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

IF420 – ANALISIS NUMERIK

Pertemuan ke 8 – Taylor Series

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 8: Mahasiswa mampu memahami dan menerapkan deret Taylor – C3

Outlines

- Expressing Functions with Taylor Series
- Approximations with Taylor Series
- Discussion on Errors

Motivation

- Many functions, such as $\sin(x)$ and $\cos(x)$, are useful for engineers and scientists, but they are impossible to compute explicitly.
- In practice, these functions can be **approximated** by **sums of functions** that are easy to compute, such as **polynomials**.
- In fact, most functions common to engineers and scientists cannot be computed without **approximations** of this kind.
- Since these functions are used so often, it is important to know how these approximations **work** and their **limitations**.
- In this lesson, we will learn about **Taylor series**, which is one method of **approximating** **complicated** **functions**.

Expressing Functions with Taylor Series

- A **sequence** is an **ordered set** of **numbers** denoted by the list of numbers inside **parentheses**. For example, $s = (s_1, s_2, s_3, \cdots)$ means s is the sequence $s_1, s_2, s_3, \cdots$ and so on.
- In this context, "**ordered**" means that s_1 **comes before** s_2, not that $s_1 < s_2$.
- Many sequences have a more **complicated** structure. For example, $s = (n^2, n \in N)$ is the sequence 0, 1, 4, 9, $\cdots$.
- A **series** is the **sum** of a **sequence** up to a **certain element**.
- An **infinite sequence** is a **sequence** with an **infinite number of terms**, and an **infinite series** is the **sum** of an **infinite sequence**.

Expressing Functions with Taylor Series

- A **Taylor series expansion** is a representation of a function by an **infinite series** of **polynomials** around a **point**.
- Mathematically, the **Taylor series** of a function, $f(x)$, is defined as:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)(x-a)^n}{n!}$$

where $f^{(n)}$ is the n-th **derivative** of f and $f^{(0)}$ is the function of f.

- **Example**: Compute the Taylor series expansion for $f(x) = 5x^2 + 3x + 5$ around $a = 0$ and $a = 1$. Verify that f and its Taylor series expansions are identical.
- First compute derivatives analytically:

$$f(x) = 5x^2 + 3x + 5$$
$$f'(x) = 10x + 3$$
$$f''(x) = 10$$

- Around $a\ = \ 0$:

$$f(x) = \frac{5x^0}{0!} + \frac{3x^1}{1!} + \frac{10x^2}{2!} + 0 + 0 + \cdots = 5x^2 + 3x + 5$$

- Around $a\ = 1$:

$$f(x) = \frac{13(x-1)^0}{0!} + \frac{13(x-1)^1}{1!} + \frac{10(x-1)^2}{2!} + 0 + 0 + \cdots$$
$$= 13 + 13x - 13 + 5x^2 - 10x + 5 = 5x^2 + 3x + 5$$

Expressing Functions with Taylor Series

- **Example**: Write the Taylor series for $\sin(x)$ around the point $a = 0$.
- Let $f(x) = \sin(x)$. Then according to the Taylor series expansion,

$$f(x) = \frac{\sin(0)}{0!}x^0 + \frac{cos(0)}{1!}x^1 + \frac{-\sin(0)}{2!}x^2 + \frac{-cos(0)}{3!}x^3 + \frac{\sin(0)}{4!}x^4 + \frac{cos(0)}{5!}x^5 + \cdots$$

- The expansion can be written compactly by the formula

$$f(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{(2n+1)}}{(2n+1)!}$$

which **ignores** the terms that contain $\sin(0)$ (i.e., the **even** terms). However, because these terms are ignored, the terms in this series and the proper Taylor series expansion are **off by a factor** of $2n + 1$; for example the $n = 0$ term in formula is the $n = 1$ term in the Taylor series, and the $n = 1$ term in the formula is the $n = 3$ term in the Taylor series.

Approximations with Taylor Series

- Clearly, it is **not useful** to express functions as **infinite sums** because we cannot even compute them that way.
- However, it is often **useful** to **approximate functions** by using an $\boldsymbol{N}$**-th order Taylor series approximation** of a function, which is a **truncation** of its **Taylor expansion** at some $n = N$.
- This technique is especially **powerful** when there is a **point** around which we have knowledge about a function for all its **derivatives**.
- For example, if we take the **Taylor expansion** of e^x around $a = 0$, then $f^{(n)}(a) = 1$ for all n, we don't even have to compute the derivatives in the Taylor expansion to approximate e^x!

Approximations with Taylor Series

- **Example**: Use Python to plot the **sin** function along with the **first**, **third**, **fifth**, and **seventh** order Taylor series approximations. Note that this is the zero-th to third in the compact formula given in the example earlier.

In [1]:

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

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

In [2]:

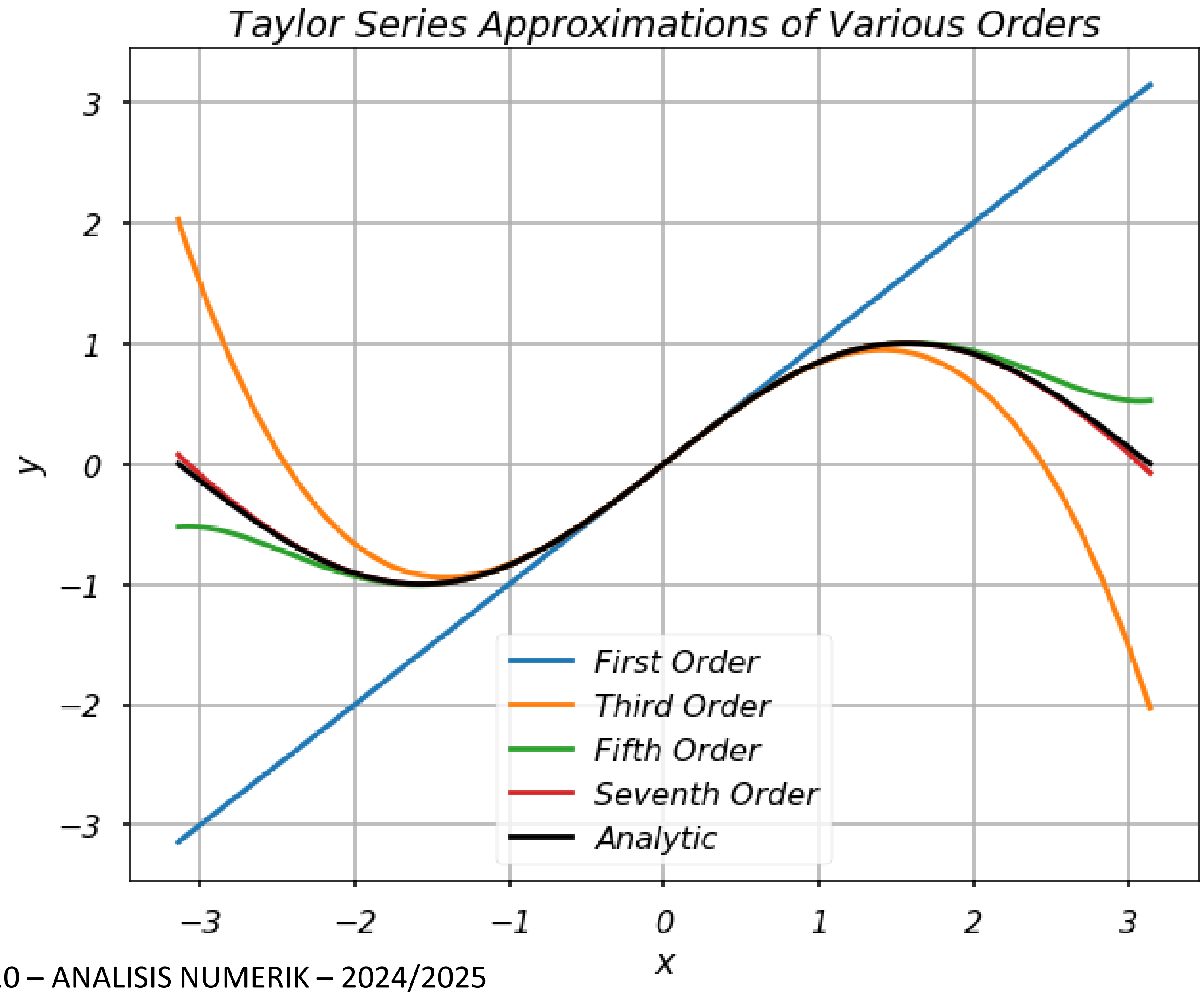
```python
x = np.linspace(-np.pi, np.pi, 200)
y = np.zeros(len(x))

labels = ['First Order', 'Third Order', 'Fifth Order', 'Seventh Order']

plt.figure(figsize = (10,8))
for n, label in zip(range(4), labels):
    y = y + ((-1)**n * (x)**(2*n+1)) / np.math.factorial(2*n+1)
    plt.plot(x,y, label = label)

plt.plot(x, np.sin(x), 'k', label = 'Analytic')
plt.grid()
plt.title('Taylor Series Approximations of Various Orders')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

Approximations with Taylor Series



- As you can see, the **approximation** **approaches** the **analytic** function quickly, even for x **not near** to $a = 0$.
- Note that in the above code, we also used a new function – **zip()**, which can allow us to loop through two parameters **range(4)** and **labels,** and use that in our plot.

Approximations with Taylor Series

- **Example**: Compute the seventh order Taylor series approximation for $\sin(x)$ around $a = 0$ at $x = \pi/2$. Compare the value to the correct value, 1.

In [3]:
```
x = np.pi/2
y = 0

for n in range(4):
    y = y + ((-1)**n * (x)**(2*n+1)) / np.math.factorial(2*n+1)

print(y)
```

```
0.9998431013994987
```

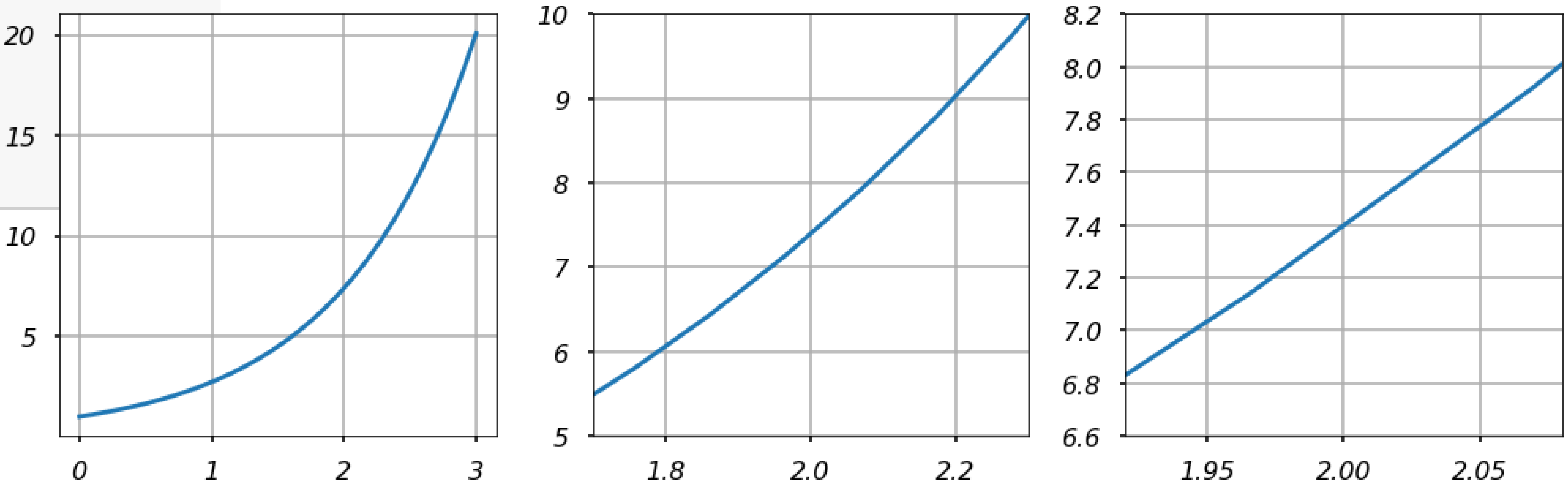
- The **seventh** order Taylor series approximation is **very close** to the theoretical value of the function even if it is computed **far** from the point around which the Taylor series was computed (i.e., $x = \pi/2$ and $a = 0$).

Approximations with Taylor Series

- The most common Taylor series approximation is the **first order approximation**, or **linear approximation**.
- Intuitively, for "**smooth**" functions the linear approximation of the function around a point, a, can be made as good as you want **provided** you stay sufficiently **close to $\boldsymbol{a}$**.
- In other words, "**smooth**" functions look more and more like a **line** the more you zoom into any point.
- This fact is depicted in the following figure, which we plot **successive levels** of **zoom** of a **smooth function** to illustrate the **linear nature** of functions **locally**.
- Linear approximations are useful **tools** when **analyzing complicated** functions **locally**.

```python
x = np.linspace(0, 3, 30)
y = np.exp(x)

plt.figure(figsize = (14, 4.5))
plt.subplot(1, 3, 1)
plt.plot(x, y)
plt.grid()
plt.subplot(1, 3, 2)
plt.plot(x, y)
plt.grid()
plt.xlim(1.7, 2.3)
plt.ylim(5, 10)
plt.subplot(1, 3, 3)
plt.plot(x, y)
plt.grid()
plt.xlim(1.92, 2.08)
plt.ylim(6.6, 8.2)
plt.tight_layout()
plt.show()
```



Approximations with Taylor Series

- **Example**: Take the **linear approximation** for e^x around the point $a = 0$. Use the linear approximation for e^x to approximate the value of e^1 and $e^{0.01}$. Use Numpy's function **exp** to compute **exp(1)** and **exp(0.01)** for comparison.
- The linear approximation of e^x around $a = 0$ is $1 + x$.
- Numpy's **exp** function gives the following:

```
In [5]:  np.exp(1)
Out[5]:  2.718281828459045

In [6]:  np.exp(0.01)
Out[6]:  1.010050167084168
```

- The linear approximation of e^1 is 2, which is **inaccurate**, and the linear approximation of $e^{0.01}$ is 1.01, which is **very good**.
- This example illustrates how the linear approximation becomes **close** to the functions **close** to the point around which the **approximation** is taken.

Discussion on Errors

- We will discuss errors on the following three concepts:

1. **Truncation** errors for Taylor series
2. **Estimate truncation** errors
3. **Round-off** errors for Taylor series

Truncation errors for Taylor series

- When we are doing numerical analysis, there are usually **two sources** of error, **round-off** and **truncation** **error**.
- The **round-off** errors are due to the **inexactness** in the representation of **real numbers** on a computer and the **arithmetic** **operations** done with them.
- While the **truncation** errors are due to the **approximate nature** of the method used, they are usually from using an approximation in place of an exact Mathematical procedure, such as that we use the **Taylor series** to approximate a function.
- For example, we can use Taylor series to approximate the function e^x:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots.$$

Truncation errors for Taylor series

- Since it takes the **infinite sequence** to approximate the function, if we only take a few items, we will have a **truncation error**.
- For example, if we only use the first 4 items to approximate e^2, which will be:

$$e^2 \approx 1 + 2 + \frac{2^2}{2!} + \frac{2^3}{3!} = 6.3333$$

- We see there is an **error** associated with it, since we **truncate** the rest of the terms in the Taylor series.
- Therefore the function $f(x)$ can be written as the **Taylor series approximation** plus a **truncation error** term:

$$f(x) = f_n(x) + E_n(x)$$

- With **more terms** we use, the approximation will be **closer** to the **exact** value.

Truncation errors for Taylor series

- **Example**: Approximate e^2 using different order of Taylor series, and print out the results.

In [7]:
```
exp = 0
x = 2
for i in range(10):
    exp = exp + \
      ((x**i)/np.math.factorial(i))
    print(f'Using {i}-term, {exp}')

print(f'The true e^2 is: \n{np.exp(2)}')
```

```
Using 0-term, 1.0
Using 1-term, 3.0
Using 2-term, 5.0
Using 3-term, 6.333333333333333
Using 4-term, 7.0
Using 5-term, 7.266666666666667
Using 6-term, 7.355555555555555
Using 7-term, 7.380952380952381
Using 8-term, 7.387301587301587
Using 9-term, 7.3887125220458545
The true e^2 is:
7.38905609893065
```

Estimate truncation errors

- We can see that the **higher order** we use to approximate the function at the value, the **closer** we are to the **true value**.
- For each order we choose, there is an **error** associated with it, and the approximation is only **useful** if we have an idea of how **accurate** the **approximation** is.
- This is the motivation why we need to understand more about the errors.
- From the Taylor series, if we use only the first n terms, we can see:

$$f(x) = f_n(x) + E_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(a)(x-a)^k}{k!} + E_n(x)$$

- The $E_n(x)$ is the **remainder** of the Taylor series, or the **truncation error** that measures how **far off** the approximation $f_n(x)$ is from $f(x)$. We can estimate the error using the **Taylor Remainder Estimation Theorem**, which is discussed next.

Taylor Remainder Estimation Theorem

- If the function $f(x)$ has $n+1$ **derivatives** for all x in an interval I containing a, then, for each x in I, there exists z between x and a such that

$$E_n(x) = \frac{f^{(n+1)}(z)(x-a)^{(n+1)}}{(n+1)!}$$

- In many times, if we know M is the **maximum** value of $\left|f^{(n+1)}\right|$ in the interval, we will have:

$$|E_n(x)| \leq \frac{M|x-a|^{(n+1)}}{(n+1)!}$$

- Therefore, we can get a **bound** for the **truncation error** using this theorem.

- **Example**: Estimate the remainder bound for the approximation using Taylor series for e^2 using $n = 9$.
- Let's work on the error when we use $n = 9$. We know that $(e^x)' = e^x$, and $a = 0$. Therefore, the error related to $x = 2$ is:

$$E_n(x) = \frac{f^{(9+1)}(z)(x)^{(9+1)}}{(9+1)!} = \frac{e^z 2^{10}}{10!}$$

- Recall that $0 \le z \le 2$, and $e < 3$, we will have

$$|E_n(x)| \le \frac{3^2 2^{10}}{10!} = 0.00254$$

- Therefore, if we use Taylor series with $n = 9$ to approximate e^2, our **absolute error** should be less than 0.00254. Let's also verify it below.

```
In [8]:  abs(7.3887125220458545-np.exp(2))

Out[8]: 0.0003435768847959153
```

Round-off errors for Taylor series

- Numerically, to add many terms in a sum, we should be mindful of **numerical accumulation of errors** that is due to **floating point round-off** errors.
- **EXAMPLE**: Approximate e^{-30} using different order of Taylor series, and print out the results.

In [13]:

```python
exp = 0
x = -30
for i in range(200):
    exp = exp + \
        ((x**i)/np.math.factorial(i))

print(f'Using {i}-term, our result is {exp}')
print(f'The true e^-30 is: {np.exp(x)}')
```

```
Using 199-term, our result is -8.553016433669241e-05
The true e^-30 is: 9.357622968840175e-14
```

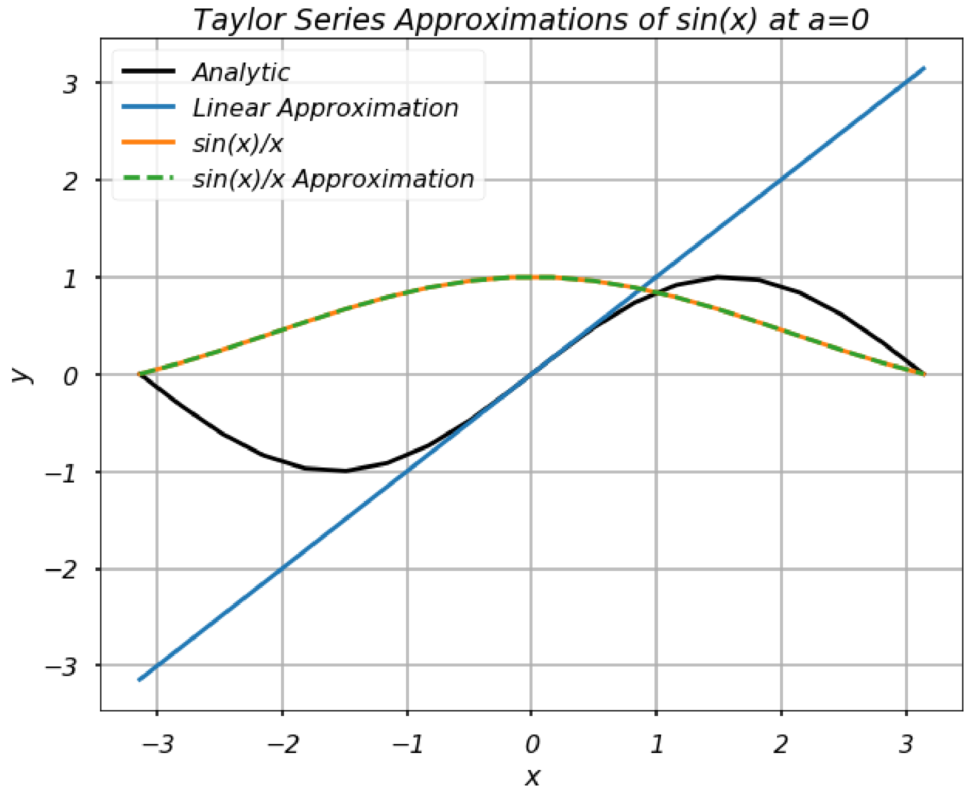
Round-off errors for Taylor series

- From the previous example, it is clear that our estimation using Taylor series are **not close** to the true value anymore, no matter how many terms you include into the calculation.
- This is due to the **round-off** errors we discussed before.
- When using **negative large arguments**, in order to get a **small result**, the Taylor series needs **alternating** large numbers to **cancel** to achieve that.
- We need many digits of **precision** in the series to capture both the **large** and the **small numbers** with enough remaining digits to get the result in the **desired output precision**.
- Thus we have this error in the previous example.

Practice

1. Use the **linear approximation** of $\sin(x)$ around $a = 0$ to show that $\frac{\sin(x)}{x} \approx 1$ for $x \approx 0$.

```
For -3.141592653589793,     3.8981718325193755e-17
For -2.8108986900540254,     0.1155144688613601
For -2.4802047265182576,     0.24764597298058835
For -2.14951076298249,     0.38946838167999825
For -1.8188167994467224,     0.5329840070941826
For -1.4881228359109546,     0.6696923593653548
For -1.157428872375187,     0.7912134805966758
For -0.8267349088394194,     0.8899151382224201
For -0.4960409453036516,     0.9594921498783174
For -0.16534698176788387,     0.9954496206758333
For 0.16534698176788387,     0.9954496206758333
For 0.4960409453036516,     0.9594921498783174
For 0.8267349088394189,     0.8899151382224201
For 1.1574288723751867,     0.791213480596676
For 1.4881228359109544,     0.6696923593653549
For 1.8188167994467221,     0.5329840070941827
For 2.14951076298249,     0.38946838167999825
For 2.4802047265182576,     0.24764597298058835
For 2.8108986900540254,     0.1155144688613601
For 3.141592653589793,     3.8981718325193755e-17
```



Next Week's Outline

- Root Finding Problem Statement
- Tolerance
- Bisection Method
- Newton-Raphson Method
- Root Finding 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.