

2/20/2022

**Comsats University**

**Islamabad**

**Numerical Computation**



**Final Semester Project**

**Presented**

**By**

**Iqra Arshad**

CIIT/SP20/BSM/018/ISB

**Aliza Jabbar**

CIIT/SP20/BSM/006/ISB

**DEPARTMENT OF MATHEMATICS**

COMSATS INSTITUTE OF INFORMATION TECHNOLOGY, ISLAMABAD

An Undergraduate Semester Project Report Submitted to

**Dr. Umair Umar**

*By*

**Name** **Registration Number**

**Aliza Jabbar** CIIT/SP20-BSM-006/ISB

**Iqra Arshad** CIIT/SP20-BSM-018/ISB

Supervised by

Dr. Umair Umar,

Faculty Member,

Department of Mathematics

CIIT Islamabad.

COMSATS INSTITUTE OF INFORMATION TECHNOLOGY, ISLAMABAD

***Declaration***

*I, hereby declare that this project neither as a whole nor as a part there of has been copied out from any source. It is further declared that I have developed this project and the accompanied report entirely on the basis of my personal efforts made under the sincere guidance of my supervisor. No portion of the work presented in this report has been submitted in the support of any other degree or qualification of this or any other University or Institute of learning, if found I shall stand responsible.*

**Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Aliza Jabbar**

**Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Iqra Arshad**

COMSATS INSTITUTE OF INFORMATION TECHNOLOGY, ISLAMABAD

Dedication

Dedicated to

My Beloved Parents and Brothers

***Acknowledgements***

I offer my humblest gratitude to Almighty Allah, the Beneficent and the most Merciful, who blessed me an opportunity to stay on the way of knowledge. Great blessings to Holly Prophet Muhammad (S.A.W), who built the height and standard of character for the world and guided his ‘Ummah’ to seek knowledge from cradle to grave.

Foremost, I would like express my sincere gratitude to my instructor Dr. Umair Umar for the continuous support to my study and research, for his patience, motivation, enthusiasm, and immense knowledge. The door to his office was always open whenever I ran into a trouble spot or had a question about my research or writing. She consistently allowed this project to be my own work, but steered me in the right direction whenever he thought I needed it.

Finally, I must express my very profound gratitude to my parents and to my siblings for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of writing this project. This accomplishment would not have been possible without them. Thank you.

Aliza Jabbar

SP20-BSM-006

**Contents**

**Introduction and Error Analysis**

Types of errors

**Methods to Solve Non-linear equations**

The Bisection Method

Fixed-Point Iteration

Newton Raphson Method

Secant Method

Method of False Position

**Direct Methods for Solving System of Linear equations**

Gaussian Elimination Method

Pivoting Strategies

Matrix Factorization

**Iterative Techniques in Matrix Algebra**

Norms of Vectors

Eigenvalues and Eigenvectors

Iterative Techniques for Solving Linear Systems

* Gauss Jacobi Method……………………………………………………………………………………………….
* Gauss Seidel Method………………………………………………………………………………………………..
* Successive Over Relaxation (SOR) Method……………………………………………………………….

**Interpolation and Polynomial Approximation**

Interpolation and Weierstrass Approximation Theorem

Finite Differences

* Forward……………………………………………………………………………………………………………………
* Backward…………………………………………………………………………………………………………………
* Central……………………………………………………………………………………………………………………..

Differences Interpolation

* Newton’s Forward Difference…………………………………………………………………………………..
* Newton’s Backward Difference………………………………………………………………………………..
* Newton’s Central Difference…………………………………………………………………………………….

Spline Interpolation

* Linear Spline Interpolation………………………………………………………………………………………
* Quadratic Spline Interpolation………………………………………………………………………………..
* Cubic Spline Interpolation………………………………………………………………………………………

**Numerical Integration**

Trapezoidel Rule

Simpson’s 1/3 Rule

Simpson’s 3/8 Rule

**Preface**

A fundamental practice in mathematics is to determine the number of ways a certain outcome may occur. In this project, we are about to discuss the numerical methods; a complete set of procedures which gives an approximate solution to the mathematical problems.

**Why we use numerical methods for solving equations?**

As analytic solutions are often either too tiresome or simply do not exist, we need to find an approximate method of solution. This is where numerical analysis comes into picture.

**Introduction**

**Numerical Analysis:**

**Numerical Analysis is the branch of mathematics that provides tools and methods for solving mathematical problems in numerical form.**

**In numerical analysis we are mainly interested in implementation and analysis of numerical algorithms for finding an approximate solution to a mathematical problem.**

**Error Analysis:**

Error Analysis is the study and evaluation of error. An error in a numerical computation is simply the difference between the actual (true) value of a quantity and its computed (approximate) value. There are three common ways to express the size of error in a computed result:

* Absolute error
* Relative error
* Percentage error

1. **Absolute Error:** Suppose that \* is an approximation (computed value) to.

absolute error = |true value – approximate value|

1. **Relative Error:**
2. **Percentage Relative Error:**

Relative error expressed in percentage is called percentage relative error; defined by;

PE = 100 × Er

Chapter 01:

**SOLUTION OF NON-LINEAR EQUATIONS**

**Method for solutions of single variable non-linear equations:**

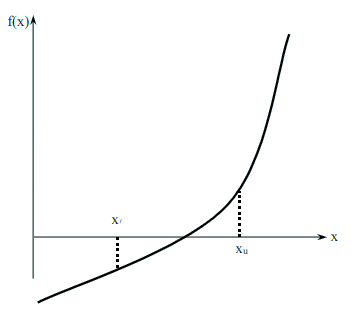
1. Bisection Method/ Binary Search Method/ Midpoint Method/ Halving Method
2. False Position Method (Regula-Falsi Method)
3. Newton-Raphson Method
4. Secant Method
5. **Fixed Point Method/ Fixed Position Iteration Method

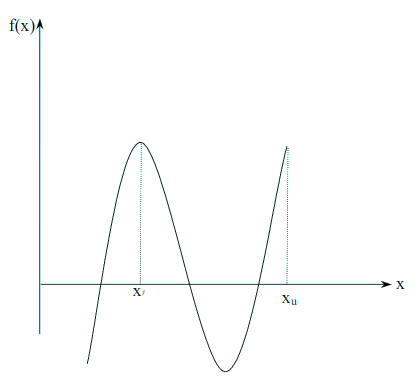
Figure 1 At least one root exists between the two points if the function is real, continuous, and changes sign.

**Convergence Criteria for a Numerical Computation:**

If the method leads to the value close to the exact solution, then we say that the method is convergent otherwise the method is divergent. i.e. 

**Bisection Method:**

**Theorem:** An equation f(x) = 0, where f(x)is a real continuous function, has at least one root between xl and xu if f(xl) f(xu) < 0.



**Figure 2**  If function *f(x)* does not change sign between two points, roots of the equation *f(x)=0* may still exist between the two points.

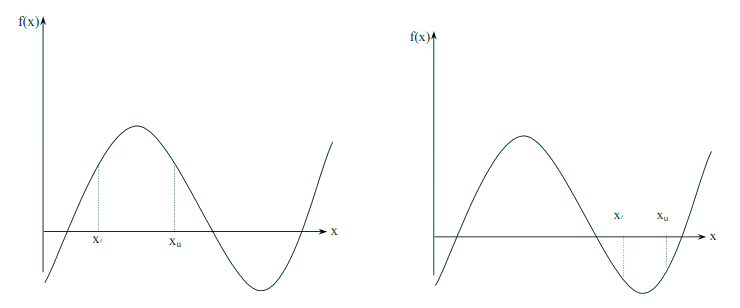
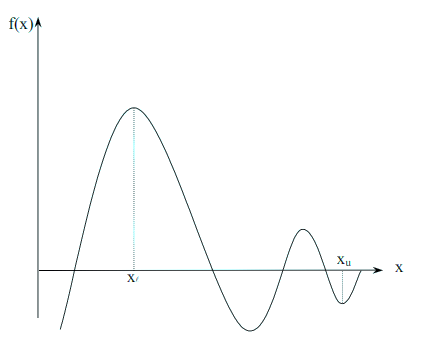


Figure 3 If the function *f(x)* does not change sign between two points, there may not be any roots for the equation *f(x)* between the two points.



**Figure 4** If the Function *f(x)* changes sign between two points, more than one root for the equation *f(x) = 0*  may exist between the two points.

**Algorithm for Bisection Method:**

**Step 1:** Choose xl and xu as two guesses for the root such that f(xl) f(xu ) < 0, or in other words, f(x) changes sign between xl and xu . This was demonstrated in Figure 1.

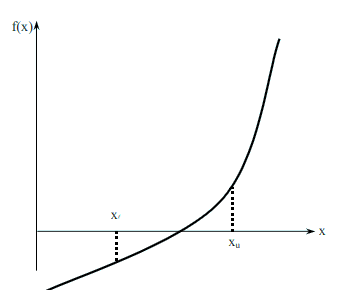


Figure 1

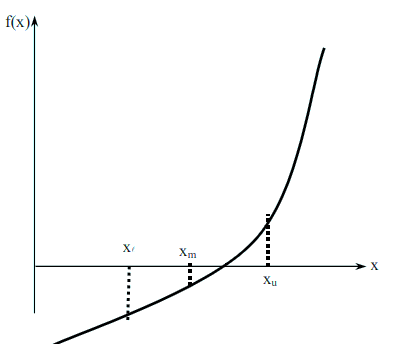
**Step 2:** Estimate the root, xm of the equation f (x) = 0 as the mid-point between xl and xu as

Figure 2 Estimate of xm

**Step 3:**

Now check the following

1. If , then the root lies between xl and xm; then xl = xl ; xu = xm.
2. If , then the root lies between xm and xu; then xl = xm; xu = xu.
3. If , then the root is xm. Stop the algorithm if this is true.

**Step 4:**

Find the new estimate of the root

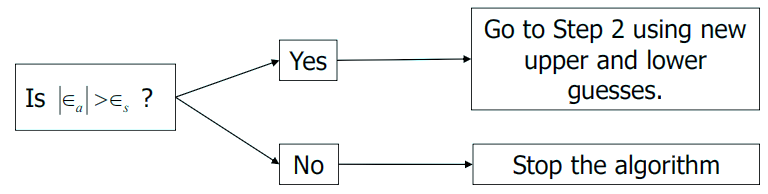
Find the absolute relative approximate error

Where = previous estimate of root

= current estimate of root

**Step 5:**

Compare the absolute relative approximate error with the pre-specified error tolerance .

****

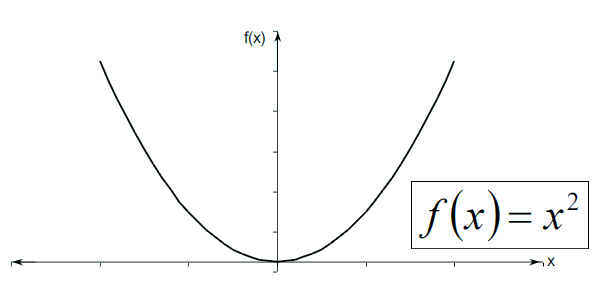
Note one should also check whether the number of iterations is more than the maximum number of iterations allowed. If so, one needs to terminate the algorithm and notify the user about it.

**Advantages:**

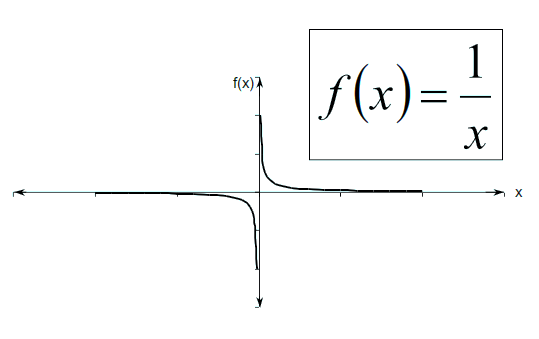
* Always convergent.
* The root bracket gets halved with each iteration - guaranteed.

**Drawbacks:**

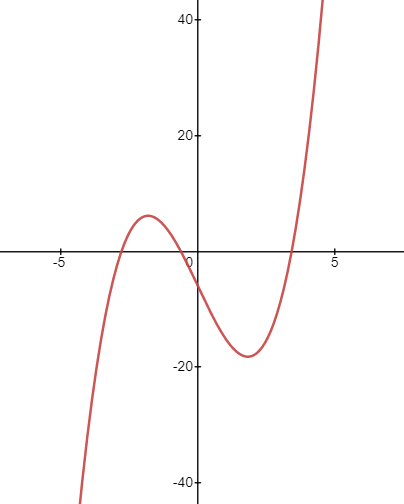
* Slow convergence.
* If one of the initial guesses is close to the root, the convergence is slower.
* If a function f(x) is such that it just touches the x-axis it will be unable to find the lower and upper guesses.



* Function changes sign but root does not exist.



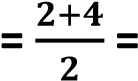
**Example:**

**Solve for the simple polynomial equation f(x)= x3-10x-6 for roots between x=2 and x=4**

**Solution:**

|  |  |  |
| --- | --- | --- |
| X | 2 | 4 |
| f(x) | -18 | 18 |

Since f (2). f (4) <0 therefore the root lies between 2 and 4 and the interval is [2,4] .So, now bisecting the interval;

Xm 3 so f(3) = -9 (-ve)

So the next interval in which the root lies is [3,4]; because f(3).f(4)< 0

Bisecting the interval [3,4] ;

Xm =

f (3.5) = 1.875(+ve)

For interval [3,3.5];

xm *=*3.25 so f(3.25) = -4.17 (-ve)

For interval [3.25,3.5];

xm = = 3.375

so, f (3.375) = -1.3 (-ve)

For interval [3.375,3.5];

Xm = (3.375+3.5)/2 = 3.438

so, f (3.438) = 0.266 (+ve)

For interval [3.375,3.438];

so, xm = (3.375+3.438)/2 = 3.407 ;

and f(3.407)=-0.523 (-ve)

For interval [3.407,3.438];

Xm = (3.407+3.438)/2 = 3.423

so, f (3.423) = -0.123 (-ve)

For interval [3.423,3.438];

Xm = (3.423+3.438)/2 = 3.431

So, f (3.431) = 0.079 (+ve)

For interval [3.423,3.431];

Xm = (3.423+3.431)/2 = 3.427

So, f (3.427) = -0.022(-ve)

For interval [3.427,3.431];

Xm = (3.427+3.431)/2 = 3.429

So, f (3.429) = 0.028(+ve)

For interval [3.427,3.429];

Xm = (3.427+3.429)/2 = 3.428

So, f (3.428) = 0.0031(+ve)

For interval [3.427,3.428];

Xm = (3.427+3.428)/2 = 3.428

So, f (3.428) = 0.0031(+ve)

Hence root is 3.428 because roots are repeated.

**Coding:**

from math import sin,cos # importing from math library

def bisection(x0,x1,e): # fuction starts here

step = 1 # initializing step variable with 1

condition = True # boolean variable with true as initial value

while condition: # if boolean is true then execute while else dont

x2 = (x0+x1)/2 # finding mean of x0 and x1

print('%d : %d \t\t %d \t\t %d \t\t %d \t\t %d \t\t %d ' %(step,x0 , x1 , f(x0) , f(x1), x2 , f(x2))) # printing iteration, x2 and f(x2)

if f(x0) \* f(x2) < 0: # if product of functional value of x0 and x2 is less than 0

x1 = x2 # then exchange value of x1 with mean value(x2)

else:

x0 = x2 # else exchange value of x0 with mean value(x2)

step = step +1 # increment step number

# storing true in condition if abs of functional value of x2 is greater than tolerance

condition = abs(f(x2)) > e

print('root is :%0.8f '%x2)

# return x2

def f(x): # finding functional value of x0 and x1 through this function

return x\*\*3 -10\*x -6

x0 = float(input('first guess: ')) # taking input of first guess

x1 = float(input('second guess: ')) # taking input of second guess

e = float(input('tolerance: ')) # taking input of tolerance

if f(x0) \* f(x1) > 0.0: # if product of functional value of x0 and x1 is greater than 0

print('given guess values do not bracket the root')

else: # if product of functional value of x0 and x1 is less than 0

root = bisection(x0,x1,e) # call bisection function with x0,x1 and tolerance values, storing return value in root

**0utput:**

first guess: 2

second guess: 4

tolerance: 0.001

1 : 2 4 -18 18 3 -9

2 : 3 4 -9 18 3 1

3 : 3 3 -9 1 3 -4

4 : 3 3 -4 1 3 -1

5 : 3 3 -1 1 3 0

6 : 3 3 -1 0 3 0

7 : 3 3 0 0 3 0

8 : 3 3 0 0 3 0

9 : 3 3 0 0 3 0

10 : 3 3 0 0 3 0

11 : 3 3 0 0 3 0

12 : 3 3 0 0 3 0

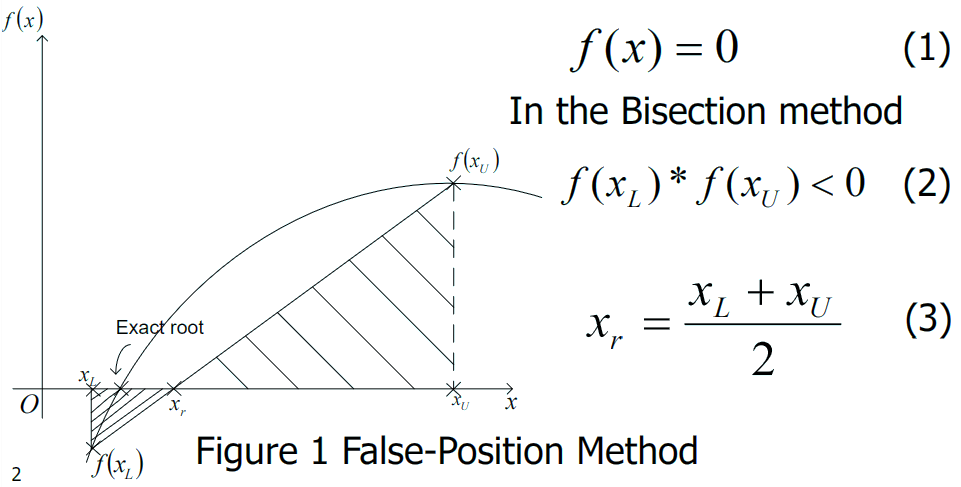
13 : 3 3 0 0 3 0

14 : 3 3 0 0 3 0

**root is :3.42785645**

**False Position Method:**

**Introduction:**



Based on two similar triangles, shown in Figure 1, one gets:

(4)

The signs for both sides of Eq. (4) are consistent, since:

From Eq. (4), one obtains

The above equation can be solved to obtain the next predicted root xr, as

(5)

The above equation,

(6)

Or

**Step-By-Step False-Position Algorithm:**

1. Choose xL and xU as two guesses for the root such that
2. Estimate the root,
3. Now check the following
   1. If , then the root lies between and ; then and
   2. If , then the root lies between and ; then and
   3. If , then the root is . Stop the algorithm if this is true.

4. Stop the new estimate of the root

Find the absolute relative approximate error

where

= estimated root from present iteration

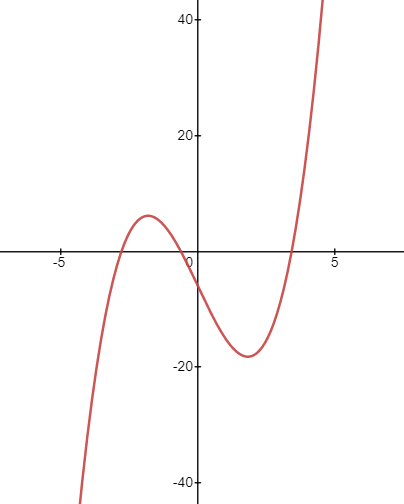
= estimated root from previous iteration

10-3 = 0.001.If then go to step 3, else stop the algorithm.

**Note:** The False-Position and Bisection algorithms are quite similar. The only difference is the formula used to calculate the new estimate of the root , shown in steps no 2 and 4.

**Example:**

**Using Regula Falsi method; Solve x3-10x-6 for roots between x=1 and x=4**

**Solution:**

|  |  |  |
| --- | --- | --- |
| X | 1 | 4 |
| f(x) | -15 | 18 |

Since f(2).f(4)<0 therefore root lies between 1 and 4

Using formula

For interval [1,4] we have

Which implies (-ve)

So, the next root lies between the interval [3,4]; because f(3).f(4)< 0

Similarly, other terms are given below

|  |  |  |
| --- | --- | --- |
| Interval | xr | f(xr) |
| [1,4] | 2.3636 | -16.431 |
| [2.3636,4] | 3.145 | -6.3520 |
| [3.145,4] | 3.3677 | -1.4806 |
| [3.3677,4] | 3.4158 | -0.3031 |
| [3.4158,4] | 3.4255 | -0.0603 |
| [3,4255,4] | 3.4274 | -0.0119 |
| [3.4274,4] | 3.4278 | -0.0024 |
| [3.4278,4] | 3.4278 | -0.00039 |
| [3,4278,4] | 3.4278 | -0.00039 |

So , the root is 3.4278

**Coding:**

from math import sin ,cos # importing from math library

def reg\_falsi(f,x1,x2,tol=1.0e-6,maxfpos=100): # receiving parameters

if f(x1) \* f(x2)<0: # if product of functional value of x1 and x2 is less than 0

for fpos in range(1,maxfpos+1): # loop from 1 to 101

# using the formula for regula falsi method

xh = x2 - (x2-x1)/(f(x2)-f(x1)) \* f(x2) # receiving root in xh

if abs(f(xh)) < tol: # if f(xh) is less than tolerance

break # break the loop and go to return statement

elif f(x1) \* f(xh) < 0: # if product of functional value of x1 and xh is less than 0

x2 = xh # if product is less than 0 then x2 = xh

else:

x1 = xh # if product is greater than 0 then x1 = xh

else: # if the root doesnt exist and if doesnt execute then this else executes

print('No roots exists within the given interval')

return xh, fpos # returning values

y = lambda x: x\*\*3 - 10\*x - 6 # this equation is defined in the form of function

x1 = float(input('enter x1: ')) # taking initial guess x1

x2 = float(input('enter x2: ')) # taking initial guess x2

r, n = reg\_falsi(y,x1,x2) # calling function and receiving in r and n

print('The root = %f at %d false position'%(r,n))

**Output:**

enter x1: 1

enter x2: 4

The root = 3.427879 at 12 false position

**Newton Raphson Method:**

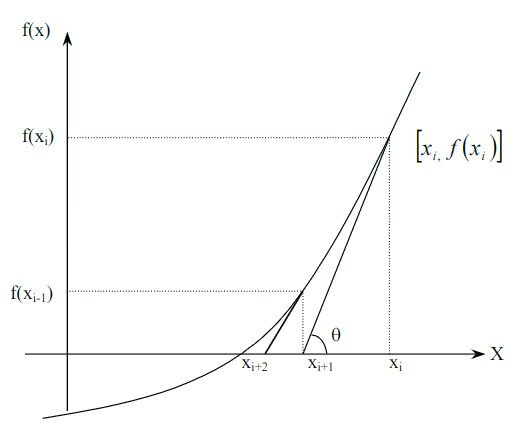


Figure 1 Geometrical illustration of Newton-Raphson method.

**Derivation:**

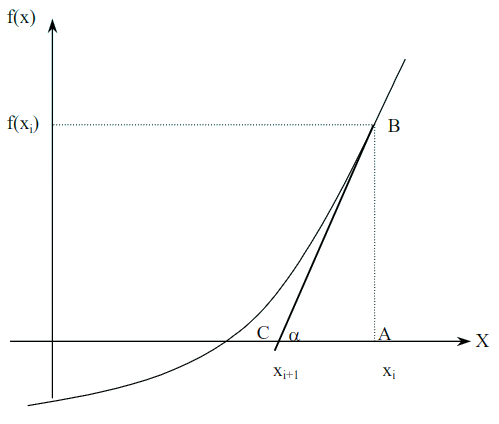


Figure 2 Derivation of Newton-Raphson method.

**Algorithm for Newton-Raphson Method:**

**Step 1:**

Evaluate symbolically.

**Step 2:**

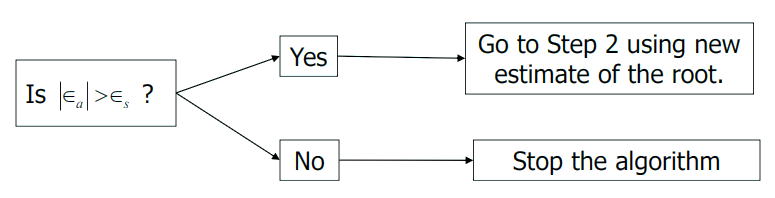
Use an initial guess of the root,, to estimate the new value of the root,, as

**Step 3:**

Find the absolute relative approximate error, as

**Step 4:**

Compare the absolute relative approximate error with the pre-specified relative error tolerance ,



Also, check if the number of iterations has exceeded the maximum number of iterations allowed. If so, one needs to terminate the algorithm and notify the user.

**Advantages and Drawbacks of Newton Raphson Method:**

**Advantages:**

* Converges fast (quadratic convergence), if it converges.
* Requires only one guess

**Drawbacks:**

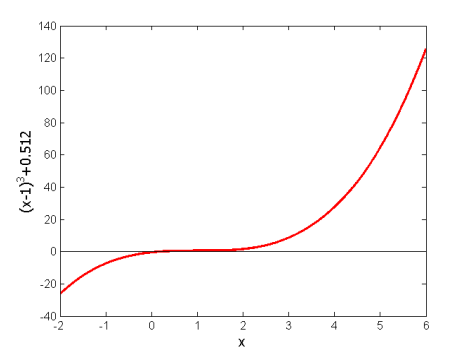
1. Divergence at inflection points

Selection of the initial guess or an iteration value of the root that is close to the inflection point of the function *f(x)* may start diverging away from the root in the Newton-Raphson method.

For example, to find the root of the equation f(x) = (*x*-1)3+0.512=0

The Newton-Raphson method reduces to Table 1 shows the iterated values of the root of the equation.

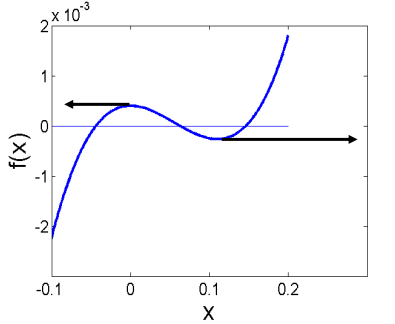
The root starts to diverge at Iteration 6 because the previous estimate of 0.92589 is close to the inflection point of *x* = 1.

Eventually after 12 more iterations the root converges to the exact value of *x* = 0.2.

|  |  |
| --- | --- |
| **Iteration Number** | **x*i*** |
| 0 | 5.0000 |
| 1 | 3.6560 |
| 2 | 2.7465 |
| 3 | 2.1084 |
| 4 | 1.6000 |
| 5 | 0.92589 |
| 6 | -30.119 |
| 7 | -19.746 |
| 18 | 0.2000 |

Figure 3 Divergence at inflection point for f(x) = (*x*-1)3+0.512=0

Table 1 Divergence near inflection point.



1. Division by zero

For the equation

The Newton-Raphson method reduces to

Figure 4 Pitfall of division by zero or near a zero number.

For *x*o = 0 or *x*o = 0.02, the denominator will equal zero.

3. Oscillation near local maximum and minimum

Results obtained from the Newton-Raphson method may oscillate about the local maximum or minimum without converging on a root but converging on the local maximum or minimum.

Eventually, it may lead to division by a number close to zero and may diverge.

For example for the equation has no real roots.

|  |  |  |  |
| --- | --- | --- | --- |
| **Iteration**  **Number** | ***x*i** | ***f*(*xi*)** |  |
| 0 | -1.0000 | 3.00 |  |
| 1 | 0.5 | 2.25 | 300.0 |
| 2 | -1.75 | 5.053 | 128.0 |
| 3 | -0.30357 | 2.092 | 476.47 |
| 4 | 3.1423 | 11.874 | 109.66 |
| 5 | 1.2529 | 3.570 | 150.80 |
| 6 | -0.17166 | 2.029 | 829.88 |
| 7 | 5.7395 | 34.942 | 102.99 |
| 8 | 2.6955 | 9.266 | 112.93 |
| 9 | 0.97678 | 2.954 | 175.96 |

Table 2 Oscillation near local maxima and minima in Newton-Raphson method.

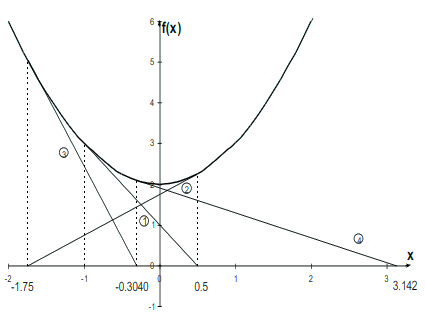
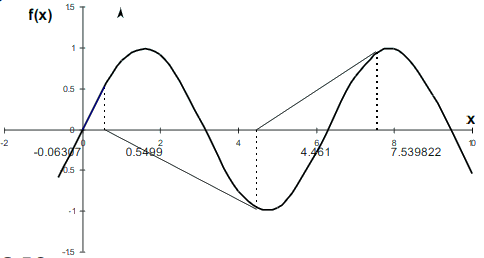
  
4. Root Jumping.

Figure 10 Oscillation around local minima for

In some cases where the function *f*(*x*) is oscillating and has a number of roots, one may choose an initial guess close to a root. However, the guesses may jump and converge to some other root.

For example

Choose

It will converge to instead of

**Example:**

Apply Newton’s Raphson method for correct upto three decimal places.

**Solution:** Given that

Using Formula; at

Similarly

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Itterations |  |  |  |  |
| 1 | 0.5 | -2.426 | -232.09 | 509.48 |
| 2 | -2.426 | -1.970 | -73.325 | 218.196 |
| 3 | -1.970 | -1.634 | -21.609 | 100.407 |
| 4 | -1.634 | -1.419 | -5.222 | 55.107 |
| 5 | -1.419 | -1.324 | -0.704 | 40.812 |
| 6 | -1.324 | -1.307 | -0.0196 | 38.524 |
| 7 | -1.307 | -1.306 | -0.000028 | 38.46 |
| 8 | -1.306 | -1.306 | -0.0000075 | 38.458 |

Hence root is **“-1.306”**

**Coding:**

from math import sin,cos # importing from math library

def newton(fn,dfn,x,tol,maxiter): # receiving parameters in function newton

for i in range(maxiter): # loop that run until maxiter

# using formula x - FN / F'x

xnew = x - fn(x)/dfn(x) # taking result in xnew

print('%d \t\t %0.6f \t\t %0.6f' %( i+1 , x , xnew )) # printing iteration, x2 and f(x2)

if abs(xnew-x)<tol: # checking if xnew-x is less than tolerance value

break # end the iterations

x = xnew # if xnew-x > tolerance , subtitute x with xnew

return xnew, i

y = lambda x: 3\*x\*\*5 + 2\*x\*\*2 +8 # original cubic equation

dy = lambda x: 15\*x\*\*4 + 4\*x # derivative of cubic equation

x, n = newton(y, dy,0.5, 0.000001, 50) # calling function and storing returning value in x and n

print('the root is %.3f at %d iterations.'%(x,n+1))

**Output:**

1 0.500000 -2.425532

2 -2.425532 -1.969985

3 -1.969985 -1.634039

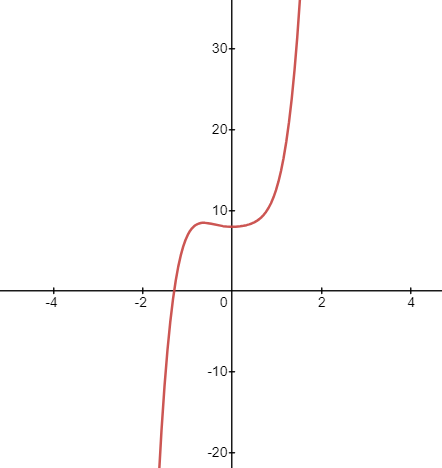
4 -1.634039 -1.418821

5 -1.418821 -1.324053

6 -1.324053 -1.306852

7 -1.306852 -1.306340

8 -1.306340 -1.306340

the root is -1.306 at 8 iterations. 

**Secant Method:**

**Derivation:**

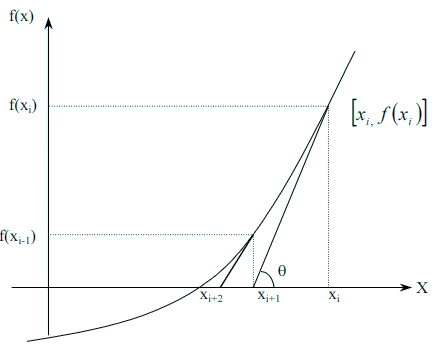
Newton’s Method

Figure 1 Geometrical illustration of the Newton-Raphson method.

Approximate the derivative

Substituting Equation (2) into Equation (1) gives the Secant method

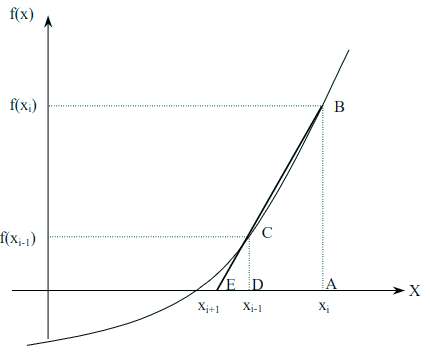
The secant method can also be derived from geometry:

Figure 2 Geometrical representation of the Secant Method.

The Geometric Similar Triangles

Can be written as

On rearranging, the secant method is given as

**Algorithm for Secant Method:**

**Step 1:**

Calculate the next estimate of the root from two initial guesses

Find the absolute relative approximate error

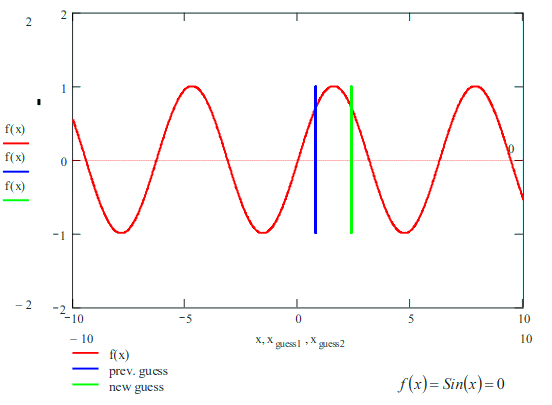
**Step 2:**

Find if the absolute relative approximate error is greater than the prespecified relative error tolerance.

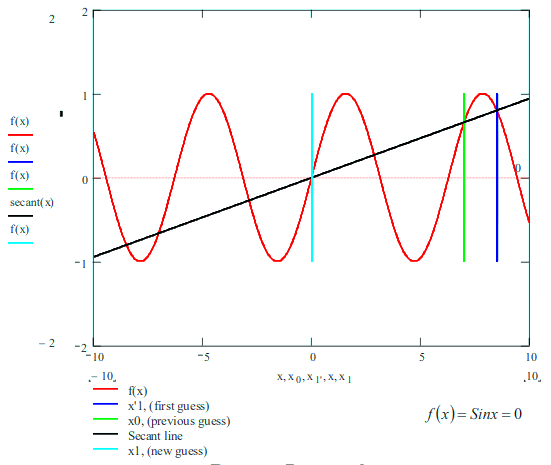
If so, go back to step 1, else stop the algorithm.

Also check if the number of iterations has exceeded the maximum number of iterations.

**Advantages:**

* Converges fast, if it converges.
* Requires two guesses that do not need to bracket the root.

Division by zero

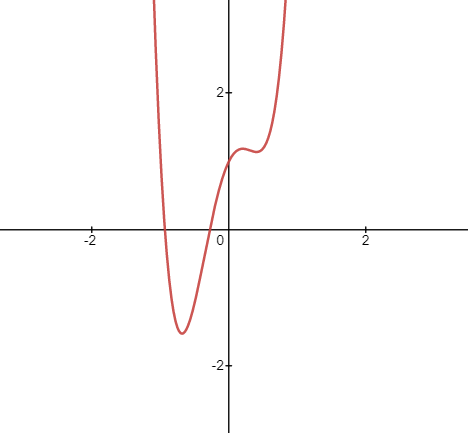
**Drawbacks:**

Root Jumping

**Example:**

Use Secant method to find the root of the function f(x) =

**Solution:**



**:** Given that

Using Formula;

at and

0.5275

And hence ,

Similarly

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Iterations |  |  |  |  |  |  |
| 1 | -2 | 0.5 | 0.5275 | 109 | 1.1875 | 1.229 |
| 2 | 0.5 | 0.52475 | -0.2999 | 1.1875 | 1.224 | -0.0937 |
| 3 | 0.52475 | -0.2999 | -0.2413 | 1.224 | --0.0937 | 0.1844 |
| 4 | -0.2999 | -0.2413 | -0.2801 | -0.0936 | 0.1844 | 0.00212 |
| 5 | -0.2413 | -0.2801 | -0.2806 | 0.1845 | 0.00248 | -0.0000612 |
| 6 | -0.2801 | -0.2806 | -0.2806 | 0.00248 | 0.000083 | -0.0000000232 |

Hence root is **“-0.2806”**

**Coding:**

from math import sin # importing from math library

def secant(fn,x1,x2,tol,maxiter): # receiving the parameters

for i in range(maxiter): # loop until maxiter(100)

# using formula for secant method

xnew = x2 - (x2-x1)/(fn(x2)-fn(x1))\*fn(x2) # storing root in xnew

print('\t%d \t\t %d \t\t %0.6f' %( x1,x2, xnew )) # printing iteration, x2 and f(x2)

if abs(xnew-x2) < tol: # if xnew-x2 is less than toleranc

break # then break the loop

else:

x1 = x2 # substituting value of x2 in

x2 = xnew # substituting value of xnew in x2

else:

print('warning: Maximum number of iterations is reached')

return xnew, i+1 # return the root and step number

f = lambda x: 9\*x\*\*4 +x\*\*3 -6\*x\*\*2 +2\*x+1 # this equation is defined in the form of function

x1 = float(input('enter x1: ')) # taking first value

x2 = float(input('enter x2: ')) # taking second value

r, n = secant(f,x1,x2,1.0e-5,10) # calling function and receive in r and n

print('Root = %f at %d iterations'%(r,n))

**Output:**

enter x1: -2

enter x2: 0.5

-2 0 0.527536

0 0 -0.285165

0 0 -0.270926

0 0 -0.280593

0 0 -0.280617

0 0 -0.280617

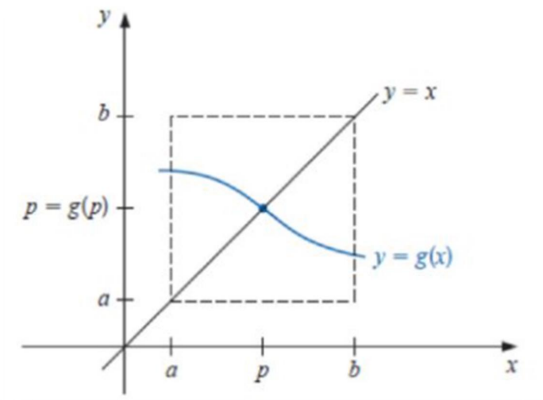
Root = -0.280617 at 6 iterations

**Fixed Point Iteration Method:**

The number is a **fixed point** for a given function if,

If fixed point iterations are (in same sense) equivalent to root finding, why just not stick to root finding?

1. Sometimes easier to analyze.
2. What we learn from the will help us find good root finding strategies.
3. Fixed point iterations pose some “cute” problems by themselves.



**is a fixed point** of

**is a fixed point** of

Root finding of

Finding **fixed point** of

**Simple fixed point iteration:**

**Step 1**

**Example 1**

Can be simply manipulated to yield

Rearranging the function

So that x is on the left-hand side of the equation:

**Example 2**

Can be simply manipulated to yield

**Step 2**

Given an initial guess at the root , (\*) can be used to compute a new estimate as expressed by the iterative formula

Note

Convert the root from root-finding to finding fixed-point.

**When does fixed-point iteration converge?**

**Existence and Uniqueness Theorem**

1. If and for all , then has a fixed-point in
2. If, in addition exists on (a,b) and a positive constant exists with

for all

Then there is **exactly one** fixed point in

**Note:**

1. is continuous in
2. takes values in

**Convergence:**

**Fixed-Point Theorem**

Let be such that , for all

Suppose, in addition, that exists on (a,b) and that a constant exists with

for all

Then, for any number in [a,b], the sequence defined by

Converges to the unique fixed point in [a,b].

**Algorithm**

(Fixed-Point Iteration) Let be a continuous function defined on the interval [a,b].

The following algorithm computes a number that is a solution to the equation

Choose an initial guess in [a,b].

**for**  **do**

**if**  is sufficiently small **then**

**return**

**end**

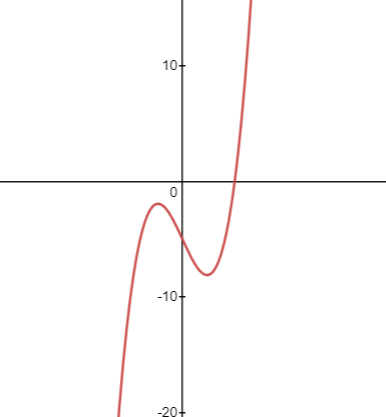
**end**

**Example:** Find the root of equation given as at the interval [2, 3] ; correct to three decimal points using fixed point iteration method.

**Solution:**

Given that

This can be written in the forms;



now taking the initial guess as the mid values from the interval

i.e; (2+3)/2 = 2.5 so,

putting value in the formula, we get

|  |  |  |
| --- | --- | --- |
| n |  |  |
| 1 | 2.5 | 2.466 |
| 2 | 2.466 | 2.458 |
| 3 | 2.458 | 2.4569 |
| 4 | 2.4569 | 2.4567 |
| 5 | 2.4567 | 2.45668 |
| 6 | 2.45668 | 2.45668 |

Hence the root is **2.45668**

**Coding:**

import numpy as np

from math import \*

def f(x):

return x\*\*3 - 4\*x -5

def g(x):

return 1/sqrt(1+x)

def fixedPointIteration(x, e, n):

step = 1

flag = 1

condition = True

while condition:

x1 = g(x)

print('Iteration : %d, x1 = %0.6f and f(x1) = %0.6f' % (step, x1, f(x1)))

x = x1

step = step + 1

if step > n:

flag=0

break

condition = abs(f(x1)) > e

if flag==1:

print('\nRequired root is: %0.8f' % x1)

else:

print('\nNot Convergent.')

x = float(input('Enter Guess : '))

e = float(input('Tolerable Error : '))

n = int(input('No of Steps : '))

fixedPointIteration(x,e,n)

**Output:**

Enter Guess : 2.5

Tolerable Error : 0.0001

No of Steps : 6

Iteration : 1, x1 = 0.534522 and f(x1) = -6.985369

Iteration : 2, x1 = 0.807260 and f(x1) = -7.702974

Iteration : 3, x1 = 0.743857 and f(x1) = -7.563836

Iteration : 4, x1 = 0.757259 and f(x1) = -7.594793

Iteration : 5, x1 = 0.754366 and f(x1) = -7.588178

Iteration : 6, x1 = 0.754988 and f(x1) = -7.589603

Not Convergent.

**CHAPTER 02:**

**Iterative Techniques in Matrix Algebra**

**NUMERICAL ITERATION METHOD**

It is a mathematical procedure that generates a sequence of improving approximate solution for a class of problems. i.e. the process of finding successive approximations.

**Norms of Vectors and Matrices:**

A Vector norm on ℝn is a function mapping ℝn ℝ with following properties:

1. for all x∈ℝn
2. if and only if x=0
3. for all and (scalar multiplication)
4. for all x,y ∈ℝn (triangle inequality)

**Common Norms:**

The I1 norm is given by

The I2 norm or Euclidean norm is given by

The norm or Max Norm is given by

**Convergence:**

A sequence of vectors in ℝn  is said to converge to x with respect to norm if given any ∈ > 0 there exists an integer N(∈) such that

**Matrix Norm:**

A matrix norm on the set of all n x n matrices is a real-valued function , defined on this set satisfying for all n x n matrices A and B and all real numbers α.

1. if and only if A is 0 (all zero entries)
2. (scalar multiplication)
3. (triangle inequality)

**Natural Matrix Norm:**

If is a vector norm on ℝn , then

is a matrix norm.

This is the natural or induced matrix norm associated with the vector norm.

**Matrix Mapping:**

An n x m matrix is a function that takes m-dimensional vectors into n-dimensional vectors.

For square matrices A, we have A: ℝn  ℝn

Certain vectors are parallel to Ax, so Ax = ⋋x or (Ax - ⋋I)x = 0.

These values ⋋, the eigenvalues, are significant for convergence of iterative methods.

**Eigenvalues and Eigenvectors:**

If A is an n x n matrix, the characteristics polynomial of A is defined by

If p is the characteristic polynomial of the matrix A, the zeroes of p are eigenvalues (or characteristics values) of A. If ⋋ is an eigenvalue of A and x≠0, then x is an eigenvector (or characteristic vector) of A corresponding to the eigenvalue ⋋.

**Spectral Radius:**

The spectral radius, ρ(A), provides a valuable measure of the eigenvalues, which helps to determine if a numerical scheme will converge.

The spectral radius, ρ(A), of a matrix A is defined by

ρ(A) = max ,

where ; ⋋ is an eigenvalue of A.

**Convergence of Matrix:**

An n *x* n matrix A is convergent if

**Direct method :**

* Guassian Elimination method

**Guassian elimination method:**

**Algorithm:**

• In the first stage, the given system of equations is reduced to an equivalent upper triangular form using elementary transformation.

• In the second stage, the upper triangular system is solved using back substitution

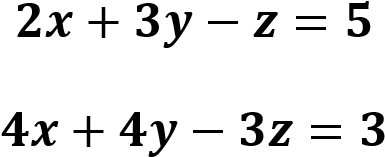
procedure by which we obtain the solution in the order

**Remark:**

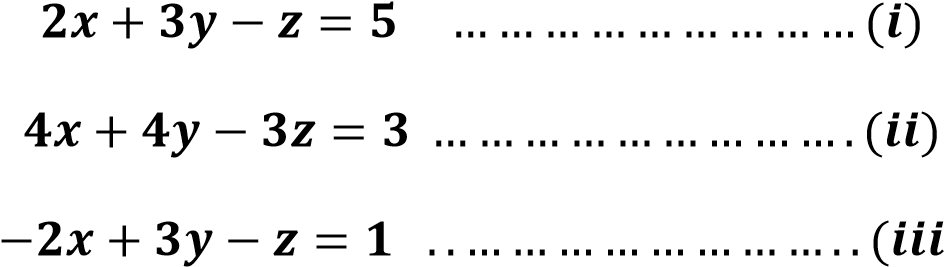
Guass’s Elimination method fails if any one of the Pivotal coefficient become zero. In such a situation, we rewrite the equation in a different order to avoid zero Pivotal coefficients.

**Example:**

**Solve the following system of equations using Elimination Method.**





**Solution:** given that;

Adding equation 1 and 3; we get

……………………………………..(iv)

Similarly multiplying equation (i) with 4 and subtracting it with equation (ii)

So we get;

Now by solving equation (iv) and (v) we get; (by multiplying equation (v) with 2 and subtracting it with equation (iv))

Now putting this value of y in equation (iv), we get the value of z

So ,

Now putting the value of y and z in equation (i); we get the value for x. so;

So we have the values as;

**Iterative Methods:**

* Jacobi iterative method
* Guass-Seidal iterative method
* Successive Over Relaxation method

**Jacobi Method:**

The Jacobi iterative method is obtained by solving the ith equation in Ax=b for xi to obtain (provided aii ≠0)

For each k≥1, generate the components of from the components of

The Jacobi method can be written in the form x(k)=Tx(k-1)+c by splitting A into its diagonal and off-diagonal parts. To see this, let D be the diagonal matrix whose diagonal entries are those of A, -L be the strictly lower-triangular part of A, and –U be the strictly upper-triangular part of A. with this notation,

Is split into,

**=**

The equation Ax=b , or ()x=b , is then transformed into

Dx=

and, if exists, that is , if for each I, then

b.

This results in the matrix form of the Jacobi iterative technique:

where k=1,2,….

So, introducing the notation and gives the Jacob technique the form

**Algorithm:**

* To solve Ax=b given an initial approximation x(0)

STEP 1: Set k=1.

STEP 2: While (k ≤ N) do steps 3-6

STEP 3: for i=1,…,n

Step 4 If ||x – **X0**|| < *TOL* then OUTPUT (*x1, ... , xn*);

(The procedure was successful.)

STOP.

Step 5 Set k = k + 1.

Step 6 For i = 1, ... , n set XOi = xi.

Step 7 OUTPUT (‘Maximum number of iterations exceeded’);

(The procedure was successful.)

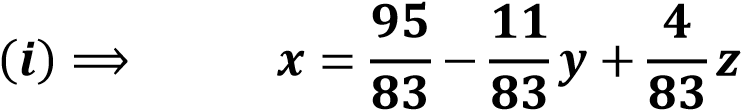
STOP.

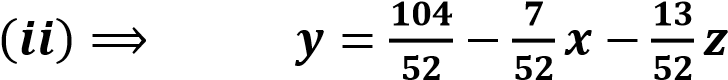
Step 3 of the algorithm requires that aii ≠ 0, for each i = 1, 2, ... , n. If one of the aii entries is 0 and the system is nonsingular, a reordering of the equations can be performed so that no aii = 0. To speed convergence, the equations should be arranged so that aii is as large as possible.

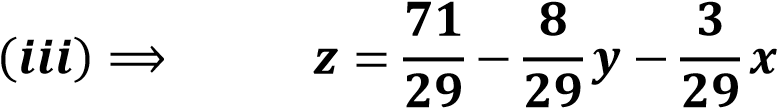
Another possible stopping criterion in Step 4 is to iterate until

is smaller than some prescribed tolerance. For this purpose, any convenient norm can be used, the usual being the l∞ norm.

**Example: : Find the solution of the system of equation using Jacobi iterative method for the first five iterations.**

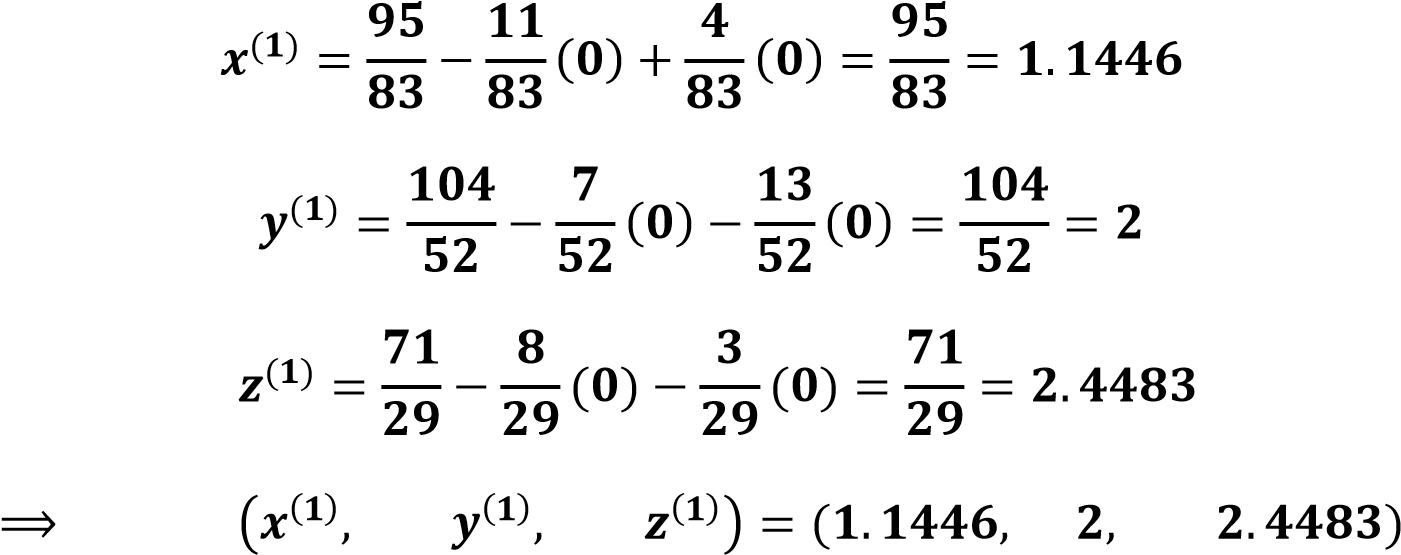




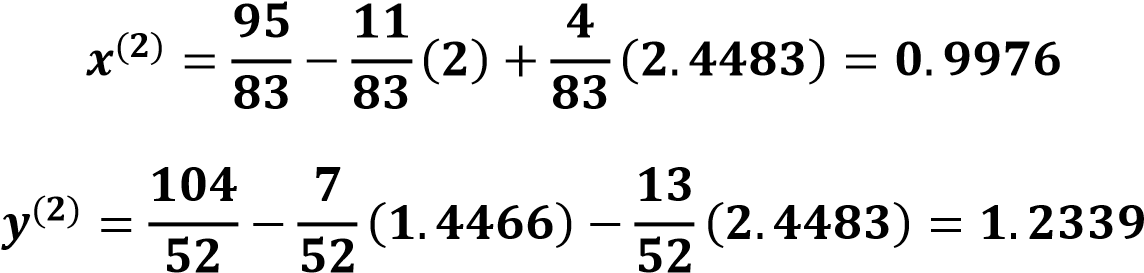


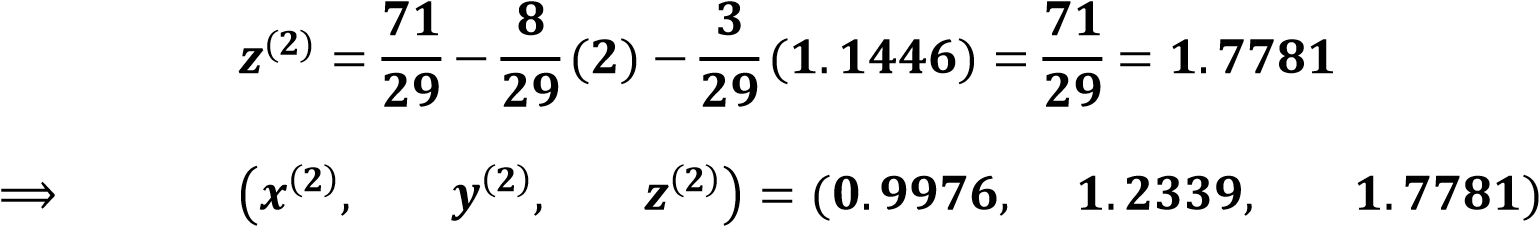
Taking initial guess as (0, 0, 0) and using formula

Put k = 0 for first iteration

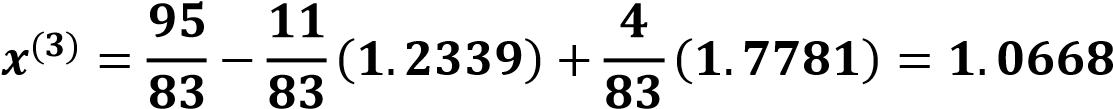


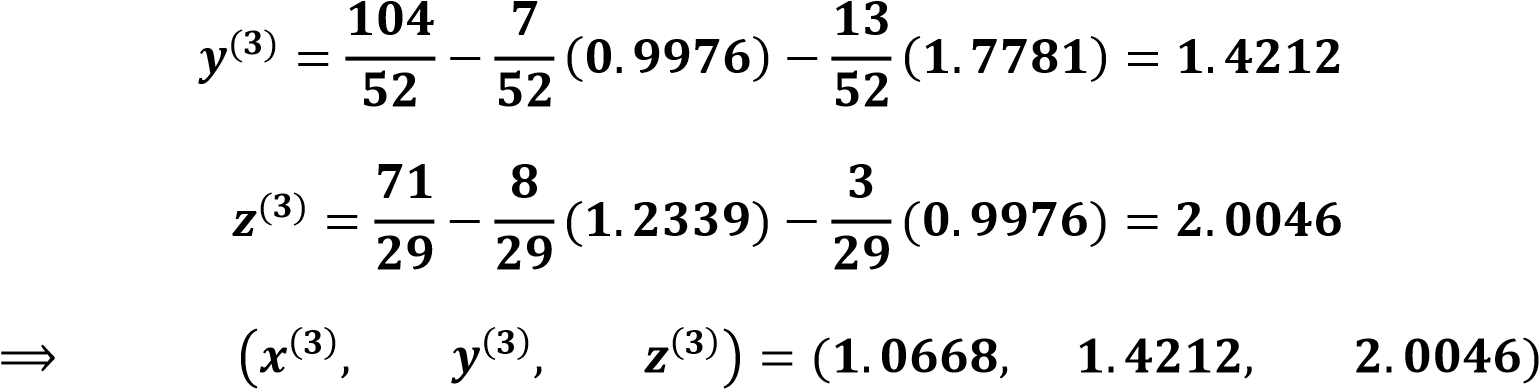
for second iteration



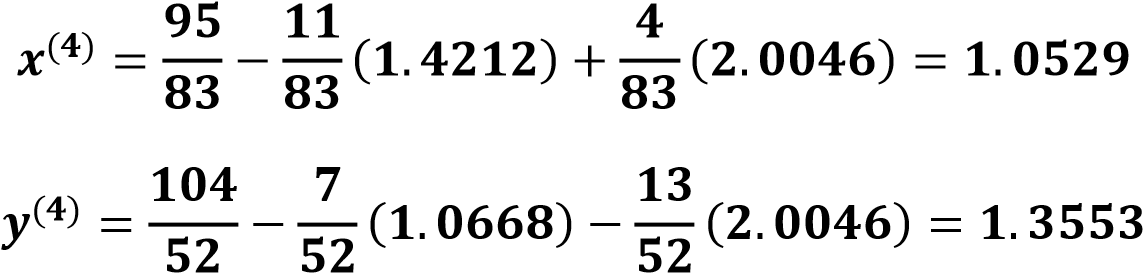


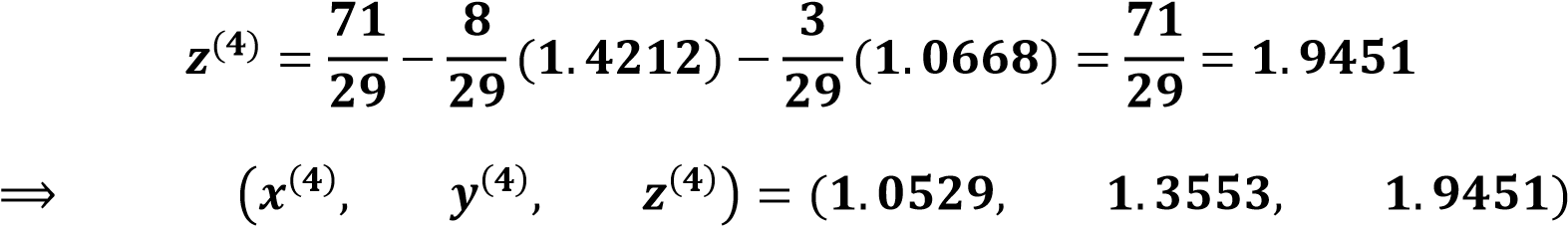
for third iteration



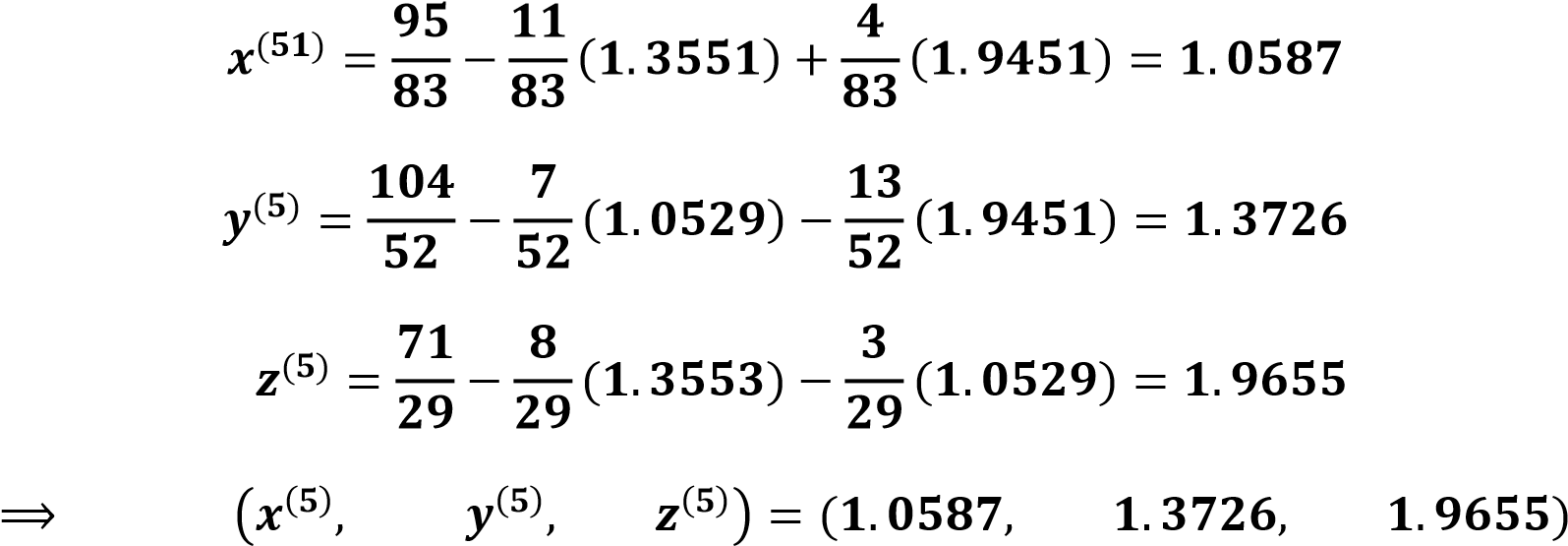


for fourth iteration





for fifth iteration



**Coding:**

from pprint import pprint

from numpy import array, zeros, diag, diagflat, dot

def jacobi(A,b,N=25,x=None):

"""Solves the equation Ax=b via the Jacobi iterative method."""

if x is None:

x = zeros(len(A[0]))

D = diag(A)

R = A - diagflat(D)

for i in range(N):

x = (b - dot(R,x)) / D

return x

A = array([[2.0,1.0],[5.0,7.0]])

b = array([11.0,13.0])

guess = array([1.0,1.0])

sol = jacobi(A,b,N=25,x=guess)

print ("A:")

pprint(A)

print ("B:")

pprint(b)

print ("X:")

pprint(sol)

**Output:**

A:

array([[2., 1.],

[5., 7.]])

B:

array([11., 13.])

X:

array([ 7.11110202, -3.22220342])

**Guass-Seidal iterative method:**

Jacobi method is modified so that the new approximations to the unknown are used as soon as they are available.

Formula of Jacobi method; for is then modified as follows:

Guass-Seidel method will always converge if Jacobi’s method converges and will do so more rapidly. If Jacobi’s method diverges, so does Guass-Seidel. Since the most recently computed values of x are used in the subsequent iterations, this makes the method more realistic and converges faster. This is generally the case but is not always true. In fact, there are linear systems for which Jacobi’s method converges and the Guass-Seidel method does not and conversely. On the whole, we can say that Guass-Seidel method converges faster than Jacobi’s method.

As already mentioned that the solutions are quickly reached if a11,a22,…,ann are numerically larger compared with other coefficients. If necessary, the equations are rearranged so that the bigger coefficients are on the main diagonal.

**Algorithm:**

STEP 1: Choose the initial guess.

STEP 2: Set r=0

STEP 3: For each i=1, 2, 3, …, n, compute

STEP 4: If the solution vector

1. x(r+1) is sufficiently accurate, go to STEP 5,
2. Otherwise, add 1 to r and go to STEP 3.

STEP 5: Stop the process. Stopping criteria are the same as mentioned under Jacobi’s method.

**Example:**

**QUESTION: Find the solutions of the following system of equations using Guass Seidel method and perform the first five iterations.**

6x+4y-z=4

-2x+7y+3z=10

x+y+5z=9

**Solution**

= (4-4yk+zk)

= (10+2xk+1-3zk)

= (9-xk+1-yk+1)

Initial guesses are (0, 0, 0)

1st iteration

= (4-0-0) =0.6667

= (10+2(0.6667)-0)=1.619

= (9-(0.6667)-(1.619)) =1.3429

|  |  |  |  |
| --- | --- | --- | --- |
| Iteration | x | Y | z |
| 1 | 0.6667 | 1.619 | 1.3429 |
| 2 | -0.1889 | 0.7991 | 1.678 |
| 3 | 0.4136 | 0.8276 | 1.5518 |
| 4 | 0.3735 | 0.8703 | 1.5512 |
| 5 | 0.345 | 0.8623 | 1.5585 |
| 6 | 0.3515 | 0.8611 | 1.5575 |
| 7 | 0.3522 | 0.8617 | 1.5572 |

From table we can see that after seven iterations ; the answer is accurate upto 2 decimal places.

So the solution by gauss seidel method is:

**Coding:**

f1 = lambda x,y,z: (4-4\*y+z)/4

f2 = lambda x,y,z: (10+2\*x-3\*z)/7

f3 = lambda x,y,z: (9-x-y)/5

x0 = 0

y0 = 0

z0 = 0

count = 1

e = float(input('Enter tolerable error: '))

print('\nCount\tx\ty\tz\n')

condition = True

while condition:

x1 = f1(x0,y0,z0)

y1 = f2(x1,y0,z0)

z1 = f3(x1,y1,z0)

print('%d\t%0.4f\t%0.4f\t%0.4f\n' %(count, x1,y1,z1))

e1 = abs(x0-x1);

e2 = abs(y0-y1);

e3 = abs(z0-z1);

count += 1

x0 = x1

y0 = y1

z0 = z1

condition = e1>e and e2>e and e3>e

print('\nSolution : X=%0.3f, Y=%0.3f and Z= %0.3f\n'% (x1,y1,z1))

**Successive Over Relaxation method:**

**Derivation:**

Here ω is the relaxation reimeter.

1. If then this method will reduce to Guass-Seidel method.
2. If then this method is called Over Relaxation.
3. If then this method is called Under Relaxation.
4. If lies between then the method converges otherwise diverges.
5. We can find by the formula:

and p(tj)=D-1(L+U)

**Proof:**

We prove this formula form Guass-Seidel Iterative formula. As generalized form of Guass-Seidel is:

Now adding and subtracting on both sides of equation

Replace “1” by “ω” in above equation we get

this equation is the required equation of successive over relaxation (SOR) method

if we expand the “∑” and put the values of i=1,2,3,…,n then we get the equations

**solve the following system of equations using SOR method.**

**6*x*+4y-z=4  
-2x+7y+3z=10  
*x*+y+5z=9**From the above equations, First write down the equations for Gauss Seidel method

= (4-4yk+zk)

= (10+2xk+1-3zk)

= (9-xk+1-yk+1)

Now multiply the right hand side by the parameter *w* and add to it the vector *xk* from the previous iteration multiplied by the factor of (1-*w*)

=(1-w) (4-4yk+zk)

=( 1-w)(10+2xk+1-3zk)

=(1-w) (9-xk+1-yk+1)  
 Initial gauss (*x*,*y*,*z*)=(0,0,0) and *w*=1.25Solution steps are  
1*st* Approximation  
*x*1=(1-1.25)⋅0+1.25⋅[4+(0)-(0)]= 0.8333   
*y*1=(1-1.25)⋅0+1.25⋅[10+2(0.8333)+(0)]=2.08333   
*z*1=(1-1.25)⋅0+1.25⋅[9-(0.8333)-(2.08333)]=1.5208

|  |  |  |  |
| --- | --- | --- | --- |
| ITTERATION | X | Y | Z |
| 1 | 0.8333 | 2.0833 | 1.5208 |
| 2 | -0.7943 | 0.1665 | 2.2067 |
| 3 | 1.3194 | 1.1281 | 1.1324 |
| 4 | -0.1997 | 0.8257 | 1.8108 |
| 5 | 0.5723 | 0.8138 | 1.4509 |
| 6 | 0.3143 | 0.9173 | 1.5794 |
| 7 | 0.3194 | 0.8244 | 1.5692 |
| 8 | 0.3934 | 0.8795 | 1.5395 |
| 9 | 0.3228 | 0.8564 | 1.5703 |
| 10 | 0.3661 | 0.8611 | 1.5506 |
| 11 | 0.3473 | 0.8638 | 1.5596 |
| 12 | 0.3516 | 0.8599 | 1.5572 |
| 13 | 0.3533 | 0.8627 | 1.5567 |
| 14 | 0.3504 | 0.8612 | 1.5579 |
| 15 | 0.3526 | 0.8617 | 1.5569 |

From table we can see that after fifteen iterations ; the answer is accurate upto 2 decimal places.

So the solution by SOR method is:

**Coding**:

f1 = lambda x,y,z: (4-4\*y+z)/4

f2 = lambda x,y,z: (10+2\*x-3\*z)/7

f3 = lambda x,y,z: (9-x-y)/5

x0 = 0

y0 = 0

z0 = 0

count = 1

e = float(input('Enter tolerable error: '))

w = float(input("Enter relaxation factor: "))

print('\nCount\tx\ty\tz\n')

condition = True

while condition:

x1 = (1-w) \* x0 + w \* f1(x0,y0,z0)

y1 = (1-w) \* y0 + w \* f2(x1,y0,z0)

z1 = (1-w) \* z0 + w \* f3(x1,y1,z0)

print('%d\t%0.4f\t%0.4f\t%0.4f\n' %(count, x1,y1,z1))

e1 = abs(x0-x1);

e2 = abs(y0-y1);

e3 = abs(z0-z1);

count += 1

x0 = x1

y0 = y1

z0 = z1

condition = e1>e and e2>e and e3>e

print('\nSolution: x = %0.3f, y = %0.3f and z = %0.3f\n'% (x1,y1,z1))

**Chapter 03**

**Finite differences**

**DIFFERENCE TABLE:** Suppose we have a function f(x) which is tabulated over a range of values of x and let h be the uniform difference between any two successive values

h = x1-x0 = x2- x1 = xn -xn-1

or x1 = x0 + h

x2 = x1 + h = x0 + h + h = x0 + 2h

…

Xp = x 0+ ph

…

Xn = x 0+ nh

And f (Xp ) = fp = f (x0 +ph)

In many numerical processes concerned with tabulated function certain quantities called finite differences. The word finite referred to the finite size of the interval use in the table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Xi | fi | 1st order | 2nd order | 3rd order | 4th order |
| X0 | f0 |  |  |  |  |
|  |  | f1 -f0 |  |  |  |
| X1 | f1 |  | f2 -2f1 +f0 |  |  |
|  |  | f2 -f1 |  | f3-3f2 +3f1-f0 |  |
| X2 | f2 |  | f3 -2f2 +f1 |  | f4-4f3+6f2-4f1-f0 |
|  |  | f3 -f2 |  | f4-3f3 +3f2-f1 |  |
| X3 | f3 |  | F4 -2f3 +f2 |  |  |
|  |  | f4 -f3 |  |  |  |
| X4 | f4 |  |  |  |  |

The standard format of displaying finite differences is called difference table.

3rd , 4th 5th  and 6th columns are first second third and fourth order differences.

**Important points:**

* For a constant function all differences are zero.
* It helps in determining the behavior of the derivative of a given function.
* It plays an important role in interpolation, numerical differentiation, numerical integration, numerical solution of ordinary and partial differential equations.
* The nth- difference of an exact polynomial of degree n are constant.
* The (n+1) st differences of that polynomial are zero.
* The above values are only true of polynomial when they are tabulated at equal intervals.
* If the function does not represent an exact polynomial, the above 3 points will not hold.

**Example:** construct the difference table for the function f(x)=x2 for x=-2 to x=2 , at interval of 1.

Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Xi | fi= xi2 | 1st order | 2nd order | 3rd order |
| -2 | 4 |  |  |  |
|  |  | -3 |  |  |
| -1 | 1 |  | 2 |  |
|  |  | -1 |  | 0 |
| 0 | 0 |  | 2 |  |
|  |  | 1 |  | 0 |
| 1 | 1 |  | 2 |  |
|  |  | 3 |  |  |
| 2 | 4 |  |  |  |

**Difference operator:**

To refer to specific entries in a difference table we use some operators, called difference operators.

The following operators are commonly used:

∆ forward difference operator (commonly read as delta)

∇ backward difference operator (commonly read as nabla)

µ average operator (commonly read as mu)

𝛿 central difference operator (commonly read as sigma)

E shift operator

**Forward Difference Operator:**

The difference operator ∆ is defined by the following relation:

∆fr=fr+1-fr

Where r is an integer, and ∆fr=∆f (xr )

Also, fr+1 = ∆f(xr+h) and ∆fr+1/2 = ∆f(xr + )

Nth order difference is given by,

∆nfr=∆n-1fr+1-∆n-1fr

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Xi | fi | ∆f | ∆2f | ∆3f | ∆4f |
| X0 | f0 |  |  |  |  |
|  |  | ∆ f0 |  |  |  |
| X1 | f1 |  | ∆2 f0 |  |  |
|  |  | ∆ f1 |  | ∆3 f0 |  |
| X2 | f2 |  | ∆2 f1 |  | ∆4 f0 |
|  |  | ∆ f2 |  | ∆3 f1 |  |
| X3 | f3 |  | ∆2 f2 |  |  |
|  |  | ∆ f3 |  |  |  |
| X4 | f4 |  |  |  |  |

**forward difference table:**

**Backward Difference Operator:**

The difference operator ∇ is defined by the following relation:

∇ fr=fr-fr-1

Nth order difference is given by,

∇nfr=∇n-1fr-∇n-1fr-1  ; for n>=1

**backward difference table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Xi | fi | ∇f | ∇2f | ∇ 3f | ∇ 4f |
| X0 | f0 |  |  |  |  |
|  |  | ∇f1 |  |  |  |
| X1 | f1 |  | ∇ 2 f1 |  |  |
|  |  | ∇f2 |  | ∇ 3 f1 |  |
| X2 | f2 |  | ∇ 2 f2 |  | ∇ 4 f1 |
|  |  | ∇f3 |  | ∇ 3 f2 |  |
| X3 | f3 |  | ∇ 2 f3 |  |  |
|  |  | ∇f4 |  |  |  |
| X4 | f4 |  |  |  |  |

**Central Difference Operator:**

The difference operator 𝛿 is defined by the following relation:

𝛿fr=fr+1/2 -fr-1/2

Nth order difference is given by,

𝛿nfr=𝛿n-1 fr+1/2 -𝛿n-1 fr-1/2

**central difference table:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Xi | fi | 𝛿f | 𝛿2f | 𝛿 3f | 𝛿 4f |
| X0 | f0 |  |  |  |  |
|  |  | 𝛿 f |  |  |  |
| X1 | f1 |  | 𝛿2 f1 |  |  |
|  |  | 𝛿f |  | 𝛿 3 f |  |
| X2 | f2 |  | 𝛿 2 f2 |  | 𝛿 4 f2 |
|  |  | 𝛿 f |  | 𝛿 3 f |  |
| X3 | f3 |  | 𝛿 2 f3 |  |  |
|  |  | 𝛿f |  |  |  |
| X4 | f4 |  |  |  |  |

**Shift operator (step operator):**

Efr=fr+1

E-1fr=fr-1

E2fr=fr+2

In general, Enfr=fr+n

Mean operator:

µfr=(fr+ +fr-)

important relationship between operator:

∆fr=fr+1-fr

∆fr=Efr-fr=(E-1) fr

∆fr=(E-1) fr

∇ fr=fr- E-1 fr

∇ fr=(1- E-1) fr

Therefore,

∆=E-1

∇=1- E-1

Likewise, = E1/2-E-1/2

µ = E1/2+E-1/2

**prove that: ∆-**∇ =∆∇

**L.H. S= ∆-**∇ = E-1-1+E-1

= E-2+E-1

= E-1-E

**R.H. S=** ∆∇

=(E-1) (1-E-1)

**=** E-EE-1-1+ E-1

=E-2+ E-1

L.H. S=R.H.S

**prove that**

**R.H. S=**

**= ∆2-**∇2/∆∇

= (**∆-**∇)(**∆+**∇)/∆∇

= ∆∇ (**∆+**∇) / ∆∇

=**∆+**∇ using above prove

= R.H.S

**Interpolation**

For a given table of values(xi,yi) for all i=0,1,2,…,N the process of estimating the values of “y=f(x)” for any intermediate values of “x = g(x)” is called “interpolation”.

**Polynomial Interpolation**

If g(x) is a Polynomial, Then the process is called “Polynomial” Interpolation. Polynomial Interpolation is the simplest and the most common type of interpolation. One feature of it is that there’s always a unique polynomial of degree at most n-1 passing through *n* data points.

**Choice of suitable interpolation formula:**

* Whether the given point xi are equally or unequally spaced
* Whether the interpolation is desired towards the beginning, center end of the difference table

**Interpolation for equally spaced data points**

1. Newton forward difference interpolation formula
2. Newton backward difference interpolation formula
3. Central difference interpolation formula

**Interpolation for unequally spaced data points**

1. Lagrange formula
2. Newton divided difference interpolation formula

Now we will describe all formulas one by one,

**Newton forward difference interpolation formula**

* This formula is used for interpolating the values of y near the beginning of a set of tabulated values, it may also be applicable in other parts by suitably shifting the origin
* Values of ‘x’ must have equal distance i.e., equally spaced

Let y=f(x) x0=f(x0)= f0 and  Xn = x 0+ nh xp = x0 +ph

P= where h = Xn- Xn-1

And fp= f(x0 +ph) = Epf0=(1+∆)pf0

Expanding (1+∆)p by binomial expansion

Fp= {1+p∆+p(p-1) ∆2+p(p-1) (p-2) ∆3+…p(p-1)(p-2)…(p-n+1) ∆n}f0

Fp= f0+p∆ f0+p(p-1) ∆2 f0+p(p-1) (p-2) ∆3 f0+…p(p-1)(p-2)…(p-n+1) ∆nf0

**Remarks:**

This formula is usually applicable for 0<p<1

The first two terms of this formula give the linear interpolation while the first three terms give a parabolic interpolation and so on…

**Example: (a)** **compute the difference table for the following set of data point**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | **1** | **2** | **3** | **4** |
| **F(x)** | **7** | **11** | **28** | **63** |

**(b) use newton forward difference formula to find 3rd degree polynomial**

**(c) use the above formula to interpolate for f (1.25)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Xi | fi | ∆f | ∆2f | ∆3f |
| 1 | 7 |  |  |  |
|  |  | 4 |  |  |
| 2 | 11 |  | 13 |  |
|  |  | 17 |  | 5 |
| 3 | 28 |  | 18 |  |
|  |  | 35 |  |  |
| 4 | 63 |  |  |  |

1. The forward differences are computed as follow
2. h = X1- X0 = 2-1 = 1

P= = = 0.25

Since the value of p is within the range 0 to 1 it makes the forward difference formula applicable.

Fp= f0+p∆ f0+p(p-1) ∆2 f0+p(p-1) (p-2) ∆3 f0

= 7 + p (4) + +

**=** 0.833 P3 + P2 (6.5-2.5) +P (4-6.5+1.67) + 7

**=**0.833 P3 +4P2 -0.83P+ 7

1. Inserting p=0.25 in above polynomial, we get

Fp= 0.833 (0.25)3 +4(0.25)2 -0.83(0.25) + 7

=7.055

# Algorithm for forward difference table:

# Program:

# Calculating u mentioned in the formula

def u\_cal(u, n):

temp = u;

for i in range(1, n):

temp = temp \* (u - i);

return temp;

# calculating factorial of given number n

def fact(n):

f = 1;

for i in range(2, n + 1):

f \*= i;

return f;

# Driver Code

# Number of values given

n = 4;

x = [ 1, 2,3 , 4];

# y[][] is used for difference table

# with y[][0] used for input

y = [[0 for i in range(n)]

for j in range(n)];

y[0][0] = 7;

y[1][0] = 11;

y[2][0] = 28;

y[3][0] = 63;

# Calculating the forward difference

# table

for i in range(1, n):

for j in range(n - i):

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

# Displaying the forward difference table

for i in range(n):

print(x[i], end = "\t");

for j in range(n - i):

print(y[i][j], end = "\t");

print("");

# Value to interpolate at

value = 1.25;

# initializing u and sum

sum = y[0][0];

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

for i in range(1,n):

sum = sum + (u\_cal(u, i) \* y[0][i]) / fact(i);

print("\nValue at", value, "is", round(sum, 6));

output :

1 7 4 13 5

2 11 17 18

3 28 35

4 63

Value at 1.25 is 7.054688

**Newton backward difference interpolation formula**

* This formula is used for interpolating the values of y near the end of a set of tabulated values, it may also be applicable in other parts by suitably shifting the origin
* Values of ‘x’ must have equal distance i.e., equally spaced

P= where h = Xn- Xn-1

And fp= f(x0 +ph) = Epf0=(1-∇)-pf0

Expanding (1+∆) p by binomial expansion

Fp= {1+p∇ +p(p+1) ∇ 2+p(p+1) (p+2) ∇ 3+…p(p+1) (p+2) …(p+n-1) ∇ n} f0

Fp= f0+ p∇ f0 +p(p+1) ∇ 2 f0+p(p+1) (p+2) ∇ 3 f0+…p(p+1) (p+2) …(p+n-1) ∇ nf0

**Example: (a)** **compute the difference table for the following set of data point**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | **1** | **2** | **3** | **4** |
| **F(x)** | **7** | **11** | **28** | **63** |

**(b) use newton backward difference formula to find 3rd degree polynomial**

**(c) use the above formula to interpolate for f (3.35)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Xi | fi | ∇f | ∇2f | ∇3f |
| 1 | 7 |  |  |  |
|  |  | 4 |  |  |
| 2 | 11 |  | 13 |  |
|  |  | 17 |  | 5 |
| 3 | 28 |  | 18 |  |
|  |  | 35 |  |  |
| 4 | 63 |  |  |  |

1. The forward differences are computed as follow
2. h = X1- X0 = 2-1 = 1

P= = = 2.35

Since the value of p is not within the range 0 to 1 so we cannot use 1 as origin therefore we have to shift origin to 3 then,

P= = 0.35

Since the value of p is within the range 0 to 1 so we can use x0 = 3 as origin so it makes the backward difference formula applicable

Fp= f0+ p∇ f0 +p(p+1) ∇ 2 f0

= 28 + p (17) +

**=** 6.5P2+ (17+6.5) P+28

**=**6.5P2+ 23.5P+28

1. Inserting p=0.35 in above polynomial, we get

Fp= 6.5(0.35)2 +23.5(0.35) + 28

=36.953

# Program:

# calculating u mentioned in the formula

def u\_cal(u, n):

temp = u;

for i in range(1, n):

temp = temp \* (u +i);

return temp;

# calculating factorial of given number n

def fact(n):

f = 1;

for i in range(2, n + 1):

f \*= i;

return f;

# Driver Code

# Number of values given

n = 4;

x = [ 1, 2,3 , 4];

# y[][] is used for difference table

# with y[][0] used for input

y = [[0 for i in range(n)]

for j in range(n)];

y[0][0] = 7;

y[1][0] = 11;

y[2][0] = 28;

y[3][0] = 63;

# Calculating the forward difference

# table

for i in range(1, n):

for j in rangen-1,i-2,-1):

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

# Displaying the forward difference table

for i in range(n):

print(x[i], end = "\t");

for j in range(i):

print(y[i][j], end = "\t");

print("");

# Value to interpolate at

value = 3.35;

# initializing u and sum

sum = y[n-1][0];

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

for i in range(1,n):

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

print("\nValue at", value, "is", round(sum, 6));

1

2 11

3 28 17

4 63 35 18

Value at 3.35 is 37.946562

In [ ]:

**Center interpolation method:**

The two formulas that we mentioned above are most occasionally used for the beginning and at the end of the difference table. More important are formulas which make use of central differences. Now we will discuss some well-known central differences formulas.

Stirling interpolation formula:

x-1

x0 f0 𝛿2f0 𝛿4f0

x1

it is expressed as follows

fp= f0 +  p( ) +  P2𝛿2f0 + ( + + 𝛿4f0 + (+ ) + 𝛿6f0 + …

**remark:** this formula is suitable for small value of p, for example, -0.25≤p≤0.25

**Bessel interpolation method:**

Bessel formula follows the path through the difference table:

X0 f0  𝛿2f0  𝛿4f0

X1 f1 𝛿2f1 𝛿4f1

Bessel formula can be expressed as follows

fp= f0 + p +( + + + ( + +…

**remark:** this formula is suitable for small values of p not far from 0.5, for example, 0.25≤p≤0.75

**Everett’s interpolation formula**

Everett’s formula follows the path through the difference table:

X0 f0  …

X1 f1  …

Everett’s formula can be expressed as follows

fp= qf0 + 𝛿2f0 + 𝛿4f0 +…

q=1-p

+ pf1 + 𝛿2f1 + 𝛿4f1 +…

Everett’s formula is simple and fast and generally most useful.

**Gaussian interpolation formula**

* Gauss forward interpolation formula
* Gauss backward interpolation formula

**Gauss forward interpolation formula**

Gauss forward interpolation formula follows the path through the difference table:

x0 f0 𝛿2f0 𝛿4f0

X1 f1

Gauss difference formula can be expressed as

fp= f0 + p + + + (

**Gauss backward interpolation formula**

Gauss backward interpolation formula follows the path through the difference table:

x-1 f-1

x0 f0 𝛿2f0 𝛿4f0

Gauss backward formula can be expressed as:

fp= f0 + p + + + (

**remark:** Gaussian interpolation formulas are of interest almost exclusively from theoretical standpoint.

**Example: (a)** **compute the difference table up to 3rd order only for the following set of data point**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | **1** | **2** | **3** | **4** |
| **F(x)** | **7** | **11** | **28** | **63** |

**(b) interpolate f (3.2) using the following formulas centered at x= 3**

(I) Stirling (ii) Bessel (iii) Everett and (iv) gaussian forward and backward

(a) difference table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Xi | fi | 𝛿f | 𝛿2f | 𝛿 3f |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 7 |  |  |  |
|  |  | 4 |  |  |
| 2 | 11 |  | 13 |  |
|  |  | 17 |  | 5 |
| X = 3 | 28 |  | 18 |  |
|  |  | 35 |  | -6 |
| 4 | 63 |  | 12 |  |
|  |  | 47 |  |  |
| 5 | 110 |  |  |  |

1. **Stirling**

xp = 2.2 x0 = 2 h = 1

P= = = 0.2

fp= f0 +  p( ) +  P2𝛿2f0 + ( +

fp= 28 + (0.2) ( ) +  (0.2)2(18) + (

= 34.656

**(ii) Bessel**

fp= f0 + p +( + +

fp= 28 + (0.2) (35) + (18+22) +

= 48.352

1. **Everett**

q=1-p=1-0.2 =0.8

fp= qf0 + 𝛿2f0 + pf1 + 𝛿2f1

= (0.8) (28) + (18)+(0.2)(63)+ (12)

=33.752

**(iv) gaussian forward and backward**

**Forward** fp= f0 + p + +

= 28 + (0.2) (35) + (18) + (-6)

= 33.752

**Backward**  fp= f0 + p + +

= 28 + (0.2) (17) + (18) + (5)

= 33.4

**Lagrange interpolation formula:**

Difference table could be used for interpolation, but this was restricted to equidistant points. Now we will find nth degree polynomial f(X) form the set of data x0, x1, x2,...,xn that may or may not be equidistant using Lagrange formula .

Lagrange formula can be derived by writing

F(x)= A0 (x-x1) (x-x2) …(x-xn)

+A1 (x-x0) (x-x2) …(x-xn)

+A2 (x-x0) (x-x1) …(x-xn)

A0, A1,..,An are unknown constants

.

.

.

+ An (x-x0) (x-x1) …(x-xn-1) … (4.1)

If we substitute x=x0 in above equation (4.1)

We get F(x0) = A0 (x0-x1) (x0-x2) …(x0-xn) + 0+0 +…+0

A0 =

If we substitute x=x1, we will have

F(x1) = A1 (x1-x0) (x1-x2) …(x1-xn)

A1 =

Similarly, If we substitute x=xn ,we will have

An =

Now substituting A0, A1,,An in equation (4.1) we have

F(x)= (x-x1) (x-x2) …(x-xn)

+ (x-x0) (x-x2) …(x-xn)

+ …

+ (x-x0) (x-x1) …(x-xn-1) … (4.2)

Now let Li(x)= j=0, j≠i … (4.3)

Then equation 4.2 becomes

F(x)= L0(x0) + L1(x1) +…+ Ln(xn)

It can be written as

F(x)= … (4.4)

Using equation (4.3) in (4.4) we have

F(x)= jig

**PROS AND CONS OF LAGRANGE’S POLYNOMIAL**

* Elegant formula (+)
* Applicable for Equally and unequally space grid points
* Fail when difference of point is too small because in denominator of Li(x)= we will get zero.
* difficult to compute by hand if data point is more, each (Li) is different
* Not flexible; if one changes a point xj, or add an additional point xn+1 one must re-compute all (Li).

**Example:** **(a) Use the numbers (called nodes) x0 = 2, x1 = 2.75, and x2 = 4 to find the second Lagrange interpolating polynomial for f (x) = 1/x.**

**(b) calculate f (3)**

**Solution**

F(x)= L0(x0) + L1(x1)+ L2(x2) (1)

1. We first determine the coefficient polynomials L0(x), L1(x), and L2(x).

let Li(x)= j=0, j≠i therefore

L0(x) = = (x − 2.75) (x − 4)

L1(x) = = (x − 2) (x − 4),

and L2(x) = = (x − 2) (x − 2.75).

Also, f (x0) = f (2) = 1/2, f (x1) = f (2.75) = 4/11, and f (x2) = f (4) = ¼

Now putting calculated values in equation (1) we have

F(x)= (x − 2.75) (x − 4) + (x − 2)(x − 4)+ (x − 2)(x − 2.75).

=

1. put x=3

f (3) =

=

= 0.32955.

**Algorithm:**

F(x)= Li(x)= j=0, j≠i

The algorithm for Lagrange interpolation is

F(x)= j≠i

1.Start

2. Read number of data (n)

3. Read data Xi and Yi for i=1 ton n

4. Read value of independent variables say xp

whose corresponding value of dependent say yp is to be determined.

5. Initialize: yp = 0

6. For i = 1 to n

Set p = 1

For j =1 to n

If i ≠ j then

Calculate p = p \* (xp - Xj)/(Xi - Xj)

End If

Next j

Calculate yp = yp + p \* Yi

Next i

7. Display value of yp as interpolated value.

8. Stop

**By Program:**

x=[2,2.75,4]

y=[0.5,0.3636,0.25]

m=len(x)

n= m - 1

xp=float(input("enter the value of x:"))

yp=0

for i in range(n+1):

p=1

for j in range(n+1):

if j != i:

p \*= (xp-x[j])/(x[i]-x[j])

yp += y[i]\*p

print('for x= %.2f, y=%f'%(xp,yp))

**computer generated output**

enter the value of x:3

for x= 3.00, y=0.329507

**we can see this result is equivalent to over calculation we did by hand**

**Newton divided difference interpolation:**

Given a set of data x0, x1, x2,...,xn [ (n+1) points] that may or may not be equally spaced

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Xi | fi | 1st order divided difference | 2nd order divided difference | 3rd order divided difference |
| X0 | f0 |  |  |  |
|  |  | F [x0, x1]= |  |  |
| X1 | f1 |  | F [x0, x1,x2]= |  |
|  |  | F [x1, x2]= |  | F [x0, x1,x2 ,x3]= |
| X2 | f2 |  | f [x1, x2 ,x3]= |  |
|  |  | F [x2, x3]= |  |  |
| X3 | f3 |  |  |  |
| .  .  . | .  .  . | .  .  . |  |  |

Then the polynomial of degree ‘n’ through (x0, y0), (x1, y1),, (xn,yn), is given by the newton’s Divided difference Interpolation formula (Also known as Newton’s General Interpolation formula.

F(x) = f(X0)+()F[x0,x1]+() () F[x0,x1,x2]+()() )() F[x0,x1,x2 ,x3]+…+ ]+()()…() F[x0,x1,x2 ,xn]

Gives nth degree polynomial from (n+1) points.

**Example: (a)** **compute the newton divided difference table for the following set of data point**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | **0.01** | **0.02** | **0.03** | **0.04** |
| **F(x)** | **1.2** | **2.5** | **3.6** | **4.3** |

**(b) find polynomial of degree 3 by newton divided difference method**

**(c) interpolate f (0.018)**

1. **newton divided difference table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Xi | fi | 1st order divided difference | 2nd order divided difference | 3rd order divided difference |
| 0.01 | 1.2 |  |  |  |
|  |  | F[x0, x1]==130 |  |  |
| 0.02 | 2.5 |  | F [x0, x1,x2]= =-1000 |  |
|  |  | F [x1, x2]==110 |  | F [x0, x1,x2 ,x3]= =-3333.3 |
| 0.03 | 3.6 |  | F [x1, x2 ,x3]= =-2000 |  |
|  |  | F [x2, x3]==70 |  |  |
| 0.04 | 4.3 |  |  |  |

**(b) polynomial by newton divided difference method**

F(x) = f(X0) +()F[x0, x1]+() () F[x0,x1,x2]+()() )() F[x0,x1,x2 ,x3]

Substituting values from the table, we get

F(x)=1.2+(x-0.01)130+(x-0.01) (x-0.02) (-1000)+ (x-0.01)(x-0.02)(0.03)(-3333.33)

=130x-0.1-1000x2+30x-0.2+x3-0.06x2+0.0602x2-0.000006

=x3-1000.06x2+159.9398x-0.300006

**(c) f (0.018)**

F (0.018) = (0.018)3-1000.06(0.018)2+159.9398(0.018)-0.300006

=2.2549

# Program:

# Newton divided difference formula

# Function to find the product term

def proterm(i, value, x):

pro = 1;

for j in range(i):

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

return pro;

# Function for calculating

# divided difference table

def dividedDiffTable(x, y, n):

for i in range(1, n):

for j in range(n - i):

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

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

return y;

# Function for applying Newton's

# divided difference formula

def applyFormula(value, x, y, n):

sum = y[0][0];

for i in range(1, n):

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

return sum;

# Function for displaying divided

# difference table

def printDiffTable(y, n):

for i in range(n):

for j in range(n - i):

print(round(y[i][j], 4), "\t",

end = " ");

print("");

# Driver Code

# number of inputs given

n = 4;

y = [[0 for i in range(10)]

for j in range(10)];

x = [0.01,0.02,0.03,0.04];

# y[][] is used for divided difference

# Table where y[][0] is used for input

y[0][0] = 1.2;

y[1][0] = 2.5;

y[2][0] = 3.6;

y[3][0] = 4.3;

# calculating divided difference table

y=dividedDiffTable(x, y, n);

# displaying divided difference table

printDiffTable(y, n);

# value to be interpolated

value = 0.018;

# printing the value

print("\nValue at", value, "is",

round(applyFormula(value, x, y, n), 2))

# output:

1.2 130.0 -1000.0 -33333.3333

2.5 110.0 -2000.0

3.6 70.0

4.3

Value at 0.018 is 2.25

**Remark :** till now we have studied polynomial interpolation which have the following **Drawbacks:**

* Polynomial of High-Degree → an appropriate choice of basic functions and interpolation points can mitigate some of the difficulties associated with high-degree polynomial
* Over-fitting → fitting a single polynomial to a large number of data points which would likely yield unsatisfactory oscillating behavior in the interpolant

**As how different methods are born, Piece-wise Interpolation solves these complications.**

**Definition (Piecewise polynomial)** Let [a, b] be an interval that is divided into subintervals [xi, xi-1], where i = 0, . . . , n − 1, x0=a ,xn=b . A piecewise polynomial is a function f(x) defined on [a, b] by f(X) = fi(x) xi-1 ≤ x≤ xi, i = 1, 2, . . . , n each function fi(x) is a polynomial defined on [xi-1 ,xi ]. The degree of f(x) is the maximum degree of each polynomial fi(x), for i = 1, 2, . . . ,n .

**Spline interpolation**

We know that high-degree polynomial interpolation can be problematic. However, if the fitting function is only required to have a few continuous derivatives, then one can construct a piecewise polynomial to fit the data.

**There** are the following types of spline interpolation

1. linear interpolation
2. quadratic interpolation
3. cubic interpolation
4. quartic interpolation

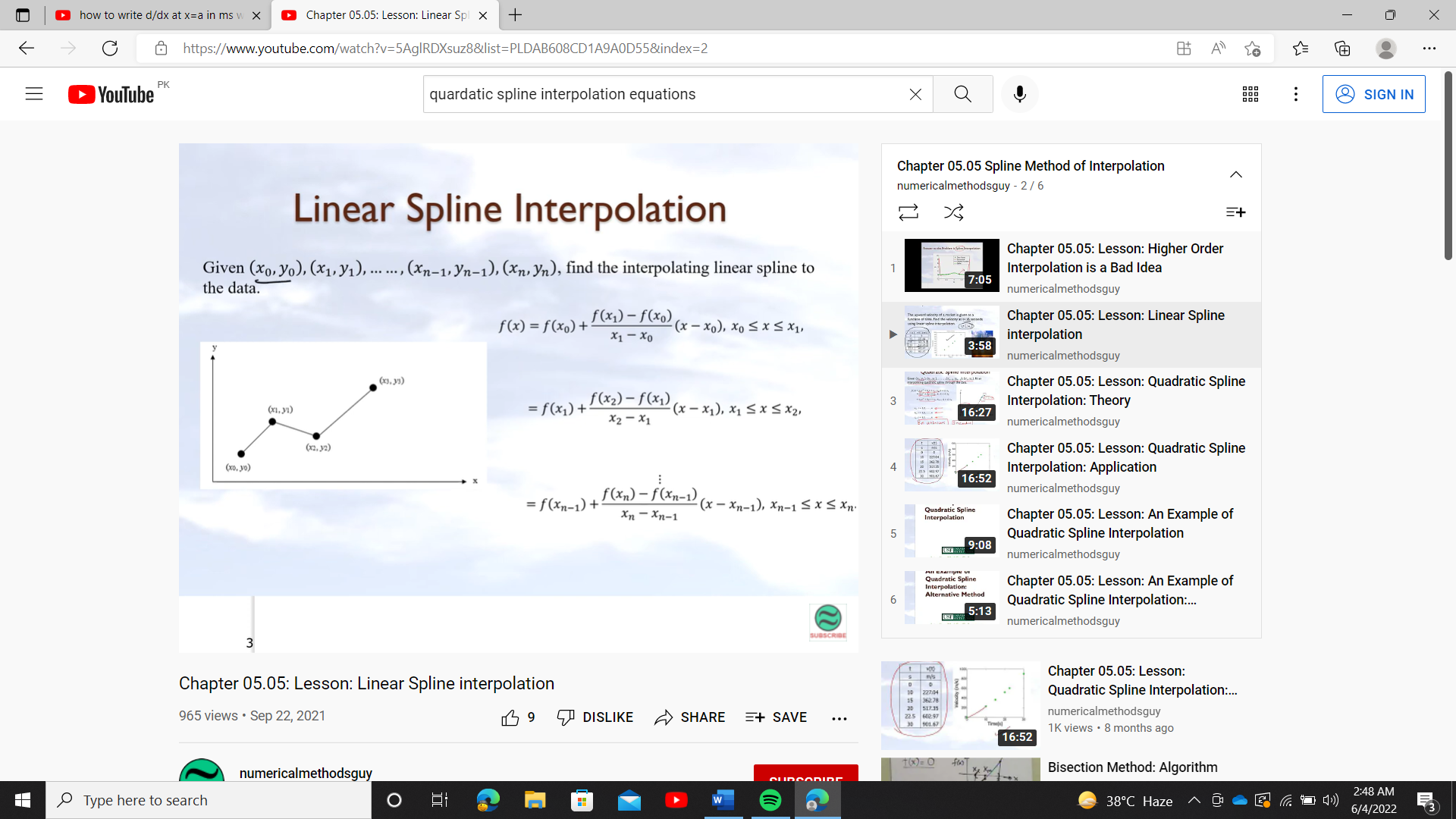
let us discuss one by one

**linear interpolation:** The simplest piecewise-polynomial approximation is piecewise-linear interpolation, which consists of joining a set of data points {(x0, y0), (x1,y1),.., (xn,yn) } by a series of straight lines ( forming the consecutive data through straight lines)

**Note**: Function must be continuous at the intersection point.

**Formula:**

*F(x) = f(x0) + (x-x0)*  x0 ≤ x≤ x1

*F(x) = f(x1) + (x-x1)*  x1 ≤ x≤ x2

*F(x) = f(xn-1) + (x-xn-1)*

xi-1 ≤ x≤ xi, i = 1, 2, . . ., n

**Example: from the given set of a data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | **0** | **2** | **4** | **7** |
| **F(x)** | **3** | **4** | **8** | **10** |

**(b) find piece wise polynomial by linear spline interpolation**

**(c)find f (3)**

*F(x) = f(x0) + (x-x0)*  0 ≤ x≤ 2

=3+*(x-0)*

= x+3

*F(x) = f(x1) + (x-x1)*  2 ≤ x≤ 4

=4+*(x-2)*

= 4+2x-4=2x

*F(x) = f(x2) + (x-x2)*  4 ≤ x≤ 7

=8+*(x-4)*

=8+ x-4) = x-=x-

For f (3) we will use *F(x) =*2x 2 ≤ x≤ 4

*F(3) =*2(3) =6

# Algorithm:

1. start .
2. from SciPy. Interpolate import interp1d.
3. defining x and corresponding y values for which we must do interpolation (with the help of arrays).
4. defining x1 values for which we will find y1 by interpolation (with the help of arrays)
5. Use SciPy and interp1d to perform linear spline interpolation f=interp1d(x,y,"linear")
6. Finding y1=f(x1)
7. Take value from user x2 in the range of x values and then find interpolated value for x2
8. show interpolated value
9. Plotting graph for x1,y1, values
10. Show graph
11. stop

# By program:

import numpy as np

import matplotlib.pyplot as plt

from scipy.interpolate import interp1d

x=np.array([0,2,4,7])

y=np.array([3,4,8,10])

x1=np.linspace(0,7,100)

f=interp1d(x,y,"linear") # we can also find cubic, quardatic ,nearest,zero interpolation by just writing required interpolation in place of linear but default is linear

y1=f(x1)

plt.plot(x1,y1,"o:") #plotting

plt.grid()

plt.show()

x2=float(input("enter the value for interpolation:"))

y2=f(x2)

print("x= %f , y= %d" %(x2,y2))

computer generated output:

enter the value for interpolation:3

x= 3.000000, y= 6

**Description: A picture containing chart

Description automatically generated**

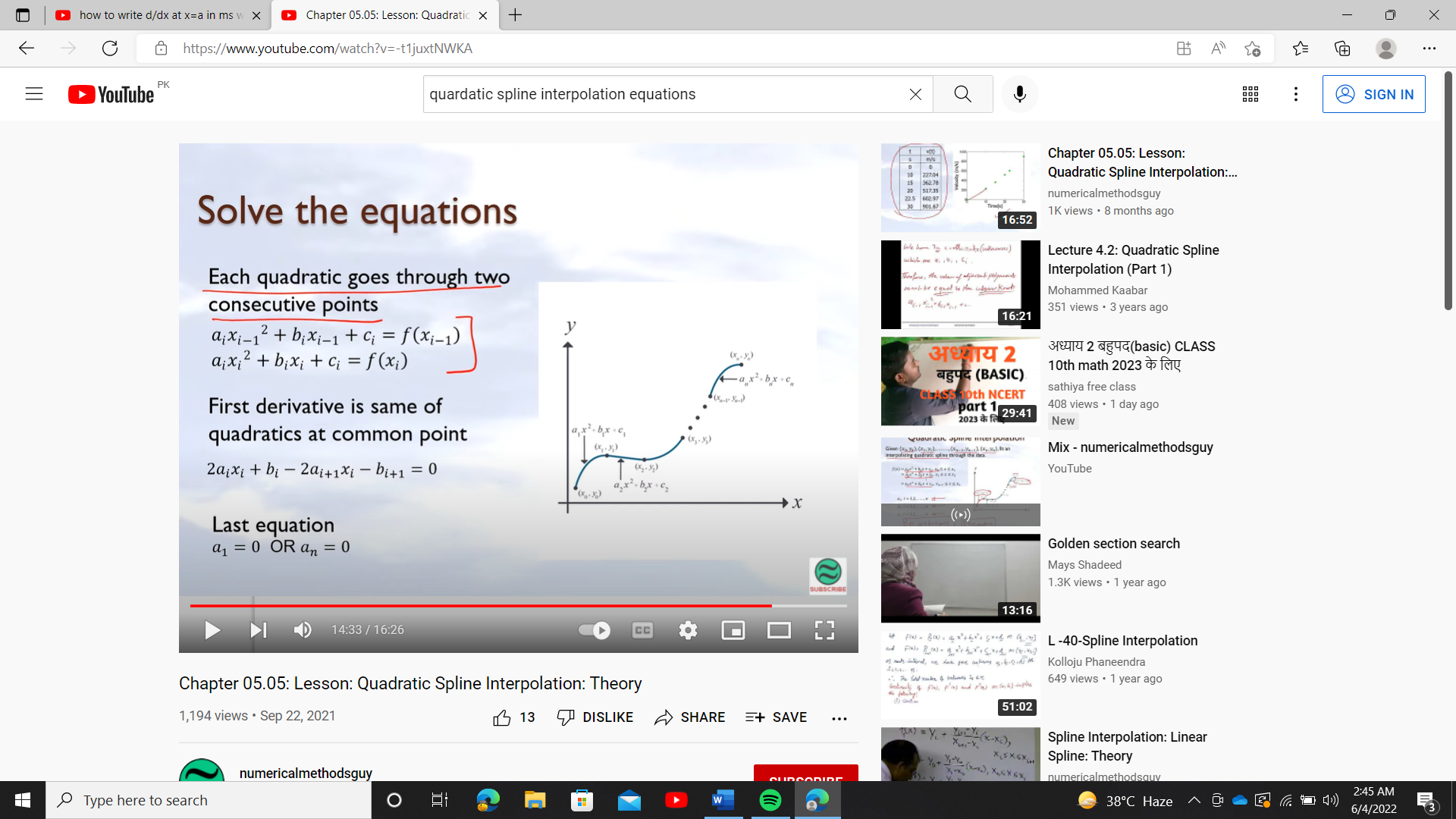
**which is same as we have calculated by hand above**

A **disadvantage of linear function approximation** is that there is likely no differentiability at the endpoints of the subintervals Slope changes abruptly at the interior points, this means that the first derivative is not continuous at these points. which means that the interpolating function is not “smooth. So how do we improve on this? We can do so by using quadratic splines.

**Quadratic polynomial:**  have the advantage over linear Splines that the sensitivity decreases as more data points are in each section. But if there are too many datapoints in each section the interpolation worsens. The interpolation can turn into a curve that is far of all the points that’s why we used quadratic interpolation.

a1x2+b1x+c1 x0≤x≤x1 (1)

a2x2+b2x+c2 x1≤x≤x2 (2)

anx2+bnx+cn xn-1≤x≤xn (3)

ai ;1, 2,…,n

bi ;1, 2,…,n 3n constants/unknown

ci ;1, 2,…,n

we have 3n constants so we should have 3n equations, now we will find all this 3n equations

**Each quadratic goes through 2 consecutives data points**

from equation (1) we have

y0= a1x02+b1x0+c1

y1= a1x12+b1x1+c1

from equation (2) we have

y1= a2x12+b2x1+c2

y2= a2x22+b2x2+c2

from equation (3) we have

Yn-1= anxn-12+bnxn-1+cn

Yn= anxn2+bnxn+cn

So, we have find “2n” equations, now we will find remaining “n” equation

Removing x0 we will have n points (remaining) and removing x1 we will have n-1 points (remaining) so we will have **n-1 interior points from n+1 points** (2 points x0, xn are exterior points which doesn’t satisfy the slopes are continuous because there is no quadratic before x0 and after xn)

**First derivative at two consecutive quadratics is continuous at common interior points**

a1x2+b1x+c1) = a2x2+b2x+c2)

a1x+b1) = a2x+b2)

a1x1+b1) = a2x1+b2)

a1x1+b1-a2x1-b2=0

(an-1x2+bn-1x+cn-1) = (anx2+bnx+cn)

an-1x+bn-1) = anx+bn)

an-1xn-1+bn-1= anxn-1+bn

an-1xn-1+bn-1-anxn-1-bn=0

So we will get n-1 equations by equating the slopes at n-1 interior points.

So far, **we have 2n+ (n-1) =3n-1 equation’s** so we are left with only one equation

**Last equation**

Choose a1 = 0 or an = 0

Criteria if |x1-x0|≤|xn-xn-1|

Then choose a1 = 0

Else choose an = 0

So, we will have **2n+n-1+1=3n** equations (as required)

**Example: The upward velocity of a rocket is given as a function of time as**

**Velocity as a function of time.**

|  |  |
| --- | --- |
| **t (s)** | **V(t) ((m/s)** |
| **0** | **0** |
| **10** | **227.04** |
| **15** | **362.78** |
| **20** | **517.35** |
| **22.5** | **602.97** |
| **30** | **901.67** |

(**a) Determine the value of the velocity at t =16 seconds using quadratic splines.**

**(b)Using the quadratic splines as velocity functions, find the acceleration of the rocket at t = 16s.**

Solution a) Since there are six data points, five quadratic splines pass through them.

V(t) = a1t2+b1t+c1 0≤t≤10

a2t2+b2t+c2 10≤t≤15

a3t2+b3t+c3 15≤t≤20

a4t2+b4t+c4 20≤t≤22.5

a5t2+b5t+c5 22.5≤t≤30

**1. Each quadratic spline passes through two consecutive data points**.

**a1t2+b1t+c1 passes through t = 0 and t = 10.**

a1(0)2+b1(0) +c1 =0(1)

a1(10)2+b1t (10)+c1 =227.04(2)

**a2t2+b2t+c2 passes through t = 10 and t = 15**.

a2(10)2+b2(10) +c2 =227.04 (3)

a2(15)2+b2(15) +c2 =362.78(4)

**a3t2+b3t+c3 passes through t = 15 and t = 20.**

a3(15)2+b3(15) +c3 =517.35 (5)

a3(20)2+b3(20) +c3 =517.35 (6)

**a4t2+b4t+c4 passes through t = 20 and t = 22.5**.

a4(20)2+b4(20) +c4 =602.97 (7)

a4(22.5)2+b4(22.5) +c4 =602.97 (8)

**a5t2+b5t+c5 passes through t = 22.5 and t = 30.**

a5(22.5)2+b5(22.5) +c5 = 901.67 (9)

a5(30)2+b5(30) +c5 = 901.67 (10)

**2. Quadratic splines have continuous derivatives at the interior data points.**

**At t = 10**

a1(10) +b1-a2(10)-b2=0 (11)

**At t = 15**

a2(15) +b2-a3(15)-b3=0 (12)

**At t = 20**

a3(20) +b3-a4(20)-b4=0 (13)

**At t = 22.5**

a4(22.5) +b4-a5(22.5)-b5=0 (14)

3. for last equation

|t1-t0|≤|t5-t4|

|10-0|≤|30-22.5|

10≤7.5 not true

So, we will choose a5=0

a5t2+b5t+c5

b5t+c5 = 901.67 (15)

now we will write matrix from those 15 equations

A = =

|  |  |  |  |
| --- | --- | --- | --- |
| i | ai | bi | ci |
| 1 | 0 | 22.704 | 0 |
| 2 | 0.8888 | 4.928 | 88.88 |
| 3 | –0.1356 | 35.66 | -141.61 |
| 4 | 1.6048 | –33.956 | 554.55 |
| 5 | 0.20889 | 28.86 | –152.13 |

By solving this matrix we will have

Therefore, splines are given by

V(t) = 22.704t0≤t≤10

= 0.8888t2+4.928t+88.8810≤t≤15

=-0.1356t2+35.66t-141.6115≤t≤20

=1.6048t2+-33.956t+554.5520≤t≤22.5

=0.20889t2+28.86t+-152.1322.5≤t≤30

At t =16 s

V (16)=-0.1356(16)2+35.66(16)-141.61

=394.24m/s

**(b) What is acceleration at t=16**

a (16)=

a (16)=-0.1356t2+35.66t-141.61)

= (-0.2712t+35.66

= -0.2712(16) +35.66

=31.321m/s2

# Algorithm:

1. start.
2. from SciPy. Interpolate import interp1d.
3. defining x and corresponding y values for which we must do interpolation (with the help of arrays).
4. defining x1 values for which we will find y1 by interpolation (with the help of arrays)
5. Use SciPy and interp1d to perform quadratic spline interpolation f=interp1d(x, y,"linear")
6. Finding y1=f(x1)
7. Take value from user x2 in the range of x values and then find interpolated value for x2
8. show interpolated value
9. Plotting graph for x1, y1, values
10. Show graph
11. stop

**Program:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.interpolate import interp1d

x=np.array([0,10,15,20,22.5,30])

y=np.array([0,227.04,362.78,517.35,602.97,901.67])

x1=np.linspace(0,30,100)

f=interp1d(x,y,"quadratic")

y1=f(x1)

plt.plot(x1,y1,"o:")

plt.grid()

plt.show()

x2=float(input("enter the value for interpolation:"))

y2=f(x2)

print("x= %f , y= %d" %(x2,y2))

**computer generated output:**

enter the value for interpolation:16

x= 16.000000, y= 392

**Description: A picture containing shape

Description automatically generated**

**which is approximately same to calculation we did by hand**

**cubic spline interpolation :**

A function ‘S’ is called a spline of degree ‘k’ if it satisfied the following conditions.

1. S is defined in the interval,-
2.  is continuous on, ; 
3. S is polynomial of degree Description: Shape

   Description automatically generated with medium confidence on each subinterval

,

cubic spline interpolation is the way of finding a curve that connect data points with a degree of three. Splines are polynomial that are smooth and continuous across a given plot and continuous first and second derivatives where they join.

Cubic spline has continuous second derivative whereas quadratic spline only has continuous first derivative so cubic spline is smoother

a1x3+b1x2+c1x+d1 x0≤x≤x1 (1)

a2x3+b2x2+c2x+d2 x1≤x≤x2 (2)

anx3+bnx2+cnx+dn xn-1≤x≤xn (3)

ai ;1, 2,…,n

bi ;1, 2,…,n 4n constants/unknown

ci ;1, 2,…,n  
di ;1,2,…,n

we have 4n constants so we should have 4n equations, now we will find all this 4n equations

**Each cubic goes through 2 consecutives data points**

from equation (1) we have

y0= a1x03+b1x02+c1x0+d1

y1= a1x13+b1x12+c1x1+d1

from equation (2) we have

y1= a2x13+b2x12+c2x1+d2

y2= a2x23+b2x22+c2x2+d2

from equation (3) we have

Yn-1= anxn-13+bnxn-12+cnxn-1+dn-1

Yn= anxn3+bnxn2+cnxn+dn

So we have find “2n” equations, now we will find remaining “2n” equation

Removing x0 we will have n points (remaining) and removing x1 we will have n-1 points (remaining) so we will have **n-1 interior points from n+1 points** (2 points x0 , xn are exterior points which doesn’t satisfy the slopes are continuous because there is no cubic before x0 and after xn)

**First derivative at two consecutive cubic are continuous at common interior points**

a1x3+b1x2+c1x+d1) = a2x3+b2x2+c2x+d2)

a1x2+2b1x+c1) = a2x2+2b2x+c2)

a1x12+2b1x1+c1) = a2x12+2b2x1+c2)

a1x12+2b1x1+c1 a2x12-2b2x1-c2=0

a1a2)x12+(2b1 -2b2) x1+c1 -c2=0

(an-1x3+bn-1x2+cn-1x+dn-1) = (anx3+bnx2+cnx+dn)

(3an-12+2bn-1+cn-1) = (3an2+2bn+cn)

(3an-1- 3an)2+(2bn-1- 2bn)+ (cn-1- cn)=0

So we will get n-1 equations by equating the slopes at n-1 interior points.

**Second derivative at two consecutive cubic are continuous at common interior points**

a1x3+b1x2+c1x+d1) = a2x3+b2x2+c2x+d2)

a1x2+2b1x+c1) = a2x2+2b2x+c2)

a1x+2b1) = a2x+2b2)

a1x1+2b1) = a2x1+2b2)

(a1a2)x1+2b1-2b2=0

(an-1x3+bn-1x2+cn-1x+dn-1) = (anx3+bnx2+cnx+dn)

(3an-1x2+2bn-1x+cn-1) = (3anx2+2bnx+cn)

(6an-1x+2bn-1) = (6anx+2bn)

(6an-1+2bn-1) = (6an+2bn)

(6an-1-6an)+2bn-1-2bn=0

So we will get n-1 equations by equating the second derivative at n-1 interior points.

So far, **we have 2n+ (n-1)+(n-1) =4n-2 equation’s** so we are left with 2 equations.

For last 2 equations we will use natural spline

Natural spline is defined as 2nd derivative for first and last polynomial equal to zero in the interpolation function boundary point .

6a1x0 +2b1=0

6anxn+2bn=0

So far we have 4n equations for n+1 data points

**Example: Interpolate a cubic spline between the three points (0, 1), (2, 2) and (4, 0).**

|  |  |
| --- | --- |
| **xi** | **yi** |
| **0** | **1** |
| **1** | **2** |
| **4** | **0** |

(**a) Determine the value of the velocity at x =3 seconds using cubic splines.**

**Solution** :

a) Since there are 3 data points, 2 cubic splines pass through them

a1x3+b1x2+c1x+d1 0≤x≤2

a2x3+b2x2+c2x+d2 2≤x≤4

Since we have 3 data points n+1 =3 so n=2 so we should have 4(2) =8 equations

**Each cubic goes through 2 consecutives data points**

a1x3+b1x2+c1x+d1  passes through the point x=0 and x=2

d1= 0 (1)

8a1+4b1+2c1+d1 =2 (2)

a2x3+b2x2+c2x+d2 passes through the point x=2and x=4

8a2+4b2+2c2+d2 =2 (3)

64a2+16b2+4c2+d2 =0 (4)

**First derivative at two consecutive cubic are continuous at common interior points**

a1x3+b1x2+c1x+d1) = a2x3+b2x2+c2x+d2)

a1x2+2b1x+c1) = a2x2+2b2x+c2)

a1(2)2+2b1(2)+c1) = a2(2)2+2b2(2)+c2)

a1+4b1+c1 a2-4b2-c2=0 (5)

**Second derivative at two consecutive cubic are continuous at common interior points**

a1x3+b1x2+c1x+d1) = a2x3+b2x2+c2x+d2)

a1x2+2b1x+c1) = a2x2+2b2x+c2)

a1x+2b1) = a2x+2b2)

a1+2b1) = a2+2b2)

a1+2b1 -a2-2b2 =0 (6)

**2nd derivative for first and last polynomial equal to zero in the interpolation function boundary point .**

6a1(0)+2b1=0 (7)

6a2(4)+2b2=0 (8)

**So we have eight equations for eight unknown**

**Matrix for this eight equation is**

= =

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **i** | **ai** | **bi** | **ci** | **di** |
| **1** | **-0.94** | **1** | **0.875** | **1.00** |
| **2** | **0.94** | **-1.125** | **3.125** | **-0.5** |

By solving matrix, we will have

Our two polynomials are

P1(x)= -0.94x3+.875t+10≤x≤2

P2(x)= x3-1.125x2+3.125x-0.5 2≤x≤4

at x=3

P2(3) = 33-1.125(3)2+3.125(3)-0.5 = 25.75

# Algorithm:

1. start.
2. from SciPy. Interpolate import Cubic Spline.
3. defining x and corresponding y values for which we must do interpolation (with the help of arrays).
4. defining x1 values for which we will find y1 by interpolation (with the help of arrays)
5. Use SciPy to perform cubic spline interpolation f= CubicSpline(x,y,bc\_type="natural")
6. Finding y1=f(x1)
7. Take value from user x2 in the range of x values and then find interpolated value for x2
8. show interpolated value
9. Plotting graph for x1, y1, values
10. Show graph
11. Stop

# Program:

import numpy as np

import matplotlib.pyplot as plt

from scipy.interpolate import CubicSpline

x=np.array([0,2,4])

y=np.array([1,2,0])

f=CubicSpline(x,y,bc\_type="natural")

x1=np.linspace(0,4,20)

y1=f(x1)

plt.plot(x,y,"o:",x1,y1,"-x") # plot both graphs in 1

plt.legend(["oringinal values","interpolated values"])

x2=float(input("enter the value for interpolation:"))

y2=f(x2)

print("x= %f , y= %d" %(x2,y2))

# output:

enter the value for interpolation:3

x= 3.000000 , y= 1

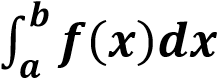
Description: Chart, line chart

Description automatically generated

**Numerical integration**

**Integration** is the process of finding Area under the curve. But this is not always possible to find exact value of integration (when function is not continuous) so we will find approximate(numerical) value of integration.

**Numerical integration:** The process of producing a numerical value for the defining integral



is called Numerical Integration. Numerical Integration is the study of how the numerical value of an integral can be found.

Also called Numerical Quadrature if Description: Shape

Description automatically generated with medium confidence which refers to finding a square whose area is the same as the area under the curve.

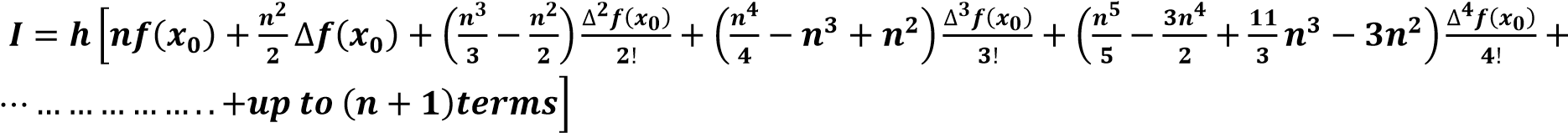
**A GENERAL formula FOR SOLVING NUMERICAL INTEGRATION**

This formula is also called a general quadrature formula.

Suppose f(x) is given for equidistant value of ‘x’ say a=x0, x0+h,x0+2h …. x0+nh = b

Let the range of integration (a, b) is divided into ‘n’ equal parts each of width ‘h’ so that “b-a=nh”.

By using fundamental theorem of numerical analysis, It has been proved the general quadrature formula which is as follows



By putting n into different values various formulae is used to solve numerical integration.

That are Trapezoidal Rule, Simpson’s 1/3, Simpson’s 3/8, Boole’s, Weddle’s etc.

IMPORTANCE: Numerical integration is useful when

* Function cannot be integrated analytically.
* Function is defined by a table of values.
* Function can be integrated analytically but resulting expression is so complicated.

**COMPOSITE (MODIFIED) NUMERICAL INTEGRATION**

Trapezoidal and Simpson’s rules are limited to operating on a single interval. Of course, since definite integrals are additive over subinterval, we can evaluate an integral by dividing the interval up into several subintervals, applying the rule separately on each one and then totaling up. This strategy is called Composite Numerical Integration.

TRAPEZOIDAL RULE

Trapezoidal Rule is a rule that evaluates the area under the curves by dividing the total area into smaller trapezoids rather than using rectangles. This integration works by approximating the region under the graph of a function as a trapezoid, and it calculates the area. Rule is based on approximating f(x) by a piecewise linear polynomial that interpolates f(x) at the nodes x0 ,x1 ,x2,...,xn

Description: Chart, line chart

Description automatically generated

Trapezoidal Rule defined as follows

= [ where h= n (number of trapezium)

And this is called Composite form of Trapezoidal Rule is

Derivation:

Consider a curve y=f(x) bounded by x0=a and x1=b we must find i.e.

Area under the curve y=f(x) then for one Trapezium under the area i.e. n = 1

Description: A picture containing diagram

Description automatically generated a= b=



f (x

0

)



f (x

1

)



Y



O



X



a=x

0



B=x

1

= area of trapezium =

= = [

For two trapeziums we will have n = 2

= [ + [ = [

For two trapeziums we will have n = 3

= [ + [ + [ = [

In general, for n – trapezium the points will be x0 ,x1 ,x2,...,xn and function will be y0 ,y1 ,y2,...,yn

= [

**= [**

**We can also derive formula by**

**Define y = f(x) in an interval,[a,b] then**

= + +…+

**= [ + [ +…+ [ +error**

Therefore,  **= [ where a=x0 , b =xn**

**Remarks.**

Trapezium rule is valid for n (number of trapezium) is even or odd.

The accuracy will be increase if number of trapeziums will be increased OR step size will be decreased mean number of step size will be increased.

**Error:**

**Error =**

EXAMPLE: Evaluate f(x)= using Trapezoidal Rule when h=

SOLUTION

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| X | 0 | 1/4 | 1/2 | 3/4 | 1 |
| F(x) | 1 | 0.9412 | 0.8000 | 0.6400 | 0.5000 |

Since by Trapezoidal Rule

f(x)= =  [

**here number of trapeziums is n= 4 and number of points N=5**

**=**  [

**=**  [

= 0.7828

**truncation Error:**

For error we first calculate

=

=

= = =

= = 0.4171

Now substitute calculated values in above formula:

**Error = 0.0005**

**Error =**

**Exact integration = =0.785**

**= │0.785-0.7828│**

**=0.0022**

**Algorithm:**

**Start**

**2. Define function f(x)**

**3. Read lower limit of integration, upper limit of**

**integration and number of sub interval**

**4. Calculate: step size = (upper limit - lower limit)/number of sub interval**

**5. Set: integration value = f (lower limit) + f (upper limit)**

**6. Set: i = 1**

**7. If i > number of sub interval then go to**

**8. Calculate: k = lower limit + i \* h**

**9. Calculate: Integration value = Integration Value + 2\* f(k)**

**10. Increment i by 1 i.e., i = i+1 and go to step 7**

**11. Calculate: Integration value = Integration value \* step size/2**

**12. Display Integration value as required answer**

**13. Stop**

**By program: ( Trapezoidal Method) numerical integration**

def f(x):

return 1/(1 + x\*\*2)

def trapezoidal(x0,xn,n):

h = (xn - x0) / n

integration = f(x0) + f(xn)

for i in range(1,n):

k = x0 + i\*h

integration = integration + 2 \* f(k)

integration = integration \* h/2

return integration

lower\_limit = float(input("Enter lower limit of integration: "))

upper\_limit = float(input("Enter upper limit of integration: "))

sub\_interval = int(input("Enter number of sub intervals: "))

result = trapezoidal(lower\_limit, upper\_limit, sub\_interval)

print("Integration result by Trapezoidal method is: %0.4f" % (result) )

# plotting

x=np.linspace(-1,3,50)

plt.plot(x,f(x))

plt.xlabel('x\_axis' ,fontsize =10)

plt.ylabel('y\_axis' ,fontsize =10)

plt.fill\_between(x,f(x),where = [(x>0) and (x<1) for x in x])

#exact integration

import sympy as sy

x = sy.Symbol('x')

sy.integrate(f(x),(x,0,1))

**computer generated output :**

**#Numerical integration**

Enter lower limit of integration: 0

Enter upper limit of integration: 1

Enter number of sub intervals: 4

Integration result by Trapezoidal method is: **0.7828 which is equivalent to our result calculated by hand**

**# Exact integration**

𝜋/4 which is equal to 0.785 radian

Description: Icon

Description automatically generated

**SIMPSON’S Description: Shape

Description automatically generated with low confidence RULE**

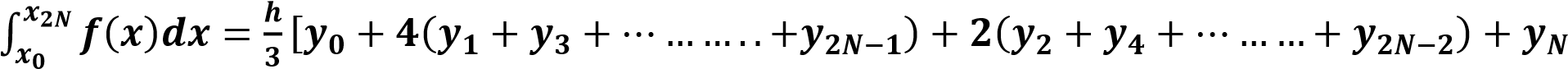
Rule is base0d on approximating f(x) by a Quadratic Polynomial that interpolate f(x) at



Simpson’s Rule is defined as for simple case

**= [**

While in composite form it is defined as

-

REMARK

In Simpson Rule number of trapezium must of Even and number of points must of Odd.

**DERIVATION Of SIMPSON’S Description: Shape

Description automatically generated with low confidence RULE (1st method)**

Given function values at 3 points as (*x*0*, f*(*x*0)), (*x*1*,f*(*x*1)), and (*x*2*,f*(*x*2)), we can estimate *f*(*x*) using Lagrange polynomial interpolation. Then

When *a* = *x*0, *b* = *x*2, (*a* + *b*)*/*2 = *x*1, and *h* = (*b* − *a*)*/*2Description: A picture containing text, shoji

Description automatically generated

I= (x-x1) (x-x2) + (x-x0) (x-x2)+ (x-x0) (x-x2) + (x-x0) (x-x1)

I **= [**

It can be proved that single segment application of Simpson’s 1*/*3 rule has a truncation error of

**Error:**

**DERIVATION OF SIMPSON’S Description: Shape

Description automatically generated with low confidence RULE (2nd method)**

= + +…+

**= [ + [ +…+ [ +error**

Therefore, [

This is required formula for Simpson’s (1/3) Rule

REMARK: Simpsons rule is better than trapezoidal rule.

**Example: Evaluate f(x)= using Simpsons 1/3 Rule when h=**

SOLUTION

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| X | 0 | 1/4 | 1/2 | 3/4 | 1 |
| F(x) | 1 | 0.9412 | 0.8000 | 0.6400 | 0.5000 |

Since by Trapezoidal Rule

f(x)= =  [

**here number of trapeziums is n= 4(even) and number of points N=5**

**=**  [

**=**  [

= 0.7854

**Error: Error :**

=

=

= = =

**=**

= = -1.807

Now substitute calculated values in above formula:

**Error = 0.0000196**

**Error =**

**Exact integration = =0.785**

**= │0.785-0.7854│**

**=0.0004**

**Algorithm:**

. Start

2. Define function f(x)

3. Read lower limit of integration, upper limit of

integration and number of sub interval

4. Calculate: step size = (upper limit - lower limit)/number of sub interval

5. Set: integration value = f (lower limit) + f(upper limit)

6. Set: i = 1

7. If i > number of sub interval then go to

8. Calculate: k = lower limit + i \* h

9. If i mod 2 =0 then

Integration value = Integration Value + 2\* f(k)

Otherwise

Integration Value = Integration Value + 4 \* f(k)

End If

10. Increment i by 1 i.e., i = i+1 and go to step 7

11. Calculate: Integration value = Integration value \* step size/3

12. Display Integration value as required answer

13. Stop

**Program: numerical integration**

# Simpson's 1/3 Rule

def f(x):

return 1/(1 + x\*\*2)

def simpson13(x0,xn,n):

h = (xn - x0) / n

integration = f(x0) + f(xn)

for i in range(1,n):

k = x0 + i\*h

if i%2 == 0:

integration = integration + 2 \* f(k)

else:

integration = integration + 4 \* f(k)

integration = integration \* h/3

return integration

lower\_limit = float(input("Enter lower limit of integration: "))

upper\_limit = float(input("Enter upper limit of integration: "))

sub\_interval = int(input("Enter number of sub intervals: "))

result = simpson13(lower\_limit, upper\_limit, sub\_interval)

print("Integration result by Simpson's 1/3 method is: %0.4f" % (result) )

x=np.linspace(-1,3,50)

plt.plot(x,f(x))

plt.xlabel('x\_axis' ,fontsize =10)

plt.ylabel('y\_axis' ,fontsize =10)

plt.fill\_between(x,f(x),where = [(x>0) and (x<1) for x in x])

#exact integration

import sympy as sy

x = sy.Symbol('x')

sy.integrate(f(x),(x,0,1))

**computer generated output :**

**numerical integration**

Enter lower limit of integration: 0

Enter upper limit of integration: 1

Enter number of sub intervals: 4

Integration result by Simpson's 1/3 method is: 0.7854

**Exact integration**

0.785

**Description: Icon

Description automatically generated**

**EXAMPLE**

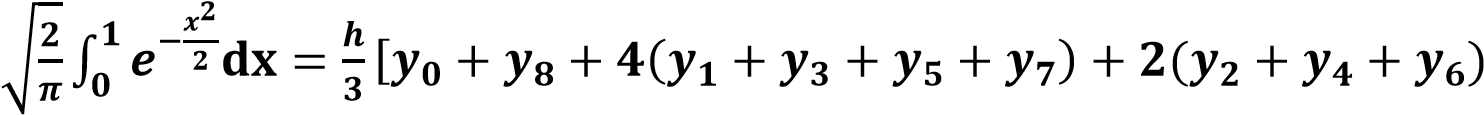
Compute Description: Shape

Description automatically generated with medium confidence using Simpson’s (1/3) Rule when 

**SOLUTION**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X | 0 | 0.125 | 0.250 | 0.375 | 0.5 | 0.625 | 0.750 | 0.875 | 1 |
| F(x) | 0.798 | 0.792 | 0.773 | 0.744 | 0.704 | 0.656 | 0.602 | 0.544 | 0.484 |

Since by Simpson’s Rule

-

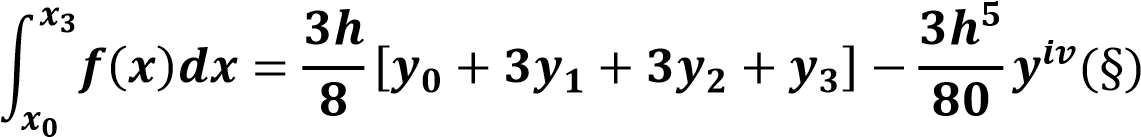
Description: Shape

Description automatically generated with medium confidenceAfter putting the values.

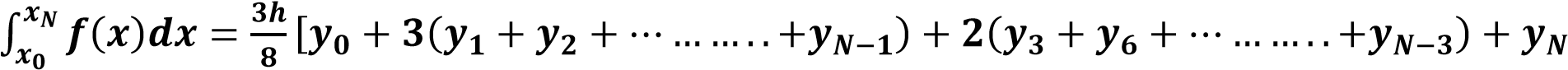
**Simpson’s Rule**

Rule is based on fitting four points by a cubic. It is completely based on the cubic interpolation rather than the quadratic interpolation.

Simpson’s Rule is defined as for simple case



While in composite form (“n” must be divisible by 3) it is defined as

-

**DERIVATION**

= + +…+

**= [ + [ +…+ [ +error**

Therefore, [

This is required formula for Simpson’s (3/8) Rule.

**Error:**

**Evaluate f(x)= using Trapezoidal Rule when h=0.1**

SOLUTION

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 |
| F(x) | 1 | 0.99 | 1.04 | 0.9412 | 0.862 | 0.8 | 0.735 | 0.671 | 0.6010 | 0.552 |

Since by Trapezoidal Rule

f(x)= =  [

**here number of trapeziums is n= 4(even) and number of points N=5**

**=**  [

**=**  [

= 0.037519.186 = 0.71

**Error :**

=

=

= = =

**=**

= = -1.807

Now substitute calculated values in above formula:

**Error = 0.0000662**

**Error =│Exact integration-numerical integration│**

**Exact integration = =0.7329**

**= │0.7329-0.71│**

**=0.0229**

**Algorithm:**

1.Start

2. Define function f(x)

3. Read lower limit of integration, upper limit of

integration and number of sub interval

4. Calculate: step size = (upper limit - lower limit)/number of sub interval

5. Set: integration value = f (lower limit) + f(upper limit)

6. Set: i = 1

7. If i > number of sub interval then go to

8. Calculate: k = lower limit + i \* h

9. If i mod 3 =0 then

Integration value = Integration Value + 2\* f(k)

Otherwise

Integration Value = Integration Value + 3 \* f(k)

End If

10. Increment i by 1 i.e., i = i+1 and go to step 7

11. Calculate: Integration value = Integration value \* step size\*3/8

12. Display Integration value as required answer

13. Stop

**Program: # Simpson's 3/8 Rule (numerical integration)**

def f(x):

return 1/(1 + x\*\*2)

def simpson38(x0,xn,n):

h = (xn - x0) / n

integration = f(x0) + f(xn)

for i in range(1,n):

k = x0 + i\*h

if i%2 == 0:

integration = integration + 2 \* f(k)

else:

integration = integration + 3 \* f(k)

integration = integration \* 3 \* h / 8

return integration

lower\_limit = float(input("Enter lower limit of integration: "))

upper\_limit = float(input("Enter upper limit of integration: "))

sub\_interval = int(input("Enter number of sub intervals: "))

result = simpson38(lower\_limit, upper\_limit, sub\_interval)

print("Integration result by Simpson's 3/8 method is: %0.6f" % (result) )

#plotting

x=np.linspace(-1,3,50)

plt.plot(x,f(x))

plt.xlabel('x\_axis' ,fontsize =10)

plt.ylabel('y\_axis' ,fontsize =10)

plt.fill\_between(x,f(x),where = [(x>0) and (x<1) for x in x])

**#Exact integration**

import sympy as sy

x = sy.Symbol('x')

sy.integrate(f(x),(x,0,0.9))

**computer generated output:**

numerical integration:

Enter lower limit of integration: 0

Enter upper limit of integration: 0.9

Enter number of sub intervals: 9

Integration result by Simpson's 3/8 method is: 0.675968

Exact integration :0.7328

**Description: Icon

Description automatically generated**

TRAPEZOIDAL AND SIMPSON’S RULE ARE CONVERGENT

If we assume Truncation error, then in the case of Trapezoidal Rule

**I-A=**  Where  is the exact integral and the approximation. If

 then assuming “” bounded

 (This the definition of convergence of Trapezoidal Rule)

For Simpson’s Rule we have the similar result

**I-A=**  Where  is the exact integral and the approximation

If  then assuming “Description: A black rectangle with a black background

Description automatically generated with low confidence” bounded

 (This the definition of convergence of Simpson’s Rule)

The End