Pt. Ravishankar Shukla University
Center for Basic Sciences

PML-501

Numerical Methods Laboratory

Submitted To:
Dr. Varsha Thakur
Asst. Professor
Computer Science

Submitted By :
.....
.....
V Semester

ko

Contents

Part I
General Python Programs

1 Class Assignments

1.1 Program for Area of Rectangle

```
l=int(input("Enter the length of rectangle"))
b=int(input("Enter the breadth of rectangle"))
area=l*b
print("Area of Rectangle=",area)
```

1.1.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/1.py ==========
Enter the length of rectangle4
Enter the breadth of rectangle3
Area of Rectangle= 12
>>>
```

1.2 Program for Pythagoras Theorem

```
import math
a=int(input("enter the length of first side"))
b=int(input("enter the length of second side"))
h=int(math.sqrt(a**2+b**2))
print("value of hypotenuse",h)
```

1.2.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/2.py ==========
enter the length of first side12
enter the length of second side5
value of hypotenuse 13
>>>
```

1.3 Program for Greatest among 3 Numbers

```
a=input("Enter a")
b=input("Enter b")
c=input("Enter c")
if((a>b) and (a>c)):
    print("Greatest is =",a)
elif(b>c):
    print("Greatest is =",b)
else:
    print("Greatest is =",c)
```

1.3.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/3.py ==========
Enter a23
Enter b45
Enter c21
Greatest is = 45
>>>
```

1.4 Program for arithmatic calculation

```
x=int(input("Enter the value of x"))
y=int(input("Enter the value of y"))
print("1. Addition")
print("2. Multiplication")
print("3. Subtraction")
print("4. Division")
i=int(input("Enter 1, 2, 3,or 4"))
if(i==1):
    print("Addition =",x+y)
elif(i==2):
    print("Multiplication =",x*y)
elif(i==3):
    print("Subtraction =",x-y)
elif(i==4):
    print("Division =",x/y)
else:
    print("Invalid Case")
```

1.4.1 Output

```
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART:/home/jhii/Documents/Python/4.py ==========
Enter the value of x23
Enter the value of y67
1. Addition
2. Multiplication
3. Subtraction
4. Division
Enter 1, 2, 3,or 42
Multiplication = 1541
>>>
```

1.5 Program for day of week

```
a=int(input("Enter a number from 1 to 7"))
if(a==1):
    print("Monday")
elif(a==2):
    print("Tuesday")
elif(a==3):
    print("Wednesday")
```

```
elif(a==4):
    print("Thursday")
elif(a==5):
    print("Friday")
elif(a==6):
    print("Saturday")
elif(a==7):
    print("Sunday")
```

1.5.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/5.py ==========
Enter a number from 1 to 74
Thursday
>>>
```

1.6 Program for left and right shift

```
a=int(input("Enter a"))
b=int(input("Enter b"))
a=a<<2
print(a)
b=b>>2
print(b)
```

1.6.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/6.py ==========
Enter a45
Enter b32
180
8
>>> Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/6.py ==========
Enter a45
Enter b32
180
8
>>>
```

1.7 Program for sum of digits

```
num=1821
sum=0
while num>0:
    n=num%10
    sum+=n
```

```
    num=num//10
print("Sum of digits = ",sum)
```

1.7.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/7.py ==========
Sum of digits =  12
>>>
```

1.8 Program for reverse of a number

```
num=1821
i=0
rev=0
while num>0:
    n=num%10
    rev=rev*10+n
    num=num//10
print("Reverse of the number=",rev)
```

1.8.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/8.py ==========
Reverse of the number= 1281
>>>
```

1.9 Program for sum of numbers upto 20 not divisible by 5

```
i=1
sum=0
while i<=20:
    if(i%5!=0):
        sum+=i
    i+=1
print("Sum=",sum)
```

1.9.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/9.py ==========
Sum= 160
>>>
```

1.10 Program for factorial of a number

```
num=5
i=1
f=1
while i<=num:
    f=f*i
    i+=1
print("Factorial =",f)
```

1.10.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/10.py ==========
Factorial = 5040
>>>
```

1.11 Program for square of first 5 natural number

```
for i in range(1,6):
    print("Square=",i*i)
```

1.11.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/11.py ==========
Square= 1
Square= 4
Square= 9
Square= 16
Square= 25
>>>
```

1.12 Program for checking armstrong number

```
num=153
x=num
sum=0
while x>0:
    r=x%10
    sum+=r**3
    x//=10
if(sum==num):
    print("This is amstrong number")
else:
        print("This is not an amstrong number")
```

1.12.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/12.py ==========
This is amstrong number
>>>
```

1.13 Program for sum of first five natural numbers

```
num=5
sum=0
for i in range(1,num+1):
    sum=sum+i
print("Sum of number =",sum)
```

1.13.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/13.py ==========
Sum of number = 15
>>>
```

1.14 Program for fibonacci series

```
a=0
b=1
print(a)
print(b)
for i in range(1,11):
    c=a+b
    print(c)
    a=b
    b=c
```

1.14.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/14.py ==========
0
1
1
2
3
5
8
13
21
34
55
```

```
89
>>>
```

1.15 Program for sum of numbers using user-defined function without argument without return type

```
def Sum():
    x=int(input("Enter any number x"))
    y=int(input("Enter any number y"))
    z=x+y
    print("Sum of two numbers =",z)

Sum()
```

1.15.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/15.py ==========
Enter any number x18
Enter any number y21
Sum of two numbers = 39
>>>
```

1.16 Program for sum of numbers using user-defined function without argument with return type

```
def Sum():
    x=int(input("Enter any number x"))
    y=int(input("Enter any number y"))
    z=x+y
    return z
d=Sum()
print("Sum of two numbers= ",d)
```

1.16.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/16.py ==========
Enter any number x18
Enter any number y21
Sum of two numbers=  39
>>>
```

1.17 Program for sum of numbers using user-defined function with argument with return type

```
def Sum(x,y):
    z=x+y
    return z
a=int(input("Enter any number x"))
```

```
b=int(input("Enter any number y"))
d=Sum(a,b)
print("Sum of two numbers= ",d)
```

1.17.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/17.py ==========
Enter any number x18
Enter any number y21
Sum of two numbers=  39
>>>
```

1.18 Program for sum of numbers using user-defined function with argument without return type

```
def Sum(x,y):
    z=x+y
    print("Sum of two numbers= ",z)
a=int(input("Enter any number x"))
b=int(input("Enter any number y"))
Sum(a,b)
```

1.18.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/18.py ==========
Enter any number x18
Enter any number y21
Sum of two numbers=  39
>>>
```

1.19 Program for factorial of number using user-defined function

```
def factorial(n):
    f=1
    for i in range(1,n+1):
      f*=i
      i+=1
    print("Factorial of the given number",f)
m=int(input("Enter the number"))
factorial(m)
```

1.19.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/19.py ==========
Enter the number6
```

```
Factorial of the given number 720
>>>
```

1.20 Program for fibonacci series using user-defined function

```
def fibonacci(a,b,n):
    for i in range(1,n-1):
        c=a+b
        print(c)
        a=b
        b=c
a=0
b=1
m=int(input("Enter the number of terms"))
print(a)
print(b)
fibonacci(a,b,m)
```

1.20.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/20.py ==========
Enter the number of terms7
0
1
1
2
3
5
8
>>>
```

1.21 Program for greatest among 3 numbers using user-defined function

```
def greatest(a,b,c):
    if(a>b&a>c):
        print("The greatest number is",a)
    elif(b>c):
        print("The greatest number is",b)
    else:
        print("The greatest number is",c)
x=int(input("Enter x"))
y=int(input("Enter y"))
z=int(input("Enter z"))
greatest(x,y,z)
```

1.21.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/21.py ==========
Enter x32
Enter y12
```

```
Enter z78
The greatest number is 78
>>>
```

1.22 Program for checking prime number using user-defined function

```
def prime(num):
    cnt=0
    for i in range(1,num+1):
        if(num%i==0):
            cnt+=1
    if(cnt==2):
        print("This is prime number")
    else:
        print("This is not a prime number")
n=int(input("Enter a number"))
prime(n)
```

1.22.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/22.py ==========
Enter a number34
This is not a prime number
>>>
```

1.23 Program for sum of digits using user-defined function

```
def sumdigits(num):
    sum=0
    while num>0:
        r=num%10
        sum+=r
        num=num//10
    print("Sum of digits=",sum)
n=int(input("Enter a number"))
sumdigits(n)
```

1.23.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/23.py ==========
Enter a number12
Sum of digits= 3
>>>
```

1.24 Program for factorial of a number using user-defined recursive function

```
def fact(n):
    if(n!=1):
      f=n*fact(n-1)
      return f
    else:
        return 1

n=int(input("Enter any number "))
fa=fact(n)
print("Factorial of %3d is %ld" %(n,fa))
```

1.24.1 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /home/jhii/Documents/Python/24.py ==========
Enter any number 7
Factorial of   7 is 5040
>>>
```

Part II
Numerical Methods Programs

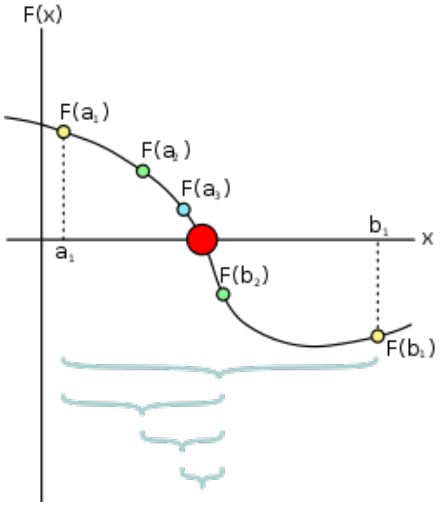
2 Roots of Nonlinear Equations

2.1 Bisection Method

The method is applicable for numerically solving the equation f(x) = 0 for the real variable x, where f is a continuous function defined on an interval [a, b] and where f(a) and f(b) have opposite signs. In this case a and b are said to bracket a root since, by the intermediate value theorem, the continuous function f must have at least one root in the interval (a, b).

At each step the method divides the interval in two by computing the midpoint c = (a+b) / 2 of the interval and the value of the function f(c) at that point. Unless c is itself a root (which is very unlikely, but possible) there are now only two possibilities: either f(a) and f(c) have opposite signs and bracket a root, or f(c) and f(b) have opposite signs and bracket a root. The method selects the subinterval that is guaranteed to be a bracket as the new interval to be used in the next step. In this way an interval that contains a zero of f is reduced in width by 50% at each step. The process is continued until the interval is sufficiently small.

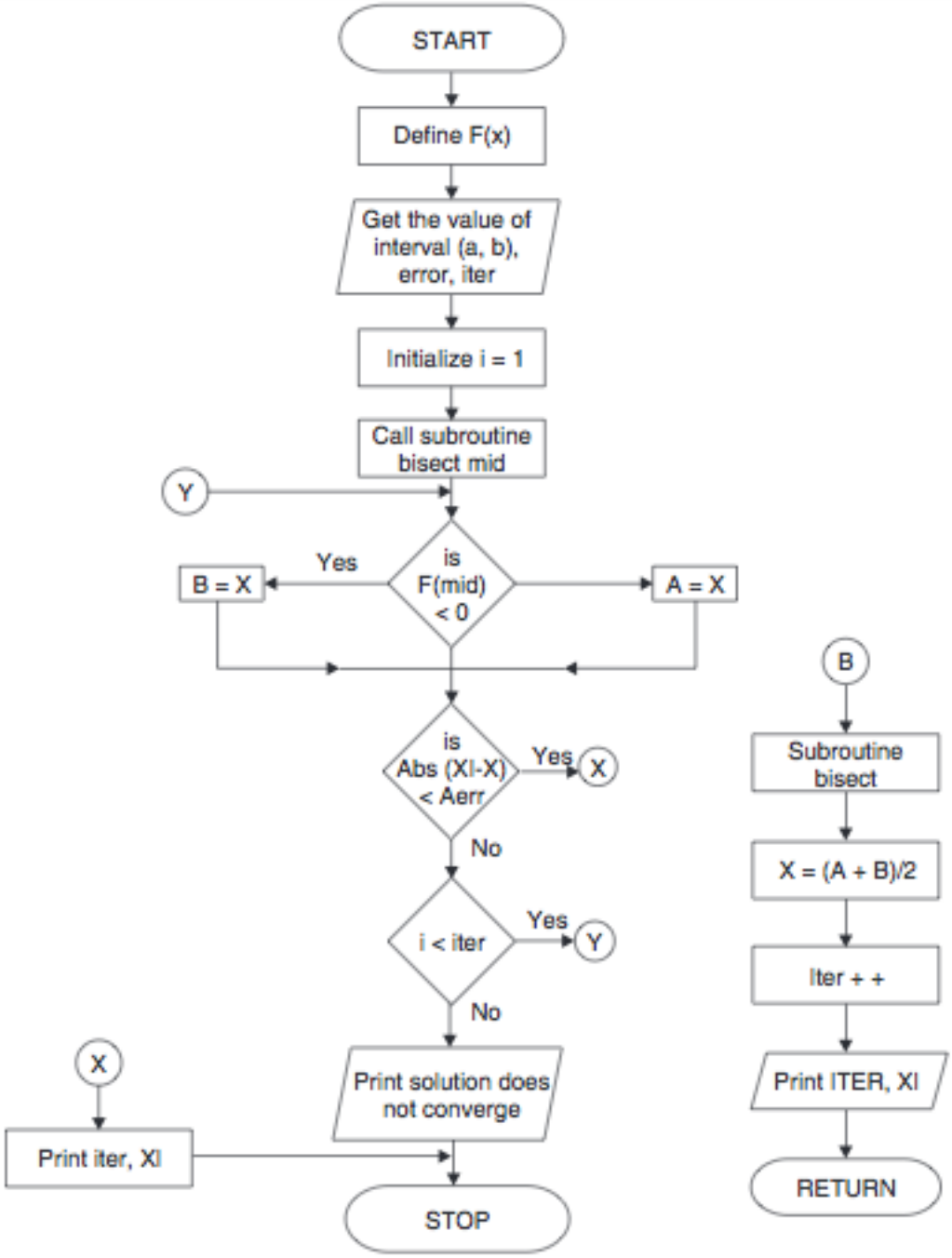
Explicitly, if f(a) and f(c) have opposite signs, then the method sets c as the new value for b, and if f(b) and f(c) have opposite signs then the method sets c as the new a. (If f(c)=0 then c may be taken as the solution and the process stops.) In both cases, the new f(a) and f(b) have opposite signs, so the method is applicable to this smaller interval.



2.1.1 Algorithm

1. Start
2. Read x1, x2, e
   Here x1 and x2 are initial guesses e is the absolute error i.e. the desired degree of accuracy
3. Compute: $f1 = f(x1)$ and $f2 = f(x2)$
4. If $(f1 * f2) > 0$, then display initial guesses are wrong and goto (11).
   Otherwise continue.
5. $x = (x1 + x2)/2$
6. If $(|(x1 - x2)/x| < e)$, then display x and goto (11).
7. Else, $f = f(x)$
8. If $((f * f1) > 0)$, then $x1 = x$ and $f1 = f$.
9. Else, $x2 = x$ and $f2 = f$.
10. Goto (5).
    Now the loop continues with new values.
11. Stop

2.1.2 Flow Chart-



2.1.3 Python Program-

```
import math
def f(x):
    return(x*x*x-4*x-9)
def bisect(a,b,p):
    x=(a+b)/2
    x=round(x,p+1)
    return x

a=int(input("Enter value of a "))
b=int(input("Enter value of b "))
maxitr=int(input("Enter max no iterations "))
p=int(input("Enter precision "))
err=pow(10,-p)
```

```
itr=1
xp=b
while(itr<=maxitr):
    x=bisect(a,b,p)
    if(f(a)*f(x) < 0):
        b=x
    else:
        a=x
    print(" After",itr,"iterations, root=",x)
    if(abs(x-xp)< err):
        print("Value of root is ",x)
        exit()
    xp=x
    itr=itr+1
print("Increase maximum iterations")
```

2.1.4 Bisection Method Output

```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Enter value of a 2
Enter value of b 3
Enter max no iterations 20
Enter precision 4
 After 1 iterations, root= 2.5
 After 2 iterations, root= 2.75
 After 3 iterations, root= 2.625
 After 4 iterations, root= 2.6875
 After 5 iterations, root= 2.71875
 After 6 iterations, root= 2.70312
 After 7 iterations, root= 2.71094
 After 8 iterations, root= 2.70703
 After 9 iterations, root= 2.70507
 After 10 iterations, root= 2.70605
 After 11 iterations, root= 2.70654
 After 12 iterations, root= 2.70629
 After 13 iterations, root= 2.70641
 After 14 iterations, root= 2.70648
Value of root is  2.70648
>>>
```

2.2 Regula-Falsi Method

The convergence rate of the bisection method could possibly be improved by using a different solution estimate.

The regula falsi method calculates the new solution estimate as the x-intercept of the line segment joining the endpoints of the function on the current bracketing interval. Essentially, the root is being approximated by replacing the actual function by a line segment on the bracketing interval and then using the classical double false position formula on that line segment.

More precisely, suppose that in the k-th iteration the bracketing interval is (a_k, b_k). Construct the line through the points $(a_k, f(a_k))$ and $(b_k, f(b_k))$, as illustrated. This line is a secant or chord of the graph of the function f. In point-slope form, its equation is given by

$$yf(b_k) = \frac{f(b_k) - f(a_k)}{b_k - a_k}(xb_k). \tag{1}$$

Now choose ck to be the x-intercept of this line, that is, the value of x for which y = 0, and substitute these values to obtain

$$f(b_k) + \frac{f(b_k) - f(a_k)}{b_k - a_k}(c_k - b_k) = 0. \tag{2}$$

Solving this equation for c_k gives:

$$c_k = b_k - f(b_k)\frac{b_k - a_k}{f(b_k) - f(a_k)} = \frac{a_k f(b_k) - b_k f(a_k)}{f(b_k) - f(a_k)}. \tag{3}$$

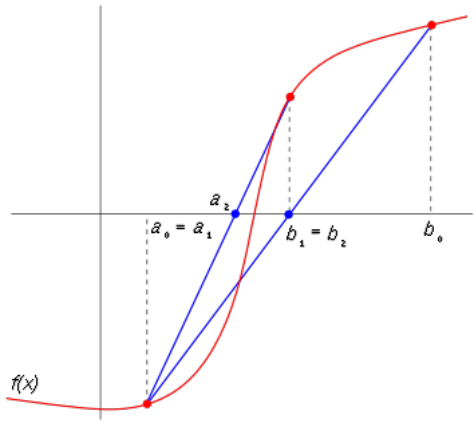
This last symmetrical form has a computational advantage:

As a solution is approached, a_k and bk will be very close together, and nearly always of the same sign. Such a subtraction can lose significant digits. Because $f(b_k)$ and $f(a_k)$ are always of opposite sign the "subtraction" in the numerator of the improved formula is effectively an addition (as is the subtraction in the denominator too).

At iteration number k, the number c_k is calculated as above and then, if $f(a_k)$ and $f(c_k)$ have the same sign,set $a_k + 1 = c_k$ and $b_k + 1 = b_k$, otherwise set $a_k + 1 = a_k$ and $b_k + 1 = c_k$. This process is repeated until the root is approximated sufficiently well.

The above formula is also used in the secant method, but the secant method always retains the last two computed points, and so, while it is slightly faster, it does not preserve bracketing and may not converge.

The fact that regula falsi always converges, and has versions that do well at avoiding slowdowns, makes it a good choice when speed is needed. However, its rate of convergence can drop below that of the bisection method.



2.2.1 Algorithm-

1. Start
2. Read values of x0, x1 and e
   Here x0 and x1 are the two initial guesses e is the degree of accuracy or the absolute error i.e. the stopping criteria
3. Computer function values f(x0) and f(x1)
4. Check whether the product of f(x0) and f(x1) is negative or not.
   If it is positive take another initial guesses.
   If it is negative then goto step 5.

5. Determine: $x = [x0 * f(x1) - x1 * f(x0)]/(f(x1) - f(x0))$

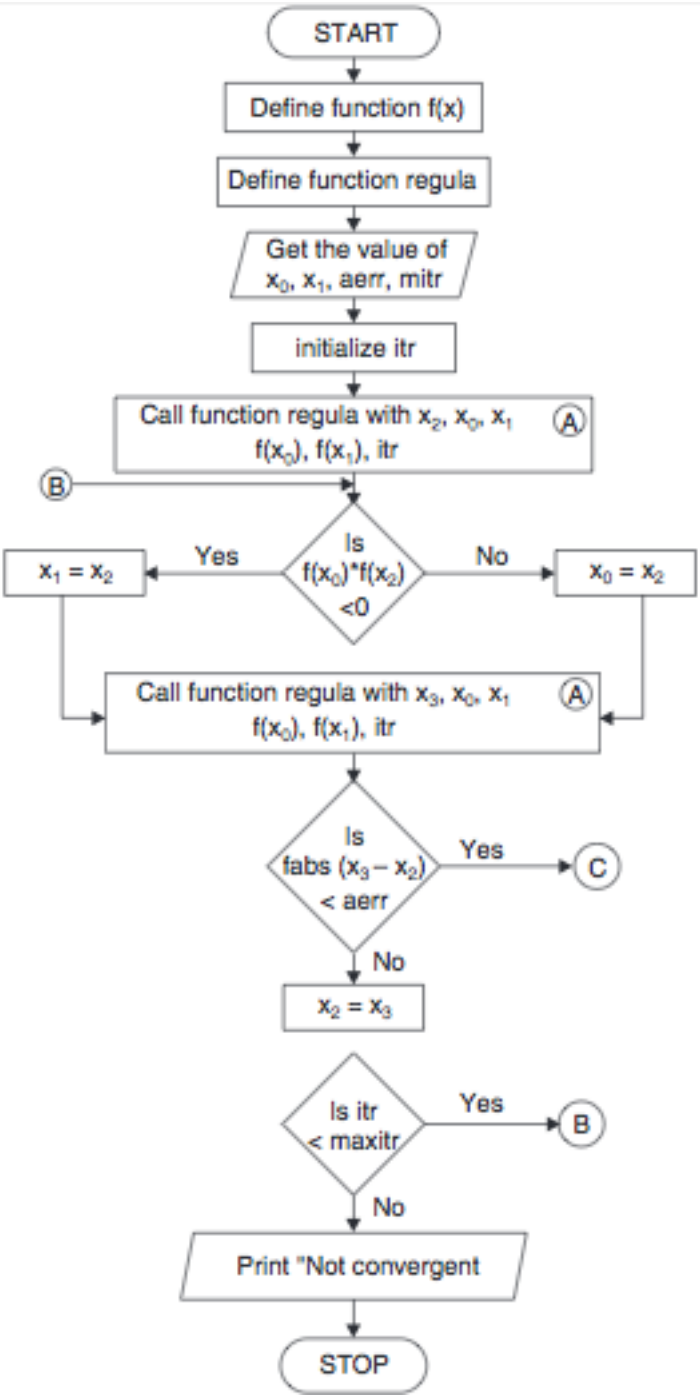
6. Check whether the product of f(x1) and f(x) is negative or not.
   If it is negative, then assign x0 = x;
   If it is positive, assign x1 = x;

7. Check whether the value of f(x) is greater than 0.00001 or not.
   If yes, goto step 5.
   If no, goto step 8.
   Here the value 0.00001 is the desired degree of accuracy, and hence the stopping criteria.

8. Display the root as x.

9. Stop

2.2.2 Flow Chart-



2.2.3 Python Program-

```
import math
def f(x):
    return(x*x*x-4*x-9)
def regula(x1,x2,p):
    x=x1-(f(x1)*(x2-x1)/(f(x2)-f(x1)))
    x=round(x,p+1)
    return x

x1=int(input("Enter value of x1 "))
x2=int(input("Enter value of x2 "))
maxitr=int(input("Enter max no iterations "))
p=int(input("Enter precision "))
err=pow(10,-p)
itr=1
xp=x1
while(itr<=maxitr):
    x=regula(x1,x2,p)
    if(f(x1)*f(x) < 0):
        x2=x
    else:
        x1=x
    print(" After",itr,"iterations, root=",x)
    if(abs(x-xp)< err):
        print("Value of root is ",x)
        exit()
    xp=x
    itr=itr+1
print("Increase maximum iterations")
```

2.2.4 Regula-Falsi Method Output

```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Enter value of x1 2
Enter value of x2 3
Enter max no iterations 10
Enter precision 4
 After 1 iterations, root= 2.6
 After 2 iterations, root= 2.69325
 After 3 iterations, root= 2.70492
 After 4 iterations, root= 2.70633
 After 5 iterations, root= 2.7065
 After 6 iterations, root= 2.70652
Value of root is  2.70652
>>>
```

2.3 Newton-Raphson Method

The idea is to start with an initial guess which is reasonably close to the true root, then to approximate the function by its tangent line using calculus, and finally to compute the x-intercept of this tangent line by elementary algebra. This x-intercept will typically be a better approximation to the original function's root than the first guess, and the method can be iterated.

More formally, suppose $f : (a, b) \longrightarrow R$ is a differentiable function defined on the interval (a, b) with values in the real numbers R, and we have some current approximation x_n. Then we can derive the formula for a better approximation, $x_n + 1$ by referring to the diagram on the right. The equation of the tangent line to the curve $y = f(x)$ at $x = x_n$ is

$$y = f'(x_n)\,(x - x_n) + f(x_n), \tag{4}$$

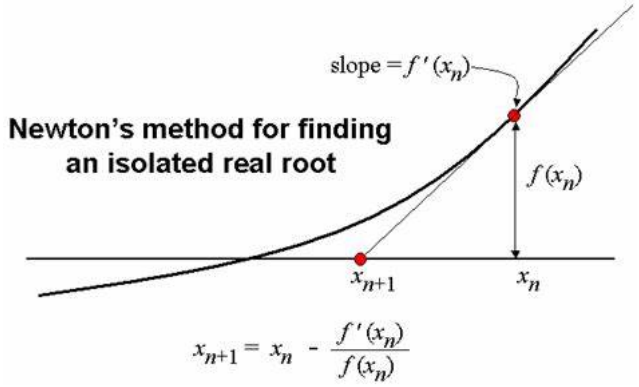
where f denotes the derivative. The x-intercept of this line (the value of x which makes y = 0) is taken as the next approximation, $x_n + 1$, to the root, so that the equation of the tangent line is satisfied when $(x, y) = (x_n + 1, 0)$

$$0 = f'(x_n)\,(x_{n+1} - x_n) + f(x_n).0 = f'(x_n)\,(x_{n+1} - x_n) + f(x_n). \tag{5}$$

Solving for $x_n + 1$ gives

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{6}$$

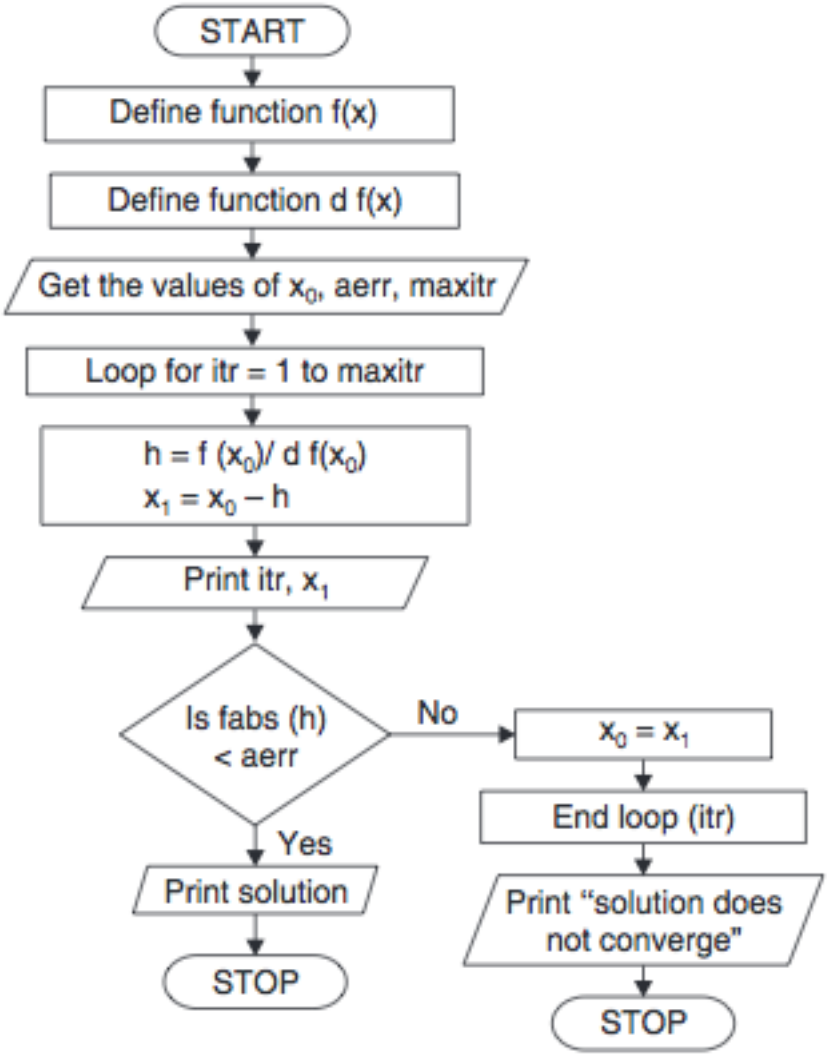
We start the process with some arbitrary initial value x_0. (The closer to the zero, the better. But, in the absence of any intuition about where the zero might lie, a "guess and check" method might narrow the possibilities to a reasonably small interval by appealing to the intermediate value theorem.) The method will usually converge, provided this initial guess is close enough to the unknown zero, and that $f(x_0)0$. Furthermore, for a zero of multiplicity 1, the convergence is at least quadratic (see rate of convergence) in a neighbourhood of the zero, which intuitively means that the number of correct digits roughly doubles in every step.



2.3.1 Algorithm-

1. Start
2. Read x, e, n, d
   *x is the initial guess
   e is the absolute error i.e the desired degree of accuracy
   n is for operating loop
   d is for checking slope*
3. Do for i =1 to n in step of 2
4. f = f(x)
5. f1 = f'(x)
6. If ($|f1| < d$), then display too small slope and goto 11.
7. x1 = x – f/f1
8. If ($|(x1–x)/x1| < e$), the display the root as x1 and goto 11.
9. x = x1 and end loop
10. Display method does not converge due to oscillation.
11. Stop

2.3.2 Flow Chart-



2.3.3 Python Program-

```
import math
def f(x):
    return(x*x*x-4*x-9)
def Df(x):
    return(3*x*x-4)
def newt_raph(x1,p):
    x=x1-(f(x1)/Df(x1))
    x=round(x,p+1)
    return x

x1=int(input("Enter approx. value of root "))
maxitr=int(input("Enter max no iterations "))
p=int(input("Enter precision "))
err=pow(10,-p)
itr=1
xp=x1
while(itr<=maxitr):
    x=newt_raph(xp,p)
    print(" After",itr,"iterations, root=",x)
    if(abs(x-xp)< err):
        print("Value of root is ",x)
```

```
        exit()
    xp=x
    itr=itr+1
print("Increase maximum iterations")
```

2.3.4 Newton-Raphson Method Output

```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Enter approx. value of root 2
Enter max no iterations 20
Enter precision 4
 After 1 iterations, root= 3.125
 After 2 iterations, root= 2.76853
 After 3 iterations, root= 2.7082
 After 4 iterations, root= 2.70653
 After 5 iterations, root= 2.70653
Value of root is  2.70653
>>> ========== RESTART ==========
>>>
Enter approx. value of root 3
Enter max no iterations 20
Enter precision 4
 After 1 iterations, root= 2.73913
 After 2 iterations, root= 2.707
 After 3 iterations, root= 2.70653
 After 4 iterations, root= 2.70653
Value of root is  2.70653
>>> ========== RESTART ==========
>>>
Enter approx. value of root 0
Enter max no iterations 20
Enter precision 4
 After 1 iterations, root= -2.25
 After 2 iterations, root= -1.23184
 After 3 iterations, root= 9.52678
 After 4 iterations, root= 6.47943
 After 5 iterations, root= 4.53511
 After 6 iterations, root= 3.38897
 After 7 iterations, root= 2.85157
 After 8 iterations, root= 2.7152
 After 9 iterations, root= 2.70656
 After 10 iterations, root= 2.70653
Value of root is  2.70653
>>>
```

2.4 Secant Method

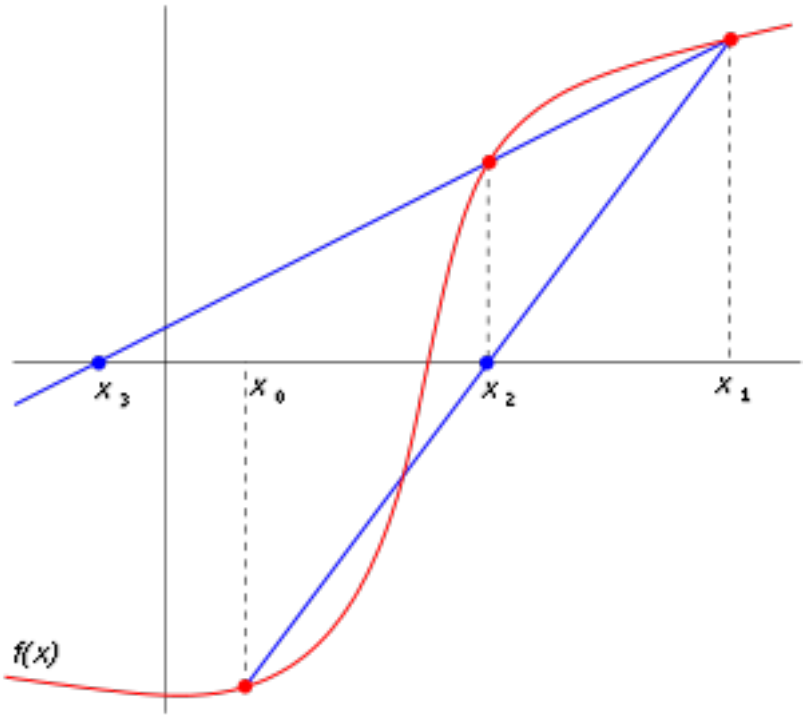
Secant method is considered to be the most effective approach to find the root of a non-linear function. It is a generalized from the Newton-Raphson method and does not require obtaining the derivatives of the function. So, this method is generally used as an alternative to Newton Raphson method.

Secant method falls under open bracket type. The programming effort may be a tedious to some extent, but the secant method algorithm and flowchart is easy to understand and use for coding in any high level programming language.

This method uses two initial guesses and finds the root of a function through interpolation approach. Here, at each successive iteration, two of the most recent guesses are used. That means, two most recent fresh values are used to find out the next approximation. The secant method is defined by the recurrence relation

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} = \frac{x_{n-2}f(x_{n-1}) - x_{n-1}f(x_{n-2})}{f(x_{n-1}) - f(x_{n-2})}. \tag{7}$$

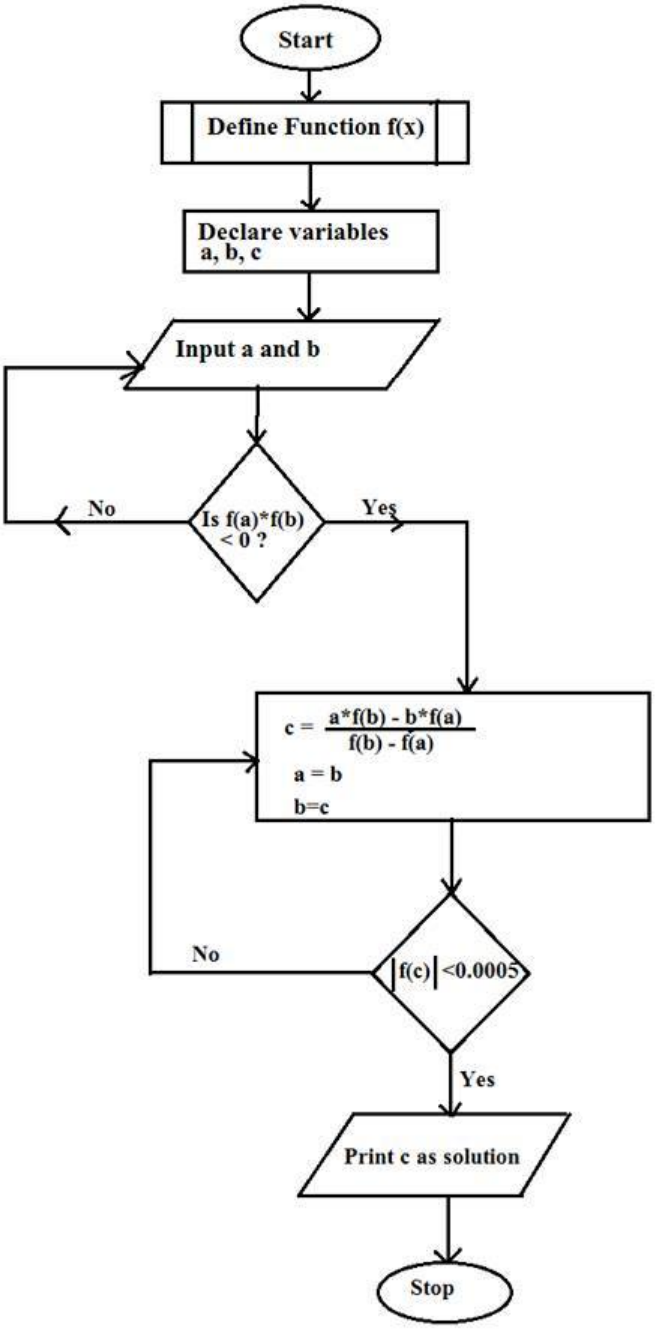
As can be seen from the recurrence relation, the secant method requires two initial values, x_0 and x_1, which should ideally be chosen to lie close to the root.



2.4.1 Algorithm-

1. Start
2. Get values of x0, x1 and e
   Here x0 and x1 are the two initial guesses e is the stopping criteria, absolute error or the desired degree of accuracy
3. Compute f(x0) and f(x1)
4. Compute x2 = [x0*f(x1) – x1*f(x0)] / [f(x1) – f(x0)]
5. Test for accuracy of x2
   If $|(x2–x1)/x2| > e$,
   then assign x0 = x1 and x1 = x2
   goto step 4
   Else,
   goto step 6
6. Display the required root as x2.
7. Stop

2.4.2 Flow Chart-



2.4.3 Python Program-

```
import math
def f(x):
    return(x*x-4*x-10)
def secant(x1,x2,p):
    x=x2-(f(x2)*(x2-x1)/(f(x2)-f(x1)))
    x=round(x,p+1)
    return x

x1=int(input("Enter value of x1 "))
x2=int(input("Enter value of x2 "))
maxitr=int(input("Enter max no iterations "))
p=int(input("Enter precision "))
err=pow(10,-p)
```

```
itr=1
while(itr<=maxitr):
    x=secant(x1,x2,p)
    print(" After",itr,"iterations, root=",x)
    if(abs(x-x2)< err):
        print("Value of root is ",x)
        exit()
    x1=x2
    x2=x
    itr=itr+1
print("Increase maximum iterations")
```

2.4.4 Secant Method Output

```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Enter value of x1 2
Enter value of x2 3
Enter max no iterations 20
Enter precision 4
 After 1 iterations, root= 2.6
 After 2 iterations, root= 2.69325
 After 3 iterations, root= 2.70719
 After 4 iterations, root= 2.70652
 After 5 iterations, root= 2.70653
Value of root is  2.70653
>>> ========== RESTART ==========
>>>
Enter value of x1 4
Enter value of x2 5
Enter max no iterations 20
Enter precision 4
 After 1 iterations, root= 3.31579
 After 2 iterations, root= 3.02361
 After 3 iterations, root= 2.77332
 After 4 iterations, root= 2.71502
 After 5 iterations, root= 2.70678
 After 6 iterations, root= 2.70653
 After 7 iterations, root= 2.70653
Value of root is  2.70653
>>> ========== RESTART ==========
>>>
Enter value of x1 1
Enter value of x2 2
Enter max no iterations 20
Enter precision 4
 After 1 iterations, root= 5.0
 After 2 iterations, root= 2.25714
 After 3 iterations, root= 2.43181
 After 4 iterations, root= 2.77958
 After 5 iterations, root= 2.69684
 After 6 iterations, root= 2.70621
 After 7 iterations, root= 2.70653
 After 8 iterations, root= 2.70653
Value of root is  2.70653
>>>
```

2.5 Fixed-Point Method

Iteration method, also known as the fixed point iteration method, is one of the most popular approaches to find the real roots of a nonlinear function. It requires just one initial guess and has a fast rate of convergence which is linear.

Iterative method is also referred to as an open bracket method or a simple enclosure method. It is based on modification approach to find the root of the function. Overall, it gives good accuracy just like the other methods.

The transcendental equation f(x) = 0 can be converted algebraically into the form x = g(x) and then using the iterative scheme with the recursive relation

$x_i + 1 = g(x_i)$, i = 0, 1, 2, . . .,

with some initial guess x_0 is called the fixed point iterative scheme.

The fixed point iteration method algorithm/flowchart work in such as way that modifications alongside iteration are progressively continued with the newer and fresher approximations of the initial approximation.

2.5.1 Solved Numerical-

Q. Find a root of $x^2 + x - 2 = 0$ using the fixed point method.
Sol^n : The given equation can be expressed as

$$x = 2 - x^2$$

Let us start with an initial value of $x_0 = 0$

$$x_1 = 2 - 0 = 2$$
$$x_2 = 2 - 4 = -2$$
$$x_3 = 2 - 4 = -2$$

Since $x_3 - x_2 = 0$, -2 is one of the roots of the equation.

2.5.2 Algorithm-

1. Start

2. Read values of x0 and e.
   Here x0 is the initial approximation e is the absolute error or the desired degree of accuracy, also the stopping criteria

3. Calculate x1 = g(x0)

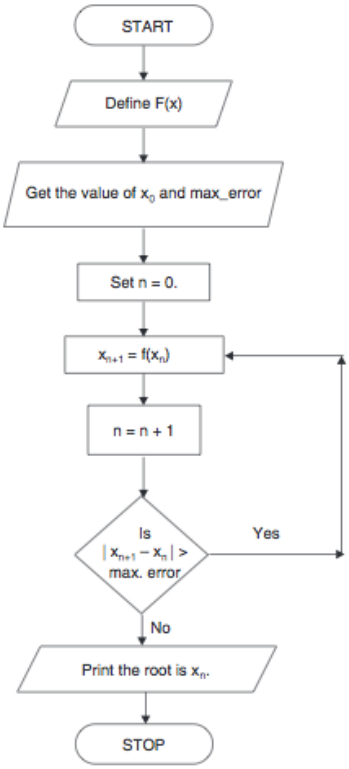
4. If $|x1–x0| <= e$, goto step 6.

5. Else, assign x0 = x1 and goto step 3.

6. Display x1 as the root.

7. Stop

2.5.3 Flow Chart-



2.5.4 Python Program-

```
def f(x):
    return(2-x*x)
x0=float(input("Enter value of X0 "))
p=int(input("Enter allowed precision "))
err=pow(10,-p)
itr=1
x=f(x0)
while (abs(x-x0)>err):
    print("After", itr," iterations value of root is",round(x,p+1))
    x0=x
    x=f(x0)
    itr=itr+1
print("Value of root is",round(x,p+1))
```

2.5.5 Fixed-Point Method Output

```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:24:06) [MSC v.1600 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Enter value of X0 0
Enter allowed precision 4
After 1  iterations value of root is 2.0
After 2  iterations value of root is -2.0
Value of root is -2.0
>>> ========== RESTART ==========
>>>
Enter value of X0 1
Enter allowed precision 4
Value of root is 1.0
```

```
>>>
```

2.6 Multiple roots by Newton's Method

A major requirement of all the methods that we have discussed so far is the evaluation of $f(x)$ at successive approximations x_k. Furthermore, Newton's method requires evaluation of the derivatives of $f(x)$ at each iteration.

If $f(x)$ happens to be a polynomial $P_n(x)$ (which is the case in many practical applications), then there is an extremely simple classical technique, known as **Horner's Method** to compute $P_n(x)$ at $x = z$.

- *The value $P_n(z)$ can be obtained recursively in n-steps from the coefficients of the polynomial $P_n(x)$.*

- *Furthermore, as a by-product, one can get the value of the derivative of $P_n(x)$ at $x = z$; that is $P_n'(z)$.* Then, the scheme can be incorporated in the Newton Method to compute a zero of $P_n(x)$.

(*Note that the Newton Method requires evaluation of $P_n(x)$ and its derivative at successive iterations*).

1.9.1 Horner's Method

Let $P_n(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$ and let $x = z$ be given.

We need to compute $P_n(z)$ and $P_n'(z)$.

Let's write $P_n(x)$ as:

$$P_n(x) = (x - z)\, Q_{n-1}\,(x) + b_o,$$

where $Q_{n-1}(x) = b_nx^{n-1} + b_{n-1}\,x^{n-2} + \cdots + b_2x + b_1$.

Thus, $P_n(z) = b_0$.

Let's see how to compute b_0 recursively by knowing the coefficients of $P_n(x)$

From above, we have

$$a_0 + a_1x + a_2x^2 + \cdots + a_nx^n = (x - z)(b_nx^{n-1} + b_{n-1}\,x^{n-2} + \cdots + b_1) + b_o$$

Comparing the coefficients of like powers of x from both sides, we obtain

$$\begin{aligned} b_n &= a_n \\ b_{n-1} - b_nz &= a_{n-1} \\ \Rightarrow b_{n-1} &= a_{n-1} + b_nz \\ &\vdots \end{aligned}$$

and so on.

In general, $b_k - b_{k+1}z = a_k$

$\Rightarrow b_k = a_k + b_{k+1}z.\ k = n-1, n-2, \ldots, 1, 0.$

Thus, knowing the coefficients $a_n, a_{n-1}, \ldots, a_1, a_0$ of $P_n(x)$, the coefficients $b_n, b_{n-1}, \ldots, b_1$ of $Q_{n-1}(x)$ can be computed recursively starting with $b_n = a_n$, as shown above. That is,

$$b_k = a_k + b_{k+1}z, \quad k = n-1, \quad k = 2, \ldots, 1, 0.$$

Again, note that $P_n(z) = b_0$. So, $P_n(z) = b_0 = a_0 + b_1\, z$.

That is, if we know the coefficients of a polynomial, we can compute the value of the polynomial at a given point out of these coefficients recursively as shown above.

Algorithm 1.23 (Evaluating Polynomial: $P_n(x) = a_0 + a_1 x_p + a_2 x^2 + \ldots + a_n x^n$ at $x = z$).

Input: (i) $a_0, a_1, ..., a_n$ - The coefficients of the polynomial $P_n(x)$.

(ii) z - The number at which $P_n(x)$ has to be evaluated.

Output: $b_0 = P_n(z)$.

Step 1. Set $b_n = a_n$.

Step 2. For $k = n-1, n-2, \ldots, 1, 0$ do.
Compute $b_k = a_k + b_{k+1} z$

Step 3. Set $P_n(z) = b_0$

End

It is interesting to note that as a by-product of above, we also obtain $P_n'(z)$, as the following computations show.

Computing $P_n'(z)$

$P_n(x) = (x - z)Q_{n-1}(x) + b_0$

Thus, $P_n'(x) = Q_{n-1}(x) + (x - z)Q_{n-1}'(x)$.

So, $P_n'(z) = Q_{n-1}(z)$.

Write $Q_{n-1}(x) = (x - z)R_{n-2}(x) + c_1$

Substituting $x = z$, we get $\boxed{Q_{n-1}(z) = c_1.}$

To compute c_1 from the coefficients of $Q_{n-1}(x)$ we proceed in the same way as before.

$$\text{Let } R_{n-2}(x) = c_n x^{n-2} + c_{n-1} x^{n-3} + \cdots + c_3 x + c_2$$

Then from above we have

$$b_n x^{n-1} + b_{n-1} x^{n-2} + \cdots + b_2 x + b_1 = (x - z)(c_n x^{n-2} + c_{n-1} x^{n-3} + \cdots + c_3 x + c_2) + c_1.$$

Equating the coefficients of like powers on both sides, we obtain

$$\begin{aligned} b_n &= c_n \\ \Rightarrow c_n &= b_n \\ b_{n-1} &= c_{n-1} - z c_n \\ \Rightarrow c_{n-1} &= b_{n-1} + z c_n \\ &\vdots \\ b_1 &= c_1 - z c_2 \\ \Rightarrow c_1 &= b_1 + z c_2 \end{aligned}$$

*Since b's have already been computed (by **Algorithm 1.23**), we can obtain c's from b's.*

Then, we have the following scheme to compute $P_n'(z)$:

Computing $P_n'(z)$

Inputs: (i) $b_1, ..., b_n$ - The coefficients of the polynomial $Q_{n-1}(x)$ obtained from the previous algorithm

(ii) z - The number at which $P_3'(x)$ has to be evaluated.

Output: $c_1 = Q_{n-1}(z)$.

Step 1. Set $c_n = b_n$,

Step 2. For $k = n-1, n-2, ...2, 1$ do
Compute $c_k = b_k + c_{k+1}\, z$,
End

Step 3. Set $P_n'(z) = c_1$.

Example 1.24

Given $P_3(x) = x^3 - 7x^2 + 6x + 5$. Compute $P_3(2)$ and $P_3'(2)$ using Horner's scheme.

Input Data:

(i) The coefficients of the polynomial: $a_0 = 5, a_1 = 6, a_2 = -7$
(ii) The point at which the polynomial and its derivative need to be evaluated: $z = 2$
(iii) The degree of the polynomial: n=3

Formula to be used:

$\begin{cases}$ (i) Computing b_k's from a_k's: $b_k = a_k + b_{k+1}z, k = n-1, n-2, \ldots, 0; P_3(z) = b_0.$
(ii) Computing c_k's from b_k's: $c_k = b_k + c_{k+1}z, k = n-1, n-2, \ldots, 1; P_3'(z) = c_1$

Solution.

- Compute $P_3(2)$.

  Generate the sequence $\{b_k\}$ from $\{a_k\}$:

$$\begin{aligned} b_3 &= a_3 = 1 \\ b_2 &= a_2 + b_3 z = -7 + 2 = -5 \\ b_1 &= a_1 + b_2 z = 6 - 10 = -4 \\ b_0 &= a_0 + b_1 z = 5 - 8 = -3. \end{aligned}$$

  So, $P_3(2) = b_0 = -3$

- Compute $P_3'(2)$.

  Generate the sequence $\{c_k\}$ from $\{b_k\}$.

$$\begin{aligned} c_3 &= b_3 = 1 \\ c_2 &= b_2 + c_3 z = -5 + 2 = -3 \\ c_1 &= b_1 + c_2 z = -4 - 6 = -10. \end{aligned}$$

  So, $P_3'(2) = -10.$

1.9.2 The Newton Method with Horner's Scheme

We now describe the Newton Method for a polynomial using Horner's scheme. Recall that the Newton iteration for finding a root of $f(x) = 0$ is:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

In case $f(x)$ is a polynomial $P_n(x)$ of degree n: $f(x) = P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$, the above iteration becomes:

$$x_{k+1} = x_k - \frac{P_n(x_k)}{P_n'(x_k)}$$

If the sequence $\{b_k\}$ and $\{c_k\}$ are generated using Horner's Scheme, at each iteration we then have

$$\boxed{x_{k+1} = x_k - \frac{b_0}{c_1}}$$

(Note that at the iteration $k, P_n(x_k) = b_0$ and $P_n'(x_k) = c_1$).

Algorithm 1.25 (The Newton Method with Horner's Scheme).

Input: $a_0, a_1, \cdots, a_n$ - The coefficients of $P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$.
x_0 - The initial approximation
ϵ - The tolerance
N - The maximum number of iterations to be performed.

Output: An approximation of the root $x = \xi$ or a failure message.

For $k = 0, 1, 2, \cdots$, do

Step 1. $z = x_k$, $b_n \equiv a_n$, $c_n \equiv b_n$

Step 2. **Computing b_1 and c_1**
For $j = n-1, n-2, \cdots, 1$ do
$b_j \equiv a_j + z\, b_{j+1}$
$c_j \equiv b_j + z\, c_{j+1}$
End.

Step 3. $b_0 = a_0 + z\, b_1$] Computing $P_n(x_k)$ and $P_n'(x_k)$

Step 4. $x_{k+1} = x_k - b_0/c_1$

Step 5. If $|x_{k+1} - x_k| < \epsilon$ or $k > N$, stop.

End

Note:

1. The outer-loop corresponds to the Newton Method

2. The inner-loop and the statement $b_0 = a_0 + zb_1$ correspond to the evaluation of $P_n(x)$ and its derivative $P_n'(x)$ at successive approximations.

Example 1.26

Find a root of $P_3(x) = x^3 - 7x^2 + 6x + 5 = 0$, using the Newton method and Horner's scheme, choosing $x_0 = 2$ (Note that $P_3(x) = 0$ has a root between 1.5 and 2).

Input Data:

$$\begin{cases} \text{(i) The coefficients of } P_3(x)\text{: } a_0 = 5, a_1 = 6, a_2 = -7, \text{ and } a_3 = 1. \\ \text{(ii) The degree of the polynomial: } n = 3. \\ \text{(iii) Initial approximation: } x_0 = 2. \end{cases}$$

Solution.

Iteration 1. ($k = 0$). Compute x_1 from x_0:

$$x_1 = x_0 - \frac{b_0}{c_1} = 2 - \frac{3}{10} = \frac{17}{10} = 1.7$$

Note that $b_0 = P_3(x_0)$ and $c_1 = P_3'(x_0)$ have already been computed in Example 1.23.

Iteration 2. ($k = 1$). Compute x_2 from x_1:

Step 1. $z = x_1 = 1.7,\ b_3 = 1,\ c_3 = b_3 = 1$

Step 2.

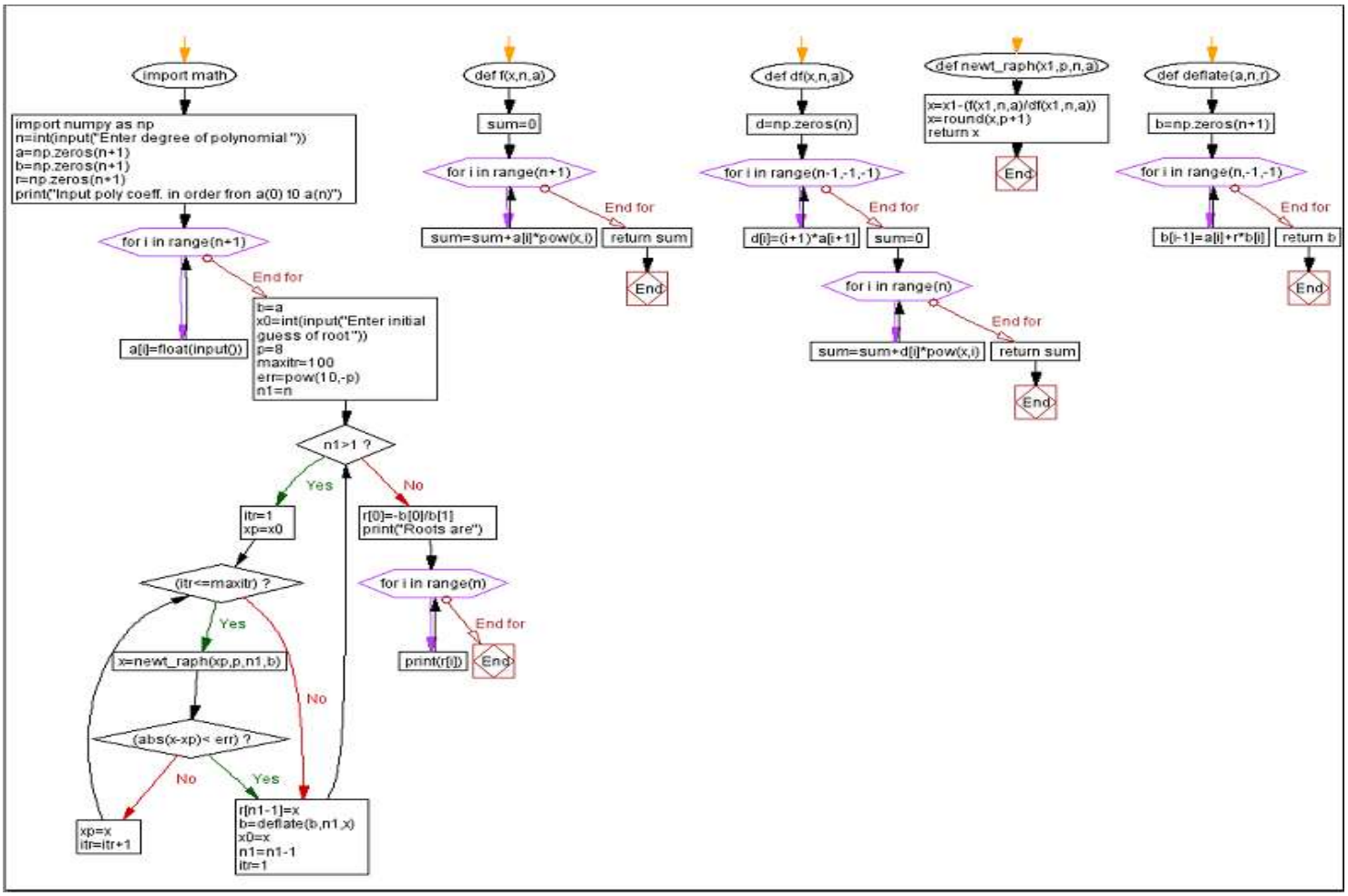
$$\begin{cases} b_2 &= a_2 + z\, b_3 = -7 + 1.7 = -5.3 \\ c_2 &= b_2 + z\, c_3 = -5.3 + 1.7 = -3.6 \\ b_1 &= a_1 + z\, b_2 = 6 + 1.7(-5.3) = -3.0100 \\ c_1 &= b_1 + z\, c_2 = -3.01 + 1.7(-3.6) = -9.130 \text{ (Value of } P_3'(x_1)). \end{cases}$$

Step 3. $b_0 = a_0 + z\, b_1 = 5 + 1.7(-3.0100) = -0.1170$ (Value of $P_3(x_1)$).

Step 4. $x_2 = x_1 - \dfrac{b_0}{c_1} = 1.7 - \dfrac{0.1170}{9.130} = 1.6872$

(The **exact root**, correct up to four decimal places *is 1.6872*).

2.6.1 Flowchart



2.6.2 Python Program-

```
import math
import numpy as np
def f(x,n,a):
    sum=0
    for i in range(n+1):
        sum=sum+a[i]*pow(x,i)
    return sum
def df(x,n,a):
    d=np.zeros(n)
    for i in range(n-1,-1,-1):
       d[i]=(i+1)*a[i+1]
    sum=0
    for i in range(n):
        sum=sum+d[i]*pow(x,i)
    return sum
def newt_raph(x1,p,n,a):
    x=x1-(f(x1,n,a)/df(x1,n,a))
    x=round(x,p+1)
    return x
def deflate(a,n,r):
    b=np.zeros(n+1)
    for i in range(n,-1,-1):
        b[i-1]=a[i]+r*b[i]
    return b
n=int(input("Enter degree of polynomial "))
a=np.zeros(n+1)
b=np.zeros(n+1)
r=np.zeros(n+1)

print("Input poly coeff. in order fron a(0) t0 a(n)")
for i in range(n+1):
    a[i]=float(input())
b=a
x0=int(input("Enter initial guess of root "))
p=8
maxitr=100
err=pow(10,-p)
n1=n
while n1>1:
    itr=1
    xp=x0
    while(itr<=maxitr):
        x=newt_raph(xp,p,n1,b)
        if(abs(x-xp)< err):
            break
        xp=x
        itr=itr+1
    r[n1-1]=x
    b=deflate(b,n1,x)
    x0=x
    n1=n1-1
    itr=1
r[0]=-b[0]/b[1]
print("Roots are")
for i in range(n):
    print(r[i])
```

2.6.3 Multiple roots by Newton's Method Output

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:\Python37\harsh\Roots of Nonlinear Equations\newton_multiple().py
Enter degree of polynomial 2
Input poly coeff. in order fron a(0) t0 a(n)
2
-3
1
Enter initial guess of root 0
Roots are
2.0
1.0
>>>
```

Bairstow's method is an algorithm used to find the roots of a polynomial of arbitrary degree (usually order 3 and higher). The method divides a polynomial

$$f_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \qquad \textbf{(2.8)}$$

by a quadratic function $(x^2 - rx - s)$, where *r* and *s* are guessed. The division gives us a new polynomial

$$f_{n-2}(x) = b_2 + b_3 x + b_4 x^2 + \ldots + b_n x^{n-2} \qquad \textbf{(2.9)}$$

and the remainder $R(x) = b_1(x - r) + b_0$ **(2.10)**

The quotient $f_{n-2}(x)$ and the remainder $R(x)$ are obtained by the standard polynomial division.

The coefficients b_i can be calculated by the following recurrence relationship

$$b_n = a_n \qquad \textbf{(2.11a)}$$

$$b_{n-1} = a_{n-1} + r b_n \qquad \textbf{(2.11b)}$$

$$b_i = a_i + r b_{i+1} + s b_{i+2}\,, \quad \text{for } i = n-2 \text{ to } 0 \qquad \textbf{(2.11c)}$$

If $(x^2 - rx - s)$ is an exact factor of $f_n(x)$ then the remainder $R(x)$ is zero and the roots of $(x^2 - rx - s)$ are the roots of $f_n(x)$. The Bairstow's method reduces to determining the value of *r* and *s* such that $R(x) = 0$, hence, $b_0 = b_1 \to 0$. Because b_0 and b_1 are functions of *r* and *s*, they can be expanded using Taylor series, as:

$$b_0(r + \Delta r, s + \Delta s) = b_0 + \frac{\partial b_0}{\partial r}\Delta r + \frac{\partial b_0}{\partial s}\Delta s \qquad \textbf{(2.12a)}$$

$$b_1(r + \Delta r, s + \Delta s) = b_1 + \frac{\partial b_1}{\partial r}\Delta r + \frac{\partial b_1}{\partial s}\Delta s \qquad \textbf{(2.12b)}$$

By setting both equations equal to zero

$$\frac{\partial b_0}{\partial r}\Delta r + \frac{\partial b_0}{\partial s}\Delta s = -b_0 \qquad \textbf{(2.13a)}$$

$$\frac{\partial b_1}{\partial r}\Delta r + \frac{\partial b_1}{\partial s}\Delta s = -b_1 \qquad \textbf{(2.13b)}$$

To solve the system of equations above, we need partial derivatives of b_0 and b_1 with respect to *r* and *s*. Bairstow showed that these partial derivatives can be obtained by a synthetic division of the *b*'s in a fashion similar to the

way in which the b's themselves were derived, that is by replacing a_i's with b_i's, and b_i's with c_i's, such that,

$$c_n = b_n \tag{2.14a}$$

$$c_{n-1} = b_{n-1} + rc_n \tag{2.14b}$$

$$c_i = b_i + rc_{i+1} + sc_{i+2}\,, \quad \text{for } i = n-2 \text{ to } 1 \tag{2.14c}$$

where

$$\frac{\partial b_0}{\partial r} = c_1\,, \quad \frac{\partial b_0}{\partial s} = \frac{\partial b_1}{\partial r} = c_2\,, \quad \text{and} \quad \frac{\partial b_1}{\partial s} = c_3\,.$$

Then substituting into equations (2.13a) and (2.13b) gives

$$c_1\Delta r + c_2\Delta s = -b_0$$
$$c_2\Delta r + c_3\Delta s = -b_1$$

These equations can be solved for Δr and Δs, which can be used to improved the initial guess of r and s.

At each step, an approximate error in r and s estimated as

$$|\epsilon_{a,r}| = \left|\frac{\Delta r}{r}\right| 100\% \quad \text{and} \quad |\epsilon_{a,s}| = \left|\frac{\Delta s}{s}\right| 100\% \tag{2.15}$$

If $|\epsilon_{a,s}| < \epsilon_s$ and $|\epsilon_{a,r}| < \epsilon_s$ where ϵ_s is a stopping criterion, the values of the roots can be determined by

$$x = \frac{r \pm \sqrt{r^2 + 4s}}{2} \qquad \textbf{(2.16)}$$

At this point, there exist three possibilities

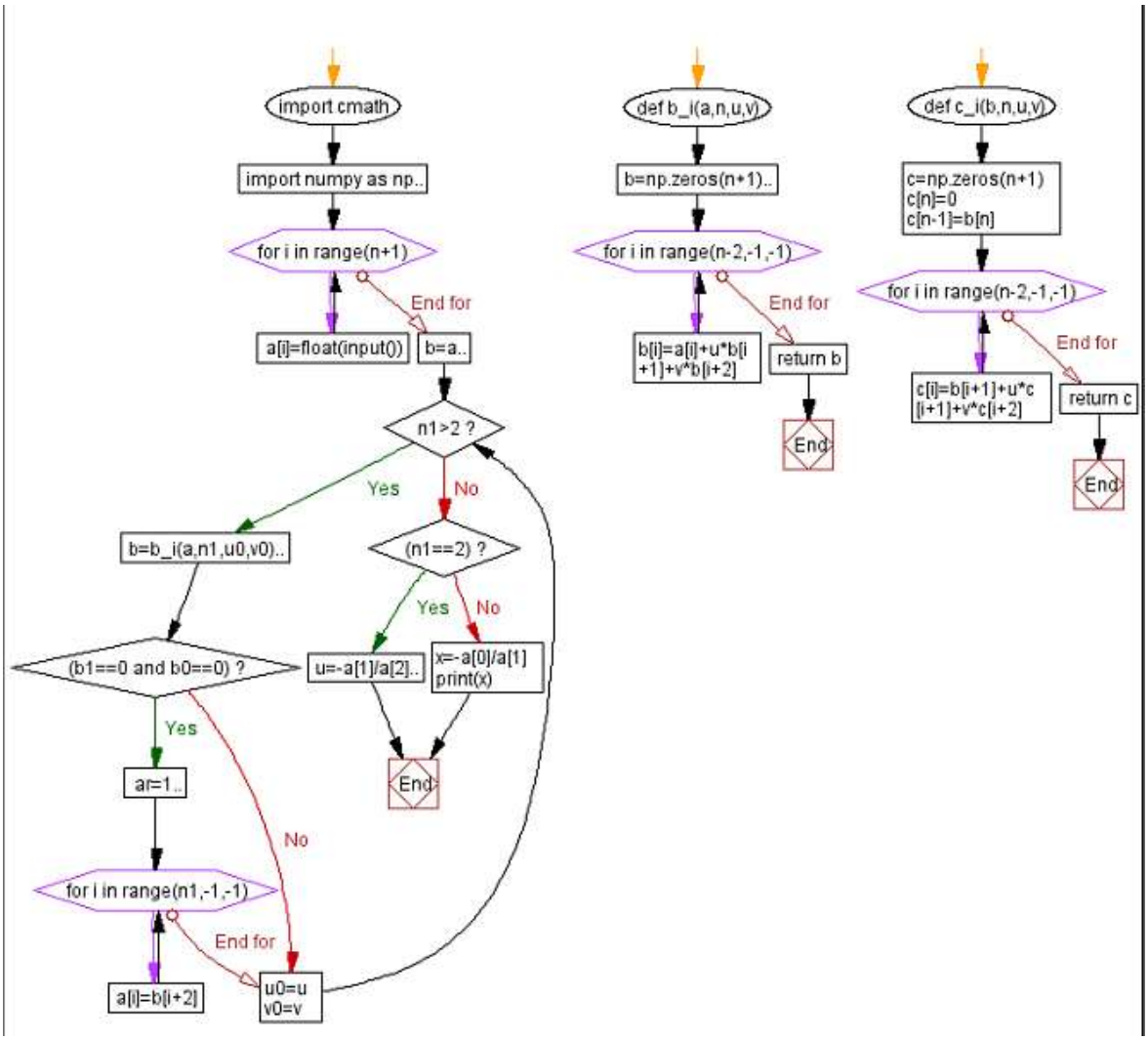
(1) If the quotient polynomial f_{n-2} is a third (or higher) order polynomial, the Bairstow's method can be applied to the quotient to evaluate new values for r and s. The previous values of r and s can serve as the starting guesses.

(2) If the quotient polynomial f_{n-2} is a quadratic function, then use eqn. (2.16) to obtain the remaining two roots of $f_n(x)$.

(3) If the quotient polynomial f_{n-2} is a linear function, then the single remaining root is given by

$$x = -\frac{s}{r} \qquad \textbf{(2.17)}$$

2.7.1 Flowchart



2.7.2 Python Program-

```
import cmath
import numpy as np
def b_i(a,n,u,v):
    b=np.zeros(n+1)
    b[n]=a[n]
    b[n-1]=a[n-1]+u*b[n]
    for i in range(n-2,-1,-1):
        b[i]=a[i]+u*b[i+1]+v*b[i+2]
    return b
def c_i(b,n,u,v):
    c=np.zeros(n+1)
    c[n]=0
    c[n-1]=b[n]
    for i in range(n-2,-1,-1):
        c[i]=b[i+1]+u*c[i+1]+v*c[i+2]
    return c
n=int(input("Enter degree of polynomial "))
a=np.zeros(n+1)
b=np.zeros(n+1)
c=np.zeros(n+1)
print("Input poly coeff. in order fron a(0) t0 a(n)")
for i in range(n+1):
    a[i]=float(input())
b=a
u0=float(input("Enter initial values of U0 "))
v0=float(input("Enter initial values of V0 "))
n1=n
while n1>2:
    b=b_i(a,n1,u0,v0)
    c=c_i(b,n1,u0,v0)
    D=c[1]*c[1]-c[0]*c[2]
    du=-(b[1]*c[1]-b[0]*c[2])/D
    dv=-(b[0]*c[1]-b[1]*c[0])/D
    u=u0+du
    v=v0+dv
    b1=a[1]+u*b[2]+v*b[3]
    b0=a[0]+u*b[1]+v*b[2]
    if(b1==0 and b0==0):
        ar=1
        br=-u
        cr=-v
        x1=(-br+cmath.sqrt(br*br-4*ar*cr))/2*ar
        x2=(-br-cmath.sqrt(br*br-4*ar*cr))/2*ar
        print("Roots are",x1,x2)
        n1=n1-2
        for i in range(n1,-1,-1):
            a[i]=b[i+2]
    u0=u
    v0=v
if(n1==2):
    u=-a[1]/a[2]
    v=-a[0]/a[2]
    x1=(-br+cmath.sqrt(br*br-4*ar*cr))/2*ar
    x2=(-br-cmath.sqrt(br*br-4*ar*cr))/2*ar
    print(x1,x2)
else:
    x=-a[0]/a[1]
    print(x)
```

2.7.3 Complex roots by Baristow Method Output

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:\Python37\harsh\Roots of Nonlinear Equations\baristo().py ==========
Enter degree of polynomial 3
Input poly coeff. in order fron a(0) t0 a(n)
10
1
0
1
Enter initial values of U0 1.8
Enter initial values of V0 -4.0
Roots are (1+2j) (1-2j)
-2.0
>>>
```

2.8 Comparison of all Methods

	Bisection	**Newton-Raphson**	**Secant**	**Bairstow**	**Regula-falsi**	**newtons multiple root**	**Fixed point**
Type	Closed.	Open.	Open.	Open.	Closed.	Open	Open
Requirements	A function f(x) and an interval [a, b] (in which f(x) is continuous) containing one and only one root.	A function f(x), the derivative df(x) of f(x) and a starting point.	A function f(x) and two starting points (different).	A polynomic function f(x) and two starting values.	A function f(x) and an interval [a, b] (in which f(x) is continuous) containing one and only one root.	given equation f(x) = 0 has multiple roots at a point say 'x = s' then the order of convergence decreases for any iterative method.	The transcendental equation **f(x) = 0** can be converted algebraically into the form **x = g(x)** and then using the iterative scheme with the recursive relation
Risks	If the function is discontinuous, the program can have an error when evaluating. Also, if the interval has more or less than one roots, unexpected behavior can happen.	If the function presents small slopes in any iteration, the method diverges. On the other hand, if the function presents really big slopes in any iteration, the method converges really slow.	Similar as Newton Raphson. Also, this method can fail if both starting values are similar.	The farther the starting values from the roots, the longer it takes to converge.	The convergence of the regula falsi method can be very slow in some cases	-	Sometime process does not converge to the solution. Like oscillatory and monotone divergence
Convergence rate	Linear	Quadratic.	Aureal number.	Quadratic.	Linear	Quadratic.	least quadratic.
Advantages	If the criteria for the interval is satisfied, it always	This method has the quickest converging rate. Also, it just requires a	It runs quicker than bisection and does	This method allows us to find complex roots, and it may take a	It always converges. It does not require the	Newton's method for finding a real or complex	we can solve both linear and nonlinear equations using this method.

	converges. Also, this is the easiest method to program and the easier to understand. The smaller the interval given, the faster it converges	starting point instead of two.	not require a derivative.	long time, but it doesn't fail.	derivative. It is a quick method.	root of a function is very efficient near a simple root because the algorithm converges quadratically in the neighborhood of such a root.	
Disadvantages	This is the method with the lowest converge rate.	This method requires the derivative, which can be hard to calculate and/or evaluate. Also, small slope functions (such as log(x)) diverge.	Can fall in the same errors as Newton-Raphson.	Only works for polynomial functions. Really hard to implement and understand.	It may slowdown in unfavourable situations.	-	Sometime process does not converge to the solution. Like oscillatory and monotone divergence
Fault tolerance	As long as the criteria are satisfied, this method doesn't run.	Some functions, such as periodic functions, can lead to unexpected behaviors with certain values. Also, small slopes fail.	Fails less than Newton-Raphson, but still can have unexpected behaviours.	This method doesn't fail.	Regula Falsi Method fails to identify multiple different roots,	-	oscillatory divergence, monotone divergence
Number of roots on each run	One.	One.	One.	All.	One	One	One

3 Solution to Partial Differential Equations

Partial Differential Equation (or briefly a PDE) is a mathematical equation that involves two or more independent variables, an unknown function (dependent on those variables), and partial derivatives of the unknown function with respect to the independent variables. The order of a partial differential equation is the order of the highest derivative involved. A solution (or a particular solution) to a partial differential equation is a function that solves the equation or, in other words, turns it into an identity when substituted into the equation. A solution is called general if it contains all particular solutions of the equation concerned.

The general linear partial differential equation of the second order in the two independent variable is of the type

$$A(x,y)\frac{\partial^2 u}{\partial x^2} + B(x,y)\frac{\partial^2 u}{\partial x \partial y} + C(x,y)\frac{\partial^2 u}{\partial y^2} + F\big(x, y, u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\big) = 0$$

Such a partial differential equation is said to be

- **Elliptic if** $B^2 - 4AC < 0$
- **Parabolic if** $B^2 - 4AC = 0$
- **Hyperbolic if** $B^2 - 4AC > 0$

Elliptic Equation The Laplace Equation $\Delta^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$ and the Poisson's Equation $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)$ are the example of elliptic partial differential equation. Laplace equation arises in steady-state flow and potential problems. Poisson's equation arises in fluid mechanics, electricity and magnetism and torsion problems.

3.1 Solution of Laplace Equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{8}$$

Consider a rectangular region R for which $u(x,y)$ is known at the boundary. Divide this region into a network of square mesh of side h, as shown in the Fig. 1 (assuming that an extra subdivision of R is possible). Use the **Standard 5-point formula** to find value of u at any mesh point as the average of values at the four neighbouring points to the left, right, above and below.

$$u_{i,j} = \frac{1}{4}[u_{i-1,j} + u_{i+1,j} + u_{i,j+1} + u_{i,j-1}] \tag{9}$$

sometimes a formula similar to 9 is used which is called **Diagonal 5-point formula** given by

$$u_{i,j} = \frac{1}{4}[u_{i-1,j+1} + u_{i+1,j-1} + u_{i+1,j+1} + u_{i-1,j-1}] \tag{10}$$

Having found all the interior values of $u_{i,j}$ once, there accuracy can be improved by two iterative methods described below -

1. **Jacobi's Method**: Denoting the n^{th} iterative value of $u_{i,j}$ by $u^n_{i,j}$, the iterative formula to solve 9 is

   $$u^{n+1}_{i,j} = \tfrac{1}{4}[u^{(}_{i-1,j}n) + u^{(}_{i+1,j}n) + u^{(}_{i,j+1}n) + u^{(}_{i,j-1}n)]$$

   It gives improved values of $u_{i,j}$ at the interior mesh points and is called the **point jacobi formula**.

2. **Gauss-Seidal Method**: The iteration formula is

   $$u^{n+1}_{i,j} = \tfrac{1}{4}[u^{(}_{i-1,j}n+1) + u^{(}_{i+1,j}n) + u^{(}_{i,j+1}n+1) + u^{(}_{i,j-1}n)]$$

   It utilises the latest values available and scans the mesh points symmetrically from left to right along successive rows.

3.1.1 Numerical

Consider a steel plate of size $15cm \times 15cm$.If two of the sides are held at 100^oC,what are the steady state temperature at interior points assuming a grid size of $5cm \times 5cm$.

100	100	100	
100	f_1	f_2	0
100	f_3	f_4	0
	0	0	0

The system of equations is as follows:
At point 1:$f_2 + f_3 - 4f_1 + 100 + 100 = 0$
At point 2:$f_1 + f_4 - 4f_2 + 100 + 0 = 0$
At point 3:$f_1 + f_4 - 4f_3 + 100 + 0 = 0$
At point 4:$f_2 + f_3 - 4f_4 + 0 + 0 = 0$
That is,

$$f_2 + f_3 - 4f_1 = -200$$
$$f_1 + f_4 - 4f_2 = -100$$
$$f_1 + f_4 - 4f_3 = -100$$
$$f_2 + f_3 - 4f_4 = 0$$

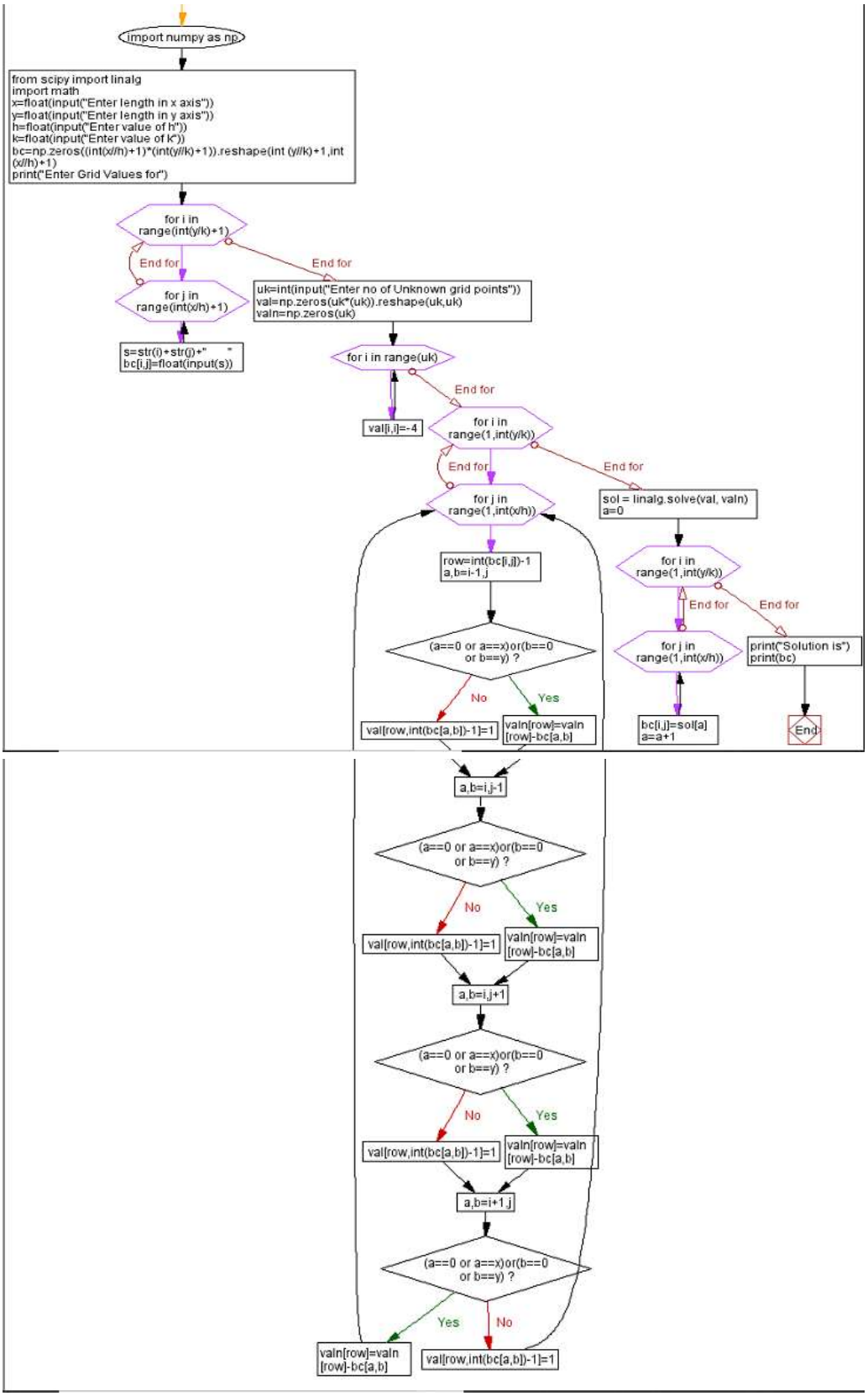
Solution of this sytem are:

$f_1 = 75$ $f_2 = 50$
$f_3 = 50$ $f_4 = 25$

3.1.2 Algorithm

Step1: Take input values x,y,h and k
Step2: Intialize array bc[y//k+1][x//h+1]
Step3:Input grid values of bc;unknown grid points uk an values in array valn[uk][uk]
Step4:Intialize i=1 and check whether i¡y//k
Step5:If true, intialize j=1 check whether j¡x//h
Step6:If true, then raw=int(bc[i,j]-1)
Step7:Then operate on a=0 or a=x or b=0 or b=y
Step8:If true valn[row]=valn[row]-bc[a,b]
Step9:If false, val[row,int(bc[a,b]-1)=1]
Step10:It is repeat for a,b=i,j-1;i,j+1;i+1,j;i-1,j
Step11:Then by using sol=linalg.solve(val,valn)
Step12:If step 4 get false then intialize i=1
Step13: Check whetheri,y//k
Step14:If true then initalize j=1
Step15: Check thenj,x//h and process through bc[i,j]=sol[a],a=a+1
Step16:Finally print bc and get output

3.1.3 Flowchart



3.1.4 Python Program-

```
import numpy as np
from scipy import linalg
import math
x=float(input("Enter length in x axis"))
y=float(input("Enter length in y axis"))
h=float(input("Enter value of h"))
k=float(input("Enter value of k"))
bc=np.zeros((int(x//h)+1)*(int(y//k)+1)).reshape(int (y//k)+1,int (x//h)+1)
print("Enter Grid Values for")
for i in range(int(y/k)+1):
    for j in range(int(x/h)+1):
        s=str(i)+str(j)+" "
        bc[i,j]=float(input(s))
uk=int(input("Enter no of Unknown grid points"))
val=np.zeros(uk*(uk)).reshape(uk,uk)
valn=np.zeros(uk)
for i in range(uk):
    val[i,i]=-4
for i in range(1,int(y/k)):
    for j in range(1,int(x/h)):
                row=int(bc[i,j])-1
                a,b=i-1,j
                if (a==0 or a==x)or(b==0 or b==y):
                    valn[row]=valn[row]-bc[a,b]
                else :
                    val[row,int(bc[a,b])-1]=1
                a,b=i,j-1
                if (a==0 or a==x)or(b==0 or b==y):
                    valn[row]=valn[row]-bc[a,b]
                else :
                    val[row,int(bc[a,b])-1]=1
                a,b=i,j+1
                if (a==0 or a==x)or(b==0 or b==y):
                    valn[row]=valn[row]-bc[a,b]
                else :
                    val[row,int(bc[a,b])-1]=1
                a,b=i+1,j
                if (a==0 or a==x)or(b==0 or b==y):
                    valn[row]=valn[row]-bc[a,b]
                else :
                    val[row,int(bc[a,b])-1]=1
sol = linalg.solve(val, valn)
a=0
for i in range(1,int(y/k)):
    for j in range(1,int(x/h)):
        bc[i,j]=sol[a]
        a=a+1

print("Solution is")
print(bc)
```

3.1.5 Method Output

```
jhii@jhii:~/Documents/Python$ python3 lech.py
Enter length in x axis3
Enter length in y axis3
Enter value of h1
Enter value of k1
Enter Grid Values for
00      100
01      100
02      100
03      inf
10      100
11      1
12      2
13      0
20      100
21      3
22      4
23      0
30      inf
31      0
32      0
33      0
Enter no of Unknown grid points4
Solution is
[[100. 100. 100.  inf]
 [100.  75.  50.   0.]
 [100.  50.  25.   0.]
 [ inf   0.   0.   0.]]
```

3.2 Solution of Poisson Equation

$$A(x,y)\frac{\partial^2 u}{\partial x^2} + B(x,y)\frac{\partial^2 u}{\partial x \partial y} + C(x,y)\frac{\partial^2 u}{\partial y^2} + F\big(x, y, u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\big) = 0$$

when A=1,B=0, C=1 and $F\big(x, y, u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\big) = g(x,y)$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = g(x,y) \tag{11}$$

$$\bigtriangledown^2 f = g(x,y)$$

This equation is know as poission's equation. Using the notation $g_{ij} = g(x_i, y_j)$

The finite difference formula for solving Poission's equation then takes the form

$$f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1} - 4f_{ij} = h^2 g_{ij}$$

By applying the replacent formula to each grid point in the domain of consedration, we will get a system of linear equations in terms of f_{ij}. These equations may be solved either by any of the elimination methods or by any iteration techniques as done in solving Laplace's equation.

3.2.1 Numerical

Solve the poission equation

$$\bigtriangledown^2 f = 2x^2 y^2$$

over the square domain $0 \leq x \leq 3$ and $0 \leq y \leq 3$ with f=0 on the boundary and h=1.

0	0	0	0
y=2	f_1	f_2	0
y=1	f_3	f_4	0
0	x=1	x=2	0

The domain is divided into squares of one unit size as illustrated below: By applying equation at each grid point, we get the following set of equations:

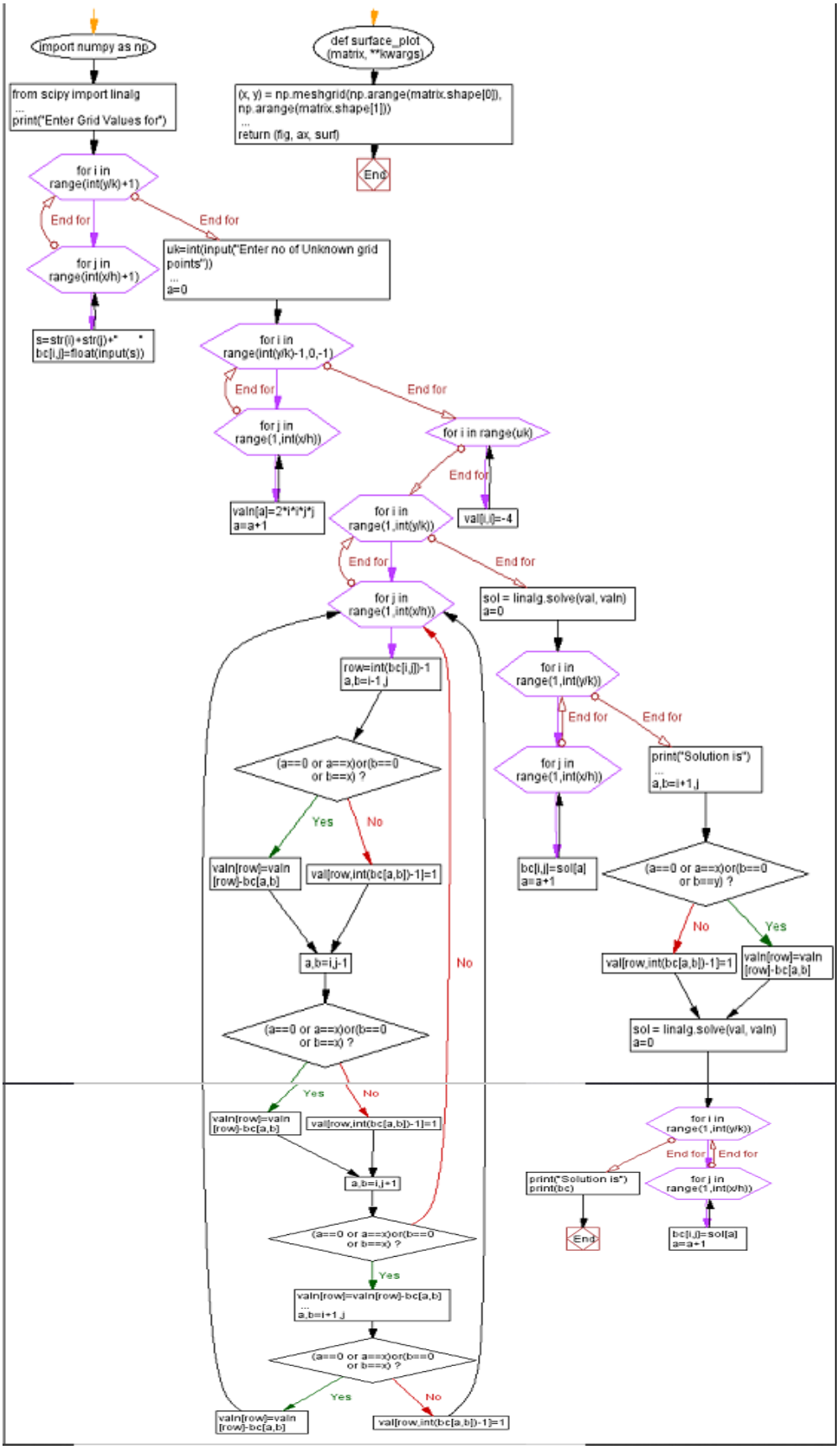
$$\text{Point1: } 0+0+f_2+f_3-4f_1 = 2(1)^2(2)^2$$
$$\text{i.e. } f_2+f_3-4f_1 = 8$$
$$\text{Point2: } 0+0+f_1+f_4-4f_2 = 2(2)^2(2)^2$$
$$\text{i.e. } f_1+f_4-4f_2 = 32$$
$$\text{Point3: } 0+0+f_1+f_4-4f_3 = 2(1)^2(1)^2$$
$$\text{i.e. } f_2+f_3-4f_1 = 2$$
$$\text{Point4: } 0+0+f_2+f_3-4f_4 = 2(2)^2(1)^2$$
$$\text{i.e. } f_2+f_3-4f_4 = 8$$

Solving the equations by elimination method, we get the answers.

$$f_1 = \frac{-22}{4} \qquad f_2 = \frac{-43}{4}$$

$$f_3 = \frac{-13}{4} \qquad f_4 = \frac{-22}{4}$$

3.2.2 Flowchart



3.2.3 Algorithm

Step1: Take input values x,y,h and k
Step2: Intialize array bc[y//k+1][x//h+1]
Step3:Input grid values of bc;unknown grid points uk an values in array valn[uk][uk]
Step4:Check whetheri=y//k-1 if i¿0
Step5 Then intialize j=1 and process through valn[a]=2*i*i*j*j,a=a+1
Step5:Intialize i=1 and check whether i¡y//k
Step6:If true, intialize j=1 check whether j¡x//h
Step7:If true, then raw=int(bc[i,j]-1)
Step8:Then operate on a=0 or a=x or b=0 or b=y
Step9:If true valn[row]=valn[row]-bc[a,b]
Step10:If false, val[row,int(bc[a,b]-1)=1]
Step11:It is repeat for a,b=i,j-1;i,j+1;i+1,j;i-1,j
Step12:Then by using sol=linalg.solve(val,valn)
Step13:If step 4 get false then intialize i=1
Step14: Check whetheri,y//k
Step15:If true then initalize j=1
Step16: Check thenj,x//h and process through bc[i,j]=sol[a],a=a+1
Step17:Finally print bc and get output

3.2.4 Python Program-

```
import numpy as np
from scipy import linalg
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

x=float(input("Enter length in x axis"))
y=float(input("Enter length in y axis"))
h=float(input("Enter value of h"))
k=float(input("Enter value of k"))
bc=np.zeros((int(x//h)+1)*(int(y//k)+1)).reshape(int (y//k)+1,int (x//h)+1)
print("Enter Grid Values for")
for i in range(int(y/k)+1):
    for j in range(int(x/h)+1):
        s=str(i)+str(j)+" "
        bc[i,j]=float(input(s))
uk=int(input("Enter no of Unknown grid points"))
val=np.zeros(uk*(uk)).reshape(uk,uk)
valn=np.zeros(uk)
a=0
for i in range(int(y/k)-1,0,-1):
    for j in range(1,int(x/h)):
         valn[a]=2*i*i*j*j
         a=a+1
for i in range(uk):
    val[i,i]=-4
for i in range(1,int(y/k)):
    for j in range(1,int(x/h)):
        row=int(bc[i,j])-1
        a,b=i-1,j
        if (a==0 or a==x)or(b==0 or b==x):
            valn[row]=valn[row]-bc[a,b]
        else :
            val[row,int(bc[a,b])-1]=1
        a,b=i,j-1
        if (a==0 or a==x)or(b==0 or b==x):
            valn[row]=valn[row]-bc[a,b]
```

```
            else :
                val[row,int(bc[a,b])-1]=1
            a,b=i,j+1
            if (a==0 or a==x)or(b==0 or b==x):
                valn[row]=valn[row]-bc[a,b]
            else :
                val[row,int(bc[a,b])-1]=1
            a,b=i+1,j
            if (a==0 or a==x)or(b==0 or b==x):
                valn[row]=valn[row]-bc[a,b]
            else :
                val[row,int(bc[a,b])-1]=1
sol = linalg.solve(val, valn)
a=0
for i in range(1,int(y/k)):
    for j in range(1,int(x/h)):
        bc[i,j]=sol[a]
        a=a+1

print("Solution is")
print(bc)

def surface_plot (matrix, **kwargs):
    (x, y) = np.meshgrid(np.arange(matrix.shape[0]), np.arange(matrix.shape[1]))
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    surf = ax.plot_surface(x, y, matrix, **kwargs)
    return (fig, ax, surf)


(fig, ax, surf) = surface_plot(bc, cmap=plt.cm.coolwarm)

fig.colorbar(surf)

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('F(X,Y)')

plt.show()
```

3.2.5 Method Output

```
jhii@jhii:~/Documents/Python$ python3 pe.py
Enter length in x axis3
Enter length in y axis3
Enter value of h1
Enter value of k1
Enter Grid Values for
00      0
01      0
02      0
03      0
10      2
11      1
12      2
13      0
20      1
21      3
22      4
23      0
30      0
31      1
32      2
33      0
Enter no of Unknown grid points4
Solution is
[[  0.           0.           0.           0.        ]
 [  2.          -4.66666667 -10.33333333   0.        ]
 [  1.          -2.33333333  -4.66666667   0.        ]
 [  0.           1.           2.           0.        ]]
```

4 Curve Fitting using Method of Least Squares

In many branch of applied mathematics, it is required to express a given data, obtain from the observation in the form of law connecting to the variables involved. Such a law inferred by some scheme, is known as empirical law. But the graphical method has the obvious drawback of being unable to give a unique curve of fit. The principle of least square, however, an elegant procedure of fitting a unique curve to a given data.

Let the curve $y = a + bx + cx^2 + ... + kx^n$ ....(1)
be fit to the set to the data points $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$.

Now we have to determine the constants $a, b, c, ...k$ such that it represent the curve of the best fit. In this case $n = m$ on substitute the values (x_i, y_i) in (1), we get n equations from which a unique set of n constraints can be found. But when n ¿ m, we have to obtain n equations which are more than m constants and hence can not be solved for these constants. so we try to determine the values of $a, b, c, ..., k$ which satisfies all the equations as nearly as possible and thus may give the best fit. In such a case we may apply the principle of least square.

At $x = x_i$ the observed(Experimental) values of the ordinate is y_i and the corresponding values on the fitting curve (1) is $y = a + bx + cx^2 + ... + kx^n$ $(= \eta_i, say)$ which is the expected(or calculated) value. The difference of the observed and the expected values i.e. $y_i - \eta_i (= e_i)$ is called the error(or residual) at $x = x_i$. Clearly some of the errors $e_1, e_2, ..., e_n$ will be positive and other will be negative. Thus to give equal weightage to each error, we square each of these and form their sum i.e. $E = {e_1}^2 + e - 2^2 + ... + {e_n}^2$.

The curve of the best fit is that for which e's are small as possible i.e. the sum of the squares of the errors is a minimum. This is known as the principle of least square and was suggested by a French mathematician Adrien Marie Legendre in 1806.

4.1 Regression curve fitting of linear curve

Q.Fit a straight line to the following data.

x	6	7	7	8	8	8	9	9	10
y	5	5	4	5	4	3	4	3	3

Sol. Let the straight line $y = a * x + b$
Then the normal equations are $\sum y = a * \sum y + 9 * b$
$\sum xy = a \sum x^2 = a * \sum x^2 + b * \sum y$
Then the values of $\sum x, \sum y etc. are calculated below$

x	y	xy	x^2
6	5	30	36
7	5	35	49
7	4	28	49
8	5	40	64
8	4	32	64
8	3	24	64
9	4	36	81
9	3	27	81
10	3	30	100
$\sum x = 72$	$\sum y = 36$	$\sum xy = 282$	$\sum x^2 = 588$

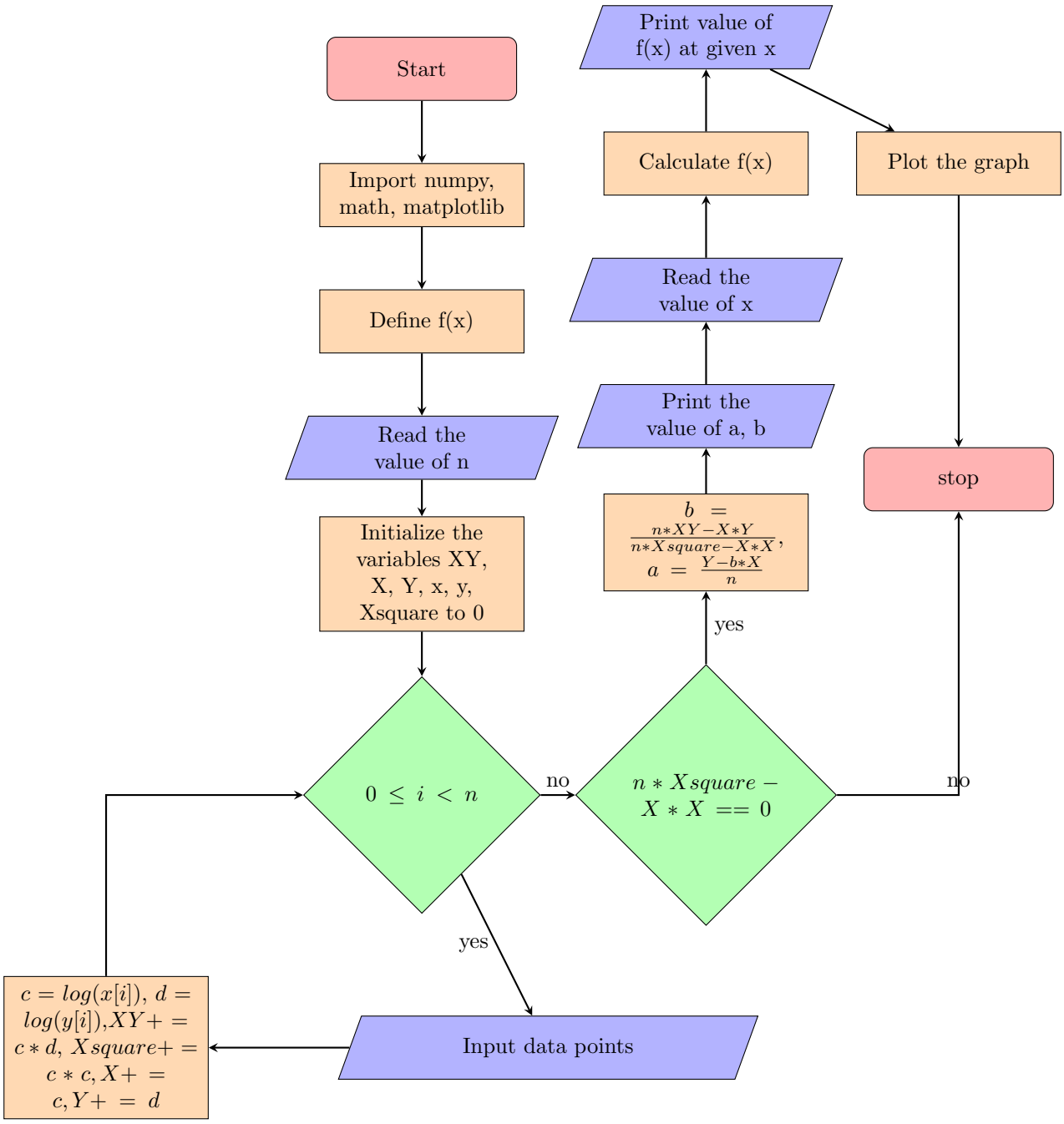
Therefore the equation (i) become $36 = 72a + 9b$ and $282 = 588a + 72b$
i.e. $8a + b = 4$ .....(ii)
$98a + 12b = 47$ .....(iii)
Multiply (ii) by 12 and subtracting from (iii), we get $a = -0.5$.
From (ii), $b = 8$.
Hence the required line for the best fit is $y = -0.5x + 8$.

4.1.1 Algorithm

Step 1: Start
Step 2: Define function f(x)
Step 3: Import numpy, matpllotlib
Step 4: Read the values of n(number of terms)
Step 5: Initialise all the variables :- XY, X, Y, Xsquare, x, y, to 0

Step 6: Read in the date points and increment the appropriate variables
Step 7: Check n*Xsquare - X*X = 0
(a)Yes- Curve can't be fitted in the graph and go to step 11.
(b)No - calculate the value of a and b
Step 8: Print the solution i.e. the value of a and b
Step 9: Evaluate the value of y at given x
Step 10: Plot the graph of curve
Step 11: Stop

4.1.2 Flow Chart-



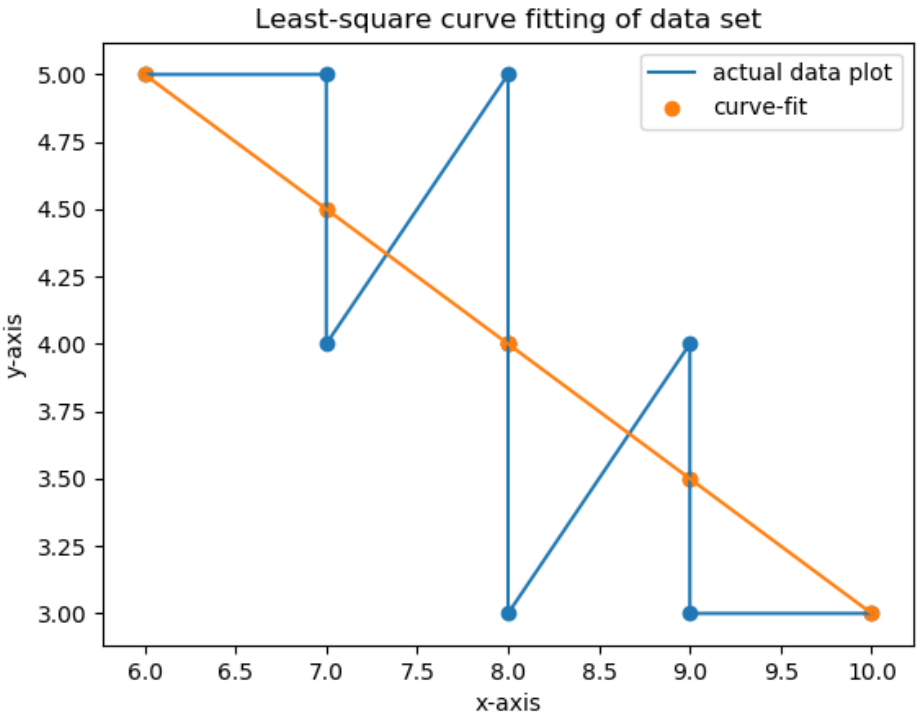
4.1.3 Python Program-

```
def f(a,b,x):
    return(a+b*x)
import numpy as np
import matplotlib.pyplot as plt
n=int(input("Enter the number of terms :"))
XY=0
Xsquare=0
Y=0
X=0
x=np.zeros(n)
y=np.zeros(n)
for i in range(0,n):
    x[i],y[i]=map(float,input("Enter elements").split())
    XY= x[i]*y[i]+XY
    Xsquare= Xsquare + x[i]*x[i]
    X= X + x[i]
    Y= Y + y[i]
if (n*Xsquare - X*X ==0):
    print(" The value of given data cannot be fitted as linear curve")
    exit()
else:
    b= ((n*XY - X*Y)/(n*Xsquare - X*X))
    a= (Y/n) - ((b*X)/n)
    print("the values of a and b for the following equation y=a+bx are :", a, b)
    g= a+b*x
    h = (g-a)/b
    xu = float(input("Enter the value of x for which the value of y is unknown")
    )
    yu = f(a,b,xu)
    print("the value of y is",yu)
    k = " y = bx + a"
    plt.plot(x,y,label='actual data plot')
    plt.scatter(x,y)
    plt.plot(h,g)
    plt.scatter(h,g, label='curve-fit')
    plt.yscale('linear')
    plt.xlabel('x-axis')
    plt.ylabel('y-axis')
    plt.title('Least-square curve fitting of data set')
    plt.text( 5,10,k,{'color': 'r','fontsize': 10})
    plt.legend()
    plt.show()
```

4.1.4 Method Output

```
= RESTART: C:\Users\hesienberg\Desktop\Python3 programing\Curve fitting3.py =
Enter the number of terms :9
Enter elements6 5
Enter elements7 5
Enter elements7 4
Enter elements8 5
Enter elements8 4
Enter elements8 3
Enter elements9 4
Enter elements9 3
Enter elements10 3
the values of a and b for the following equation y=a+bx are : 8.0 -0.5
Enter the value of x for which the value of y is unknown 5
```

```
the value of y is 5.5
```



4.2 Regression curve fitting of Logarithmic Curve.

Q.2 An experiment gave the following values :

y :	9	22	88	200	300	500	700	1000	1400
x :	1	2	3	4	5	6	7	8	9

Solve. We have $log_{10}v = log_{10}a + b * log_{10}t$

or $Y = A + bX$, where $Y = log_{10}v$, $X = log_{10}t, A = log_{10}a$

So the normal equations are

$\sum Y = 9A + b\sum X$ ....(i)

$\sum XY = A\sum X + b\sum X^2$ ...(iii)

Now $\sum X$ etc. are calculated as in the following table:

x	y	$X = log_{10}x$	$Y = log_{10}y$	XY	X^2
1	9	0	0.9542	0	0
2	22	0.3010	1.3424	0.4040	0.0906
3	88	0.4771	1.9444	0.9276	0.2276
4	200	0.6020	2.3010	1.3846	0.3624
5	300	0.6989	2.4771	1.7312	0.4884
6	500	0.7781	2.6989	2.1000	0.6054
7	700	0.8450	2.8450	2.4040	0.7140
8	1000	0.9030	3	2.709	0.8154
9	1400	0.9542	3.1461	3.002	0.9104
Total = 45	4219	5.559	20.69	14.6	4.1858

Equation (i) and (ii) become

$9A + 45b = 4219$;

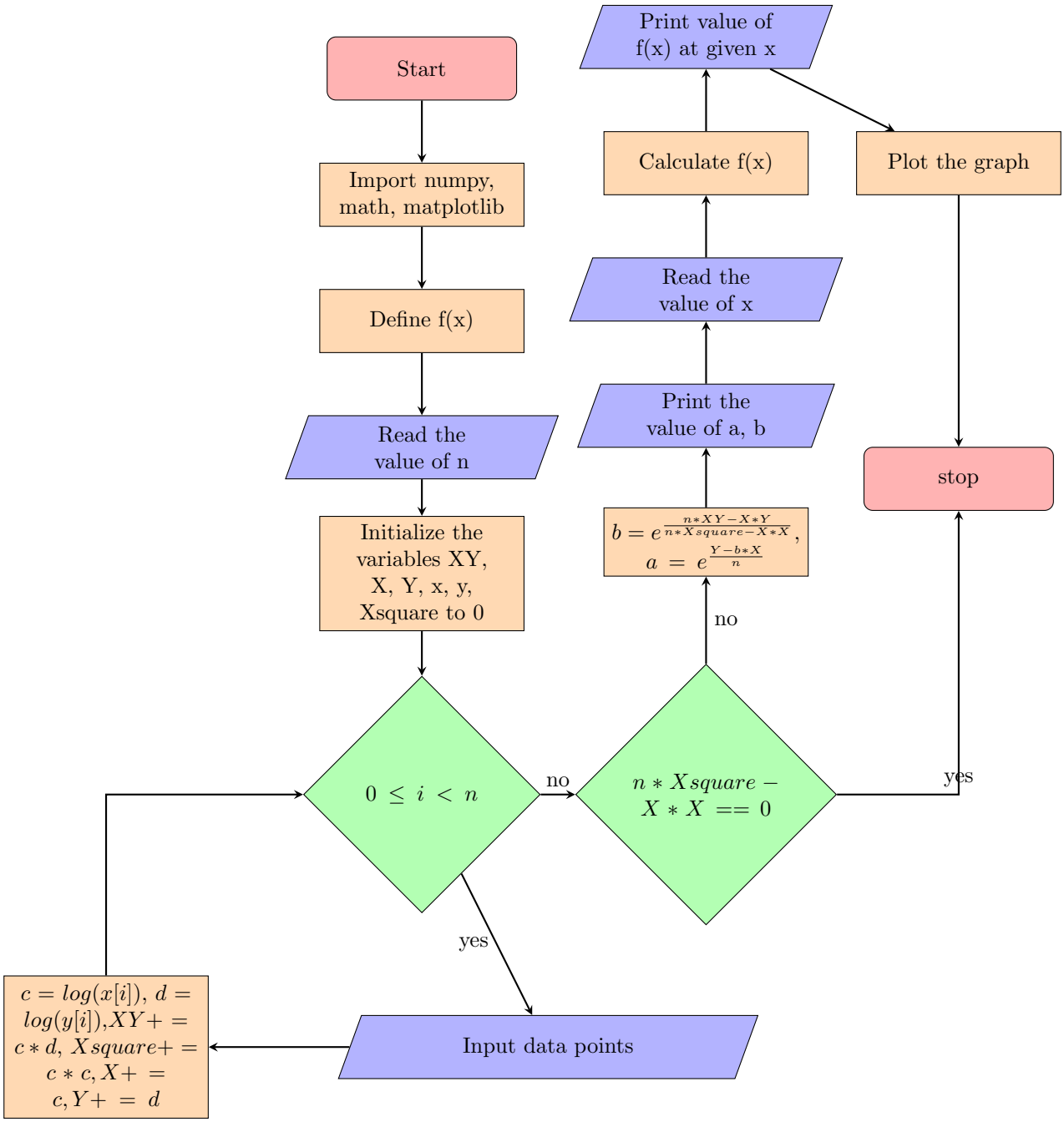
$45A + 4.1858b = 14.6$.

Solving these, A = -8.55, b = 95.46;

4.2.1 Algorithm

Step 1: Start
Step 2: Import numpy, math, matplotlib
Step 3: Define f(x)
Step 4: Read the value of n
Step 5: Initialise all the variables :- XY, Xsquare, X, Y, x, y, to 0
Step 6: Read the data points, calculate c = Log(x[i]), d = Log(y[i]) and increment the appropriate the variables.
Step 7: Check (n*Xsquare - X*X ==0)
(a) Yes : The curve cannot be fitted and go to step 11
(b) No : Calculate the values of a and b using the appropriate formula
Step 8: Print the solution i.e. the values of a and b
Step 9: Evaluate the value of y at given x
Step 10: Plot the graph
Step 11: Stop

4.2.2 Flow Chart-



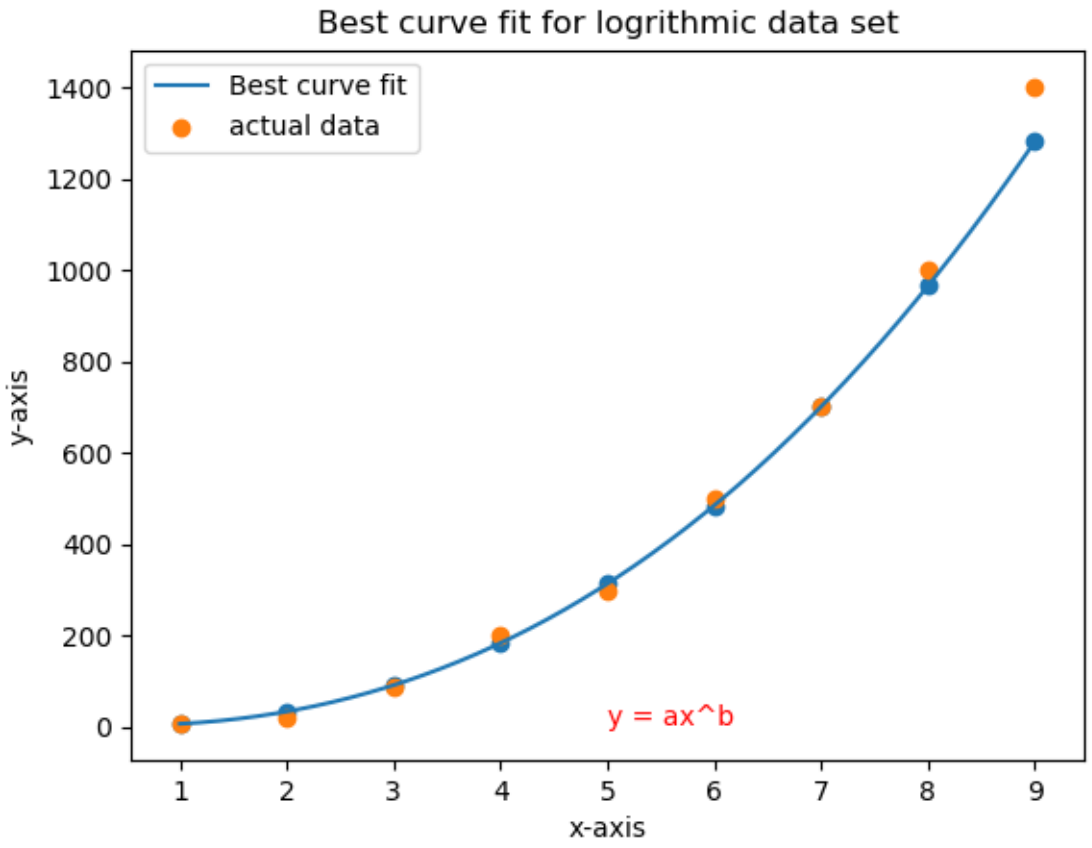
4.2.3 Python Program-

```
import numpy as np
import math
import matplotlib.pyplot as plt
import scipy
from scipy.interpolate import UnivariateSpline
def f(a,b,x):
    return(a*x**b)
n=int(input("Enter the number of terms :"))
XY=0
Xsquare=0
Y=0
X=0
x=np.zeros(n)
y=np.zeros(n)
for i in range(0,n):
    x[i],y[i]=map(float,input("Enter elements as in pairs with space ").split())
    c= float(math.log(x[i]))
    d = float(math.log(y[i]))
    XY= c*d+XY
    Xsquare= Xsquare + c*c
    X= X + c
    Y= Y + d
if (n*Xsquare - X*X ==0):
    print(" The value of given data cannot be fitted")
    exit()
else:
    b= float((n*XY - X*Y)/(n*Xsquare - X*X))
    a= float(math.exp((Y/n) - ((b*X)/n)))
print("the values of a and b for the following equation y=axˆb are:")
print("a = ",a)
print("b = ",b)
xu=float(input("Enter the value of x for which the value of y is unknown"))
yu=f(a,b,xu)
print("The value of y is",yu)
g = a*x**b
h = (g/a)**(1/b)
s=UnivariateSpline(h,g)
x1 = np.linspace(np.min(x),np.max(x),100000)
y1=s(x1)
k ='y = axˆb'
plt.scatter(h,g)
plt.plot(x1,y1,label='Best curve fit')
plt.scatter(x,y, label='actual data')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('Best curve fit for logrithmic data set')
plt.text( 5,5,k,{'color': 'r','fontsize': 10})
plt.legend()
plt.show()
```

4.2.4 Method Output

```
 RESTART: C:\Users\hesienberg\Desktop\Python3 programing\logarithmic-final.py
Enter the number of terms :9
Enter elements as in pairs with space 1 9
Enter elements as in pairs with space 2 22
Enter elements as in pairs with space 3 88
Enter elements as in pairs with space 4 200
```

```
Enter elements as in pairs with space 5 300
Enter elements as in pairs with space 6 500
Enter elements as in pairs with space 7 700
Enter elements as in pairs with space 8 1000
Enter elements as in pairs with space 9 1400
the values of a and b for the following equation y=ax^b are:
a =  6.608557135080761
b =  2.3973207620796106
Enter the value of x for which the value of y is unknown 10
The value of y is 1649.7852299629722
```



4.3 Regression curve fitting of exponential curve.

Q3.822 Predict the main radiation does at the altitude of 3000 feet by fitting an exponential curve t the given data:

Altitude(x):	50	450	780	1200	4400	4800	5300
Dose of radiation:	28	30	32	36	51	58	69

Solution. Let $y = a * b^e$ be the exponential curve,
Then $log_{10}y = log_{10}a + x * log_{10}b$
or $Y = A + Bx$ where $Y = log_{10}y, A = log_{10}a, B = log_{10}b$
So the normal equations are
$\sum Y = 7A + B\sum x$ .....(i)
$\sum xY = A\sum x + \sum x^2$ .....(ii)
Now $\sum x$ etc. are calculated as follows:

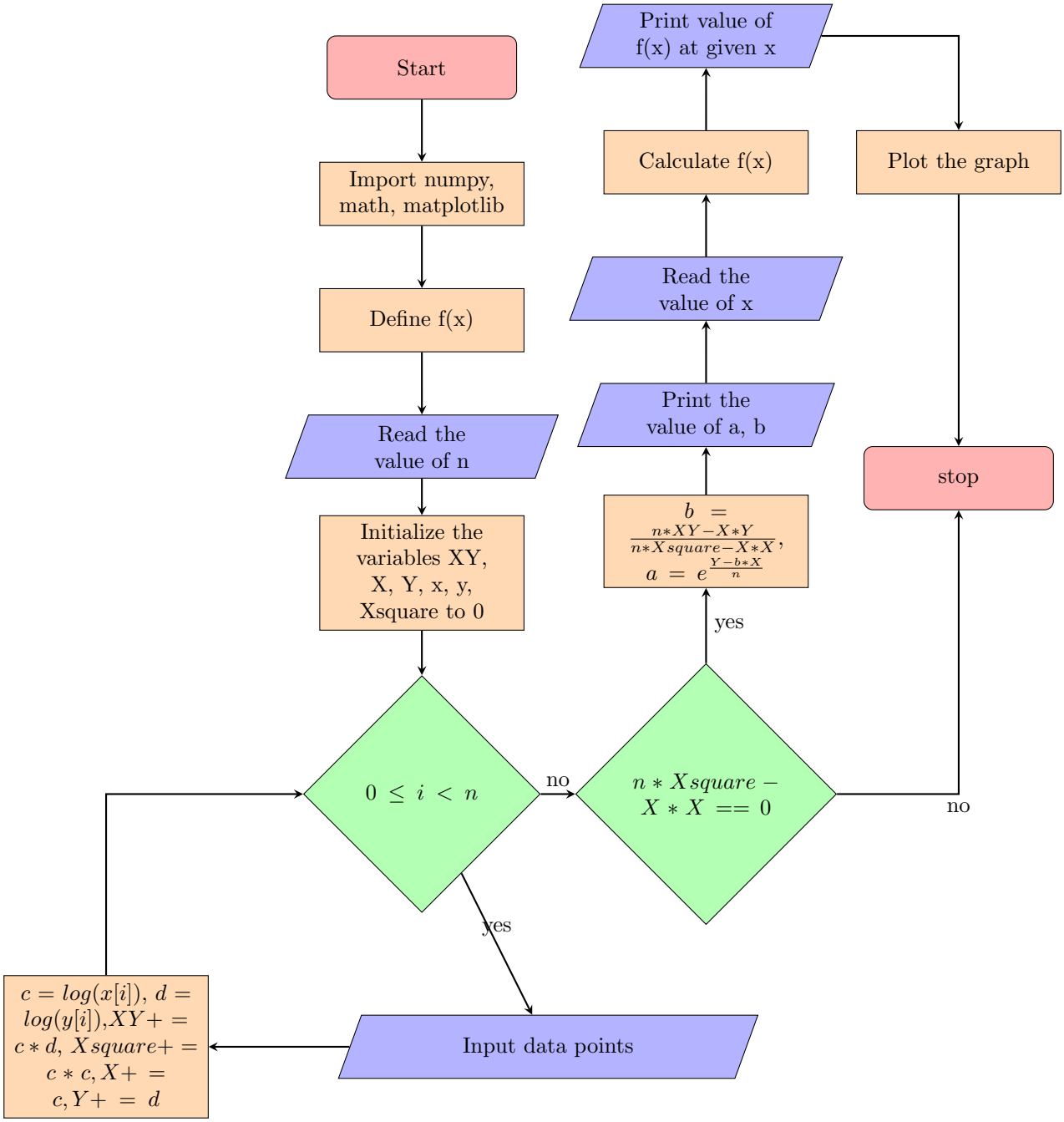
x	y	$Y = log_{10}y$	xY	x^2
50	28	1.447158	72.3579	2500
450	30	1.477121	664.7044	202500
780	32	1.505150	1174.0170	608400
1200	36	1.556303	1867.6536	1440000
4400	51	1.707570	7513.3080	19360000
4800	58	1.763428	8464.5444	23040000
5300	69	1.838849	9745.8997	28090000
Total		11.295579	29502.305	72743400

So equations (i) and (ii) become
$11.295579 = 7A + 16980B$
$29502.305 = 16980A + 72743400$
Solving these equations we get $A = 1.4521015$, $B = 0.0000666289$
So $log_{10}y = Y = 1.4521015 + .0000666289x$
Hence y(at x= 3000)=44.874 i.e. 44.9 approx.

4.3.1 Algorithm

Step 1: Start
Step 2: Import numpy, matplotlib
Step 3: Define f(x)
Step 4: Read the value of n
Step 5: Initialise all the variables :- XY, Xsquare, X, Y, x, y, to 0
Step 6: Read the data points, calculate d = Log(y[i]) and increment the appropriate variables
Step 7: Check (n*Xsquare - X*X ==0)
(a) Yes : The curve cannot be fitted and go to step 11
(b) No : calculate the values of a and b using the appropriate formula
Step 8: Print the solution i.e. the values of a and b
Step 9: Evaluate the value of y at given x
Step 10: Plot the exponential curve
Step 11: Stop

4.3.2 Flow Chart-



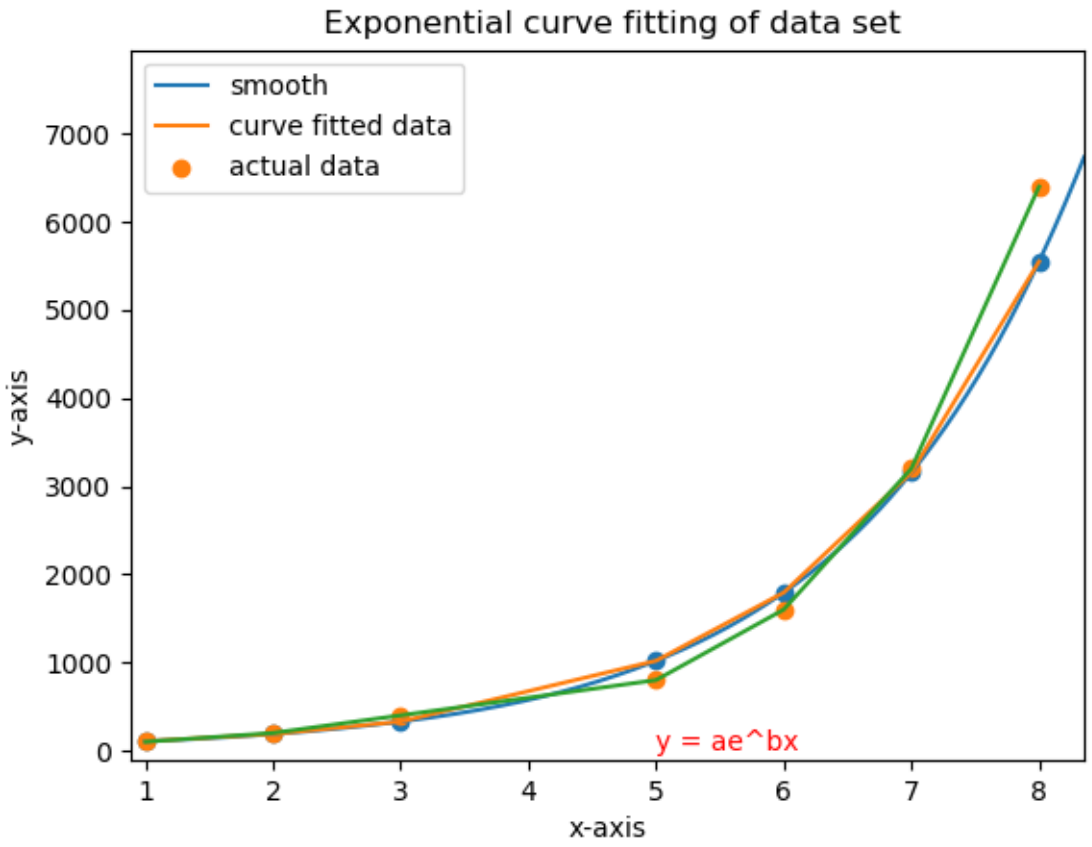
4.3.3 Python Program-

```
import numpy as np
import math
import matplotlib.pyplot as plt
def f(a,b,x):
    return(a*math.exp(b*x))
n=int(input("Enter the number of terms :"))
XY=0
Xsquare=0
Y=0
X=0
x=np.zeros(n)
y=np.zeros(n)
for i in range(0,n):
    x[i],y[i]=map(float,input("Enter elements as in pairs with space ").split())
    d = float(math.log(y[i]))
    XY= x[i]*d+XY
    Xsquare= Xsquare + x[i]*x[i]
    X= X + x[i]
    Y= Y + d
if (n*Xsquare - X*X ==0):
    print(" The value of given data cannot be fitted ")
    exit()
else:
    b= (n*XY - X*Y)/(n*Xsquare - X*X)
    a= float(math.exp((Y/n) - ((b*X)/n)))
print("the values of a and b for the following equation y=aeˆ(bx) are:")
print("a = ",a)
print("b = ",b)
xu=float(input("Enter the value of x for which the value of y is unknown"))
yu=f(a,b,xu)
print("The value of y is",yu)
g = a*np.exp(b*x)
h = np.log(g/a)/b
k ='y = aeˆbx'
plt.plot(h,g, label='curve fitted data')
plt.scatter(h,g)
plt.plot(x,y)
plt.scatter(x,y, label='actual data')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('Least-square exponential curve fitting of data set')
plt.text( 5,5,k,{'color': 'r','fontsize': 10})
plt.legend()
plt.show()
```

4.3.4 Method Output

```
 RESTART: C:/Users/hesienberg/Desktop/Python3 programming/exponential-final.py
Enter the number of terms :7
Enter elements as in pairs with space 50 28
Enter elements as in pairs with space 450 30
Enter elements as in pairs with space 780 32
Enter elements as in pairs with space 1200 36
Enter elements as in pairs with space 4400 51
Enter elements as in pairs with space 4800 58
Enter elements as in pairs with space 5300 69
the values of a and b for the following equation y=aeˆ(bx) are:
a =  28.31597858145113
```

```
b =  0.00015341861392581334
Enter the value of x for which the value of y is unknown 3000
The value of y is 44.86608218051969
```



4.4 Regression curve fitting of polynomial curve.

Q3. Fit a second order polynomial to the given data in the table below:

x:	1.0	2.0	3.0	4.0
y :	6	11	18	27

Solution. The order of polynomial is 2 and therefore we will have 3 simultaneous equations as shown below:

Then
$a_1 n + a_2 \Sigma x_i + a_3 \Sigma x_i^2 = \Sigma y_i$
$a_1 \Sigma x_i + a_2 \Sigma x_i^2 + a_3 \Sigma x_i^3 = \Sigma y_i x_i$
$a_1 \Sigma x_i^2 + a_2 \Sigma x_i^3 + a_3 \Sigma x_i^4 = \Sigma y_i x_i^2$

The sums of power and products can be evaluated in tabular form as shown below:

x	y	x^2	x^3	x^4	yx	yx^2
1	6	1	1	1	6	6
2	11	4	8	16	22	44
3	18	9	27	81	54	162
4	27	16	64	256	108	432
10	62	30	100	354	190	644

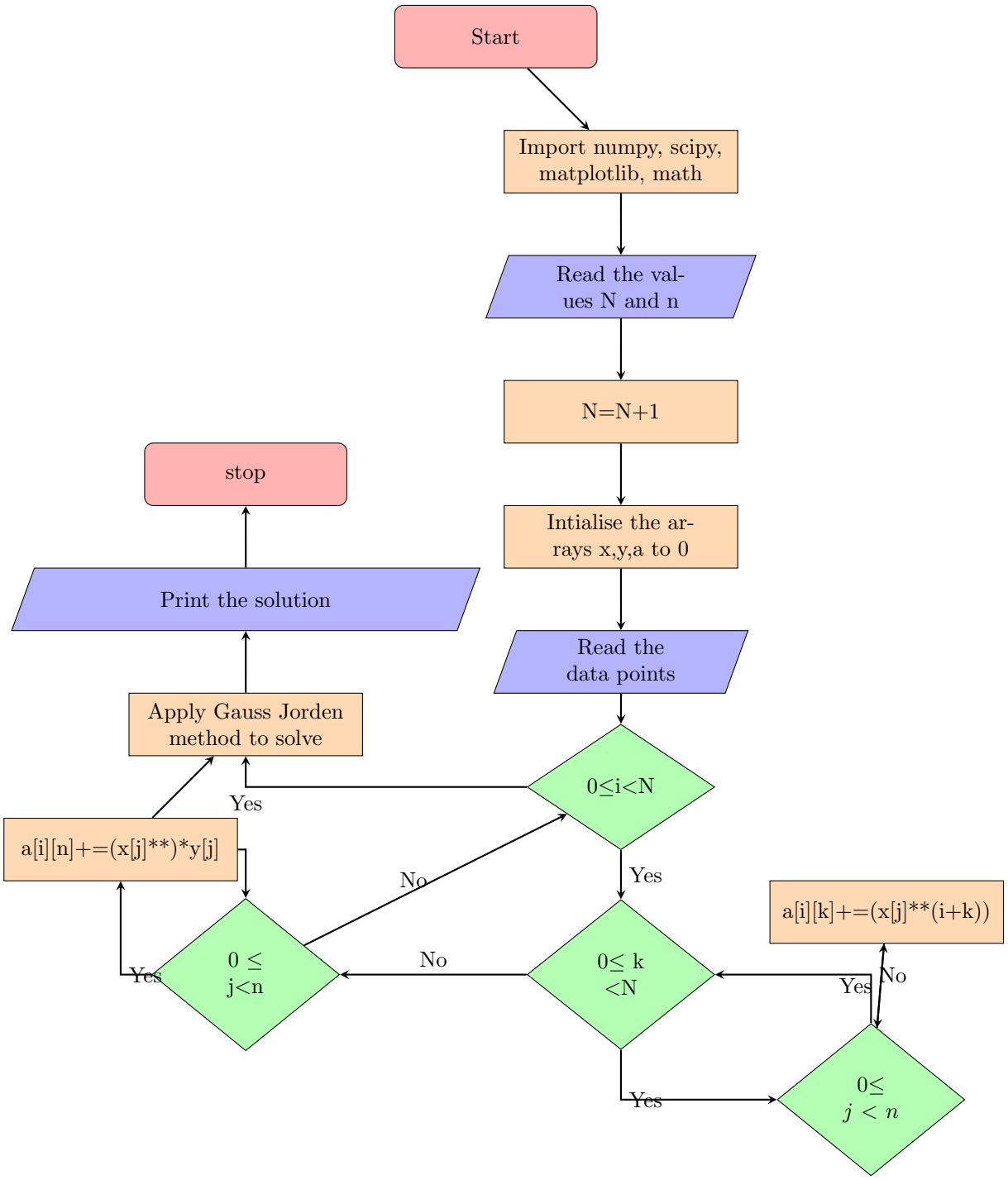
Substituting these value we get
$4a_1 + 10a_2 + 30a_3 = 62$
$10a_1 + 30a_2 + 100a_3 = 190$
$30a_1 + 100a_2 + 354a_3 = 644$

Solving these equations gives
$a_1 = 3$
$a_2 = 2$
$a_3 = 1$
Therefore, the least squares quadratic polynomial is
$y = 3 + 2x + x^2$

4.4.1 Algorithm

Step 1: Start
Step 2: Import numpy, matplotlib, scipy.
Step 3: Read the values of n,N.
Step 4: Increment the values of N by 1.
Step 5: Initialise arrays x,y,a to 0.
Step 6: Read the data points.
Step 7: Check (0≤ i < N)
(a) Yes : Check 0≤ k<N
i. Yes :- check 0≤ j<n
1. Yes:- a[i][k]+= x[j]**(i+k)
2. No:- Goto step (a) (a) No :- check 0≤¡ n
1. Yes:- a[i][N]+=(x[j]**i)*y[j]
2. No:- Goto step -7.
(b) No:- Go-to step - 8
Step 8: Apply the Gauss Jordan Method to find the value of coefficients of variable.
Step 9: Prints the solutions of equation.
Step 10: Evaluate the value of y at given x
Step 11: Plot the exponential curve
Step 12: Stop

4.4.2 Flow Chart-



4.4.3 Python Program-

```
import numpy as np
import matplotlib.pyplot as plt
import scipy
from scipy.interpolate import UnivariateSpline
N=int(input("Enter the degree of the polynomial"))
N+=1
n=int(input("Enter the number of terms :"))
x=np.zeros(n)
y=np.zeros(n)
a=np.zeros((N,N+1))
a1=np.zeros(N+1)
t=0
z=0
for i in range(0,n):
    x[i],y[i]=map(float,input("Enter elements").split())
for i in range(0,N):
    for k in range(0,N):
        for j in range(0,n):
            a[i][k]=a[i][k]+(x[j])**(i+k)
    for j in range(0,n):
            a[i][N]+=(x[j]**i)*y[j]
for j in range(0,N):
        for i in range(0,N):
            if(i!=j):
                t=a[i][j]/a[j][j]
                for k in range(0,N+1):
                    a[i][k]-=a[j][k]*t
print("The solution is :")
for i in range(0,N):
    a1[i]=a[i][N]/a[i][i]
    print(a1[i])
print(a1)
```

4.4.4 Method Output

```
 RESTART: C:\Users\hesienberg\Desktop\Python3 programing\general polynomial.py
Enter the degree of the polynomial 2
Enter the number of terms : 4
Enter elements 1 6
Enter elements 2 11
Enter elements 3 18
Enter elements 4 27
The solution is :
a[0] 3.0
a[1] 2.0
a[2] 1.0
```

5 Numerical Integration

5.1 Trapazoidal Rule

In mathematics, and more specifically in numerical analysis, the trapezoidal rule (also known as the trapezoid rule or trapezium rule) is a technique for approximating the definite integral.

$$\int_a^b f(x)\,dx.$$

The trapezoidal rule works by approximating the region under the graph of the function $f(x)$ as a trapezoid and calculating its area. It follows that

$$\int_a^b f(x)\,dx \approx (b-a)\cdot\tfrac{f(a)+f(b)}{2}.$$

The trapezoidal rule may be viewed as the result obtained by averaging the left and right Riemann sums, and is sometimes defined this way. The integral can be even better approximated by partitioning the integration interval, applying the trapezoidal rule to each subinterval, and summing the results. In practice, this "chained" (or "composite") trapezoidal rule is usually what is meant by "integrating with the trapezoidal rule". Let $\{x_k\}$ be a partition of $[a,b]$such that$a = x_0 < x_1 < \cdots < x_{N-1} < x_N = b$ and Δx_kbe the length of the$k-thsubinterval(thatis, \Delta x_k = x_k - x_{k-1}), then$

$$\int_a^b f(x)\,dx \approx \sum_{k=1}^{N}\frac{f(x_{k-1})+f(x_k)}{2}\Delta x_k = \tfrac{\Delta x}{2}\left(f(x_0)+2f(x_1)+2f(x_2)+2f(x_3)+2f(x_4)+\cdots+2f(x_{N-1})+f(x_N)\right).$$

Illustration of "chained trapezoidal rule" used on an irregularly-spaced partition of$[a,b]$.The approximation becomes more accurate as the resolution of the partition increases$(thatis, forlargerN, \Delta x_k decreases)$. When the partition has a regular spacing, as is often the case, the formula can be simplified for calculation efficiency.

5.2 Simpson Rule

In numerical analysis, Simpson's method is a method for numerical integration, the numerical approximation of definite integrals. Specifically, it is the following approximation for $n+1 values x_0 \cdots x_n$ bounding n equally spaced subdivisions (where n is even): (General Form)

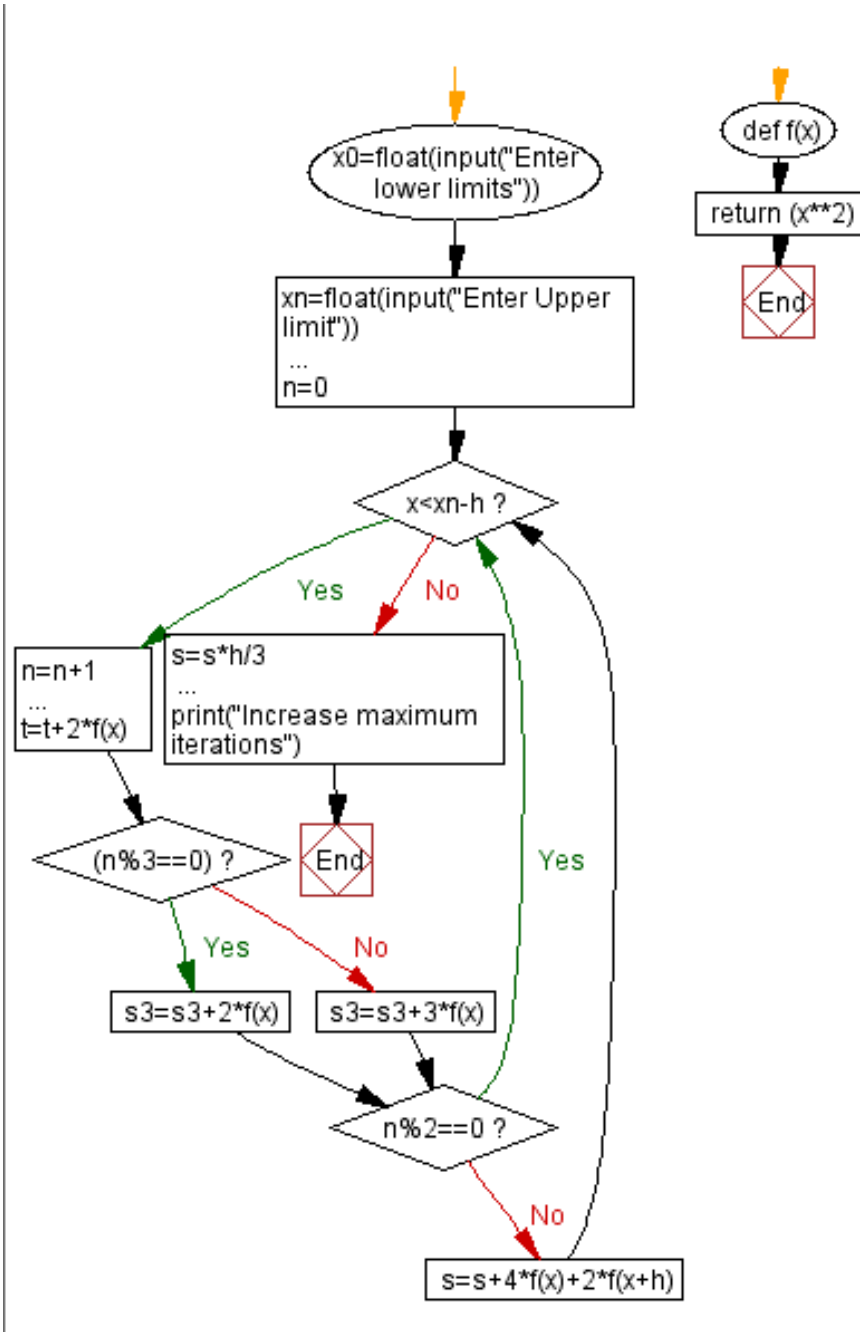
$$\int_a^b f(x)\,dx \approx \tfrac{\Delta x}{3}\left(f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+\cdots+4f(x_{n-1})+f(x_n)\right),$$

$$where\Delta x = \frac{b-a}{n} and x_i = a + i\Delta x.$$

Simpson's rule also corresponds to the three-point Newton-Cotes quadrature rule.

In English, the method is credited to the mathematician Thomas Simpson (1710–1761) of Leicestershire, England. However, Johannes Kepler used similar formulas over 100 years prior, and for this reason the method is sometimes called Kepler's rule, or Keplersche Fassregel (Kepler's barrel rule) in German.

5.2.1 Flowchart



5.2.2 Algorithm

1. define function f(x)
2. input the values of x0,xn,n
3. set h=(xn-x0)/n
   t=f(x0)+f(xn)
   s=t
   s3=t
   x=x0
   n=0
4. applying while loop for the condition x¡xn-h while condition is true set n=n+1
   x=x+h
   t=t+2*f(x)
5. if n is multiple of 3 then
6. set s3=s3+2*f(x)
7. if n is multiple of 3
8. set s3=s3+3*f(x)
9. otherwise continue
10. set s=s+4*f(x)+2*f(x+h)
    s=s*h/3
    t=t*h/2
    s3=s3*3*h/8
11. print t,d,s3 and maximum iteration

5.2.3 Python Program-(Trapezoidal, Simpsons one-third an Simpsons three-eighth

```
def f(x):
    return (x**2)

x0=float(input("Enter lower limits"))
xn=float(input("Enter Upper limit"))
n=int(input("Enter no of sub intervals"))

h=(xn-x0)/n
t=f(x0)+f(xn)
s=t
s3=t
x=x0
n=0
while x<xn-h :
    n=n+1
    x=x+h
    t=t+2*f(x)
    if(n%3==0):
        s3=s3+2*f(x)
    else:
        s3=s3+3*f(x)
    if n%2==0 :
        continue
    s=s+4*f(x)+2*f(x+h)
s=s*h/3
t=t*h/2
```

```
s3=s3*3*h/8
print("Solution of integral is from trapezoidal",t)
print("Solution of integral is from simpsons",s)
print("Solution of integral is from simpsons 3-8th",s3)
print("Increase maximum iterations")
```

5.2.4 Method Output

```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 bit
    (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: C:/Users/TEMP.CBSW7.002/AppData/Local/Programs/Python/Python37/p1.py
Enter lower limits0
Enter Upper limit6
Enter no of sub intervals6
Solution of integral is from trapezoidal 73.0
Solution of integral is from simpsons 96.0
Solution of integral is from simpsons 3-8th 72.0
Increase maximum iterations
>>>
>>>
>>>
 RESTART: C:/Users/TEMP.CBSW7.002/AppData/Local/Programs/Python/Python37/p1.py
Enter lower limits0
Enter Upper limit6
Enter no of sub intervals3000
Solution of integral is from trapezoidal 72.07200399999192
.Solution of integral is from simpsons 72.04799999999194
Solution of integral is from simpsons 3-8th 72.05399999999179
Increase maximum iterations
>>>
```

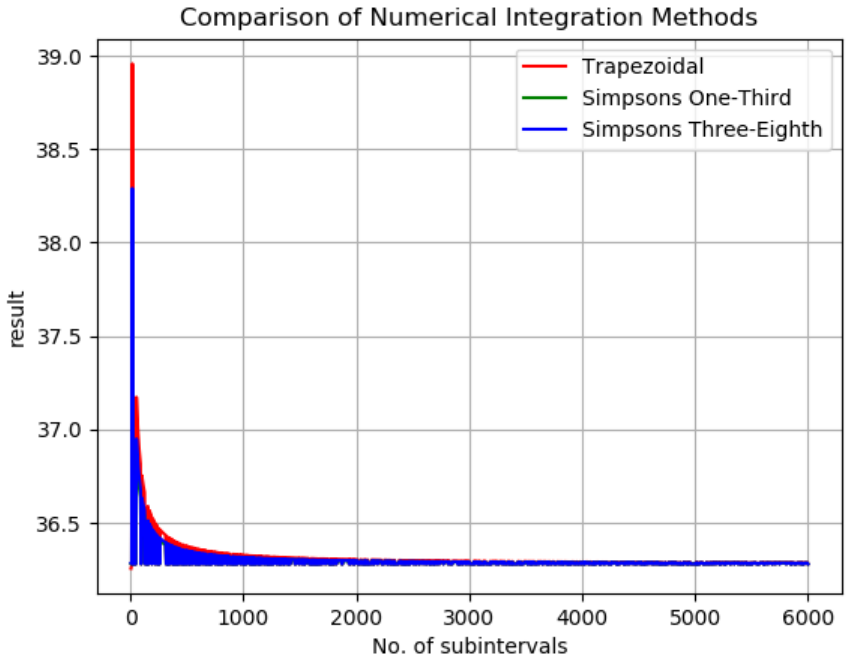
5.2.5 A program for comparison the above methods of integration

```
import matplotlib.pyplot as plt
import numpy as np

def f(x):
    return (x-np.cos(x)+3)
t=np.zeros(1000)
s=np.zeros(1000)
s3=np.zeros(1000)
ni=np.arange(6,6006,6)

x0=float(input("Enter lower limits"))
xn=float(input("Enter Upper limit"))

i=0
n=6
while n<=6000:
    h=(xn-x0)/n
    t[i]=f(x0)+f(xn)
    s[i]=t[i]
    s3[i]=t[i]
    x=x0
    n1=0
    while x<xn-h :
        n1=n1+1
        x=x+h
```

```
            t[i]=t[i]+2*f(x)
            if(n1%3==0):
                s3[i]=s3[i]+2*f(x)
            else:
                s3[i]=s3[i]+3*f(x)
            if n1%2==0 :
                s[i]=s[i]+2*f(x)
            else :
                s[i]=s[i]+4*f(x)
        s[i]=s[i]*h/3
        t[i]=t[i]*h/2
        s3[i]=s3[i]*3*h/8
        n=n+6
        i=i+1
plt.title("Comparison of Numerical Integration Methods")
plt.plot(ni,t,c='r',label='Trapezoidal')
plt.plot(ni,s,c='g',label='Simpsons One-Third')
plt.plot(ni,s3,c='b',label='Simpsons Three-Eighth')
plt.xlabel('No. of subintervals')
plt.ylabel('result')
plt.legend()
plt.xticks()
plt.yticks()
plt.grid()
plt.show()
```

5.2.6 Method Output

```
========== RESTART: C:/Users/Param/Documents/cs/Romberg.py ==========
Enter lower limits0
Enter Upper limit6
```



5.3 Romberg's Integration

Richardson extrapolation is not only used to compute more accurate approximations of derivatives, but is also used as the foundation of a numerical integration scheme called *Romberg integration*. In this scheme, the integral

$$I(f) = \int_a^b f(x)\,dx$$

is approximated using the Composite Trapezoidal Rule with step sizes $h_k = (b-a)2^{-k}$, where k is a nonnegative integer. Then, for each k, Richardson extrapolation is used $k-1$ times to previously computed approximations in order to improve the order of accuracy as much as possible.

More precisely, suppose that we compute approximations $T_{1,1}$ and $T_{2,1}$ to the integral, using the Composite Trapezoidal Rule with one and two subintervals, respectively. That is,

$$\begin{aligned} T_{1,1} &= \frac{b-a}{2}[f(a)+f(b)] \\ T_{2,1} &= \frac{b-a}{4}\left[f(a)+2f\left(\frac{a+b}{2}\right)+f(b)\right]. \end{aligned}$$

Suppose that f has continuous derivatives of all orders on $[a,b]$. Then, the Composite Trapezoidal Rule, for a general number of subintervals n, satisfies

$$\int_a^b f(x)\,dx = \frac{h}{2}\left[f(a)+2\sum_{j=1}^{n-1} f(x_j)+f(b)\right] + \sum_{i=1}^{\infty} K_i h^{2i},$$

where $h=(b-a)/n$, $x_j = a+jh$, and the constants $\{K_i\}_{i=1}^{\infty}$ depend only on the derivatives of f. It follows that we can use Richardson Extrapolation to compute an approximation with a higher order of accuracy. If we denote the exact value of the integral by $I(f)$ then we have

$$\begin{aligned} T_{1,1} &= I(f) + K_1 h^2 + O(h^4) \\ T_{2,1} &= I(f) + K_1 (h/2)^2 + O(h^4) \end{aligned}$$

Neglecting the $O(h^4)$ terms, we have a system of equations that we can solve for K_1 and $I(f)$. The value of $I(f)$, which we denote by $T_{2,2}$, is an improved approximation given by

$$T_{2,2} = T_{2,1} + \frac{T_{2,1}-T_{1,1}}{3}.$$

It follows from the representation of the error in the Composite Trapezoidal Rule that $I(f) = T_{2,2} + O(h^4)$.

Suppose that we compute another approximation $T_{3,1}$ using the Composite Trapezoidal Rule with 4 subintervals. Then, as before, we can use Richardson Extrapolation with $T_{2,1}$ and $T_{3,1}$ to obtain a new approximation $T_{3,2}$ that is fourth-order accurate. Now, however, we have two approximations, $T_{2,2}$ and $T_{3,2}$, that satisfy

$$\begin{aligned} T_{2,2} &= I(f) + \tilde{K}_2 h^4 + O(h^6) \\ T_{3,2} &= I(f) + \tilde{K}_2 (h/2)^4 + O(h^6) \end{aligned}$$

for some constant $\tilde{K}_2$. It follows that we can apply Richardson Extrapolation to these approximations to obtain a new approximation $T_{3,3}$ that is *sixth-order* accurate. We can continue this process to obtain as high an order of accuracy as we wish. We now describe the entire algorithm.

Algorithm (Romberg Integration) Given a positive integer J, an interval $[a, b]$ and a function $f(x)$, the following algorithm computes an approximation to $I(f) = \int_a^b f(x)\,dx$ that is accurate to order $2J$.

$h = b - a$
for $j = 1, 2, \ldots, J$ **do**
  $T_{j,1} = \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{2^{j-1}-1} f(a + jh) + f(b)\right]$ (Composite Trapezoidal Rule)
  for $k = 2, 3, \ldots, j$ **do**
    $T_{j,k} = T_{j,k-1} + \frac{T_{j,k-1} - T_{j-1,k-1}}{4^{k-1}-1}$ (Richardson Extrapolation)
  end
  $h = h/2$
end

It should be noted that in a practical implementation, $T_{j,1}$ can be computed more efficiently by using $T_{j-1,1}$, because $T_{j-1,1}$ already includes more than half of the function values used to compute $T_{j,1}$, and they are weighted correctly relative to one another. It follows that for $j > 1$, if we split the summation in the algorithm into two summations containing odd- and even-numbered terms, respectively, we obtain

$$\begin{aligned} T_{j,1} &= \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{2^{j-2}} f(a + (2j-1)h) + 2\sum_{j=1}^{2^{j-2}-1} f(a + 2jh) + f(b)\right] \\ &= \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{2^{j-2}-1} f(a + 2jh) + f(b)\right] + \frac{h}{2}\left[2\sum_{j=1}^{2^{j-2}} f(a + (2j-1)h)\right] \\ &= \frac{1}{2}T_{j-1,1} + h\sum_{j=1}^{2^{j-2}} f(a + (2j-1)h). \end{aligned}$$

5.3.1 Numerical

Consider

$$\int_1^2 \frac{1}{x}dx = \ln 2.$$

We will use this integral to illustrate how Romberg integration works. First, compute the trapezoid approximations starting with $n = 2$ and doubling n each time:

$n = 1:\ R_1^0 = \left(1 + \frac{1}{2}\right)\frac{1}{2} = 0.75;$

$n = 2:\ R_2^0 = 0.5\left(\frac{1}{1.5}\right) + \frac{0.5}{2}(1 + \frac{1}{2}) = 0.708333333$

$n = 4:\ R_3^0 = 0.25\left(\frac{1}{1.25} + \frac{1}{1.5} + \frac{1}{1.75}\right) + \frac{0.25}{2}(1 + \frac{1}{2}) = 0.69702380952$

$n = 8:\ R_4^0 = 0.69412185037$

$n = 16: R_5^0 = 0.69314718191.$

Next we use the formula:

$$R_k^i = \frac{4^i R_k^{i-1} - R_{k-1}^{i-1}}{4^i - 1}$$

The easiest way is to keep track of computations is to build a table of the form:

$$\begin{array}{lllll} R_1^0 & & & & \\ R_2^0 & R_2^1 & & & \\ R_3^0 & R_3^1 & R_3^2 & & \\ R_4^0 & R_4^1 & R_4^2 & R_4^3 & \\ R_5^0 & R_5^1 & R_5^2 & R_5^3 & R_5^3 \end{array}$$

Starting with the first column (which we just computed), all other entries can be easily computed. For example starting with $R_1^0,\ R_2^0$ we find

$$R_2^1 = \frac{4R_2^0 - R_1^0}{3} = 0.694444$$

$$R_3^1 = \frac{4R_3^0 - R_2^0}{3} = 0.693253; \quad R_3^2 = \frac{16R_3^1 - R_2^1}{15} = 0.69317460$$

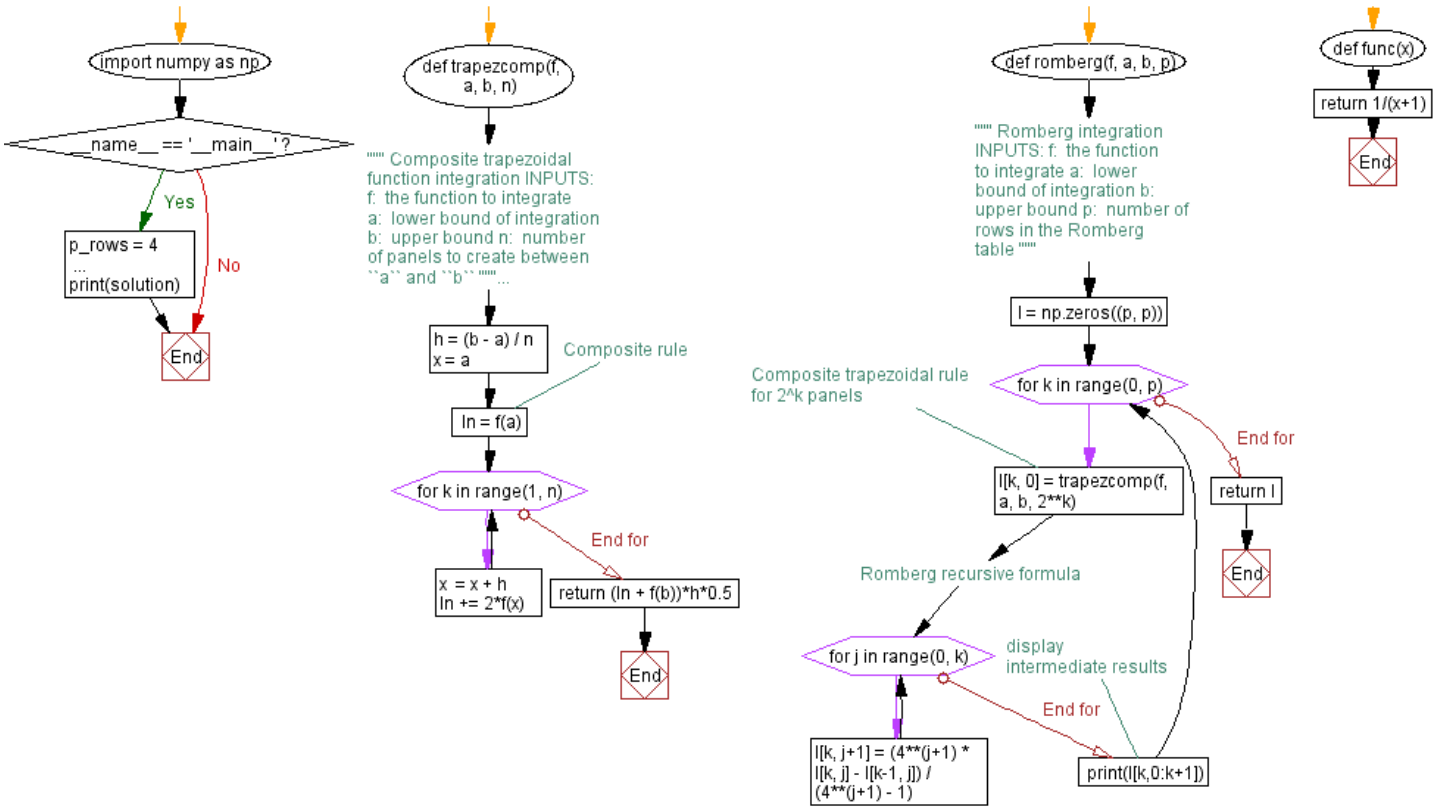
and so on. Every entry depends only on its left and left-top neighbour. Continuing in this way, we get the following table:

0.75000000000				
0.70833333333	**0.69**444444444			
0.69702380952	**0.693**25396825	**0.6931**7460317		
0.69412185037	**0.6931**5453065	**0.69314**90148	**0.69314**47764	
0.69339120220	**0.69314**65281	**0.693147**19429	**0.6931471**8307	**0.69314718**191

The correct digits are shown in bold (the exact answer to 15 digits is given by $\ln 2 = 0.693147180559945$). Here is the table listing error $R_i^k - \ln 2$.

5.7e-02				
1.5e-02	1.3e-03			
3.9e-03	1.1e-04	2.7e-05		
9.7e-04	7.4e-06	7.2e-07	3.0e-07	
2.4e-04	4.7e-07	1.4e-08	2.5e-09	1.4e-09

5.3.2 Flowchart



5.3.3 Algorithm

1. define function f(x)
2. INPUTS:
   f: the function to integrate
   a: lower bound of integration
   b: upper bound
   n: number of panels to create between “a“ and “b‘
3. Initialization 15h = (b - a) / n
   x = a
4. Composite rule 19In = f(a)
5. for k in range(1, n):
   x = x + h
   In += 2*f(x)
6. deine function f romberg(f, a, b, p):
7. romberg integration INPUTS:
   f: the function to integrate
   a: lower bound of integration
   b: upper bound
   p: number of rows in the Romberg table
   I = np.zeros((p, p))
   for k in range(0, p):
8. Composite trapezoidal rule for 2 k panels
   I[k, 0] = trapezcomp(f, a, b, 2**k)
9. Romberg recursive formula
10. for j in range(0, k):
    I[k, j+1] = (4**(j+1) * I[k, j] - I[k-1, j]) / (4**(j+1) - 1)
11. print(I[k,0:k+1]) display intermediate results
12. return I
13. if __name__ == ’__main__’:
14. define function(x):
15. return 1/(x+1)
16. set p_rows = 4
    I = romberg(func, 1, 2, p_rows)
    solution = I[p_rows-1, p_rows-1]
17. print solution

5.3.4 Python Program-using Numpy

```
import numpy as np

def trapezcomp(f, a, b, n):
    """
    Composite trapezoidal function integration

    INPUTS:
    f:  the function to integrate
```

```
    a:  lower bound of integration
    b:  upper bound
    n:  number of panels to create between ``a`` and ``b``
    """

    # Initialization
    h = (b - a) / n
    x = a

    # Composite rule
    In = f(a)
    for k in range(1, n):
        x  = x + h
        In += 2*f(x)

    return (In + f(b))*h*0.5

def romberg(f, a, b, p):
    """
    Romberg integration

    INPUTS:
    f:  the function to integrate
    a:  lower bound of integration
    b:  upper bound
    p:  number of rows in the Romberg table
    """

    I = np.zeros((p, p))
    for k in range(0, p):
        # Composite trapezoidal rule for 2^k panels
        I[k, 0] = trapezcomp(f, a, b, 2**k)

        # Romberg recursive formula
        for j in range(0, k):
            I[k, j+1] = (4**(j+1) * I[k, j] - I[k-1, j]) / (4**(j+1) - 1)

        print(I[k,0:k+1])   # display intermediate results

    return I


if __name__ == '__main__':
    def func(x):
        return 1/(x+1)

    p_rows = 4
    I = romberg(func, 1, 2, p_rows)
    solution = I[p_rows-1, p_rows-1]
    print(solution)
```

5.3.5 Method Output

```
========== RESTART: C:/Users/Param/Documents/cs/Romberg.py ==========
Romberg integration of <function vectorize1.<locals>.vfunc at 0x0000021B60131438
   > from [1, 2]

 Steps  StepSize   Results
     1  1.000000  0.750000
     2  0.500000  0.708333  0.694444
```

```
     4   0.250000   0.697024   0.693254   0.693175
     8   0.125000   0.694122   0.693155   0.693148   0.693147
    16   0.062500   0.693391   0.693148   0.693147   0.693147   0.693147
    32   0.031250   0.693208   0.693147   0.693147   0.693147   0.693147   0.693147

The final result is 0.6931471805622968 after 33 function evaluations.
1.38629 0.842701
```

5.3.6 Python Program-using Scipy

```
from scipy import integrate
from scipy.special import erf
import numpy as np
gaussian = lambda x: 1/np.sqrt(np.pi) * np.exp(-x**2)
result = integrate.romberg(gaussian, 0, 1, show=True)
print("%g %g" % (2*result, erf(1)))
```

5.3.7 Method Output

```
========== RESTART: C:/Users/Param/Documents/cs/Romberg.py ==========
Romberg integration of <function vectorize1.<locals>.vfunc at 0x000002CBDB031438
   > from [0, 1]

 Steps  StepSize    Results
     1  1.000000   0.385872
     2  0.500000   0.412631   0.421551
     4  0.250000   0.419184   0.421368   0.421356
     8  0.125000   0.420810   0.421352   0.421350   0.421350
    16  0.062500   0.421215   0.421350   0.421350   0.421350   0.421350
    32  0.031250   0.421317   0.421350   0.421350   0.421350   0.421350   0.421350

The final result is 0.421350396474754 after 33 function evaluations.
0.842701 0.842701
```

Comparison between methods of integration

Trapazoidal	Simpson	Romberg's
The boundary between the ordinates is considered to be straight	The boundary between the ordinates is considered to be an arc of a parabola	The boundary between the ordinates is considered to be straight
There is no limitation, it can be applied for any number of ordinates	To applying this rule, the number of ordinates must be odd. That is, the number of divisions must be even	There is no limitation, it can be applied for any number of ordinates
It gives an approximate result.	It gives more accurate result.	It gives most accurate result.

Comparison Graph between the accuracy of simpson's ⅓th , ⅜th rule and trapazoidal rule\
Is also discussed above by the program ...

6 Solution to Ordinary Differential Equation

A differential function is a mathematical equation that relates some functions with its derivatives. The solution of differential equation means finding an explicit expression for y in terms of a finite number of elementary function of x. The method of Picard and Taylor series are single step method. The value of y is directly evaluated at a given x, provided the initial conditions. The method of Euler, Heuns, Runge-Kutta, Milne,Polygon etc belong to later class of solutions. In these method , the next point on the curve is evaluated in short steps ahead, by performing iterations till sufficient accuracy is achieved. These methods are called step-by-step methods.

6.1 Taylor's Series

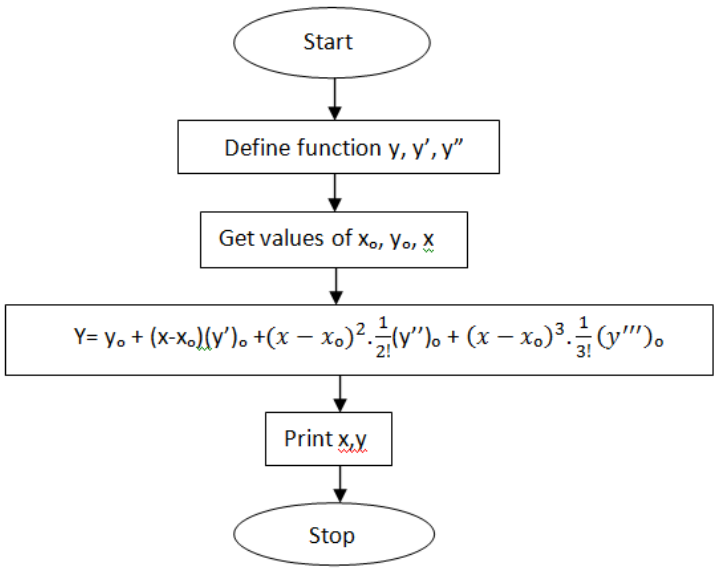
6.1.1 Theory

Consider the first order equation y'=f(x,y)
Differentiating(1) we get
y"=$f_x + f_y f'$
Differentiating this successively we get y"' and y"" etc. Putting x=x_0 and the values of $(y')_0$, $(y")_0$, $(y"')_0$ can be obtained. Hence the Taylor's series
y=y_0+(x-x_0)$(y')_0$+(x-x_0)(x-x_0)$(y")_0$ /2+....
gives the value of y for every value of x for which the equation converges.
Putting the value of y', y" etc. can be evaluated at x =x_1 .The value y_1 can be evaluated for x=x_1.

6.1.2 Numerical

Solve y'=x+y, y(0)=1 by Taylor's method. Hence find the value of y at x=0.1.
Differentiating successively, we get
y'=x+y y'(0)=1
y"=1+y' y"(0)=2
y"'=y" y"'(0)=2
y^{iv} $y(iv)$(0)=2,etc
Taylor's series is
y=y_0+(x-x_0)$(y')_0$+(x-x_0)(x-x_0)$(y")_0$ /2+....
Here x_0=0, y_0=1
Therefore
y=1 + x(1) + $2x^2/2$ + $2x^3/3!$ + $2x^3/4!$ +.....
Thus y(0.1) = 1.1103

6.1.3 Flowchart



6.1.4 Algorithm

Step 1: Start
Step 2: Define function y', y", y"'

Step 3: Get the values of x_0, y_0, x
Step 4: Evaluate y at given x
Step 5: Print the value of y and x
Step 6: Stop

6.1.5 Python Program

```
import math

def Df(x,y):
    return (x+y)
def D2f(x,y):
    return (1+Df(x,y))
def D3f(x,y):
    return (D2f(x,y))

x0=float(input("Enter value of x0"))
y0=float(input("Enter value of y0"))
x=float(input("Enter value of x for which y is required"))
y=y0+(x-x0)*Df(x0,y0)+pow((x-x0),2)*D2f(x0,y0)/2+pow((x-x0),3)*D3f(x0,y0)/6
print("Solution of ODE is", y,"for x=", x)
```

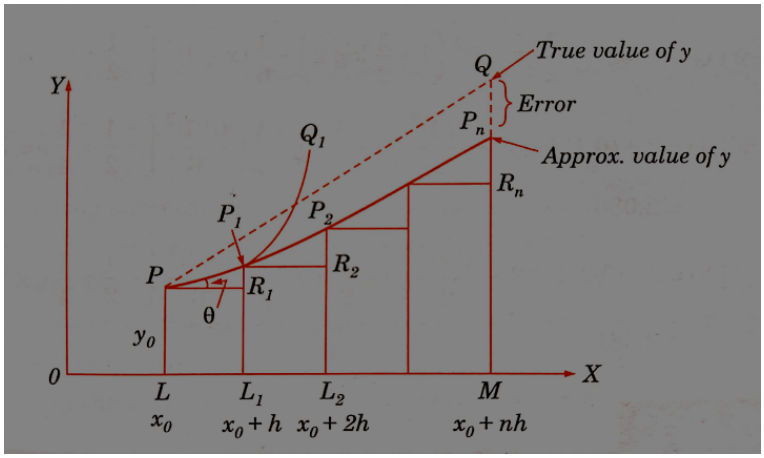
6.1.6 Output

```
Enter value of x00
Enter value of y01
Enter value of x for which y is required0.1
Solution of ODE is 1.1103333333333334 for x= 0.1
```

6.2 Euler's method

6.2.1 Theory

Consider the equation y'=f(x,y)
given that x(x_0)=y_0 .Its curve of solution through P(x_0,y_0) is shown dotted in figure. Now we have to find the ordinate of any other point Q on this given curve.



Let us divide LM into n sub-intervals each of width h at L_1, L_2...so that h is quite small.
In the interval LL_1, we approximate the curve by the tangent P. If the ordinate through L_1 meet this tangent in P_1(x_0+h,y_1), then

y_1=y_0+h f(x_0,y_0)
Let P_1 Q_1 be the curve of the equation through line P_1 and let its tangent at P_1 meet the ordinate through L_2 in P_2(x_0+2h, y_2). Then

y_2=y_1 + h f(x_0+h, y_1)
Repeating this process n times, we finally reach on an approximation MP_n of MQ given by

y_n=y_{n-1}+ hf(x_0+(n-1)h,y_{n-1})
This is Euler's Method of finding an approximate solution.

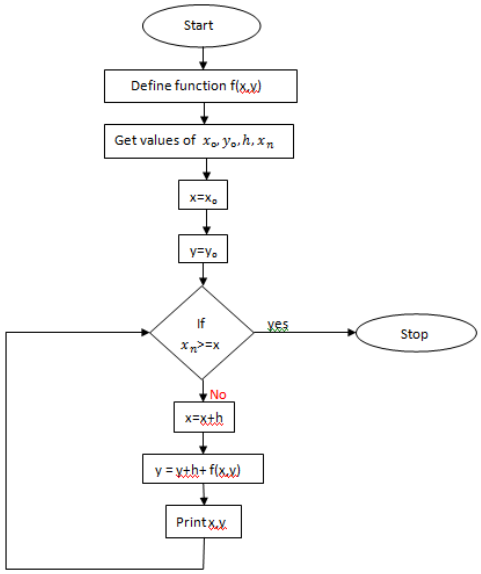
6.2.2 Numerical

Using Euler's method, find an approximate value of y corresponding to x=1, given that y'=x+y and y=1 when x=0.
We take n=10 and h=0.1 which is sufficiently small.
Table

x	y	x + y = dy / dx	Old y + 0.1 (dy / dx) = new y
0.0	1.00	1.00	1.00 + 0.1 (1.00) = 1.10
0.1	1.10	1.20	1.10 + 0.1 (1.20) = 1.22
0.2	1.22	1.42	1.22 + 0.1 (1.42) = 1.36
0.3	1.36	1.66	1.36 + 0.1 (1.66) = 1.53
0.4	1.53	1.93	1.53 + 0.1 (1.93) = 1.72
0.5	1.72	2.22	1.72 + 0.1 (2.22) = 1.94
0.6	1.94	2.54	1.94 + 0.1 (2.54) = 2.19
0.7	2.19	2.89	2.19 + 0.1 (2.89) = 2.48
0.8	2.48	3.29	2.48 + 0.1 (3.29) = 2.81
0.9	2.81	3.71	2.81 + 0.1 (3.71) = 3.18
1.0	3.18		

Thus the required value of y=3.18.

6.2.3 Flowchart



6.2.4 Algorithm

Step 1: Start
Step 2: Define function y'=f(x,y)
Step 3: Get values of x_0, y_0, h, x_n
Step 4: Put x_0 and y_0
Step 5: Check if x_n¿=x
  (a) If yes: Go to Step 8
  (b) If no: Then y=y+h*f(x,y) and x=x+h
Step 6: Print x, y
Step 7: Go to Step 5
Step 8: Stop

6.2.5 Python Program

```
import numpy as np
import math
def f(x,y):
    return (x+y)

def euler(x0,y0,xn,h):
    x=x0
    y=y0
    i=i+1
    print("Euler's method")
    print("%2d %2.6f %2.6f" %(i,x,y))
    while x<=xn :
        x=x+h
        y=y+h*f(x,y)
        i+1
        print("%2d %2.6f %2.6f" %(i,x,y))

x0=float(input("Enter value of x0 "))
y0=float(input("Enter value of y0 "))
x=float(input("Enter value of x "))
n=int(input("Enter number of intervals "))
h=(x-x0)/n
euler(x0,y0,x,h)
```

6.2.6 Output

```
Enter value of x0 0
Enter value of y0 1
Enter value of x 1
Enter number of intervals 10
Euler's method
 1  0.000000 1.000000
 2  0.100000 1.110000
 3  0.200000 1.241000
 4  0.300000 1.395100
 5  0.400000 1.574610
 6  0.500000 1.782071
 7  0.600000 2.020278
 8  0.700000 2.292306
 9  0.800000 2.601537
 10 0.900000 2.951690
 11 1.000000 3.346859
 12 1.100000 3.791545
```

6.3 Heun's Method

In Euler's method, the slope at the beginning of the interval is used to extrapolate y_i to y_{i+1} over the entire interval. Thus,

$y_{i+1}= y_i + m_i$h

where m_1 is the slope at (x_i,y_i).

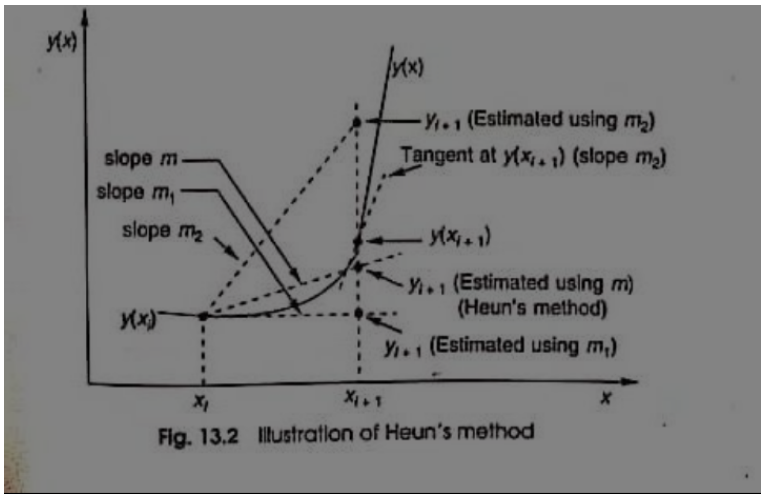


Fig. 13.2 Illustration of Heun's method

An alternative is to use the line which is parallel to the tangent at the point $(x_{i+1}$,y(x_{i+1}) to extrapolate from y_i to y_{i+1}. That is

$y_{i+1}=y_i+m_2$h

where m_2 is the slope at $(x_{i+1}$,y$x_{i+1})$.

A third approach is to use a line whose slope is the average of the slope at the end point of the interval. Then

$y_{i+1}=y_i + (m_1+m_2)$*h/2

This approach is known as Heun's method.

6.3.1 Numerical

Solve the difference equation y'=2y/x using Heun's method. Find an approximate value of y corresponding to x=2, given that y=2 when x=1.

Take h=0.25

Iteration 1

m_1=4

y_e(1.25)=2+0.25(4.0)=3.0

m_2=4.8

y(1.25)=2+ (0.25(4.0+4.8)/2) = 3.1

Iteration 2

m_1=4.96

y_e(1.5) = 3.1+0.25(4.96) = 4.43

m_2=4.8

y(1.5) = 3.1 + (0.25(4.96+5.79)/2) = 4.44

Iteration 3

m_1=5.92

y_e(1.75)=4.44+0.25(5.92)=5.92

m_2=6.77

y(1.75) = 4.44 + (0.25(5.92+6.77)/2) = 6.03

Iteration 4

m_1=6.89

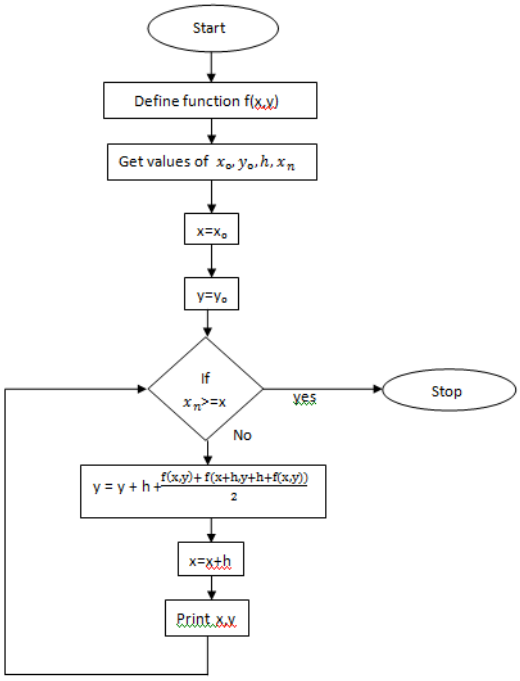
y_e(2.0) = 6.03 + 0.25(6.89) = 7.75

m_2=7.75

y(2.0) = 6.03 + (0.25(6.89+7.75)/2) = 7.86

Hence the value of y is 7.86 at x=2.

6.3.2 Flowchart



6.3.3 Algorithm

Step 1: Start
Step 2: Define function y'=f(x,y)
Step 3: Get values of x_0, y_0,x_n, h
Step 4: Put x=x_0 and y=y_0
Step 5: Check if x_n¿=x
  (a) If yes: Go to Step 8
  (b) If no: y=y+h*(f(f,y)+f(x+h,y+h*f(x,y)))/2 and x=x+h
Step 6: Print x, y
Step 7: Go to Step 5
Step 8: Stop

6.3.4 Python Program

```
import numpy as np
import math
def f(x,y):
    return ((2*y)/x)

def heun(x0,y0,xn,h):
    x=x0
    y=y0
    i=1
    print("heun's method")
    print("%2d %2.6f %2.6f" %(i,x,y))
    while x<=xn :
        y=y+h*(f(x,y)+f(x+h,y+h*f(x,y)))/2
        x=x+h
        i=i+1
        print("%2d %2.6f %2.6f" %(i,x,y))

x0=float(input("Enter value of x0 "))
y0=float(input("Enter value of y0 "))
x=float(input("Enter value of x "))
n=int(input("Enter value of intervals"))
```

```
h=(x-x0)/n
heun(x0,y0,x,h)
```

6.3.5 Output

```
Enter value of x0 1
Enter value of y0 2
Enter value of x 2
Enter value of intervals4
heun's method
 1   1.000000    2.000000
 2   1.250000    3.100000
 3   1.500000    4.443333
 4   1.750000    6.030238
 5   2.000000    7.860846
 6   2.250000    9.935236
```

6.4 Runge kutta method

6.4.1 Theory

The Taylor's series method for solving differential equations numerically is restricted by the labour involved in finding the higher order derivatives. However there is a class of method known as Runge-Kutta methods which do not require the calculation of higher order derivatives and give great accuracy. The Runge-Kutta formulae possess the advantage of requiring only the function values at some selected points. these method agree with Taylor's series solution upto the them in

$$h^r \tag{12}$$

where r differs from method and is called the order of that method.

(1) first order R-K method: Euler's method is the R-K method of first order.

y_1=y_0+h*f(x_0,y_0)=y_0+h*y_0'.

(2)second order R-K method: The modified Euler's method is the R-K method of the second order. The second order Runge Kutta formula is

y_1=y_0+(1/2)*(k_1+k_2)

where

k_1=h*f(x_0,y_0)and k_2=h*f(x_0+h,y_0+k_1).

(3)Third order R-K method: The third order Runge Kutta formula is

y_1=y_0+(1/2)*(k_1+4*k_2+k_3)

where k_1=h*f(x_0,y_0)

k_2=h*f(x_0+(1/2)*h,y_0+(1/2)*k_1)

and k_3=h*f(x_0+h,y_0+k')

where k'=h*f(x_0+h,y_0+k).

(4)Forth order R-K method: This method is most commonly used and is often referred to as R-K method only. ***workingrule*** for finding the increment k of y corresponding to an increment h of x by R-K method from dy/dx=f(x,y),y(x_0)=y_0 is as follows: Calculate successively

k1=h*f(x_0,y_0)

k_2=h*f(x_0+(1/2)*h,y_0+(1/2)*k_1)

k_3=h*f(x_0+(1/2)*h,y_0+(1/2)*k_2) and k_4=h*f(x_0+h,y_0+k_3)

finally compute k=(1/6)(k_1+2*k_2+2*k_3+k_4)

which gives the required approximate value as y_1=y_0+k.

6.4.2 Numerical

Apply R-K fourth order method to find an approximate value of y when x=0.2 given that dy/dx=x+y and y=1 when x=0.

sol.

Here x_0=0,y_0=1,h=0.2,f(x_0,y_0)=1

k_1=h*f(x_0,y_0)=0.2*1=0.2000

k_2=h*f(x_0+(1/2)*h,y_0+(1/2)*k_1)=0.2*f(0.1,1.1)=0.2400

k_3=h*f(x_0+(1/2)*h,y_0+(1/2)*k_2)=0.2*f(0.1,1.12)=0.2440

k_4=h*f(x_0+h,y_0+k_3)=0.2*f(0.2,1.2440)=0.2888

k=1/6(k_1+2*k_2+2*k_3+k_4)=1/6(0.2000+0.4800+0.4880+0.2888)

=(1/6)*1.4568=0.2428

Hence the required approximate value of y is 1.2428.

6.4.3 algorithm

step1:- Start

Step2:-define function

Step3:- get value of x0,y0,x,h

Step4:- call function

Step5:-evaluate the value of n and y=y0

step6:- i=1

step7:- i¡=n+1

a) if no, go to step11

b) if yes, go to step8

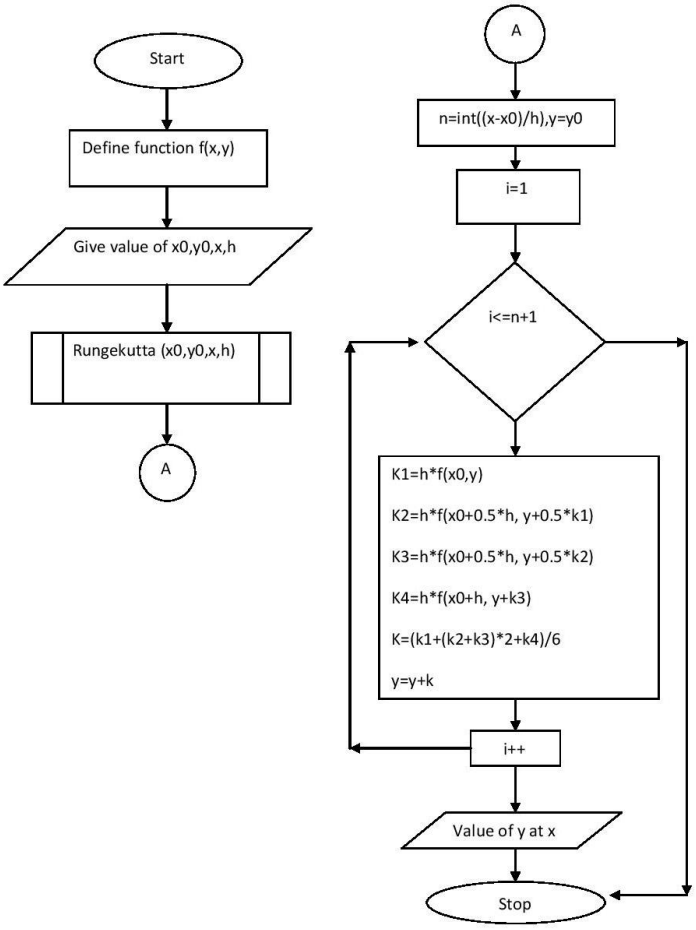
step8:- evaluate k1,k2,k3,k4,k and y

step9:- print value of y at x

step10:- i++,go to step7

step11:- spot

6.4.4 flowchart



6.4.5 Python Program

```
def f(x,y):
    return (x+y)
def rungekutta(x0,y0,x,h):
    n=int((x-x0)/h)
    y=y0
    for i in range(1,n+1):
        k1=h*f(x0,y)
        k2=h*f(x0+0.5*h,y+0.5*k1)
        k3=h*f(x0+0.5*h,y+0.5*k2)
        k4=h*f(x0+h,y+k3)
        k=(k1+(k2+k3)*2+k4)/6
        y=y+k
        x0=x0+h
        print("the value of y at x=:",x0,"is",y)

x0=float(input("enter the value of x0"))
y0=float(input("enter the value of y"))
x=float(input("ente the value of x"))
h=float(input("enter the value of h"))
rungekutta(x0,y0,x,h)
```

6.4.6 Output

```
Python 3.7.4rc1 (tags/v3.7.4rc1:b26441ee1f, Jun 18 2019, 23:26:33) [MSC v.1916
    64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:\python37\rungekutta method.py ==========
enter the value of x00
enter the value of y1
ente the value of x0.2
enter the value of h0.2
the value of y at x=: 0.2 is 1.2428
>>>
```

6.5 Milne's method

6.5.1 Theory

In numerical analysis, predictor–corrector methods belong to a class of algorithms designed to integrate ordinary differential equations – to find an unknown function that satisfies a given differential equation. All such algorithms proceed in two steps:

The initial, "prediction" step, starts from a function fitted to the function-values and derivative-values at a preceding set of points to extrapolate ("anticipate") this function's value at a subsequent, new point. The next, "corrector" step refines the initial approximation by using the predicted value of the function and another method to interpolate that unknown function's value at the same subsequent point. Given dy/dx=f(x,y) and y=y_0,x=x_0;to find an approximate value for x=x_0+nh by milne's method,we proceed as follows: The value y_0=y(x_0) being given,we compute y_1=y(x_0+h),y_2=y(x_0+2h),y_3=y(x_0+3h), by Picard's and taylor's series method. next we calculate the corresponding value of f_0,f_1,f_2,f_3, then to find y_4=y(x_0+4h),we substitute Newton's forward interpolation formula and finaly we get y_4(p)=y_0+(4*h/3)*(2*f_1-f_2+2*f_3) which is called a predictor. Having found y_4,we obtain a first approximation to f_4=f(x_0+4*h,y_4). Then a better value of y4 is found simpson's rule as y_4(c)=y_2+(1*h/3)*(f_2+4*f_3+f_4) Which is called a corrector . Then an improved value of f_4 is computed and again the corrector is applied to find a still better value of f_4.We get this step until y_4 remains unchanged. similarly we can do for y_5 and so on.

6.5.2 Numerical

Apply milne's method, to find a solution of the differential equation y'=x-y^2 in the range 0¡=x¡=1 for the boundary condition y=0 at x=0

sol.

We have y'=x-y^2 =f(x)

by taylor's series method ,we get

y'=x-y^2 y'(0)=0

y"=1-2*y*y' y"(0)=1

y"'= -(2*y*y" + 2*(y')^2) y"'(0)=0

y=y_0+(x-x_0)y'(0)+(((x−x_0)^2)/2)*y"(0)+(((x−x_0)^3)/6)*y"'(0)+.....

hence ,We get

x_0=0, y_0=0,f_0=0

x_1=0.2, y_1=0.02, f_1=0.1996

x_2=0.4, y_2=0.08, f_2= 0.3936

x_3=0.6 , y_3=0.18, f_3=0.5676

since y_4 is required , We use the predictor

y_4(p)=y_0+(4*h/3)*(2*f_1-f_2+f_3)

x=0.8 , y_4(p)=0+(4*(0.2)/3) *(2*0.1996-0.3936+0.5676)= 0.3049

f_4=0.7070 now using the corrector

y_4(c)=y_2+(h/3)*(f_2+4*f_3+f_4)

y_4(c)=0.08+(0.2/3)*(0.3936+4*0.5676+0.7070)=0.3046

f_4(c)=0.7070

Since y_5 is required, We use the predictor

y_5(p)=y_1+(4*h/3)(2*f_2-f_3+2*f_4)

x=1, y_5(p)=0.02+(4*(0.2)/3)*(2*0.3936-0.5676+2*0.7070)=0.4554

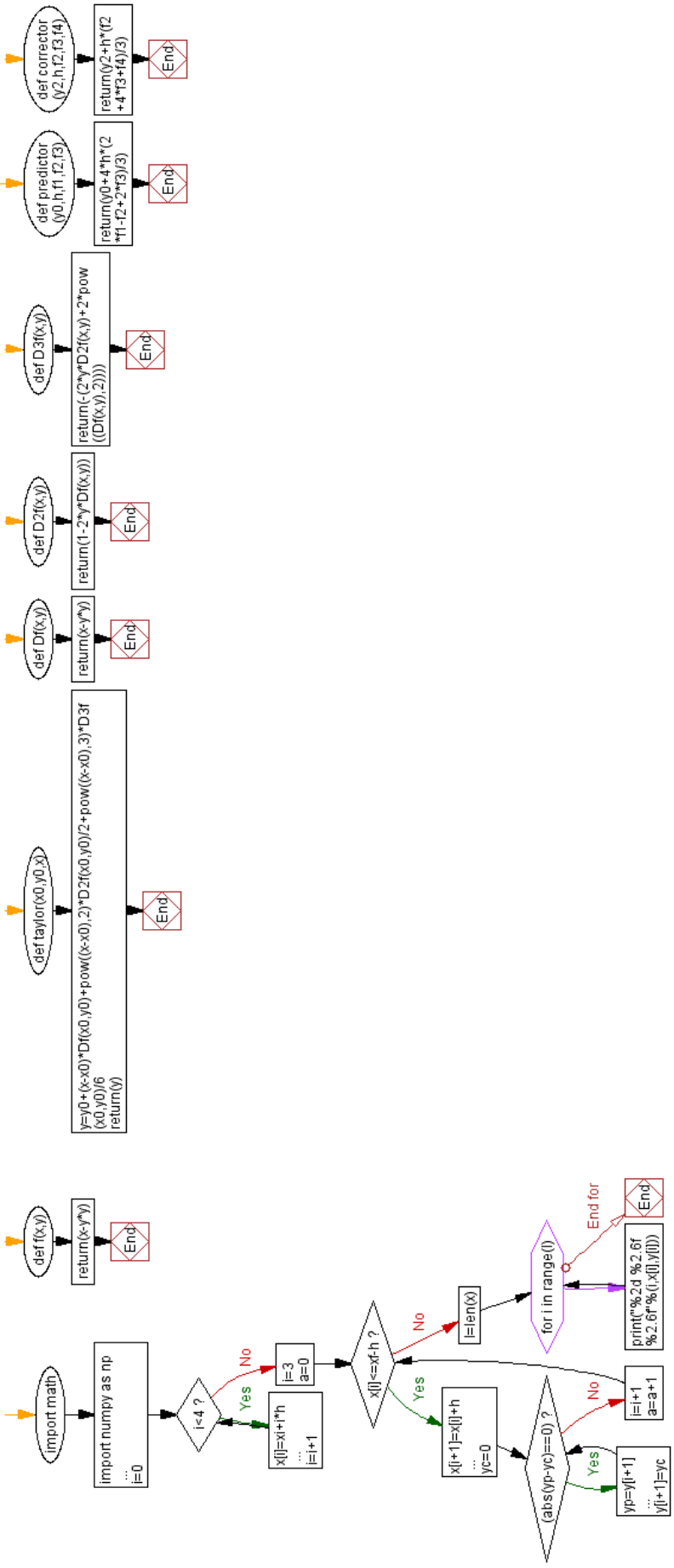
f_5=x-$y5(p)^2$=0.7926

Now using the corrector y_5(c)= y_3+(h/3)*(f_3+4*f_4+f_5),We get

y_5(c)=0.18+(0.2/3)*(0.5676+4*0.7070+0.7926)=0.4555

Hence y(1)=0.7925.

6.5.3 Flowchart



6.5.4 algorithm

6.5.5 Python Program

```
import math
import numpy as np
def f(x,y):
    return(x-y*y)
def taylor(x0,y0,x):
    def Df(x,y):
        return(x-y*y)
    def D2f(x,y):
        return(1-2*y*Df(x,y))
    def D3f(x,y):
        return(-(2*y*D2f(x,y)+2*pow((Df(x,y),2))))
    y=y0+(x-x0)*Df(x0,y0)+pow((x-x0),2)*D2f(x0,y0)/2+pow((x-x0),3)*D3f(x0,y0)/6
    return(y)
def predictor(y0,h,f1,f2,f3):
    return(y0+4*h*(2*f1-f2+2*f3)/3)
def corrector(y2,h,f2,f3,f4):
    return(y2+h*(f2+4*f3+f4)/3)
x0=float(input("enter the value of x0"))
y0=float(input("enter the value of y0"))
xi=float(input("enter initial value of x"))
xf=float(input("enter final value of x"))
h=float(input("enter the value of h"))
x=np.arange(1+int(xf-xi)/h)
y=np.arange(1+int(xf-xi)/h)
i=0
while i<4:
    x[i]=xi+i*h
    y[i]=taylor(x0,y0,x[i])
    i=i+1
i=3
a=0
while x[i]<=xf-h:
    x[i+1]=x[i]+h
    yp=predictor(y[a],h,f(x[i-2],y[i-2]),f(x[i-1],y[i-1]),f(x[i],y[i]))
    yp=round(yp,5)
    y[i+1]=yp
    yc=0
    while(abs(yp-yc)==0):
        yp=y[i+1]
        yc=corrector(y[i-1],h,f(x[i-1],y[i-1]),f(x[i],y[i]),f(x[i+1],yp))
        yc=round(yc,5)
        y[i+1]=yc
    i=i+1
    a=a+1
l=len(x)
for i in range(l):
    print("%2d %2.6f %2.6f"%(i,x[i],y[i]))
```

6.5.6 Output

```
Python 3.7.4rc1 (tags/v3.7.4rc1:b26441ee1f, Jun 18 2019, 23:26:33) [MSC v.1916
    64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:\python37\milne's method.py ==========
enter the value of x00
enter the value of y00
```

```
enter initial value of x0
enter final value of x1
enter the value of h0.2
 0 0.000000 0.000000
 1 0.200000 0.020000
 2 0.400000 0.080000
 3 0.600000 0.180000
 4 0.800000 0.304210
 5 1.000000 0.455870
>>>
```

6.6 Picard's method

6.6.1 Theory

Consider the equation

$$\frac{dy}{dx} = f(x, y) \tag{13}$$

. We can integrate this to obtain the solution in the interval(x0,x)

$$\int_{x_0}^{x} dy = \int_{x_0}^{x} f(x, y)dx \tag{14}$$

or

$$y(x) = y(x_0) + \int_{x_0}^{x} f(x, y)dx$$

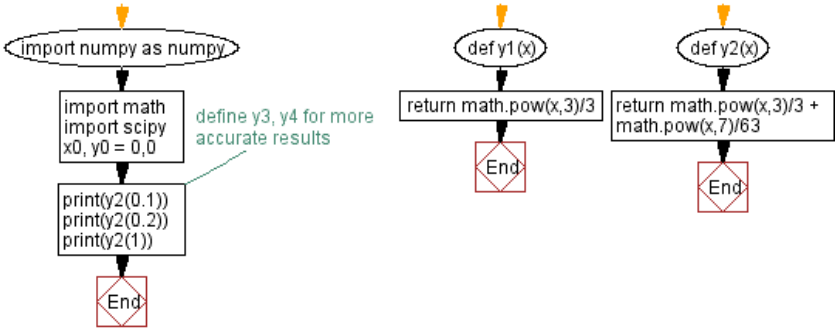
. Since y appears under the integral sign on the right,the integration can not be formed. The dependent variable y should be replaced by a constant or a function of x. Since we know the initial value of y(at x = x0), we may use this as a first approximation to the solution and the result can be used on the right hand side to obtain the next appropriation. The iterative equation is written as

(15)

. This equation is known as the Picard's method. Since this method involves actual integration, sometimes it may not be possible to carry out this integration. It can be seen that the Picard's method is not convenient for computer based solutions. Like Taylor series method, this method is also semi-numeric.

6.6.2 Numerical

6.6.3 Flowchart



6.6.4 algorithm

1. Start
2. Define function for first integration iteration.
3. Define function for second integration iteration.
4. Call function2 at required values
5. Stop

6.6.5 Python Program

```
import numpy as numpy
import math
import scipy
x0, y0 = 0,0

def y1(x):
  return math.pow(x,3)/3
def y2(x):
  return math.pow(x,3)/3 + math.pow(x,7)/63
```

```
#define y3, y4 for more accurate results

print(y2(0.1))
print(y2(0.2))
print(y2(1))
```

6.6.6 Output

```
(base) ayush@feynman:~/Desktop/Python ODE programs/ python3 picards.py
0.00033333492063492075
0.002666869841269842
0.3492063492063492
```

6.7 Polygon method

6.7.1 Theory

Polygon method is a modification of Euler's method in which slope of the mid point of initial and final point is used to approximate y_{i+1}. Thus
$y_{i+1} = y_i + f(\frac{x_i+x_{i+1}}{2}, \frac{y_i+y_{i+1}}{2})h = y_i + f(x_i + h/2, y_i + dy/2)h$

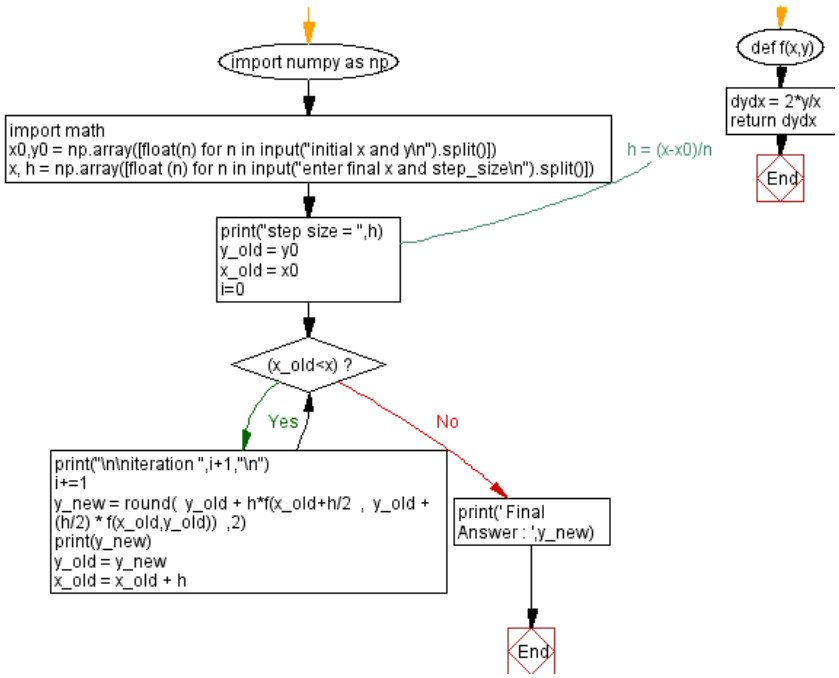
Where dy is the estimated incremental value of y from y_i and can be obtained from Euler's formula as $dy = hf(x_i, y_i)$ Therefore
$y_{i+1} = y_i + hf(x_i + h/2, y_i + h/2 * f(x_i, y_i) = y_i + hf(x_i + h/2, y_i + m_1h/2) = y_i + m_2h$

This is known as the modified Euler's method or improved polygon method. The method is also called the midpoint method.

Like Heun's method, this method is also of the order h^2 and therefore the local truncation error is of the order h^3 adn the global truncation error is of the order h^3.

6.7.2 Numerical

6.7.3 Flowchart



6.7.4 algorithm

1. Start
2. Define dy/dx as a function
3. Take input of initial x and y, final x, and step size
4. set $x_i = x_0$ and $y_i = y_i = y_0$
5. while x_i ¡ x_final, $y_{i+1} = y_old + h * f(x_old + h/2, y_old + (h/2) * f(x_old, y_old))$
6. When loop ends, return latest y_{i+1}
7. Stop

6.7.5 Python Program

```
import numpy as np
import math
def f(x,y):
  dydx = 2*y/x
  return dydx
```

```
x0,y0 = np.array([float(n) for n in input("initial x and y\n").split()])
x, h = np.array([float (n) for n in input("enter final x and step_size\n").split
    ()])

#h = (x-x0)/n
print("step size = ",h)
y_old = y0
x_old = x0
i=0
while (x_old<x):
  print("\n\niteration ",i+1,"\n")
  i+=1
  y_new = round(  y_old + h*f(x_old+h/2  ,  y_old + (h/2) * f(x_old,y_old))  ,2)
  print(y_new)
  y_old = y_new
  x_old = x_old + h
print(' Final Answer : ',y_new)
```

6.7.6 Output

```
(base) ayush@feynman:~/Desktop/Python ODE programs/ python3 polygon.py
initial x and y
1 2
enter final x and step_size
1.5 0.25
step size =  0.25
iteration  1
3.11
iteration  2
4.47
Final Answer :  4.47
```

Comparison of all methods

The comparison of all methods is on the basis of accuracy,convergence and stability.

Euler's method is the simplest one step method but if major weakness is large truncation error.

Heun's method is an improvement to Euler's method which provide solution or the ODE's with greater accuracy.

The **Taylor series method** is highly disliked for computation because evaluation of higher order derivatives took a lot of time.

Picard's method involves indefinite integration. Moreover the truncation error in above two methods shows that Taylor series method gives accuracy upto 7 decimal places while Picard's method gives accuracy upto 8 decimal places.

In **Euler's method** the computed value of y deviates rapidly, also the self starting characteristic of **Runge-kutta method** made it more favorable than **milne's**.

7 Interpolation Methods

INTRODUCTION Scientists and engineers are often faced with the task of estimating the value of dependent variable y for an intermediate value of the independent variable x, given a table of discrete data points (x_i, y_i), $i = 0, 1, ..., n$. This task can be accomplished by constructing a function $y(x)$ that will pass through the given set of points and then evaluating $y(x)$ for the specified value of x. The process of construction of $y(x)$ to fit a table of data points is called *curve fitting.*

Suppose y_i be the values of $y = f(x)$ for a set of values of x_i. Then the process of finding the value of y corresponding to any value of $x = x_i$ between x_0 and x_n is called *interpolation.* thus *interpolation is the technique of estimating the value of a function for any intermediate value of the independent variable* while the process of computing the value of the function outside the given range is called *extrapolation.*

POLYNOMIAL FORMS The most common form of an nth order polynomial is

$$p(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n \tag{16}$$

This form, known as the *power form*, is very convenient for differentiating and integrating the polynomial function and, therefore, are most widely used in mathematical analysis.

.....

Question 1 Consider the power form of $p(x)$ for $n = 1$,
Given that

$$p(100) = +3/7$$
$$p(101) = -4/7$$

obtain the linear polynomial $p(x)$ using four-digit floating point arithmetics. Verify the polynomial by substituting back the value $x = 100$ and $x = 101$.

Solution:

$$p(100 = a_0 + 100a_1 = +0.4286$$
$$p(101 = a_0 + 101a_1 = -0.5714$$

By solving the above equations, we get

$$a_1 = -1$$
$$a_0 = 100.4\text{(only four significant digits)}$$

Therefore,

$$p(x) = 100.4 - x$$

using this polynomial, we obatin

$$p(100) = 0.4$$
$$p(101) = -0.6$$

We can compare these results with the original values of $p(100)$ and $p(101)$. We have lost three decimal digits.

.....

The above question shows that the polynomials obtained using the power form may not always produce accurate results. In order to overcome such problems, we have alternative forms of representing a polynomial. One of them is the *shifted power form* as shown below:

$$p(x) = a_0 + a_1(x - C) + a_2(x - C)^2 + ... + a_n(x - C)^n. \tag{17}$$

where C is the point somewhere in the interval of interest. This form of representation significantly improves the accuracy of the polynomail evaluation.

.....

Question 2 Repeat **Question 1** using the shifted power form and four-digit arithmetic.

Solution: Shifted power form of first order $p(x)$ is

$$p(x) = a_0 + a_1(x - C)$$

Let us choose the center C as 100. Then

$$p(x) = a_0 + a_1(x - 100)$$

This gives,

$$p(100) = a_0 = 3/7 = 0.4286$$
$$p(101) = 0.4286 + a_1(101 - 100) = -0.5714$$
$$a_1 = -1$$

Thus the linear polynomial becomes

$$p(x) = 0.4286 - (x - 100)$$

Using this polynomial, we obtain

$$p(100) = 0.4286$$
$$p(101) = -0.5714$$

We can see the improvements in the results.

.....

Equation (2) is the *Taylor expansion* of $p(x)$ around the point C, when the coefficients a_i are replaced by appropriate function derivatives. It can be easily verified that

$$a_i = \frac{p^{(i)}(C)}{i!}, i = 0, 1, 2, ..., n$$

where $p^{(i)}(C)$ is the ith derivative of $p(x)$ a C.
There is a third form of $p(x)$ known as *Newton form*. This is a generalised shifted power form as shown below:

$$p(x) = a_0 + a_1(x - C_1) + a_2(x - C_1)(x - C_2) + ... + a_n(x - C_1)(x - C_2)...(x - C_n)$$

$$p_2(x) = b_0(x - x_1)(x - x_2) + b_1(x - x_0)(x - x_2) + b_2(x - x_0)(x - x_2)$$

In general form,

$$P_n(x) = \sum_{i=0}^{n} b_i \prod_{(j \neq i)(j=0)}^{n} (x - x_j) \tag{18}$$

7.1 Linear Interpolation

7.1.1 Theory

Linear interpolation is a method of curve fitting using linear polynomials to construct new data points within the range of a discrete set of known data points. The simplest form of interpolation is to approximate two data points by a straight line. Suppose we are given two points $(x_1, f(x_1))$ and $(x_2, f(x_2))$. These two points can be connected linearly. Using the concept of similar triangles, we can show that

$$\frac{f(x) - f(x_1)}{x - x_1} = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

Solving for $f(x)$, we get

$$f(x) = f(x_1) + (x - x_1)\frac{f(x_2) - f(x_1)}{x_2 - x_1} \tag{19}$$

Equation (5) is known as *linear interpolation formula*. The term

$$\frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

represents the slope of the line. Furthur, the similarity of equation (5) with the *Newton form* of polynomial of first-order.

$$C_1 = x_1$$
$$a_0 = f(x_1)$$
$$a_1 = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

The coefficient a_1 represents the first derivative of the function.

7.1.2 Numerical

Find the value of y at $x = 4$ given some set of values $(2, 4), (6, 7)$?
Solution:
Given the known values are,
$x = 4; x_1 = 2; x_2 = 6; y_1 = 4; y_2 = 7$
The interpolation formula is,

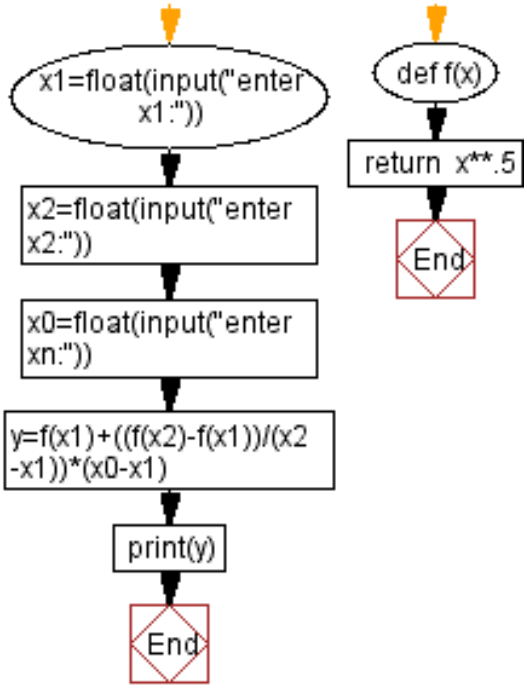
$$y = y_1 + \frac{(x-x_1)(y_2-y_1)}{x_2-x_1}$$

$$y = 4 + \frac{(4-2)(7-4)}{6-2}$$

$$y = 4 + \frac{6}{4}$$

$$y = \frac{11}{2}$$

7.1.3 Flowchart



(2).png (2).png

7.1.4 Algorithm

Step 1: Define the function f(x).
Step 2: Read the values of x1, x2 and x0.
Step 3: Set y=f(x1)+((f(x2)-f(x1))/(x2-x1))*(x0-x1).
Step 4: Print the value of y.

7.1.5 Python Program

```
def f(x):
    return  x**.5
x1=float(input("enter x1:"))
x2=float(input("enter x2:"))
x0=float(input("enter xn:"))
y=f(x1)+((f(x2)-f(x1))/(x2-x1))*(x0-x1)
print(y)
```

7.1.6 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======== RESTART: /home/ayush/Documents/python Files/CS lab/linear.py ========
enter x1:2
enter x2:3
enter xn:2.5
1.5731321849709863
>>>
======== RESTART: /home/ayush/Documents/python Files/CS lab/linear.py ========
enter x1:2
enter x2:4
enter xn:2.5
1.5606601717798214
>>>
```

7.2 Newton's Forward Interpolation

Interpolation is the technique of estimating the value of a function for any intermediate value of the independent variable, while the process of computing the value of the function outside the given range is called *extrapolation.*

Forward Differences: The differences $y_1-y_0, y_2-y_1, y_3-y_2, ..., y_n-y_{n-1}$ when denoted by $dy_0, dy_1, dy_2, ..., dy_{n-1}$ are respectively, called the first forward differences. Thus the first forward differences are:

$$\Delta Y_r = Y_{r+1} - Y_r$$

x	y	Δy	$\Delta^2 y$	$\Delta^3 y$	$\Delta^4 y$
x_0	y_0				
		Δy_0			
x_1	y_1		$\Delta^2 y_0$		
		Δy_1		$\Delta^3 y_0$	
x_2	y_2		$\Delta^2 y_1$		$\Delta^4 y_0$
		Δy_2		$\Delta^3 y_1$	
x_3	y_3		$\Delta^2 y_2$		
		Δy_3			
x_4	y_4				

NEWTON'S FORWARD INTERPOLATION FORMULA:

$$f(a+hu) = f(a) + u\Delta f(a) + \frac{u(u-1)}{2!}\Delta^2 f(a) + ... + \frac{u(u-1)(u-2)...(u-n+1)}{n!}\Delta^n f(a)$$

This formula is particularly useful for interpolating the values of $f(x)$ near the beginning of the set of values given. h is called the interval of difference and $u = (x-a)/h$, here a is the first term.

7.2.1 Numerical

Input: Value of Sin 52

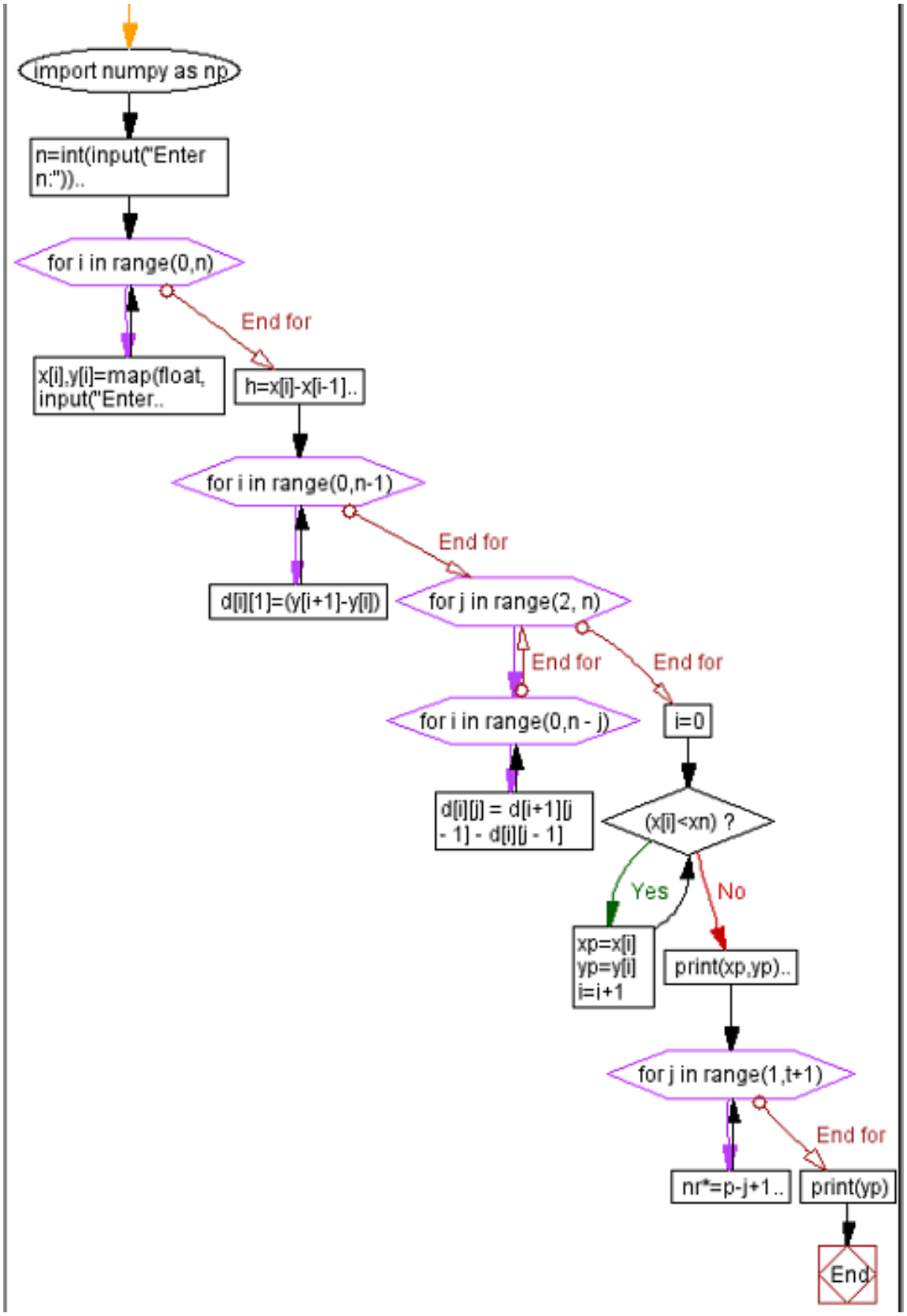
θ	45	50	55	60
$sin\theta$	.7071	.7660	.8192	.8660

Outut:

x	$10^1 y$	$10^1\Delta y$	$10^1\Delta^2 y$	$10^1\Delta^3 y$
45	.7071			
		589		
50	.7660		−57	
		532		−7
55	.8192		−64	
		468		
60	.8660			

After putting the values of the above table in the Newton's forward interpolation formula, the value of Sin 52 comes out to be 0.788003.

7.2.2 Flowchart



7.2.3 Algorithm

Step 1: $Importnumpy.$
Step 2: $Readthevaluesofnandorder.$
Step 3: $Readx, y, d.$
Step 4: $Introduceforloopfrom0ton.$
Step 5: $Seth = x[i] - x[i-1].$
Step 6: $Print(h).$
Step 7: $Readthevalueofx_n.$
Step 8: $foriinrange(0, n-1):$
$d[i][l] = (y[i+1] - y[i])$
Step 9: $forjinrange(2, n):$
$foriinrange(0, n-j):$
$d[i][j] = d[i+1][j-1] - d[i][j-1]$
Step 10: $Intializei = 0.$
Step 11: $Introducewhileloopas$
$while(x[i] < xn):$
$xp = x[i]$
$yp = y[i]$
$i = i + 1$
Step 12: $Print(xp, yp)$
Step 13: $Setp = (xn - xp)/h$
Step 14: $Printp$
Step 15: $Introduceforloopas$
$forjinrange(l, t+l):$
$nr* = p - j + 1$
$dr* = l$
$yp+ = (nr * d[2][j])/dr$
Step 16: $Printyp.$

7.2.4 Python Program

```
import numpy as np
n=int(input("Enter n:"))
t=int(input("Enter order:"))
x=np.zeros(n)
y=np.zeros(n)
d=np.zeros(n*n).reshape(n,n)
for i in range(0,n):
    x[i],y[i]=map(float,input("Enter x=,y=").split())
h=x[i]-x[i-1]
print(h)
xn=float(input("Enter xn:"))
for i in range(0,n-1):
   d[i][1]=(y[i+1]-y[i])
for j in range(2, n):
        for i in range(0,n - j):
            d[i][j] = d[i+1][j - 1] - d[i][j - 1]
i=0
while(x[i]<xn):
    xp=x[i]
   yp=y[i]
   i=i+1
```

```
    print(xp,yp)
    p=(xn-xp)/h
    print(p)
    nr=1
    dr=1
    for j in range(1,t+1):
        nr*=p-j+1
        dr*=1
        yp+=(nr*d[2][j])/dr

    print(yp)
```

7.2.5 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======== RESTART: /home/ayush/Documents/python Files/CS lab/forward.py ========
Enter n:7
Enter order:4
Enter x=,y=100 10.63
Enter x=,y=150 13.03
Enter x=,y=200 15.04
Enter x=,y=250 16.81
Enter x=,y=300 18.42
Enter x=,y=350 19.90
Enter x=,y=400 21.27
50.0
Enter xn:218
200.0 15.04
0.36
15.735375078399976
>>>
```

7.3 Newton's Backward Interpolation

7.3.1 Theory

Backward Differences: The differences $y_1 - y_0, y_2 - y_1, y_3 - y_2, ..., y_n - y_{n-1}$ when denoted by $dy_1, dy_2, ..., dy_n$ are respectively, called the first backward difference. Thus the first backward differences are:

$$Y_r = Y_r - Y_{r-1}$$

x	y	y	^2y	^3y	^4y
x_0	y_0				
		y_1			
x_1	y_1		$^2y_2$		
		y_2		$^3y_3$	
x_2	y_2		$^2y_3$		$^4y_5$
		y_3		$^3y_4$	
x_3	y_3		$^2y_4$		
		y_4			
x_4	y_4				

NEWTON'S BACKWARD INTERPOLATION FORMULA:

$$f(a + nh + uh) = f(a + nh) + uf(a + nh) + \frac{u(u+1)}{2!}{}^2 f(a + nh) + ... + \frac{u(u+1)...(u+n-1)}{n!}{}^n f(a + nh)$$

This formula is useful when the value of $f(x)$ is required near the end of the table. h is called the interval of difference and $u = (x - an)/h$, here an is the last term.

7.3.2 Numerical

Input: Population in 1925

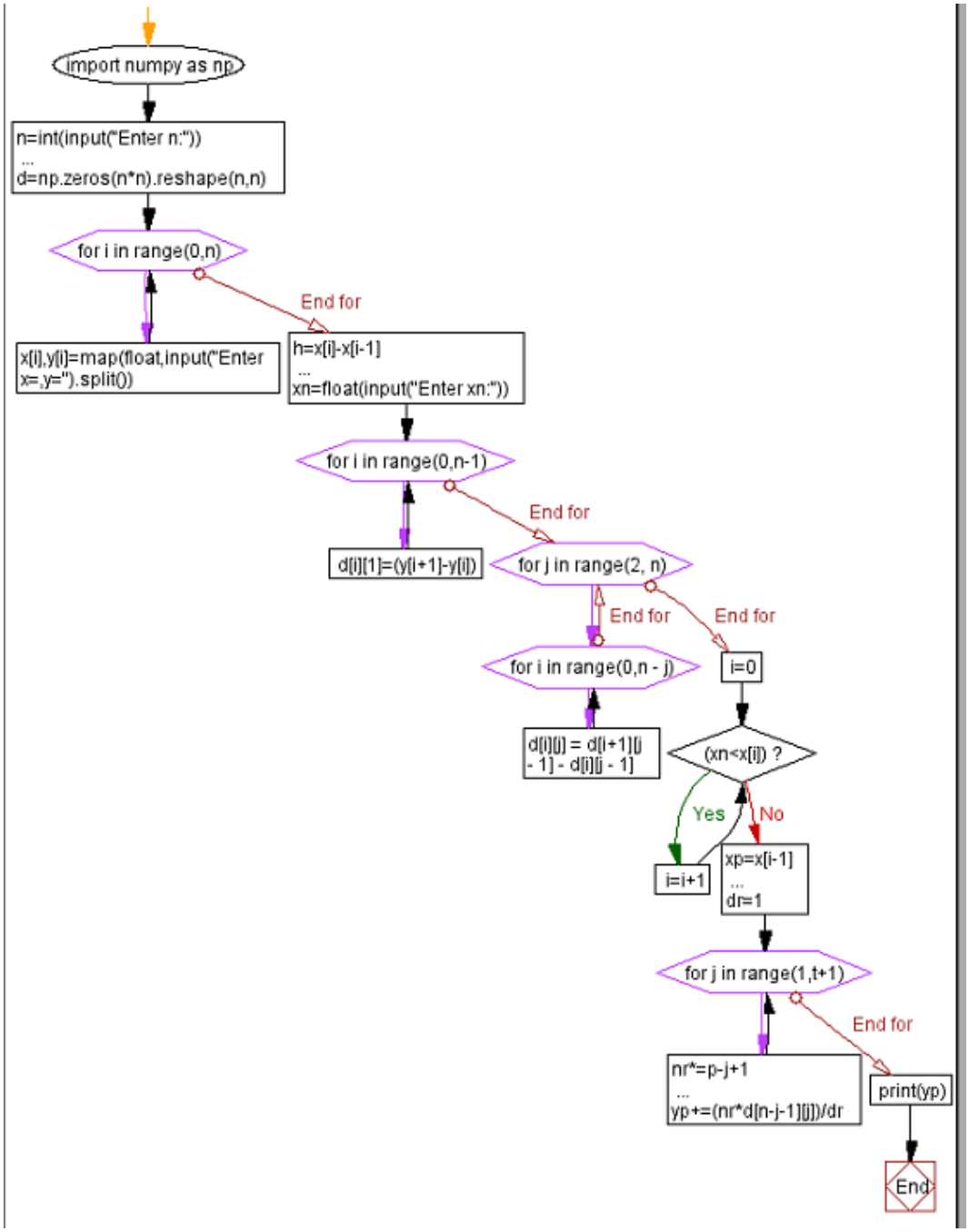
$Year(x)$:	1891	1901	1911	1921	1931
$Population(y)(thousands)$:	46	66	81	93	101

Output:

x	y	y	^2y	^3y	^4y
1891	46				
		20			
1901	66		-5		
		15		2	
1911	81		-3		-3
		12		-1	
1921	93		-4		
		8			
1931	101				

After putting the above values in Newton's backward interpolation formula, the values of 1925 comes out to be 96.8368.

7.3.3 Flowchart



7.3.4 algorithm

Step 1: Import numpy.
Step 2: Read the values of n and order.
Step 3: Read x, y, d.
Step 4: Introduce for loop from 0 to n.
Put $x[i], y[i]$=map(float, input("Enter x=,y=").slit()).
Step 5: Set $h = x[i] - x[i-1]$.
Step 6: Print(h).
Step 7: Read the value of x_n.
Step 8: for i in range(0,n-1):
d[i][l]=(y[i+1]-y[i])
Step 9: for j in range(2,n):
for i in range(0,n-j):
d[i][j]=d[i+1][j-1]-d[i][j-1]
Step 10: Initialize $i = 0$.

Step 11: Introduce while loop as
while($x[i] < xn$):
$i = i + 1$
xp=x[i-1]
$yp = y[i - 1]$
Step 12: Set p = (xn-xp)/h
Step 13: Introduce for loop as
for j in range(l,t+l):
nr*=p-j+1
dr*=l
yp+=(nr*d[n-j-1][j])/dr
Step 14: Print yp.

7.3.5 Python Program

```
import numpy as np
n=int(input("Enter n:"))
t=int(input("Enter order:"))
x=np.zeros(n)
y=np.zeros(n)
d=np.zeros(n*n).reshape(n,n)
for i in range(0,n):
    x[i],y[i]=map(float,input("Enter x=,y=").split())
h=x[i]-x[i-1]
print(h)
xn=float(input("Enter xn:"))
for i in range(0,n-1):
    d[i][1]=(y[i+1]-y[i])
for j in range(2, n):
        for i in range(0,n - j):
            d[i][j] = d[i+1][j - 1] - d[i][j - 1]

i=0
while(xn<x[i]):
    i=i+1

xp=x[i-1]
print(xp)
yp=y[i-1]
print(yp)
p=(xn-xp)/h
print(p)

nr=1
dr=1
for j in range(1,t+1):
    nr*=p-j+1
    dr*=1
    yp+=(nr*d[n-j-1][j])/dr

print(yp)
```

7.3.6 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

```
======= RESTART: /home/ayush/Documents/python Files/CS lab/backward.py =======
Enter n:7
Enter order:4
Enter x=,y=100 10.63
Enter x=,y=150 13.03
Enter x=,y=200 15.04
Enter x=,y=250 16.81
Enter x=,y=300 18.42
Enter x=,y=350 19.90
Enter x=,y=400 21.27
50.0
Enter xn:410
400.0
21.27
0.2
yp= 21.575423999999984
>>>
```

7.4 Newton's Divided Difference Method

7.4.1 Theory

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

Suppose $f(x_0), f(x_2), ..., f(x_n)$ be the $(n+1)$ values of the function $y = f(x)$ corresponding to the arguments $x = x_0, x_1, ..., x_n$, where interval difference are not same. Then the first divided difference is given by

$$f[x_0, x_1] = \frac{f(x_1)-f(x_0)}{x_1-x_0}$$

The second divided difference is given by

$$f[x_0, x_1, x_2] = \frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0}$$

and so on... Divided differences are symmetric with respect to the arguments i.e **indepedent of the order of arguments**.
so, $f[x_0, x_1] = f[x_1, x_0]$
$f[x_0, x_1, x_2] = f[x_2, x_1, x_0] = f[x_1, x_2, x_0]$
By using first divided difference, second divided difference as so on. A table is formed which is called the divided difference table.
Divided difference table:

x_i	f_i	$F(x_i, x_j)$	$F(x_i, x_j, x_k)$
x_1	f_1		
		$f[x_1, x_2] = \frac{f_2-f_1}{x_2-x_1}$	
x_2	f_2		$f[x_1, x_2, x_3] = \frac{f[x_3,x_2]-f[x_2,x_1]}{x_3-x_1}$
		$f[x_2, x_3] = \frac{f_3-f_2}{x_3-x_2}$	
x_3	f_3		

NEWTON'S DIVIDED DIFFERENCE INTERPOLATION FORMULA:

$$f(x) = f(x_0)+(x-x_0)f[x_0, x_1]+(x-x_0)(x-x_1)f[x_0, x_1, x_2]+...+(x-x_0)(x-x_1)...(x-x_{k-1})f[x_0, x_1, x_2..., x_k]$$

7.4.2 Numerical

Input: Find the value at 7

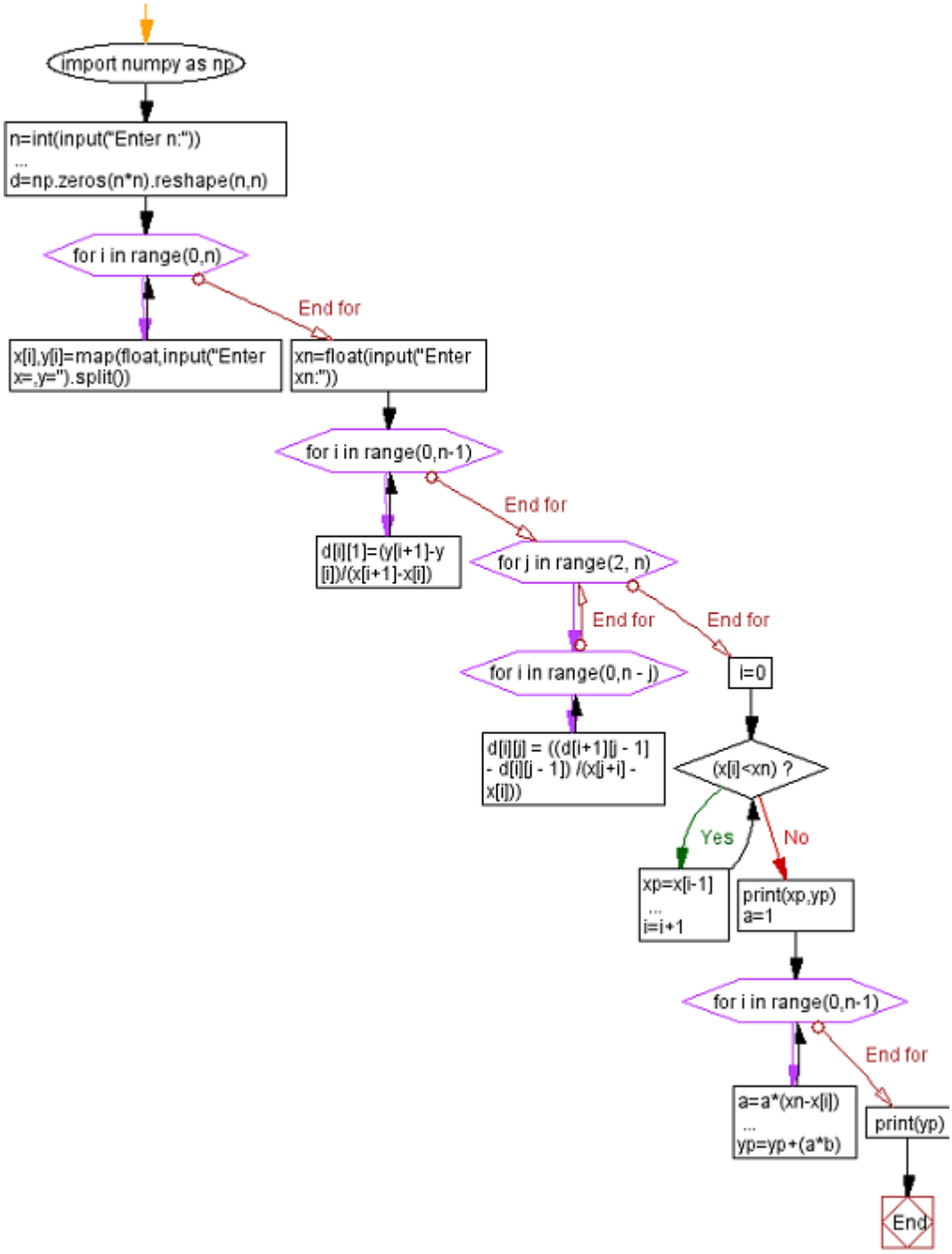
x	5	6	9	11
$y = f(x)$	12	13	14	16

Output:

x	$y = f(x)$	$f(x_i, x_j)$	$f(x_i, x_j, x_k)$	$f(x_i, x_j, x_k, x_l)$
5	12			
		1		
6	13		$\frac{-1}{6}$	
		$\frac{1}{3}$		$\frac{1}{20}$
9	14		$\frac{2}{15}$	
		1		
11	16			

After putting the values of the above table in Newton's divided difference interpolation formula, the value of 7 comes out to be 13.47.

7.4.3 Flowchart



7.4.4 algorithm

Step 1: Import numpy.
Step 2: Read the values of n.
Step 3: Read x, y, d.
Step 4: Introduce for loop from 0 to n.
Put $x[i], y[i]$=map(float, input("Enter x=,y=").slit()).
Step 5: Read the value of x_n.
Step 6: for i in range(0,n-1):
d[i][l]=(y[i+1]-y[i])/(x[i+1]-x[i])
Step 7: for j in range(2,n):
for i in range(0,n-j):
d[i][j]=(d[i+1][j-1]-d[i][j-1])/(x[j+1]-x[i]))
Step 8: Intialize $i = 0$.

Step 9: Introduce while loop as
while($x[i] < xn$):
xp=x[i-1]
$yp = y[i-1]$
$i = i+1$
Step 10: Print(xp,yp)
Step 11: Initialize $a = 1$.
Step 12: Introduce for loop as
for i in range(0,n-1):
$a = a*(xn-x[i])$
b=d[0][i+1]
$yp = yp+(a*b)$
Step 13: Print yp.

7.4.5 Python Program

```
import numpy as np
n=int(input("Enter n:"))
x=np.zeros(n)
y=np.zeros(n)
d=np.zeros(n*n).reshape(n,n)
for i in range(0,n):
    x[i],y[i]=map(float,input("Enter x=,y=").split())
xn=float(input("Enter xn:"))
for i in range(0,n-1):
    d[i][1]=(y[i+1]-y[i])/(x[i+1]-x[i])

for j in range(2, n):
        for i in range(0,n - j):
            d[i][j] = ((d[i+1][j - 1] - d[i][j - 1]) /(x[j+i] - x[i]));

i=0
while(x[i]<xn):
    xp=x[i-1]
    yp=y[i-1]
    i=i+1
print(xp,yp)
a=1
for i in range(0,n-1):
        a=a*(xn-x[i])

        b=d[0][i+1]

        yp=yp+(a*b)

print(yp)
```

7.4.6 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======== RESTART: /home/ayush/Documents/python Files/CS lab/divided.py ========
Enter n:5
Enter x=,y=5 150
Enter x=,y=7 392
Enter x=,y=11 1452
```

```
Enter x=,y=13 2366
Enter x=,y=17 5202
Enter xn:9
5.0 150.0
810.0
>>>
```

Comparison Chart Between Different types of Interpolation Method

Parameter	Forward	Backward	Divided
Convergence	Fast	Fast	More Faster
Intervals	Equal	Equal	Unequal
Execution Time	2.6226043701171875e-06 sec	2.1457672119140625e-06 se	2.1457672119140625e-06 sec
Error	En(x) = (x - x0)(x - x0 - h) . . .(x - x0 - nh) f(n+1)(x) / (n+1)!	Same as forward	En(x) =f(x)-pN(x)

8 Solution of linear equations

8.1 Gauss Elimination Method

8.1.1 Theory

In this method, the unknown are eliminated successively and the system is reduced to an upper triangular system from which the unknowns are found by back substitution. The method is quite general and is well-adapted for computer operations. Here we shall explain it by considering a system of three equations for sake of clarity. Consider the equations

$$a_1x + b_1y + c_1z = d_1$$

$$a_2x + b_2y + c_2z = d_2$$

$$a_3x + b_3y + c_3z = d_3$$

Step 1.To eliminate x from second third equations. Assuming $a_1$is not equal to 0, we eliminate x from the second equation by subtracting(a_2/a_1) times the first equation from the second equation. Similarly we eliminate x from the third equation by eliminating (a_3/a_1) times the first equation from the third equation. We thus, get the new system

$$a_1x + b_1y + c_1z = d_1,$$

$$b_2'y + c_2'z = d_2,$$

$$b_3'y + c_3'z = d_3$$

Here the first equation is called the pivotal equation and a_1 is called the first pivot.

Step 2.To eliminate y from third equation. Assuming b_2' is not equal to 0, we eliminate y from the third equation, by subtracting b_3'/b_2' times the second equation from the third equation. We thus get the new system

$$a_1x + b_1y + c_1z = d_1,$$

$$b_2' + c_2' = d_2'$$

$$c_3''z = d_3''$$

Here the second equation is the pivotal equation and b_2' is the new pivot.

Step 3. To evaluate the unknowns. The values of x, y, z are found from the reduced system (3) by back substitution.

8.1.2 Numerical

Apply Gauss elimination method to solve the equations x+4y-z=-5; x+y-6z=-12; 3x-y-z=4. **solution.** We have

$$x + 4y - z = -5....(1)$$

$$x + y - 6z = -12....(2)$$

$$3x - y - z = 4....(3)$$

Step 1. To eliminate x, operate (2)-(1) and (3)-3(1):

$$-3y - 5z = -7....(4)$$

$$-13y + 2z = 19....(5)$$

Step 2. To eliminate y, operate (5)-13/3(4):

$$(71/3)z = (148/3)....(6)$$

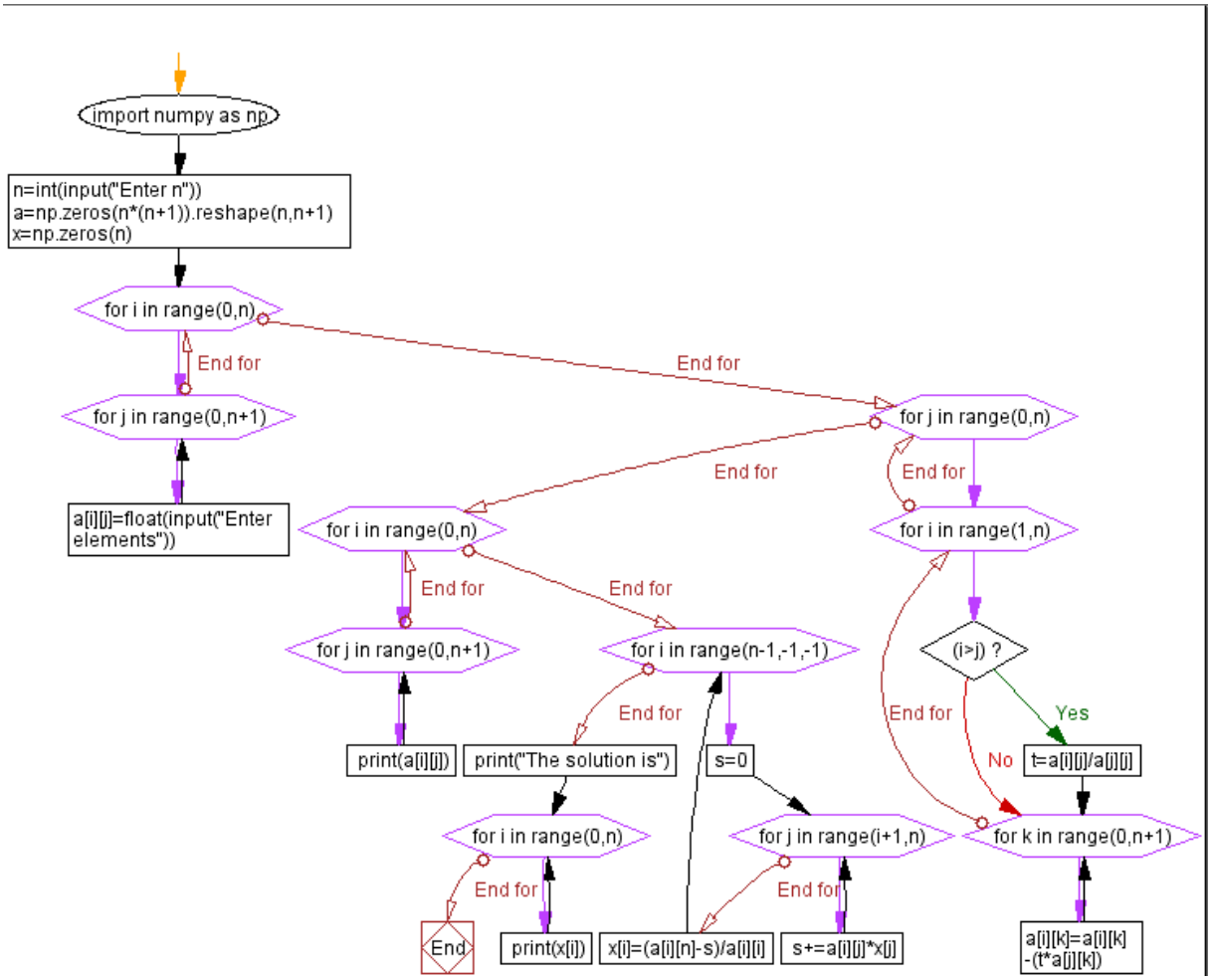
Step 3. By back substitution, we get

$$From(6) : z = (148/71)$$

$$From(4) : y = (-81/71)$$

$$From(1) : x = (117/71)$$

8.1.3 Flowchart



8.1.4 Algorithm

1.Start
2.Declare the variables and read the order of the matrix n.
3.Take the coefficients of the linear equation as: Do for k=1 to n Do for j=1 to n+1 Read a[k][j] End for j End for k
4.Do for k=1 to n-1 Do for i=k+1 to n Do for j=k+1 to n+1 a[i][j] = a[i][j] – a[i][k] /a[k][k] * a[k][j] End for j End for i End for k
5.Compute x[n] = a[n][n+1]/a[n][n]
6.Do for k=n-1 to 1 sum = 0 Do for j=k+1 to n sum = sum + a[k][j] * x[j] End for j x[k] = 1/a[k][k] * (a[k][n+1] – sum) End for k
7.Display the result x[k]
8.Stop

8.1.5 Python Program

```
import numpy as np
n=int(input("Enter n"))
a=np.zeros(n*(n+1)).reshape(n,n+1)
x=np.zeros(n)
for i in range(0,n):
    for j in range(0,n+1):
        a[i][j]=float(input("Enter elements"))

for j in range(0,n):
    for i in range(1,n):
        if(i>j):
            t=a[i][j]/a[j][j]
```

```
        for k in range(0,n+1):
            a[i][k]=a[i][k]-(t*a[j][k])



for i in range(0,n):
    for j in range(0,n+1):
        print(a[i][j])


for i in range(n-1,-1,-1):
    s=0
    for j in range(i+1,n):
        s+=a[i][j]*x[j]
    x[i]=(a[i][n]-s)/a[i][i]

print("The solution is")
for i in range(0,n):
    print(x[i])
```

8.1.6 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
====== RESTART: /home/ayush/Documents/python Files/Programs/eliminate.py ======
Enter n3
Enter elements1
4
-1
-5
1
1
-6
-12
3
-1
-1
4

Enter elementsEnter elementsEnter elementsEnter elementsEnter elementsEnter
    elementsEnter elementsEnter elementsEnter elementsEnter elementsEnter
    elements

The solution is

1.6478873239436744
-1.1408450704225384
2.0845070422535215
>>>
```

8.2 Gauss Jordan Method

8.2.1 Theory

This is a modification of the Gauss elimination method. In this method, elimination of unknowns is performed not in the equations below but in the equations above also, ultimately reducing the system to a diagonal matrix from i.e. each equation involving only one unknown. From these equations the unknowns x, y, z can be obtained readily.

Thus in this method, the labour of back-substitution for finding the unknowns is saved at the cost of additional calculations.

8.2.2 Numerical

Apply Gauss Jordan method to solve the equations x+y+z=9; 2x-3y+4z=13; 3x+4y+5z=40. **solution.** We have

$$x + y + z = 9....(1)$$

$$2x - 3y + 4z = 13....(2)$$

$$3x + 4y + 5z = 40....(3)$$

*Step 1.*To eliminate x from (2) and (3), operate (2)-2(1) and (3)-3(1):

$$x + y + z = 9....(4)$$

$$-5y + 2z = -5....(5)$$

$$y + 2z = 13....(6)$$

*Step 2.*To eliminate y from (4) and (6), operate (4)+1/5(5) and (6)+1/5(5):

$$x + (7/5)z = 8....(7)$$

$$-5y + 2z = -5....(8)$$

$$(12/5)z = 12....(9)$$

*Step 3.*To eliminate z from (7) and (8), operate (7)-(7/12)(9) and (8)-(5/6)(9):

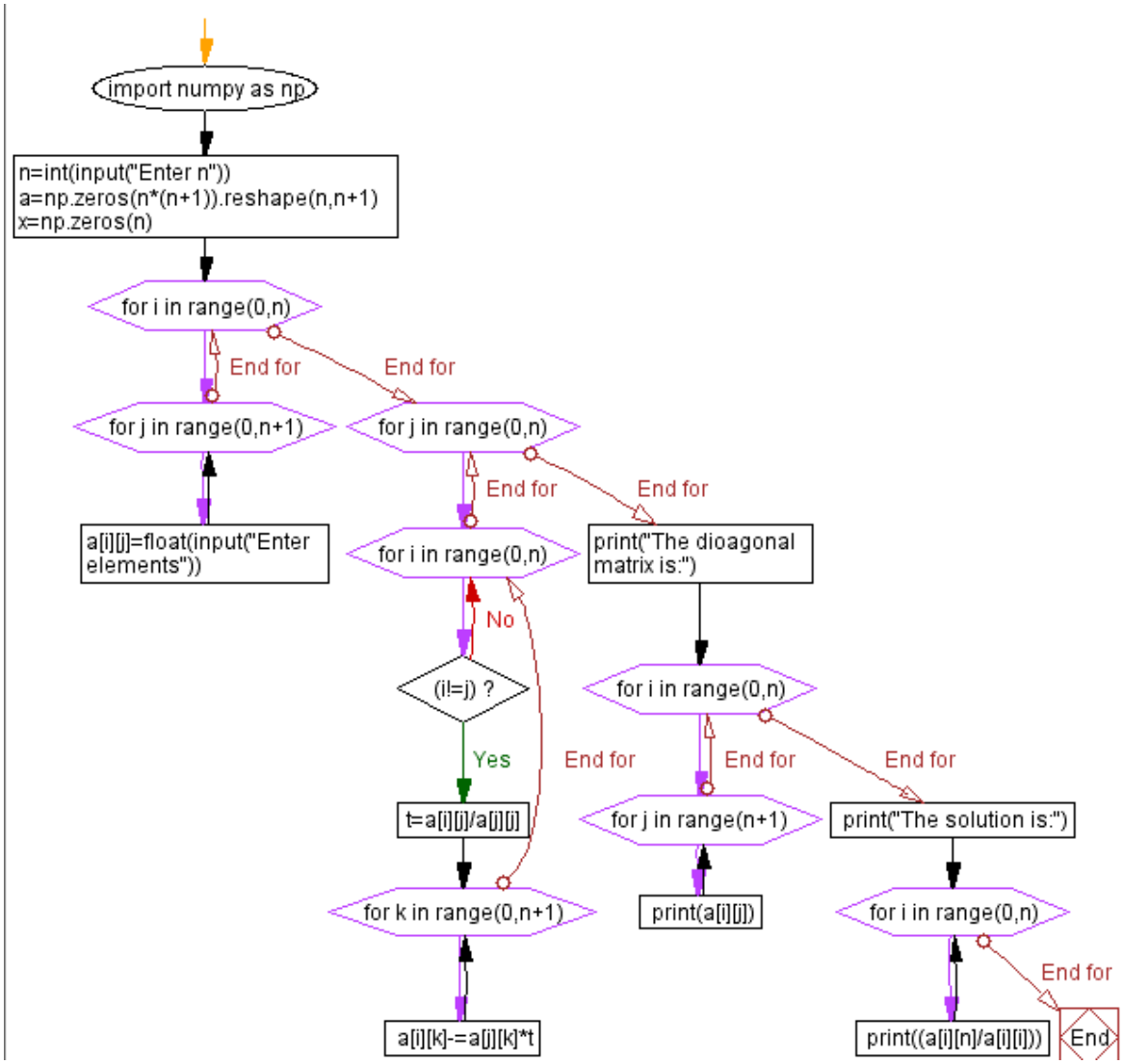
$$x = 1$$

$$-5y = -15$$

$$(12/5)z = 12$$

Hence the solution is x=1, y=3, z=5.

8.2.3 Flowchart



8.2.4 Algorithm

1.Start
2.Read the order of the matrix 'n' and read the coefficients of the linear equations.
3.Do for k=1 to n Do for l=k+1 to n+1 a[k][l] = a[k][l] / a[k][k] End for l Set a[k][k] = 1 Do for i=1 to n if (i not equal to k) then, Do for j=k+1 to n+1 a[i][j] = a[i][j] – (a[k][j] * a[i][k]) End for j End for i End for k
4.Do for m=1 to n x[m] = a[m][n+1] Display x[m] End for m
5.Stop

8.2.5 Python Program

```
import numpy as np
n=int(input("Enter n"))
a=np.zeros(n*(n+1)).reshape(n,n+1)
x=np.zeros(n)
for i in range(0,n):
    for j in range(0,n+1):
        a[i][j]=float(input("Enter elements"))

for j in range(0,n):
    for i in range(0,n):
        if(i!=j):
            t=a[i][j]/a[j][j]
            for k in range(0,n+1):
```

```
                a[i][k]-=a[j][k]*t

print("The dioagonal matrix is:")
for i in range(0,n):
    for j in range(n+1):
        print(a[i][j])

print("The solution is:")
for i in range(0,n):
    print((a[i][n]/a[i][i]))
```

8.2.6 Output

```
Python 3.7.3 (default, Oct  7 2019, 12:56:13)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======= RESTART: /home/ayush/Documents/python Files/Programs/jordan.py =======
Enter n3
Enter elements1
Enter elements1
Enter elements1
Enter elements9
Enter elements2
Enter elements-3
Enter elements4
Enter elements13
Enter elements3
Enter elements4
Enter elements5
Enter elements40
The dioagonal matrix is:
1.0
0.0
-2.220446049250313e-16
1.0
0.0
-5.0
0.0
-15.0
0.0
0.0
2.4
12.0
The solution is:
1.0
3.0
5.0
>>>
```

Part III

Programs to solve Physical Problem

9 Solar Panel

Theory:-

We can simply guess that as we apply more intensity of light on LDR and solar cell, LDR becomes less resistive and solar gives more voltage, so that they are inversely praprotional in relation and we get graph according to that.

Python Code

```
# expeyes intialization
import eyes17.eyes
p= eyes17.eyes.open()
from pylab import*
from scipy import stats
import numpy as np

v=np.zeros(200)
r=np.zeros(200)
for i in range(0,200):
    v[i]=(p.get_voltage('A2'))
    r[i]=(p.get_resistance())
m,c, corr, s, err=stats.linregress(r,v)
r1=np.linspace(r[0],r[199])
v1=m*r1+c
scatter(r,v)
plot(r1,v1)
plot(r1,v1)
xlabel("Resistance(Ohm)")
ylabel("Voltage(V)")
title("Voltage vs Resistance")
show()
```

Observation:-