



PPPPPPP**PROJECT**

***Project pr Numerical analysis***

**Submitted by:**

**Areesha Shoukat (SP20-BSM-009)**

**Malaika bibi (SP20-BSM-022)**

**Course instructor:**

**Dr Umair Umer**

***Course introduction:***

***Numerical analysis is the area of mathematics and computer science that creates , analyze ,and implements algorithms for solving numerically the problems of continuous mathematics.***

***Table of contents:***

**1.Solution of equation in one variable:**

**1.1 bisection method**

**1.2 false position method**

**1.3newton Rapson method**

**1.4 secent method**

**2.direct method for solving linear problem:**

**2.1** gauss elimination method

**2.2** gauss jacobi method

**2.3** gauss seidel method

**2.4** Sor method

**3.iterative techniques in matrix algebra:**

**3.1** Norms of vector

**3.2** eigenvalues and eigen vectors

**4.Interpolation:**

**4.1** Forward

**4.2** Backward

**4.3** Central

**4.4 spline interpolation**

**4.5 Quadratic**

**4.6 cubic interpolation**

**5.Numerical interation**

**5.1 trapezoidal rule:**

**5.2 simpson’s one third rule**

**BISECTION METHOD:**

Bisection method is also known as Bolzano’s method

**Working rule:**

Let f(x)=0 be the given equation .

Find a and b such that f(a)<0 and f(b)>0, **f(a) , f(b)<0**

Find mid-point 

.

***Example:***

X= 

F(x)=1-2=-4 <0

f(2)=22 -2 =2 >0

a=1, b=2 , c=a+b/2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| a | b | F(a) | F(b) | x=a+b/2 | F(x) | E(a) |
| 1 | 2 | -4 | 2 | 1.5000 | o.2500 |  |
| 1 | 1.5000 | -4 | 0.2500 | 2.5000 | -0.43 | 57% |
| 1.2500 | 1.5000 | -0.43 | 0.2500 | 1.3750 | -0.1 | 43% |
| 1.3750 | 1.5000 | -0.1 | 0.2500 | 1.4375 | 0.06 | 16% |
| 1.3750 | 1.4375 | -0.1 | 0.06 | 1.4063 | -0.02 | 29 |
| 1.4063 | 1.4375 | -0.02 | 0.06 | 1.4219 | 0.02 | 10 |
| 1.4063 | 1.4219 | -0.02 | 0.02 | 1.4141 | -0.0003 | 6 |
| 1.4141 | 1.4180 | -0.0003 | 0.02 | 1.4180 | 0.01 | 3 |
| 1.4141 | 1.4161 | -0.0003 | 0.0005 | 1.4161 | 0.005 | 19 |
| 1.4141 | 1.4151 | -0.0003 | 0.001 | 1.4151 | 0.001 | 2 |

*Root lies between [1,2]*

* **Advantages :**
* Bisection method is simple and easy to implement.
* Always convergent
* **Disadvantages :**
* Convergence is slow .
* Bisection method finds only one root at a time.

1. start

2. Define function f(x)

3. Choose initial guesses x0 and x1 such that f(x0)f(x1) < 0

4. Choose pre-specified tolerable error e.

5. Calculate new approximated root as x2 = (x0 + x1)/2

6. Calculate f(x0)f(x2)

a. if f(x0)f(x2) < 0 then x0 = x0 and x1 = x2

b. if f(x0)f(x2) > 0 then x0 = x2 and x1 = x1

c. if f(x0)f(x2) = 0 then go to (8)

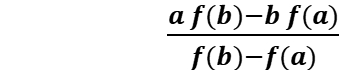
7. if |f(x2)| > e then go to (5) otherwise go to (8)

8. Display x2 as root.

9. Stop

* ***False position method:***

**Formula:**



The false position method is **one of the iterative methods of finding the roots of a non-linear equation of the form f(x) = 0**

**Advantages:**

1. This method is used for the numerical solution of algebraic equations which have a single equation.
2. 2. Convergence is guaranteed.
3. 3. This method is bracketing method and it is always convergent.

**Disadvantages:**

2. Slow rate of convergence.

3. Although convergence of Regula Falsi method is guaranteed, it is generally slow.

Example: find the root of f(x)=3x 

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ***Iteration*** | ***a*** | **b** | **c** | **F(a)** | **F(b)** | **F(c)** |  |
| 1 | -1 | 1 | -0.761 | -1.103 | 8.154 | -1.066 |  |
| 2 | -0.761 | 1 | -0,557 | -1.066 | 8.154 | -0.958 |  |
| 3 | -0.557 | 1 | -0.394 | -0.958 | 8.154 | -0.618 |  |
| 4 | -0.394 | 1 | -0.269 | -0.797 | 8.154 | -0.618 |  |
| 5 | -0.269 | 1 | -0.180 | -0.618 | 8.154 | -0.451 |  |
| 6 | -0.180 | 1 | -0.118 | -0.451 | 8.154 | -0.315 |  |
| 7 | -0.118 | 1 | -0.076 | -0.315 | 8.154 | -0.213 |  |
| 8 | -0.076 | 1 | -0.049 | -0.213 | 8.154 | -0.140 |  |

***Secent Method***

**Find the root of the equation .**

Let f(x)=   
  
here

|  |  |  |  |
| --- | --- | --- | --- |
| ***x*** | 0 | 1 | 2 |
| ***f*(*x*)** | -1 | -1 | 5 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ***n*** | ***x*0** | ***f*(*x*0)** | ***x*1** | ***f*(*x*1)** | ***x*2** | ***f*(*x*2)** | **Update** |
| 1 | 1 | -1 | 2 | 5 | 1.16667 | -0.5787 | *x*0=*x*1 *x*1=*x*2 |
| 2 | 2 | 5 | 1.16667 | -0.5787 | 1.25311 | -0.28536 | *x*0=*x*1 *x*1=*x*2 |
| 3 | 1.16667 | -0.5787 | 1.25311 | -0.28536 | 1.33721 | 0.05388 | *x*0=*x*1 *x*1=*x*2 |
| 4 | 1.25311 | -0.28536 | 1.33721 | 0.05388 | 1.32385 | -0.0037 | *x*0=*x*1 *x*1=*x*2 |
| 5 | 1.33721 | 0.05388 | 1.32385 | -0.0037 | 1.32471 | -0.00004 | *x*0=*x*1 *x*1=*x*2 |

***Newton Raphson method:***

The Newton-Raphson method (also known as Newton’s method) is a way to quickly find a good approximation for the root of a real-valued function f(x) = 0f(x)=0. It uses the idea that a continuous and differentiable function can be approximated by a straight line tangent to it. In numerical analysis, Newton’s method, also known as the “***Newton–Raphson method, named after Isaac Newton and Joseph Raphson*** “is a root-finding algorithm which produces successively better approximations to the roots of a real-valued function.

**Advantages:**

* Fast convergence as long as Initial guess is close to solution.
* Different solutions can be found with a different starting guess.

**Disadvantages.**

* It will only work if the Jacobian can be completed.
* If the Jacobian is singular the algorithm breaks
* The number of iteration can not be determined before the algorithm breaks

.

**Find the roots of the equation x3.-x-1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***n*** | ***x*0** | ***f*(*x*0)** | ***f*′(*x*0)** | ***x*1** | **Update** |
| 1 | 1.5 | 0.875 | 5.75 | 1.34783 | *x*0=*x*1 |
| 2 | 1.34783 | 0.10068 | 4.44991 | 1.3252 | *x*0=*x*1 |
| 3 | 1.3252 | 0.00206 | 4.26847 | 1.32472 | *x*0=*x*1 |
| 4 | 1.32472 | 0 | 4.26463 | 1.32472 | *x*0=*x*1 |

***secant method:***

In numerical analysis, the secant method is a root-finding algorithm that uses a succession of roots of secant lines to better approximate a root of a function f. The secant method can be thought of as a finite-difference approximation of Newton’s method.

Starting with initial values x0 and x1, we construct a line through the points (x0, f(x0)) and (x1, f(x1)), as shown in the picture above. In slope–intercept form, the equation of this line is

***Advantages*:**

* It may converge even faster and it doesn’t need to bracket the root
* It requires two guesses that do not need to bracket the root.

It doesn’t require use of derivative of a given function.

**Disadvantages:**

* It is not guaranteed to converge.
* It may diverge.

***Python Source Code: Secant Method***

**# Defining Function**

**def f**(**x):**

**return x\*\*3 - 5\*x - 9**

**# Implementing Secant Method**

**def secant(x0,x1,e,N):**

**print('\n\n\*\* SECANT METHOD IMPLEMENTATION \*\*')**

**step = 1**

**condition = True**

**while condition:**

**if f(x0) == f(x1):**

**print('Divide by zero error!')**

**break**

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

**print('Iteration-%d, x2 = %0.6f and f(x2) = %0.6f' % (step, x2, f(x2)))**

**x0 = x1**

**x1 = x2**

**step = step + 1**

**if step > N:**

**print('Not Convergent!')**

**break**

**condition = abs(f(x2)) > e**

**print('\n Required root is: %0.8f' % x2)**

**# Input Section**

**x0 = input('Enter First Guess: ')**

**x1 = input('Enter Second Guess: ')**

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

**N = input('Maximum Step: ')**

**# Converting x0 and e to float**

**x0 = float(x0)**

**x1 = float(x1)**

**e = float(e)**

**# Converting N to integer**

**N = int(N)**

**#Note: You can combine above three section like this**

**# x0 = float(input('Enter First Guess: '))**

**# x1 = float(input('Enter Second Guess: '))**

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

**# N = int(input('Maximum Step: '))**

**# Starting Secant Method**

**secant(x0,x1,e,N)**

**Python Program Output: Secant Method**

**Enter First Guess: 2**

**Enter Second Guess: 3**

**Tolerable Error: 0.000001**

**Maximum Step: 10**

**\* SECANT METHOD IMPLEMENTATION \***

**Iteration-1, x2 = 2.785714 and f(x2) = -1.310860**

**Iteration-2, x2 = 2.850875 and f(x2) = -0.083923**

**Iteration-3, x2 = 2.855332 and f(x2) = 0.002635**

**Iteration-4, x2 = 2.855196 and f(x2) = -0.000005**

**Iteration-5, x2 = 2.855197 and f(x2) = -0.000000**

**Required root is: 2.85519654**

***Fixed point iteration:***

The fixed point iteration method in numerical analysis is used to find an approximate solution to algebraic and transcendental equations. Sometimes, it becomes very tedious to find solutions to cubic, bi-quadratic and transcendental equations; then, we can apply specific numerical methods to find the solution; one among those methods is the fixed point iteration method.

* The fixed point iteration method uses the concept of a fixed point in a repeated manner to compute the solution of the given equation. A fixed point is a point in the domain of a function g such that g(x) = x. In the fixed point iteration method.
* Fixed Point Iteration Method
* Suppose we have an equation f(x) = 0, for which we have to find the solution. The equation can be expressed as x = g(x). Choose g(x) such that |g’(x)| < 1 at x = xo where xo,is some initial guess called fixed point iterative scheme. Then the iterative method is applied by successive approximations given by xn = g(xn – 1), that is, x1 = g(xo), x2 = g(x1) and so on.

* ***Algorithm of Fixed Point Iteration Method***

Choose the initial value xo for the iterative method. One way to choose xo is to find the values x = a and x = b for which f(a) < 0 and f(b) > 0. By narrowing down the selection of a and b, take xo as the average of a and b.

Express the given equation, in the form x = g(x) such that |g’(x)| < 1 at x = xo. If there more than one possibility of g(x), choose the g(x) which has the minimum value of g’(x) at x = xo.

* By applying the successive approximations xn = g(xn – 1), if f is a continuous function, we get a sequence of {xn} which converges to a point lo, which is the approximate solution of the given equation.
* ***Example 1:***

* Find the first approximate root of the equation 2x3 – 2x – 5 = 0 up to 4 decimal places.

* Solution:

* Given f(x) = 2x3 – 2x – 5 = 0
* As per the algorithm, we find the value of xo, for which we have to find a and b such that f(a) < 0 and f(b) > 0

* Now, f(0) = – 5

* F(1) = – 5

* F(2) = 7

* Thus, a = 1 and b = 2

* Therefore, xo = (1 + 2)/2 = 1.5

* Now, we shall find g(x) such that |g’(x)| < 1 at x = xo

* 2x3 – 2x – 5 = 0

* ⇒ x = [(2x + 5)/2]1/3

* G(x) = [(2x + 5)/2]1/3 which satisfies |g’(x)| < 1 at x = 1.5

* Now, applying the iterative method xn,= g(xn – 1) for n = 1, 2, 3, 4, 5, …

* For n = 1; x1 = g(xo) = [{2(1.5) + 5}/2]1/3 = 1.5874
* For n = 2; x2 = g(x1) = [{2(1.5874) + 5}/2]1/3 = 1.5989

* For n = 3; x3 = g(x2) = [{2(1.5989) + 5}/2]1/3 = 1.60037

* For n = 4; x4 = g(x3) = [{2(1.60037) + 5}/2]1/3 = 1.60057

* For n = 5; x5 = g(x4) = [{2(1.60057) + 5}/2]1/3 = 1.60059

* For n = 6; x6 = g(x5) = [{2(1.60059) + 5}/2]1/3 = 1.600597 ≈ 1.6006

* The approximate root of 2x3 – 2x – 5 = 0 by the fixed point iteration method is 1.6006.

: **Fixed Point Iteration Method Python Program**

**# Fixed Point Iteration Method**

**# Importing math to use sqrt function**

**import math**

**def f(x):**

**return x\*x\*x + x\*x -1**

**# Re-writing f(x)=0 to x = g(x)**

**def g(x):**

**return 1/math.sqrt(1+x)**

**# Implementing Fixed Point Iteration Method**

**def fixedPointIteration(x0, e, N):**

**print('\n\n\*\* FIXED POINT ITERATION \*\*')**

**step = 1**

**flag = 1**

**condition = True**

**while condition:**

**x1 = g(x0)**

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

**x0 = 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.')**

**# Input Section**

**x0 = input('Enter Guess: ')**

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

**N = input('Maximum Step: ')**

**# Converting x0 and e to float**

**x0 = float(x0)**

**e = float(e)**

**# Converting N to integer**

**N = int(N)**

**#Note: You can combine above three section like this**

**# x0 = float(input('Enter Guess: '))**

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

**# N = int(input('Maximum Step: '))**

**# Starting Newton Raphson Method**

**fixedPointIteration(x0,e,N)**

**Output**

**Enter Guess: 2**

**Tolerable Error: 0.00001**

**Maximum Step: 10**

**\* FIXED POINT ITERATION \***

**Iteration-1, x1 = 0.577350 and f(x1) = -0.474217**

**Iteration-2, x1 = 0.796225 and f(x1) = 0.138761**

**Iteration-3, x1 = 0.746139 and f(x1) = -0.027884**

**Iteration-4, x1 = 0.756764 and f(x1) = 0.006085**

**Iteration-5, x1 = 0.754472 and f(x1) = -0.001305**

**Iteration-6, x1 = 0.754965 and f(x1) = 0.000281**

**Iteration-7, x1 = 0.754859 and f(x1) = -0.000060**

**Iteration-8, x1 = 0.754882 and f(x1) = 0.000013**

**Iteration-9, x1 = 0.754877 and f(x1) = -0.000003**

**Required root is: 0.75487680**

--------------------------------------------------------------------------------------

***Chapter#2***

***2.Direct method for solving linear system:***

* 2.1:Gauss elimination method
* 2.2:Gauss seidel method
* 2.3:Gauss Jacobi method

***Gauss elimination method:***

In mathematics, the Gaussian elimination method is known as the row reduction algorithm for solving linear equations systems. It consists of a sequence of operations performed on the corresponding matrix of coefficients. We can also use this method to estimate either of the following:

* **The rank of the given matrix**
* **The determinant of a square matrix**
* **The inverse of an invertible matrix**

***Example:***

**Question:**

***Solve the following system of equations:***

**x + y + z = 2**

**x + 2y + 3z = 5**

**2x + 3y + 4z = 11**

***Solution:***

Given system of equations are:

x + y + z = 2

x + 2y + 3z = 5

2x + 3y + 4z = 11

Let us write these equations in matrix form.

Subtracting R1 from R2 to get the new elements of R2, i.e. R2 → R2 – R1.

From this we get,

From this we get,

Let

us make another operation as R3 → R3 – 2R1

[

Subtract R2 from R1 to get the new elements of R1, i.e. R1 → R1 – R2.

Now

subtract, R2 from R3 to get the new elements of R3, i.e. R3 → R3 – R2

means

Here,

x – z = -1

y + 2z = 3

0 = 4

, there is no solution for the given system of equations. That

***Gauss seidel method:***

In numerical linear algebra, the Gauss–Seidel method, also known as the Liebman method or the method of successive displacement, is an iterative method used to solve a system of linear equations. It is named after the **German** **mathematicians** **Carl** **Friedrich** **Gauss** **and** **Philipp** **Ludwig** **von** Seidel, and is similar to the Jacobi method. Though it can be applied to any matrix with non-zero elements on the diagonals, convergence is only guaranteed if the matrix is either strictly diagonally dominant ,or symmetric and positive definite

***Advantages***:

Calculations are simple and so the programming task is lessees. The memory requirement is less. Useful for small systems

***Disadvantages***:

Requires large no. of iterations to reach converge .Not suitable for large systems. Convergence time increases with size of the system

***Convergence:***

* The convergence properties of the Gauss–Seidel method are dependent on the matrix A. Namely, the procedure is known to converge if either

A is symmetric positive-definite, or

A is strictly or irreducibly diagonally dominant.

* ***Algorithms***:

Inputs: A, b

Output: φ

Choose an initial guess φ to the solution

Repeat until convergence

For I from 1 until n do

Σ ← 0

For j from 1 until n do

If j ≠ I then

Σ ← σ + aijφj

End if

End (j-loop)

Φi ← (bi – σ) / aii

End .

**Python Source Code: Gauss Seidel Method**

**# Gauss Seidel Iteration**

**# Defining equations to be solved**

**# in diagonally dominant form**

**f1 = lambda x,y,z: (17-y+2\*z)/20**

**f2 = lambda x,y,z: (-18-3\*x+z)/20**

**f3 = lambda x,y,z: (25-2\*x+3\*y)/20**

**# Initial setup**

**x0 = 0**

**y0 = 0**

**z0 = 0**

**count = 1**

**# Reading tolerable error**

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

**# Implementation of Gauss Seidel Iteration**

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

**Python Program Output: Gauss Seidel Method**

**Enter tolerable error: 0.0001**

**Count x y z**

**1 0.8500 -1.0275 1.0109**

**2 1.0025 -0.9998 0.9998**

**3 1.0000 -1.0000 1.0000**

**4 1.0000 -1.0000 1.0000**

**Solution: x=1.000, y=-1.000 and z = 1.000**

***Gauss Jacobi method:***

In numerical linear algebra, the Jacobi eigenvalue algorithm is an iterative method for the calculation of the eigenvalues and eigenvectors of a real symmetric matrix (a process known as diagonalization). It is named after Carl Gustav Jacob Jacobi, who first proposed the method in 1846,[1] but only became widely used in the 1950s with the advent of computers. Jacobian method or Jacobi method is one the iterative methods for approximating the solution of a system of n linear equations in n variables. The Jacobi iterative method is considered as an iterative algorithm which is used for determining the solutions for the system of linear equations in numerical linear algebra, which is diagonally dominant. In this method, an approximate value is filled in for each diagonal element. Until it converges, the process is iterated. This algorithm was first called the Jacobi transformation process of matrix diagonalization. Jacobi Method is also known as the simultaneous displacement method.

* **Python Source Code: Jacobi Method**

**# Defining equations to be solved**

**# in diagonally dominant form**

**f1 = lambda x,y,z: (17-y+2\*z)/20**

**f2 = lambda x,y,z: (-18-3\*x+z)/20**

**f3 = lambda x,y,z: (25-2\*x+3\*y)/20**

**# Initial setup**

**x0 = 0**

**y0 = 0**

**z0 = 0**

**count = 1**

**# Reading tolerable error**

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

**# Implementation of Jacobi Iteration**

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

**condition = True**

**while condition:**

**x1 = f1(x0,y0,z0)**

**y1 = f2(x0,y0,z0)**

**z1 = f3(x0,y0,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))**

**Python Program Output: Jacobi Method**

**Enter tolerable error: 0.00001**

**Count x y z**

**1 0.8500 -0.9000 1.2500**

**2 1.0200 -0.9650 1.0300**

**3 1.0012 -1.0015 1.0032**

**4 1.0004 -1.0000 0.9996**

**5 1.0000 -1.0001 1.0000**

**6 1.0000 -1.0000 1.0000**

**7 1.0000 -1.0000 1.0000**

**Solution: x=1.000, y=-1.000 and z = 1.000**

### ***1. Example 2x+5y=21, x+2y=8***

**Solve Equations 2x+5y=21,x+2y=8 using Gauss Jacobi method**  
  
**Solution:**  
Total Equations are 2  
  
2*x*+5*y*=21  
  
*x*+2*y*=8  
  
  
From the above equations  
*xk*+1=12(21-5*yk*)  
  
*yk*+1=12(8-*xk*)  
  
Initial gauss (*x*,*y*)=(0,0)  
  
Solution steps are  
**1*st*** **Approximation**  
  
*x*1=12[21-5(0)]=12[21]=10.5  
  
*y*1=12[8-(0)]=12[8]=4  
  
**2*nd*** **Approximation**  
  
*x*2=12[21-5(4)]=12[1]=0.5  
  
*y*2=12[8-(10.5)]=12[-2.5]=-1.25  
  
**3*rd*** **Approximation**  
  
*x*3=12[21-5(-1.25)]=12[27.25]=13.625  
  
*y*3=12[8-(0.5)]=12[7.5]=3.75  
  
**4*th*** **Approximation**  
  
*x*4=12[21-5(3.75)]=12[2.25]=1.125  
  
*y*4=12[8-(13.625)]=12[-5.625]=-2.8125  
  
**5*th*** **Approximation**  
  
*x*5=12[21-5(-2.8125)]=12[35.0625]=17.5312  
  
*y*5=12[8-(1.125)]=12[6.875]=3.4375  
  
**6*th*** **Approximation**  
  
*x*6=12[21-5(3.4375)]=12[3.8125]=1.9062  
  
*y*6=12[8-(17.5312)]=12[-9.5312]=-4.7656  
  
**7*th*** **Approximation**  
  
*x*7=12[21-5(-4.7656)]=12[44.8281]=22.4141  
  
*y*7=12[8-(1.9062)]=12[6.0938]=3.0469  
  
  
Equations are Divergent...

|  |  |  |
| --- | --- | --- |
| **Iteration** | **x** | **y** |
| 1 | 10.5 | 4 |
| 2 | 0.5 | -1.25 |
| 3 | 13.625 | 3.75 |
| 4 | 1.125 | -2.8125 |
| 5 | 17.5312 | 3.4375 |
| 6 | 1.9062 | -4.7656 |
| 7 | 22.4141 | 3.0469 |

***Sor method***

In numerical linear algebra, the method of successive over-relaxation (SOR) is a variant of the Gauss–Seidel method for solving a linear system of equations, resulting in faster convergence. A similar method can be used for any slowly converging iterative process.

***Algorithm SIR Method:***

Set k = 1

Step 2 While (k ≤ N) do Steps 3–6:Step 3 For I = 1, . . . , nSet xi = (1 – ω)XOi +AiiPi−1

J=1Aijxj –Pn

J=i+1

aijXOj + bii

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

STOP (The procedure was successful)

Step 5 Set k =k+1

---------------------------------------------------------------------------------------------

***Chapter #3***

1. ***Iterative techniques in matrix algebra:***

* ***3.1:Norm of vector***
* **3.2: Eigenvalues and Eigenvectors**
* ***Norm of a vector:***

The length of the vector is referred to as the vector norm or the vector’s magnitude.

*The length of a vector is a nonnegative number that describes the extent of the vector in space, and is sometimes referred to as the vector’s magnitude or the norm.*

The length of the vector is always a positive number, except for a vector of all zero values. It is calculated using some measure that summarizes the distance of the vector from the origin of the vector space. For example, the origin of a vector space for a vector with 3 elements is (0, 0, 0).

# ***Representation of a norm***

The norm of any vector **x** is denoted by a double bar around it and is written as



## *Properties of norm*

Consider two vectors **x** and **y** having the same size and a scalar- 𝝀. A function is considered a norm if and only if it satisfies the following properties:

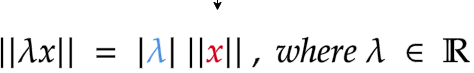
* **Nonnegativity —**Itshould always be nonnegative.



* **Definiteness —**It is zero if and only if the vector is zero, i.e., zero vector.



* **Triangle Inequality**— The norm of a sum of two vectors is no more than the sum of their norms.
* **Homogeneity-**Multiplying a vector by a scalar multiplies the norm of the vector by the absolute value of the scalar.



* ***Eigenvalues and Eigenvectors***

Eigenvalues are the special set of scalars associated with the system of linear equations. It is mostly used in matrix equations. ‘Eigen’ is a German word that means ‘proper’ or ‘characteristic’. Therefore, the term eigenvalue can be termed as characteristic value, characteristic root, proper values or latent roots as well. In simple words, the eigenvalue is a scalar that is used to transform the eigenvector. The basic equation is

**Ax = λx**

The number or scalar value “**λ”**is an eigenvalue of A.

In Mathematics, an eigenvector corresponds to the real non zero eigenvalues which point in the direction stretched by the transformation whereas eigenvalue is considered as a factor by which it is stretched. In case, if the eigenvalue is negative, the direction of the transformation is negative.

For every real matrix,  there is an eigenvalue. Sometimes it might be complex. The existence of the eigenvalue for the complex matrices is equal to the fundamental theorem of algebra.

Eigenvectors are the vectors (non-zero) that do not change the direction when any linear transformation is applied. It changes by only a scalar factor. In a brief, we can say, if A is a linear transformation from a vector space V and **x** is a vector in V, which is not a zero vector, then v is an eigenvector of A if A(X) is a scalar multiple of **x**.

An **Eigenspace**of vector **x** consists of a set of all eigenvectors with the equivalent eigenvalue collectively with the zero vector. Though, the zero vector is not an eigenvector.

Let us say A is an “n × n” matrix and λ is an eigenvalue of matrix A, then **x**, a non-zero vector, is called as eigenvector if it satisfies the given below expression;

A**x** = λ**x**

**x** is an eigenvector of A corresponding to eigenvalue, λ.

***Properties of Eigenvalues***

* Eigenvectors with Distinct Eigenvalues are Linearly Independent
* Singular Matrices have Zero Eigenvalues
* If A is a square matrix, then λ = 0 is not an eigenvalue of A
* **For a scalar multiple of a matrix:** If A is a square matrix and λ is an eigenvalue of A. Then, aλ is an eigenvalue of aA.
* **For Matrix powers:** If A is square matrix and λ is an eigenvalue of A and n≥0 is an integer, then λn is an eigenvalue of An.
* **For polynomials of matrix:** If A is a square matrix, λ is an eigenvalue of A and  p(x) is a polynomial in variable x, then p(λ) is the eigenvalue of matrix p(A).
* **Inverse Matrix:** If A is a square matrix, λ is an eigenvalue of A, then λ-1 is an eigenvalue of A-1
* **Transpose matrix:**If A is a square matrix, λ is an eigenvalue of A, then λ is an eigenvalue of At

***Chapter #4***

***4.Interpolation*** :

4.1: Newton forward

4.2: Newton backward

4.3:Central difference

***Interpolation:***

|  |
| --- |
| *p*=*x*-*x*0*h* *y*(*x*)=*y*0+*p*Δ*y*0+*p*(*p*-1)2!⋅Δ2*y*0+*p*(*p*-1)(*p*-2)3!⋅Δ3*y*0+*p*(*p*-1)(*p*-2)(*p*-3)4!⋅Δ4*y*0+... |

**Examples**  
**1. Find** Solution using Newton's **Forward Difference formula**

|  |  |
| --- | --- |
| x | f(x) |
| 1891 | 46 |
| 1901 | 66 |
| 1911 | 81 |
| 1921 | 93 |
| 1931 | 101 |

**x = 1895**

**Solution:**  
The value of table for *x* and *y*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **x** | 1891 | 1901 | 1911 | 1921 |  |
| **y** | 193146 | 66 | 81 | 93 | 101 |

Newton's forward difference interpolation method to find solution  
  
Newton's forward difference table is

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **x** | **y** | **Δ*y*** | **Δ2*y*** | **Δ3*y*** |  | **Δ4*y*** |
|  | **46** |  |  |  |  |  |
|  |  | **20** | **1891** |  |  |  |
| 1901 | 66 |  | **-5** |  |  |  |
|  |  | 15 |  | **2** |  |  |
| 1911 | 81 |  | -3 |  |  | **-3** |
|  |  | 12 |  | -1 |  |  |
| 1921 | 93 |  | -4 |  |  |  |
|  |  | 8 |  |  |  |  |
| 1931 | 101 |  |  |  |  |  |

The value of *x* at you want to find the *f*(*x*):*x*=1895  
  
*h*=*x*1-*x*0=1901-1891=10  
  
*p*=*x*-*x*0*h*=1895-189110=0.4  
  
Newton's forward difference interpolation formula is  
*y*(*x*)=*y*0+*p*Δ*y*0+*p*(*p*-1)2!⋅Δ2*y*0+*p*(*p*-1)(*p*-2)3!⋅Δ3*y*0+*p*(*p*-1)(*p*-2)(*p*-3)4!⋅Δ4*y*0  
  
*y*1(895)=46+0.4×20+0.4(0.4-1)2×-5+0.4(0.4-1)(0.4-2)6×2+0.4(0.4-1)(0.4-2)(0.4-3)24×-3  
  
*y*(1895)=46+8+0.6+0.128+0.1248  
  
*y*(1895)=54.8528

**Python Source Code: Forward Difference Table**

**# Reading number of unknowns**

**n = int(input('Enter number of data points: '))**

**# Making numpy array of n & n x n size and initializing**

**# to zero for storing x and y value along with differences of y**

**x = np.zeros((n))**

**y = np.zeros((n,n))**

**# Reading data points**

**print('Enter data for x and y: ')**

**for i in range(n):**

**x[i] = float(input( 'x['+str(i)+']='))**

**y[i][0] = float(input( 'y['+str(i)+']='))**

**# Generating forward difference table**

**for i in range(1,n):**

**for j in range(0,n-i):**

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

**print('\nFORWARD DIFFERENCE TABLE\n');**

**for i in range(0,n):**

**print('%0.2f' %(x[i]), end='')**

**for j in range(0, n-i):**

**print('\t\t%0.2f' %(y[i][j]), end='')**

**print()**

**Python Output: Forward Difference Table**

**Enter number of data points: 5**

**Enter data for x and y:**

**x[0]=40**

**y[0]=31**

**x[1]=50**

**y[1]=73**

**x[2]=60**

**y[2]=124**

**x[3]=70**

**y[3]=159**

**x[4]=80**

**y[4]=190**

FORWARD DIFFERENCE TABLE

40.00 31.00 42.00 9.00 -25.00 37.00

50.00 73.00 51.00 -16.00 12.00

60.00 124.00 35.00 -4.00

70.00 159.00 31.00

80.00 190.00

|  |
| --- |
| Newton's Backward Difference formula |
| *p*=*x*-*xnh* *y*(*x*)=*yn*+*p*∇*yn*+*p*(*p*+1)2!⋅∇2*yn*+*p*(*p*+1)(*p*+2)3!⋅∇3*yn*+*p*(*p*+1)(*p*+2)(*p*+3)4!⋅∇4*yn*+... |

**Examples**  
**1. Find Solution using Newton's Backward Difference formula**

|  |  |
| --- | --- |
| x | f(x) |
| 1891 | 46 |
| 1901 | 66 |
| 1911 | 81 |
| 1921 | 93 |
| 1931 | 101 |

**x = 1925**

**Solution:**  
The value of table for *x* and *y*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **x** | 1891 | 1901 | 1911 | 1921 | 1931 |
| **y** | 46 | 66 | 81 | 93 | 101 |

Newton's backward difference interpolation method to find solution  
  
Newton's backward difference table is

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

The value of x at you want to find the *f*(*x*):*x*=1925  
  
*h*=*x*1-*x*0=1901-1891=10  
  
*p*=*x*-*xnh*=1925-193110=-0.6  
  
Newton's backward difference interpolation formula is  
*y*(*x*)=*yn*+*p*∇*yn*+*p*(*p*+1)2!⋅∇2*yn*+*p*(*p*+1)(*p*+2)3!⋅∇3*yn*+*p*(*p*+1)(*p*+2)(*p*+3)4!⋅∇4*yn*  
  
*y*(1925)=101+(-0.6)×8+-0.6(-0.6+1)2×-4+-0.6(-0.6+1)(-0.6+2)6×-1+-0.6(-0.6+1)(-0.6+2)(-0.6+3)24×-3  
  
*y*(1925)=101-4.8+0.48+0.056+0.1008  
  
*y*(1925)=96.8368  
  
  
Solution of newton's backward interpolation method *y*(1925)=96.8368

Python Source Code: Forward Difference Table

# Reading number of unknowns

n = int(input('Enter number of data points: '))

# Making numpy array of n & n x n size and initializing

# to zero for storing x and y value along with differences of y

x = np.zeros((n))

y = np.zeros((n,n))

# Reading data points

print('Enter data for x and y: ')

for i in range(n):

x[i] = float(input( 'x['+str(i)+']='))

y[i][0] = float(input( 'y['+str(i)+']='))

# Generating forward difference table

for i in range(1,n):

for j in range(0,n-i):

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

print('\nFORWARD DIFFERENCE TABLE\n');

for i in range(0,n):

print('%0.2f' %(x[i]), end='')

for j in range(0, n-i):

print('\t\t%0.2f' %(y[i][j]), end='')

print()

Python Output: Forward Difference Table

Enter number of data points: 5

Enter data for x and y:

x[0]=40

y[0]=31

x[1]=50

y[1]=73

x[2]=60

y[2]=124

x[3]=70

y[3]=159

x[4]=80

y[4]=190

FORWARD DIFFERENCE TABLE

40.00 31.00 42.00 9.00 -25.00 37.00

50.00 73.00 51.00 -16.00 12.00

60.00 124.00 35.00 -4.00

70.00 159.00 31.00

80.00 190.00

**Central** **difference** **interpolation**: So, in Newton's backward and forward difference interpolation formula, it is learned that they can be used for interpolation around the beginning and end of a table of data. Now let’s introduce the central difference interpolation formula that is best applicable for determining the values near the middle of the table.

***5.Numerical integration:***

**5.1:Trapizodal rule**

**5.2:Simpson one third rule**

**5.3 simpson 3/8 rule**

**Newton divided difference:**

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

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

Then the first divided difference is given by

F[x\_0, x\_1]=\frac{f(x\_1)-f(x\_0)}{x\_1-x\_0}

The second divided difference is given by

F[x\_0, x\_1, x\_2]=\frac{f[x\_1, x\_2]-f[x\_0, x\_1]}{x\_2-x\_0}

And so on…

Divided differences are symmetric with respect to the arguments i.e independent of the order of arguments.

So,

F[x0, x1]=f[x1, x0]

F[x0, x1, x2]=f[x2, x1, x0]=f[x1, x2, x0]

By using first divided difference, second divided difference as so on .A table is formed which is called the divided difference table

Divided difference table:

***Advantages of NEWTON’S DIVIDED DIFFERENCE INTERPOLATION FORMULA***

* These are useful for interpolation.
* Through difference table, we can find out the differences in higher order.
* Differences at each stage in each of the columns are easily measured by subtracting the previous value from its immediately succeeding value.
* The differences are found out successively between the two adjacent values of the y variable till the ultimate difference vanishes or become a constant.

***Trapezoidal Rule:***

, “Trapezoidal Rule” is one of the important integration rules. The name trapezoidal is because when the area under the curve is evaluated, then the total area is divided into small trapezoids instead of rectangles. This rule is used for approximating the definite integrals where it uses the linear approximations of the functions.

The trapezoidal rule is mostly used in the numerical analysis process. To evaluate the definite integrals, we can also use Riemann Sums, where we use small rectangles to evaluate the area under the curve.

***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. This rule takes the average of the left and the right sum.

The Trapezoidal Rule does not give accurate value as Simpson’s Rule when the underlying function is smooth. It is because Simpson’s Rule uses the quadratic approximation instead of linear approximation. Both Simpson’s Rule and Trapezoidal Rule give the approximation value, but Simpson’s Rule results in even more accurate approximation value of the integrals.

**Trapezoidal Rule Formula:**

Let f(x) be a continuous function on the interval [a, b]. Now divide the intervals [a, b] into n equal subintervals with each of width,

Δx = (b-a)/n, Such that a = x0 < x1< x2< x3<…..<xn = b

Then the Trapezoidal Rule formula for area approximating the definite integral

Is given by:

Where, xi = a+iΔx

If n →∞, R.H.S of the expression approaches the definite integral

Go through the below given Trapezoidal Rule example.

***Example 1:***

Approximate the area under the curve y = f(x) between x =0 and x=8 using Trapezoidal Rule with n = 4 subintervals. A function f(x) is given in the table of values.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **x** | **0** | **2** | **4** | **6** | **8** |
| **F(x)** | **3** | **7** | **11** | **9** | **3** |

***Solution:***

The Trapezoidal Rule formula for n= 4 subintervals is given as:

T4 =(Δx/2)[f(x0)+ 2f(x1)+ 2f(x2)+2f(x3) + f(x4)]

Here the subinterval width Δx = 2.

Now, substitute the values from the table, to find the approximate value of the area under the curve.

A≈ T4 =(2/2)[3+ 2(7)+ 2(11)+2(9) + 3]

A≈ T4 = 3 + 14 + 22+ 18+3 = 60

Therefore, the approximate value of area under the curve using Trapezoidal Rule is 60.

***Example 2:***

Approximate the area under the curve y = f(x) between x =-4 and x= 2 using Trapezoidal Rule with n = 6 subintervals. A function f(x) is given in the table of values.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x | \_4 | -3 | -2 | -1 | 0 |
| F(x) | 0 | 4 | 5 | 3 | 10 |

***Solution:***

The Trapezoidal Rule formula for n= 6 subintervals is given as:

T6 =(Δx/2)[f(x0)+ 2f(x1)+ 2f(x2)+2f(x3) + 2f(x4)+2f(x5)+ f(x6)]

Here the subinterval width Δx = 1.

Now, substitute the values from the table, to find the approximate value of the area under the curve.

A≈ T6 =(1/2)[0+ 2(4)+ 2(5)+2(3) + 2(10)+2(11) +2]

A≈ T6 =(½) [ 8 + 10 + 6+ 20 +22 +2 ] = 68/2 = 34

Therefore, the approximate value of area under the curve using Trapezoidal rule

***Simpson ‘s rule:***

***Simpson 1/3 rule:***

***Simpson’s 1/3rd rule is an extension of the trapezoidal rule in which the integrand is approximated by a second-order polynomial. Simpson rule can be derived from the various way using Newton’s divided difference polynomial, Lagrange polynomial and the method of coefficients. Simpson’s 1/3 rule is defined by:***

|  |  |
| --- | --- |
| ∫abf(x) dx = h/3[(y0 + yn)+4(y1 + y3 + y5 + …. + yn-1)+2(y2 + y4 + y6 + ….. + yn-2)] |  |

***Simpson ‘s rule 3/8:***

3/8 rule, also called Simpson's second rule, is another method for numerical integration proposed by Thomas Simpson. It is based upon a cubic interpolation rather than a quadratic interpolation. Simpson's 3/8 rule is as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **Formula**   |  | | --- | | 3. Simpsons 38 Rule ∫*ydx*=3*h*8(*y*0+2(*y*3+*y*6+...+*yn*-3)+3(*y*1+*y*2+*y*4+*y*5+...+*yn*-2+*yn*-1)+*yn*) |   **Examples** **1. Find Solution using Simpson's 3/8 rule**   |  |  | | --- | --- | | x | f(x) | | 1.4 | 4.0552 | | 1.6 | 4.9530 | | 1.8 | 6.0436 | | 2.0 | 7.3891 | | 2.2 | 9.0250 |   **Solution:** The value of table for *x* and *y*   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **x** | 1.4 | 1.6 | 1.8 | 2 | 2.2 | | **y** | 4.0552 | 4.953 | 6.0436 | 7.3891 | 9.025 |   Using Simpson's 38 Rule  ∫*ydx*=3*h*8[(*y*0+*y*4)+2(*y*3)+3(*y*1+*y*2)]  ∫*ydx*=3×0.28[(4.0552+9.025)+2×(7.3891)+3×(4.953+6.0436)]  ∫*ydx*=3×0.28[(4.0552+9.025)+2×(7.3891)+3×(10.9966)]  ∫*ydx*=4.5636  Solution by Simpson's 38 Rule is 4.5636  **2. Find Solution using Simpson's 3/8 rule**   |  |  | | --- | --- | | x | f(x) | | 0.0 | 1.0000 | | 0.1 | 0.9975 | | 0.2 | 0.9900 | | 0.3 | 0.9776 | | 0.4 | 0.8604 |   **Solution:** The value of table for *x* and *y*   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **x** | 0 | 0.1 | 0.2 | 0.3 | 0.4 | | **y** | 1 | 0.9975 | 0.99 | 0.9776 | 0.8604 |   Using Simpson's 38 Rule  ∫*ydx*=3*h*8[(*y*0+*y*4)+2(*y*3)+3(*y*1+*y*2)]  ∫*ydx*=3×0.18[(1+0.8604)+2×(0.9776)+3×(0.9975+0.99)]  ∫*ydx*=3×0.18[(1+0.8604)+2×(0.9776)+3×(1.9875)]  ∫*ydx*=0.36668  Solution by Simpson's 38 Rule is 0.36668 | | | | |  | | --- | |  | | | |  | | --- | |  | | | |
|  |
|  |