Problem Set # 04
Computer Science 117 - S

Nathaniel Aquino
nbaquino2@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
November 16, 2023

1. Suppose you want the interpolating polynomial P3(x) about the points (0, 0),(0.5, y), (1, 3), and (2, 2) to have a leading coefficient of 1. Find the value of y.

Solution: Given the points, we have $x_0 = 0,\ x_1 = 0.5,\ x_2 = 1, x_3 = 2$ and $f(x_0) = 0,$ $f(x_1) = y, f(x_2) = 3, f(x_3) = 2$. The Lagrange polynomial of order 3, is given by

$$P_3(x) = L_0(x)f(x_0) + L_1(x)f(x_1) + L_2(x)f(x_2) + L_3(x)f(x_3)$$

where $L_i(x) = \prod_{j=0,\, j\neq i}^{3}\left(\frac{x-x_j}{x_i-x_j}\right)$. Hence we have,

$$L_0(x) = \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)}, \qquad L_1(x) = \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)},$$

$$L_2(x) = \frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)}, \qquad L_3(x) = \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)}.$$

Plugging in the given x_n values we now have,

$$L_0(x) = \frac{(x-0.5)(x-1)(x-2)}{(0-0.5)(0-1)(0-2)} = \frac{x^3-\frac{7}{2}x^2+\frac{7}{2}x-1}{-1} = -x^3 + \frac{7}{2}x^2 - \frac{7}{2}x + 1$$

$$L_1(x) = \frac{(x-0)(x-1)(x-2)}{(0.5-0)(0.5-1)(0.5-2)} = \frac{x^3-3x^2+2x}{\frac{3}{8}} = \frac{8}{3}x^3 - 8x^2 + \frac{16}{3}x$$

$$L_2(x) = \frac{(x-0)(x-0.5)(x-2)}{(1-0)(1-0.5)(1-2)} = \frac{x^3-\frac{5}{2}x^2+x}{-\frac{1}{2}} = -2x^3 + 5x^2 - 2x$$

$$L_3(x) = \frac{(x-0)(x-0.5)(x-1)}{(2-0)(2-0.5)(2-1)} = \frac{x^3-\frac{3}{2}x^2+\frac{1}{2}x}{3} = \frac{1}{3}x^3 - \frac{1}{2}x^2 + \frac{1}{6}x$$

Now, plugging in the given $f(x_n)$ values and the computed $L_i(x)$ we now have,

$$\begin{aligned} P_3(x) &= L_0(x)f(x_0) + L_1(x)f(x_1) + L_2(x)f(x_2) + L_3(x)f(x_3) \\ &= L_0(x) * 0 + L_1(x) * y + L_2(x) * 3 + L_3(x) * 2 \\ &= \left(\frac{8}{3}x^3 - 8x^2 + \frac{16}{3}x\right) * y + \left(-2x^3 + 5x^2 - 2x\right) * 3 + \left(\frac{1}{3}x^3 - \frac{1}{2}x^2 + \frac{1}{6}x\right) * 2 \\ &= \left(\frac{8}{3}y - 6 + \frac{2}{3}\right)x^3 + \left(-8y + 15 - 1\right)x^2 + \left(\frac{16}{3}y - 6 + \frac{1}{3}\right)x \\ &= \left(\frac{8y-16}{3}\right)x^3 + \left(-8y + 14\right)x^2 + \left(\frac{16y-17}{3}\right)x \end{aligned}$$

Since the leading coefficient is equal to 1, we can solve for y by using,

$\frac{8y-16}{3} = 1$.

Therefore, we have **y =** $\frac{19}{8}$ or **y = 2.375.**

2. Write a function in polyinterp.py that implements the Newton form of Lagrange Polynomial. Use this to plot the interpolating polynomial pn of degree n = 5, 10, 20 of the function $f(x) = \frac{x}{\left(1-x^2+x^4\right)^2}$ **on the interval [−2, 2] (use equidistant nodes including the endpoints). Use three different Cartesian planes to compare the plot of f(x) and pn for n = 5, 10, 20. Include the approximate function norm** $||f - p||_{-2,2,\infty}$ **for n = 5, 10, 20, each with 9,000 nodes.**

```python
def NewtonLagrangeInterpolation(f,x):
  n=len(x)
  d=np.zeros((n,n), dtype=float)
  d[:, 0] = f(x)
```

```
  for k in range (1,n):
    for l in range(k,n):
      d[l,k]= (d[l,k-1]- d[l-1, k-1]) / (x[l]- x[l-k])

  def p(z):
    v= d[n-1, n-1]
    for k in range(1,n):
      v= v * (z-x [n-1-k]) + d[n-1-k, n-1-k]
    return v

  return p
```

To answer the problem, we first wrote a code that implements the Newton Form of the Lagrange Polynomial. This function takes the parameters **f** and **x**, as the function to be interpolated and the x-coordinated of the points to be used for the interpolation and returns the **p** for the interpolating polynomial function. This function initializes by computing the divided differences based on the input data and creates a smooth curve passing through the data points to interpolate values in given data points.

Implementation:

Python

```python
import numpy as np
import matplotlib.pyplot as plt
from polyinterp import NewtonLagrangeInterpolation

def f(x):
  return x / (1 - x**2 + x**4)**2

def compute_norm(f, pn, start, end, norm_type='2'):
  x_values = np.linspace(start, end, 9000)
  difference = f(x_values) - pn(x_values)
  return np.linalg.norm(difference, ord=float(norm_type)) # Convert norm_type to
float

# Plotting
start, end = -2, 2
n_values = [5, 10, 20]

# Create subplots
fig, axs = plt.subplots(1, len(n_values), figsize=(15, 5))

for i, n in enumerate(n_values):
  # Generate equidistant nodes
  nodes = np.linspace(start, end, n+1)

  # Create interpolation function using Newton-Lagrange method
  interpolation_function = NewtonLagrangeInterpolation(f, nodes)
  # Plot f(x) and pn for each n
  x_values = np.linspace(start, end, 1000)
  axs[i].plot(x_values, f(x_values), label='f(x)')
  axs[i].plot(x_values, interpolation_function(x_values), label=f'p{n}(x)')
  axs[i].set_title(f'n = {n}')
  axs[i].legend()

  # Plot the nodes
  axs[i].plot(nodes, f(nodes), "ro")

  # Add grid and labels
```

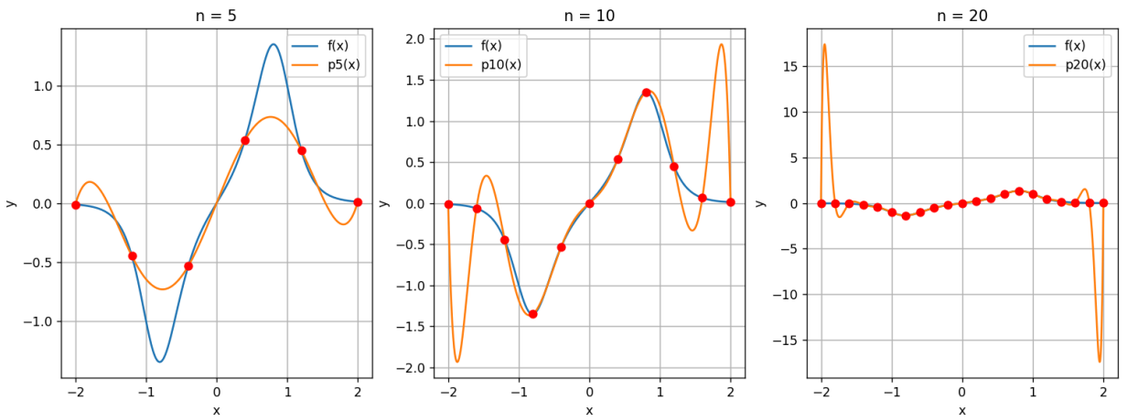
```
    axs[i].grid(True)
    axs[i].set_xlabel('x') # Fix the function name to set_xlabel
    axs[i].set_ylabel('y') # Fix the function name to set_ylabel

    # Compute and print the norms
    norm_inf = compute_norm(f, interpolation_function, start, end, np.inf)
    print(f"For n = {n}: ||f - p{n}||_∞ = {norm_inf}")

plt.show()
```

This code performs the polynomial interpolation using the Newton-Lagrange method and displays the results for different node counts. The target function, f(x), is defined from the given problem, and the NewtonLagrangeInterpolation function is imported to generate interpolating polynomials. Plots show the target function and interpolating polynomials for n= 5, 10, 20, making it easier to assess interpolation accuracy. The norms of differences between the target function and interpolating polynomials are computed and outputted, and equidistant nodes are constructed.

Results:

```
For n = 5: ||f - p5||_inf = 0.6208527860397671
For n = 10: ||f - p10||_inf = 1.9142173178303574
For n = 20: ||f - p20||_inf = 17.41142038548993
```

Approximate function norm for n = 5, 10, 20



Plot of the Interpolating Polynomial p_n of degree n = 5, 10, 20

Using the code, we computed the approximate function norm between the function and interpolating polynomial to measure the accuracy. We also outputted the plot of the function and the interpolating polynomial to provide visuals for the approximation.

3. With the same function in item number 2, minimize the supremum norm by considering three subintervals of [-2,2] and using appropriate Lagrange interpolating polynomials with degree of at most 10 at each subinterval. In your computation for function norm, use 3000 nodes on each of the considered intervals and take the maximum of the norms. Also, include the plot of the piecewise interpolating polynomial.

Implementing the same code from the polyinterp.py with the Newton Form of the Lagrange Polynomial, the supremum norm is calculated by determining the maximum absolute difference between the original function and the corresponding interpolating

polynomial within each subinterval. We first iterated through the subintervals, identifying the one with the maximum function norm. It then displayed the piecewise interpolating polynomial for the interval with the maximum norm, including the interpolation points. The results, including maximum norm values and associated subintervals, are printed as well as the plots for visualization into the accuracy of the piecewise interpolation, highlighting the subinterval where the supremum norm is maximized.

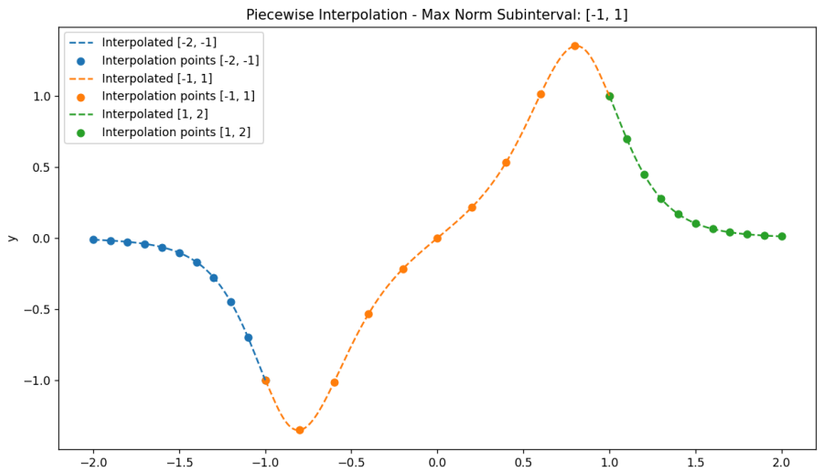
Code:

Python

```python
import numpy as np
import matplotlib.pyplot as plt
from polyinterp import *

def supnorm(f, a, b, numnodes):
  x = np.linspace(a, b, numnodes)
  absfx = np.abs(f(x))
  return max(absfx)

def f(x):
  return x / ((1 - x**2 + x**4)**2)

[a, b] = [-2, 2]

subintervals = [[-2, -1], [-1, 1], [1, 2]]

max_norm = 0
max_norm_interval = None

fig, ax = plt.subplots(figsize=(10, 6))

for interval in subintervals:
  a_i, b_i = interval
  x_i = np.linspace(a_i, b_i, 11)
  p_i = NewtonInterpolation(f, x_i)

  norm_i = supnorm(lambda x: f(x) - p_i(x), a_i, b_i, 3000)

  if norm_i > max_norm:
    max_norm = norm_i
    max_norm_interval = interval

  graph_i = np.linspace(a_i, b_i, 3000)

  ax.plot(graph_i, [p_i(z) for z in graph_i], label=f'Interpolated {interval}',
linestyle='dashed')
  ax.scatter(x_i, f(x_i), label=f'Interpolation points {interval}')

  print(f"Subinterval {interval}: Maximum function norm value: {norm_i}")

print("\nMaximum function norm:")

if max_norm_interval is not None:
    print(f"   Subinterval with the max norm: {max_norm_interval}")
    print(f"   Maximum function norm value  : {max_norm}")

    # Plot the subinterval with the maximum norm
    ax.set_title(f'Piecewise Interpolation - Max Norm Subinterval:
{max_norm_interval}')
    ax.legend()
    ax.set_xlabel('x')
    ax.set_ylabel('y')
```

```
    plt.tight_layout()
    plt.show()
  else:
    print("   No maximum function norm found.")
```

Output:




```
Subinterval [-2, -1]: Maximum function norm value: 0.00021575453014288648
Subinterval [-1, 1]: Maximum function norm value: 0.007161575490480576
Subinterval [1, 2]: Maximum function norm value: 0.00021575453014266444

Maximum function norm:
    Subinterval with the max norm: [-1, 1]
    Maximum function norm value   : 0.007161575490480576
```

The result shows the maximum function norm values for three subintervals [−2,−1],[−1,1], and [1,2]. The supremum norm measures the maximum absolute difference between the original function and the Lagrange interpolating polynomial within each subinterval. The norm values for [−2,−1] and [1,2] are 0.00021575, indicating highly accurate interpolation, while [−1,1] has a slightly larger norm of 0.00716158 which can indicate a more substantial difference. With this, we can conclude that the maximum function norm occurs in [−1,1] with 0.00716158.

4. Show using Newton's finite difference that $f^{(4)}(x_0)$ can be approximated by the function. Use this to approximate $f^{(4)}(1)$ by the given data points.

Solution: Given the points, we have

$x_0 = 0,\ x_1 = 0.025,\ x_2 = 0.05, x_3 = 0.1\ x_4 = 0.125,\ x_5 = 0.15,\ x_6 = 0.175, x_7 = 0.2$

$f(x_0) = 0.01, \quad f(x_1) = 0.03, \quad f(x_2) = 0.005, \quad f(x_3) = 0.04, \quad f(x_4) = 0.02,$
$f(x_5) = 0.01, f(x_6) = 0.03, f(x_7) = 0.02$

The Newton's finite difference, is given by

$$f^4(x_0) \approx \frac{f(x+2h) - 4f(x+h) + 6f(x) - 4f(x-h) + f(x-2h)}{h^4}$$

Now, we use h = 0.05 and approximate $f^4(0.1)$:

$$f^4(0.1) \approx \frac{f(x+2h) - 4f(x+h) + 6f(x) - 4f(x-h) + f(x-2h)}{h^4}$$

$$f^4(0.1) \approx \frac{f(0.1+2(0.05)) - 4f(0.1+0.05) + 6f(0.1) - 4f(0.1-0.05) + f(0.1-2(0.05))}{(0.05)^4}$$

$$f^4(0.1) \approx \frac{f(0.2) - 4f(0.15) + 6f(0.1) - 4f(0.05) + f(0)}{(0.05)^4}$$
$$f^4(0.1) \approx \frac{(0.02) - 4(0.01) + 6(0.04) - 4(0.005) + 0.01}{(0.05)^4}$$
$$f^4(0.1) \approx \frac{0.02 - 0.04 + 0.24 - 0.02 + 0.01}{0.00000625}$$
$$f^4(0.1) \approx \frac{0.21}{0.00000625}$$
$$f^4(0.1) \approx 33600$$

Therefore, we can conclude that the 4th derivative of the function with x=1 and h=0.05, is approximated to be 33600.

5. Approximate these integrals by Composite Simpson 1/3 (Simpson Thirds) and 3/8 (Simpsoneights) rules. Divide the interval of integration into 3,000 equal parts.

A. $\int_0^{\Pi} \sqrt{1 + cos^2 x}\, dx$ **B.** $\int_1^2 \frac{\sin x}{x} dx$

To approximate these integrals we need to consider the composite Simpsons ⅓ and Simpsons ⅜ rules. These two are numerical integration methods based on Simpson's rule. Simpson's rule is a numerical technique for approximating the definite integral of a function. It uses quadratic approximations for small intervals of the function and is known for its simplicity and accuracy for smooth functions.

For quadrature.py

```python
def simpsonsthird(f, a, b, n):
    '''
      Simpson's 1/3 rule for numerical integration.

    Parameters:
    - f: Function to be integrated.
    - a: Lower limit of integration.
    - b: Upper limit of integration.
    - n: Number of subintervals.

    Returns:
    - Result of the integration.
    '''
    h = (b - a) / n
    x = [a + i * h for i in range(n + 1)]

    result = h / 3
    fours = 4 * sum(f(x[i]) for i in range(1, n, 2))
    twos = 2 * sum(f(x[i]) for i in range(2, n - 1, 2))
    result *= f(x[0]) + fours + twos + f(x[n])

    return np.array(result)
```

This works by dividing the interval [a, b] into smaller subintervals and picking specific points within each. Furthermore, It calculates a weighted sum of function values at these points, using a pattern of 1-4-2-4-...-2-4-1. This sum is scaled by the width of the subintervals (h) and divided by 3. Mathematically, the process can be represented as:

$$\text{Integral} \approx \frac{h}{3}\left[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\right]$$

The weighted sum estimates the area under the curve. The algorithm's simplicity lies in its systematic approach to approximating the integral by combining quadratic approximations within each subinterval.

```python
def Simpsonseights(a, b, f, n):
  '''
    Simpson's 3/8 rule for numerical integration.

  Parameters:
  - a: Lower limit of integration.
  - b: Upper limit of integration.
  - f: Function to be integrated.
  - n: Number of subintervals (must be a multiple of 3).

  Returns:
  - Result of the integration.
  '''
  if n % 3 != 0:
    raise ValueError("Number of subintervals must be a multiple of 3")

  h = (b - a) / n
  sum_y = 0
  sum_y_3 = 0
  yn = 0 # Initialize yn outside the loop

  for i in range(n + 1):
    x = a + i * h
    y = f(x)

    if i == 0:
      y0 = y
    elif i == n:
      yn = y
    else:
      sum_y = sum_y + y
      if i % 3 == 0:
        sum_y_3 = sum_y_3 + y

  result = ((3 * h) / 8) * (y0 + 3 * sum_y - sum_y_3 + yn)
  return np.array(result)
```

In simple terms, it divides the integration interval [a, b] into smaller subintervals, each of width h. The algorithm systematically calculates the function values at various points within these subintervals, accumulating sums with specific weights. Notably, it places emphasis on every third function value. Mathematically, the result is given by

$$\frac{3h}{8}\left(y_0 + 3 \cdot \sum_{i=1}^{n-1} y_i - \sum_{i=1}^{n-1} y_{3i} + y_n\right)$$

This approach simplifies the understanding of Simpson's 3/8 rule, illustrating how the algorithm strategically combines function values to provide an approximation of the integral.

Implementation:

The first function, f_1(x) = sqrt(1 + cos(x)^2), involves the square root of the sum of 1 and the square of the cosine of x. The second function, f_2(x) = sin(x)/x, represents the ratio of the sine of x to x. Both functions are then integrated over specified intervals using two different numerical integration methods: Simpson's 1/3 rule and Simpson's 3/8

rule. The integration is performed on two different intervals: [0, π] for f_1(x) and [1, 2] for f_2(x). A total of 3000 subintervals (n = 3000) are used for both integrations, providing a reasonably accurate approximation.

Code:

```python
import numpy as np
import quadrature as qd

def f_1(x):
  return np.sqrt(1 + np.cos(x)**2)

def f_2(x):
  return np.sin(x)/x

a = 0
b = np.pi
n = 3000

result_thirds_1 = qd.simpsonsthird(f_1, a, b, n)
result_eights_1 = qd.Simpsonseights(a, b, f_1, n)

print('For function a')
print(f"Approximate integral using Simpson's 1/3: {result_thirds_1}")
print(f"Approximate integral using Simpson's 3/8: {result_eights_1}" '\n')

a = 1
b = 2
n = 3000

result_thirds_2 = qd.simpsonsthird(f_2, a, b, n)
result_eights_2 = qd.Simpsonseights(a, b, f_2, n)

print('For function b')
print(f"Approximate integral using Simpson's 1/3: {result_thirds_2}")
print(f"Approximate integral using Simpson's 3/8: {result_eights_2}")
```

The results of the numerical integrations using Simpson's 1/3 rule and Simpson's 3/8 rule are displayed for two different functions over their specified intervals.

```
For function a
Approximate integral using Simpson's 1/3: 3.820197789027711
Approximate integral using Simpson's 3/8: 3.8201977890277066
```

The close result between the results indicates that both methods are highly accurate for integrating the function f_1(x) = sqrt(1 + cos(x)^2) over the interval [0, π]. But notice that the approximation using Simpsons's ⅜ is quite accurate than Simpson's ⅓ that is because Simpson's 3/8 rule uses cubic approximations (third-degree polynomials) rather than Simpson's 1/3 who rule uses quadratic approximations (second-degree polynomials).

```
For function b
Approximate integral using Simpson's 1/3: 0.6593299064355115
Approximate integral using Simpson's 3/8: 0.6593299064355141
```

Similarly, for letter b, both methods can be accurate for smooth functions, but the choice between them depends on the specific needs and characteristics of the integration problem at hand.

6. Solve the initial value problem $\frac{dy}{dx} = y - \frac{1}{x}y^2$, y(1) = 1, to find y(1.2) by Runge-Kutta method of order four (RK4) and Runge-Kutta-Heun method of order three (RKH3) with step size 0.001.

We are solving a math problem: finding (y) in the equation $(\frac{dy}{dx} = y - \frac{1}{x}y^2)$, starting at $(y(1) = 1)$. The goal is to discover (y) at $(x = 1.2)$, using two methods, RK4 and RKH3, with a step size of 0.001. The solution is organized neatly in ODEscalar.py, containing the RK4 and RKH3 tools, and the final answer is in Item6Imp.py. This methodical approach efficiently tackles the initial value problem using numerical methods.

ODEscalar.py

```python
def runge_kutta4(f, y0, t0, tk, h):
  '''
    Solve an ordinary differential equation (ODE) using the Runge-Kutta method
of order four (RK4).

    Parameters:
    - f: Function representing the ODE in the form f(t, y).
    - y0: Initial value of y at t0.
    - t0: Initial value of the independent variable.
    - tk: Final value of the independent variable.
    - h: Step size for numerical integration.

    Returns:
    - t_values: Array of time values.
    - y_values: Array of corresponding y values.
    '''

  num_steps = int((tk - t0) / h) + 1
  t_values = np.linspace(t0, tk, num_steps)
  y_values = np.zeros(num_steps)
  y_values[0] = y0

  for i in range(1, num_steps):
    k1 = h * f(t_values[i-1], y_values[i-1])
    k2 = h * f(t_values[i-1] + h/2, y_values[i-1] + k1/2)
    k3 = h * f(t_values[i-1] + h/2, y_values[i-1] + k2/2)
    k4 = h * f(t_values[i-1] + h, y_values[i-1] + k3)

    y_values[i] = y_values[i-1] + (k1 + 2*k2 + 2*k3 + k4) / 6

  return t_values, y_values
```

Based on the course notes, the definition and formula for RK4 is given by:

Example 7.2.3 A popular example of a four-stage RK method of order 4 utilizes the weights of the Simpson rule and takes the form

$$u_{k+1} = u_k + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where $k_1 = f(t_k, u_k)$, $k_2 = f(t_k + \frac{h}{2}, u_k + \frac{h}{2}k_1)$, $k_3 = f(t_k + \frac{h}{2}, u_k + \frac{h}{2}k_2)$ and $k_4 = f(t_k + h, u_k + hk_3)$.

The algorithm iteratively approximates the solution by evaluating four weighted increments (k1, k2, k3, k4) at different points within each step. These increments are determined by the function f(t, y) representing the ODE and are combined using a weighted average to update the current value of y.

```python
def runge_kutta3(f, y0, t0, tk, h):
  '''
    Solve an ordinary differential equation (ODE) using the Runge-Kutta-Heun
method of order three (RKH3).

    Parameters:
    - f: Function representing the ODE in the form f(t, y).
    - y0: Initial value of y at t0.
    - t0: Initial value of the independent variable.
    - tk: Final value of the independent variable.
    - h: Step size for numerical integration.

    Returns:
    - t_values: Array of time values.
    - y_values: Array of corresponding y values.
    '''

  num_steps = int((tk - t0) / h) + 1
  t_values = np.linspace(t0, tk, num_steps)
  y_values = np.zeros(num_steps)
  y_values[0] = y0

  for i in range(1, num_steps):
    k1 = f(t_values[i-1], y_values[i-1])
    k2 = f(t_values[i-1] + h/3, y_values[i-1] + (h/3)*k1)
    k3 = f(t_values[i-1] + 2*h/3, y_values[i-1] + (2*h/3)*k2)

    y_values[i] = y_values[i-1] + (h/4) * (k1 + 3*k3)

  return t_values, y_values
```

The algorithm divides the problem into small steps, each calculated based on the current state of the system. For each step, it estimates the rate of change at the current point (k1) , as well as at two intermediate points within the step (k2 and k3). Based on the course note, the formula for rk3,

On the other hand, the *standard three-stage RK3 method* utilizes the parameters $c_1 = \frac{1}{6}$, $c_2 = \frac{2}{3}$, $c_3 = \frac{1}{6}$, $a_2 = \frac{1}{2}$, $a_3 = 1$ and $b_{32} = 2$. In this case, the iteration is based from

$$u_{k+1} = u_k + \frac{h}{6}(k_1 + 4k_2 + k_3)$$

where $k_1 = f(t_k, u_k)$, $k_2 = f(t_k + \frac{h}{2}, u_k + \frac{h}{2}k_1)$ and $k_3 = f(t_k + h, u_k - hk_1 + 2hk_2)$.

Implementation:

```python
from odescalar import *
import numpy as np
import matplotlib.pyplot as plt

def f(x, y):
  return y - (1/x) * y**2
```

```
if __name__ == '__main__':
  t0 = 1 # Initial time
  tk = 1.2 # Final time
  y0 = 1 # Initial condition
  h = 0.001 # Step size

  # Solve the ODE using Runge-Kutta Method of order three (RKH3)
  t_values_rkh3, y_values_rkh3 = runge_kutta3(f, y0, t0, tk, h)

  # Find the approximate value of y(1.2) using RKH3
  index_rkh3 = int((tk - t0) / h)
  y_approx_rkh3 = y_values_rkh3[index_rkh3]
  print(f"Approximate value of y(1.2) using RKH3: {y_approx_rkh3}")


    # Solve the ODE using Runge-Kutta Method of order four (RK4)
    t_values_rk4, y_values_rk4 = runge_kutta4(f, y0, t0, tk, h)

    # Find the approximate value of y(1.2) using RK4
    index_rk4 = int((tk - t0) / h)
    y_approx_rk4 = y_values_rk4[index_rk4]
    print(f"Approximate value of y(1.2) using RK4: {y_approx_rk4}")
```

Result:

```
Approximate value of y(1.2) using RKH3: 1.0167211704224606
Approximate value of y(1.2) using RK4: 1.016721170435565
```

The small difference between the two results can be attributed to the inherent differences in the numerical methods. RK4 is known for its higher accuracy compared to RKH3, as it involves more stages and higher-order approximations. The results indicate that both RKH3 and RK4 provide reasonably close approximations for y(1.2), with RK4 offering a slightly more accurate result. Because it is higher-order method. The "order" of a numerical method indicates the rate at which the method's error decreases with smaller step sizes. Higher-order methods generally provide more accurate results for the same step size compared to lower-order methods.