Contents

Bisection Method

The Bisection Method is a root-finding method that applies to any continuous function for which one knows two values with opposite signs. The method consists of repeatedly bisecting the interval defined by these values and then selecting the subinterval in which the function changes sign, and therefore must contain a root. -(Wikipedia)

Algorithm:

The steps for implementing the Bisection Method are as follows:

1. Decide an interval $[x_1, x_2]$ and tolerance E such that $f(x_1)$ and $f(x_2)$ have opposite signs.
2. Calculate the midpoint x_0 of the interval:

$$x_0 = \frac{x_1 + x_2}{2}$$

3. If the absolute value of $f(x_0)$ is smaller than the given tolerance E, stop the iteration.
4. If the function changes sign between $f(x_1)$ and $f(x_0)$, set $x_2 = x_0$; otherwise, set $x_1 = x_0$.
5. Repeat the process until the desired accuracy is achieved.

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\Bisection.py
-4 -3
14x2+41x-14
Root approximately at x = -3.2375
```

False Position Method

The False Position Method is the trial and error technique of using test ("false") values for the variable and then adjusting the test value according to the outcome. This is sometimes also referred to as "guess and check". -(Wikipedia)

Algorithm

The steps for implementing the False Position Method are as follows:

1. Identify an interval $[x_1, x_2]$ and tolerance E such that $f(x_1)$ and $f(x_2)$ have opposite signs.
2. Calculate the next approximation using the formula:

$$x_0 = x_1 - \frac{f(x_1) \cdot (x_2 - x_1)}{f(x_2) - f(x_1)}$$

3. If the absolute value of $f(x_0)$ is smaller than the given tolerance E, stop the iteration.
4. Update the approximations: if $f(x_0)$ and $f(x_1)$ have opposite signs, set $x_2 = x_0$; otherwise, set $x_1 = x_0$.
5. Repeat until the desired accuracy is achieved.

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\FalsePosition.py
-4 -3
14x2+41x-14
Root approximately at x = -3.2374
```

Newton-Raphson Method

The Newton-Raphson method is an iterative numerical method used to find the roots of a real-valued function. It is an algorithm to approximate the roots of zeros of the real-valued functions, using guess for the first iteration x_0 and then approximating the next iteration x_1 which is close to roots. -(GeeksForGeeks)

Algorithm

The steps for implementing the Newton-Raphson Method are as follows:

1. Given a function $f(x)$, start with an initial guess x_1 for the root.
2. Calculate the derivative of the function, $f'(x)$, and use the following formula to calculate the next approximation:

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

3. Repeat the process until the difference between successive approximations is smaller than the given tolerance E, i.e., until $|x_{n+1} - x_n| < E$.
4. The approximation x_n is considered the root of the function.

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\NewtonRaphson.py
x2-3x+2
Approximated root 1.0
```

Secant Method

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, so it is considered a quasi-Newton method. -(Wikipedia)

Algorithm

The steps for implementing the Secant Method are as follows:

1. Select two initial approximations, x_1, x_2 and tolerance E, for the root.
2. Calculate the next approximation x_3 using the secant formula:

$$x_3 = x_2 - \frac{f(x_2) \cdot (x_2 - x_1)}{f(x_2) - f(x_1)}$$

3. If the absolute difference between the successive approximations, $|x_2 - x_1|$, is smaller than the given tolerance E, stop the iteration.
4. Update the approximations: set $x_1 = x_2$ and $x_2 = x_3$.
5. Repeat the process until the desired accuracy is achieved.

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\"Secant.py
x2 - 4x - 10
4 2
5.741658764379768
```

Fixed-Point Iteration Method

The fixed point iteration method is used to find an approximate solution to algebraic and transcendental equations. 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, the given function is algebraically converted in the form of g(x) = x. -(Byju's)

Algorithm

The steps for implementing the Fixed-Point Iteration Method are as follows:

1. Rearrange the given equation $f(x) = 0$ into an iterative form:

$$x = g(x)$$

2. Start with an initial guess x_0.

3. Evaluate the function $g(x)$ at x_0 to get the next approximation x_1.

4. Repeat the iteration:

$$x_{n+1} = g(x_n)$$

5. Stop the iteration when the absolute difference between successive approximations is smaller than the given tolerance E, i.e., when $|x_{n+1} - x_n| < E$.

6. The final approximation x_n is considered the root of the equation.

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\FixedPointIteration.py
x**2 + x - 2
Root: 1.0, Iterations: 5
```

Jacobi Method

The Jacobi Method (a.k.a. the Jacobi iteration method) is an iterative algorithm for determining the solutions of a strictly diagonally dominant system of linear equations. Each diagonal element is solved for, and an approximate value is plugged in. The process is then iterated until it converges. This algorithm is a stripped-down version of the Jacobi transformation method of matrix diagonalization. -(Wikipedia)

Algorithm

The steps for implementing the Jacobi Method are as follows:

1. Rewrite the system of equations in terms of each variable:

$$x_1 = \frac{d_1 - b_1 x_2 - c_1 x_3}{a_1}, \quad x_2 = \frac{d_2 - a_2 x_1 - c_2 x_3}{b_2}, \quad x_3 = \frac{d_3 - a_3 x_1 - b_3 x_2}{c_3}$$

2. Start with an initial guess for all unknowns, typically $x_1 = 0$, $x_2 = 0$, and $x_3 = 0$.

3. At each iteration:

   - Compute the new values of x_1, x_2, and x_3 using the equations derived in Step 1.
   - Use only the values from the previous iteration for all variables when performing updates.

4. Repeat the process until:

   - The changes in x_1, x_2, and x_3 are smaller than a specified tolerance, or
   - The maximum number of iterations is reached.

Input and Output

Below is an example of the input and output for the Jacobi Method:

```
(env) D:\programs\JacobiMethod.py
4x-1y-1z=5
-1x+3y-1z=6
-1x-1y+5z=-4
x = 1.9565, y = 2.6956, z = 0.1304
```

Gauss-Seidel Method

The Gauss-Seidel method is an iterative algorithm used to solve systems of linear equations, particularly for matrices that are strictly diagonally dominant or symmetric and positive definite. In this method, each equation is solved sequentially for each variable, and the updated values are immediately used in the next steps within the same iteration. This contrasts with the Jacobi method, where the updated values are only used in the next iteration. -(Wikipedia)

Algorithm

The steps for implementing the Gauss-Seidel Method are as follows:

1. Rewrite the system of equations in terms of the variables:

$$x_1 = \frac{d_1 - b_1x_2 - c_1x_3}{a_1}, \quad x_2 = \frac{d_2 - a_2x_1 - c_2x_3}{b_2}, \quad x_3 = \frac{d_3 - a_3x_1 - b_3x_2}{c_3}$$

2. Start with an initial guess for all the unknowns, say $x_1 = 0$, $x_2 = 0$, and $x_3 = 0$.

3. For each variable, use the equation from the previous step to update its value based on the most recent values of the other variables.

4. Repeat the process until the values of x_1, x_2, and x_3 converge to a desired tolerance, or a set maximum number of iterations is reached.

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\GaussSeidel.py
4x-1y-1z=5
-1x+3y-1z=6
-1x-1y+5z=-4
x = 1.9565217391304346, y = 2.6956521739130435, z = 0.13043478260869562
```

Basic Gaussian Elimination

Gaussian Elimination is a method used to solve systems of linear equations by transforming the augmented matrix of the system into row-echelon form and then back-substituting to find the unknowns. This method is widely used in numerical analysis and computational mathematics. -(Wikipedia)

Algorithm

The steps for implementing the Gaussian Elimination Method are as follows:

1. Start with the augmented matrix representing the system of linear equations.
2. Perform forward elimination to transform the matrix into an upper triangular form:
    - For each row, use the pivot element (the leading coefficient) to eliminate the coefficients below it in the same column.
    - Repeat this step for all rows.
3. Perform back substitution to solve for the unknowns:
    - Start with the last row and solve for the unknown.
    - Substitute this value into the equations of the previous rows to find the remaining unknowns.
4. The final result will be the values of the unknowns, such as x, y, and z.

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\GaussianElimination.py
3
x1 + x2 + x3 = 9
2x1 + 3x2 + 3x3 = 15
x1 + 2x2 + 3x3 = 10
[ [ 9, 3, 5, 10], [ 15, 8, 9, 20], [ 10, 4, 6, 15]]
[ [ 9, 3, 5, 10], [ 0, 2, 3, 5], [ 0, 0, 1, 5]]
[ [ 9, 3, 5, 10], [ 0, 2, 3, 5], [ 0, 0, 1, 5]]
Root values:
x = 1.0, y = 2.0, z = 3.0
```

Gauss-Jordan Elimination Method

The Gauss-Jordan elimination method is an algorithm for solving systems of linear equations by transforming the augmented matrix into its reduced row echelon form (RREF). This method involves two main stages: forward elimination and back substitution.

Algorithm

The steps for implementing the Gauss-Jordan Elimination Method are as follows:

1. Start with the augmented matrix representing the system of linear equations.
2. Perform forward elimination to transform the matrix into an upper triangular form:
    - Use the pivot element (the leading coefficient) to eliminate the coefficients below it in the same column.
    - Repeat this step for all rows.
3. Perform additional row operations to eliminate the elements above each pivot element, turning the matrix into the reduced row echelon form.
4. After reducing the matrix, the solutions can be found by directly reading off the values of the variables from the final matrix.

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\GaussJordan.py
3
0x1 + 6x2 + 0x3  = 16
2x1 + 4x2 + 3x3 = 13
1x1 + 3x2 + 2x3 = 9

x = 1.6666666666666667
y = 2.6666666666666665
z = -0.3333333333333333
```

Least-Squares Regression

The Least Squares Method is a statistical technique used to find the best-fitting linear relationship between a dependent variable and one or more independent variables. It minimizes the sum of the squared differences between the observed values and the values predicted by the linear equation. The method is commonly used in regression analysis to estimate the coefficients of the regression line. -(Wikipedia)

Algorithm

The steps to implement linear regression using the Least Squares Method are as follows:

1. Given the data points (x_i, y_i), calculate the following summations:

$$\sum x_i, \quad \sum y_i, \quad \sum x_i^2, \quad \sum x_i y_i$$

2. Use the formulas for the slope b and the intercept a:

$$b = \frac{n\sum x_i y_i - (\sum x_i)(\sum y_i)}{n\sum x_i^2 - (\sum x_i)^2}, \quad a = \frac{\sum y_i}{n} - b\frac{\sum x_i}{n}$$

   Where n is the number of data points.

3. Finally, the linear regression equation is given by:

$$y = a + bx$$

4. Output the values of a and b as the coefficients of the regression line.

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\LeastSquareRegression.py
n: 5
x, y: 1 3
x, y: 2 4
x, y: 3 5
x, y: 4 6
x, y: 5 8

y = 1.6000000000000005 + 1.2x
```

Fitting Transcendental Curve

Fitting a transcendental curve involves finding the best-fitting curve for a set of data points that follow a non-linear relationship. The curve can be represented by a transcendental function, such as an exponential, logarithmic, or power function. The goal is to estimate the parameters of the function that minimize the sum of the squared differences between the observed and predicted values.

Algorithm

The steps for implementing Log-Log Regression are as follows:

1. Take the logarithm of the x and y values.
2. Calculate the following summations:

$$\sum \log(x), \quad \sum \log(y), \quad \sum \log(x)^2, \quad \sum \log(x) \log(y)$$

3. Use the formulas for the slope b and the intercept a:

$$b = \frac{n \sum \log(x) \log(y) - (\sum \log(x))(\sum \log(y))}{n \sum \log(x)^2 - (\sum \log(x))^2}$$

$$a = \exp\left(\frac{\sum \log(y)}{n} - b\frac{\sum \log(x)}{n}\right)$$

4. The final regression equation is:

$$y = a \cdot x^b$$

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\FittingTranscendentalCurve.py
n: 5
x, y: 1 0.5
x, y: 2 2
x, y: 3 4.5
x, y: 4 8
x, y: 5 12.5
y = 0.49999999999999983 x ^ 2.0000000000000004
```

Fitting Polynomial

Polynomial fitting finds a polynomial that best represents a set of data points by minimizing the error between the actual data and the polynomial's predicted values.

Algorithm

The steps to implement polynomial fitting using the Least Squares Method are as follows:

1. For the given data points (x_i, y_i), calculate the following summations:

$$\sum x_i, \quad \sum y_i, \quad \sum x_i^2, \quad \sum x_i^3, \quad \sum x_i^4, \quad \sum x_i y_i, \quad \sum x_i^2 y_i$$

2. Set up a system of linear equations for the coefficients of the polynomial (e.g., a_0, a_1, a_2 for a quadratic polynomial):

$$\begin{aligned} na_0 + \sum x_i a_1 + \sum x_i^2 a_2 &= \sum y_i, \\ \sum x_i a_0 + \sum x_i^2 a_1 + \sum x_i^3 a_2 &= \sum x_i y_i, \\ \sum x_i^2 a_0 + \sum x_i^3 a_1 + \sum x_i^4 a_2 &= \sum x_i^2 y_i \end{aligned}$$

3. Solve for the coefficients $[a_0, a_1, a_2]$ using $A \cdot c = b$, where $c = [a_0, a_1, a_2]$.

4. Construct the polynomial equation:

$$y = a_0 + a_1 x + a_2 x^2$$

Input and Output

Below is an example of the code execution:

```
(env) D:\programs\FittingPolynomial.py
4
1 6
2 11
3 18
4 27
The linear equation is:
4a1 + 10.0a2 + 30.0a3 = 62.0
10.0a1 + 30.0a2 + 100.0a3 = 190.0
30.0a1 + 100.0a2 + 354.0a3 = 644.0

The coefficients of the polynomial are:
a1 (constant term) = 3.0000000000000844
a2 (linear term)   = 1.9999999999999276
a3 (quadratic term) = 1.0000000000000133

The polynomial equation is:
y = 3.0000000000000844 + 1.9999999999999276x + 1.0000000000000133x^2
```

Euler's Method

The Euler method is a first-order numerical procedure for solving ordinary differential equations (ODEs) with a given initial value. It is the most basic explicit method for numerical integration of ordinary differential equations and is the simplest Runge-Kutta method. - (Wikipedia)

Algorithm

The following steps outline the Euler's method algorithm:

1. Start with an initial value of y_0 at x_0.
2. Define the step size h, which is the difference between consecutive x-values.
3. For each step, calculate the new value of y using the formula:

$$y_{n+1} = y_n + h \cdot f(x_n, y_n)$$

   where $f(x_n, y_n)$ is the function that represents the derivative $\frac{dy}{dx}$.
4. Update $x_{n+1} = x_n + h$.
5. Repeat the process until the desired range of x is reached.

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\FittingTranscendentalCurve.py
n: 5
x, y: 1 0.5
x, y: 2 2
x, y: 3 4.5
x, y: 4 8
x, y: 5 12.5
y = 0.49999999999999983 x ^ 2.0000000000000004
```

Cramer's Rule Algorithm

Cramer's rule is an explicit formula for the solution of a system of linear equations with as many equations as unknowns, valid whenever the system has a unique solution. It expresses the solution in terms of the determinants of the (square) coefficient matrix and of matrices obtained from it by replacing one column by the column vector of right-sides of the equations. - (Wikipedia)

Algorithm

Given a system of linear equations:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned}$$

The solution can be found by using the following steps:

1. Start by representing the system of equations as a matrix, and extract the coefficients for each variable and the constant terms:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

2. Compute the determinant of the coefficient matrix $D = \det(A)$.

3. For each unknown x_i, replace the i-th column of the matrix A with the column vector $\mathbf{b}$, and compute the determinant of the resulting matrix.

$$D_{a1} = \det(A_1), \quad D_{a2} = \det(A_2), \quad D_{a3} = \det(A_3)$$

4. Finally, solve for each unknown by dividing the determinant of the modified matrix by the determinant of the original matrix:

$$x_1 = \frac{D_{a1}}{D}, \quad x_2 = \frac{D_{a2}}{D}, \quad x_3 = \frac{D_{a3}}{D}$$

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\Cramer.py
3
4x1 + 10x2 + 30x3  = 62
10x1 + 30x2 + 100x3 = 190
30x1 + 100x2 + 354x3 = 644
a1 = 3.0 a2 = 2.0 a3 = 1.0
```

Trapezoidal Rule

The trapezoidal rule is one of the fundamental rules of integration which is used to define the basic definition of integration. It is a widely used rule and named so because it gives the area under the curve by dividing the curve into small trapezoids instead of rectangles. -(GeeksForGeeks)

Algorithm

To compute the integral of a function $f(x)$ over the interval $[a, b]$, using n subintervals, follow these steps:

1. Start by defining the function $f(x)$ that you want to integrate. For example:

$$f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$$

2. Define the limits of integration a and b, and the number of subintervals n.

$$a = 0, \quad b = 0.8, \quad n = 2000$$

3. Compute the step size h, which is the width of each subinterval:

$$h = \frac{b - a}{n}$$

4. Generate the list of x-values by dividing the interval $[a, b]$ into n subintervals:

$$x_i = a + i \cdot h \quad \text{for} \quad i = 0, 1, 2, \ldots, n$$

5. Use the Trapezoidal Rule formula to compute the approximate integral:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left(f(x_0) + f(x_n) + 2\sum_{i=1}^{n-1} f(x_i)\right)$$

6. Finally, calculate the result using the formula above, where $f(x_0)$ and $f(x_n)$ are the values of the function at the endpoints of the interval, and the sum of the intermediate values $f(x_i)$ is multiplied by 2.

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\Trapezoidal.py
0.2 + 25x - 200x2 + 675x3 - 900x4 + 400x5
0 2 2000
Integral result using trapezoidal rule: 723.3342083332601
```

Simpson's 1/3 Rule

In Simpson's rule, we use three equally spaced points for finding a fitting polynomial and the endpoints are two of them. Thus Simpsons rule is also called the 3 point closed rule. It is used to approximate the integral of a function over a given interval. The rule is more accurate than the trapezoidal rule and is often used when a higher degree of accuracy is required. -(Wikipedia)

Algorithm

- Define the function $f(x)$ and the interval $[a, b]$.
- Set n (number of subintervals) to be even.
- Calculate the step size $h = \frac{b-a}{n}$.
- Generate the x-values: $x_i = a + i \cdot h$ for $i = 0, 1, 2, \ldots, n$.
- Apply Simpson's 1/3 Rule:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left(f(x_0) + f(x_n) + 4\sum_{i=1,3,5,\ldots}^{n-1} f(x_i) + 2\sum_{i=2,4,6,\ldots}^{n-2} f(x_i)\right)$$

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\Simpson1_3.py
0.2 + 25x - 200x2 + 675x3 - 900x4 + 400x5
0 2 1000
Integral result using Simpson's 1/3 rule: 723.3333333380267
```

Simpson's 3/8 Rule Algorithm

Simpson's 3/8 rule is another method used to approximate the integral of a function over a given interval. This rule is more accurate than Simpson's 1/3 rule as it uses cubic interpolation rather than quadratic interpolation. It has one more functional value which helps to give us more accurate results than any other method. -(Testbook.com)

Algorithm

- Define the function $f(x)$ and the interval $[a, b]$.
- Set n (number of subintervals) to be a multiple of 3.
- Calculate the step size $h = \frac{b-a}{n}$.
- Generate the x-values: $x_i = a + i \cdot h$ for $i = 0, 1, 2, \ldots, n$.
- Apply Simpson's 3/8 Rule:

$$\int_a^b f(x)\,dx \approx \frac{3h}{8}\left(f(x_0) + f(x_n) + 3\sum_{i=1,4,7,\ldots}^{n-1} f(x_i) + 3\sum_{i=2,5,8,\ldots}^{n-2} f(x_i) + 2\sum_{i=3,6,9,\ldots}^{n-3} f(x_i)\right)$$

Input and Output

Below is the input and output of the code execution:

```
(env) D:\programs\Simpson3_8.py
0.2 + 25x - 200x2 + 675x3 - 900x4 + 400x5
0 2 1000
Integral result using Simpson's 3/8 rule: 721.813600574931
```

Heun's Method

Heun's Method is an improved version of Euler's method for solving ordinary differential equations (ODEs). It uses a predictor-corrector approach to improve the accuracy of the numerical solution by considering the average of slopes at the start and end of each step.

Algorithm

The steps to implement Heun's Method are as follows:

1. Start with initial values x_0, y_0, step size h, and the endpoint x_{end}.
2. Compute the predicted y_{pred} using Euler's estimate:

$$y_{\text{pred}} = y + h \cdot f(x, y)$$

3. Compute the corrected value y_{corr} using the average slope:

$$y_{\text{corr}} = y + \frac{h}{2} \left(f(x, y) + f(x + h, y_{\text{pred}}) \right)$$

4. Update x and y:

$$x = x + h, \quad y = y_{\text{corr}}$$

5. Repeat until $x \geq x_{\text{end}}$.

Input and Output

Below is an example of input and output for Heun's Method:

```
(env) D:\programs\Huens.py
2xy
0 1 1 0.1
1.0999999999999999, 3.3375582412603206
```

Runge-Kutta 4th Order Method

The Runge-Kutta 4th Order Method (RK4) is a widely used numerical technique for solving ODEs. It provides high accuracy by considering intermediate slopes within each step.

Algorithm

The steps to implement the Runge-Kutta 4th Order Method are as follows:

1. Start with initial values x_0, y_0, step size h, and the endpoint x_{end}.
2. For each step, calculate:

$$k_1 = h \cdot f(x, y)$$

$$k_2 = h \cdot f\left(x + \frac{h}{2}, y + \frac{k_1}{2}\right)$$

$$k_3 = h \cdot f\left(x + \frac{h}{2}, y + \frac{k_2}{2}\right)$$

$$k_4 = h \cdot f(x + h, y + k_3)$$

3. Update y using:

$$y = y + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}$$

4. Update x:

$$x = x + h$$

5. Repeat until $x \geq x_{\text{end}}$.

Input and Output

Below is an example of input and output for the Runge-Kutta 4th Order Method:

```
(env) D:\programs\RK.py

2xy
0 1 1 0.1
1.0999999999999999, 3.3534601917406692
```