PROGRAM STUDI INFORMATIKA
FAKULTAS TEKNIK DAN INFORMATIKA
UNIVERSITAS MULTIMEDIA NUSANTARA
SEMESTER GENAP TAHUN AJARAN 2024/2025

IF420 – ANALISIS NUMERIK

Pertemuan ke 12 – Ordinary Differential Equation - Initial Value Problems

Dr. Ivransa Zuhdi Pane, M.Eng., B.CS.
Marlinda Vasty Overbeek, S.Kom., M.Kom.
Seng Hansun, S.Si., M.Cs.

Capaian Pembelajaran Mingguan Mata Kuliah (Sub-CPMK):

Sub-CPMK 12: Mahasiswa mampu memahami dan menerapkan Ordinary Differential Equations: Permasalahan nilai awal – C3

Reviews

- Numerical Integration Problem Statement
- Riemann’s Integral
- Trapezoid Rule
- Simpson’s Rule
- Computing Integrals in Python

Outlines

- ODE Initial Value Problem Statement
- Reduction of Order
- The Euler Method
- Numerical Error and Instability
- Predictor-Corrector Methods
- Python ODE Solvers
- Advanced Topics

Motivation

- **Differential equations** are **relationships** between a **function** and its **derivatives**, and they are used to model systems in every engineering and science field.
- For example, a simple differential equation relates the **acceleration** of a car with its **position**.
- Unlike **differentiation** where **analytic solutions** can usually be **computed**, in general finding exact solutions to **differential equations** is very **hard**.
- Therefore, **numerical solutions** are critical to making these equations useful for designing and understanding engineering and science systems.
- This lesson covers **ordinary differential equations** with **specified initial values**, a **subclass** of **differential equations** problems called **initial value problems**.
- To reflect the importance of this class of problem, **Python** has a whole suite of **functions** to solve this kind of problem.

ODE Initial Value Problem Statement

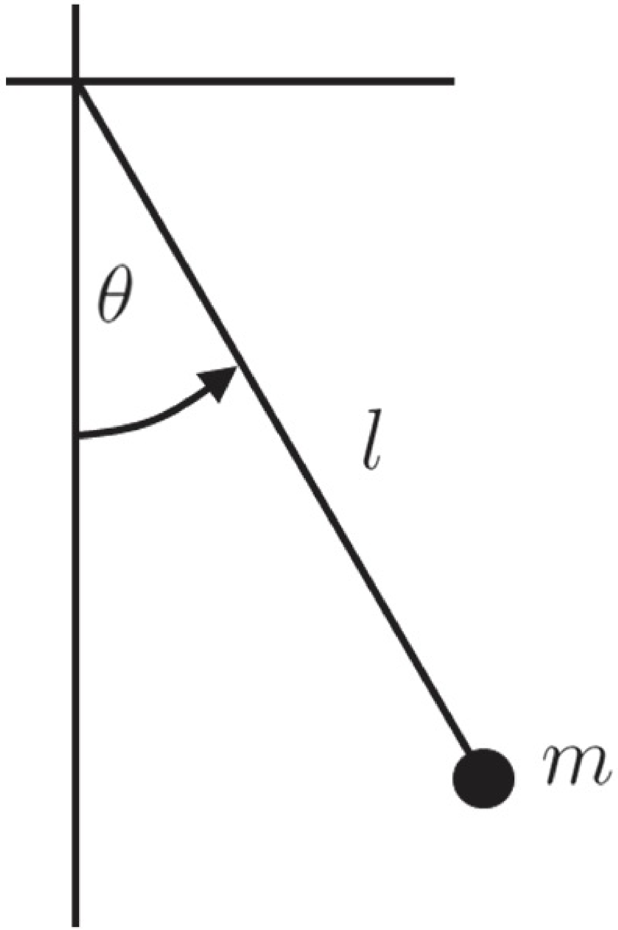
- A **differential equation** is a **relationship** between a **function**, $f(x)$, its independent variable, x, and any number of its **derivatives**.
- An **ordinary differential equation** or **ODE** is a **differential equation** where the **independent variable**, and therefore also the **derivatives**, is in **one dimension**.
- Here, we assume that an ODE can be written

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

where F is an **arbitrary function** that incorporates one or all of the input arguments, and n is the **order** of the differential equation.

- This equation is referred to as an **n-th order ODE**.

ODE Initial Value Problem Statement



- To give an **example** of an ODE, consider a **pendulum** of length l with a mass, m, at its end (see the left figure).
- The **angle** the pendulum makes with the **vertical axis** over time, $\Theta(t)$, in the presence of vertical gravity, g, can be described by the pendulum equation, which is the ODE

$$ml\frac{d^2\Theta(t)}{dt^2} = -mg\sin(\Theta(t))$$

- This equation can be derived by **summing** the **forces** in the x and y direction, and then changing them to **polar** **coordinates**.

ODE Initial Value Problem Statement

- In contrast, a **partial differential equation** or **PDE** is a **general form of differential equation** where x is a vector containing the independent variables $x_1, x_2, x_3, \ldots, x_m$, and the **partial derivatives** can be of **any order** and with respect to **any** combination of **variables**.

- An example of a PDE is the **heat equation**, which describes the evolution of temperature in space over time:

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

- Here, $u(t,x,y,z)$ is the temperature at (x,y,z) at time t, and α is a thermal diffusion constant.

ODE Initial Value Problem Statement

- A **general solution** to a differential equation is a $g(x)$ that **satisfies** the **differential equation**.
- Although there are usually **many solutions** to a differential equation, they are still **hard to find**.
- For an **ODE** of order n, a **particular solution** is a $p(x)$ that **satisfies** the **differential equation** and n explicitly **known values** of the solution, or its **derivatives**, at **certain points**.
- Generally stated, $p(x)$ must satisfy the differential equation and $p^{(j)}(x_i) = p_i$, where $p^{(j)}$ is the j-th derivative of p, for n triplets, (j, x_i, p_i).
- For the purpose of this lesson, we refer to the particular solution simply as the **solution**.

ODE Initial Value Problem Statement

- **Example**: Returning to the pendulum example, if we assume the angles are very small (i.e., $\sin(\Theta(t)) \approx \Theta(t)$), then the pendulum equation reduces to

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

- Verify that $\Theta(t) = \cos\left(\sqrt{\frac{g}{l}}t\right)$ is a **general solution** to the pendulum equation.
- If the angle and angular velocities at $t = 0$ are the known values, Θ_0 and 0, respectively, verify that $\Theta(t) = \Theta_0 \cos\left(\sqrt{\frac{g}{l}}t\right)$ is a **particular solution** for these known values.

- For the **general solution**, the derivatives of $\Theta(t)$ are

$$\frac{d\Theta(t)}{dt} = -\sqrt{\frac{g}{l}}\sin\left(\sqrt{\frac{g}{l}}t\right)$$

- and

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

- By plugging the **second derivative** back into the differential equation on the left side, it is easy to verify that $\Theta(t)$ satisfies the equation and so is a **general solution**.
- For the **particular solution**, the Θ_0 coefficient will carry through the derivatives, and it can be verified that the equation is satisfied.
- $\Theta(0) = \Theta_0\cos(0) = \Theta_0$, and $0 = -\Theta_0\sqrt{\frac{g}{l}}\sin(0) = 0$, therefore the **particular solution** also satisfies the known values.

ODE Initial Value Problem Statement

- A pendulum swinging at **small angles** is a very uninteresting pendulum indeed.
- Unfortunately, there is **no explicit solution** for the pendulum equation with **large angles** that is as simple algebraically.
- Since this system is much simpler than most practical engineering systems and has no obvious analytic solution, the **need** for **numerical solutions** to ODEs is clear.
- A common **set of known values** for an **ODE solution** is the **initial value**.
- For an ODE of order n, the **initial value** is a known value for the 0-th to $(n-1)$-th **derivatives** at $x = 0, f(0), f^{(1)}(0), f^{(2)}(0), \dots, f^{(n-1)}(0)$.
- For a certain class of ordinary differential equations, the **initial value** is **sufficient** to find a **unique particular solution**.
- Finding a **solution** to an ODE given an **initial value** is called the **initial value problem.**

ODE Initial Value Problem Statement

- Although the name suggests we will only cover ODEs that evolve in time, **initial value problems** can also include systems that evolve in **other dimensions,** such as **space.**
- Intuitively, the pendulum equation can be solved as an initial value problem because under only the **force of gravity**, an **initial position** and **velocity** should be sufficient to describe the motion of the pendulum for all time afterward.
- The remainder of this lesson covers several **methods** of **numerically approximating** the **solution** to **initial value problems** on a **numerical grid**.
- Although **initial value problems** encompass more than just **differential equations** in time, we use **time** as the **independent variable**.
- We also use **several notations** for the **derivative** of $f(t)$: $f'(t), f^{(1)}(t), \frac{df(t)}{dt}$, and $\dot{f}$, whichever is most convenient for the context.

Reduction of Order

- Many **numerical methods** for solving initial value problems are designed specifically to solve **first-order differential equations**.
- To make these solvers useful for solving **higher order differential equations**, we must often **reduce** the **order** of the **differential equation** to **first order**.
- To reduce the order of a differential equation, consider a **vector**, $S(t)$, which is the **state** of the **system** as a **function** of **time**.
- In general, the state of a system is a collection of all the **dependent variables** that are relevant to the **behavior** of the **system**.
- Recalling that the ODEs of interest can be expressed as

$$f^{(n)}(t) = F\left(t, f(t), f^{(1)}(t), f^{(2)}(t), \ldots, f^{(n-1)}(t)\right)$$

- For **initial value** problems, it is useful to take the **state** to be

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

- Then the **derivative** of the state is

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

where $S_i(t)$ is the i-th element of $S(t)$.

Reduction of Order

- With the **state** written in this way, $\frac{dS(t)}{dt}$ can be written using only $S(t)$ (i.e., no $f(t)$) or its derivatives.
- In particular, $\frac{dS(t)}{dt} = \mathcal{F}(t, S(t))$, where $\mathcal{F}$ is a **function** that appropriately assembles the **vector** describing the **derivative** of the **state**.
- This equation is in the form of a **first-order differential equation** in S.
- Essentially, what we have done is turn an **n-th order** ODE into n **first order** ODEs that are coupled together, meaning they share the **same terms**.
- **Example**: Reduce the **second order** pendulum equation to **first order**, where

$$S(t) = \begin{bmatrix} \Theta(t) \\ \dot{\Theta}(t) \end{bmatrix}$$

- Taking the **derivative** of $S(t)$ and **substituting** gives the correct expression.

$$\frac{dS(t)}{dt} = \begin{bmatrix} S_2(t) \\ -\frac{g}{l} S_1(t) \end{bmatrix}$$

- It happens that this ODE can be written in **matrix** form:

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

- ODEs that can be written in this way are said to be **linear** ODEs.
- Although reducing the order of an ODE to **first order** results in an ODE with **multiple variables**, all the derivatives are still taken with respect to the same **independent** variable, t. Therefore, the **ordinariness** of the differential equation is **retained**.
- It is worth noting that the state can hold **multiple dependent** variables and their **derivatives** as long as the derivatives are with respect to the **same independent variable**.

- **Example**: A very simple model to describe the change in population of rabbits, $r(t)$, and wolves, $w(t)$, might be

$$\frac{dr(t)}{dt} = 4r(t) - 2w(t)$$

and

$$\frac{dw(t)}{dt} = r(t) + w(t)$$

- The first ODE says that at each time step, the rabbit population multiplies by 4, but each wolf eats two of the rabbits. The second ODE says that at each time step, the population of wolves increases by the number of rabbits and wolves in the system.
- Write this **system of differential equations** as an equivalent **differential equation** in $S(t)$ where

$$S(t) = \begin{bmatrix} r(t) \\ w(t) \end{bmatrix}.$$

- The following **first-order** ODE is equivalent to the pair of ODEs.

$$\frac{dS(t)}{dt} = \begin{bmatrix} 4 & -2 \\ 1 & 1 \end{bmatrix} S(t).$$

The Euler Method

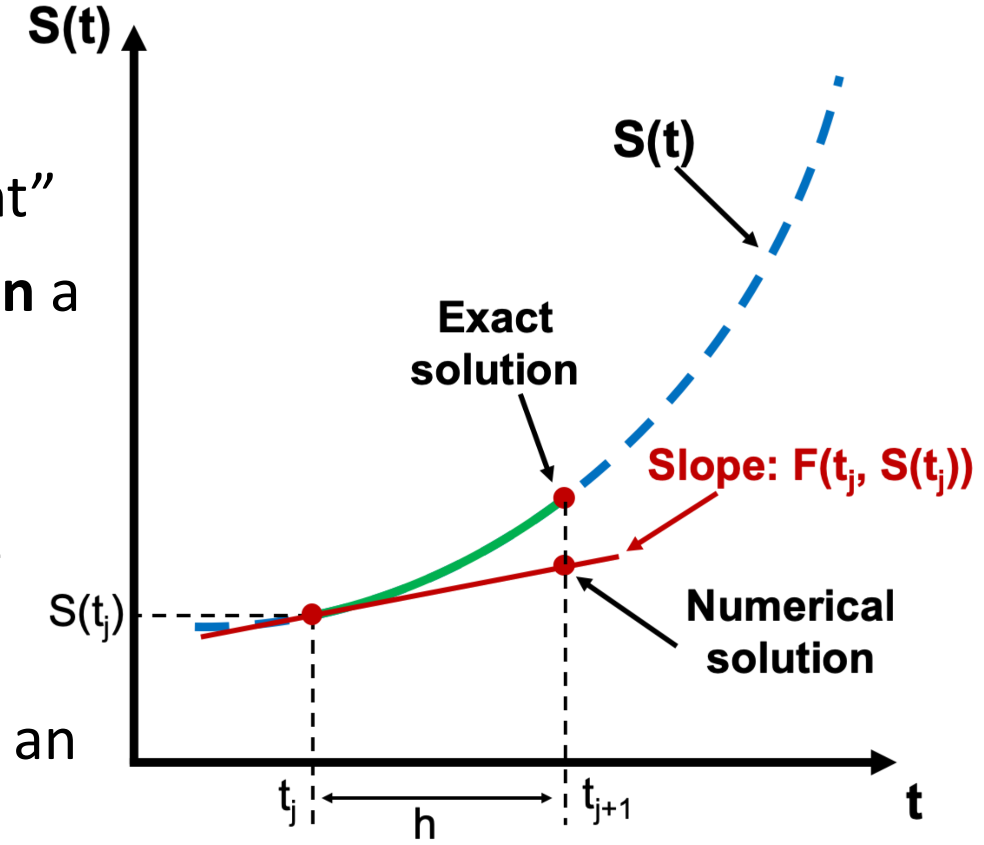
- Let $\frac{dS(t)}{dt} = F(t, S(t))$ be an explicitly defined **first order** ODE. That is, F is a function that returns the **derivative**, or **change**, of a state given a time and state value.
- Also, let t be a **numerical grid** of the **interval** $[t_0, t_f]$ with spacing h. **Without loss of generality**, we assume that $t_0 = 0$, and that $t_f = Nh$ for some positive integer, N.
- The **linear approximation** of $S(t)$ around t_j at t_{j+1} is

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

which can also be written

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

- This formula is called the **Explicit Euler Formula**, and it allows us to compute an **approximation** for the **state** at $S(t_{j+1})$ given the state at $S(t_j)$.
- Starting from a given **initial value** of $S_0 = S(t_0)$, we can use this formula to integrate the states up to $S(t_f)$; these $S(t)$ values are then an **approximation** for the **solution** of the differential equation.
- The **Explicit Euler** formula is the **simplest** and **most intuitive** method for solving **initial value problems**.
- At any state $\left(t_j, S(t_j)\right)$, it uses F at that state to "point" toward the next state and then moves in that **direction** a distance of h.
- Although there are more **sophisticated** and **accurate** methods for solving these problems, they all have the same **fundamental structure**.
- As such, we **enumerate** explicitly the steps for solving an initial value problem using the **Explicit Euler** formula.

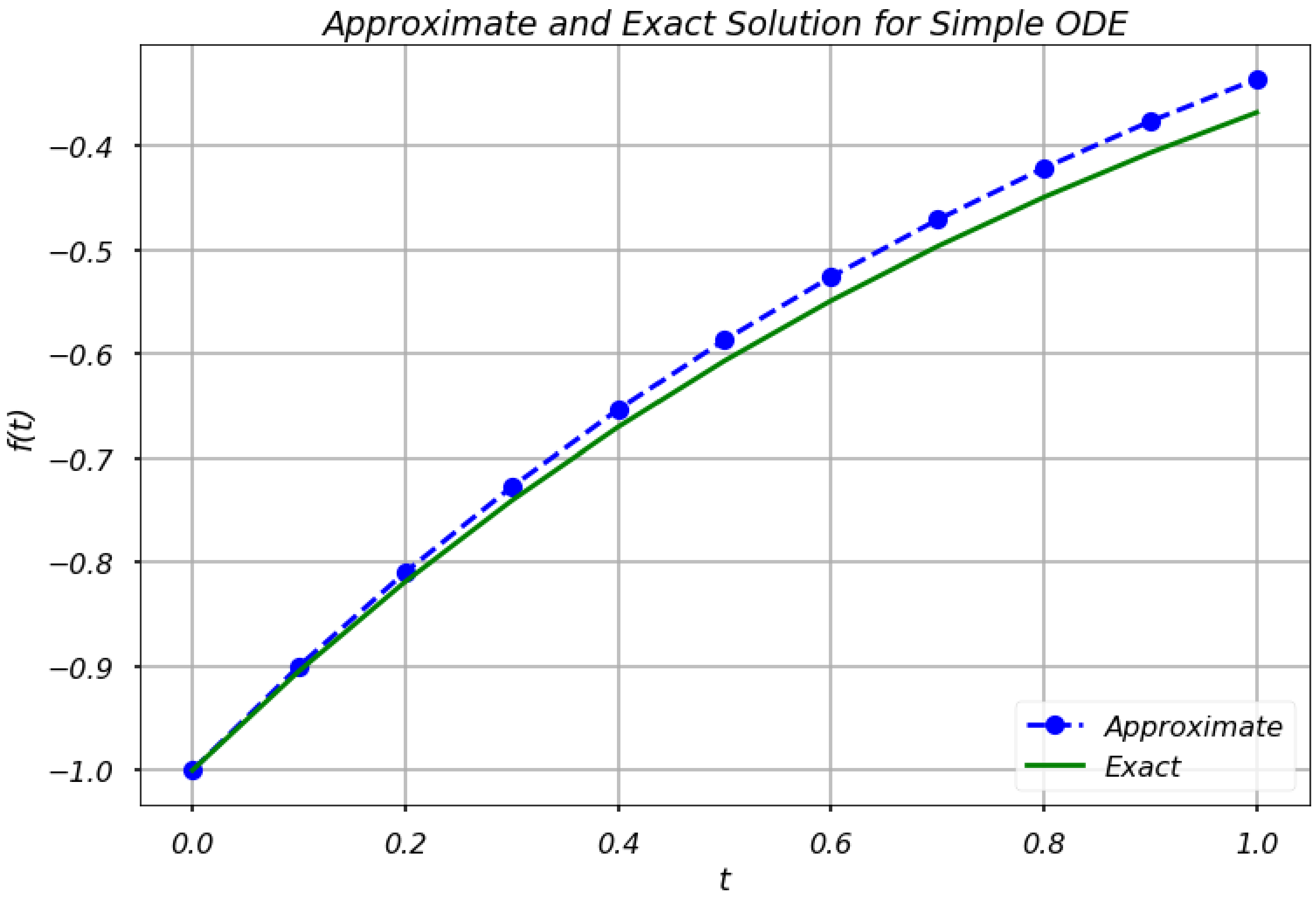


- **WHAT IS HAPPENING?** Assume we are given a **function** $F(t, S(t))$ that computes $\frac{dS(t)}{dt}$, a **numerical grid**, t, of the interval, $[t_0, t_f]$, and an **initial state** value $S_0 = S(t_0)$. We can compute $S(t_j)$ for every t_j in t using the following steps.
  1. Store $S_0 = S(t_0)$ in an array, S.
  2. Compute $S(t_1) = S_0 + hF(t_0, S_0)$.
  3. Store $S_1 = S(t_1)$ in S.
  4. Compute $S(t_2) = S_1 + hF(t_1, S_1)$.
  5. Store $S_2 = S(t_2)$ in S.
  6. $\cdots$
  7. Compute $S(t_f) = S_{f-1} + hF(t_{f-1}, S_{f-1})$.
  8. Store $S_f = S(t_f)$ in S.
  9. S is an approximation of the solution to the initial value problem.
- When using a method with this structure, we say the method **integrates** the **solution** of the ODE.

- **Example**: The differential equation $\frac{df(t)}{dt} = e^{-t}$ with initial condition $f_0 = -1$ has the exact solution $f(t) = -e^{-t}$. Approximate the solution to this **initial value problem** between 0 and 1 in increments of 0.1 using the **Explicit Euler Formula**. Plot the difference between the approximated solution and the exact solution.

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

plt.style.use('seaborn-poster')
%matplotlib inline

# Define parameters
f = lambda t, s: np.exp(-t) # ODE
h = 0.1 # Step size
t = np.arange(0, 1 + h, h) # Numerical grid
s0 = -1 # Initial Condition
```

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

for i in range(0, 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()
```

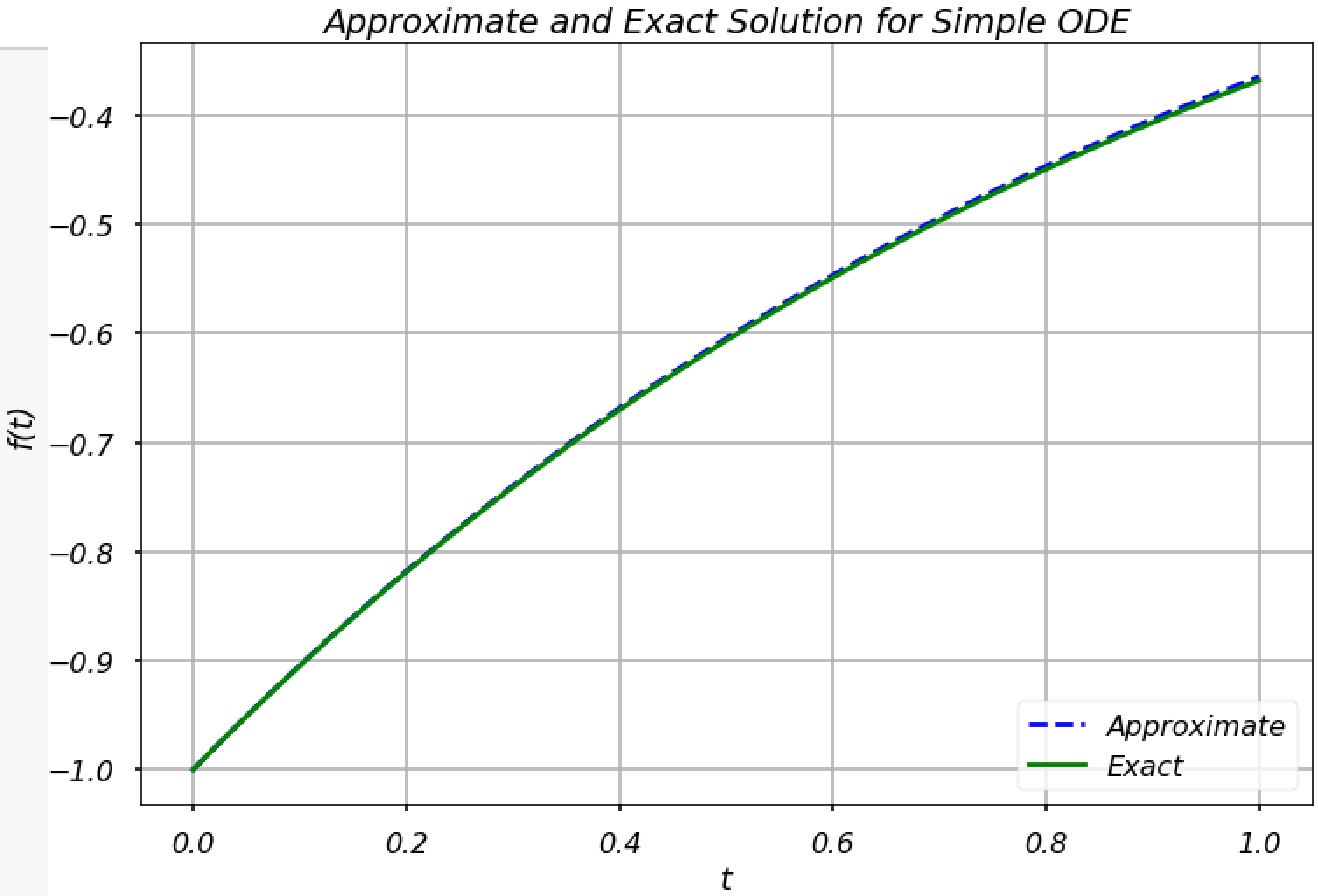


- In the above figure, we can see **each dot** is **one approximation** based on the **previous dot** in a linear fashion.
- From the **initial value**, we can eventually get an **approximation** of the **solution** on the **numerical grid**.

The Euler Method

- If we repeat the process for $h = 0.01$, we get a **better approximation** for the solution:

```
h = 0.01 # Step size
t = np.arange(0, 1 + h, h) # Numerical grid
s0 = -1 # Initial Condition

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

for i in range(0, len(t) - 1):
    s[i + 1] = s[i] + h*f(t[i], s[i])

plt.figure(figsize = (12, 8))
plt.plot(t, s, 'b--', 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 **Explicit Euler Formula** is called "**explicit**" because it only requires information at t_j to compute the **state** at t_{j+1}. That is, $S(t_{j+1})$ can be **written explicitly** in terms of values we have (i.e., t_j and $S(t_j)$).
- The **Implicit Euler Formula** can be derived by taking the **linear approximation** of $S(t)$ around t_{j+1} and computing it at t_j:

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

- This formula is **peculiar** because it requires that we know $S(t_{j+1})$ to compute $S(t_{j+1})$!
- However, it happens that **sometimes** we can use **this formula** to **approximate** the **solution** to initial value problems.
- Before we give details on how to solve these problems using the **Implicit Euler Formula**, we give **another implicit formula** called the **Trapezoidal Formula**, which is the **average** of the **Explicit** and **Implicit Euler** Formulas:

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

- To illustrate how to **solve** these **implicit schemes**, consider again the pendulum equation, which has been reduced to **first order**.

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

- For this equation,

$$F\left(t_j, S(t_j)\right) = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l} & 0 \end{bmatrix} S\left(t_j\right).$$

- If we plug this expression into the **Explicit Euler Formula**, we get the following equation:

$$S\left(t_{j+1}\right) = S\left(t_j\right) + h \begin{bmatrix} 0 & 1 \\ -\frac{g}{l} & 0 \end{bmatrix} S\left(t_j\right) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} S\left(t_j\right) + h \begin{bmatrix} 0 & 1 \\ -\frac{g}{l} & 0 \end{bmatrix} S\left(t_j\right)$$

$$= \begin{bmatrix} 1 & h \\ -\frac{gh}{l} & 1 \end{bmatrix} S\left(t_j\right)$$

- Similarly, we can plug the same expression into the **Implicit Euler** to get

$$\begin{bmatrix} 1 & -h \\ \frac{gh}{l} & 1 \end{bmatrix} S(t_{j+1}) = S(t_j)$$

and into the **Trapezoidal Formula** to get

$$\begin{bmatrix} 1 & -\frac{h}{2} \\ \frac{gh}{2l} & 1 \end{bmatrix} S(t_{j+1}) = \begin{bmatrix} 1 & \frac{h}{2} \\ -\frac{gh}{2l} & 1 \end{bmatrix} S(t_j).$$

- With some rearrangement, these equations become, respectively,

$$S(t_{j+1}) = \begin{bmatrix} 1 & -h \\ \frac{gh}{l} & 1 \end{bmatrix}^{-1} S(t_j),$$

$$S(t_{j+1}) = \begin{bmatrix} 1 & -\frac{h}{2} \\ \frac{gh}{2l} & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & \frac{h}{2} \\ -\frac{gh}{2l} & 1 \end{bmatrix} S(t_j).$$

- These equations allow us to **solve** the **initial value problem**, since at each state, $S(t_j)$, we can compute the next state at $S(t_{j+1})$. In general, this is possible to do when an ODE is **linear**.

Numerical Error and Instability

- There are **two main issues** to consider with regard to integration schemes for ODEs: **accuracy** and **stability**.
- **Accuracy** refers to a scheme's ability to get **close** to the **exact solution**, which is usually **unknown**, as a function of the **step size** h. Previous lessons have referred to accuracy using the notation $O(h^p)$. The same notation translates to solving ODEs.
- The **stability** of an integration scheme is its ability to **keep the error** from **growing** as it integrates forward in **time**. If the error **does not grow**, then the scheme is **stable**; otherwise, it is **unstable**. Some integration schemes are **stable** for certain choices of h and **unstable** for others; these integration schemes are also referred to as **unstable**.
- To illustrate issues of **stability**, we numerically solve the pendulum equation using the **Euler Explicit**, **Euler Implicit**, and **Trapezoidal** Formulas.

Numerical Error and Instability

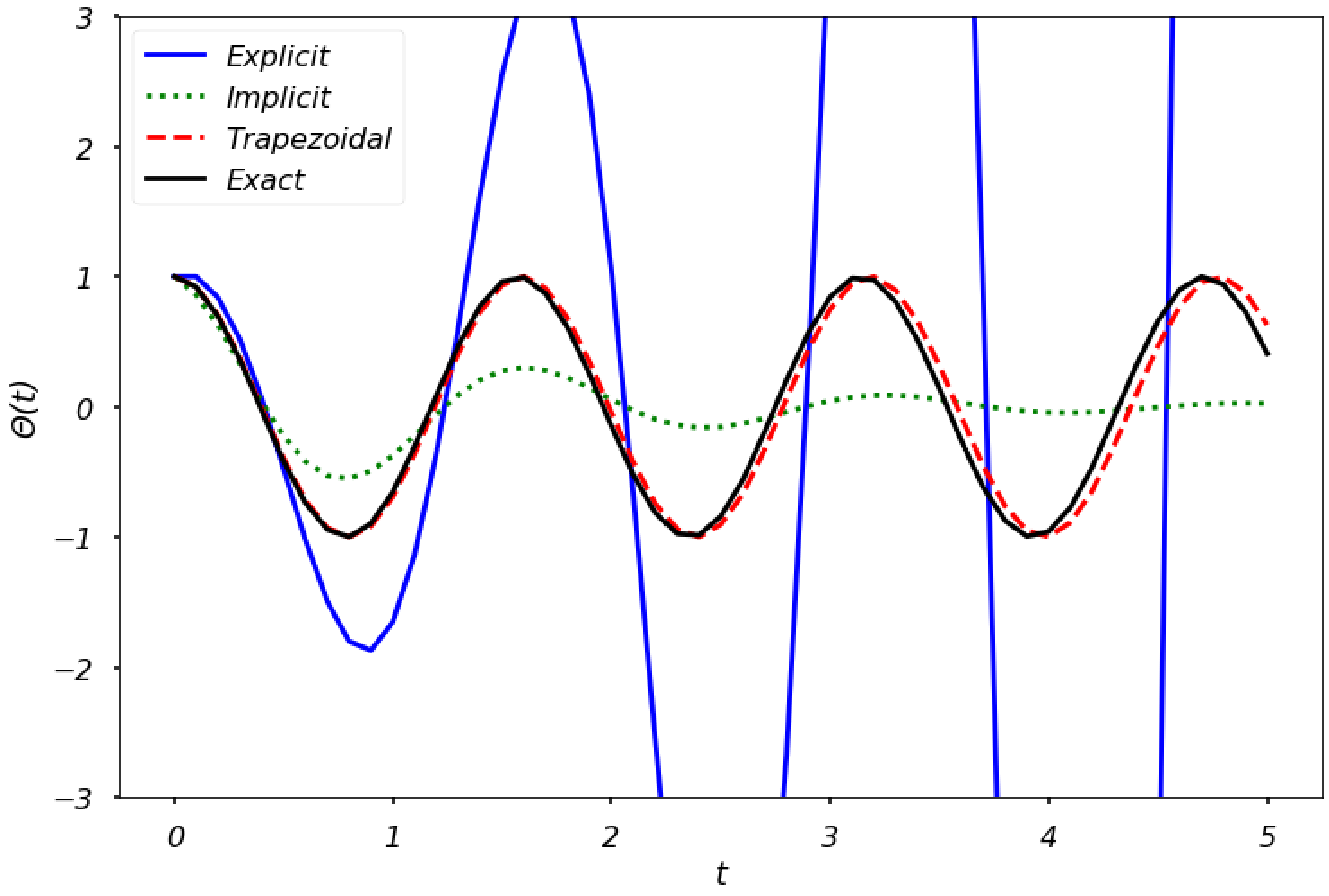
- **Example**: Use the **Euler Explicit**, **Euler Implicit**, and **Trapezoidal Formulas** to solve the pendulum equation over the time interval [0,5] in increments of 0.1 and for an initial solution of $S_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. For the model parameters use $\sqrt{\frac{g}{l}} = 4$. Plot the approximate solution on a single graph.

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

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

%matplotlib inline
```

```python
# define step size
h = 0.1
# define numerical grid
t = np.arange(0, 5.1, h)
# oscillation freq. of pendulum
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 = np.dot(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))
```

```python
# do integrations
s_e[0, :] = s0.T
s_i[0, :] = s0.T
s_t[0, :] = s0.T

for j in range(0, len(t)-1):
    s_e[j+1, :] = np.dot(m_e,s_e[j, :])
    s_i[j+1, :] = np.dot(m_i,s_i[j, :])
    s_t[j+1, :] = np.dot(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('$\Theta (t)$')
plt.legend(['Explicit', 'Implicit', \
            'Trapezoidal', 'Exact'])
plt.show()
```

Numerical Error and Instability



- The generated figure shows the comparisons of numerical solution to the pendulum problem.
- The **exact** solution is a pure **cosine wave**. The **Explicit Euler** scheme is clearly **unstable**. The **Implicit Euler** scheme **decays exponentially**, which is not correct. The **Trapezoidal** method captures the solution correctly, with a **small phase shift** as time increases.

Predictor-Corrector Methods

- Given any **time** and **state** value, the function, $F(t, S(t))$, returns the **change of state** $\frac{dS(t)}{dt}$.
- Predictor-corrector methods of solving **initial value problems** improve the approximation accuracy of **non-predictor-corrector** methods by **querying** the F function **several times** at **different locations** (**predictions**), and then using a **weighted average** of the results (**corrections**) to **update** the state.
- Essentially, it uses **two formulas**: the **predictor** and **corrector**.
- The **predictor** is an **explicit formula** and first **estimates** the **solution** at t_{j+1}, i.e. we can use **Euler method** or some other methods to finish this step. After we obtain the solution $S(t_{j+1})$, we can apply the **corrector** to **improve** the **accuracy**. Using the found $S(t_{j+1})$ on the right-hand side of an otherwise implicit formula, the corrector can calculate a **new, more accurate** solution.

Predictor-Corrector Methods

- The **midpoint method** has a **predictor** step:

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

which is the **prediction** of the solution value **halfway** between t_j and t_{j+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 $S(t_{j+1})$ from $S(t_j)$ but using the **derivative** from $S\left(t_j + \frac{h}{2}\right)$.

Runge Kutta Methods

- **Runge Kutta (RK)** methods are one of the most widely used methods for solving ODEs.
- Recall that the **Euler method** uses the **first two terms** in **Taylor series** to approximate the numerical integration, which is **linear**: $S(t_{j+1}) = S(t_j + h) = S(t_j) + h \cdot S'(t_j)$.
- We can greatly improve the **accuracy** of numerical integration if we keep **more terms** of the series in

$$S(t_{j+1}) = S(t_j + h) = S(t_j) + S'(t_j)h + \frac{1}{2!}S''(t_j)h^2 + \cdots + \frac{1}{n!}S^{(n)}(t_j)h^n \quad \textbf{...(1)}$$

- In order to get this more **accurate** solution, we need to **derive** the **expressions** of $S''(t_j), S'''(t_j), \ldots, S^{(n)}(t_j)$.
- This extra work can be **avoided** using the **RK methods**, which is based on **truncated Taylor** series, but not require computation of these higher derivatives.

Second Order Runge Kutta Method

- Let us first derive the **second order** RK method.
- Let $\frac{dS(t)}{dt} = F(t, S(t))$, then we can assume an **integration formula** in the form of

$$S(t+h) = S(t) + c_1 F(t, S(t))h + c_2 F\left[t + ph, S(t) + qhF(t, S(t))\right]h \quad \textbf{...(2)}$$

- We can attempt to find these parameters c_1, c_2, p, q by matching the above equation to the **second-order Taylor** series, which gives us

$$S(t+h) = S(t) + S'(t)h + \frac{1}{2!}S''(t)h^2 = S(t) + F(t, S(t))h + \frac{1}{2!}F'(t, S(t))h^2 \quad \textbf{...(3)}$$

- Noting that $F'(t, S(t)) = \frac{\partial F}{\partial t} + \frac{\partial F}{\partial S}\frac{\partial S}{\partial t} = \frac{\partial F}{\partial t} + \frac{\partial F}{\partial S}F$.

- Therefore, equation (3) can be written as:

$$S(t+h) = S + Fh + \frac{1}{2!}\left(\frac{\partial F}{\partial t} + \frac{\partial F}{\partial S}F\right)h^2 \quad \textbf{...(4)}$$

- In equation (2), we can rewrite the **last term** by applying **Taylor series** in several variables, which gives us:

$$F[t+ph, S+qhF] = F + \frac{\partial F}{\partial t}ph + qh\frac{\partial F}{\partial S}F$$

thus equation (2) becomes:

$$S(t+h) = S + (c_1 + c_2)Fh + c_2\left[\frac{\partial F}{\partial t}p + q\frac{\partial F}{\partial S}F\right]h^2 \quad \textbf{...(5)}$$

- Comparing equation (4) and (5), we can easily obtain:

$$c_1 + c_2 = 1, c_2 p = \frac{1}{2}, c_2 q = \frac{1}{2} \quad \textbf{...(6)}$$

Second Order Runge Kutta Method

- Because (6) has **four unknowns** and only **three equations**, we can assign any value to one of the parameters and get the rest of the parameters. One popular choice is:

$$c_1 = \frac{1}{2}, c_2 = \frac{1}{2}, p = 1, q = 1$$

- We can also define:

$$k_1 = F\left(t_j, S(t_j)\right)$$

$$k_2 = F\left(t_j + ph, S(t_j) + qhk_1\right)$$

where we will have:

$$S(t_{j+1}) = S(t_j) + \frac{1}{2}(k_1 + k_2)h$$

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 for the **second-order** method.
- This method uses **four points** k_1, k_2, k_3, and k_4.
- A **weighted average** of these is used to produce the **approximation** of the **solution**.

- The formula is as follows.

$$k_1 = F\left(t_j, S(t_j)\right)$$

$$k_2 = F\left(t_j + \frac{h}{2}, S(t_j) + \frac{1}{2}k_1 h\right)$$

$$k_3 = F\left(t_j + \frac{h}{2}, S(t_j) + \frac{1}{2}k_2 h\right)$$

$$k_4 = F\left(t_j + h, S(t_j) + k_3 h\right)$$

- Therefore, we will have:

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

- As indicated by its name, the RK4 method is **fourth-order accurate**, or $O(h^4)$.

Python ODE Solvers

- In **scipy**, there are several **built-in functions** for solving **initial value problems**. The most common one used is the **scipy.integrate.solve_ivp** function. The function construction are shown below:

- Let F be a **function object** to the function that computes

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

$$S(t_0) = S_0$$

- t is a one-dimensional independent variable (**time**), $S(t)$ is an n-dimensional vector-valued function (**state**), and the $F\big(t, S(t)\big)$ defines the **differential** equations. S_0 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** $S\big(t_j\big) = S_0$.

Python ODE Solvers

- 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 (t_0, t_f), where t_0 is the start and t_f is the end of the interval. S_0 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

- **EXAMPLE**: Consider the ODE

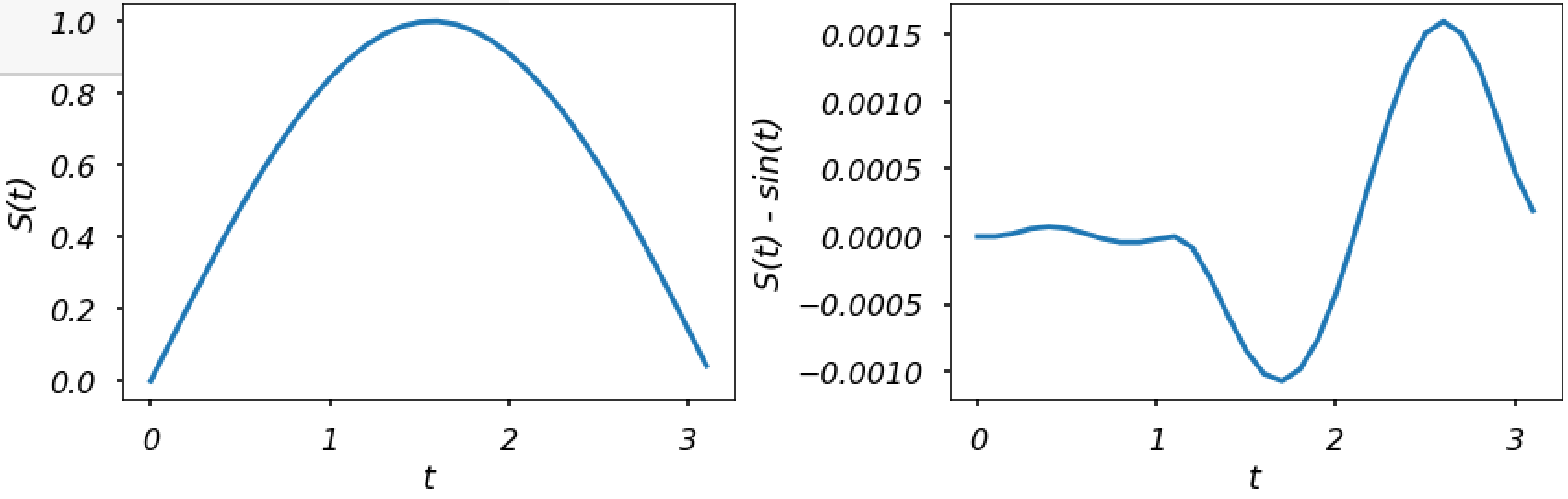
$$\frac{dS(t)}{dt} = \cos(t)$$

for an initial value $S_0 = 0$. The exact solution to this problem is $S(t) = \sin(t)$. Use **solve_ivp** to approximate the solution to this **initial value problem** over the interval $[0, \pi]$. Plot the approximate solution versus the exact solution and the relative error over time.

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

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

%matplotlib inline
```

```python
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()
```

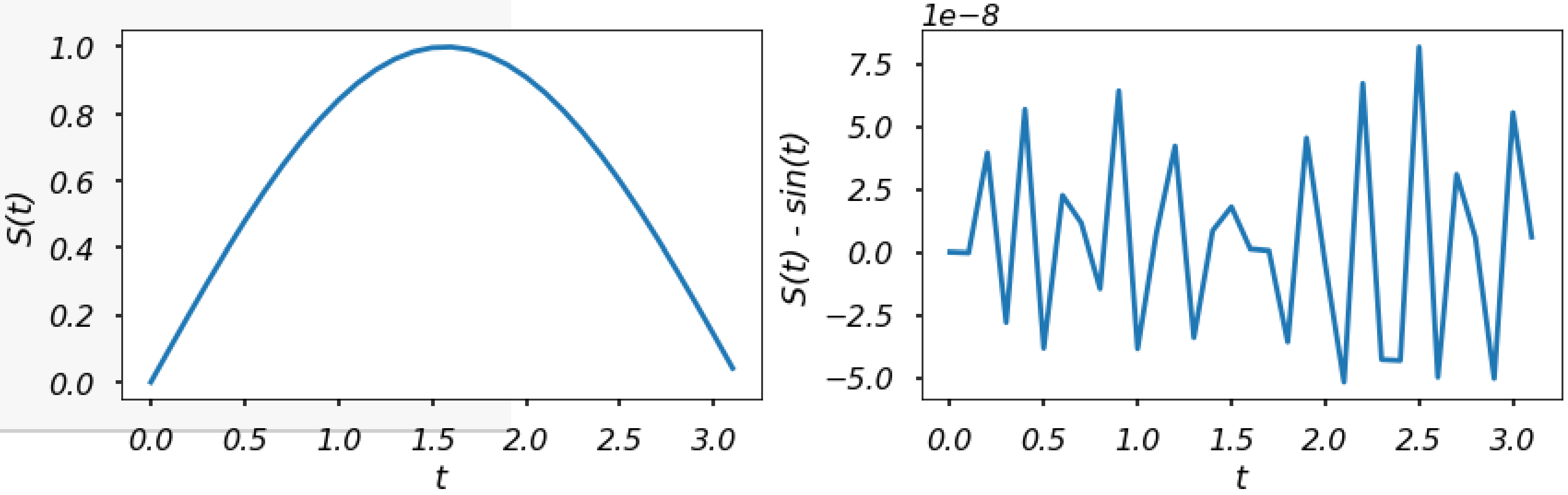
- The **left** figure shows the integration of $\frac{dS(t)}{dt} = \cos(t)$ with **solve_ivp**. The **right** figure computes the **difference** between the solution of the integration by **solve_ivp** and the evaluation of the **analytical** solution to this ODE.



- As can be seen from the figure, the difference between the approximate and exact solution to this ODE is **small**.

- 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**.
- **Example**: Using the **rtol** and **atol** to make the difference between the approximate and exact solution is less than 1e-7.

```
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()
```



Python ODE Solvers

- **Example**: Consider the ODE

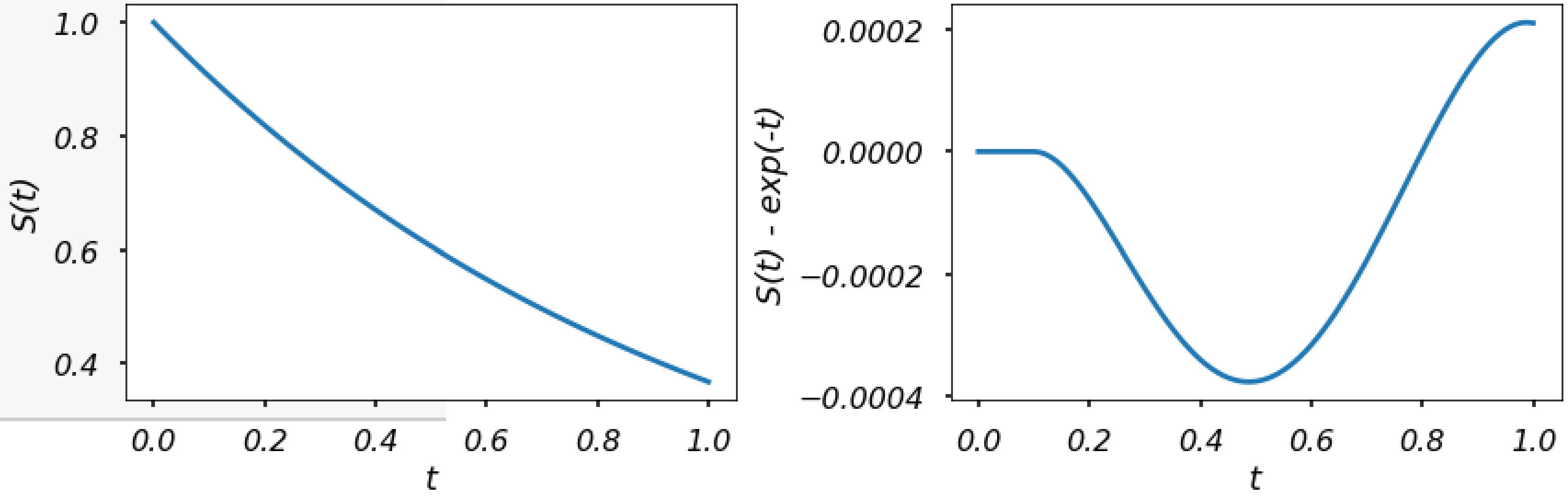
$$\frac{dS(t)}{dt} = -S(t)$$

with an initial value of $S_0 = 1$. The exact solution to this problem is $S(t) = e^{-t}$. Use **solve_ivp** to approximate the solution to this initial value problem over the interval [0,1]. Plot the approximate solution versus the exact solution, and the relative error over time.

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 figure shows the corresponding numerical results. As in the previous example, the **difference** between the result of **solve_ivp** and the evaluation of the **analytical** solution by Python is **very small** in comparison to the value of the function.

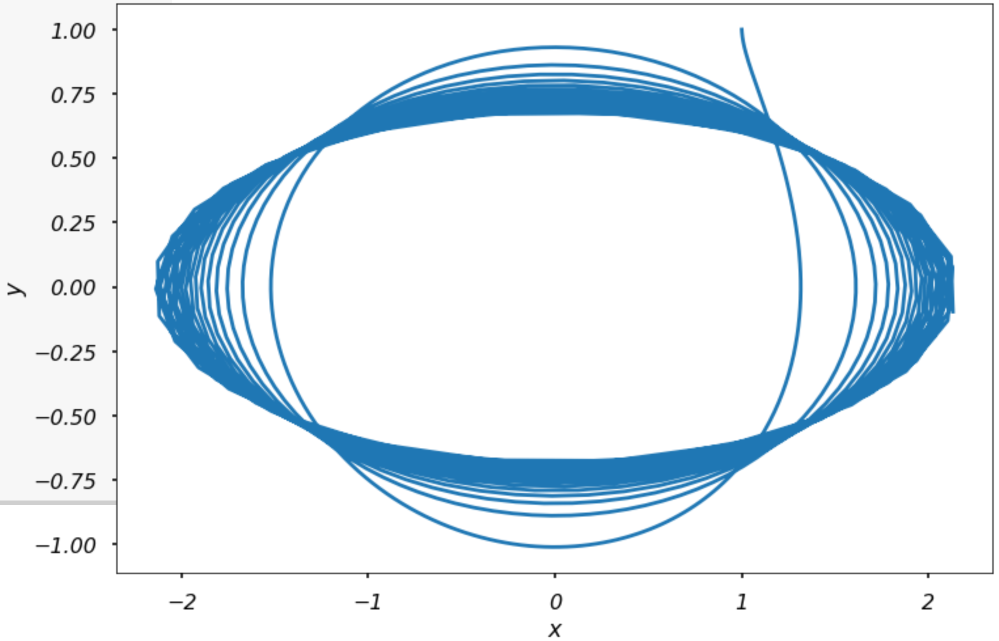


- **Example**: Let the state of a system be defined by $S(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$, and let the evolution of the system be defined by the ODE

$$\frac{dS(t)}{dt} = \begin{bmatrix} 0 & t^2 \\ -t & 0 \end{bmatrix} S(t)$$

Use **solve_ivp** to solve this ODE for the time interval [0,10] with an initial value of $S_0 = [1 \;\; 1]$. Plot the solution in $(x(t), y(t))$.

```
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()
```



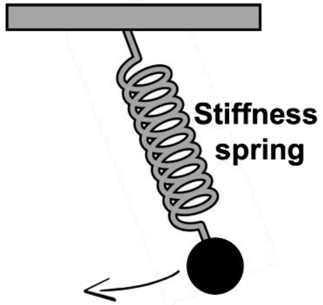
Advanced Topics

- In this section, we will briefly discuss some more advanced topics in IVP ODE.
- We will **not go** into the **details** of them, but if you are interested, we do suggest you check out some great books, such as **Ordinary Differential Equations** by Morris Tenenbaum and Harry Pollard, **Numerical Methods for Engineers and Scientists** by Amos Gilat and Vish Subramaniam, as well as **Numerical Methods for Ordinary Differential Equations** by J.C. Butcher.

Multistep Methods

- So far, most of the methods we discussed are called **one-step methods** because the **approximation** for the **next point** t_{j+1} is obtained by using information only from $S(t_j)$ and t_j at the **previous point**. Although some of the methods, such as RK methods, might use function-evaluation information at points between t_j and t_{j+1}, they do not retain the information for direct use in future approximations.
- The **multistep methods** attempt to gain efficiency by using **two** or **more previous points** to approximate the solution at the **next point** t_{j+1}. For **linear multistep methods**, we can use a linear combination of the previous points and derivative values to approximate the next point. The coefficients can be determined using **polynomial interpolation** we discussed in Week 7.
- There are **three families** of **linear multistep** methods commonly used: **Adams–Bashforth** methods, **Adams–Moulton** methods, and the **backward differentiation formulas** (BDFs).

Stiffness ODE



- **Stiffness** is a difficult and important concept in the **numerical solution** of ODEs. A stiff ODE equation will make the solution being sought **vary slowly**, and **not stable**, i.e. if there are **nearby solutions**, the solution will **change dramatically**. This will force us to take **small steps** to obtain **reasonable results**. Therefore, stiffness is usually an **efficiency issue**: if we do not care about computation cost, we would not be concerned about stiffness.
- In science and engineering, we often need to model physical phenomena with very **different time scales** or **spatial scales**. These applications usually lead to systems of ODEs whose solution include **several terms** with **magnitudes** varying with time at a significantly **different rate**. For example, the above figure shows a **spring mass system**, which the mass can swings from left to right as well as oscillates up and down due to the spring. Therefore, we have **two different time scales**, that is the time scale of the **swinging motion** as well as the **oscillation motion**. If the spring is really **stiff**, thus the **oscillation** motion time scale will be much **smaller** than that of the **swinging** motion. In order to study the system, we have to use a **very tiny time step** to get a **good solution** for the oscillation.

Tips

- **Depending** on the **properties** of the ODE and the **desired level of accuracy**, you might need to use **different methods** for **solve_ivp**.
- There are many methods to choose from for the method argument in **solve_ivp**; browse through the **documentation** for additional information.
- As suggested by the documentation, use the "**RK45**" or "**RK23**" method for **non-stiff** problems and "**Radau**" or "**BDF**" for **stiff** problems.
- If not sure, first try to run "RK45". Should this solution experience an **unusually high** number of **iterations**, **diverges**, or **fails**, this problem is likely to be **stiff**, and you should use "Radau" or "BDF". "**LSODA**" can also be a good universal choice, but it might be somewhat less convenient to work with as it wraps old Fortran code.

Practice

The **logistics equation** is a simple differential equation model that can be used to relate the change in population $\frac{dP}{dt}$ to the current population, P, given a growth rate, r, and a carrying capacity, K. The logistics equation can be expressed by:

$$\frac{dP}{dt} = rP\left(1 - \frac{P}{K}\right)$$

Write a function $\boldsymbol{my\_logistics\_eq(t, P, r, K)}$ that represents the logistics equation with a return of dP. Note that this format allows **my_logistics_eq** to be used as an input argument to **solve_ivp**. You may assume that the arguments $dP, t, P, r,$ and K are all scalars, and dP is the value $\frac{dP}{dt}$ given r, P, and K. Note that the input argument, t, is obligatory if **my_logistics_eq** is to be used as an input argument to **solve_ivp**, even though it is part of the differential equation.

Practice

Note: The logistics equation has an **analytic solution** defined by:

$$P(t) = \frac{KP_0e^{rt}}{K + P_0(e^{rt} - 1)}$$

where P_0 is the initial population. As an exercise, you should verify that this equation is a solution to the logistics equation.

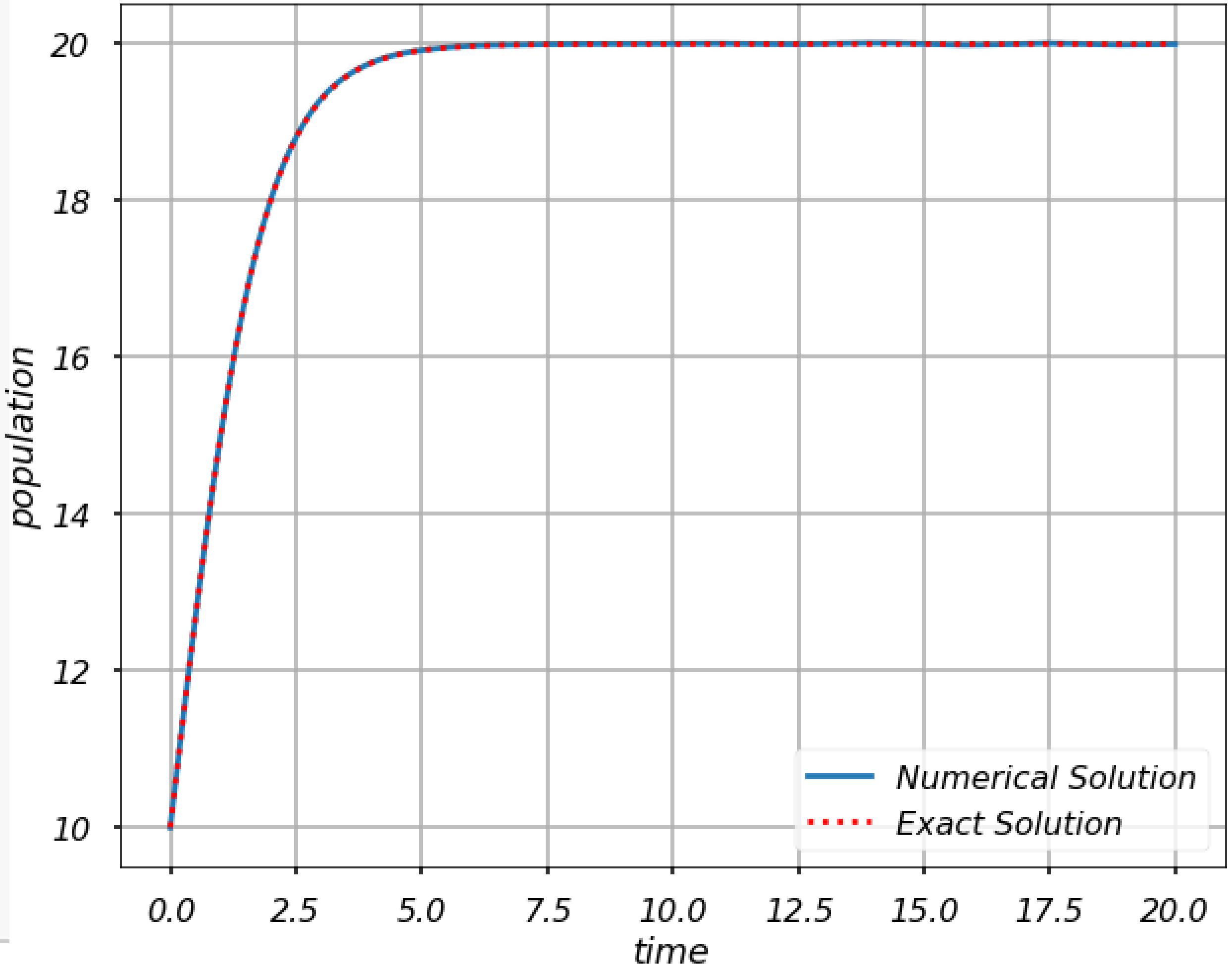
```python
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
from functools import partial
plt.style.use('seaborn-poster')

%matplotlib inline
```

```python
def my_logistics_eq(t, P, r, K):
    # put your code here

    return dP
```

```python
dP = my_logistics_eq(0, 10, 1.1, 15)
dP
```

```
]:  3.6666666666666667
```

```python
from functools import partial

t0 = 0
tf = 20
P0 = 10
r = 1.1
K = 20
t = np.linspace(0, 20, 2001)

f = partial(my_logistics_eq, r=r, K=K)
sol=solve_ivp(f,[t0,tf],[P0],t_eval=t)

plt.figure(figsize = (10, 8))
plt.plot(sol.t, sol.y[0])
plt.plot(t, \
  K*P0*np.exp(r*t)/(K+P0*(np.exp(r*t)-1)),'r:')
plt.xlabel('time')
plt.ylabel('population')

plt.legend(['Numerical Solution', \
            'Exact Solution'])
plt.grid(True)
plt.show()
```



Next Week's Outline

- ODE Boundary Value Problem Statement
- The Shooting Method
- Finite Difference Method
- Numerical Error and Instability

References

- Kong, Qingkai; Siauw, Timmy, and Bayen, Alexandre. 2020. Python Programming and Numerical Methods: A Guide for Engineers and Scientists. Academic Press. https://www.elsevier.com/books/python-programming-and-numerical-methods/kong/978-0-12-819549-9
- Other online and offline references

Visi

Menjadi Program Studi Strata Satu Informatika **unggulan** yang menghasilkan lulusan **berwawasan internasional** yang **kompeten** di bidang Ilmu Komputer (*Computer Science*), **berjiwa wirausaha** dan **berbudi pekerti luhur**.

INFORMATIKA UMN UNIVERSITAS MULTIMEDIA NUSANTARA

Misi

1. Menyelenggarakan pembelajaran dengan teknologi dan kurikulum terbaik serta didukung tenaga pengajar profesional.
2. Melaksanakan kegiatan penelitian di bidang Informatika untuk memajukan ilmu dan teknologi Informatika.
3. Melaksanakan kegiatan pengabdian kepada masyarakat berbasis ilmu dan teknologi Informatika dalam rangka mengamalkan ilmu dan teknologi Informatika.