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

IF420 – ANALISIS NUMERIK

Pertemuan ke 13 – Ordinary Differential Equation - Boundary 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 13: Mahasiswa mampu memahami dan menerapkan Ordinary Differential Equations: Permasalahan nilai batas – C3

Reviews

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

Outlines

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

Motivation

- After the discussion of ODE **initial value problems**, in this lesson, we will introduce another type of problems - the **boundary value problems**.
- The **boundary value problem** in ODE is an **ordinary differential equation** together with a **set of additional constraints**, that is **boundary conditions**.
- There are many **boundary value problems** in science and engineering. Therefore, we cover the basics of ordinary differential equations with **specified boundary values**.
- We will discuss **two methods** for solving boundary value problems, the **shooting method** and **finite difference method**.

ODE Boundary Value Problem Statement

- In the previous lesson, we talked about **ordinary differential equation initial value problems**.
- We can see that in the **initial value problems**, all the **known values** are **specified** at the **same value** of the **independent variable**, usually at the **lower boundary** of the interval, thus this is where the term '**initial**' comes from.
- While in this lesson, we will discuss another type of problems - **boundary value problems.**
- As the name suggested, the **known values** are **specified** at the **extremes** of the **independent variable**, therefore, **boundaries** of the **interval**.

ODE Boundary Value Problem Statement

- For example, if we have a simple **2nd order ordinary differential equation**,

$$\frac{d^2f(x)}{dx^2} = \frac{df(x)}{dx} + 3$$

  if the **independent variable** is **over** the domain of [0, 20], the initial value problem will have the **two conditions** on the value 0, that is, we know the value of $f(0)$ and $f'(0)$.

- In contrast, the **boundary value** problems will specify the values at $x = 0$ and $x = 20$.
- Note that solving a **first-order ODE** to get a particular solution, we need **one constraint**, while an $\boldsymbol{n}$**-th order ODE**, we need n **constraints**.

- The **boundary value problem** statement for an **n-th order ordinary differential equation** is stated as:

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

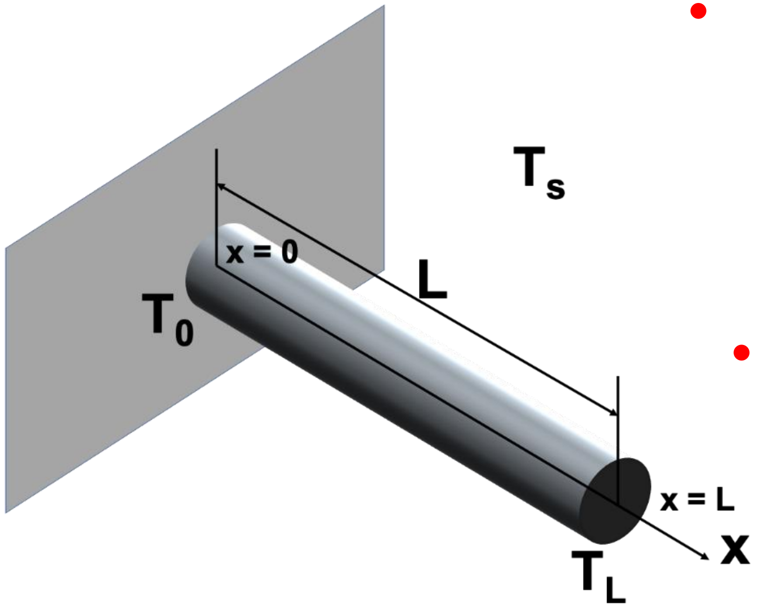
- To solve this equation on an interval of $x \in [a, b]$, we need **n known boundary conditions** at value a and b.
- For the **2nd order** case, since we can have the **boundary condition** either be a value of $f(x)$ or a value of derivative $f'(x)$, we can have **several different cases** for the specified values.
- For example, we can have the **boundary condition values** specified as:

1. **Two values** of $f(x)$ are given, that is $f(a)$ and $f(b)$ are known.
2. **Two derivatives** of $f'(x)$ are given, that is $f'(a)$ and $f'(b)$ are known.
3. Or **mixed conditions** from the above two cases are known, that is either $f(a)$ and $f'(b)$ are known or $f'(a)$ and $f(b)$ are known.

ODE Boundary Value Problem Statement

UMN
UNIVERSITAS MULTIMEDIA NUSANTARA

- Anyway, to get the particular solution, we need **two boundary conditions** to get the solution.
- The **second-order ODE boundary value** problem is also called **Two-Point boundary value problems**.
- The **higher order ODE problems** need additional boundary conditions, usually the values of **higher derivatives** of the **independent** variables.
- In this lesson, let's focus on the **two-point boundary value** problems.
- Let's see an example of the **boundary value problem** and see how we can solve it in the next few sections.



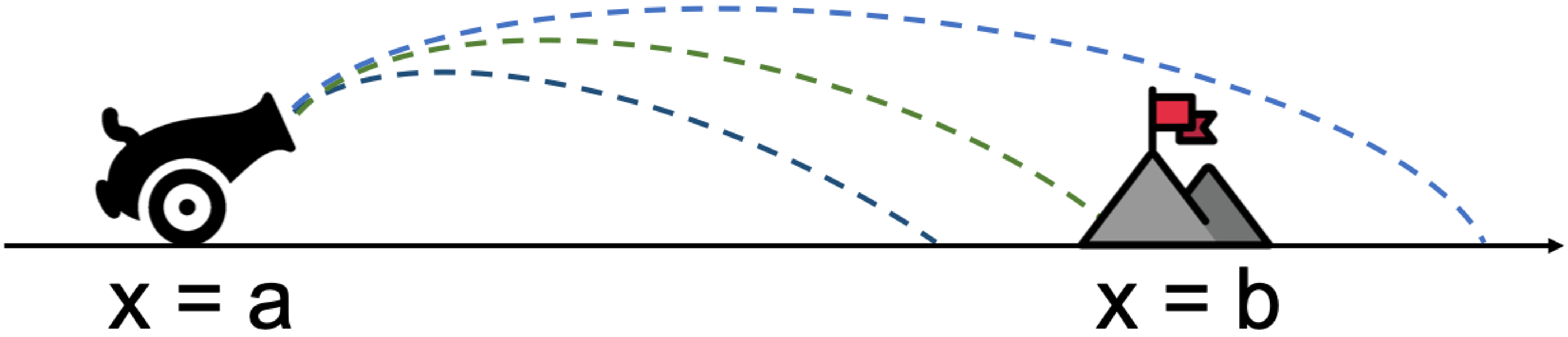
- **Fins** are used in many applications to increase the **heat transfer from surfaces**. Usually, the design of cooling pin fins is encountered in many applications, such as the pin fin used as a heat sink for cooling an object.
  - We can model the **temperature distribution** in a pin fin as shown in the figure, where the length of the fin is L, and the start and the end of the fin is $x = 0$ as well as $x = L$. The temperature at the two ends are T_0 and T_L. T_S is the temperature of the surrounding environment.
- If we consider both convection and radiation, the **steady-state temperature distribution** of the pin fin $T(x)$ can be modeled with the following equation:

$$\frac{d^2T}{dx^2} - \alpha_1(T - T_S) - \alpha_2(T^4 - T_S^4) = 0$$

with the boundary conditions: $T(0) = T_0$ and $T(L) = T_L$, and α_1 and α_2 are coefficients. This is a second-order ODE with two boundary conditions; therefore, we can solve it to get particular solutions.

The Shooting Method

- The **shooting method** was developed with the goal of **transforming** the ODE **boundary value** problems **to** an equivalent **initial value** problems, then we can solve it using the methods we learned from the previous lesson.
- In the **initial value** problems, we can start at the initial value and march forward to get the solution. But this method is **not working** for the **boundary value** problems, because there are **not enough initial value** conditions to solve the ODE to get a unique solution. Therefore, the **shooting method** was developed to overcome this difficulty.



The Shooting Method

- The name of the shooting method is derived from **analogy** with the **target shooting.** As shown in the previous figure, we shoot the target and observe where it hits the target. Based on the errors, we can adjust our aim and shoot again in the hope that it will hit closer to the target.
- We can see from the analogy that the **shooting** method is an **iterative method**.
- Let's see how the shooting methods works using the **second-order** ODE given $f(a) = f_a$ and $f(b) = f_b$,

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

- **Step 1**: We start the whole process by guessing $f'(a) = \alpha$, together with $f(a) = f_a$, we turn the above problem into an **initial value problem** with two conditions all on value $x = a$. This is the **aim** step.
- **Step 2**: Using what we learned from previous lesson, i.e. we can use **Runge-Kutta** method, to integrate to the other boundary b to find $f(b) = f_\beta$. This is the **shooting** step.
- **Step 3**: Now we compare the value of f_β with f_b, usually our initial guess is not good, and $f_\beta \neq f_b$, but what we want is $f_\beta - f_b = 0$; therefore, we adjust our initial guesses and repeat. Until the **error** is **acceptable**, we can stop. This is the **iterative** step.
- We can see that the idea behind the **shooting methods** is very simple. But the comparing and finding the **best guesses** are not easy, this procedure is **very tedious**. But essentially, finding the best guess to get $f_\beta - f_b = 0$ is a **root-finding** problem. Once we realize this, we have a systematic way to search for the best guess. Since f_β is a function of α; therefore, the problem becomes finding the root of $g(\alpha) - f_b = 0$. We can use any methods from Week 9 (**Root Finding**) to solve it.

The Shooting Method

- **Example**: We are going out to launch a rocket, and let $y(t)$ is the altitude (meters from the surface) of the rocket at time t. We know the gravity $g = 9.8\ m/s^2$. If we want to have the rocket at $50\ m$ off the ground after 5 seconds after launching, what should be the velocity at launching? (we ignore the drag of the air resistance).
- To answer this question, we can frame the problem into a **boundary value problem** for a **second-order** ODE. The ODE is:

$$\frac{d^2y}{dt^2} = -g$$

with the two boundary conditions are: $y(0) = 0$ and $y(5) = 50$. And we want to answer the question, what's the $y'(0)$ at the launching?

The Shooting Method

- This is a quite simple question, we can solve it **analytically** easily, with the correct answer $y'(0) = 34.5$.
- Now let's solve it using the **shooting method**. First, we will **reduce** the **order** of the function, the **second-order** ODE becomes:

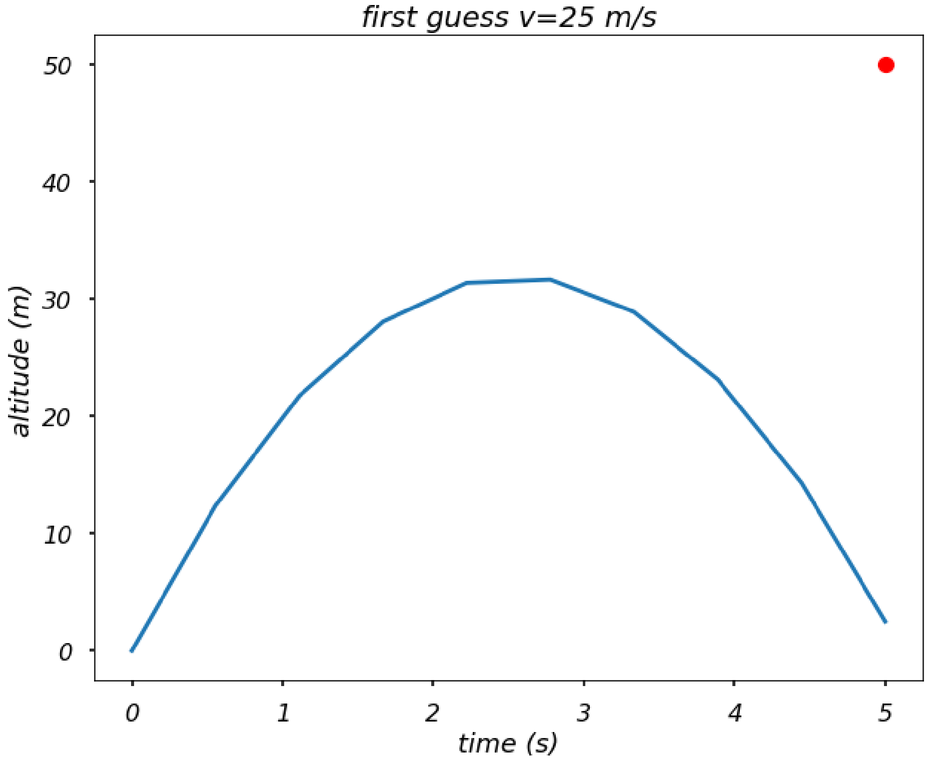
$$\frac{dy}{dt} = v$$
$$\frac{dv}{dt} = -g$$

- Therefore, we have $S(t) = \begin{bmatrix} y(t) \\ v(t) \end{bmatrix}$:

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

- Let's start our **first guess**, we guess the velocity at launching is 25 m/s.

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

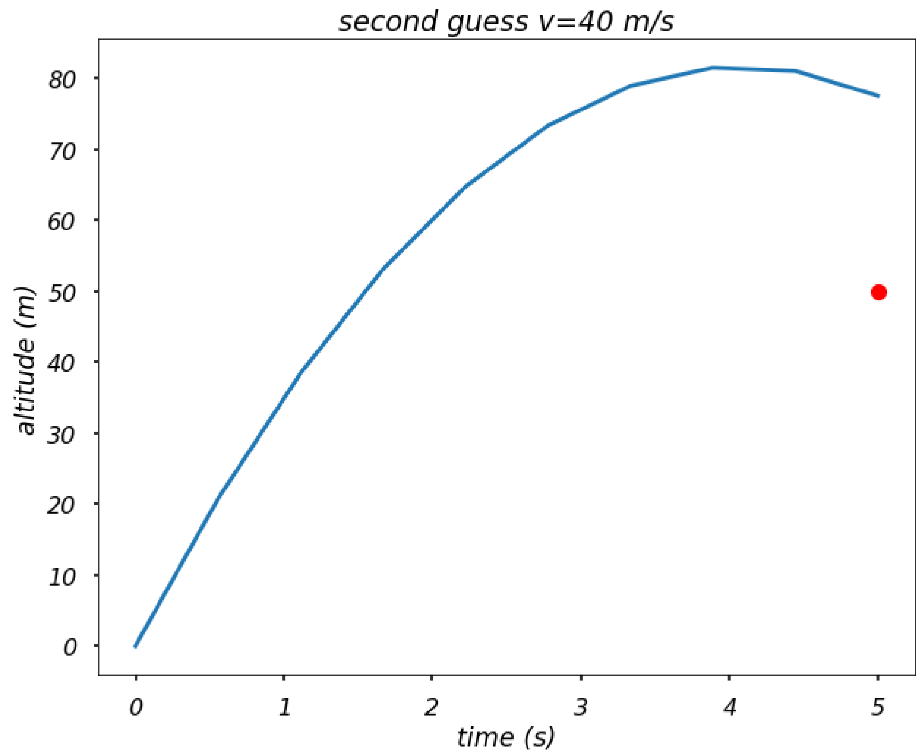



```
F = lambda t, s: \
  np.dot(np.array([[0,1],[0,-9.8/s[1]]]),s)

t_span = np.linspace(0, 5, 100)
y0 = 0
v0 = 25
t_eval = np.linspace(0, 5, 10)
sol = solve_ivp(F, [0, 5], \
                [y0, v0], t_eval = t_eval)

plt.figure(figsize = (10, 8))
plt.plot(sol.t, sol.y[0])
plt.plot(5, 50, 'ro')
plt.xlabel('time (s)')
plt.ylabel('altitude (m)')
plt.title(f'first guess v={v0} m/s')
plt.show()
```

The Shooting Method

- From the figure we see that the first guess is a **little small**, since with this velocity at $5\ s$, the altitude of the rocket is less than $10\ m$. The red dot in the figure is the target we want to hit.
- Now let's adjust our guess and **increase** the **velocity** to $40\ m/s$.

```python
v0 = 40
sol = solve_ivp(F, [0, 5], \
                [y0, v0], t_eval = t_eval)

plt.figure(figsize = (10, 8))
plt.plot(sol.t, sol.y[0])
plt.plot(5, 50, 'ro')
plt.xlabel('time (s)')
plt.ylabel('altitude (m)')
plt.title(f'second guess v={v0} m/s')
plt.show()
```



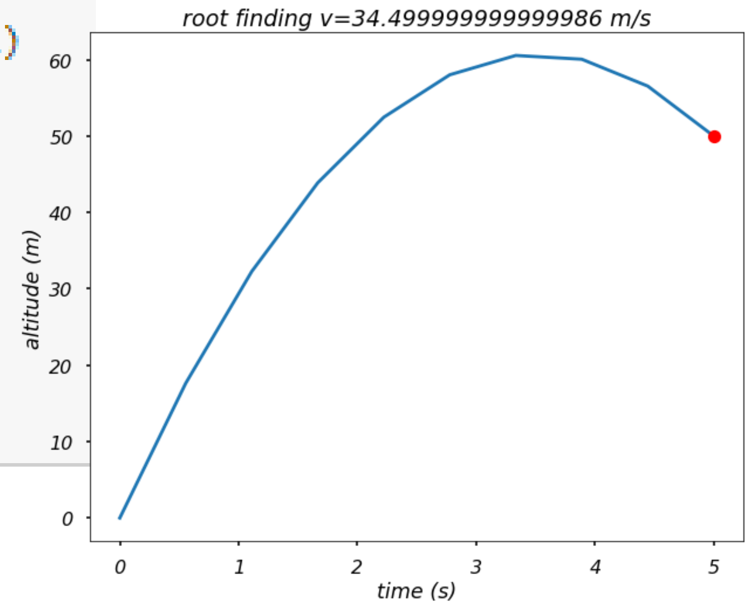
The Shooting Method

- We can see this time we **overestimate** the **velocity**. Therefore, this **random guess** is not easy to find the **best result**. As we mentioned above, if we treat this procedure as **root-finding**, then we will have a good way to search the **best result**.
- Let's use Python's **fsolve** to find the **root**. We can see from the following example, we find the **correct answer** directly.

```
from scipy.optimize import fsolve

def objective(v0):
    sol = solve_ivp(F, [0, 5], \
            [y0, v0], t_eval = t_eval)
    y = sol.y[0]
    return y[-1] - 50

v0, = fsolve(objective, 10)
print(v0)
```

```
34.499999999999986
```

```
sol = solve_ivp(F, [0, 5], \
            [y0, v0], t_eval = t_eval)

plt.figure(figsize = (10, 8))
plt.plot(sol.t, sol.y[0])
plt.plot(5, 50, 'ro')
plt.xlabel('time (s)')
plt.ylabel('altitude (m)')
plt.title(f'root finding v={v0} m/s')
plt.show()
```



The Shooting Method

- **Example**: Let’s change the initial guess and see if that changes our result.

```python
for v0_guess in range(1, 100, 10):
    v0, = fsolve(objective, v0_guess)
    print('Init: %d, Result: %.1f' \
          %(v0_guess, v0))
```

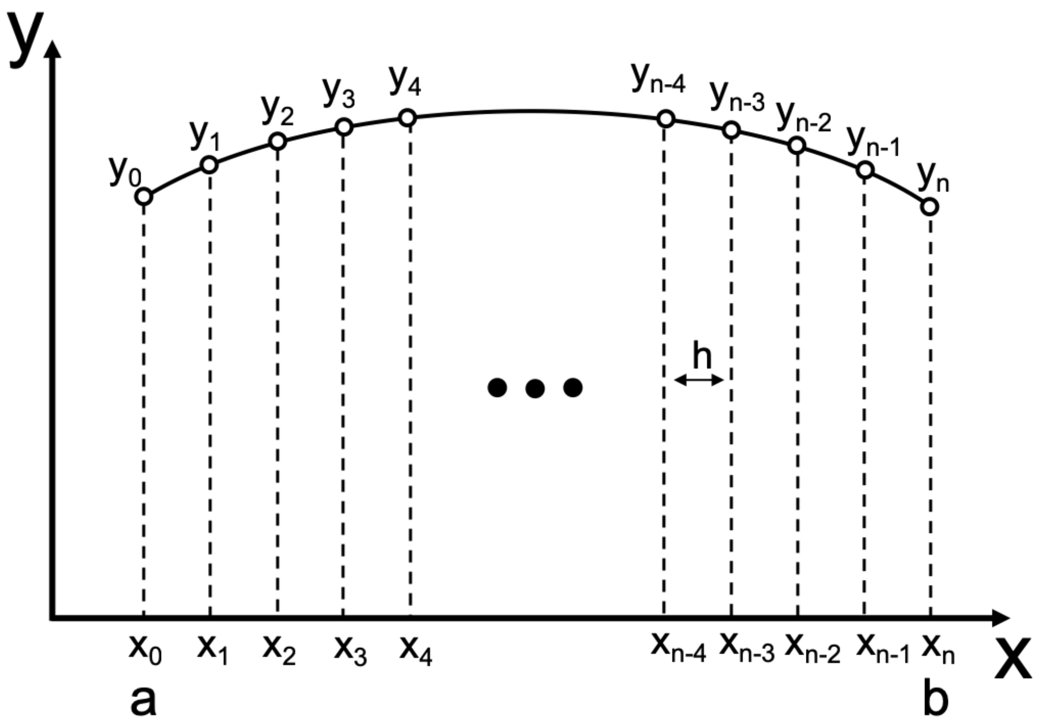
```
Init: 1, Result: 34.5
Init: 11, Result: 34.5
Init: 21, Result: 34.5
Init: 31, Result: 34.5
Init: 41, Result: 34.5
Init: 51, Result: 34.5
Init: 61, Result: 34.5
Init: 71, Result: 34.5
Init: 81, Result: 34.5
Init: 91, Result: 34.5
```

- We can see that change on the initial guesses doesn’t change the result here, which means that the **stability** of the method is **good**.

Finite Difference Method

- Another way to solve the ODE **boundary value** problems is the **finite difference method**, where we can use **finite difference** formulas at **evenly spaced grid points** to **approximate** the **differential equations**. This way, we can **transform** a **differential equation** into a **system** of **algebraic equations** to solve.

- In the **finite difference** method, the derivatives in the differential equation are approximated using the **finite difference formulas**. We can divide the interval of $[a, b]$ into n **equal subintervals** of length h as shown in the following figure.



Finite Difference Method

- Commonly, we usually use the **central difference** formulas in the **finite difference** methods due to the fact that they yield **better accuracy**.
- The differential equation is enforced only at the **grid points**, and the **first** and **second derivatives** are:

$$\frac{dy}{dx} = \frac{y_{i+1} - y_{i-1}}{2h}$$

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

- These **finite difference expressions** are used to **replace** the **derivatives** of y in the differential equation which leads to a system of $n + 1$ **linear algebraic equations** if the differential equation is **linear**. If the differential equation is **nonlinear**, the algebraic equations will also be **nonlinear**.

Finite Difference Method

- **Example**: Solve the rocket problem in the previous section using the **finite difference** method, plot the altitude of the rocket after launching. The ODE is

$$\frac{d^2y}{dt^2} = -g$$

with the two boundary conditions are: $y(0) = 0$ and $y(5) = 50$. Let's take $n = 10$.

- Since the time interval is $[0,5]$ and we have $n = 10$; therefore, $h = 0.5$, using the **finite difference approximated derivatives**, we have

$$y_0 = 0$$

$$y_{i-1} - 2y_i + y_{i+1} = -gh^2, \qquad i = 1,2,\ldots,n-1$$

$$y_{10} = 50$$

Finite Difference Method

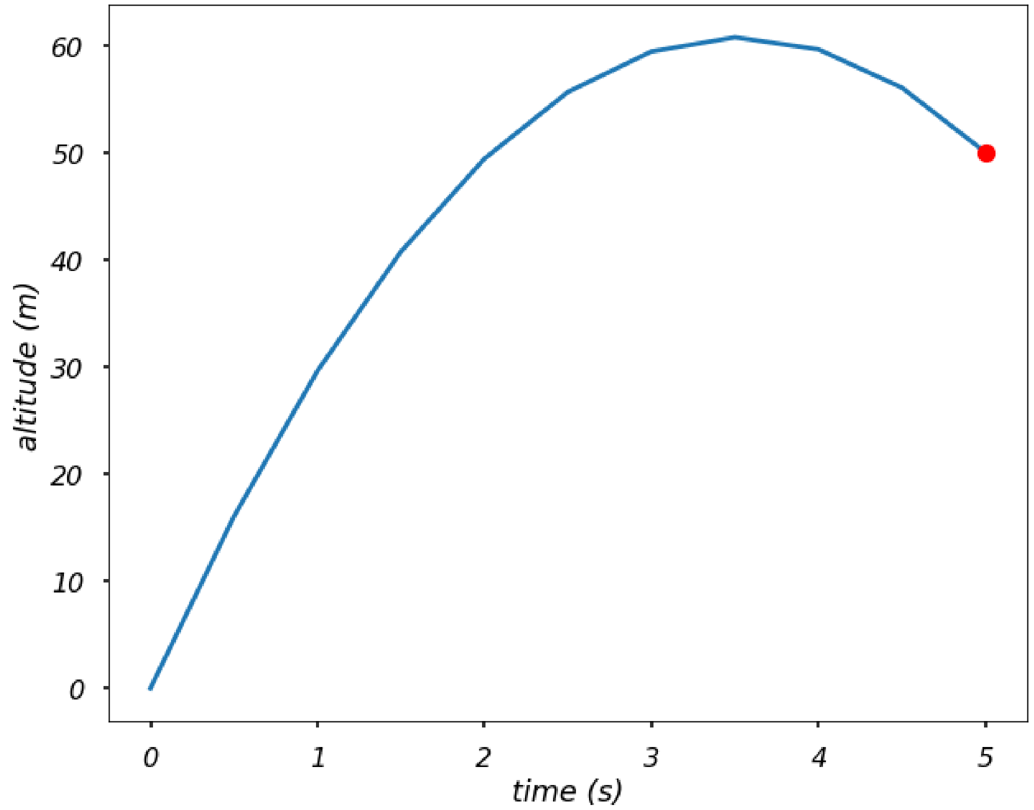
- If we use **matrix notation**, we will have:

$$\begin{bmatrix} 1 & 0 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \dots \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} 0 \\ -gh^2 \\ \dots \\ -gh^2 \\ 50 \end{bmatrix}$$

- Therefore, we have **11 equations** in the system, we can solve it using the method we learned in Week 4 (**Linear Algebra and Systems of Linear Equations**).

```python
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-poster')
%matplotlib inline

n = 10
h = (5-0) / n

# Get A
A = np.zeros((n+1, n+1))
A[0, 0] = 1
A[n, n] = 1
for i in range(1, n):
    A[i, i-1] = 1
    A[i, i] = -2
    A[i, i+1] = 1

print(A)

# Get b
b = np.zeros(n+1)
b[1:-1] = -9.8*h**2
b[-1] = 50
print(b)

# solve the linear equations
y = np.linalg.solve(A, b)

t = np.linspace(0, 5, 11)

plt.figure(figsize=(10,8))
plt.plot(t, y)
plt.plot(5, 50, 'ro')
plt.xlabel('time (s)')
plt.ylabel('altitude (m)')
plt.show()
```

```
[[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 1. -2.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1. -2.  1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1. -2.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1. -2.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1. -2.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1. -2.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1. -2.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1. -2.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  1. -2.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]]
[ 0.   -2.45 -2.45 -2.45 -2.45 -2.45 -2.45 -2.45 -2.45 -2.45 50.  ]
```



Finite Difference Method

- Now, let's solve $y'(0)$, from the **finite difference** formula, we know that $\frac{dy}{dx} = \frac{y_{i+1}-y_{i-1}}{2h}$, which means that $y'(0) = \frac{y_1-y_{-1}}{2h}$, but we **don't know** what is y_{-1}.
- Actually, we can calculate y_{-1} since we know the y values on each grid point. From the **2nd derivative finite difference** formula, we know that $\frac{y_{-1}-2y_0+y_1}{h^2} = -g$, therefore, we can solve for y_{-1} and then get the **launching velocity**. See the calculation below.

```
y_n1 = -9.8*h**2 + 2*y[0] - y[1]
(y[1] - y_n1) / (2*h)
```

```
]: 34.5
```

- We can see that we get the **correct launching velocity** using the **finite difference method**. To make you more comfortable with the method, let's see another example.

- **Example**: Use finite difference method to solve the following linear boundary value problem

$$y'' = -4y + 4x$$

with the boundary conditions as $y(0) = 0$ and $y'(\pi/2) = 0$. The exact solution of the problem is $y = x - \sin 2x$, plot the errors against the n grid points (n from 3 to 100) for the boundary point $y(\pi/2)$.

- Using the **finite difference approximated derivatives**, we have

$$y_0 = 0$$

$$y_{i-1} - 2y_i + y_{i+1} - h^2(-4y_i + 4x_i) = 0, \qquad i = 1,2,\dots,n-1$$

$$2y_{n-1} - 2y_n - h^2(-4y_n + 4x_n) = 0$$

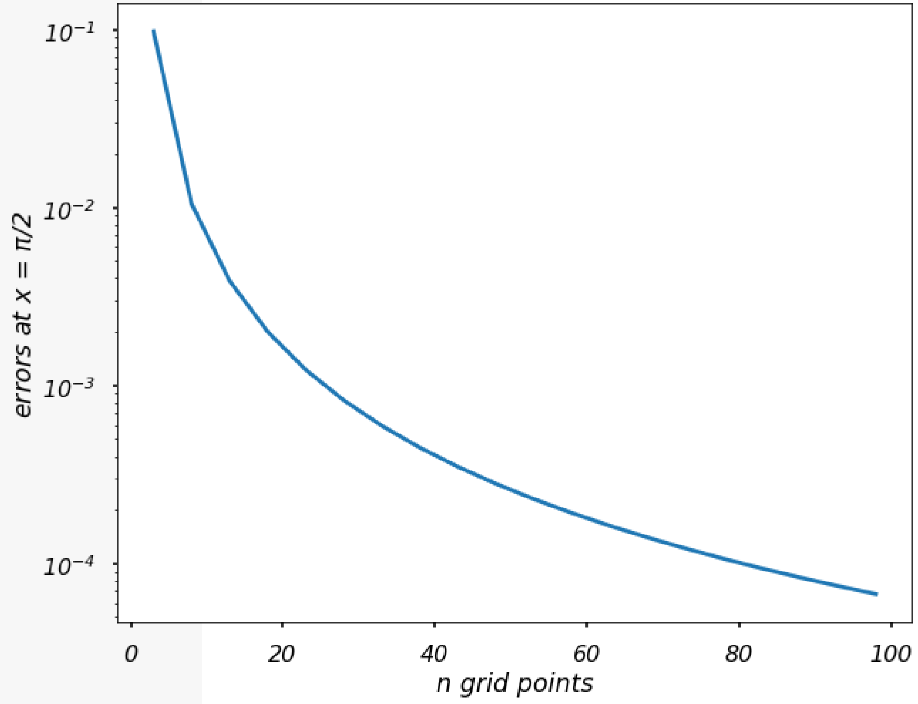
- The **last equation** is derived from the fact that $\frac{y_{n+1} - y_{n-1}}{2h} = 0$ (the boundary condition $y'(\pi/2) = 0$). Therefore, $y_{n+1} = y_{n-1}$.

- If we use matrix notation, we will have:

$$\begin{bmatrix} 1 & 0 & & & \\ 1 & -2+4h^2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2+4h^2 & 1 \\ & & & 2 & -2+4h^2 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \dots \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} 0 \\ 4h^2 x_1 \\ \dots \\ 4h^2 x_{n-1} \\ 4h^2 x_n \end{bmatrix}$$

```
def get_a_b(n):
    h = (np.pi/2-0) / n
    x = np.linspace(0, np.pi/2, n+1)
    # Get A
    A = np.zeros((n+1, n+1))
    A[0, 0] = 1
    A[n, n] = -2+4*h**2
    A[n, n-1] = 2
    for i in range(1, n):
        A[i, i-1] = 1
        A[i, i] = -2+4*h**2
        A[i, i+1] = 1

    # Get b
    b = np.zeros(n+1)
    for i in range(1, n+1):
        b[i] = 4*h**2*x[i]

    return x, A, b

x = np.pi/2
v = x - np.sin(2*x)
```

```
n_s = []
errors = []

for n in range(3, 100, 5):
    x, A, b = get_a_b(n)
    y = np.linalg.solve(A, b)
    n_s.append(n)
    e = v - y[-1]
    errors.append(e)

plt.figure(figsize = (10,8))
plt.plot(n_s, errors)
plt.yscale('log')
plt.xlabel('n grid points')
plt.ylabel('errors at x = $\pi/2$')
plt.show()
```



Finite Difference Method

- We can see with **denser grid points**; we are **approaching** the **exact solution** on the **boundary point**.
- The **finite difference** method can be also applied to **higher-order ODEs**, but it needs **approximation** of the **higher-order derivatives** using the **finite difference** formula.
- For example, if we are solving a **fourth-order ODE**, we will need to use the following:

$$\frac{d^4y}{dx^4} = \frac{y_{i-2} - 4y_{i-1} + 6y_i - 4y_{i+1} + y_{i+2}}{h^4}$$

- We won't talk more on the **higher-order ODEs**, since the idea behind to solve it is similar to the **second-order ODE** we discussed before.

Numerical Error and Instability (BVP)

- **Boundary value problems** also have the **two main issues** we talked in the previous lesson, the numerical **error** - **accuracy** and the **stability**. Depending on the different methods used, either the **shooting** or **finite difference** method, they are **different**.
- For the **shooting method**, the **numerical error** is similar to what we described for the **initial value problems**, since the **shooting method** is essentially **transform** the **boundary value problem** into a **series of initial value problems**.
- In terms of the **stability** of the method, we can see from the example in the **shooting method** that even our initial guesses are not close to the true answer, the method returns an **accurate numerical solution**. This is due to the adding of the **rightmost constraint** keeps the errors from increasing unboundedly.

Numerical Error and Instability (BVP)

- In the case of **finite difference methods**, the numerical **error** is determined by the **order** of **accuracy** of the **numerical scheme** used. The **accuracy** of the different scheme used for derivative approximations are discussed in Week 10 (**Numerical Differentiation**).
- The **accuracy** of the **finite difference** method is determined by the **larger** of the **two truncation errors**, the **difference scheme used** for the differential equation or that of the difference scheme used to discretize the boundary conditions (we see that **step size** has a **strong effect** on the **accuracy** of the finite difference method).
- Since the finite difference methods essentially turns the **BVP** into solving a **system of equations**; therefore, the **stability** of it depends on the **stability** of the **scheme** used to solve the resulting system of equations simultaneously.

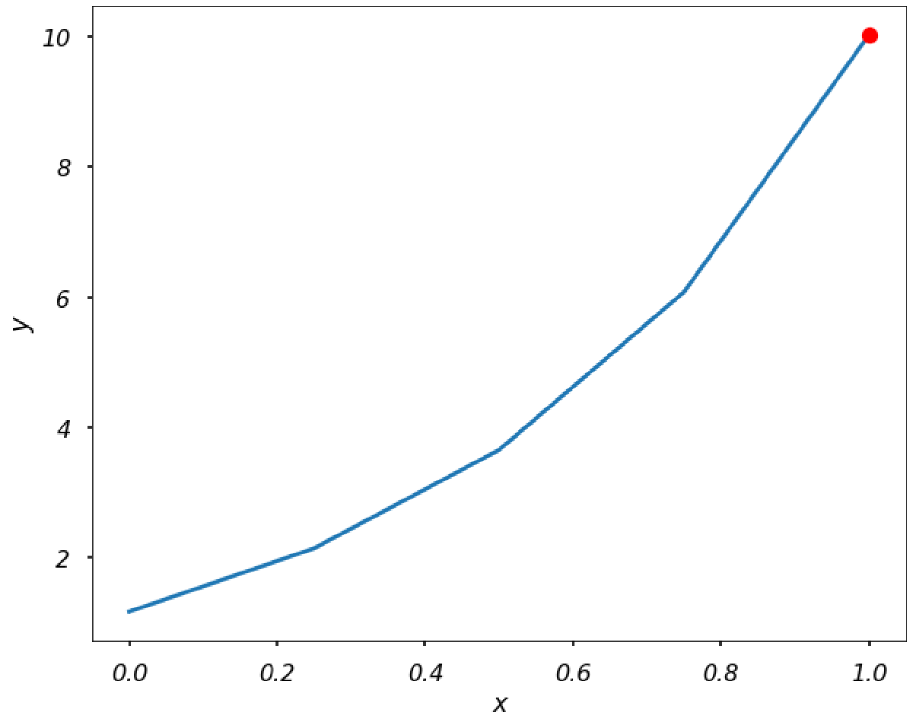
Practice

1. Using **finite difference** method to solve the following **linear boundary value problem**

$$y'' = 4y$$

with the boundary conditions as $y(0) = 1.1752$ and $y(1) = 10.0179$. Let's take $n = 4$.

```
[[ 1.     0.     0.     0.     0.    ]
 [ 1.    -2.25   1.     0.     0.    ]
 [ 0.     1.    -2.25   1.     0.    ]
 [ 0.     0.     1.    -2.25   1.    ]
 [ 0.     0.     0.     0.     1.    ]]
[ 1.1752  0.       0.       0.      10.0179]
Results:
 [[ 0.          1.1752    ]
 [ 0.25        2.14670658]
 [ 0.5         3.6548898 ]
 [ 0.75        6.07679546]
 [ 1.         10.0179    ]]
```



Next Week's Outline

- The Basics of Waves
- Discrete Fourier Transform (DFT)
- Fast Fourier Transform (FFT)
- FFT in Python

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.