            else
                cout << "+" << coeff[i] << "=0";
        }
        else
        {
            if (m)
            {
                cout << coeff[i] << "X^" << pow;
                m = false;
            }
            else
            {
                if (coeff[i] > 0)
                    cout << "+" << coeff[i] << "X^" << pow;
                else
                    cout << coeff[i] << "X^" << pow;
            }
            pow--;
        }
    }
    cout << endl;
}

double f(double x, vector<double> &coeff)
{
    double val = 0;
    int n = coeff.size();
    int p = n - 1;
    for (int i = 0; i < n; i++)
    {
        val += coeff[i] * pow(x, p);
        p--;
    }
    return val;
}

double simp1_3rd(double u, double l, int interval, vector<double> &coeff)
{
    if (interval % 2 != 0)
        interval++; // ensure even
    double h = (u - l) / interval;
    double ans = f(u, coeff) + f(l, coeff);
    for (int i = 1; i < interval; i++)
    {
        double x = l + i * h;
        double y = f(x, coeff);
        if (i % 2 == 0)
            ans += 2 * y;
        else
            ans += 4 * y;
    }
    ans = ans * (h / 3.0);
    return ans;
}

int main()
{
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);

    int test;
    cin >> test;

    for (int t = 1; t <= test; t++)
    {
        cout << "Testcase: " << t << endl;

        int n;
        cout << "Enter the degree: ";
        cin >> n;
        cout << "Enter equation coefficients:" << endl;
        vector<double> coeff(n + 1);
        for (int i = 0; i <= n; i++)
            cin >> coeff[i];

        double u, l;
        cout << "Enter upper limit: ";
        cin >> u;
        cout << "Enter lower limit: ";
        cin >> l;

        int interval;
        cout << "Enter the interval: ";
        cin >> interval;

        double p;
        cout << "Enter the value of p: ";
        cin >> p;

        cout << "Polynomial: ";
        print(coeff);

        double result = simp1_3rd(u, l, interval, coeff);
        cout << "Integral of f(x) from " << l << " to " << u << " is: " << result << endl
             << endl;
    }
}
```

Simpson's One-Third Rule Input

```
5

2
1 -3 2
2
0
4
1

3
2 0 -1 1
5
1
10
2

1
4 -2
6
0
3
1

0
5
0
1
2
1

2
1 0 -1
3
-1
2
2
```

Simpson's One-Third Rule Output

```
 Testcase: 1
Enter the degree: Enter equation coefficients:
Enter upper limit: Enter lower limit: Enter the interval: Enter the value of p: Polynomial: 1X^2-3X^1+2=0
Integral of f(x) from 0 to 2 is: 0.666667

Testcase: 2
Enter the degree: Enter equation coefficients:
Enter upper limit: Enter lower limit: Enter the interval: Enter the value of p: Polynomial: 2X^3-1X^1+1=0
Integral of f(x) from 1 to 5 is: 304

Testcase: 3
Enter the degree: Enter equation coefficients:
Enter upper limit: Enter lower limit: Enter the interval: Enter the value of p: Polynomial: 4X^1-2=0
Integral of f(x) from 0 to 6 is: 60

Testcase: 4
Enter the degree: Enter equation coefficients:
Enter upper limit: Enter lower limit: Enter the interval: Enter the value of p: Polynomial: +5=0
Integral of f(x) from 1 to 0 is: -5

Testcase: 5
Enter the degree: Enter equation coefficients:
Enter upper limit: Enter lower limit: Enter the interval: Enter the value of p: Polynomial: 1X^2-1=0
Integral of f(x) from -1 to 3 is: 5.33333
```

Simpson's Three-Eighths Rule

Simpson's Three-Eighths Rule Theory

Simpson’s 3/8 Rule is a numerical integration method used to approximate the definite integral of a function when an exact analytical solution is difficult or impossible to obtain. It is an extension of Simpson’s 1/3 Rule and uses cubic polynomials (third-degree) to approximate the integrand, providing higher accuracy for certain functions.

Basic Idea

In Simpson’s 3/8 Rule, the interval [a, b] is divided into a multiple of 3 equal sub-intervals of width:

$h = (b - a) / n$, where n is a multiple of 3

The function values at these equally spaced points are used to construct cubic polynomials over sets of three intervals. The area under each cubic curve is calculated to approximate the total integral.

Mathematical Formula

Let:

$x_0 = a$ $x_1 = a + h$ $x_2 = a + 2h$ $x_3 = a + 3h$ ... $x_n = b$

and

$y_i = f(x_i)$

Then Simpson’s 3/8 Rule is given by:

$\int_a^b f(x)\,dx \approx (3h / 8) [\ y_0 + y_n + 3(y_1 + y_2 + y_4 + y_5 + \dots + y_{n-1} + y_{n-2}) + 2(y_3 + y_6 + \dots + y_{n-3})\ ]$

Conditions of Applicability

- The number of sub-intervals must be a multiple of 3
- The data points must be equally spaced
- The function should be smooth and continuous over the interval

Advantages

- More accurate than the Trapezoidal Rule and 1/3 Rule for functions requiring cubic approximation
- Can handle curves with higher-order behavior
- Simple to apply for equally spaced data

Limitations

- Cannot be applied if the number of intervals is not a multiple of 3
- Not suitable for unequally spaced data
- Accuracy decreases for highly oscillatory functions

Simpson's Three-Eighths Rule Code

```
#include <bits/stdc++.h>
using namespace std;

double f(double x)
{
  return 1.0 / (1 + x * x);
}

void printFunction()
{
  cout << "f(x) = 1 / (1 + x^2)" << endl;
}

double simp_3_8th(double u, double l, int interval)
{

  if (interval % 3 != 0)
  {
    interval += (3 - interval % 3);
  }

  double h = (u - l) / interval;
  double ans = f(u) + f(l);

  for (int i = 1; i < interval; i++)
  {
    double x = l + i * h;
    double y = f(x);

    if (i % 3 == 0)
      ans += 2 * y;
    else
      ans += 3 * y;
  }

  ans = ans * (3 * h / 8.0);
  return ans;
}

int main()
{
  freopen("input.txt", "r", stdin);
  freopen("output.txt", "w", stdout);

  int test;
  cin >> test;

  for (int t = 1; t <= test; t++)
  {
    cout << "Testcase: " << t << endl;

    double u, l;
    cout << "Enter upper limit: ";
    cin >> u;

    cout << "Enter lower limit: ";
    cin >> l;

    int interval;
    cout << "Enter the interval: ";
    cin >> interval;

    printFunction();

    double result = simp_3_8th(u, l, interval);
    cout << "Integral of f(x) from " << l << " to " << u
         << " is: " << result << endl
         << endl;
  }

  return 0;
}
```

Simpson's Three-Eighths Rule Input

```

5

1
0
3

2
0
6

3
0
9

4
0
12

5
0
15
```

Simpson's Three-Eighths Rule Output

```
 Testcase: 1
Enter upper limit: Enter lower limit: Enter the interval: f(x) = 1 / (1 + x^2)
Integral of f(x) from 0 to 1 is: 0.784615

Testcase: 2
Enter upper limit: Enter lower limit: Enter the interval: f(x) = 1 / (1 + x^2)
Integral of f(x) from 0 to 2 is: 1.10638

Testcase: 3
Enter upper limit: Enter lower limit: Enter the interval: f(x) = 1 / (1 + x^2)
Integral of f(x) from 0 to 3 is: 1.2483

Testcase: 4
Enter upper limit: Enter lower limit: Enter the interval: f(x) = 1 / (1 + x^2)
Integral of f(x) from 0 to 4 is: 1.32508

Testcase: 5
Enter upper limit: Enter lower limit: Enter the interval: f(x) = 1 / (1 + x^2)
Integral of f(x) from 0 to 5 is: 1.37267
```

Solution of Curve Fitting Model

Least Square Regression Method for Linear Equations

Least Square Regression Method for Linear Equations Theory

Add theory here

Least Square Regression Method for Linear Equations Code

```
// add your code here
```

Least Square Regression Method for Linear Equations Input

```
add input here
```

Least Square Regression Method for Linear Equations Output

```
add output here
```

Least Square Regression Method for Transcendental Equations

Least Square Regression Method for Transcendental Equations Theory

Add theory here

Least Square Regression Method for Transcendental Equations Code

```
// add your code here
```

Least Square Regression Method for Transcendental Equations Input

```
add input here
```

Least Square Regression Method for Transcendental Equations Output

```
add output here
```

Least Square Regression Method for Polynomial Equations

Least Square Regression Method for Polynomial Equations Theory

Add theory here

Least Square Regression Method for Polynomial Equations Code

```
// add your code here
```

Least Square Regression Method for Polynomial Equations Input

```
add input here
```

Least Square Regression Method for Polynomial Equations Output

```
add output here
```