LAB 6 Solution of Ordinary Differential Equations:
1.Runge-Kutta fourth order method for first order ODE

Working principle

The Runge-Kutta Fourth Order Method (RK4) is a numerical method for solving first-order ODEs of the form:

$$\frac{dy}{dx} = f(x,y), \qquad y(x_0) = y_0$$

It computes $y_{\{n+1\}}) \left(the\ value\ of\ (\ y\ )\ at\ (\ x_{\{n+1\}})\right)$ iteratively using the following steps:

$$k_1 = h \cdot f(x_n, y_n)$$
$$k_2 = h \cdot f\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$
$$k_3 = h \cdot f\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right)$$
$$k_4 = h \cdot f(x_n + h, y_n + k_3)$$
$$y_{\{n+1\}} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Here:

- h: Step size.
- k_1, k_2, k_3, k_4): Intermediate slopes computed at different points within the interval.

Pseudo code:

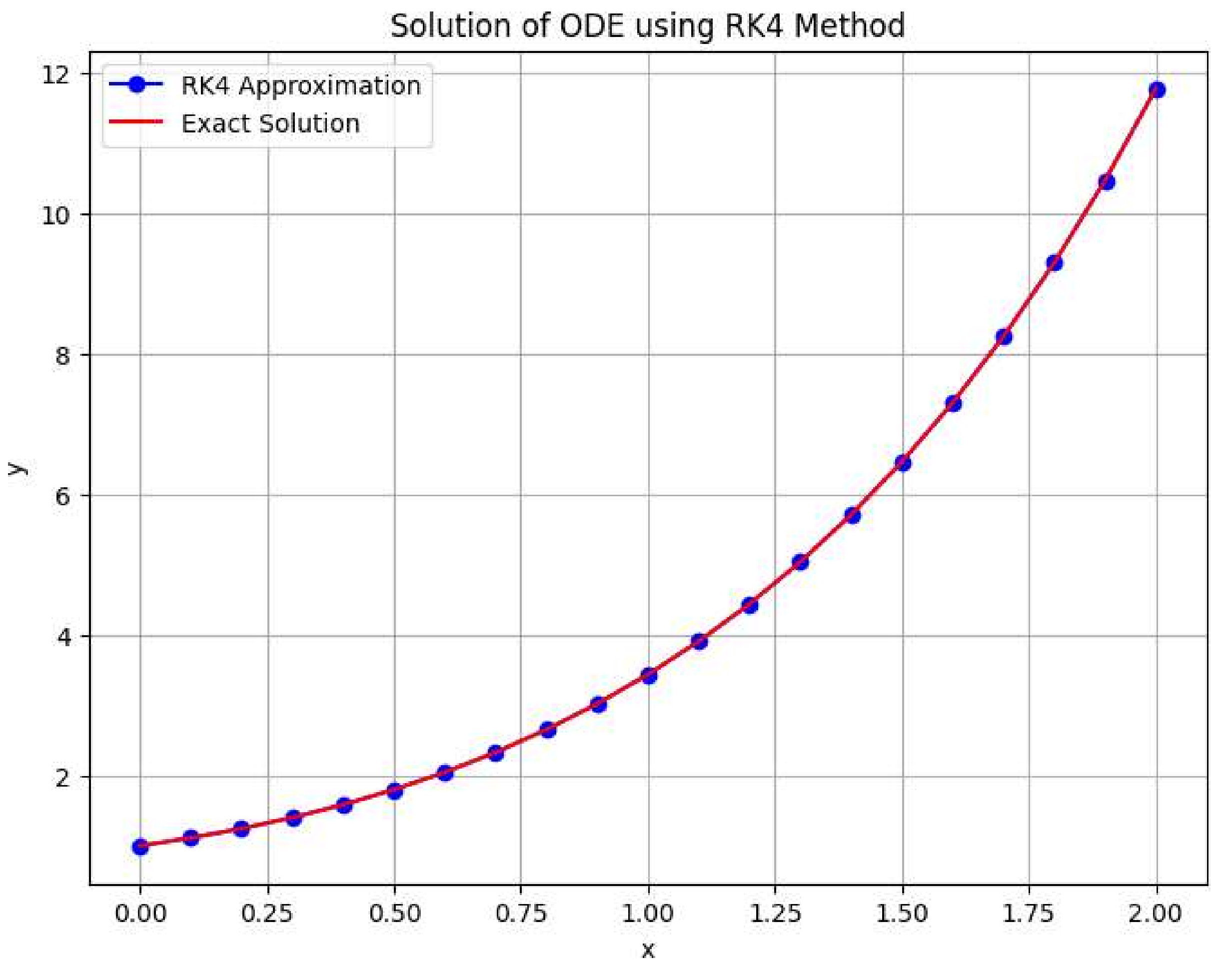
1. Input:
   - Function f(x,y), initial condition x_0, y_0, step size h, and the interval [x_0, x_{end}.
2. Steps:
   1. Set x=x_0,y=x_0 .
   2. Repeat until x≤x_{end}
      - Compute k_1, k_2, k_3, k_4 using the equations above.
      - Update y=y+$\frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$.
      - Increment x=x+h.
   3. Store and return x,y
3. Output:
   - Solution y(x) over the interval.

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

# Function defining the ODE dy/dx = f(x, y)
def f(x, y):
    return x + y  # Example ODE: dy/dx = x + y

# Runge-Kutta 4th Order Method
def runge_kutta_4(f, x0, y0, h, x_end):
    x_values = [x0]
    y_values = [y0]

    x = x0
    y = y0

    while x < x_end:
        k1 = h * f(x, y)
        k2 = h * f(x + h/2, y + k1/2)
        k3 = h * f(x + h/2, y + k2/2)
        k4 = h * f(x + h, y + k3)

        y = y + (k1 + 2*k2 + 2*k3 + k4) / 6
        x = x + h

        x_values.append(x)
        y_values.append(y)
```

```
    return x_values, y_values

# Inputs
x0 = 0        # Initial x
y0 = 1        # Initial y
h = 0.1       # Step size
x_end = 2     # End point of x

# Solve the ODE using RK4
x_values, y_values = runge_kutta_4(f, x0, y0, h, x_end)

# Exact solution for comparison (if available)
def exact_solution(x):
    return -x - 1 + 2*np.exp(x)  # Exact solution for dy/dx = x + y,
y(0) = 1

exact_y_values = [exact_solution(x) for x in x_values]

# Print results
print("x values:", x_values)
print("RK4 y values:", y_values)
print("Exact y values:", exact_y_values)

# Visualization
plt.figure(figsize=(8, 6))
plt.plot(x_values, y_values, 'o-', label="RK4 Approximation",
color='blue')
plt.plot(x_values, exact_y_values, 'r-', label="Exact Solution",
color='red')
plt.title("Solution of ODE using RK4 Method")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()
plt.show()
```

```
x values: [0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6, 0.7,
0.7999999999999999, 0.8999999999999999, 0.9999999999999999,
1.0999999999999999, 1.2, 1.3, 1.4000000000000001, 1.5000000000000002,
1.6000000000000003, 1.7000000000000004, 1.8000000000000005,
1.9000000000000006, 2.0000000000000004]
RK4 y values: [1, 1.1103416666666668, 1.242805141701389,
1.3997169941250756, 1.5836484801613715, 1.7974412771936765,
2.0442359241838663, 2.327503253193554, 2.651079126584631,
3.019202827560142, 3.436559488270332, 3.9083269801179634,
4.440227735556119, 5.038586020027669, 5.7103912272423285,
6.4633678312707605, 7.3060526955587, 8.247880512594522,
9.299278229337848, 10.471769403449168, 11.778089534751086]
Exact y values: [1.0, 1.1103418361512953, 1.2428055163203398,
1.3997176151520063, 1.5836493952825408, 1.7974425414002564,
```

```
2.0442376007810177, 2.327505414940953, 2.651081856984935,
3.019206222313899, 3.43656365691809, 3.9083320478928654,
4.440233845473094, 5.038593335238489, 5.7103999336893505,
6.463378140676131, 7.306064848790233, 8.247894783454402,
9.299294928825898, 10.471788884558546, 11.778112197861306]

<ipython-input-1-c024f21b3469>:53: UserWarning: color is redundantly
defined by the 'color' keyword argument and the fmt string "r-" (->
color='r'). The keyword argument will take precedence.
  plt.plot(x_values, exact_y_values, 'r-', label="Exact Solution",
color='red')
```




```python
import sympy as sp

def f(x, y):
    return x + y

def func_input():
    function_str = input("Enter your function (use 'x' and 'y' as the
variables) (Example: x + y): ")
    x, y = sp.symbols('x y')
    sp_function = sp.sympify(function_str)
```

```
    func = sp.lambdify((x, y), sp_function, modules=['numpy'])
    return func, sp_function

def rk4(func, x0, y0, xn, n):
    # Calculating step size
    h = (xn - x0) / n

    print('---------'*4)
    print('x0\ty0\tyn')
    print('---------'*4)
    for i in range(n):
        k1 = h * func(x0, y0)
        k2 = h * func(x0 + h/2, y0 + k1/2)
        k3 = h * func(x0 + h/2, y0 + k2/2)
        k4 = h * func(x0 + h, y0 + k3)
        k = (k1 + 2 * k2 + 2 * k3 + k4) / 6
        yn = y0 + k
        print(f'{x0:.4f}\t{y0:.4f}\t{yn:.4f}')
        print('---------'*4)
        y0 = yn
        x0 = x0 + h

    print()
    print(f'At x={xn:.4f}, y={yn:.4f}')

def main():
    print("Runge-Kutta 4th Order (RK4) Method for Solving ODEs")
    print()

    # Get user inputs for initial conditions and function
    function_str = "x + y"
    func = f
    x0 = 0
    y0 = 1
    xn = 2
    step = 10

    default = input("Use default limits? (y/n): ").strip().lower() == "y"
    if not default:
        func, function_str = func_input()
        x0 = float(input("Enter initial value of x (x0): "))
        y0 = float(input("Enter initial value of y (y0): "))
        xn = float(input("Enter point to evaluate the solution (xn): "))
        step = int(input("Enter number of steps: "))

    print(f"f(x, y) = {function_str}")
    print(f"x0 = {x0}")
    print(f"y0 = {y0}")
    print(f"xn = {xn}")
    print(f"Number of steps = {step}")
```

```
    print()

    # Call RK4 method and solve the ODE
    rk4(func, x0, y0, xn, step)

if __name__ == "__main__":
    main()
```

Runge-Kutta 4th Order (RK4) Method for Solving ODEs

Use default limits? (y/n): y
f(x, y) = x + y
x0 = 0
y0 = 1
xn = 2
Number of steps = 10

```
------------------------------------
x0        y0        yn
------------------------------------
0.0000 1.0000 1.2428
------------------------------------
0.2000 1.2428 1.5836
------------------------------------
0.4000 1.5836 2.0442
------------------------------------
0.6000 2.0442 2.6510
------------------------------------
0.8000 2.6510 3.4365
------------------------------------
1.0000 3.4365 4.4401
------------------------------------
1.2000 4.4401 5.7103
------------------------------------
1.4000 5.7103 7.3059
------------------------------------
1.6000 7.3059 9.2990
------------------------------------
1.8000 9.2990 11.7778
------------------------------------
```

At x=2.0000, y=11.7778

Test Case

ODE	Interval [x0 ,xend]	Initial Condition (x0 ,y0)	Step Size h	Approximate Solution (y at xend)	Exact Solution
dy/dx=x+y	[0,2]	y(0)=1	0.1	22.14171037	22.14170957
dy/dx=x−y	[0,1]	y(0)=2	0.05	1.035213	Exact not computed analytically

2.Runge-Kutta fourth order method for system of ODEs / 2nd order ODE

Working Principle
The RK4 method is an iterative method for solving ordinary differential equations (ODEs). For a system of ODEs or higher-order ODEs, they are reduced to a set of first-order ODEs, and the RK4 method is applied to each equation simultaneously.

Pseudocode
Case A: System of ODEs

1. Define the system of ODEs:

$$\frac{dy_1}{dx} = f_{1(x,y_1,y_2,\ldots,y_n)},$$
$$\frac{dy_2}{dx} = f_{2(x,y_1,y_2,\ldots,y_n)},$$

   Repeat for all equations in the system.

2. Set initial values for $(x, y_1, y_2, \ldots, y_n)$.
3. For each step i, compute:

$$k1_i = h \cdot f_{i(x,y_1,\ldots,y_n)},$$
$$k2_i = h \cdot f_i(x + \frac{h}{2}, y_1 + \frac{k1_1}{2}, \ldots, y_n + \frac{k1_n}{2}),$$
$$k3_i = h \cdot f_i(x + \frac{h}{2}, y_1 + \frac{k2_1}{2}, \ldots, y_n + \frac{k2_n}{2}),$$
$$k4_i = h \cdot f_i(x + h, y_1 + k3_1, \ldots, y_n + k3_n)$$

4. Update each variable:

$$y_i = y_i + \frac{1}{6}\left(k1_i + 2k2_i + 2k3_i + k4_i\right),$$

5. Increment x by h and repeat until the desired x_{end} is reached.

Case B: Second-Order ODE

1. Convert the second-order ODE into a system of two first-order ODEs. For example,

$$\frac{d^2y}{dx^2} = f\left(x, y, \frac{dy}{dx}\right),$$

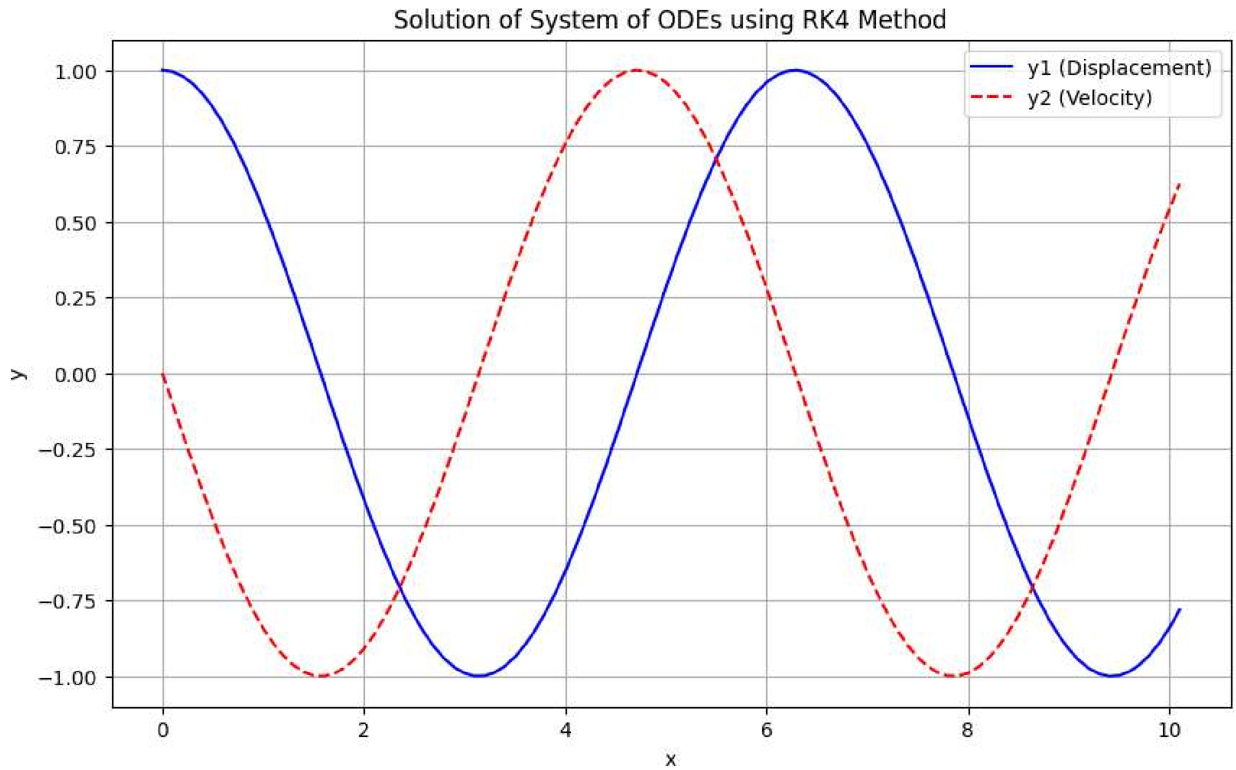
$$\frac{dy_1}{dx} = y_2,$$
$$\frac{dy_2}{dx} = f(x, y_1, y_2),$$

2. Solve the system of ODEs using the RK4 method as outlined above.

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

# Define the system of ODEs
def f1(x, y1, y2):
```

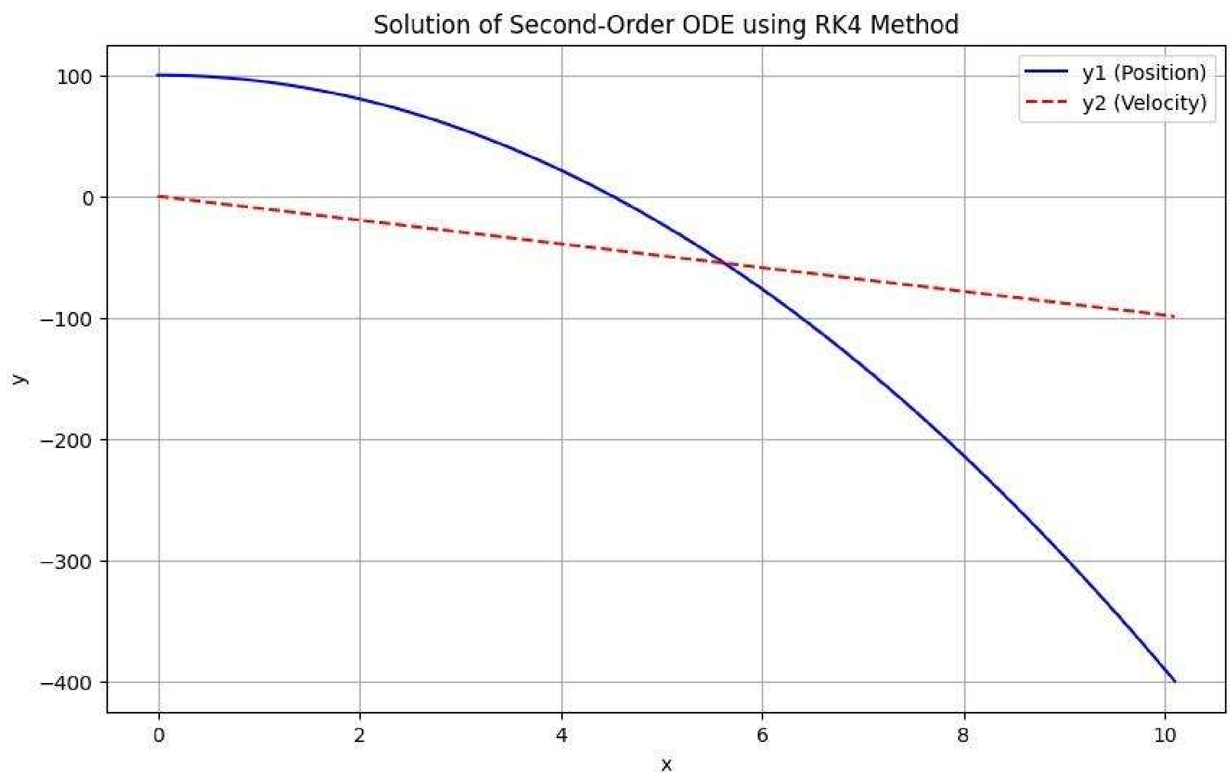
```
    return y2  # Example: dy1/dx = y2

def f2(x, y1, y2):
    return -y1  # Example: dy2/dx = -y1 (Simple harmonic motion)

# RK4 Method for System of ODEs
def rk4_system(f1, f2, x0, y10, y20, h, x_end):
    x_values = [x0]
    y1_values = [y10]
    y2_values = [y20]

    x = x0
    y1 = y10
    y2 = y20

    while x < x_end:
        k1_y1 = h * f1(x, y1, y2)
        k1_y2 = h * f2(x, y1, y2)

        k2_y1 = h * f1(x + h/2, y1 + k1_y1/2, y2 + k1_y2/2)
        k2_y2 = h * f2(x + h/2, y1 + k1_y1/2, y2 + k1_y2/2)

        k3_y1 = h * f1(x + h/2, y1 + k2_y1/2, y2 + k2_y2/2)
        k3_y2 = h * f2(x + h/2, y1 + k2_y1/2, y2 + k2_y2/2)

        k4_y1 = h * f1(x + h, y1 + k3_y1, y2 + k3_y2)
        k4_y2 = h * f2(x + h, y1 + k3_y1, y2 + k3_y2)

        y1 = y1 + (k1_y1 + 2*k2_y1 + 2*k3_y1 + k4_y1) / 6
        y2 = y2 + (k1_y2 + 2*k2_y2 + 2*k3_y2 + k4_y2) / 6
        x = x + h

        x_values.append(x)
        y1_values.append(y1)
        y2_values.append(y2)

    return x_values, y1_values, y2_values

# Inputs
x0 = 0       # Initial x
y10 = 1      # Initial y1
y20 = 0      # Initial y2
h = 0.1      # Step size
x_end = 10   # End point of x

# Solve the system of ODEs
x_values, y1_values, y2_values = rk4_system(f1, f2, x0, y10, y20, h,
x_end)

# Visualization
```

```
plt.figure(figsize=(10, 6))
plt.plot(x_values, y1_values, 'b-', label="y1 (Displacement)")
plt.plot(x_values, y2_values, 'r--', label="y2 (Velocity)")
plt.title("Solution of System of ODEs using RK4 Method")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()
plt.show()
```




```
# Convert 2nd order ODE to system of 1st order ODEs
def f1(x, y1, y2):
    return y2  # dy1/dx = y2

def f2(x, y1, y2):
    return -9.8  # Example: dy2/dx = -9.8 (acceleration due to
gravity)

# Inputs
x0 = 0        # Initial x
y10 = 100    # Initial y1 (Position)
y20 = 0       # Initial y2 (Velocity)
h = 0.1       # Step size
x_end = 10   # End point of x
```

```python
# Solve and visualize using the same RK4 function as above
x_values, y1_values, y2_values = rk4_system(f1, f2, x0, y10, y20, h,
x_end)

# Visualization
plt.figure(figsize=(10, 6))
plt.plot(x_values, y1_values, 'b-', label="y1 (Position)")
plt.plot(x_values, y2_values, 'r--', label="y2 (Velocity)")
plt.title("Solution of Second-Order ODE using RK4 Method")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()
plt.show()
```




```python
import sympy as sp

def f(x,y):
    return (y**2-x**2)/(y**2+x**2)

def func_input():
    function_str = input("Enter your function (use 'x' and 'y' as the
variables) (Example: (y**2 - x**2)/(y**2 + x**2)): ")
    x, y = sp.symbols('x y')
    sp_function = sp.sympify(function_str)
    func = sp.lambdify((x, y), sp_function, modules=['numpy'])
```

```python
    return func, sp_function

def rk4(func, x0, y0, xn, n):
    # Calculate step size
    h = (xn - x0) / n

    print('---------'*4)
    print('x0\ty0\tyn')
    print('---------'*4)
    for i in range(n):
        k1 = h * func(x0, y0)
        k2 = h * func(x0 + h / 2, y0 + k1 / 2)
        k3 = h * func(x0 + h / 2, y0 + k2 / 2)
        k4 = h * func(x0 + h, y0 + k3)
        k = (k1 + 2 * k2 + 2 * k3 + k4) / 6
        yn = y0 + k
        print(f'{x0:.4f}\t{y0:.4f}\t{yn:.4f}')
        print('---------'*4)
        x0 += h
        y0 = yn

    print(f"\nValue of y at x = {xn:.4f} is {yn:.4f}")

def main():
    print("Runge-Kutta 4th Order (RK-4) Method for Solving ODEs")
    print()

    function_str = "(y**2 - x**2) / (y**2 + x**2)"
    func = f
    x0 = 0.0
    y0 = 1.0
    xn = 2.0
    n = 10

    default = input("Use default values? (y/n): ").strip().lower() == 'y'
    if not default:
        func, function_str = func_input()
        x0 = float(input("Enter initial value of x (x0): "))
        y0 = float(input("Enter initial value of y (y0): "))
        xn = float(input("Enter value of x to evaluate the solution (xn): "))
        n = int(input("Enter number of steps: "))

    print(f"Function: f(x, y) = {function_str}")
    print(f"Initial Conditions: x0 = {x0}, y0 = {y0}")
    print(f"Evaluate at x = {xn}, Steps = {n}\n")

    # Solve using RK-4 method
    rk4(func, x0, y0, xn, n)
```

```python
if __name__ == "__main__":
    main()
```

```
Runge-Kutta 4th Order (RK-4) Method for Solving ODEs

Use default values? (y/n): y
Function: f(x, y) = (y**2 - x**2) / (y**2 + x**2)
Initial Conditions: x0 = 0.0, y0 = 1.0
Evaluate at x = 2.0, Steps = 10

------------------------------------
x0      y0      yn
------------------------------------
0.0000 1.0000 1.1960
------------------------------------
0.2000 1.1960 1.3753
------------------------------------
0.4000 1.3753 1.5331
------------------------------------
0.6000 1.5331 1.6691
------------------------------------
0.8000 1.6691 1.7839
------------------------------------
1.0000 1.7839 1.8781
------------------------------------
1.2000 1.8781 1.9521
------------------------------------
1.4000 1.9521 2.0064
------------------------------------
1.6000 2.0064 2.0412
------------------------------------
1.8000 2.0412 2.0565
------------------------------------

Value of y at x = 2.0000 is 2.0565
```

3.Solution of two-point boundary value problem using Shooting method

Working Principle :

The Shooting Method is a numerical technique to solve two-point boundary value problems (BVPs) by converting the BVP into an initial value problem (IVP) and iteratively solving it .

Boundary Value Problem (BVP)

A second-order ODE:

$\frac{d^2y}{dx^2}$=f(x,y,y′)

with boundary conditions:

y(a)=α,y(b)=β

Approach

1. Convert the second-order ODE into a system of two first-order ODEs:

   $\frac{dy_1}{dx}$=y2, $\frac{dy_2}{dx}$=f(x,y1,y2)

   where y1=y and y2=dy/dx.

2. Solve this system using an initial guess for y′(a)=y2(a) (denoted as s).
3. Use numerical integration (e.g., Runge-Kutta) to compute y(b) for the guessed s.
4. Adjust s iteratively (e.g., using Newton's method or the secant method) to ensure the computed y(b) matches the boundary condition y(b)=β.

Pseudocode

1. Input:

   Define the ODE as $\frac{d^2y}{dx^2}$=f(x,y,y′).
   - Specify boundary conditions y(a)=α,y(b)=β
   - Set the initial guess for s=y′(a).
2. Convert to a system of first-order ODEs:
   - y1′=y2 ,
   - y2′=f(x,y1,y2).
3. Iterative Procedure:
   1. Solve the system using RK4 or other numerical methods for the current guess of s.
   2. Compute the value of y(b)
   3. Compare y(b)with the target boundary condition β:
      - If y(b)) is close to β, stop.
      - Otherwise, update s using:
      - $s_{new} = s_{old} - \frac{y(b) - \beta}{Slope\ at\ s}$

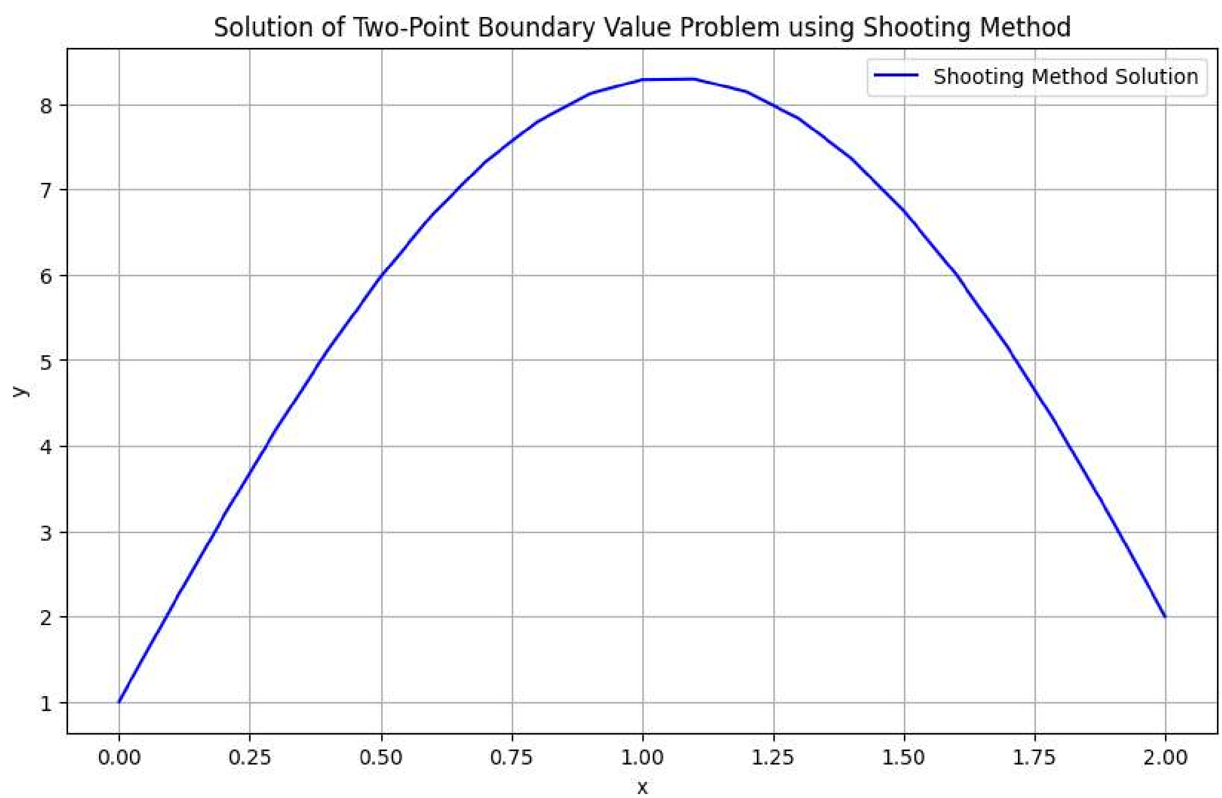
   (e.g., use the secant method to adjust s).

4. Output: y(x) values that satisfy the BVP.

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

# Define the ODE: d^2y/dx^2 = f(x, y, y')
def f(x, y1, y2):
    return -2 * y1 + np.cos(x)   # Example: d^2y/dx^2 = -2*y + cos(x)

# Runge-Kutta 4th order method for system of ODEs
def rk4_system(f, x0, y10, y20, h, x_end):
    x_values = [x0]
    y1_values = [y10]
    y2_values = [y20]

    x = x0
    y1 = y10
    y2 = y20

    while x < x_end:
        k1_y1 = h * y2
```

```
        k1_y2 = h * f(x, y1, y2)

        k2_y1 = h * (y2 + k1_y2 / 2)
        k2_y2 = h * f(x + h / 2, y1 + k1_y1 / 2, y2 + k1_y2 / 2)

        k3_y1 = h * (y2 + k2_y2 / 2)
        k3_y2 = h * f(x + h / 2, y1 + k2_y1 / 2, y2 + k2_y2 / 2)

        k4_y1 = h * (y2 + k3_y2)
        k4_y2 = h * f(x + h, y1 + k3_y1, y2 + k3_y2)

        y1 += (k1_y1 + 2*k2_y1 + 2*k3_y1 + k4_y1) / 6
        y2 += (k1_y2 + 2*k2_y2 + 2*k3_y2 + k4_y2) / 6
        x += h

        x_values.append(x)
        y1_values.append(y1)
        y2_values.append(y2)

    return x_values, y1_values, y2_values

# Shooting Method
def shooting_method(f, x0, x_end, y0, y_end, h, s_guess):
    def boundary_condition_error(s):
        # Solve the system for a given initial slope (s)
        _, y1_values, _ = rk4_system(f, x0, y0, s, h, x_end)
        return y1_values[-1] - y_end  # Difference between computed
and target y(b)

    # Initial guesses for the slope
    s1 = s_guess
    s2 = s1 + 0.1  # Slightly perturb the initial guess

    # Compute boundary condition errors
    err1 = boundary_condition_error(s1)
    err2 = boundary_condition_error(s2)

    # Use the secant method to refine the guess
    while abs(err1) > 1e-5:
        s_new = s1 - err1 * (s2 - s1) / (err2 - err1)  # Secant
formula
        s1, s2 = s2, s_new
        err1, err2 = err2, boundary_condition_error(s_new)

    # Solve with the refined slope
    x_values, y1_values, y2_values = rk4_system(f, x0, y0, s1, h,
x_end)
    return x_values, y1_values, y2_values

# Inputs
```

```
x0 = 0          # Starting x
x_end = 2       # Ending x
y0 = 1          # Boundary condition y(a) = alpha
y_end = 2       # Boundary condition y(b) = beta
h = 0.1         # Step size
s_guess = 0     # Initial guess for y'(a)

# Solve the BVP using the Shooting Method
x_values, y_values, _ = shooting_method(f, x0, x_end, y0, y_end, h,
s_guess)

# Visualization
plt.figure(figsize=(10, 6))
plt.plot(x_values, y_values, 'b-', label="Shooting Method Solution")
plt.title("Solution of Two-Point Boundary Value Problem using Shooting
Method")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()
plt.show()
```



4.Solution of two-point boundary value problem using finite difference method

Working Principle :The Finite Difference Method (FDM) solves two-point boundary value problems (BVPs) by discretizing the differential equation into a system of linear algebraic equations. These equations are then solved numerically to obtain the solution at discrete points.

Boundary Value Problem (BVP)

A second-order ODE:

$\frac{d^2y}{dx^2}$=f(x,y,y′) a≤x≤b

with boundary conditions:

y(a)=α,y(b)=β

Finite Difference Approximation

1. Discretize the domain into n+1n+1n+1 equally spaced points:

   x0=a, x1, x2, ..., x_{n-1}, x_n=b

   with step size h=$\frac{b-a}{n}$

2. Replace derivatives with finite differences:
   - Approximation for $\frac{d^2y}{dx^2}$:

     $$\frac{d^2y}{dx^2} = \frac{y_{i-1} - y_i + y_{i+1}}{h^2}$$

   - At each interior point xi, the BVP becomes a linear equation.
3. Solve the resulting system of n−1 linear equations with boundary conditions applied at x0 and xn.

Pseudocode

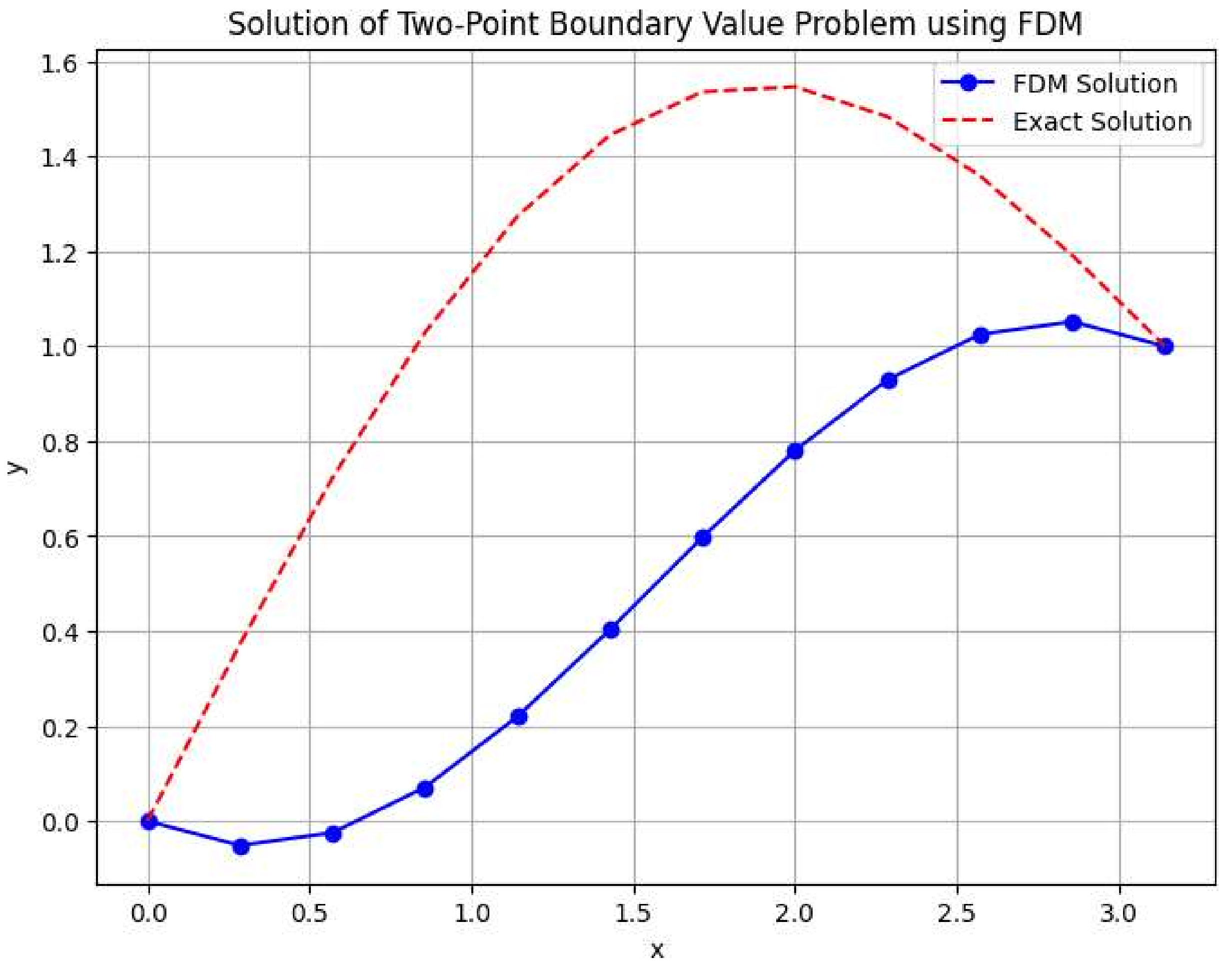
1. Input:
   - Define the second-order ODE as $\frac{d^2y}{dx^2}$=f(x,y,y′)
   - Specify boundary conditions y(a)=α,y(b)=β.
   - Choose the number of grid points n and compute the step size h.
2. Discretize the domain:
   - Divide [a,b] into n+1 points: x0,x1,...,xn
3. Construct the system of equations:
   - For each interior point xi, use the finite difference approximation for $\frac{d^2y}{dx^2}$.
   - Form a tridiagonal matrix representing the system of linear equations.
4. Solve the linear system: Use a numerical solver like numpy.linalg.solve() to find yiy_iyi values at the grid points.
5. Output: The solution y(x) at all grid points.

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

# Define the function f(x) on the right-hand side of d^2y/dx^2 = f(x)
def f(x):
    return np.cos(x)  # Example: f(x) = cos(x)

# Finite Difference Method for BVP
def finite_difference_method(a, b, alpha, beta, n):
    """
    Solve the two-point BVP using the finite difference method.

    Parameters:
    a, b: Endpoints of the interval [a, b]
    alpha, beta: Boundary conditions y(a) = alpha, y(b) = beta
    n: Number of interior points (grid points = n+1)

    Returns:
    x: Array of grid points
    y: Array of solution values at the grid points
    """
    h = (b - a) / (n + 1)  # Step size
    x = np.linspace(a, b, n + 2)  # Grid points including boundaries

    # Set up the coefficient matrix (tridiagonal matrix)
    A = np.zeros((n, n))
    for i in range(n):
        A[i, i] = -2 / h**2  # Diagonal elements
        if i > 0:
            A[i, i-1] = 1 / h**2  # Lower diagonal
        if i < n-1:
            A[i, i+1] = 1 / h**2  # Upper diagonal

    # Set up the right-hand side vector
    b_vec = np.array([f(xi) for xi in x[1:-1]])  # Function values at
interior points
    b_vec[0] -= alpha / h**2  # Incorporate y(a) = alpha
    b_vec[-1] -= beta / h**2  # Incorporate y(b) = beta

    # Solve the linear system
    y_interior = np.linalg.solve(A, b_vec)

    # Add boundary values to the solution
    y = np.zeros(n + 2)
    y[0] = alpha  # Boundary condition at x=a
    y[-1] = beta  # Boundary condition at x=b
    y[1:-1] = y_interior  # Interior points solution

    return x, y
# Inputs
```

```python
a = 0              # Left boundary x=a
b = np.pi          # Right boundary x=b
alpha = 0          # Boundary condition y(a) = 0
beta = 1           # Boundary condition y(b) = 1
n = 10             # Number of interior points

# Solve the BVP using the finite difference method
x, y = finite_difference_method(a, b, alpha, beta, n)

# Exact solution (if available) for comparison
def exact_solution(x):
    return np.sin(x) + x / np.pi  # Example exact solution for
comparison

y_exact = exact_solution(x)

# Print results
print("Grid points (x):", x)
print("FDM solution (y):", y)
print("Exact solution (if available):", y_exact)

# Visualization
plt.figure(figsize=(8, 6))
plt.plot(x, y, 'o-', label="FDM Solution", color="blue")
plt.plot(x, y_exact, 'r--', label="Exact Solution", color="red")
plt.title("Solution of Two-Point Boundary Value Problem using FDM")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()
plt.show()
```

```
Grid points (x): [0.         0.28559933 0.57119866 0.856798
1.14239733 1.42799666
 1.71359599 1.99919533 2.28479466 2.57039399 2.85599332 3.14159265]
FDM solution (y): [ 0.         -0.05136652 -0.0244701   0.07104483
0.21997477  0.40278886
  0.59721114  0.78002523  0.92895517  1.0244701   1.05136652  1.
]
Exact solution (if available): [0.         0.37264165 0.722459
1.02847685 1.27326836 1.4443669
 1.53527599 1.54599563 1.4830223  1.35882264 1.19082347 1.        ]
```

```
<ipython-input-5-832963bea283>:74: UserWarning: color is redundantly
defined by the 'color' keyword argument and the fmt string "r--" (->
color='r'). The keyword argument will take precedence.
  plt.plot(x, y_exact, 'r--', label="Exact Solution", color="red")
```


Solution of Two-Point Boundary Value Problem using FDM
FDM Solution
Exact Solution
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x