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

IF420 – ANALISIS NUMERIK

Pertemuan ke 9 – Root Finding

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 9: Mahasiswa mampu memahami dan menerapkan teknik mencari akar – C3

Reviews

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

Outlines

- Root Finding Problem Statement
- Tolerance
- Bisection Method
- Newton-Raphson Method
- Root Finding in Python

Motivation

- As the name suggests, the **roots** of a function are one of its most important **properties**.
- **Finding** the **roots** of functions is important in many engineering applications, such as signal processing and optimization.
- For simple functions such as $f(x) = ax^2 + bx + c$, you may already be familiar with the "**quadratic formula**,"

$$x_r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

  which gives x_r, the **two roots** of f exactly.

- However for more **complicated** functions, the roots can **rarely** be **computed** using such **explicit**, or **exact**, means.

Root Finding Problem Statement

- The **root** or **zero** of a function, $f(x)$, is an x_r such that $f(x_r) = 0$.
- For functions such as $f(x) = x^2 - 9$, the roots are clearly 3 and −3.
- However, for other functions such as $f(x) = \cos(x) - x$, determining an **analytic**, or **exact**, **solution** for the **roots** of functions can be **difficult**.
- For these cases, it is useful to generate **numerical approximations** of the **roots** of f and understand the **limitations** in doing so.

Root Finding Problem Statement

- **Example**: Using **fsolve** function from **scipy** to compute the root of $f(x) = \cos(x) - x$ near −2 (initial guess). Verify that the solution is a root (or close enough).

```python
import numpy as np
from scipy import optimize

f = lambda x: np.cos(x) - x
r = optimize.fsolve(f, -2)
print("r =", r)

# Verify the solution is a root
result = f(r)
print("result=", result)
```

```
r = [0.73908513]
result= [0.]
```

Root Finding Problem Statement

- **Example**: The function $f(x) = \frac{1}{x}$ has no root. Use the **fsolve** function to try to compute the root of $f(x) = \frac{1}{x}$. Turn on the **full_output** to see what's going on. Remember to check the function's documentation for details.

```python
f = lambda x: 1/x

r, infodict, ier, mesg = optimize.fsolve(f, -2, full_output=True)
print("r =", r)

result = f(r)
print("result=", result)

print(mesg)
```

```
r = [-3.52047359e+83]
result= [-2.84052692e-84]
The number of calls to function has reached maxfev = 400.
```

- We can see that, the value r we got is **not a root**, even though the $f(r)$ is a **very small number**.
- Since we turned on the full_output, which have more information. A **message** will be returned if **no solution** is found.

Tolerance

- In engineering and science, **error** is a **deviation** from an **expected** and **computed** value.
- **Tolerance** is the **level of error** that is **acceptable** for an engineering application.
- We say that a computer **program** has **converged** to a solution when it has found a solution with an **error smaller** than the **tolerance**.
- When computing **roots numerically**, or conducting any other kind of **numerical analysis**, it is important to establish both a **metric** for **error** and a **tolerance** that is **suitable** for a given engineering/science application.

Tolerance

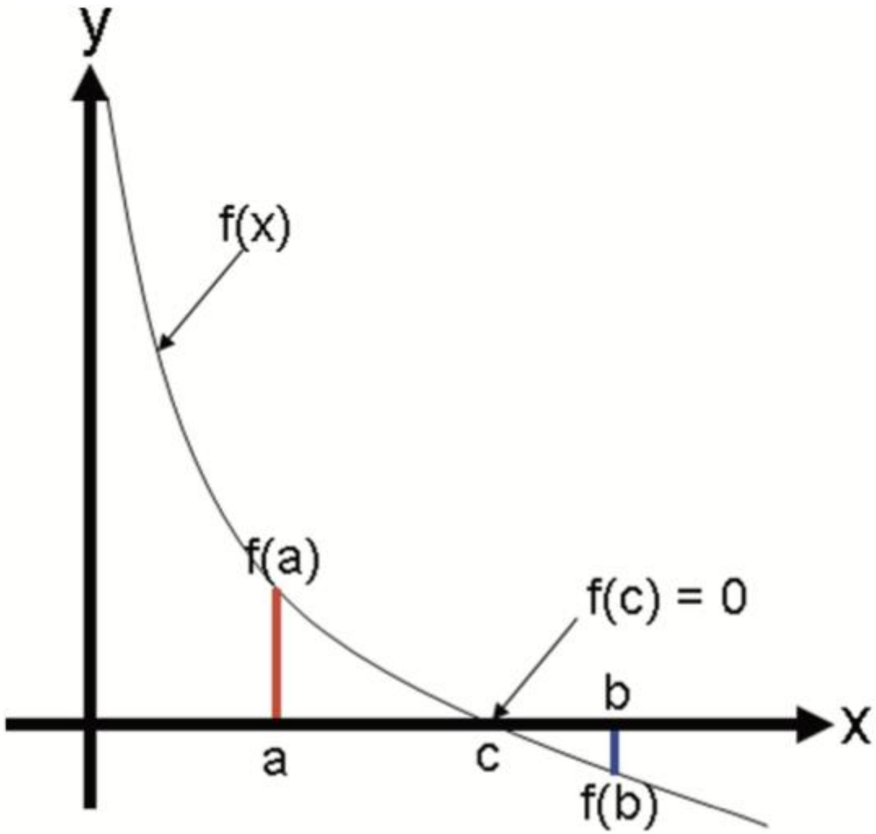
- For computing **roots**, we want an x_r such that $f(x_r)$ is **very close** to **0**.
- Therefore $|f(x)|$ is a possible choice for the **measure of error** since the **smaller** it is, the **likelier** we are to a **root**.
- Also if we assume that x_i is the i-th **guess** of an algorithm for finding a root, then $|x_{i+1} - x_i|$ is **another** possible **choice** for **measuring error**, since we expect the improvements between **subsequent guesses** to **diminish** as it **approaches** a **solution**.
- As will be demonstrated in the following examples, these different choices have their **advantages** and **disadvantages**.

Tolerance

- **Example**: Let **error** be measured by $e = |f(x)|$ and **tol** be the acceptable level of error. The function $f(x) = x^2 + tol/2$ has **no real roots**. However, $|f(0)| = tol/2$ and is therefore **acceptable as a solution** for a root finding program.
- **Example**: Let **error** be measured by $e = |x_{i+1} - x_i|$ and **tol** be the acceptable level of error. The function $f(x) = 1/x$ has **no real roots**, but the guesses $x_i = -tol/4$ and $x_{i+1} = tol/4$ have an error of $e = tol/2$ and is an **acceptable solution** for a computer program.
- Based on these observations, the use of **tolerance** and **converging criteria** must be done very **carefully** and in the context of the program that uses them.

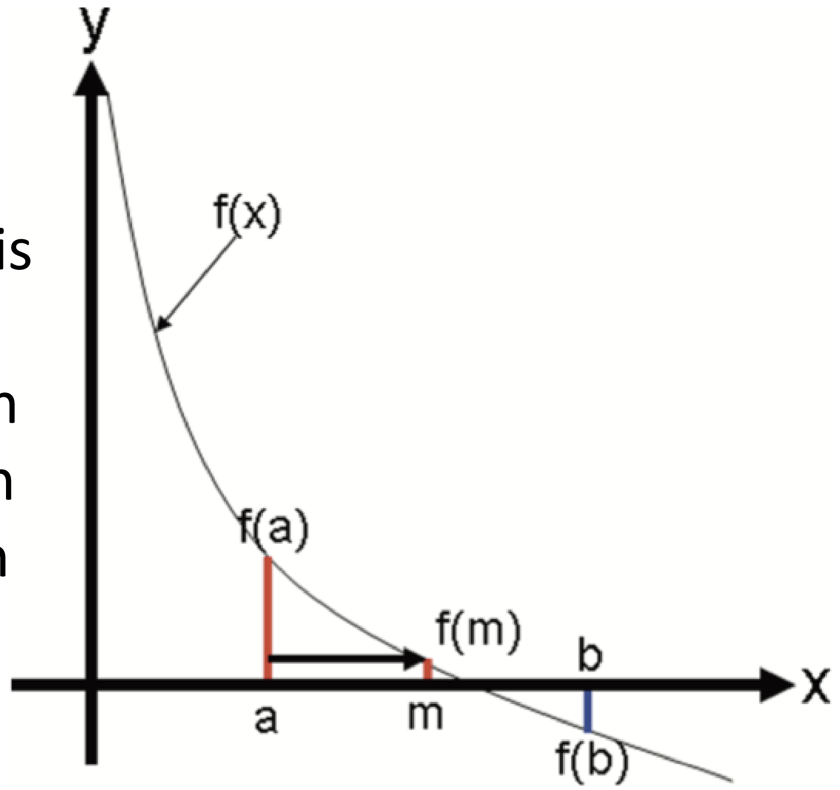
Bisection Method

- The **Intermediate Value Theorem** says that if $f(x)$ is a **continuous function** between a and b, and $sign(f(a)) \neq sign(f(b))$, then there must be a c, such that $a < c < b$ and $f(c) = 0$.
- This is illustrated in the following figure.



https://pythonnumericalmethods.berkeley.edu/notebooks/chapter19.03-Bisection-Method.html

- The **bisection** method uses the **intermediate value theorem iteratively** to find roots.
- Let $f(x)$ be a **continuous function**, and a and b be **real scalar values** such that $a < b$. Assume, without **loss of generality**, that $f(a) > 0$ and $f(b) < 0$. Then by the **intermediate value theorem**, there must be a **root** on the open interval (a, b). Now let $m = \frac{b+a}{2}$, the **midpoint** between a and b. If $f(m) = 0$ or is **close enough**, then m is a root. If $f(m) > 0$, then m is an **improvement** on the **left bound**, a, and there is guaranteed to be a root on the open interval (m, b). If $f(m) < 0$, then m is an **improvement** on the **right bound**, b, and there is guaranteed to be a root on the open interval (a, m). This scenario is depicted in the following figure.



- The process of **updating** a and b can be repeated until the error is acceptably low.

Bisection Method

- **Example**: Program a function **my_bisection(f, a, b, tol)** that approximates a root r of f, bounded by a and b to within $|f(\frac{a+b}{2})| < tol$.

```python
import numpy as np

def my_bisection(f, a, b, tol):
    # approximates a root, R, of f bounded
    # by a and b to within tolerance
    # | f(m) | < tol with m the midpoint
    # between a and b. Recursive implementation

    # check if a and b bound a root
    if np.sign(f(a)) == np.sign(f(b)):
        raise Exception(
         "The scalars a and b do not bound a root")

    # get midpoint
    m = (a + b)/2

    if np.abs(f(m)) < tol:
        # stopping condition, report m as root
        return m
    elif np.sign(f(a)) == np.sign(f(m)):
        # case where m is an improvement on a.
        # Make recursive call with a = m
        return my_bisection(f, m, b, tol)
    elif np.sign(f(b)) == np.sign(f(m)):
        # case where m is an improvement on b.
        # Make recursive call with b = m
        return my_bisection(f, a, m, tol)
```

Bisection Method

- **Example**: The $\sqrt{2}$ can be computed as the root of the function $f(x) = x^2 - 2$. Starting at $a = 0$ and $b = 2$, use **my_bisection** to approximate the $\sqrt{2}$ to a tolerance of $|f(x)| < 0.1$ and $|f(x)| < 0.01$. Verify that the results are close to a root by plugging the root back into the function.

```
f = lambda x: x**2 - 2

r1 = my_bisection(f, 0, 2, 0.1)
print("r1 =", r1)
r01 = my_bisection(f, 0, 2, 0.01)
print("r01 =", r01)

print("f(r1) =", f(r1))
print("f(r01) =", f(r01))
```

```
r1 = 1.4375
r01 = 1.4140625
f(r1) = 0.06640625
f(r01) = -0.00042724609375
```

Bisection Method

- **Example:** See what will happen if you use $a = 2$ and $b = 4$ for the above function.

```
my_bisection(f, 2, 4, 0.01)
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-19-4158b7a9ae67> in <module>
----> 1 my_bisection(f, 2, 4, 0.01)

<ipython-input-17-a4164672a36c> in my_bisection(f, a, b, tol)
     10     if np.sign(f(a)) == np.sign(f(b)):
     11         raise Exception(
---> 12          "The scalars a and b do not bound a root")
     13 
     14     # get midpoint

Exception: The scalars a and b do not bound a root
```

Newton-Raphson Method

- Let $f(x)$ be a **smooth** and **continuous** function and x_r be an **unknown root** of $f(x)$. Now assume that x_0 is a **guess** for x_r. Unless x_0 is a very lucky guess, $f(x_0)$ will not be a root. Given this scenario, we want to find an x_1 that is an **improvement** on x_0 (i.e., closer to x_r than x_0). If we assume that x_0 is "close enough" to x_r, then we can improve upon it by taking the **linear approximation** of $f(x)$ around x_0, which is a line, and finding the **intersection** of this line with the x-axis.

- Written out, the **linear approximation** of $f(x)$ around x_0 is $f(x) \approx f(x_0) + f'(x_0)(x - x_0)$. Using this approximation, we find x_1 such that $f(x_1) = 0$. Plugging these values into the **linear approximation** results in the equation

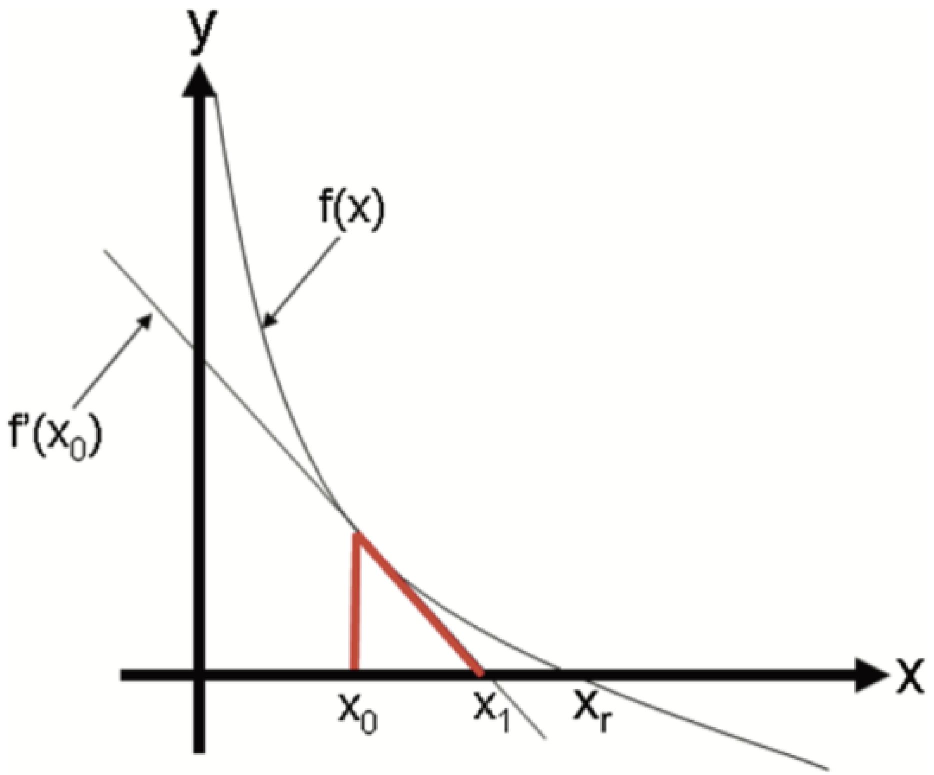
$$0 = f(x_0) + f'(x_0)(x_1 - x_0),$$

which when solved for x_1 is

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Newton-Raphson Method

- An illustration of how this **linear approximation** **improves** an **initial guess** is shown in the following figure.



- Written **generally**, a **Newton** step computes an **improved guess**, x_i, using a previous guess x_{i-1}, and is given by the equation

$$x_i = x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})}.$$

- The **Newton-Raphson Method** of finding roots **iterates** **Newton steps** from x_0 until the error is **less** than the **tolerance**.

Newton-Raphson Method

- **Example**: Again, the $\sqrt{2}$ is the root of the function $f(x) = x^2 - 2$. Using $x_0 = 1.4$ as a starting point, use the previous equation to estimate $\sqrt{2}$. Compare this approximation with the value computed by Python's **sqrt** function.

$$x = 1.4 - \frac{1.4^2 - 2}{2(1.4)} = 1.4142857142857144$$

```
import numpy as np

f = lambda x: x**2 - 2
f_prime = lambda x: 2*x
newton_raphson = 1.4 - (f(1.4))/(f_prime(1.4))

print("newton_raphson =", newton_raphson)
print("sqrt(2) =", np.sqrt(2))
```

```
newton_raphson = 1.4142857142857144
sqrt(2) = 1.4142135623730951
```

Newton-Raphson Method

- **Example**: Write a function **my_newton(f,df,x0,tol)**, where the output is an estimation of the root of f, **f** is a function object $f(x)$, **df** is a function object to $f'(x)$, **x0** is an initial guess, and **tol** is the error tolerance. The error measurement should be $|f(x)|$.

```python
def my_newton(f, df, x0, tol):
    # output is an estimation of the root of f
    # using the Newton Raphson method
    # recursive implementation
    if abs(f(x0)) < tol:
        return x0
    else:
        return my_newton(f, df, x0 - f(x0)/df(x0), tol)
```

- **Example**: Use **my_newton** to compute $\sqrt{2}$ to within tolerance of 1e-6 starting at x0 = 1.5.

```python
estimate = my_newton(f, f_prime, 1.5, 1e-6)
print("estimate =", estimate)
print("sqrt(2) =", np.sqrt(2))
```

```
estimate = 1.4142135623746899
sqrt(2) = 1.4142135623730951
```

Newton-Raphson Method

- If x_0 is **close** to x_r, then it can be proven that, in general, the **Newton-Raphson** method **converges** to x_r much **faster** than the **bisection** method.
- However since x_r is initially **unknown**, there is no way to know if the initial guess is **close enough** to the root to get this behavior unless some **special information** about the function is known a priori (e.g., the function has a root close to $x = 0$).
- In addition to this **initialization problem**, the Newton-Raphson method has other serious **limitations**.
- For example, if the **derivative** at a **guess** is **close** to 0, then the Newton step will be very **large** and probably lead **far away** from the **root**.
- Also, depending on the behavior of the function **derivative** between x_0 and x_r, the Newton-Raphson method may **converge** to a **different root** than x_r that may not be useful for our engineering application.

Newton-Raphson Method

- **Example**: Compute a **single** Newton step to get an improved approximation of the root of the function $f(x) = x^3 + 3x^2 - 2x - 5$ and an initial guess, $x_0 = 0.29$.

```
x0 = 0.29
x1 = x0-(x0**3+3*x0**2-2*x0-5)/(3*x0**2+6*x0-2)
print("x1 =", x1)
```

```
x1 = -688.4516883116648
```

- Note that $f'(x_0) = -0.0077$ (**close** to 0) and the error at x_1 is approximately -324880000 (**very large**).

- **Example**: Consider the polynomial $f(x) = x^3 - 100x^2 - x + 100$. This polynomial has a root at $x = 1$ and $x = 100$. Use the Newton-Raphson to find a root of f starting at $x_0 = 0$.

- At $x_0 = 0$, $f(x_0) = 100$, and $f'(x_0) = -1$. A Newton step gives $x_1 = 0 - \frac{100}{-1} = 100$, which is a root of f. However, note that this root is much farther from the initial guess than the other root at $x = 1$, and it **may not be the root** you wanted from an **initial guess** of 0.

Root Finding in Python

- As you may think, Python has the existing **root-finding** functions for us to use to make things easy.
- The function we will use to find the root is **fsolve** from the **scipy.optimize**.
- The **fsolve** function takes in many **arguments** that you can find in the documentation, but the most **important two** is the **function** you want to find the root, and the **initial guess**.
- **Example**: Compute the root of the function $f(x) = x^3 - 100x^2 - x + 100$ using **fsolve()**.

```
f = lambda x: x**3-100*x**2-x+100

fsolve(f, [2, 80])
```

```
]: array([  1., 100.])
```

Practice

1. Write a function **my_nth_root(x,n,tol)**, where x and tol are strictly positive scalars, and n is an integer strictly greater than 1. The output argument, **r**, should be an approximation $r = \sqrt[N]{x}$, the N-th root of x. This approximation should be computed by using the **Newton-Raphson** method to find the root of the function $f(y) = y^N - x$. The error metric should be $|f(y)|$.

```
# Test case
estimate = my_nth_root(2, 2, 1e-6)
print("estimate =", estimate)
print("sqrt(2) =", np.sqrt(2))
```

```
estimate = 1.4142135623746899
sqrt(2) = 1.4142135623730951
```

```
# Another Test case
estimate = my_nth_root(3, 2, 1e-6)
print("estimate =", estimate)
print("sqrt(3) =", np.sqrt(3))
```

```
estimate = 1.7320508100147276
sqrt(3) = 1.7320508075688772
```

Next Week's Outline

- Numerical Differentiation Problem Statement
- Finite Difference Approximating Derivatives
- Approximation of Higher Order Derivatives
- Numerical Differentiation with Noise

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.