Ordinary Differential Equations

17 November 2025

Presented by

24083010055 Angelina N.P.D.P.

24083010076 Indy D.M.A.

24083010080 Retno Puji A.

Table of Content

17 November 2025

The motion of Pendulum

What is a pendulum?

A pendulum is a simple system consisting of a mass (the bob) attach to a string that swings back and forth under the influence of gravity.

Main Forces

- Gravity (mg) : pulls it downward
- Air drag ($c\ell\dot{\theta}$) : slows down the motion
- Tension (T) : keeps it moving in a circular path

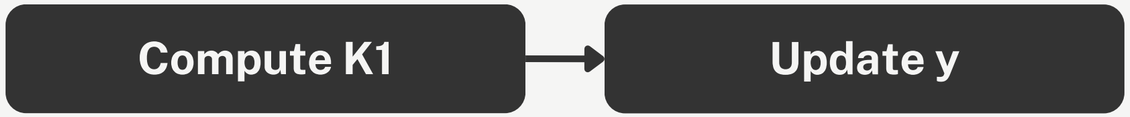
$$\ddot{\theta}+mc\dot{\theta}+\ell g\sin\theta=0$$

This equation describes how the pendulum’s angle evolves over time by incorporating the effects of air resistance, the gradual loss of energy in each swing, and the nonlinear behavior that become significant when the pendulum moves with a large amplitude.

Taylor Methods

Euler’s methods estimates the next value of a solution by using the slope at the current point, advancing forward in small steps, and applying the derivative as a constant guide over each interval, producing a simple but less accurate approximation that becomes smoother and more reliable only when small step sizes are used.

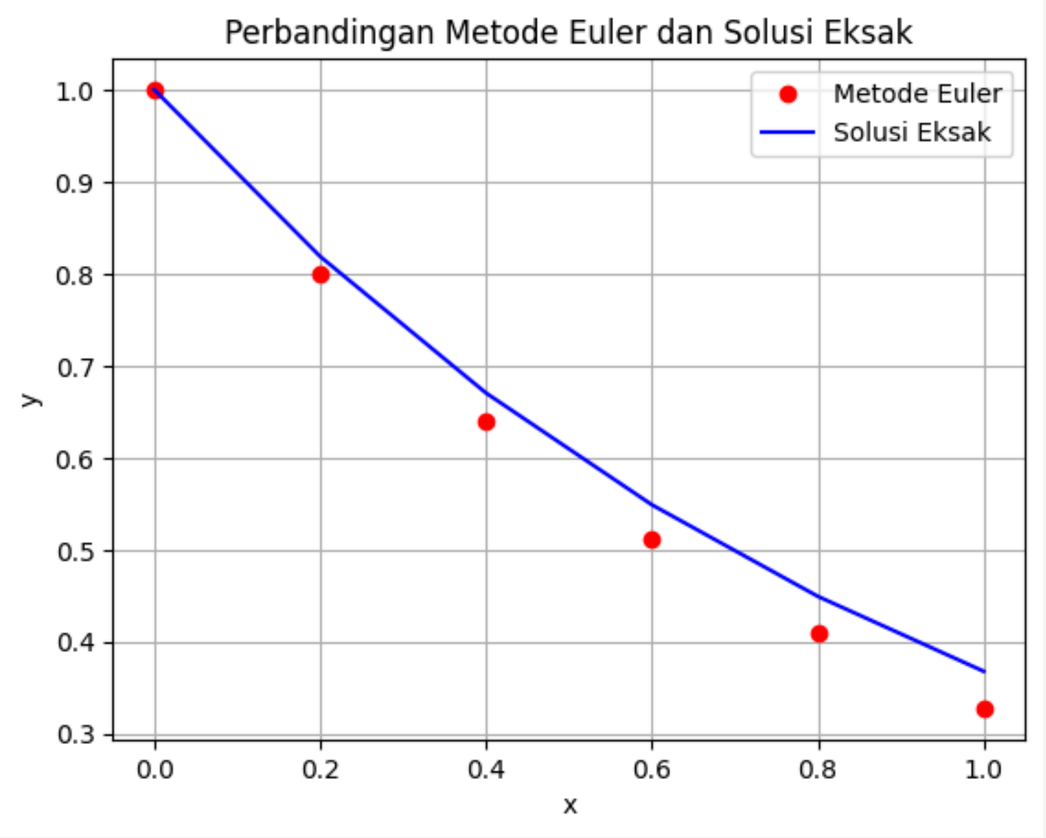
Method steps :



```python
# METODE EULER
for n in range(N):
    y[n + 1] = y[n] + dx * f(x[n], y[n])
    print(f"x = {x[n + 1]:.2f}, y = {y[n + 1]:.6f}")

# Solusi eksak: y = e^{-x}
y_exact = np.exp(-x)

# Visualisasi
plt.plot(x, y, 'or', label='Metode Euler')
plt.plot(x, y_exact, '-b', label='Solusi Eksak')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.title('Perbandingan Metode Euler dan Solusi
plt.show()
```

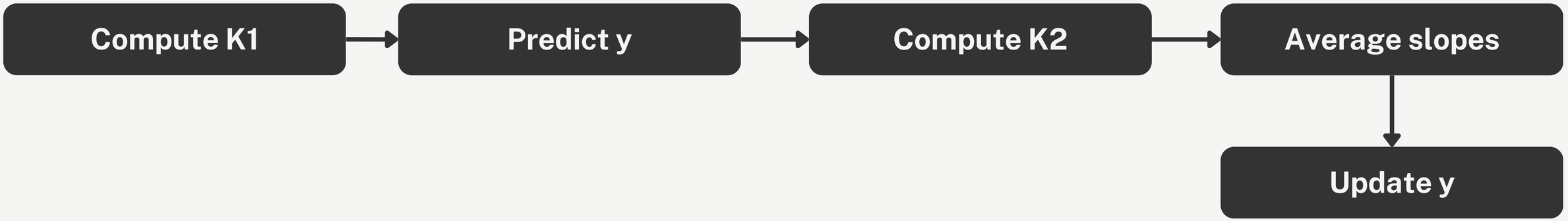
```
x = 0.20, y = 0.800000
x = 0.40, y = 0.640000
x = 0.60, y = 0.512000
x = 0.80, y = 0.409600
x = 1.00, y = 0.327680
```

$$y_{n+1} = y_n + hf(x_n, y_n)$$



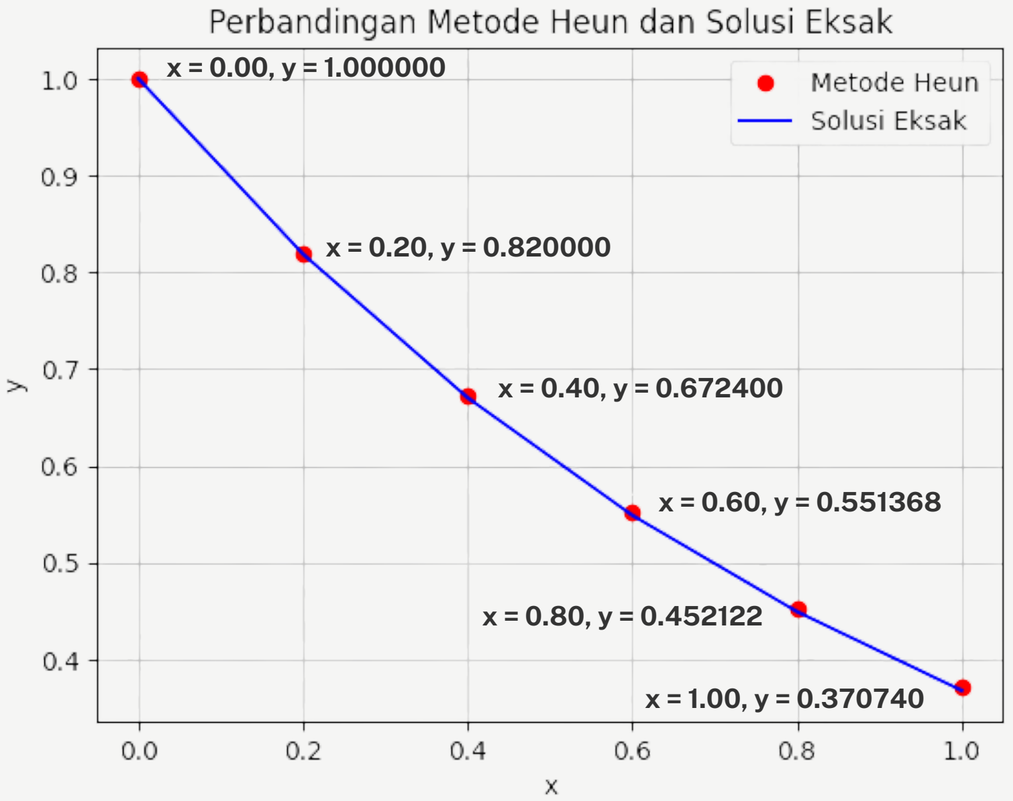
The Heun method, also known as the Improved Euler method, is used to correct the weaknesses of the standard Euler method, resulting in a more accurate value of y. This improvement comes from calculating two slopes, k1 and k2, so the final result is closer to the exact solution.

Method steps :



```
#METODE_HEUN
for n in range(N):
    k1 = f(x[n], y[n])
    k2 = f(x[n+1], y[n] + h*k1)
    y[n+1] = y[n] + (h/2) * (k1 + k2)
    print(f"x = {x[n + 1]:.2f}, y = {y[n + 1]:.6f}")

#SOLUSI_EKSAK
y_exact = np.exp(-x)

#VISUALISASI
plt.plot(x, y, 'or', label='Metode Heun')
plt.plot(x, y_exact, '-b', label='Solusi Eksak')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.title('Perbandingan Metode Heun dan Solusi Eksak')
plt.show()
```

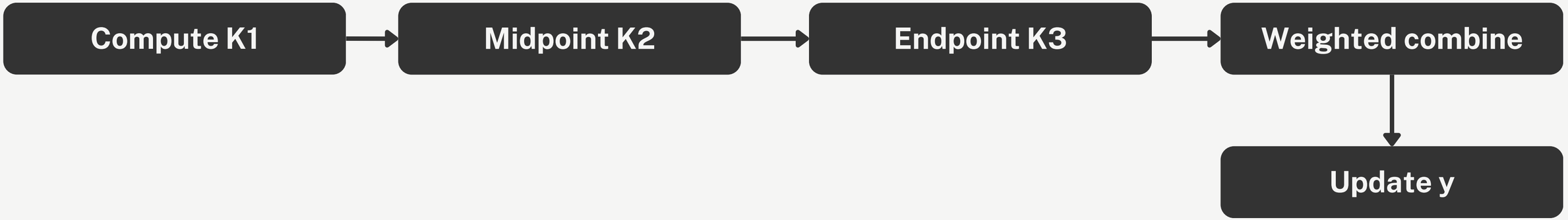
$$
\begin{aligned}
k_1 &= f(t_i, y_i) \\
k_2 &= f(t_{i+1}, y_i + hk_1) \\
y_{i+1} &= y_i + \frac{h}{2}(k_1 + k_2).
\end{aligned}
$$



Foundations for Runge-Kutta

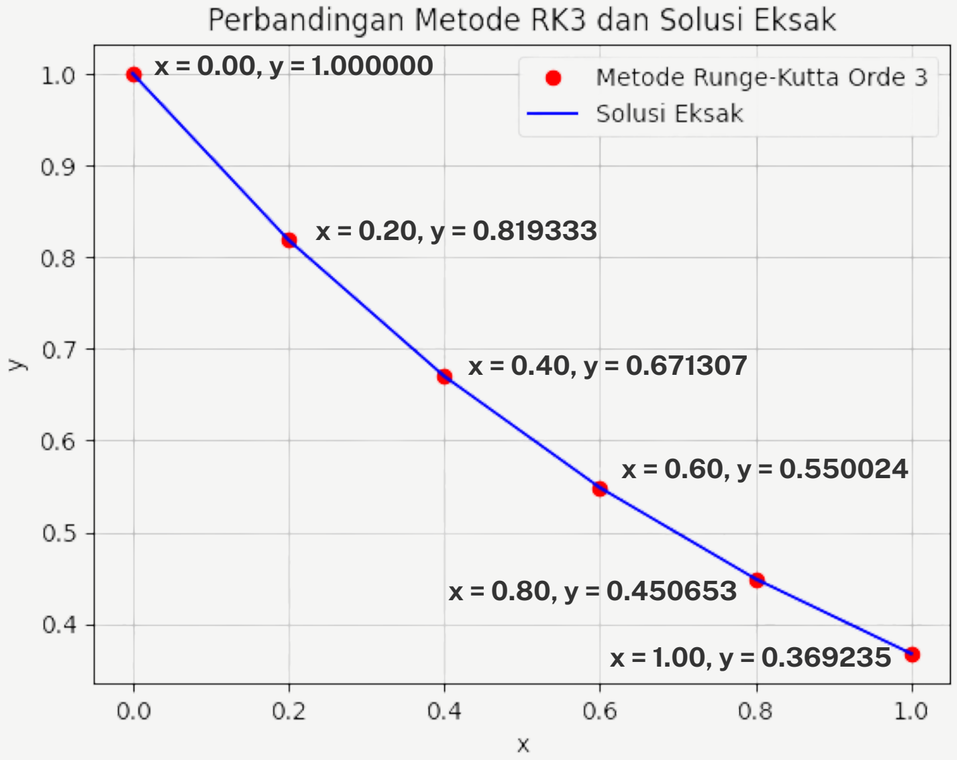
RK3 is an extension of The Heun method by adding one more slope calculation, which increases accuracy without making the computation too heavy. It uses three slopes in k1, k2, and k3 to produce smoother and more reliable results.

Method steps :



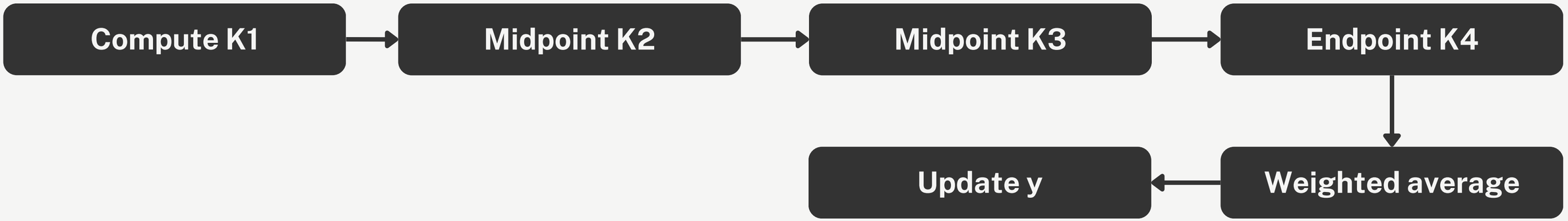
```python
#METODE_RUNGE-KUTTA_ORDE_3
for n in range(N):
    k1 = f(x[n], y[n])
    k2 = f(x[n] + h/2, y[n] + h*k1/2)
    k3 = f(x[n+1], y[n] + h*k2)
    y[n + 1] = y[n] + (h/6) * (k1 + 4*k2 + k3)
    print(f"x = {x[n+1]:.2f}, y = {y[n+1]:.6f}")

#SOLUSI_EKSAK
y_exact = np.exp(-x)

#VISUALISASI
plt.plot(x, y, 'or', label='Metode Runge-Kutta Orde 3')
plt.plot(x, y_exact, '-b', label='Solusi Eksak')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.title('Perbandingan Metode RK3 dan Solusi Eksak')
plt.show()
```

$$
\begin{aligned}
k_1 &= f(t_i, y_i) \\
k_2 &= f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_1\right) \\
k_3 &= f(t_{i+1}, y_i + hk_2) \\
y_{i+1} &= y_i + \frac{h}{6}\left[k_1 + 4k_2 + k_3\right].
\end{aligned}
$$



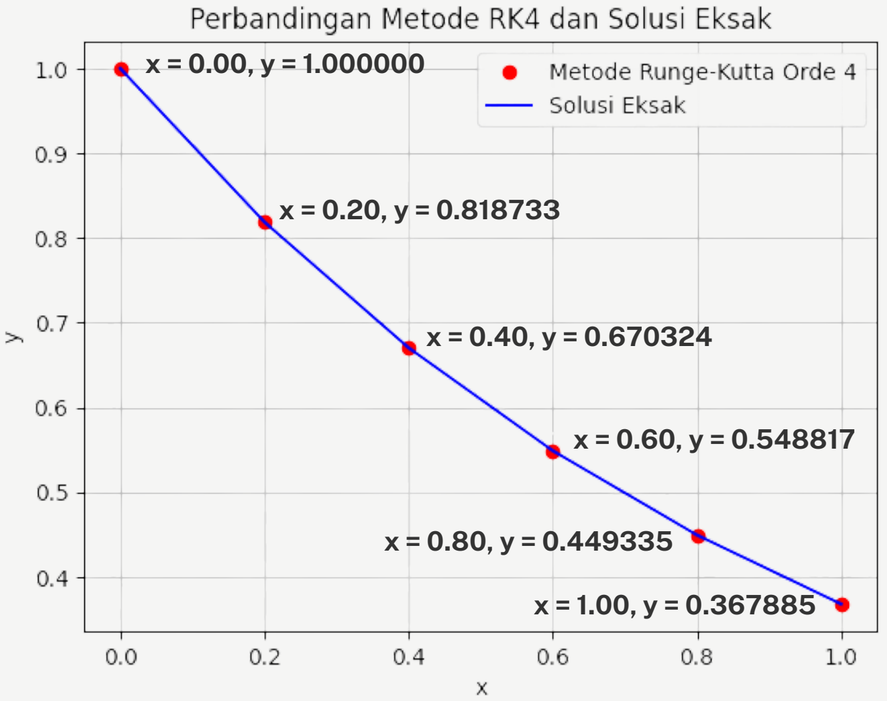
RK4 is the most commonly used method because it provides very high accuracy and remains stable even when the step size (h) is relatively large. RK4 calculates four slopes in k1, k2, k3, and k4 to produce values of y that are extremely close to the exact solution.

Method steps :



```python
#METODE_RUNGE-KUTTA_ORDE_4
for n in range(N):
    k1 = f(x[n], y[n])
    k2 = f(x[n] + h/2, y[n] + h*k1/2)
    k3 = f(x[n] + h/2, y[n] + h*k2/2)
    k4 = f(x[n] + h, y[n] + h*k3)
    y[n + 1] = y[n] + (h/6) * (k1 + 2*k2 + 2*k3 + k4)
    print(f"x = {x[n+1]:.2f}, y = {y[n+1]:.6f}")

#SOLUSI_EKSAK
y_exact = np.exp(-x)

#VISUALISASI
plt.plot(x, y, 'or', label='Metode Runge-Kutta Orde 4')
plt.plot(x, y_exact, '-b', label='Solusi Eksak')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.title('Perbandingan Metode RK4 dan Solusi Eksak')
plt.show()
```

$$
\begin{aligned}
k_1 &= f(t_i, y_i) \\
k_2 &= f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_1\right) \\
k_3 &= f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_2\right) \\
k_4 &= f\left(t_i + h, y_i + hk_3\right) \\
y_{i+1} &= y_i + \frac{h}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right].
\end{aligned}
$$



Error Analysis

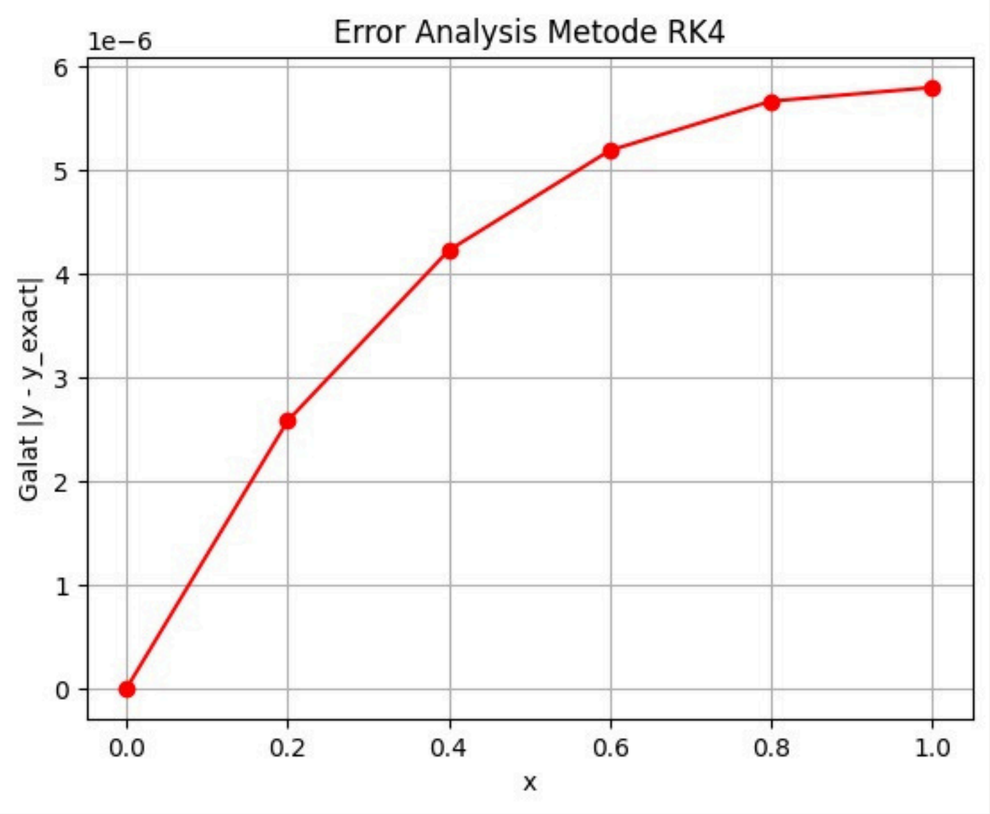
If the differential equation cannot be solved exactly, then a numerical method is used to approximate the solution. Error analysis is used to assess how accurate the approximation is and what factors affect its accuracy.

Two types of errors in the numeric ODE method:
1. Local Truncation Error (LTE)
2. Global Truncation Error (GTE)

The accuracy of numerical methods is determined by their order: LTE is generally O(h^{p+1}), while GTE is O(h^p). The higher the order, the faster the error decreases as h is reduced.

```python
# ERROR ANALYSIS
error = np.abs(y - y_exact)
for i in range(len(x)):
    print(f"x = {x[i]:.2f}, y_num = {y[i]:.6f}, y_exact = {y_exact[i]:.6f}, error = {error[i]:.6e}")

# Total error dan error maksimum
error_max = np.max(error)
print(f"\nError maksimum = {error_max:.6e}")

# Visualisasi
plt.figure()
plt.plot(x, error, 'o-r')
plt.xlabel('x')
plt.ylabel('Galat |y - y_exact|')
plt.title('Error Analysis Metode RK4')
plt.grid(True)
plt.show()
```

```
x = 0.00, y_num = 1.000000, y_exact = 1.000000, error = 0.000000e+00
x = 0.20, y_num = 0.818733, y_exact = 0.818731, error = 2.580255e-06
x = 0.40, y_num = 0.670324, y_exact = 0.670320, error = 4.225075e-06
x = 0.60, y_num = 0.548817, y_exact = 0.548812, error = 5.188807e-06
x = 0.80, y_num = 0.449335, y_exact = 0.449329, error = 5.664323e-06
x = 1.00, y_num = 0.367885, y_exact = 0.367879, error = 5.796954e-06
```



Adaptive Runge-Kutta Methods

Adaptive Runge Kutta is an RK method that automatically adjusts the step size when solving ODE, so that accuracy is maintained without excessive calculations.

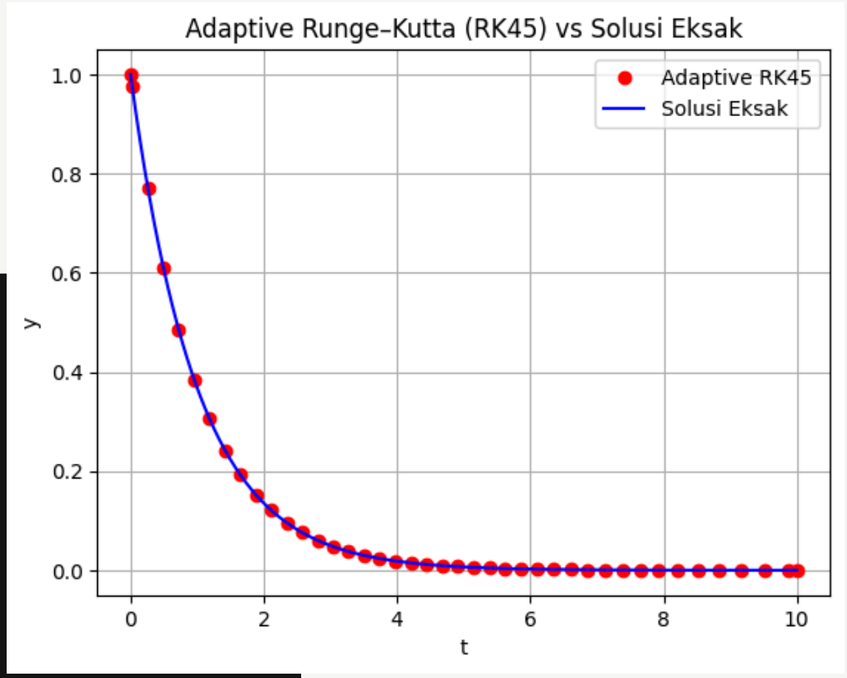
This method works with two solution estimates (high order and low order), and the difference between the two is used to estimate local error. In this way, the method can determine whether the current step is too large or too small.

This method calculates two solutions in one step and then uses the difference between them (yhigh - ylow) as an error estimate. If the error is too large, the step size is reduced to make the calculation more accurate. Conversely, if the error is small, the step size can be increased.

In this way, the method works more efficiently because it only uses small steps in difficult parts and large steps in stable parts, so that the process is faster but still has a good level of accuracy compared to RK4 with fixed steps.

```python
# METODE RUNGE-KUTTA ORDE 45
t_span = (0, 10)
y0_adapt = [1]

# Jalankan solver adaptif RK45
sol = solve_ivp(f, t_span, y0_adapt, method='RK45', rtol=1e-6, atol=1e-9)

# Solusi eksak untuk perbandingan
t_exact = np.linspace(0, 10, 1000)
y_exact_adapt = np.exp(-t_exact)

# Tampilkan sebagian hasil adaptif
for i in range(0, len(sol.t), max(1, len(sol.t)//10)):
    print(f"t = {sol.t[i]:.3f}, y = {sol.y[0][i]:.6f}")
print(f"\nJumlah langkah adaptif yang digunakan: {len(sol.t)}")

# Visualisasi
plt.figure()
plt.plot(sol.t, sol.y[0], 'or', label='Adaptive RK45')
plt.plot(t_exact, y_exact_adapt, '-b', label='Solusi Eksak')
plt.xlabel('t')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.title('Adaptive Runge–Kutta (RK45) vs Solusi Eksak')
plt.show()
```

```
t = 0.000, y = 1.000000
t = 0.724, y = 0.484874
t = 1.651, y = 0.191902
t = 2.578, y = 0.075900
t = 3.508, y = 0.029969
t = 4.441, y = 0.011784
t = 5.385, y = 0.004586
t = 6.353, y = 0.001741
t = 7.378, y = 0.000625
t = 8.519, y = 0.000200
t = 9.878, y = 0.000051

Jumlah langkah adaptif yang digunakan: 42
```



The motion of a pendulum follows Newton’s Second Law, which leads to an ODE that often cannot be solved exactly. Numerical methods are used instead: Taylor estimates derivatives, Heun averages two slopes, RK3 adds more accuracy, and RK4 provides highly precise results with four slope evaluations. Error analysis then measures how far the numerical solution is from the exact one, showing how accurate each method is. Higher-order methods give smaller errors as the step size (h) decreases. In Adaptive Runge–Kutta, (h) changes automatically based on the local error, allowing the solver to take large steps in smooth regions and small steps in sensitive areas, keeping the solution accurate and efficient.

LINK COLAB IMPLEMENTASI MATERI ODE

Terima kasih 🙏