ORDINARY DIFFERENTIAL EQUATIONS (ODES): INITIAL-VALUE PROBLEMS

by group 9

GROUP MEMBER

- Baihaqi Nur Muhammad
  24083010111
- Achmad Dany Gunawan
  24083010075
- Erik Saputra Rifki
  24083010069

ODE INITIAL VALUE PROBLEM STATEMENT

A differential equation is a relationship between a function, , its independent variable, , and any number of its
derivatives. An ordinary differential equation or ODE is a differential equation where the independent variable, and
therefore also the derivatives, is in one dimension

The purpose of ODE is:

1. Describing physical phenomena that change with respect to a single variable
2. Determining the system's solution from given initial conditions (Initial Value Problem)
3. Demonstrating the need for numerical methods when analytic solutions are not available

ORDINARY DIFFERENTIAL EQUATION (ODE)

$$F\left(x, f(x), \frac{df(x)}{dx}, \frac{d^2f(x)}{dx^2}, \frac{d^3f(x)}{dx^3}, \ldots, \frac{d^{n-1}f(x)}{dx^{n-1}}\right) = \frac{d^nf(x)}{dx^n},$$

This formula shows the general form of an n-th order ODE, relating the highest derivative to x, the function, and all lower-order derivatives together.

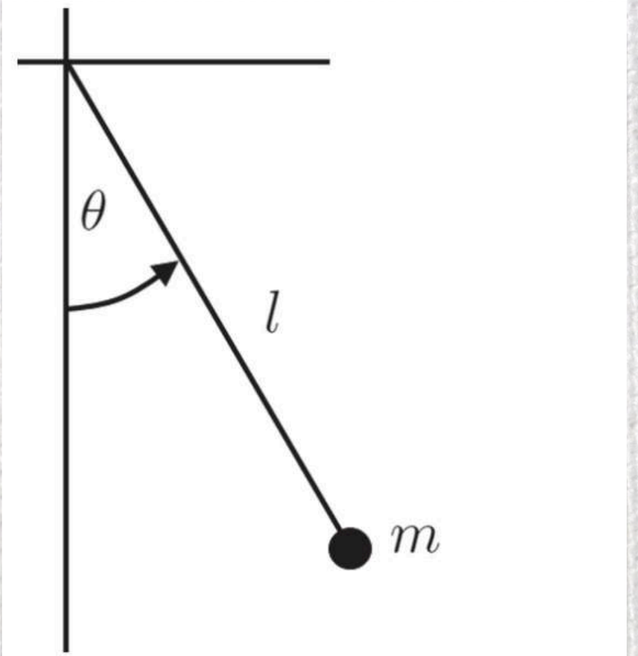
PARTIAL DIFFERENTIAL EQUATION (PDE)

$$\frac{\partial u(t,x,y,z)}{\partial t} = \alpha\left(\frac{\partial u(t,x,y,z)}{\partial x} + \frac{\partial u(t,x,y,z)}{\partial y} + \frac{\partial u(t,x,y,z)}{\partial z}\right).$$

This formula represents the heat equation, showing how temperature changes over time based on its spatial partial derivatives in the x, y, and z directions.

ORDINARY DIFFERENTIAL EQUATION (ODE)

EXAMPLE OF AN ODE APPLICATION



The given equation $= I\frac{d^2\theta(t)}{dt^2} = -g\theta(t)$

Given:

$\Theta(t)$ = Pendulum Angle

$\Theta''(t)$ = Angular Acceleration

$\Theta(t) = \cos\left(\sqrt{\frac{g}{l}}t\right)$

Question : Determine whether the LHS and RHS are identical!

Solution:

1. Determining the left-hand side (LHS)

Guide! :

First Derivative : $\cos(u) = -\sin(u)\times u'$,

Second Derivative: $-\sin(u) = -\cos(u)\times u'$

$\Theta(t) = \cos\left(\sqrt{\frac{g}{l}}t\right)$

$\Theta'(t) = -\sqrt{\frac{g}{l}}\sin\left(\sqrt{\frac{g}{l}}t\right)$

$\Theta''(t) = -\frac{g}{l}\cos\left(\sqrt{\frac{g}{l}}t\right)$

2. Calculate the right-hand side (RHS)

RHS → $I\frac{d^2\theta(t)}{dt^2} = -g\theta(t)$

---------- ÷I

$\frac{d^2\theta(t)}{dt^2} = -\frac{g}{l}\theta(t)$

$\frac{d^2\theta(t)}{dt^2} = -\frac{g}{l}\cos\left(\sqrt{\frac{g}{l}}t\right)$

Conclution : The computations confirm that the proposed cosine function satisfies the small-angle pendulum ODE and, with given initial conditions, yields the corresponding particular solution

REDUCTION OF ORDER

Reduction of order is a method that converts a higher-order ODE into a system of first-order ODEs, allowing the equation to be solved using numerical methods.

The purpose of Reduction of Order:

1. To convert a higher-order ODE into an equivalent system of first-order ODEs.
2. To make the equation solvable using numerical methods like Euler or Runge-Kutta.

REDUCTION OF ORDER
FIRST-ORDER SYSTEM REPRESENTATION

$$S(t) = \begin{bmatrix} f(t) \\ f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ \cdots \\ f^{(n-1)}(t) \end{bmatrix}.$$

The equation can be derived into

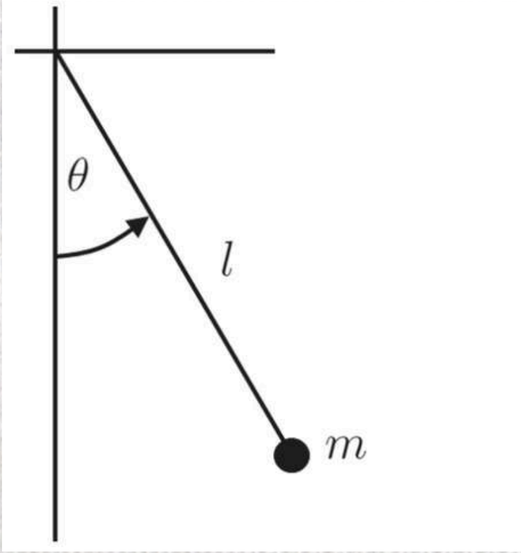
$$\frac{dS(t)}{dt} = \begin{bmatrix} f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ f^{(4)}(t) \\ \cdots \\ f^{(n)}(t) \end{bmatrix} = \begin{bmatrix} f^{(1)}(t) \\ f^{(2)}(t) \\ f^{(3)}(t) \\ f^{(4)}(t) \\ \cdots \\ F\left(t, f(t), f^{(1)}(t), \ldots, f^{(n-1)}(t)\right) \end{bmatrix} = \begin{bmatrix} S_2(t) \\ S_3(t) \\ S_4(t) \\ S_5(t) \\ \cdots \\ F\left(t, S_1(t), S_2(t), \ldots, S_{n-1}(t)\right) \end{bmatrix},$$

This equation shows how the derivative of the state vector S(t)S(t)S(t) is constructed. Each component of the vector is differentiated once, forming a system of first-order ODEs:

- S1'(t)=S2(t)
- S2'(t)=S3(t)
- S3'(t)=S4(t)
- ⋮
- Sn'(t)=F(t,S1(t),S2(t),...,Sn−1(t))

In this way, an nnn-th order ODE is converted into a system of first-order ODEs that can be solved using numerical methods such as Euler or Runge-Kutta.

REDUCTION OF ORDER
REDUCTION OF ORDER IN THE CASE OF THE PENDULUM.



→ Given the pendulum equation: $\Theta''(t) = -\frac{g}{l}\Theta(t)$

Given:

$S_1'(t) = \Theta'(t) = \dot{\Theta}(t) = S_2(t)$

$S_2'(t) = \Theta''(t)$

Question = Convert the second-order pendulum equation into a system of two first-order ODEs.

$S_2'(t) = \Theta''(t)$

$S_2'(t) = -\frac{g}{l}\Theta(t)$

$S_2'(t) = -\frac{g}{l}S_1'(t)$

→ Matrix Form:

$$\frac{dS(t)}{dt} = \begin{bmatrix} S'1(t) \\ S'2(t) \end{bmatrix} = \begin{bmatrix} S2(t) \\ -\frac{g}{l}\,S1(t) \end{bmatrix}$$

$$\frac{dS(t)}{dt} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l} & 0 \end{bmatrix} \begin{bmatrix} S1(t) \\ S2(t) \end{bmatrix}$$

$$\frac{dS(t)}{dt} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l} & 0 \end{bmatrix} S(t)$$

Because its form is already: $\frac{dS}{dt}$ **= AS(t),**

It can be concluded that the second-order pendulum equation has been successfully converted into a first-order form, and its final form is a linear ODE.

THE EULER METHOD

FIRST ORDER ODE AND EXPLICIT EULER METHOD

Consider a first-order ordinary differential equation (ODE):

$$\frac{dS(t)}{dt} = F(t, S(t))$$

F(t,S) is a function that returns the derivative (rate of change) of the state S at time t.

$$t_j = t_0 + jh, \quad j = 0, 1, \ldots, N$$

Without loss of generality, we assume:

$$t_0 = 0, \quad t_f = Nh$$

THE EULER METHOD

LINEAR APPROXIMATION AND EXPLICIT EULER FORMULA

Linear approximation of S(t) around tj evaluated at tj+1:

$$S(t_{j+1}) \approx S(t_j) + (t_{j+1} - t_j)\frac{dS(t_j)}{dt}$$

Since tj+1−tj=h and dS/dt=F(t,S):

$$S(t_{j+1}) = S(t_j) + h\,F(t_j, S(t_j))$$

Explicit Euler Formula:

$$S_{j+1} = S_j + h\,F(t_j, S_j)$$

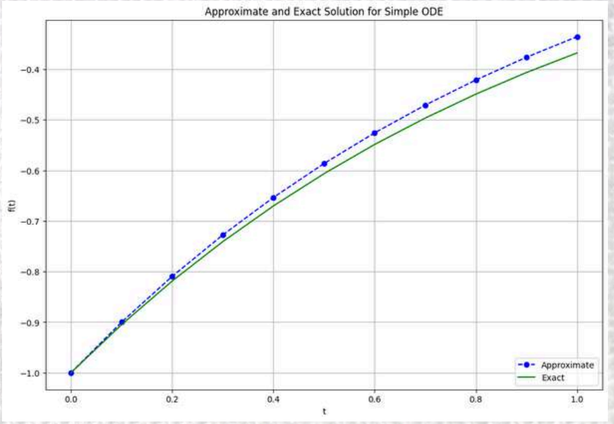
Starting from an initial value S0=S(t0), we can iterate this formula to obtain

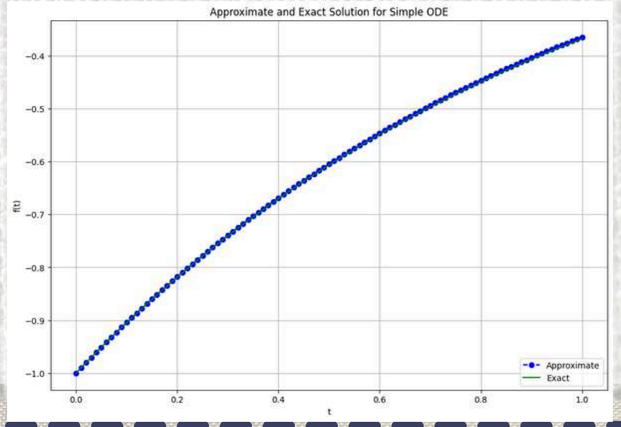
$$S_1, S_2, \ldots, S_N \approx S(t_1), S(t_2), \ldots, S(t_N)$$

Each step uses the slope at (tj,Sj) to "point" toward the next state and moves a distance h in that direction.

THE EULER METHOD
EXPLICIT EULER ALGORITHM AND PYTHON IMPLEMENTATION

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

f = lambda t, s: np.exp(-t)
h = 0.01
t = np.arange(0, 1 + h, h)
s0 = -1

# Explicit Euler Method
s = np.zeros(len(t))
s[0] = s0

for i in range(len(t) - 1):
    s[i + 1] = s[i] + h * f(t[i], s[i])

plt.figure(figsize=(12, 8))
plt.plot(t, s, 'bo--', label='Approximate')
plt.plot(t, -np.exp(-t), 'g', label='Exact')
plt.title('Approximate and Exact Solution for Simple ODE')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.grid()
plt.legend(loc='lower right')
plt.show()
```





THE EULER METHOD
APPROXIMATION QUALITY AND STEP SIZE EFFECT

The exact solution of the IVP is: $f(t) = -e^{-t}$

The numerical solution {fj} obtained by the Explicit Euler method is only an approximation

For h=0.1:

- The approximate curve follows the exact solution but with visible error.

If we repeat the process with a smaller step size, e.g. h=0.01:

- The numerical solution becomes closer to the exact solution.
- The grid is denser and the local linear approximation is more accurate.

The difference between approximate and exact values: $\text{error}_j = f_{\text{approx}}(t_j) - f_{\text{exact}}(t_j)$

NUMERICAL ERROR DAN INSTABILITY
NUMERICAL ERROR AND STABILITY IN EULER-TYPE METHODS

When solving ODEs numerically, we must consider:

- Accuracy
  - How close the numerical solution is to the exact solution.
  - Explicit Euler has global error of order O(h).
- Stability
  - How numerical errors behave as we march forward in time.
  - A method is stable if errors do not grow uncontrollably.

Test problem: linearized pendulum

$$\frac{dS}{dt} = AS(t), \quad A = \begin{bmatrix} 0 & 1 \\ -\omega^2 & 0 \end{bmatrix}$$

Parameters:

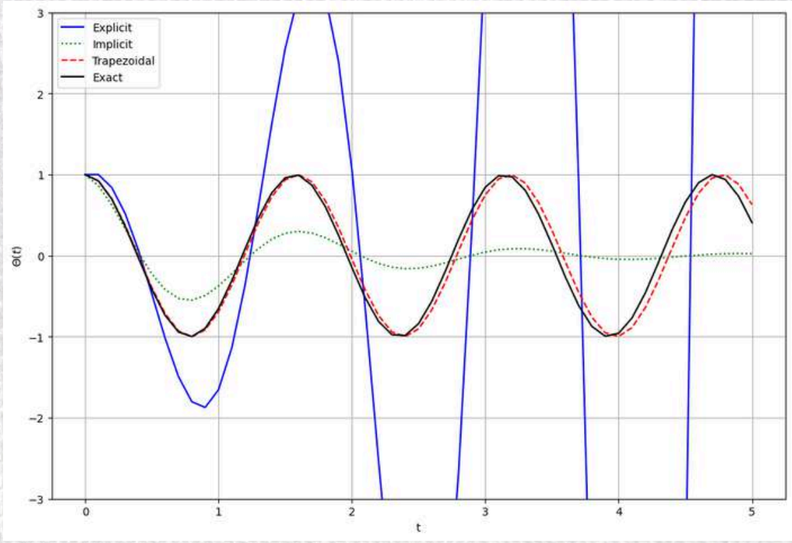
- Angular frequency: ω=4
- Initial condition: S(0)= [1, 0]T
- Time interval: 0≤t≤50
- Time step: h=0.1

Exact solution for the angle:

$$\theta(t) = \cos(\omega t)$$

NUMERICAL ERROR DAN INSTABILITY
PYTHON IMPLEMENTATION AND BEHAVIOUR OF THE METHODS

```python
import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt

h = 0.1
t = np.arange(0, 5.1, h)
w = 4
s0 = np.array([[1], [0]])

m_e = np.array([[1, h], [-w**2*h, 1]])
m_i = inv(np.array([[1, -h], [w**2*h, 1]]))
m_t = inv(np.array([[1, -h/2], [w**2*h/2, 1]])) @ \
      np.array([[1, h/2], [-w**2*h/2, 1]])

s_e = np.zeros((len(t), 2))
s_i = np.zeros((len(t), 2))
s_t = np.zeros((len(t), 2))
s_e[0,:] = s0.T; s_i[0,:] = s0.T; s_t[0,:] = s0.T

for j in range(len(t) - 1):
    s_e[j+1,:] = m_e @ s_e[j,:]
    s_i[j+1,:] = m_i @ s_i[j,:]
    s_t[j+1,:] = m_t @ s_t[j,:]

plt.figure(figsize=(12, 8))
plt.plot(t, s_e[:, 0], 'b-')
plt.plot(t, s_i[:, 0], 'g:')
plt.plot(t, s_t[:, 0], 'r--')
plt.plot(t, np.cos(w * t), 'k')

plt.ylim([-3, 3])
plt.xlabel('t')
plt.ylabel(r'$\Theta (t)$')
plt.legend(['Explicit', 'Implicit', 'Trapezoidal', 'Exact'])
plt.grid(True)
plt.show()
```



- Explicit Euler: Amplitude grows with time → unstable.
- Implicit Euler: Amplitude decays artificially → stable but overdamped.
- Trapezoidal: Nearly constant amplitude, small phase shift → good stability and accuracy.

PREDIKTOR-KOREKTOR METHODS AND RUNGE KUTTA

Numerical techniques for solving differential equations, which are mathematical formulas that describe the change in a value with respect to time or another variable. Both are used when analytical solutions (exact formulas) are difficult or impossible to find.

PREDIKTOR-KOREKTOR METHODS AND RUNGE KUTTA

The midpoint method has a predictor step:

$$S\left(t_j + \frac{h}{2}\right) = S(t_j) + \frac{h}{2} F(t_j, S(t_j)),$$

which is the prediction of the solution value
halfway between tj and tj+1.

It then computes the corrector step:

$$S(t_{j+1}) = S(t_j) + hF\left(t_j + \frac{h}{2}, S\left(t_j + \frac{h}{2}\right)\right)$$

which computes the solution at S(tj+1) from S(tj)
but using the derivative from S(tj+h2).

PREDiKTOR-KOREKTOR METHODS AND RUNGE KUTTA

Fourth-order Runge Kutta method

A classical method for integrating ODEs with a high order of accuracy is the Fourth Order Runge Kutta (RK4) method. It is obtained from the Taylor series using similar approach we just discussed in the second-order method. This method uses four points k1,k2,k3, and k4. A weighted average of these is used to produce the approximation of the solution. The formula is as follows.

$$
\begin{aligned}
k_1 &= F(t_j, S(t_j)) \\
k_2 &= F\left(t_j + \frac{h}{2}, S(t_j) + \frac{1}{2}k_1 h\right) \\
k_3 &= F\left(t_j + \frac{h}{2}, S(t_j) + \frac{1}{2}k_2 h\right) \\
k_4 &= F(t_j + h, S(t_j) + k_3 h)
\end{aligned}
$$

Therefore, we will have:

$$S(t_{j+1}) = S(t_j) + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)\,.$$

Click Here

PYTHON ODE SOLVERS

An ODE solver is an algorithm designed to compute numerical solutions to ordinary differential equations (ODEs).

CONSTRUCTION:

Let F be a function object to the function that computes

$$\frac{dS(t)}{dt} = F(t, S(t))$$

$$S(t0) = S0$$

S(t0)=S0

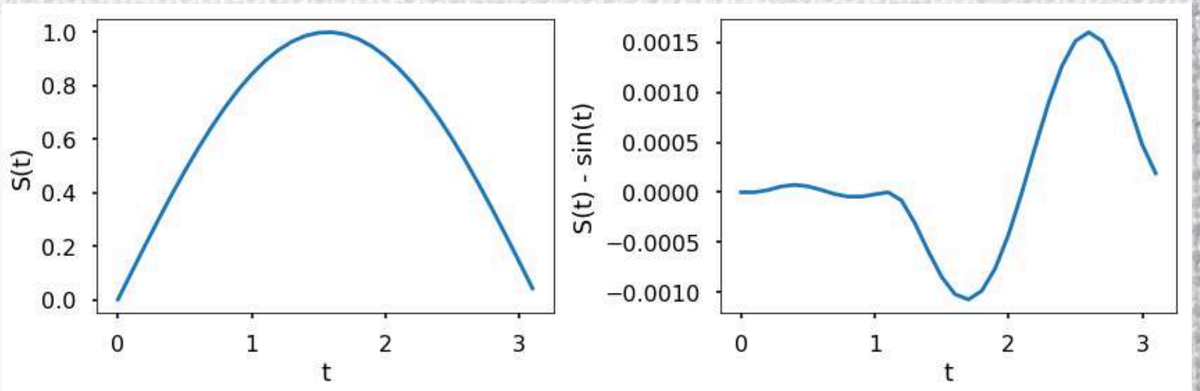
t is a one-dimensional independent variable (time), S(t) is an n-dimensional vector-valued function (state), and the F(t,S(t)) defines the differential equations. S0 be an initial value for S. The function F must have the form dS=F(t,S), although the name does not have to be F. The goal is to find the S(t) approximately satisfying the differential equations, given the initial value S(t0)=S0.

The way we use the solver to solve the differential equation is: solve_ivp(fun, t_span, s0, method = 'RK45', t_eval=None)

where fun takes in the function in the right-hand side of the system. t_span is the interval of integration (t0,tf), where t0 is the start and tf is the end of the interval. s0 is the initial state. There are a couple of methods that we can choose, the default is 'RK45', which is the explicit Runge-Kutta method of order 5(4). There are other methods you can use as well, see the end of this section for more information. t_eval takes in the times at which to store the computed solution, and must be sorted and lie within t_span.

PYTHON ODE SOLVERS

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp

plt.style.use('seaborn-poster')

%matplotlib inline

F = lambda t, s: np.cos(t)

t_eval = np.arange(0, np.pi, 0.1)
sol = solve_ivp(F, [0, np.pi], [0], t_eval=t_eval)

plt.figure(figsize = (12, 4))
plt.subplot(121)
plt.plot(sol.t, sol.y[0])
plt.xlabel('t')
plt.ylabel('S(t)')
plt.subplot(122)
plt.plot(sol.t, sol.y[0] - np.sin(sol.t))
plt.xlabel('t')
plt.ylabel('S(t) - sin(t)')
plt.tight_layout()
plt.show()
```



The above left figure shows the integration of dS(t)dt=cos(t) with solve_ivp. The right figure computes the difference between the solution of the integration by solve_ivp and the evalution of the analytical solution to this ODE. As can be seen from the figure, the difference between the approximate and exact solution to this ODE is small. Also, we can control the relative and absolute tolerances using the rtol and atol arguments, the solver keeps the local error estimates less than atol+rtol * abs(S). The default values are 1e-3 for rtol and 1e-6 for atol.

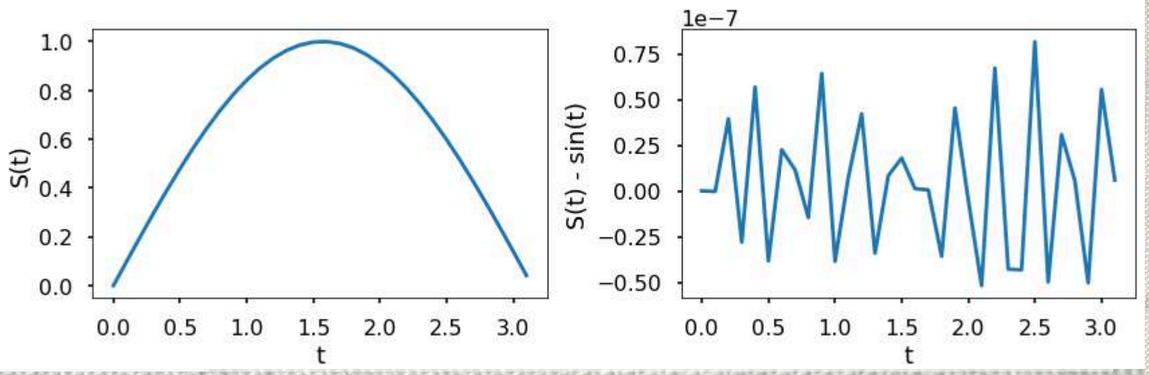
PYTHON ODE SOLVERS

```
sol = solve_ivp(F, [0, np.pi], [0], t_eval=t_eval, \
                rtol = 1e-8, atol = 1e-8)

plt.figure(figsize = (12, 4))
plt.subplot(121)
plt.plot(sol.t, sol.y[0])
plt.xlabel('t')
plt.ylabel('S(t)')
plt.subplot(122)
plt.plot(sol.t, sol.y[0] - np.sin(sol.t))
plt.xlabel('t')
plt.ylabel('S(t) - sin(t)')
plt.tight_layout()
plt.show()
```

The first plot shows the results of the numerical integration of the differential equation dS/dt=cos(t).{dS}/{dt} = \cos(t)dtdS =cos(t) using the solve_ivp method with very tight tolerances (rtol = 1e-8, atol = 1e-8). The resulting curve is nearly identical to the analytical solution S(t)=sin(t)S(t) = \sin(t)S(t)=sin(t). This is evident from the shape of the graph, which rises to a peak at t=π2t ={\pi}{2}t=2π, then decreases again to near zero at t=πt = \pit=π.

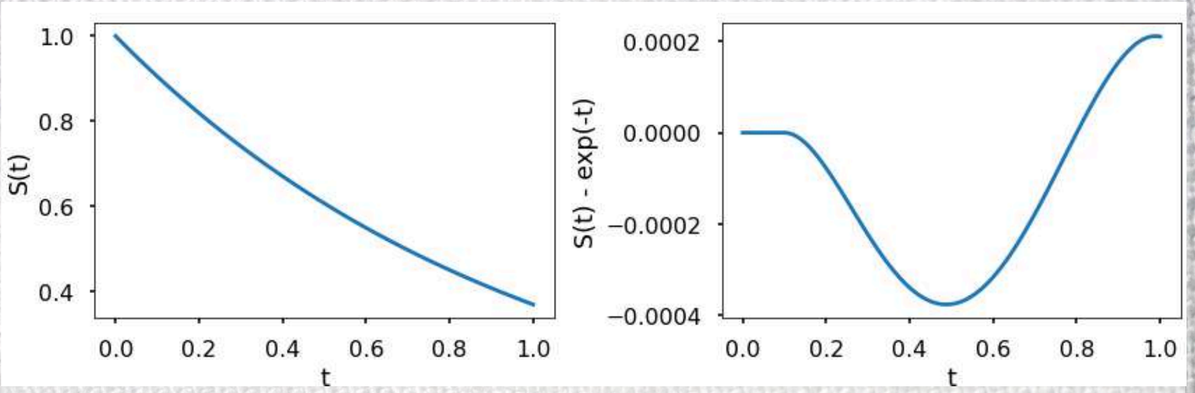
The second plot shows the difference between the numerical and analytical solutions. The error appears to oscillate very slightly around zero, with a magnitude of only about 10−710^{-7}10−7. This indicates that the numerical error is very small.

Overall, these results confirm that the numerical method used is highly accurate and capable of producing solutions that are nearly identical to the exact solution.



PYTHON ODE SOLVERS

```
F = lambda t, s: -s

t_eval = np.arange(0, 1.01, 0.01)
sol = solve_ivp(F, [0, 1], [1], t_eval=t_eval)

plt.figure(figsize = (12, 4))
plt.subplot(121)
plt.plot(sol.t, sol.y[0])
plt.xlabel('t')
plt.ylabel('S(t)')
plt.subplot(122)
plt.plot(sol.t, sol.y[0] - np.exp(-sol.t))
plt.xlabel('t')
plt.ylabel('S(t) - exp(-t)')
plt.tight_layout()
plt.show()
```

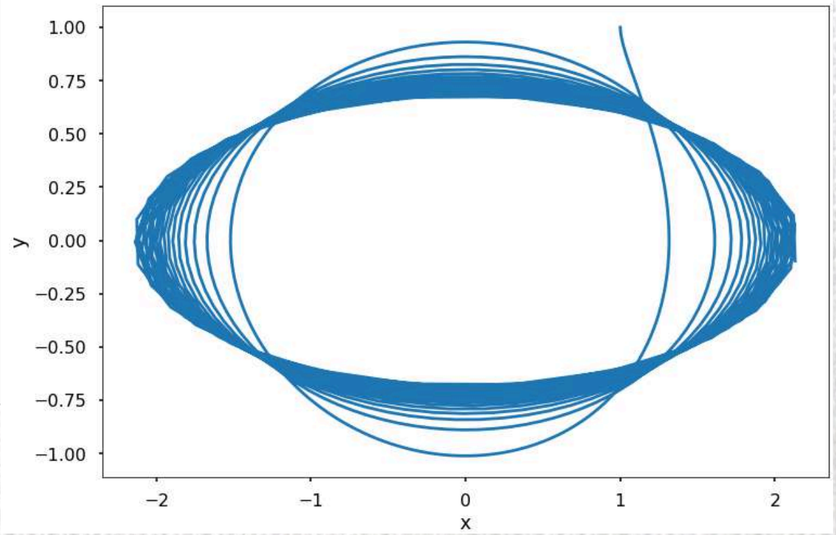


The first plot shows the numerical integration results of the differential equation dSdt=−S\frac{dS}{dt} = -SdtdS=−S with the initial conditions S(0)=1S(0) = 1S(0)=1. Analytically, this equation has a solution S(t)=e−tS(t) = e^{-t}S(t)=e−t, which illustrates an exponential decrease over time. The numerical results from solve_ivp appear very close to the exact solution, as evidenced by the smooth decrease from 1 to approximately 0.37 at t=1t = 1t=1.

The second plot shows the difference between the numerical and analytical solutions. The error is very small—approximately ±0.0004—and oscillates slightly around zero. This indicates that the numerical method used has high accuracy and is able to approximate the analytical solution very well.

PYTHON ODE SOLVERS

```
F = lambda t, s: np.dot(np.array([[0, t**2], [-t, 0]]), s)

t_eval = np.arange(0, 10.01, 0.01)
sol = solve_ivp(F, [0, 10], [1, 1], t_eval=t_eval)

plt.figure(figsize = (12, 8))
plt.plot(sol.y.T[:, 0], sol.y.T[:, 1])
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



The graph above illustrates the trajectory of the solution to a time-dependent differential equation system. The function F(t,s)F(t, s)F(t,s) describes how the system's state evolves, governed by a matrix whose elements change over time according to ttt and t2t^2t2. Using the numerical method solve_ivp, the solution is computed from time 0 to 10 with a small step size of 0.01, producing a smooth curve. The resulting plot shows a trajectory that is unstable and continuously shifting, forming an oval-like shape that expands and contracts as time progresses. This behavior indicates that the system has evolving dynamics where even small changes in time influence the direction and geometry of the solution's motion.

KESIMPULAN

1. ODEs are used to model single-variable physical phenomena, especially those that depend on time.
2. IVPs determine the solution of an ODE based on initial conditions, making the solution well-defined.
3. PDEs involve multiple variables, whereas ODEs involve only one, making ODEs simpler.
4. Reduction of Order converts higher-order ODEs into first-order systems to simplify numerical solving.
5. The small-angle approximation in the pendulum problem simplifies the nonlinear model into a linear one.
6. The Euler method is simple but less accurate and can be unstable for large step sizes.
7. Methods such as RK4 provide higher accuracy and stability compared to Euler.
8. solve_ivp makes numerical ODE solving easier and more efficient in Python.

TERIMAKASIH

Q & A?