Solving Ordinary Differential Equations (ODEs) using Numerical Methods

Cosmic Charade

Solving Ordinary Differential Equations (ODEs)

Introduction

Ordinary Differential Equations (ODEs) are equations that describe the rate of change of a variable with respect to another variable, typically time. Many real-world systems and phenomena, such as population growth, motion of planets, electrical circuits, and heat transfer, can be modeled using ODEs. However, finding analytical (exact) solutions to ODEs is often not possible, especially for complex systems. In such cases, **numerical methods** are employed to approximate solutions to ODEs.

Two commonly used numerical methods for solving ODEs are:

1. **Euler's Method**: A first-order numerical procedure for solving ordinary differential equations.
2. **Runge-Kutta Methods**: A family of iterative methods for solving ODEs. The **4th-order Runge-Kutta method (RK4)** is particularly popular due to its accuracy and efficiency.

In this guide, we will explore these methods in detail and explain how to implement them using Python.

Euler's Method

Euler's method is a simple and intuitive numerical technique for solving ODEs. It is based on the idea of using the slope of the function at a given point to estimate the function's value at the next point.

Mathematical Formulation:

Consider a first-order ODE of the form:

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

The solution to this equation can be approximated using Euler's method as follows:

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

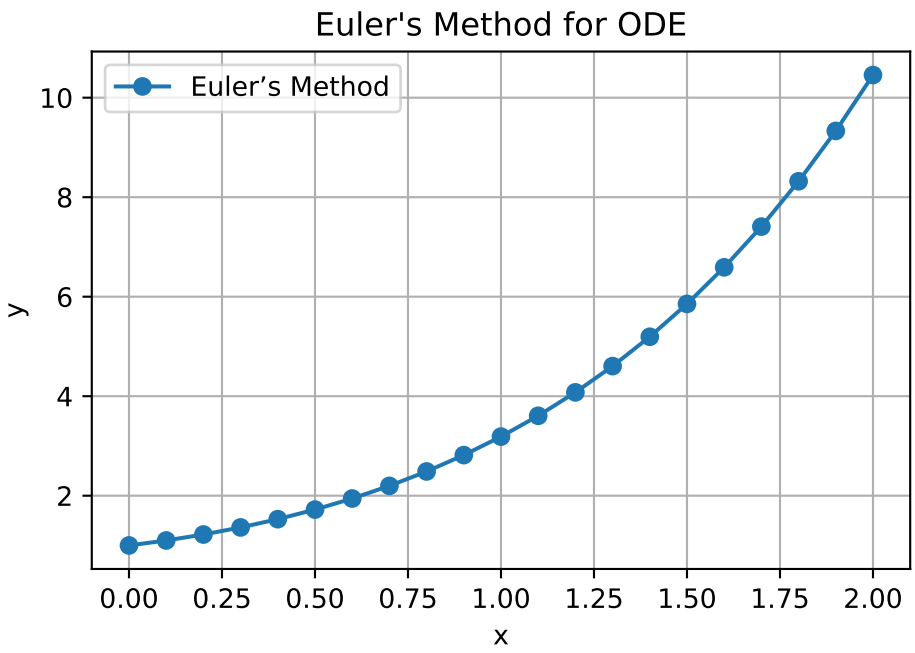
Where:

- y_n is the solution at the current step.
- y_{n+1} is the solution at the next step.
- h is the step size.
- $f(x_n, y_n)$ is the derivative of y with respect to x at x_n.

Euler's Method Code Implementation:

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

# Define the differential equation dy/dx = f(x, y)
def f(x, y):
    return x + y

# Euler's Method Implementation
def euler_method(f, x0, y0, h, n_steps):
```

```
    x_values = [x0]
    y_values = [y0]

    for i in range(n_steps):
        y_next = y_values[-1] + h * f(x_values[-1], y_values[-1])
        x_next = x_values[-1] + h

        x_values.append(x_next)
        y_values.append(y_next)

    return np.array(x_values), np.array(y_values)

# Parameters
x0 = 0  # initial value of x
y0 = 1  # initial value of y
h = 0.1  # step size
n_steps = 20  # number of steps

# Compute the solution using Euler's method
x_vals, y_vals = euler_method(f, x0, y0, h, n_steps)

# Plot the results
plt.plot(x_vals, y_vals, label='Euler's Method', marker='o')
plt.title("Euler's Method for ODE")
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
plt.legend()
plt.show()
```



Euler's Method Implementation

Function: `euler_method(f, x0, y0, h, n_steps)`

This function numerically solves an ODE $\frac{dy}{dx} = f(x, y)$ using **Euler's Method**.

1. **Parameters**:
   - `f`: The function representing the ODE $\frac{dy}{dx}$.
   - `x0`: Initial value of x.
   - `y0`: Initial value of y.
   - `h`: Step size for x (how much to increment x in each iteration).
   - `n_steps`: Number of iterations or steps to take.
2. **Initialization**:
   - `x_values = [x0]`: A list that starts with the initial x value.
   - `y_values = [y0]`: A list that starts with the initial y value.
3. **Iterative Process** (Euler's Method):

- The loop runs for `n_steps` iterations.
- Inside the loop, for each step iii:
  - **Calculate** y_{n+1}: `y_next = y_n + h \cdot f(x_n, y_n)` computes the next value of y, using the Euler method formula.
  - **Calculate** x_{n+1}: `x_next = x_n + h` increments the x value by the step size h.
- Append these newly computed values (`x_next` and `y_next`) to `x_values` and `y_values`.

4. **Return Values**:
   - The function returns the computed values as NumPy arrays, making it easier to work with them later in the plotting or further calculations.

Computing the Solution

```
x_vals, y_vals = euler_method(f, x0, y0, h, n_steps)
```

- This line calls the `euler_method` function with the defined parameters:
  - `f`: The ODE function.
  - `x0 = 0`: Initial value of x.
  - `y0 = 1`: Initial value of y.
  - `h = 0.1`: Step size.
  - `n_steps = 20`: Number of steps.
- It returns the arrays `x_vals` (all x values) and `y_vals` (corresponding y values) after applying Euler's method for the specified number of steps.

4th-Order Runge-Kutta Method (RK4)

The Runge-Kutta method is a higher-order numerical method that improves the accuracy of Euler's method by considering additional intermediate points. The **4th-order Runge-Kutta method** is the most widely used, offering a good balance between accuracy and computational efficiency.

Mathematical Formulation:

The 4th-order Runge-Kutta method estimates the solution at the next step using a weighted average of four increments:

$$k_1 = h \cdot f(x_n, y_n)$$

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

$$k_3 = h \cdot f\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) \quad k_4 = h \cdot f(x_n + h, y_n + k_3)$$

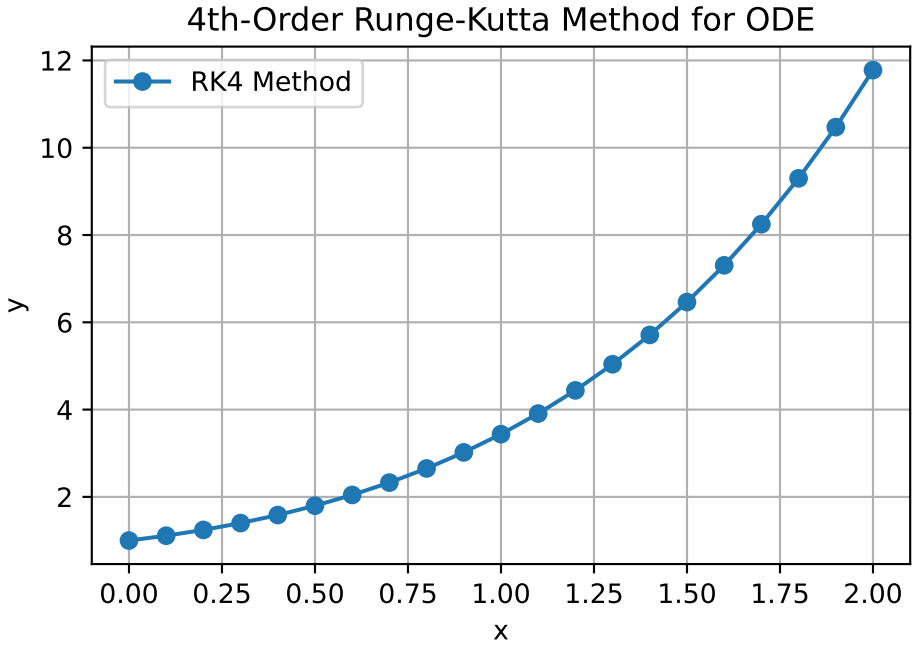
The next value of y is then given by:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Runge-Kutta Method Code Implementation:

```
# Runge-Kutta 4th-order method implementation
def runge_kutta_4(f, x0, y0, h, n_steps):
    x_values = [x0]
    y_values = [y0]

    for i in range(n_steps):
        x_n = x_values[-1]
        y_n = y_values[-1]

        k1 = h * f(x_n, y_n)
        k2 = h * f(x_n + h / 2, y_n + k1 / 2)
        k3 = h * f(x_n + h / 2, y_n + k2 / 2)
        k4 = h * f(x_n + h, y_n + k3)

        y_next = y_n + (1/6) * (k1 + 2*k2 + 2*k3 + k4)
        x_next = x_n + h

        x_values.append(x_next)
        y_values.append(y_next)
```

```
    return np.array(x_values), np.array(y_values)

# Compute the solution using RK4 method
x_vals_rk4, y_vals_rk4 = runge_kutta_4(f, x0, y0, h, n_steps)

# Plot the results
plt.plot(x_vals_rk4, y_vals_rk4, label='RK4 Method', marker='o')
plt.title("4th-Order Runge-Kutta Method for ODE")
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
plt.legend()
plt.show()
```



Function: `runge_kutta_4(f, x0, y0, h, n_steps)`

This function implements the **Runge-Kutta 4th Order (RK4) Method** to numerically solve an ODE of the form $\frac{dy}{dx} = f(x, y)$. The RK4 method is more accurate than Euler's method, as it uses multiple intermediate calculations to approximate the solution.

Parameters:

- `f`: The function representing the ODE $\frac{dy}{dx}$.
- `x0`: Initial value of x.
- `y0`: Initial value of y.
- `h`: Step size (how much to increment x in each iteration).
- `n_steps`: The number of steps to take for the approximation.

Initialization:

- `x_values = [x0]`: A list initialized with the initial x value.
- `y_values = [y0]`: A list initialized with the initial y value.

Iterative Process (RK4 Method):

The loop runs for `n_steps` iterations, updating the y values using the RK4 formula at each step:

1. **Initial values**:
   - `x_n = x_values[-1]`: Current x value.
   - `y_n = y_values[-1]`: Current y value.
2. **RK4 Calculations**: RK4 uses four weighted slopes to estimate the next value of y.
   - `k1 = h \cdot f(x_n, y_n)`: This is the slope at the beginning of the interval.
   - `k2 = h \cdot f(x_n + h/2, y_n + k1/2)`: This calculates the slope at the midpoint, using the previously computed slope k1k1k1.
   - `k3 = h \cdot f(x_n + h/2, y_n + k2/2)`: Similar to $k2$, but uses k2 for the calculation.
   - `k4 = h \cdot f(x_n + h, y_n + k3)`: This is the slope at the end of the interval, using k3 for the computation.
3. **Compute Next Values**:
   - **Next** yyy Value: `y_next = y_n + (1/6) * (k1 + 2*k2 + 2*k3 + k4)`
     - The next yyy value is computed as a weighted average of the four slopes.
     - k1 and k4 are weighted less, while k2 and k3 are weighted more, giving the overall slope a more accurate estimate.
   - **Next** xxx Value: `x_next = x_n + h`: Increment x by the step size h.

4. **Store Next Values**:
    - Append `x_next` to `x_values`.
    - Append `y_next` to `y_values`.

Return:

- The function returns two NumPy arrays: `x_values` and `y_values`, which represent the computed values of x and the corresponding approximate values of y.

Computing the Solution

```
x_vals_rk4, y_vals_rk4 = runge_kutta_4(f, x0, y0, h, n_steps)
```

This line calls the `runge_kutta_4` function with the provided parameters:

- `f`: The function representing the ODE $\frac{dy}{dx}$.
- `x0 = 0`: Initial x value.
- `y0 = 1`: Initial y value.
- `h = 0.1`: Step size.
- `n_steps = 20`: Number of steps.

The function computes and returns arrays `x_vals_rk4` and `y_vals_rk4`, which hold the x and y values for each step, calculated using the RK4 method.

Comparison of Euler's Method and Runge-Kutta Method

- **Euler's Method**: Simple but not very accurate, especially for larger step sizes. It only considers the slope at the current point to estimate the next value.
- **Runge-Kutta Method (RK4)**: More accurate because it uses multiple intermediate steps to estimate the next value. It provides a better approximation for the same step size compared to Euler's method.

Conclusion

Solving Ordinary Differential Equations (ODEs) numerically is essential for modeling real-world systems where analytical solutions are hard to find. In this guide, we explored two widely used methods:

1. **Euler's Method**: Simple and intuitive but less accurate.
2. **Runge-Kutta Method (RK4)**: A higher-order method that improves accuracy while maintaining computational efficiency.

Both methods are important tools in scientific computing, and understanding their implementation provides insight into how numerical approximations work in the context of differential equations. The code implementations provided here demonstrate how to apply these methods to approximate the solution of an ODE, with visualizations to aid understanding.