Problem Set # 02
Computer Science 117 - S

Aaron Paul D. Gapud
adgapud@up.edu.ph

Elaine T. Pajarillo
etpajarillo@up.edu.ph

Junius Wilhelm G. Bueno
University of the Philippines - Baguio
Gov. Pack Road, Baguio City, Baguio 2600

October 9, 2023

1. Given the following graph of a polynomial function, find a root of the function by applying the given method and identified inputs. Reason out your answers.

- **Bisection Method on the interval** $[-2, 2]$

  Using the Bisection Method to find a root of the function on [-6, 2] we determined that one root of the function is **x = -4.9.** Applying the Bisection Method, we performed the following steps:

$$a_1 = -6,\, b_1 = 2,\, c = -2,\, since f(c) > 0 \therefore b = c$$

$$a_1 = -6,\, b_2 = -2,\, c = -4,\, since f(c) > 0 \therefore b = c$$

$$a_1 = -6,\, b_3 = -4,\, c = -5,\, since f(c) < 0 \therefore a = c$$

$$a_2 = -5,\, b_3 = -4,\, c = -4.5,\, since f(c) > 0 \therefore b = c$$

$$a_2 = -5,\, b_4 = -4.5,\, c = -4.75,\, since f(c) > 0 \therefore b = c$$

$$a_2 = -5,\, b_5 = -4.75,\, c = -4.875,\, since f(c) > 0 \therefore b = c$$

  Since the interval is small enough after the last iteration of the Bisection Method, we can stop the process and conclude that **x = -4.9** is an estimated root of the function.

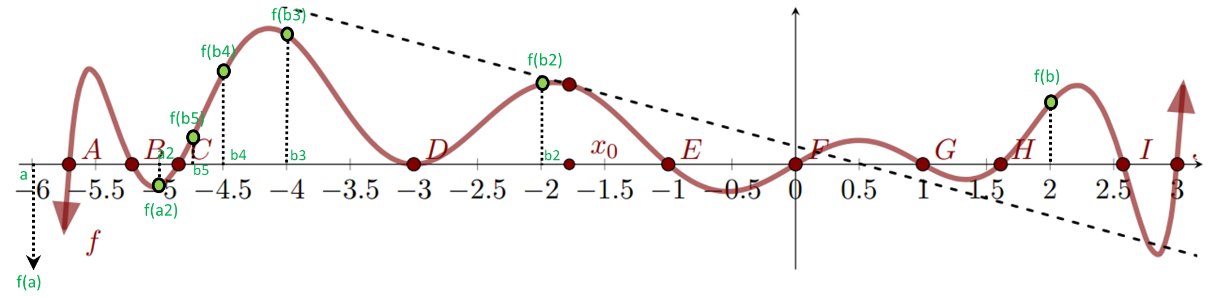

Figure 1: Implementation of the Bisection Method on the Graph

- **Newton Method, with initial iterate** x_0 **as shown in the figure.**

  Using the Newton method with the initial iterate x_0 and its corresponding function value given by the figure, we conclude that a root of the function cannot be determined using this specific method with the initial given iterate. Performing the process of getting the root using the Newton method, we started by getting the tangent line of the function value of x_0 and extending it until it intersects the x-axis, with $x = 0.5$. Next, we got the function value of $x = 0.5$ and its tangent line, however, this time, the tangent line derived is a horizontal tangent line that will never intersect the x-axis, causing the Newton Method to fail in getting the root.

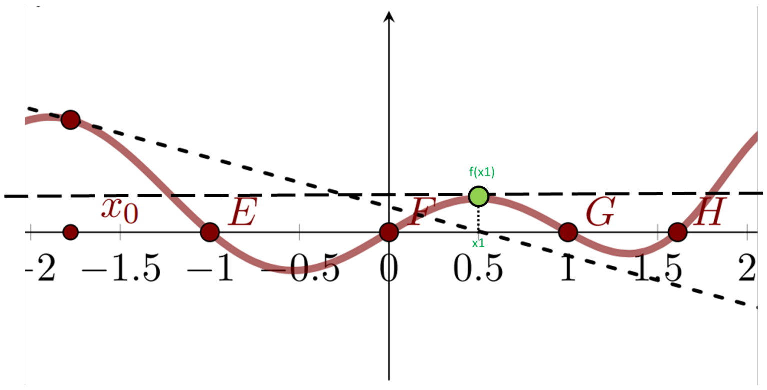

Figure 2: Implementation of the Newton Method on the Graph

2. (5 points) Use the Newton–Raphson method to approximate the value of $\sqrt{2}\sqrt[3]{3}\sqrt[4]{4}$ correct to 9 decimal places. Use the initial iterate $x_0 = 3$. How many iterations do we need to achieve the approximate value?

In order to use the Newton–Raphson method to approximate the value of $\sqrt{2}\sqrt[3]{3}\sqrt[4]{4}$ to 9 decimal places, we must first derive the function $f(x)$, we can do this by equating $\sqrt{2}\sqrt[3]{3}\sqrt[4]{4}$ to x and manipulating it to derive the function $f(x)$ and use it to get the derivative.

$$x = \sqrt{2}\sqrt[3]{3}\sqrt[4]{4}$$

$$x^2 = 4\sqrt[3]{3^2}$$

$$(x^2)^3 = (4\sqrt[3]{3^2})^3$$

$$x^6 = 576$$

$$x^6 - 576 = 0$$

$$f(x) = x^6 - 576\, and\, f'(x) = 6x^5\, with\, x_0 = 3$$

Now that we have the required values to implement the Newton-Raphson Method, let us jump into the python code that we used to get the approximated roots of the function.

```
from rootscalar import newton_raphson, printf

f = lambda x : (x**6 - 576)
df = lambda x: (6*x**5)

exact = 2**(1/2) * 3**(1/3) * 4**(1/4)
approx = newton_raphson(f, df, 3, 10, 1e-9, 0.5, 0.5)

printf("Real Value", exact, 9)
printf("Approximate Value", approx[0], 9)
printf("Iterations", approx[1], 0)
printf("Error", approx[2], 9)
```

Here, a user-defined `printf()` function was imported from `rootscalar.py` for uniform printing of the output of the code, as well as the `newton_raphson` method for approximating the root of the function. Note that the function takes in the parameters as `newton_raphson(f, qk, x, maxit, tol, wa, wf)`, and hence the following values was used: $x = 3$, $maxit = 10$, $tol = 1e-9$, $wa = 0.5$, and $wf = 0.5$. The computed $f(x)$ and $f'(x)$ above were assigned to the lambda functions `f` and `df`, respectively. We also used the original value to be approximated and saved in `exact` variable in order to have a semblance for comparison. After this, we just called the function and saved it to `approx` and then finally printed the values.

This particular code produces the output:

```
       Real Value     2.884499141e+00
Approximate Value     2.884499141e+00
       Iterations     5e+00
            Error     1.139088823e-13
```

Figure 3: Approximation of $\sqrt{2}\sqrt[3]{3}\sqrt[4]{4}$ using Newton-Raphson method.

Here, the `Real Value` outputs the approximated value of the $\sqrt{2}\sqrt[3]{3}\sqrt[4]{4}$ with 9 decimal points using the normal approximation method. While the `Approximate Value` is the approximated value outputted using the `newton_raphson` method. The `Iterations` outputs the number of iterations it took for the `newton_raphson` method to arrive at the approximated root and the `Error` outputs the error that the process incurred.

3. (10 points) Use the Newton–Horner Method with refinement in your rootpoly.py to find all the roots of the polynomial $p_{n-1}(x) = \sum_{k=1}^{n}(k^k (mod\, 7))x^{k-1}$ for $n = 20, 40, 50$. Use $z = 1 + j$, maxit = refmax = 100, tol = 10^3 eps, and reftol = 10^-3. Plot the roots you obtain on each n. Include the plot and maximum function value of approximate roots in your documentation.

This problem requires the use of the Newton-Horner Method with refinement in the rootpoly in order to find all the roots of the given polynomial. Here is our code for this item:

```
from rootpoly import *
import numpy as np
import matplotlib.pyplot as plt

def max_function_value(p, roots):
    max_value = np.linalg.norm([abs(horner(p, root)[0]) for root in roots], np.inf)
    return max_value

for n in [20, 40, 50]:

  # Generate the polynomial for n
  p = []
  for k in range(1, n+1):
    p.append(k**k % 7)

  # Find the roots using the Newton-Horner method
  eps = np.finfo(float).eps
  roots = newtonhorner(p, complex(1,1), 100, 1e3*eps, 100, 1e-3, True)[0]

  # Generate plots
  plt.style.use('seaborn-v0_8-whitegrid')
  plt.figure()
  plt.title(f"Roots for n = {n}")
  plt.plot(roots.real, roots.imag, ".")
  plt.show()

  # Calculate for the Maximum Function Value
  max_func_val = max_function_value(p, roots)
  print(f"Maximum function value of approximate roots for n = {n}: {max_func_val:.10e}")
```

In this code, we defined a number of functions along with snippets of code that function in specific ways, this section will explain and elaborate on the different codes and their specific function. An array such as `p = []` was declared, to be appended with elements necessary for the variables to be used for the user-defined functions. The roots were sought for using the `newtonhorner()` method imported from `rootpoly.py`. The maximum function values were solved using the `horner()` method from the same import and normalized using `numpy.linalg.norm`, yielding the results as shown in Table 1.

n	Maximum Function Value
20	9.4888460561e-04
40	2.7337262784e-12
50	2.4725421266e-04

Table 1: Maximum function values of the acquired roots of $p_{n-1}(x) = \sum_{k=1}^{n}(k^k (mod\, 7))x^{k-1}$.

The obtained outputs showed that, as the value of `n` increases, the degree of the polynomial also increases, leading to a more complex polynomial and a decrease in its corresponding maximum function value. For smaller values of `n`, the polynomial is relatively simpler, and the maximum function value is relatively larger. This suggests that the polynomial has roots that are closer to zero, and their magnitudes are relatively small. As `n` increases, the polynomial becomes more complex with higher-degree terms. The maximum function values for these cases are much smaller, this indicates that the polynomial has roots with larger magnitudes or that the polynomial approaches zero more rapidly as `n` increases.

The acquired roots were plotted using the library `matplotlib.pyplot`, generated in the following Figure 4.

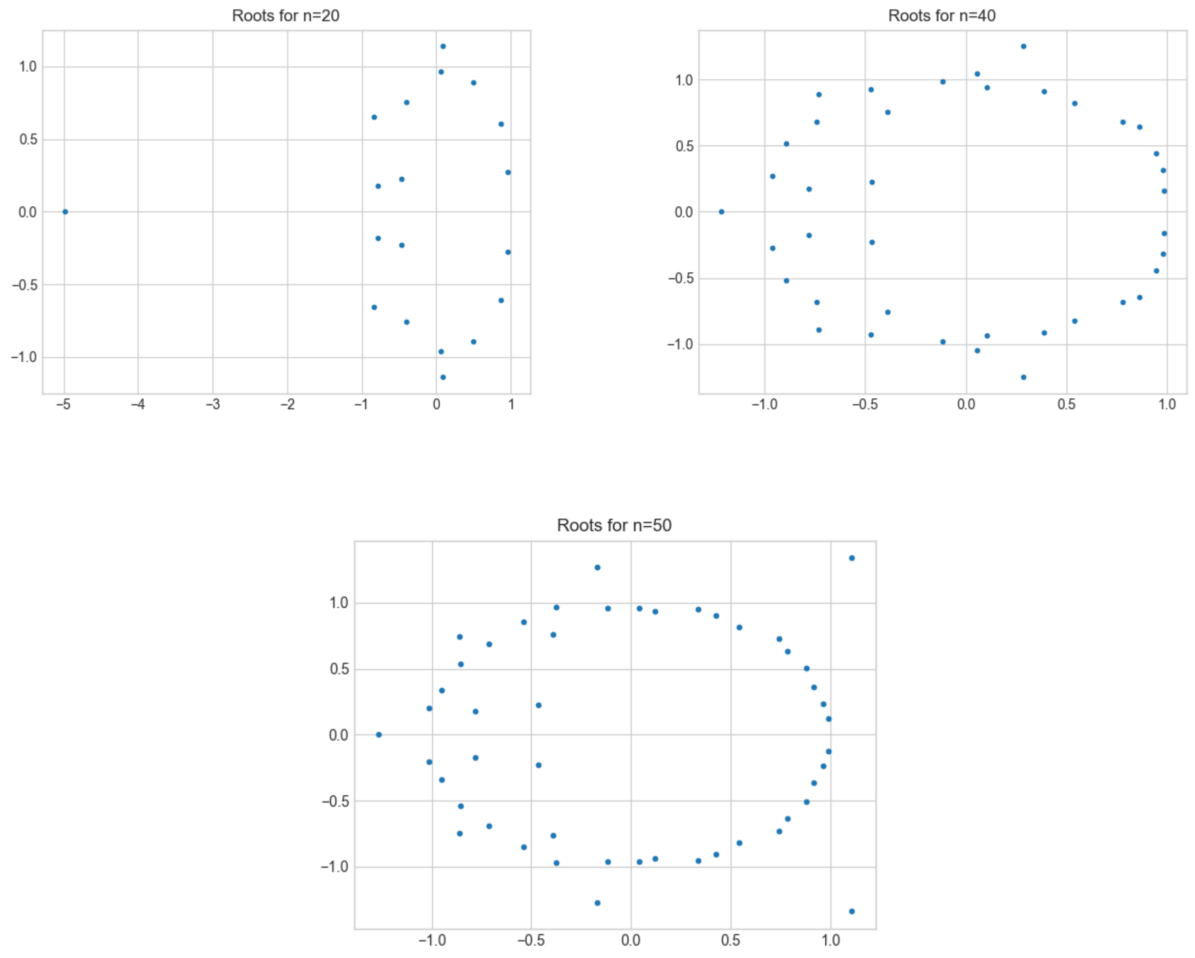

Figure 4: The acquired roots of $p_{n-1}(x) = \sum_{k=1}^{n}(k^k (mod\, 7))x^{k-1}$ for different n.

4. (15 points) Modify rootscalar.py to find an approximate solution of $e^{\cos x} - 3\sin(x) = 0$ on the interval $[-2, 2]$ using Inexact Newton-Raphson Method (forward, backward, central finite difference), Steffensen Method, and Picard Fix Point Method with tol $= 10^{-10}$, maxit $= 1000$, wa $= 0.5$, and wf $= 0.5$. Use your best judgement for the other inputs e.g. initial iterates.

For a particular method, i.e., Picard Fix Point, we first needed to find a function $g(x)$ derived from the given $f(x) = e^{\cos x} - 3\sin(x)$. We manipulate this as follows:

$$e^{\cos x} - 3\sin(x) = 0$$
$$\Rightarrow \quad e^{\cos x} = 3\sin(x)$$
$$\Rightarrow \quad \frac{e^{\cos x}}{3} = \sin(x)$$
$$\Rightarrow \sin^{-1}\left(\frac{e^{\cos x}}{3}\right) = x$$

Thus, we use this as our $g(x)$, assigned to the variable `g`. Also note that our `df` is set as the derivative of $f(x)$.

$$f'(x) = \frac{d}{dx}(e^{\cos x} - 3\sin x) = -e^{\cos x}\sin x - 3\cos x$$

The snippet below contains the code for `Item4Imp.py`:

```
from rootscalar import *

f = lambda x : (np.exp(np.cos(x)) - 3*(np.sin(x)))
g = lambda x : np.arcsin(np.exp(np.cos(x)) / 3)
df = lambda x : -np.exp(np.cos(x)) * np.sin(x) - 3 * np.cos(x)

results = {
  "Newton-Raphson": newton_raphson(f, df, 0, 1000, 1e-10, 0.5, 0.5),
  "(Forward) Newton-Raphson": newton_raphson_inexact(f, 1, 0, 1000, 1e-10, 0.5, 0.5),
  "(Backward) Newton-Raphson": newton_raphson_inexact(f, 2, 0, 1000, 1e-10, 0.5, 0.5),
  "(Central) Newton-Raphson": newton_raphson_inexact(f, 3, 0, 1000, 1e-10, 0.5, 0.5),
  "Steffensen": steffensen(f, 0, 1000, 1e-10, 0.5, 0.5),
  "Picard Fix Point": fixpoint(g, 0, 1000, 1e-10),
}

header = {
  "Method" : 27,
  "Root x" : 28,
  "Iteration k" : 14,
  "Error" : 25,
  "Function Value f(x)" : 22,
}

for string, length in header.items():
    printff(string, length)
print("")

for method, result in results.items():
  print(f"{method:<27} {result[0]:<+28.15e} {result[1]:<14} {result[2]:<+25.15e}
    {f(result[0]):<+22.15e}")
```

The code uses Python dictionaries to store data in key-value pairs.

`results = { }`	Pairs the name of the method used with the corresponding results they return. The key is retrieved as 'string' and its value as 'length'. Used to specify properly-spaced column headers.
`header = { }`	Pairs the column header with their preferred column width. The key is retrieved as 'method' and its value as 'result'. Used to quickly access the results from a particular method.

Notice the usage of a function `printff()` for bold headers, imported from `rootscalar.py`. Refer to the files attached for the codes provided with documentation in their comments. Table 2 tallies the results obtained using different approximation methods: (1) Exact Newton-Raphson, (2) Forward Newton-Raphson, (3) Backward Newton-Raphson, (4) Central Newton-Raphson, (5) Steffensen, and (6) Picard Fix Point. Observe that most of the methods obtained the same results within 4 iterations on average. However, the Picard Fix Point method was the slowest and least accurate in approximating the solution.

Method	Root x	Iteration k	Error	Function Value f(x)
1	+7.593406576780887e-01	5	+1.046440711860441e-11	-4.440892098500626e-16
2	+7.593406576780887e-01	5	+1.043876096673557e-11	-4.440892098500626e-16
3	+7.593406576780887e-01	5	+1.047961717404178e-11	-4.440892098500626e-16
4	+7.593406576780887e-01	5	+1.047961717404178e-11	-4.440892098500626e-16
5	+7.593406576780887e-01	7	+4.718447854656915e-15	-4.440892098500626e-16
6	+7.593406577139304e-01	55	+9.069056616795024e-11	-1.289492956857430e-10

Table 2: Approximate solutions of $e^{\cos x} - 3\sin(x) = 0$ on the interval $[-2, 2]$ using various methods.