|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Folder** | **File Name** | **Use** | **Output** | **Other** |
| Integration | Rectangle Rule | This code implements the rectangle rule for numerical integration. | Numerical Approximation of the definite inegral | It is recommended for functions that are relatively smooth over the interval [a, b].  Avoid using it for functions with rapidly changing behavior, as it may lead to less accurate results.  Ensure to choose an appropriate number of subintervals (n) based on the characteristics of the function |
|  | Simpsons short | Numerical Integration usin | Result of the numerical integration (float). | # This code is suitable for numerically integrating a given function using Simpson's rule.  # Avoid using this code for functions with singularities or highly oscillatory behavior.  # Ensure that the 'f' function is well-behaved within the specified integration limits. |
|  | Traezoidal | Visualizes a function and calculates the area under the curve using the trapezoid rule. | Approximation of the integral of f(x) from a to b using the trapezoid rule with N subintervals of equal length (float). | # Recommended Use:  # This code is suitable for visualizing a function and calculating the area under the curve using the trapezoid rule.  # Adjust the 'f' function for your specific integration requirements.  # Not Recommended Use:  # Avoid using this code for highly oscillatory functions or functions with singularities.  # Ensure that the 'f' function is well-behaved within the specified integration limits. |
|  | Simpsons Three Eighths | Numerically integrates a given function using Simpson's 3/8 rule. Efficient for smooth functions over the integration interval. Adjust the 'func' function for specific integration requirements. | Result of the function evaluation at x (float) for 'func', and result of the numerical integration (float) for 'calculate' | # Recommended Use:  # This code is suitable for numerically integrating a given function using Simpson's 3/8 rule.  # It is efficient for smooth functions over the integration interval.  # Adjust the 'func' function for your specific integration requirements.  # Not Recommended Use:  # Avoid using this code for functions with singularities or highly oscillatory behavior.  # Ensure that the 'func' function is well-behaved within the specified integration limits. |
|  | Simpsons Adaptive Rule | Evaluates the integral of *f*(*x*) on [a,b][*a*,*b*] using adaptive Simpson's rule. Provides an error bound given by *tol*. | The value of the integral (float). | Suitable for functions that are not highly oscillatory, have rapidly changing curvature, and lack severe oscillations or singularities. Provides accurate results with adaptive precision.  Avoid for functions with severe oscillations or singularities. Less efficient for highly oscillatory functions. Efficient alternatives exist for cases where speed is crucial. |
| Linear Algebra | Gauss AF | Performs Gaussian Elimination with Backward Substitution to solve a system of linear equations Ax = b*Ax*=*b*. | - numpy.ndarray: Solution vector x. | # Recommended Use:  # - This code is suitable for solving a system of linear equations when the matrix A is not too large, and the system has a unique solution.  # - It employs Gaussian Elimination with Backward Substitution, providing an accurate solution.  # - Ensure that the matrix A is well-conditioned for reliable results.  # Not Recommended Use:  # - Avoid using this code for very large matrices, as it may become inefficient.  # - Exercise caution when using it for ill-conditioned matrices, as it might result in numerical instability. |
|  | Linear Refression | Demonstrates two methods of performing linear regression, comparing results by fitting a linear model to given data. Provides an option to read 'x' and 'y' values from an Excel file. | Provides coefficients for manual linear regression and Python function-based linear regression.<br>- Plots data points and both regression lines for visual comparison | # Recommended Use:  # - This code is suitable for comparing manual linear regression calculations with Python function-based linear regression.  # - Use it when you want to visualize and compare the results of two different linear regression methods.  # - Helpful for educational purposes and understanding the underlying calculations.  # Not Recommended Use:  # - Avoid using this code as a standalone linear regression tool for large datasets, as it primarily focuses on comparison and visualization.  # - For production-level tasks, consider dedicated libraries like scikit-learn for more robust and efficient linear regression implementations. |
|  | Spline | Showcases two different interpolation techniques: linear and spline interpolation. Imports data from an external file, performs interpolation using both methods, and plots the results. | Performs linear and spline interpolation on the given data.  Plots the original data points, linear interpolation, and spline interpolation for comparison.  Provides a zoomed-in version of the interpolation results. | # Recommended Use:  # - Utilize this code when you need to understand and compare linear and spline interpolation methods.  # - Useful for educational purposes, demonstrating the differences between interpolation techniques.  # - Suitable for cases where smooth curves are needed between known data points.  # Not Recommended Use:  # - Avoid using this code as a standalone interpolation tool for large datasets.  # - For production-level tasks or precise interpolations, consider specialized libraries like SciPy. |
| ODEs | Euler | Uses the Euler method to solve the first-order ordinary differential equation dy/dx = -y numerically. Calculates and visualizes the approximate solution of the differential equation using the Euler method. | Makes a table of step, x, y-approx, y-exact and error % | # Recommended Usage:  # - Use 'euler' method for quick approximations with larger step sizes.  # - Use 'runge\_kutta' method for more accurate results, especially with smaller step sizes.  # - Adjust the step size based on the desired balance between accuracy and computation time. |
|  | Solve ivp system | Utilizes solve\_ivp from the SciPy library to numerically solve a system of first-order ordinary differential equations (ODEs) defined in the function 'model'. | - Prints the results (x, y\_1, y\_2) in an easy-to-understand format.  - Plots the results showing the evolution of y\_1 and y\_2 with respect to x. | # - Suitable for solving systems of first-order ODEs.  # - Convenient for cases where an appropriate step size can be determined by the solver.  # - Provides an automated approach for solving ODEs and plotting the results.  # - Suitable for quick and efficient computation of ODE solutions.  # - Not recommended for scenarios requiring manual control over the step size or other specific solver parameters. |
|  | Solve ivp system Lorenz | Solves and visualizes solutions of systems of ordinary differentialsympy equations (ODEs) using the solve\_ivp method. | - Plots the results, showing the evolution of y\_1, y\_2, and y\_3 over time.  Optionally prints the results in a text file (commented out in the code). | # Recommended Usage:  # This code is recommended for solving and visualizing solutions  # of systems of ordinary differential equations (ODEs) using the  # solve\_ivp method. It's suitable for dynamical systems described  # by ODEs, especially those exhibiting chaotic behavior.  # Not Recommended:  # This code might not be suitable for extremely stiff ODEs or  # systems with discontinuities. For such cases, consider using  # specialized methods or adjusting solver parameters. |
|  | Secant Function | Secant Method for root finding. | - Prints the approximated root. | - Suitable for finding roots of a function when the derivative is not easily available  Useful when you need a quick and simple method for root finding  Effective for continuous functions without sharp turns.  - Not ideal for functions with sharp turns or if the initial guess is far from the root. |
|  | Rk4 | Solving first-order ordinary differential equations (ODEs) using the Fourth Order Runge-Kutta method. Allows manual control over step size and iterations. | Prints results on screen with neatly aligned columns.  Generates a plot for visual comparison between the RK4 solution and the exact solution. | - Suitable for understanding and applying the Fourth Order Runge-Kutta method to solve ODEs.  Useful for educational purposes and quick numerical solutions.  Provides clear steps for manual control over step size and iterations.  Not recommended for complex or varied ODEs where higher-order methods or specialized solvers might be more appropriate.  Assumes a simple ODE with a fixed coefficient. |
|  | Euler System | Numerically solves a system of first-order ordinary differential equations (ODEs) using the Euler method. Useful for understanding the behaviour of dynamic systems. | - Plots the results for visualization. <br>- Saves the results in a text file for later use. | # Usage:  # - Applicable for solving systems of ODEs numerically with given initial conditions and parameters.  # - Useful for understanding the behavior of dynamic systems represented by ODEs.  # - The Euler method might not be accurate for stiff equations or may require very small step sizes in some cases.  # - Provides a numerical solution and allows comparison between different variables' behaviors.  # - Saves the results in a text file for later analysis or usage. |
|  | Euler Stiff | Solves a first-order ordinary differential equation (ODE) using the Euler method with adaptive step sizes. | - Numerical solution saved in 'output\_h1\_h2.dat'.  - Refined exact solution saved in 'output\_exact.dat' for comparison  - Plots numerical and exact solutions for isual comparison. | - Useful for problems where the solution undergoes rapid changes, requiring adaptive step sizes.  - May not be suitable for highly stiff ODEs; consider specialized methods for stiff problems.  - Check accuracy by comparing the numerical solution with a refined exact solution.  - Adjust initial conditions and parameters based on the specific ODE being solved. |
|  | Euler implicit import fun | Solves a first-order ordinary differential equation (ODE) using the implicit Euler method. | - Numerical solution saved in 'output\_h0.01.dat'.  - Refined exact solution saved in 'output\_exact.dat' for comparison.  - Plots numerical and exact solutions for visual comparison. | - Useful for problems where an implicit method is preferred, such as stability concerns or stiff ODEs.  - Consider adjusting the number of iterations in the secant method based on convergence.  - Check accuracy by comparing the numerical solution with a refined exact solution. <br> - Adjust initial conditions and parameters based on the specific ODE being solved. |
|  | Euler implicit | Solves a first-order ordinary differential equation (ODE) using the implicit Euler method. | Numerical solution saved in 'output\_h0.01.dat'.  Refined exact solution saved in 'output\_exact.dat' for comparison.  Plots numerical and exact solutions for visual comparison. | Adjust the step size (h) based on the specific ODE characteristics. Smaller steps may be required for stiff equations.  Check convergence and adjust the number of iterations in the secant method as needed.  Validate accuracy by comparing the numerical solution with a refined exact solution  Consider adjusting the initial conditions and parameters for different ODEs. |
| Regression and Interpolation | Error root finding | Demonstrates the Newton-Raphson and Secant methods for root-finding. Iteratively finds the root of a given function using different approaches. | - Plots show the iteration count vs. solution for both methods.  - Additional plot shows error (absolute value of f(solution) and expected error decay. | - May fail to converge or provide inaccurate results for complex functions or poor initial guesses.  - Use when a continuous function and an initial guess for the root are available.  - Requires the function to have a derivative for Newton's method.  - Avoid for functions that are discontinuous or lack a derivative.  - Be cautious for functions with multiple roots or near singular points. |
|  | Ploy div | Performs polynomial long division by iterating through coefficients. | Takes an array 'A' containing coefficients of p(x)p(x) and a value 't', outputs coefficients of the quotient q(x)q(x) and the residual r  - The coefficients of the quotient q(x)q(x) represent the result of polynomial long division.  - The residual rr is the remainder after division. | - Suitable for performing polynomial long division using the iterative method.  - Applicable for polynomials where long division is applicable.  - Use when you want to perform polynomial long division using the iterative method.  - Avoid using for non-polynomial functions or cases where long division is not the appropriate method. |
| Series calculations | Exponential series simple | Approximates the value of e^x using a Taylor series expansion. Demonstrates the iterative process to calculate the exponential function. | - Displays the approximation and relative error at each iteration.  - Final results include the numerical result, exact value, and true relative error. | - Recommended for its purpose of demonstrating a simple iterative approach to approximate e^x.  - Not recommended for high precision; specialized functions in the math module may provide more accurate results.  - Useful for understanding the iterative process of the Taylor series.  - Consider using specialized functions for more accurate results in practical applications. |
|  | Sqsum | Calculates the sum of the squares of the first n odd natural numbers using a loop to iterate through the odd numbers and accumulate their squares. | - Straightforward for its purpose  - For better efficiency, consider using a formula to directly compute the sum. |  |
| Univariate Optimization | Golden Search Method | The provided code implements the golden-section algorithm to maximize a given function **ftn**. The algorithm iteratively refines the values of **xl**, **xm**, and **xr** and terminates when the difference between **xr** and **xl** is less than or equal to the specified tolerance (**tol**). The function returns the value of **xm** at the termination of the algorithm. | The output of **gsection(ftn, xl, xm, xr, tol = 1e-9)** is the value of **xm** that maximizes the function **ftn**.  **print(ftn(xm))** prints the value of the function **ftn** evaluated at the final **xm** after the algorithm. | Use this algorithm when you need to maximize a unimodal function (a function with a single peak) over a specified interval.  Avoid using this algorithm when dealing with multimodal functions (functions with multiple peaks) or when the function does not have a well-defined peak within the specified interval.  Ensure that the initial values of xl, xm, and xr satisfy the condition xl < xm < xr and that ftn(xl), ftn(xr) <= ftn(xm). |
|  | Newton Maximise | The provided code implements the Newton-Raphson optimization algorithm to find the root of a function f(x). The algorithm uses the first and second derivatives of the function (fprime and fsecond) to iteratively refine the solution. The code includes a demonstration of the Newton-Raphson method for a specific function. | The value of **x\_star** obtained using the Newton-Raphson method, which represents the root of the function | Use this algorithm when you need to find the root of a differentiable function and have access to its first and second derivatives.  When Not to Use:  Avoid using this algorithm when the function is not differentiable, or when computing the second derivative is computationally expensive.  Ensure that the initial guess x0 is close enough to the actual root for the method to converge.  The code includes a quadratic approximation (quadratic\_approx) using the first and second derivatives. |
|  | Newton Minimize | The provided code appears to be similar to the previous example, with some modifications in the function **f(x)** and its derivatives. It still implements the Newton-Raphson optimization algorithm to find the root of a differentiable function. The code includes plotting the target function, its quadratic approximation, and marking the root found by the Newton-Raphson method. | The value of **x\_star** obtained using the Newton-Raphson method, representing the root of the function. | Use this algorithm when you need to find the root of a differentiable function and have access to its first and second derivatives.  Avoid using this algorithm when the function is not differentiable or when computing the second derivative is computationally expensive.  Ensure that the initial guess x0 is close enough to the actual root for the method to converge.  The code includes a quadratic approximation (quadratic\_approx) using the first and second derivatives. |
|  | Newton Twodimensional Optimization | The provided code implements the Newton-Raphson optimization algorithm to find the minimum of the Rosenbrock function. The Rosenbrock function is defined as f(x, y) = (1 + x)^2 + 100(y - x^2)^2*f*(*x*,*y*)=(1+*x*)2+100(*y*−*x*2)2. | The value of root obtained using the Newton-Raphson method, representing the minimum of the Rosenbrock function.  Arrays iter\_x, iter\_y, and iter\_count containing the x-coordinate, y-coordinate, and iteration count at each step of the optimization, respectively. | Use this algorithm when you need to find the minimum of a differentiable function and have access to its gradient and Hessian matrix.  Avoid using this algorithm when the function is not differentiable, or when computing the gradient and Hessian is computationally expensive.  Additional Notes:  The code uses the Rosenbrock function, a common test problem for optimization algorithms.  The initial guess is set to x = -2 and y = 2. |
| Root Finding | bis | The code uses the root\_scalar function from SciPy to find the root of a given function within a specified bracket. In this case, it defines a function f(x) as the product of the sine and exponential functions. | The code outputs the approximate root of the function within the specified bracket. | Use this code when you have a well-behaved function, and the bisect method is suitable. If dealing with functions exhibiting more complex behavior, consider exploring other root-finding methods provided by SciPy.  To modify for a different function or bracket, define a new function 'f(x)' based on the equation you want to solve and adjust the bracket in the 'root\_scalar' function to set the range for the search. |
|  | Bisection with error control T3 | The code implements the Bisection method with error control for solving equations, providing enhanced accuracy by iteratively narrowing down the interval where the root is located. | The code outputs the approximate root and the number of iterations taken to achieve the specified tolerance. | Use this code when dealing with functions where the Bisection method is suitable and a higher level of accuracy is required. Be cautious when applying it to functions with rapid changes.  To modify for a different function or interval, define a new function 'f(x)' and adjust the initial interval '[a, b]' and tolerance level accordingly. Ensure that 'a' and 'b' bound a root with opposite signs. |
|  | Synthetic Division Poly | The code performs extended synthetic division of polynomials, providing a quick and efficient method for dividing non-monic polynomials. | The code outputs the quotient and remainder of the polynomial division. | Use this code when dividing polynomials, especially non-monic ones. Be cautious when dealing with polynomials with complex roots or potential division by zero.  To modify for other polynomials, update the 'dividend' and 'divisor' lists with the coefficients of the polynomials you want to divide. Adjust the normalization and output format as needed. If the divisor is monic, you can skip the normalization step. |
|  | Week 3 Lecture exercise | The code utilizes the Newton-Raphson method to solve systems of nonlinear equations, providing a quick and accurate convergence to the solution for well-behaved functions. | The code outputs the final solution and the relative error after convergence. | Use this code when solving systems of nonlinear equations, especially when a quick and accurate convergence is required. Be cautious when dealing with poorly conditioned or ill-posed problems, as the Jacobian matrix may be close to singular.  To modify for other functions, update the 'equations' and 'jacobian' functions, adjust initial guesses ('x0' and 'y0'), and modify the convergence criteria by adjusting 'tolerance' and 'max\_iterations' based on the desired level of accuracy and computational resources. |
|  | Single Fixed Point Iteration | The code is designed for solving equations using the Fixed Point Iteration method, which is recommended for well-behaved functions where convergence is assured and when obtaining the derivative is challenging or computationally expensive. | The code outputs the iterations, the current value, and the function value at each iteration, providing insights into the convergence process. | Use this code when solving equations, especially for functions where the Fixed Point Iteration method is appropriate. Be cautious when dealing with functions having multiple roots, as the method may not converge, and convergence can be slow for certain functions. Additionally, it is not suitable for functions with sharp turns or near-vertical slopes where the derivative is close to zero.  To modify for other functions, update the 'f(x)' and 'g(x)' functions and adjust the initial guess ('x0'), tolerance ('e'), and maximum steps ('N') accordingly. |
|  | Secant (1) | The code is designed for finding approximate solutions of equations using the Secant method, which is recommended for cases where the Newton-Raphson method may not be applicable or when obtaining the derivative is challenging or computationally expensive. | The code outputs the solution and prints information about the convergence process, including whether the method fails or if an exact solution is found. | Use this code when solving equations, especially for functions where the Secant method is appropriate. Be cautious when dealing with functions having multiple roots, as the method may not converge, and convergence can be slow for certain functions. Additionally, it is not suitable for functions with sharp turns or near-vertical slopes where the derivative is close to zero.  To modify for other functions, update the 'f(x)' function and adjust the initial interval [a, b], and the number of iterations ('N') accordingly. |
|  | N-R Method T3 | The code is designed for finding roots of equations using the Newton-Raphson and Secant methods with error control, recommended for well-behaved functions where convergence is assured and when obtaining the derivative is challenging or computationally expensive. | The code outputs the roots and the number of iterations for each method, providing insights into the convergence process. | Use this code when solving equations, especially for functions where the Newton-Raphson and Secant methods are appropriate. Be cautious when dealing with functions having multiple roots, as the methods may not converge, and convergence can be slow for certain functions. Additionally, they are not suitable for functions with sharp turns or near-vertical slopes where the derivative is close to zero.  To modify for other functions, update the 'f(x)' and 'df(x)' functions and adjust the initial guess ('x0' and 'x1'), tolerance level, and maximum number of iterations accordingly. |
|  | Newton | The code is designed for finding roots of equations using Newton's method, recommended for well-behaved functions where convergence is assured and when the derivative is readily available. | The code outputs the root and the number of iterations, providing insights into the convergence process. | Use this code when solving equations, especially for functions where Newton's method is appropriate. Be cautious when dealing with functions having multiple roots, as the method may not converge, and convergence can be slow for certain functions. Additionally, it is not suitable for functions with sharp turns or near-vertical slopes where the derivative is close to zero.  To modify for other functions, update the 'f(x)' and 'Df(x)' functions and adjust the initial guess ('x0'), tolerance level, and maximum number of iterations accordingly. |
|  | Naïve Line Search | The code is designed for finding roots of equations using a naive root-finding approach, recommended for simple functions where a brute-force approach is acceptable and the solution is expected to be found within a reasonable number of iterations. | The code outputs the root and the number of steps taken, providing insights into the convergence process. | Use this code when solving equations, especially for functions where a naive approach is suitable. Be cautious when dealing with functions having multiple roots, as the method may not converge, and convergence can be slow for certain functions. Additionally, it may not work well for functions with sharp turns or near-vertical slopes.  To modify for other functions, update the 'f(x)' function and adjust the initial guess ('x\_guess'), tolerance level, and step size accordingly. |
|  | Muller | The code is designed for finding roots of equations using Muller's method, recommended for functions with complex roots where other methods may fail, and the root is expected to be found within a reasonable number of iterations. | The code outputs the value of the root, providing the result of the Muller's method. | Use this code when solving equations, especially for functions where Muller's method is appropriate. Be cautious when dealing with functions having multiple roots, as the method may not converge, and it might not be as efficient for well-behaved functions with simple roots. Additionally, it may not work well for functions with sharp turns or near-vertical slopes.  To modify for other functions, update the 'f(x)' function and adjust the initial guesses ('a', 'b', 'c'), and the maximum number of iterations ('MAX\_ITERATIONS') accordingly. |
|  | Modified False Position | The code is designed for finding roots of equations using the Regula Falsi (False Position) method. | The code prints the value of the root and the number of iterations taken to converge. | Use this code when solving equations, especially for functions where the Regula Falsi method is appropriate. Be cautious when dealing with functions having multiple roots, as the method may not converge, and it might not be as efficient for well-behaved functions with simple roots. Additionally, it may not work well for functions with sharp turns or near-vertical slopes.  To modify for other functions, update the 'func(x)' function and adjust the initial guesses ('a', 'b') and the maximum number of iterations ('MAX\_ITER') accordingly. |
|  | Inverse Quadratic Interpoilation | The code implements the Inverse Quadratic Interpolation method to find roots of equations. | It outputs the root of the equation and the number of steps taken to converge. | Recommended for functions where convergence is assured and the root is expected to be found within a reasonable number of iterations. **Avoid**: Not suitable for functions with multiple roots, and may not be efficient for well-behaved functions with simple roots or functions with sharp turns. |
|  | Horners Quadratic | The code provides three polynomial evaluation methods: Horner's Method, Naive Polynomial Evaluation, and Iterative Polynomial Evaluation. | Outputs the evaluated polynomial values for each method. | Choose the method based on specific requirements and characteristics of the polynomial. **Avoid**: While all three methods are valid, Horner's Method is generally recommended for its efficiency and numerical stability compared to the naive and iterative methods. |
|  | False Position Unmodified | The code implements the False Position (Regula Falsi) method for solving equations. | It outputs the root of the equation and the number of iterations performed. | Valid for solving equations, but not the most efficient choice due to slow convergence in some cases. **Avoid**: Consider more robust and efficient methods like Newton-Raphson or the Secant method for faster convergence. |
|  | Bisection | The code implements the Bisection method for solving equations. | It outputs the approximate root of the equation. | Bisection is reliable for finding roots within a specified interval. **Avoid**: May not perform well for functions with complex behavior, consider other methods like Newton-Raphson or the Secant method for such cases. |
| Multivariate Operations | H-Path Optimization | The code implements the steepest ascent algorithm for optimization, specifically for a mathematical function F(x, y). It assumes that the partial derivatives dFx(x) and dFy(y) with respect to x and y are available. | The code outputs the final optimized values for x, y, and F(x, y), as well as the final values of the partial derivatives df/dx and df/dy. | This function is suitable for optimization problems where you want to maximize a continuous and differentiable function F(x, y). It's not suitable for non-differentiable functions or situations where the given partial derivatives are not available. Additionally, the algorithm may not converge or provide accurate results for functions with multiple local maxima or minima. It's important to ensure the convergence criteria are appropriate for the specific problem. |
|  | Lagrange Multiplier Optimization | The code utilizes the **minimize** function from the **scipy.optimize** module to optimize an objective function subject to a constraint. The objective function is the sum of squares of x, y, and z. | The code outputs information about the optimization results, including whether the optimization was successful, the number of iterations, the optimized values of x, y, and z, the optimized objective function value, and the constraint value at the optimized solution. | This function is suitable for optimization problems with equality constraints, specifically when the constraint is defined as '2 \* x - y + z - 3 = 0'. It may not be suitable for problems with different types of constraints or when dealing with non-linear constraints that cannot be easily expressed in the form of equalities. Additionally, the choice of the initial guess [1, 4, 5] may affect the convergence to the optimal solution. Adjusting the initial guess may be necessary for some problems. |
|  |  |  |  |  |
| Tutorials | T2 | Calculates the sum of the squares of the first n odd natural numbers.  Uses a loop to iterate through odd numbers and accumulate their squares.  Generates a random array of 20 values and selects and prints values greater than 5 and less than 6.  Creates a random array of equispaced coordinates between 0 and 2π.  Plots a sine graph using the generated coordinates.  Computes the approximation of π using a truncated series.  Plots the approximation over iterations and analyzes true and extrapolation errors. | Prints the sum of the squares of the specified number of odd natural numbers.  Prints the randomly generated array and the selected values.  Displays the sine graph.  Prints the approximation, true error, and extrapolation error for each iteration.  Displays plots for the approximation and error analysis. | Efficient for small values of n.  Consider using a formula for better efficiency.  Suitable for generating and selecting values within a specified range.  Useful for visualizing sine functions.  Helpful for understanding the convergence behavior of the series. |
|  | T3 | Implements the bisection method with error control to find the root of a given function  Applies the bisection method with error control to find the root of a different function. | Prints the root and the number of iterations.  Prints the root and the number of iterations | Suitable for finding roots of functions with known initial intervals.  Suitable for finding roots of various functions with known initial intervals. |
|  | T4 | Implements the Newton-Raphson method and the Secant method to find the root of a given function.  Evaluates the convergence behavior for different tolerance values (N).  Performs synthetic division for a given polynomial and divisor. | Plots the convergence behavior for both methods with error values.  Prints the quotient coefficients and remainder. | Useful for understanding the convergence rate of Newton-Raphson and Secant methods for a given function.  Prints the quotient coefficients and remainder. |
|  | T5 | Perform cubic spline interpolation on a set of data points  Estimate a line of best fit (LOBF) for the given data using linear regression. | Plots the original data points and the cubic spline interpolation.  Plots the original data points and the line of best fit. | Analyzes the smoothness of the interpolated curve compared to the original data.  Examines the fit of the linear model to the given data. |
|  | T6 | Solve a system of linear equations representing a physics problem. | Displays the original system of linear equations and the solution vector x*x* (corresponding to masses m*m*, tensions T, and reactions R). |  |
|  | T7 | Implement numerical methods (Euler method and Fourth Order Runge-Kutta method) to solve a first-order ordinary differential equation. | Created plots for each step size, showing the numerical solutions (Euler and RK4) and the exact solution.  Plotted the error curves for each method.  Saved solution and error plots for each step size as PDF files. |  |
|  | T8 | The code solves a system of ordinary differential equations (ODEs) using the solve\_ivp method from the scipy.integrate module.  The ODEs represent the Lorenz system. | Three plots are generated:  Plot 1: Shows the values of y\_1, y\_2, and y\_3 as functions of time.  Plot 2: Displays y\_2 as a function of y\_1.  Plot 3: Illustrates y\_3 as a function of y\_1. | Use this code when you need to solve a system of ODEs, particularly the Lorenz system.  Suitable for visualization of the system's behavior over time.  Avoid using this code without adjusting the system parameters and initial conditions to match your specific problem. |
|  | T9 | The code performs numerical integration using various methods: rectangular rule, trapezoidal rule, Simpson's 1/3 rule, and Simpson's 3/8 rule.  It then calculates and compares the errors for each method. | The code prints the results of the numerical integration for each method.  The errors for each method are calculated and printed.  The code generates a plot showing how the integral values evolve with the number of intervals for each method | Use this code when you need to numerically integrate a function over a specified range.  Useful for comparing the performance of different numerical integration methods.  Avoid using this code without understanding the nature of the function being integrated.  Ensure the chosen integration method aligns with the characteristics of the function. |
|  | T10 | The code uses SymPy to perform symbolic differentiation and function definition.  It defines a function y(x) = c \* sin(a \* x) and calculates its first and second derivatives, as well as additional functions z(x) and s(x) based on these derivatives.  The code prints the original function, its first and second derivatives, and the simplified form of a derived function. | The code outputs the symbolic expressions for the original function, its first and second derivatives, as well as the derived functions z(x), s(x), and their simplified forms.  Additionally, it prints the Taylor series expansion of the simplified function s(x). | Use this code when you need to:  Symbolically define a mathematical function.  Calculate derivatives of a function with respect to a variable.  Perform simplification of mathematical expressions.  Obtain the Taylor series expansion of a function.  Avoid using this code if you only need numerical derivatives or if symbolic mathematics is not necessary for your task.  Ensure that your problem involves mathematical expressions and symbolic manipulation before using this code. |

Past Papers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **YEAR** | **Q** | **Summary of the question** | **Summary of the code** | **Code output** |
| 2  0  1  7 | Q1 a) | Plot a 5th order poly to find the max deflection of a beam | Defines beam parameters and a function for deflection.  Plots the function, highlighting the maximum deflection point. | Displays the maximum beam deflection and the corresponding x-coordinate. |
| Q1. b) | Find the maximum deflection using an optimization technique | Uses the **scipy.optimize.minimize** function to find the maximum deflection point. | Prints the x-coordinate for the maximum deflection obtained through optimization. |
| Q1. c) | Write the Taylor series expansion of a given function with a perturbation. | Uses the **sympy** library to define a function and calculates its **Taylor** **series** expansion. | Prints the Taylor series expansion of the given function. |
| Q1. d) | Calculate the sensitivity of a function to a perturbation. | Uses numerical methods to calculate **sensitivity** and the Taylor series expansion. | Prints sensitivity, Taylor series expansion, and change in the function value. |
| Q2. a) | Calculate the interest rate using a given equation. | Uses the **Newton-Raphson method** to find the interest rate that satisfies the given equation. | Prints the calculated interest rate. |
| Q2. b) | Find the displacement of a rocket after 30 seconds using a given equation and Simpson's rule. | Defines a function and uses **Simpson's rule** for numerical integration. | Prints the distance traveled by the rocket. |
| Q3.  a, b, c) | Find the volume of soil to be removed using optimization. | Defines functions for soil profiles and an objective function for optimization.  Uses **scipy.optimize.minimize** to find optimal parameters. | Prints the optimized parameters and the result of the integration. |
| 2  0  1  8 | Q1. a) | Optimize two values given initial conditions and one equation. | Defines mechanical system equations and performs **optimization** to **maximize** the buckling stress. | Prints the maximized and optimized values of d and t, along with the optimal cost |
| Q1. b) | Write the Taylor series expansion of ln(1+x) for 3 terms. | Uses the **sympy** library to define the function and calculate its **Taylor** series expansion. | Prints the Taylor series expansion of ln(1+x) |
| Q1. c) | Calculate the value of y for 7 significant figures when x=0.1 | **Evaluates** the expression for ln(1+x) with x=0.1 and rounds it to 7 significant figures. | Prints the value of ln(1+0.1) with 7 significant figures. |
| Q2. a) | Prove that C is the general solution to B. | Defines equations C and B, calculates their **derivatives**, and checks if C is the **general solution** to B. | Prints equations C, its derivatives, equation B, and checks if C is the general solution to B. |
| Q2. b) | Use a root-finding technique to calculate tension with initial conditions. | Implements the **bisection method** to find the root of a given function related to tension. | Prints the approximate tension obtained using the bisection method. |
| Q2. c) | Use fixed-point iteration to determine porosity using a given equation. | Implements the **modified fixed-point iteration** method to find the root of a given function. | Prints the required root (porosity) obtained through fixed-point iteration. |
| Q3. a) | Factorize a second-order polynomial and find the roots. | Uses the **sympy** library to factorize a second-order polynomial and find its roots. | Prints the factorization and roots of the given second-order polynomial. |
| Q3. b) | Use polynomial deflation to find the roots of a fourth-order polynomial. | Implements the **secant method** to find the roots of a given fourth-order polynomial. | Prints the roots of the given fourth-order polynomial obtained using the secant method. |
| 2  0  1  9 | Q1. a) | Find the 2 real and imaginary roots of the polynomial equation given initial conditions. | Defines a function to find roots using **NumPy's roots** function, separates real and complex roots, and prints the polynomial equation and roots. | Prints the polynomial equation and the real and imaginary roots. |
| Q1. b) | Calculate the smallest possible amount of work in 10 seconds using the smallest values from the previous answer. | Defines displacement and force equations based on given parameters and prints the displacement and force. | Prints the displacement and force. |
| Q2.  a, b) | Minimize the integral and output one variable. Use that variable to find the root. | Defines an objective function involving an integral and uses the **scipy.optimize.minimize** function to find the optimized value. | Prints whether the optimization was successful, the number of iterations, the optimized variable, and the optimized objective function value. |
| Q3. a) | Find the two lowest values for a variable satisfying a given equation. | Uses the **secant method** to find the roots of a given equation and calculates corresponding values of another variable. | Prints the roots of the equation and the corresponding values of the variable. |
| Q4 | Find the smallest radius of the satellite using polar coordinates and equations. | Uses **scipy.optimize.fsolve** to find constants in polar coordinates and then calculates the minimum radius. | Prints the constants and the minimum radius. |
| Q5 | Find the values of three angles that minimize the potential energy in the system while satisfying constraints. | Defines the objective function and constraints, uses **scipy.optimize.minimize** to find optimized angles, and calculates potential energy. | Prints the optimized angles and the absolute potential energy of the system.  Plots the mechanical system configuration. |
| 2  0  2  2 | Q1. a) | Evaluate a truncated series with a specified number of terms and find the approximation. | Calculates the approximation of π using a **truncated series** and reports errors for different numbers of terms. TUTORIAL Q | Approximation of π with specified terms and errors. |
| Q1. b) | Compute and report errors for different numbers of terms in the truncated series. | **Calculates errors** for different numbers of terms in the truncated series. | Number of terms, true error, and estimated error for each case. |
| Q2. a) | Solve a given ordinary differential equation (ODE) using the Euler method and report values at various intervals. | Solves the **ODE** using the **Euler** method and reports the values at specified intervals. | Solutions at specified intervals. |
| Q2. b) | Compute and report the ignition delay for different initial conditions. | Uses the **Euler** method to find solutions and calculates ignition delay for **different initial conditions**. | Ignition delay for each initial condition. |
| Q2. c) | Perform stability analysis for different step sizes in the Euler method. | Analyzes **stability** for various step sizes in the Euler method and **plots** the results. | Plot showing stability analysis and threshold value for instability. |
| Q2. d) | Explain stability in numerical methods for ODEs, specifically for Euler methods. | Provides a detailed explanation of stability considerations for numerical methods applied to ODEs. |  |
| Q3. a) | Use inverse quadratic interpolation to find roots of a given function. | Implements **inverse quadratic interpolation** to find roots. | Approximate root and the number of steps taken. |
| Q3. b) | Use the bisection method to find roots of a given function. | Implements the **bisection method** to find roots. | Approximate root. |
| Q3. c) | Use the secant method to find roots of a given function. | Implements the **secant method** to find roots. | Approximate root. |
| Q3. d) | Create a table with coefficients and rounded values for the roots. | Constructs a table with root values and method information. | Displayed table. |
| Q4. a) | Show that a given function is the general solution to a given differential equation. | Defines symbols, functions, and calculates derivatives to demonstrate the general solution. | Simplified form of the differential equation. |
| Q4. b) | Find the equation for instantaneous frequency and compare it to a given equation. | Sets up and **solves a quadratic equation** to find the instantaneous frequency. | Value of instantaneous frequency. |
| Q4. c) | Calculate the time taken for a pendulum mass to reach a specified amplitude. | Uses numerical methods to find the time for a specified amplitude. | First positive real solution for time. |
| Q4. d) | Find the damping coefficient required to halve the time obtained in the previous part. | Uses numerical methods to find the damping coefficient for a specified time. | Damping coefficient. |