EngMath - HW0

A. Q. Snyder

September 21, 2023

1 HW template

Engr 5011: Homework #0

Introduction to numerical integration

Please submit with this page first, and the problems attached in order.
Due 21 September in class (or by 11PM in canvas)

Name (print): Aaron Snyder **TUID:**

9	1	5	1	8	7	2	8	9
I	**H**	**G**	**F**	**E**	**D**	**C**	**B**	**A**

USE THE ABOVE VALUES (A,B,C, ETC) WHEN A QUANTIFIED (NUMERICAL) ANSWER IS NEEDED.

"Technical points" possible for each question are shown; excellent documentation will recieve a bonus.
Also, if you have not already, **download matlab to your laptop computer from**

`https://download.temple.edu`

1. Problem 1 score/comments
   (a) Problem 1.a [**2 pt**]
   (b) Problem 1.b [**5 pt**]
   (c) Problem 1.c [**18 pt**]
2. Problem 2 score/comments.
   (a) Problem 2.a [**5 pt**]
   (b) Problem 2.b [**10 pt**]
3. Problem 3 score/comments
   (a) Problem 3.a [**3 pt**]
   (b) Problem 3.b [**7 pt**]

Engr-5011: Homework #0 Problem #1: Definite Integrals

Assemble submission for this problem in the following order:
(1) this page on top, followed by
(2) your handwritten notes for parts b,c followed by
(3) listing of your matlab scripts/functions for parts a,b,c,
(4) printout of matlab plotted results for parts a,b,c

Name (print): Aaron Snyder

Collaborants (print): ______

Documents & resources used ______

1. **Definite integrals!**

   (a) Implement the simpson-rule integrator and demonstrate that it works on a problem you can do another way.

   (b) Plot $I(b)$ on $-0 \leq b \leq 5$, computing $I(b)$ to at least three significant digits.

$$\boxed{I(b) = \int_0^5 e^{-b\cos(\tau)}\, d\tau \quad \text{on} \quad b = 0 : 1/10 : 5}$$

   What is $I(4.6)$? Write it here: $I(4.6) =$ ______

   (c) Evaluate I to at least three significant digits and write it here:

$$\boxed{I = \int_0^5 \frac{dx}{2-\sqrt{x}}; \qquad I = \_\_\_\_\_\_}$$

Engr-5011: Homework #0 Problem #2. Integrals on data

Assemble submission for this problem in the following order:
(1) this page on top, followed by
(2) your handwritten notes
(3) your matlab code(s)
(4) your computed results

Name (print): Aaron Snyder

Collaborants (print): ______ **Documents or resources used** ______

2. **Predictor-corrector integration scheme**

(a) Implement a predictor/corrector integration scheme and test it on a problem you can do another way.

(b) To three significant digits, integrate and plot y_j (where $y_j = y(t_j)$) if

$$\boxed{y' = \frac{y}{\cos(\frac{t}{3}) + \alpha\ f(t)} + t; \quad y(0) = -1}$$
$$\boxed{\alpha = \text{(average of your TUID digits)}}$$

Here $f(t)$ is tabulated data given by:

0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	**t**
1.00	0.84	0.78	0.73	0.68	0.65	0.61	0.58	0.55	0.53	0.50	0.48	0.45	0.43	0.41	0.39	0.37	0.35	0.33	**f**

Estimate the minimum value of y on $0 \le t \le 1.8$ from your results and write them here:

$t =$ ______

$y =$ ______

Engr-5011: Homework #0 Problem #3. Numerical integration of initial-value problems

Assemble submission for this problem in the following order:
(1) this page on top, followed by
(2) your handwritten notes
(3) your matlab code(s)
(4) your computed results

Name (print): Aaron Snyder

Collaborants (print): ______

Documents or resources used ______

3. **Numerical integration of initial-value problems**

(a) Implement the 4^{th}-order Runge Kutte integration scheme and test it on a problem that you can do another way.

(b) To three significant digits, integrate and plot $y(t), y'(t)$, and $y''(t)$ on $t \subseteq [0,5]$ if

α	=	$\frac{\text{average(A,B,C)}}{10}$
β	=	$\frac{\text{average(D,E,F)}}{10}$
γ	=	$\frac{\text{average(G,H,I)}}{10}$

and

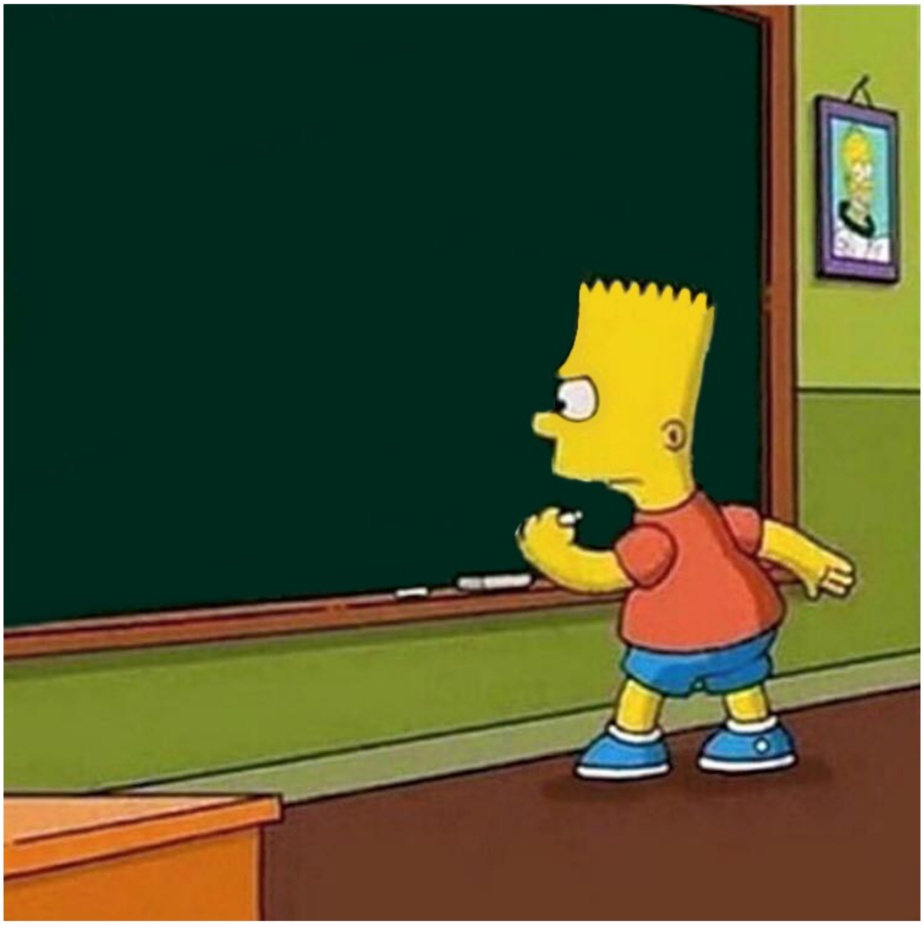
$$\boxed{y''' + \alpha\; y'' + \beta\; y\; y' + \gamma\; y = \cos(3t); \quad y(0) = \text{A};\; y'(0) = \text{B};\; y''(0) = \text{C};}$$

Here [A,B,C,D,E,F,G,H,I] are taken from your TUID.

At what value of $t \subseteq [0,5]$ is $y(t)$ maximized? Write them here:

$t =$ ______
$y =$ ______

2 Hand Calcs



1B. plot I_b on $0 \le b \le 5$

$$I_b = \int_0^5 e^{-b\cos(\tau)}\, d\tau$$

lets allow $-b\cos(\tau) = x$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$$

malaurin series

$x = -b\cos(\tau$

$$e^{-b\cos(\tau)} = 1 - b\cos(\tau) + \frac{b^2\cos^2(\tau)}{2!} + \frac{b^3\cos^3(\tau)}{3!} + \dots + \frac{b^n\cos^n(\tau)}{n!}$$

Term by term integration

$$\int 1\, d\tau = \tau$$

$$b\int \cos(\tau)\, d\tau = b\sin(\tau)$$

$$\frac{b^2}{2!}\int \cos^2(\tau)\, d\tau \xrightarrow[\text{reduction}]{\text{power}} \frac{b^2}{2!}\int \frac{1}{2} + \frac{\cos(2\tau)}{2} = \frac{b^2}{2!}\left(\frac{1}{2}\tau + \frac{1}{4}\sin 2\tau\right)$$

Higher order decomposition could follow. I'll use $n=2$ to sanity check my code

$$I(4.6) = 5 + 4.6\sin(5) + \left[\frac{4.6^2}{2!}\left(\frac{1}{2}(5) + \frac{1}{4}\sin(10)\right)\right] \approx 32.310$$

1C $I = \int_0^5 \frac{dx}{2-\sqrt{x}}$ observe singularity @ $x=4$

lets try intervals $0 \le x \le 3.999$ & $4.001 \le x \le 5$

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(a) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + 2\sum_{i=1}^{\frac{n}{2}-1} f(x_{2i}) + f(b)\right]$$

let $h = \frac{b-a}{n}$ where n is even

$$h_1 = \frac{3.999-0}{4} = 0.99975$$

$$h_2 = \frac{5-4.001}{4} = 0.24975$$

① $$\int_0^{3.999} f(x)\,dx \approx \frac{h_1}{3}\left[f(0) + 4\left(f(h_1) + f(3h_1)\right) + 2f(2h_1) + f(3.999)\right]$$

$$\int_0^{3.999} f(x)\,dx \approx \frac{0.99975}{3}\left[0.5 + 4\left(0.999875 + 3.729\right) + 2\left(1.706\right) + 3999.75\right]$$

$$\approx 340.347$$

② $$\int_{4.001}^{5} f(x)\,dx \approx \frac{h_2}{3}\left[f(4.001) + 4\left(f(4.001+h_2) + f(4.001+3h_2)\right) + 2f(4.001+2h_2) + f(5)\right]$$

$$\int_{4.001}^{5} f(x)\,dx \approx \frac{0.24975}{3}\left[-4000.25 + 4\left(-16.198\right) + \left(-5.571\right) + \left(-16.469\right) + \left(-4.234\right)\right] \approx 340.602$$

Answer = ① + ②

2b

$$TUID = \frac{9+1+5+1+8+7+2+8+9}{9} \approx 5.55\bar{5} = \alpha$$

$y(0) = -1$

time step

① define predictor $y^* = y_n + h \times \frac{dy}{dt}$ (predicted value @ next step)

② define corrector $y_{n+1} = y_n + \frac{h}{2}\left(\frac{dy}{dt}\bigg|_{t_n, y_n} + \frac{dy}{dt}\bigg|_{t_{n+1}, y^*}\right)$

corrected value

in python I'll probaly make an enumerated for loop with the table "t" values

3b

average $\sum$ TUID digits $\approx 5.55\vec{5}$

in python make a Letter $\rightarrow$ # map

"I" $\rightarrow$ 9 ... etc

$\alpha \approx 0.63\vec{3}$

$\beta \approx 0.53\vec{3}$

$\gamma = 0.5$

let $h = 0.1$?

$\frac{dy}{dt} = y'$ $\quad \frac{dy'}{dt} = y''$ $\quad \frac{dy''}{dt} = y'''$

$= \cos(3t) - \alpha y'' - \beta y y''$

Intermediate Slope (k_{1-4})

$k_1 = h \times f(t_n, y_n)$

$k_2 = h \times f(t_n + \frac{h}{2}, y_n + \frac{k_1}{2})$

$k_3 = h \times f(t_n + \frac{h}{2}, y_n + \frac{k_2}{2})$

$k_4 = h \times f(t_n + \frac{h}{2}, y_n + k_3)$

lets program :)

3 Python Outputs

1a

September 20, 2023

0.0.1 Library Imports

[ ]:
```
import numpy as np
import plotly.graph_objects as go
```

Lets imagine we have an LED in some sort of optical detection aparatus. We need multiple color LED's to detect different materials. If we want to make simultaneous detections with the LED's we might need a way to distinguish their signals. We could do this by "blinking" them really fast. The photo-diodes that detects their signals could send all of the information through a common mixer while the MCU performs an FFT to seperate the detection channels.

Lets simulate one of these detection channels

[ ]:
```
def main():
    # Constants
    TIMESTEP = 0.01
    NUMSAMPLES = 1000
    MOD_FREQ_HZ = 40
    CHANNEL_SEPARATION_HZ = 40

    # Main script execution
    signal = generate_signal(TIMESTEP, NUMSAMPLES)
    visualize_signal_and_fft_simpsons(signal, TIMESTEP, MOD_FREQ_HZ,␣
 ↪CHANNEL_SEPARATION_HZ)
```

- $y(t) = A \cdot \sin(2\pi ft + \phi)$

  Where:

  - $y(t)$ is the value of the wave at time t.
  - A is the amplitude of the wave, determining its maximum and minimum values.
  - f is the frequency of the wave, which specifies how many cycles occur in one second (measured in Hertz, Hz).
  - ϕ is the phase angle, which determines the horizontal shift of the wave along the time axis.

- $y(t) = \sin(40.0 \cdot 2\pi t)$

  In this equation, the frequency (f) is set to 40.0 Hz.

[ ]:
```
def generate_signal(timestep, numsamples):
    t = np.linspace(0, numsamples*timestep, numsamples)
    windowed_signal = np.sin(40.0 * 2.0 * np.pi * t) * np.hamming(numsamples)
    return windowed_signal

def fft_calculate(data, timestep):
    yf = np.abs(np.fft.fft(data))
    numsamples = len(data)
    freq = np.fft.fftfreq(numsamples, d=timestep)
    xf = freq[:numsamples//2]
    yf = yf[:numsamples//2] * 2.0 / numsamples
    return xf, yf

def find_nearest(array, value):
    idx = np.argmin(np.abs(array - value))
    nearestValue = array[idx]
    return idx, nearestValue
```

The Simpson's rule for integration is given by:

$$\int_{x_{2i}}^{x_{2i+2}} f(x)\,dx \approx \frac{h}{3}\left[f(x_{2i}) + 4f(x_{2i+1}) + f(x_{2i+2})\right]$$

[ ]:
```
def simpsons_integration(xf, yf, idx_start, idx_stop):
    # Ensure that the number of intervals is even
    n = idx_stop - idx_start
    if (idx_stop - idx_start) % 2 != 0:
        idx_stop -= 1  # or idx_start += 1, depending on your requirements

    h = (xf[idx_stop] - xf[idx_start]) / n
    result = 0

    # Loop in steps of 2 since Simpson's rule integrates over two intervals at␣
↪once
    for i in range(0, n - 1, 2):
        result += (h/3) * (yf[idx_start + i] + 4*yf[idx_start + i + 1] +␣
↪yf[idx_start + i + 2])

    return result
```

[ ]:
```
def visualize_signal_and_fft_simpsons(signal, timestep, mod_freq_hz,␣
↪channel_separation_hz):
    xf, yf = fft_calculate(signal, timestep)

    freq_start = mod_freq_hz - channel_separation_hz / 2
    freq_stop = mod_freq_hz + channel_separation_hz / 2
```

```
    idx_start, _ = find_nearest(xf, freq_start)
    idx_stop, _ = find_nearest(xf, freq_stop)

    # Ensure even number of intervals
    if (idx_stop - idx_start) % 2 == 0:
        idx_stop += 1

    integrated_area = simpsons_integration(xf, yf, idx_start, idx_stop)

    # Time-domain Signal plot
    fig1 = go.Figure()
    fig1.add_trace(go.Scatter(y=signal, mode='lines', name='Signal'))
    fig1.update_layout(title='Time-domain Signal')
    fig1.show()

    # FFT Magnitude plot
    fig2 = go.Figure()
    fig2.add_trace(go.Scatter(x=xf, y=yf, mode='lines', name='FFT'))
    fig2.add_trace(go.Scatter(x=xf[idx_start:idx_stop+1], y=yf[idx_start:
↪idx_stop+1], fill='tozeroy'))
    fig2.update_layout(title=f'FFT Magnitude - Integrated Simpsons Area:␣
↪{integrated_area:.3f}')
    fig2.show()
```

[ ]:
```
main()
```

Now I have a tight regions where to take my area under the curve for a given frequency. I could add another channel with a different modulation frequency now!

Library Imports

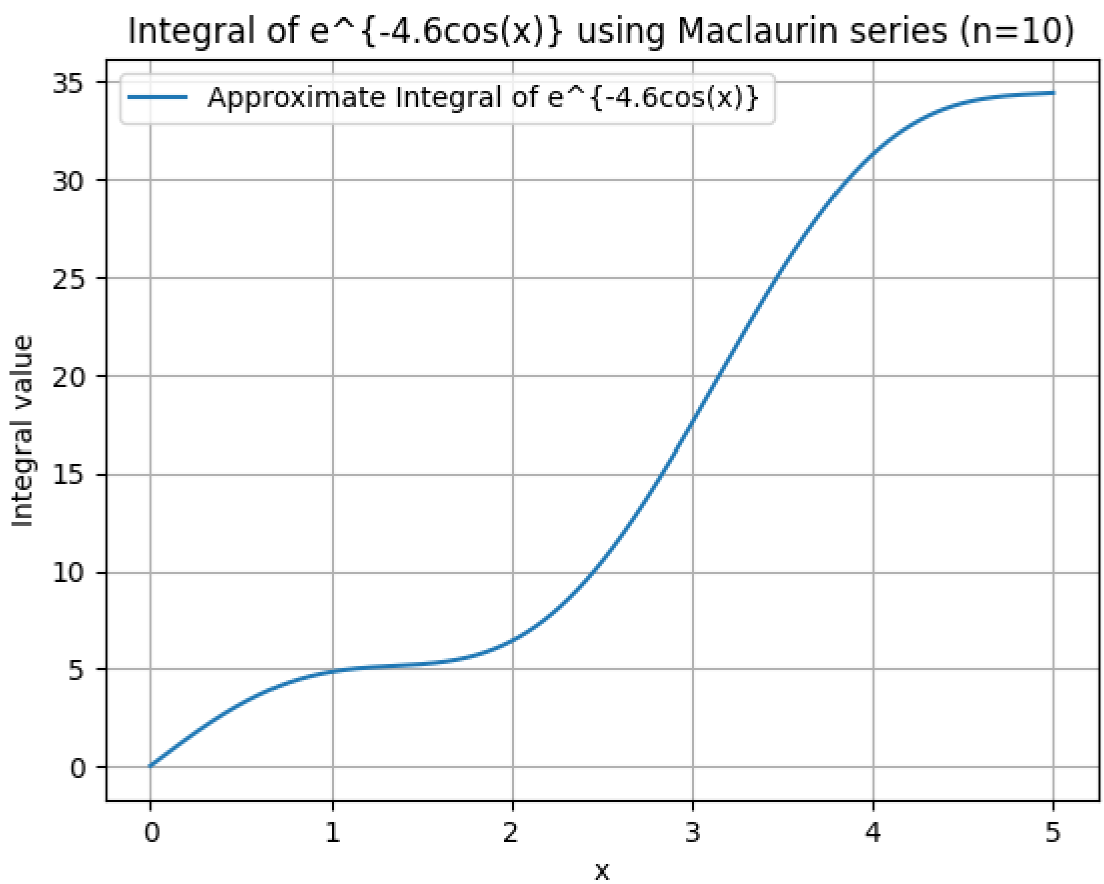
In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
```

Maclaurin Series

$$e^{-4.6\cos(x)} \approx 1 - 4.6\sin(x) + \frac{(-4.6\cos(x))^2}{2!} + \frac{(-4.6\cos(x))^3}{3!} + \ldots$$

In [ ]:
```
n = 3
# Maclaurin series approximation up to n = 10 for e^f(x)
def maclaurin_approximation(x):
    terms = [(-4.6 * np.cos(x))**i / math.factorial(i) for i in range(n)]
    return sum(terms)

# Integrate using numerical integration (trapezoid method)
x_values = np.linspace(0, 5, 1000)
y_values = maclaurin_approximation(x_values)

integral_values = np.cumsum(y_values) * (x_values[1] - x_values[0])

# Print the final value of the integral over the interval [0, 5]
print(f"Approximate value of the integral from 0 to 5 using 10 terms: {integral_val

# Plot the integral
plt.plot(x_values, integral_values, label="Approximate Integral of e^{-4.6cos(x)}")
plt.title("Integral of e^{-4.6cos(x)} using Maclaurin series (n=10)")
plt.xlabel("x")
plt.ylabel("Integral value")
plt.legend()
plt.grid(True)
plt.show()

print(integral_values[-1])
```

```
Approximate value of the integral from 0 to 5 using 10 terms: 34.44095363111059
```




```
34.44095363111059
```

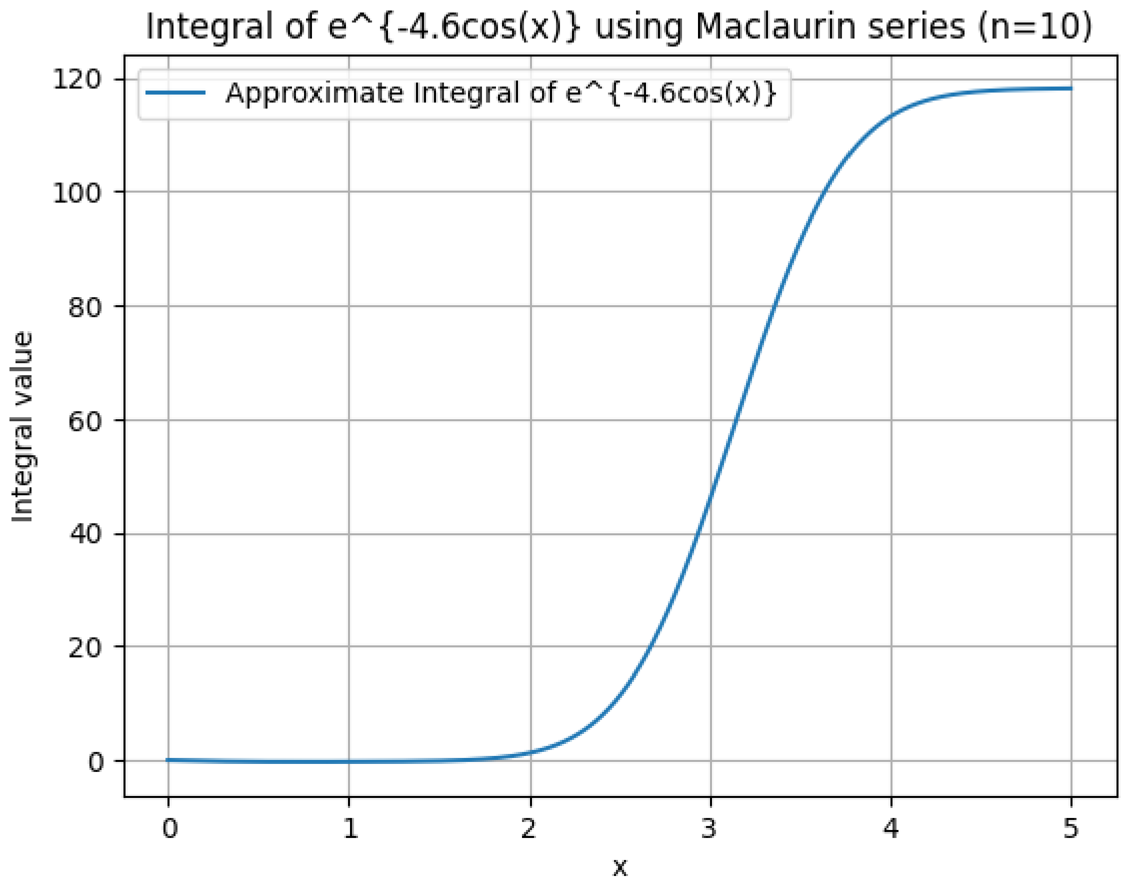
This is close to what I got in my hand calculation. Now lets step up the number of intervals!

Alt text

In [ ]:
```
n = 10
# Maclaurin series approximation up to n = 10 for e^f(x)
def maclaurin_approximation(x):
    terms = [(-4.6 * np.cos(x))**i / math.factorial(i) for i in range(n)]
    return sum(terms)

# Integrate using numerical integration (trapezoid method)
x_values = np.linspace(0, 5, 1000)
y_values = maclaurin_approximation(x_values)

integral_values = np.cumsum(y_values) * (x_values[1] - x_values[0])

# Print the final value of the integral over the interval [0, 5]
print(f"Approximate value of the integral from 0 to 5 using 10 terms: {integral_val

# Plot the integral
plt.plot(x_values, integral_values, label="Approximate Integral of e^{-4.6cos(x)}")
plt.title("Integral of e^{-4.6cos(x)} using Maclaurin series (n=10)")
plt.xlabel("x")
plt.ylabel("Integral value")
plt.legend()
```

```
plt.grid(True)
plt.show()

print(integral_values[-1])
```

Approximate value of the integral from 0 to 5 using 10 terms: 118.10686276153056

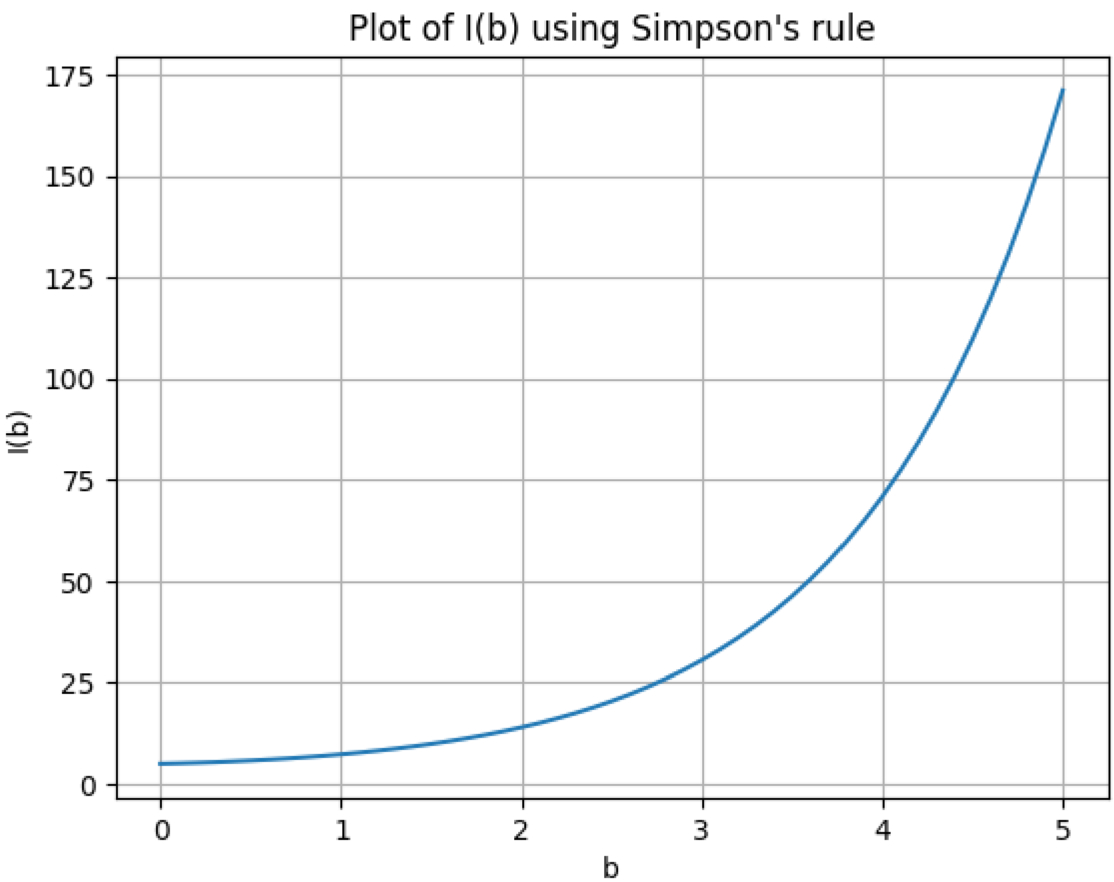


118.10686276153056

In [ ]:
```
# Define the function to integrate
def integrand(t, b):
    return np.exp(-b * np.cos(t))

# Calculate I(4.6)
num_intervals = 100  # Using 100 intervals for a good approximation
b_value_to_shade = 4.6
x_values = np.linspace(0, 5, num_intervals + 1)
h = x_values[1] - x_values[0]
y_values = integrand(x_values, b_value_to_shade)
I_4_6 = (h / 3) * (y_values[0] + 4 * np.sum(y_values[1:-1:2]) + 2 * np.sum(y_values
print(f"I(4.6) = {I_4_6:.4f}")

# Calculate I(b) for each b using Simpson's rule
b_values = np.arange(0, 5.1, 0.1)
I_values = []

for b in b_values:
    y_values = integrand(np.linspace(0, 5, num_intervals + 1), b)
    I = (h / 3) * (y_values[0] + 4 * np.sum(y_values[1:-1:2]) + 2 * np.sum(y_values
    I_values.append(I)
```

```python
# Plot the results
plt.plot(b_values, I_values)
plt.xlabel('b')
plt.ylabel('I(b)')
plt.title('Plot of I(b) using Simpson\'s rule')
plt.grid(True)
plt.show()
```

```
I(4.6) = 119.8920
```



Library Imports

In [ ]:
```
import numpy as np
import plotly.graph_objects as go
from scipy.integrate import quad
```

Method 1

In [ ]:
```
# Define the function to be integrated
def f(x):
    return 1 / (2 - np.sqrt(x))

# Integrate the function from 0 to 5
integral_value, _ = quad(f, 0, 5)

# Generate x values for plotting
x = np.linspace(0, 5, 1000)
y = f(x)
```

```
C:\Users\Aaron\AppData\Local\Temp\ipykernel_19532\3069320982.py:6: IntegrationWarnin
g: The integral is probably divergent, or slowly convergent.
  integral_value, _ = quad(f, 0, 5)
```

what my computer really means....

Alt text

good thing Gaussian Quadrature is robust

$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} w_i \cdot f(x_i)$

In [ ]:
```
# Create the plotly figure
fig = go.Figure()

# Plot the function
fig.add_trace(go.Scatter(x=x, y=y, fill='tozeroy', mode='lines', name='y = 1/(2-√x)

# Add title and labels
fig.update_layout(title='Graph of y = 1/(2-√x)',
                  xaxis_title='x',
                  yaxis_title='y')

# Show the figure
fig.show()
```

In [ ]:
```
# Print the value of the integral
print(f"Value of I from 0 to 5: {integral_value:.5f}")
```

```
Value of I from 0 to 5: 4.07499
```

Method 2

In [ ]:
```
# Define the function to be integrated
def f(x):
    return 1 / (2 - np.sqrt(x))

# Integrate the function from 0 to 3.99 and from 4.01 to 5
integral_value_1, _ = quad(f, 0, 3.99999)
integral_value_2, _ = quad(f, 4.00001, 5)

# Sum the two integrals
total_integral = integral_value_1 + integral_value_2

print(f"Value of I from 0 to 5: {total_integral:.5f}")
```

```
Value of I from 0 to 5: 4.07500
```

2a

September 20, 2023

0.0.1 Library Imports

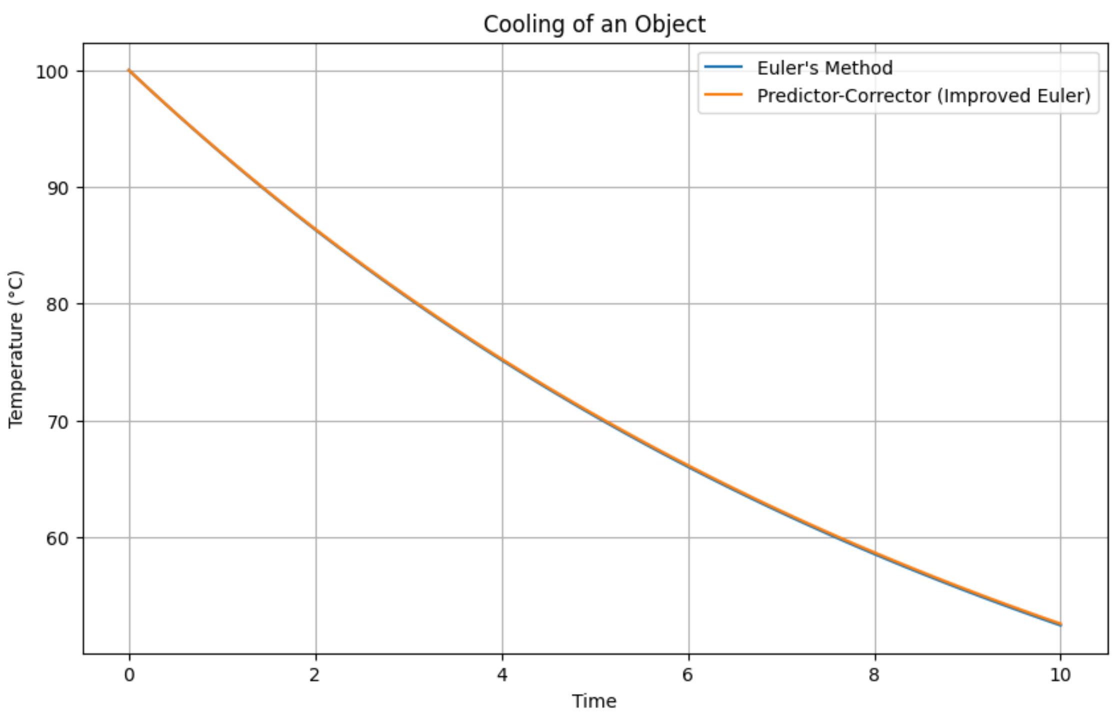
[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

$dT/dt = -k * (T - Ta)$

[ ]:
```
# Parameters
k = 0.1  # Cooling constant
Ta = 25  # Ambient temperature (degrees Celsius)

# Initial conditions
T0 = 100  # Initial temperature (degrees Celsius)
t0 = 0    # Initial time
tf = 10   # Final time

# Time step and number of steps
dt = 0.1
num_steps = int((tf - t0) / dt)

# Arrays to store results
time_euler = np.zeros(num_steps + 1)
temp_euler = np.zeros(num_steps + 1)
time_predictor_corrector = np.zeros(num_steps + 1)
temp_predictor_corrector = np.zeros(num_steps + 1)

# Euler's method
time_euler[0] = t0
temp_euler[0] = T0
for i in range(num_steps):
    time_euler[i + 1] = time_euler[i] + dt
    temp_euler[i + 1] = temp_euler[i] - k * (temp_euler[i] - Ta) * dt

# Predictor-Corrector (Improved Euler) method
time_predictor_corrector[0] = t0
temp_predictor_corrector[0] = T0
for i in range(num_steps):
```

```
    time_predictor_corrector[i + 1] = time_predictor_corrector[i] + dt
    # Predictor step
    predictor_temp = temp_predictor_corrector[i] - k *␣
↪(temp_predictor_corrector[i] - Ta) * dt
    # Corrector step
    temp_predictor_corrector[i + 1] = temp_predictor_corrector[i] - 0.5 * k *␣
↪((temp_predictor_corrector[i] - Ta) + (predictor_temp - Ta)) * dt
```

[ ]:
```
# Plot results
plt.figure(figsize=(10, 6))
plt.plot(time_euler, temp_euler, label="Euler's Method")
plt.plot(time_predictor_corrector, temp_predictor_corrector,␣
↪label="Predictor-Corrector (Improved Euler)")
plt.xlabel("Time")
plt.ylabel("Temperature (°C)")
plt.title("Cooling of an Object")
plt.legend()
plt.grid(True)
plt.show()
```



Library Imports

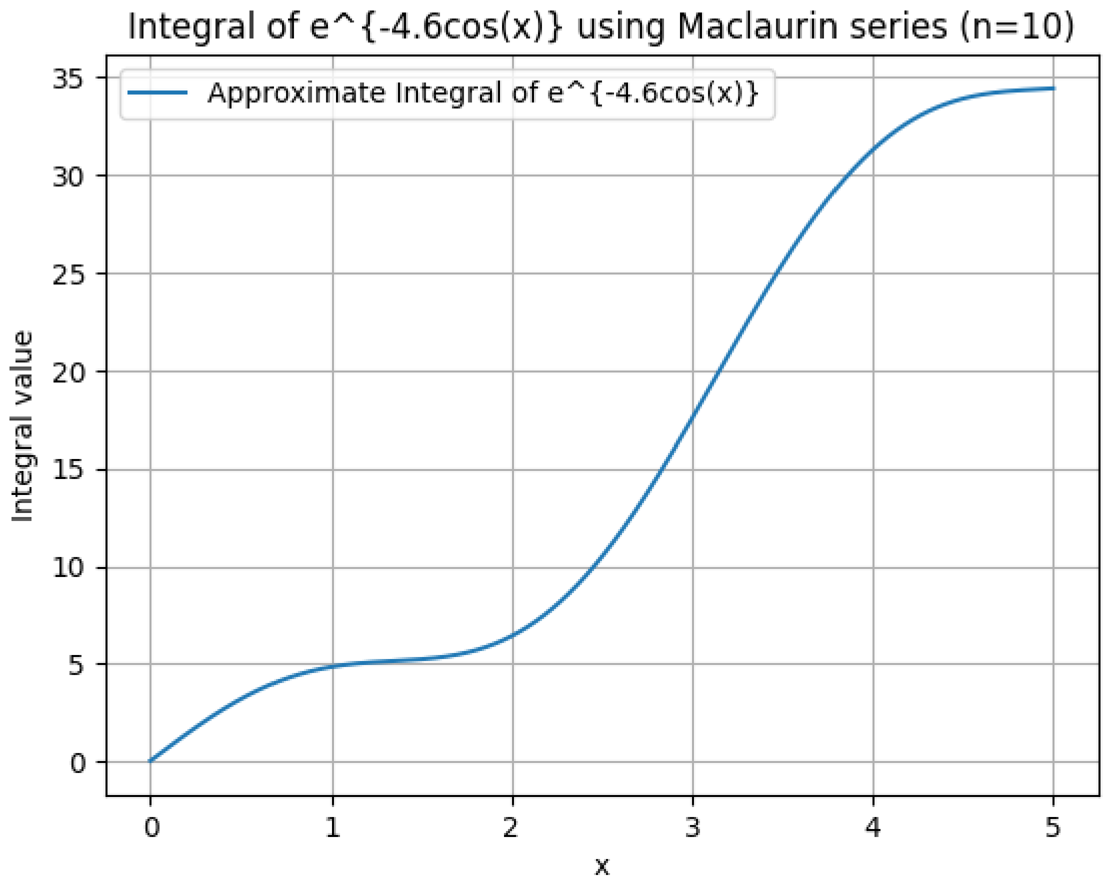
In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
```

Maclaurin Series

$$e^{-4.6\cos(x)} \approx 1 - 4.6\sin(x) + \frac{(-4.6\cos(x))^2}{2!} + \frac{(-4.6\cos(x))^3}{3!} + \ldots$$

In [ ]:
```
n = 3
# Maclaurin series approximation up to n = 10 for e^f(x)
def maclaurin_approximation(x):
    terms = [(-4.6 * np.cos(x))**i / math.factorial(i) for i in range(n)]
    return sum(terms)

# Integrate using numerical integration (trapezoid method)
x_values = np.linspace(0, 5, 1000)
y_values = maclaurin_approximation(x_values)

integral_values = np.cumsum(y_values) * (x_values[1] - x_values[0])

# Print the final value of the integral over the interval [0, 5]
print(f"Approximate value of the integral from 0 to 5 using 10 terms: {integral_val

# Plot the integral
plt.plot(x_values, integral_values, label="Approximate Integral of e^{-4.6cos(x)}")
plt.title("Integral of e^{-4.6cos(x)} using Maclaurin series (n=10)")
plt.xlabel("x")
plt.ylabel("Integral value")
plt.legend()
plt.grid(True)
plt.show()

print(integral_values[-1])
```

```
Approximate value of the integral from 0 to 5 using 10 terms: 34.44095363111059
```




```
34.44095363111059
```

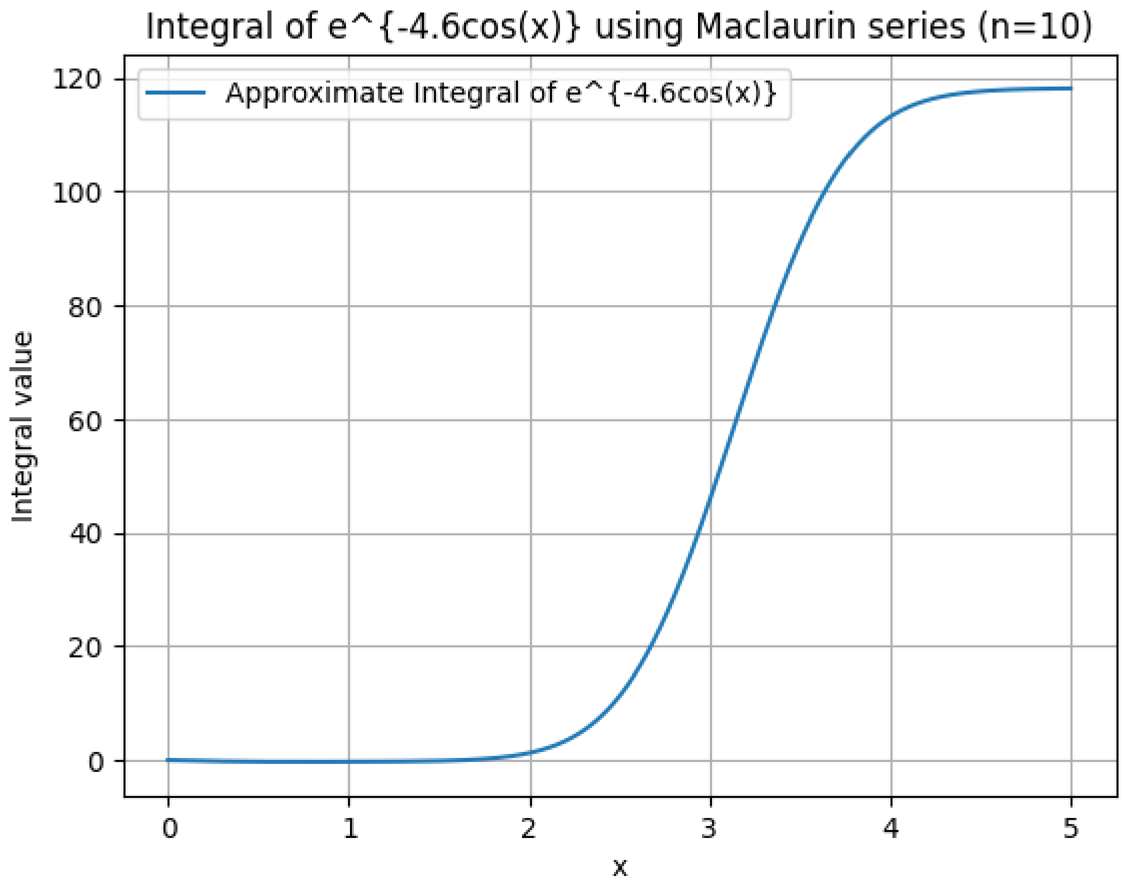
This is close to what I got in my hand calculation. Now lets step up the number of intervals!

Alt text

In [ ]:
```
n = 10
# Maclaurin series approximation up to n = 10 for e^f(x)
def maclaurin_approximation(x):
    terms = [(-4.6 * np.cos(x))**i / math.factorial(i) for i in range(n)]
    return sum(terms)

# Integrate using numerical integration (trapezoid method)
x_values = np.linspace(0, 5, 1000)
y_values = maclaurin_approximation(x_values)

integral_values = np.cumsum(y_values) * (x_values[1] - x_values[0])

# Print the final value of the integral over the interval [0, 5]
print(f"Approximate value of the integral from 0 to 5 using 10 terms: {integral_val

# Plot the integral
plt.plot(x_values, integral_values, label="Approximate Integral of e^{-4.6cos(x)}")
plt.title("Integral of e^{-4.6cos(x)} using Maclaurin series (n=10)")
plt.xlabel("x")
plt.ylabel("Integral value")
plt.legend()
```

```
plt.grid(True)
plt.show()

print(integral_values[-1])
```

```
Approximate value of the integral from 0 to 5 using 10 terms: 118.10686276153056
```



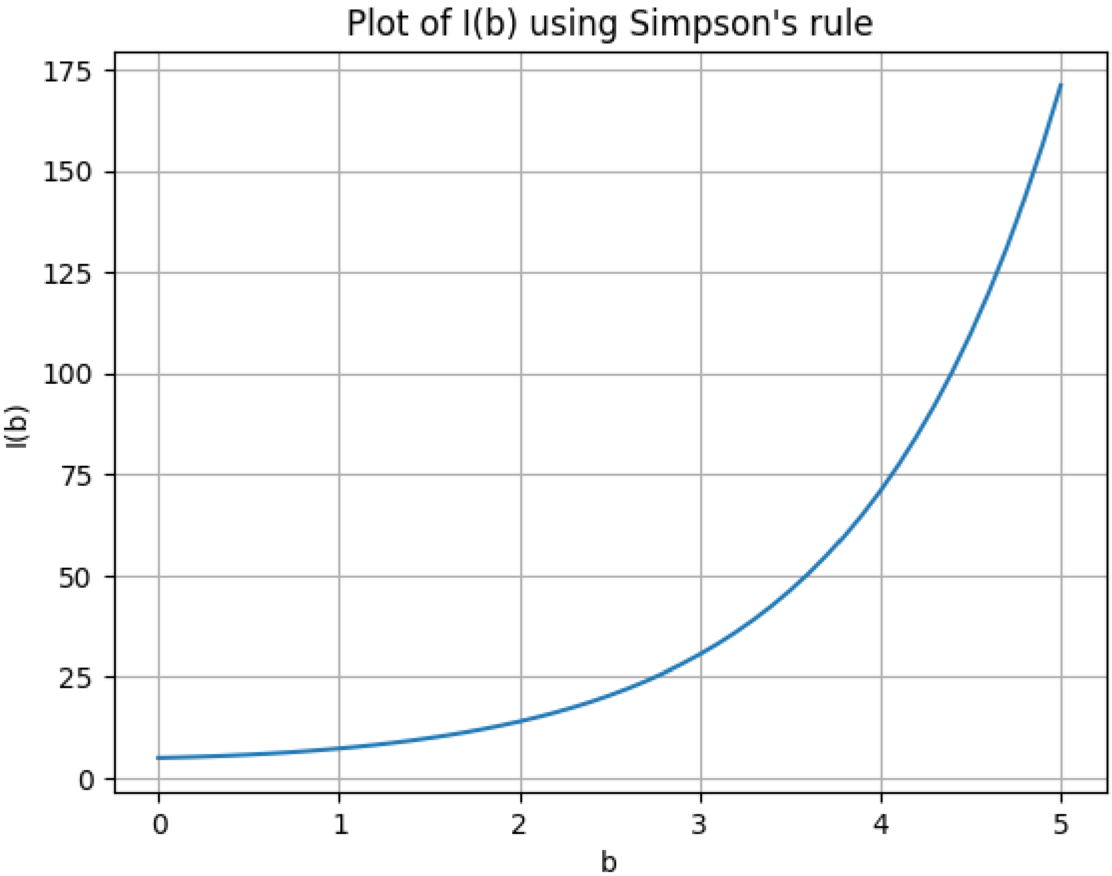

```
118.10686276153056
```

In [ ]:

```
# Define the function to integrate
def integrand(t, b):
    return np.exp(-b * np.cos(t))

# Calculate I(4.6)
num_intervals = 100  # Using 100 intervals for a good approximation
b_value_to_shade = 4.6
x_values = np.linspace(0, 5, num_intervals + 1)
h = x_values[1] - x_values[0]
y_values = integrand(x_values, b_value_to_shade)
I_4_6 = (h / 3) * (y_values[0] + 4 * np.sum(y_values[1:-1:2]) + 2 * np.sum(y_values
print(f"I(4.6) = {I_4_6:.4f}")

# Calculate I(b) for each b using Simpson's rule
b_values = np.arange(0, 5.1, 0.1)
I_values = []

for b in b_values:
    y_values = integrand(np.linspace(0, 5, num_intervals + 1), b)
    I = (h / 3) * (y_values[0] + 4 * np.sum(y_values[1:-1:2]) + 2 * np.sum(y_values
    I_values.append(I)
```

```python
# Plot the results
plt.plot(b_values, I_values)
plt.xlabel('b')
plt.ylabel('I(b)')
plt.title('Plot of I(b) using Simpson\'s rule')
plt.grid(True)
plt.show()
```

```
I(4.6) = 119.8920
```



Library Imports

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Logistic Growth Differential Equation: $\frac{dP}{dt} = rP\left(1 - \frac{P}{K}\right)$

Where:

$P(t)$ is the population at time t.
r is the growth rate.
K is the carrying capacity of the environment.

Exact Solution for Logistic Growth: $P(t) = \frac{KP_0e^{rt}}{K+P_0(e^{rt}-1)}$

Where:

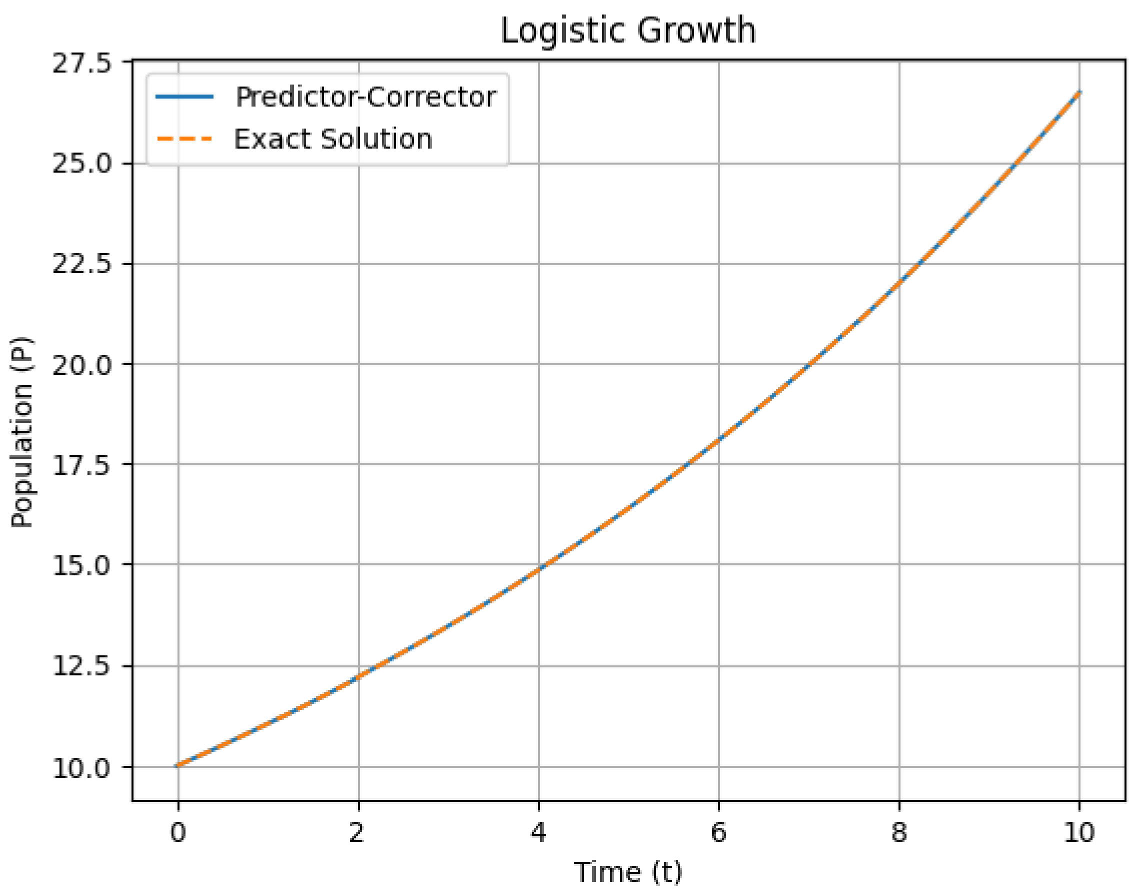
P_0 is the initial population.
Alt text

In [ ]:
```
def logistic_growth(t, P, r, K):
    return r * P * (1 - P / K)

def exact_solution(t, P0, r, K):
    return (K * P0 * np.exp(r * t)) / (K + P0 * (np.exp(r * t) - 1))

def predictor_corrector(y0, t0, h, N, r, K):
    t = [t0]
    P = [y0]

    # Bootstrap using 4th order Runge-Kutta
    for i in range(1):
        k1 = h * logistic_growth(t[-1], P[-1], r, K)
        k2 = h * logistic_growth(t[-1] + 0.5 * h, P[-1] + 0.5 * k1, r, K)
        k3 = h * logistic_growth(t[-1] + 0.5 * h, P[-1] + 0.5 * k2, r, K)
        k4 = h * logistic_growth(t[-1] + h, P[-1] + k3, r, K)

        P.append(P[-1] + (k1 + 2 * k2 + 2 * k3 + k4) / 6)
        t.append(t[-1] + h)

    for i in range(1, N):
        # Predictor
        P_predict = P[-1] + h * (1.5 * logistic_growth(t[-1], P[-1], r, K) - 0.5 *
        t_predict = t[-1] + h

        # Corrector
        P_correct = P[-1] + h/2 * (logistic_growth(t[-1], P[-1], r, K) + logistic_g
```

```
        P.append(P_correct)
        t.append(t_predict)

    return t, P

# Parameters
t0 = 0.0
P0 = 10.0
h = 0.1
N = 100
r = 0.1
K = 1000

t_vals, P_vals = predictor_corrector(P0, t0, h, N, r, K)
```

In [ ]:

```
# Visualization
plt.plot(t_vals, P_vals, label="Predictor-Corrector")
plt.plot(t_vals, [exact_solution(t, P0, r, K) for t in t_vals], '--', label="Exact
plt.xlabel('Time (t)')
plt.ylabel('Population (P)')
plt.legend()
plt.title("Logistic Growth")
plt.grid(True)
plt.show()
```



3b

September 20, 2023

0.0.1 Library Imports

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[ ]:
```
TUID = [9, 1, 5, 1, 8, 7, 2, 8, 9]
LETTER_MAP = ['I', 'H', 'G', 'F', 'E', 'D', 'C', 'B', 'A']
total_sum = 0

for i in range(len(TUID)):
    total_sum += TUID[i]

average = total_sum / len(TUID)

print(f'my TUID average: {average}')  # Corrected variable name to 'average'

# Create a dictionary to map letters to integers
letter_to_int_map = {letter: integer for letter, integer in zip(LETTER_MAP,␣
↪TUID)}

# Now, calculate the average of the letters of interest
alpha_letters = ["A", "B", "C"]
beta_letters = ["D", "E", "F"]
gamma_letters = ["G", "H", "I"]

alpha = np.average([letter_to_int_map[letter] for letter in alpha_letters]) / 10
print(alpha)

beta = np.average([letter_to_int_map[letter] for letter in beta_letters]) / 10
print(beta)

gamma = np.average([letter_to_int_map[letter] for letter in gamma_letters]) / 10
print(gamma)
```

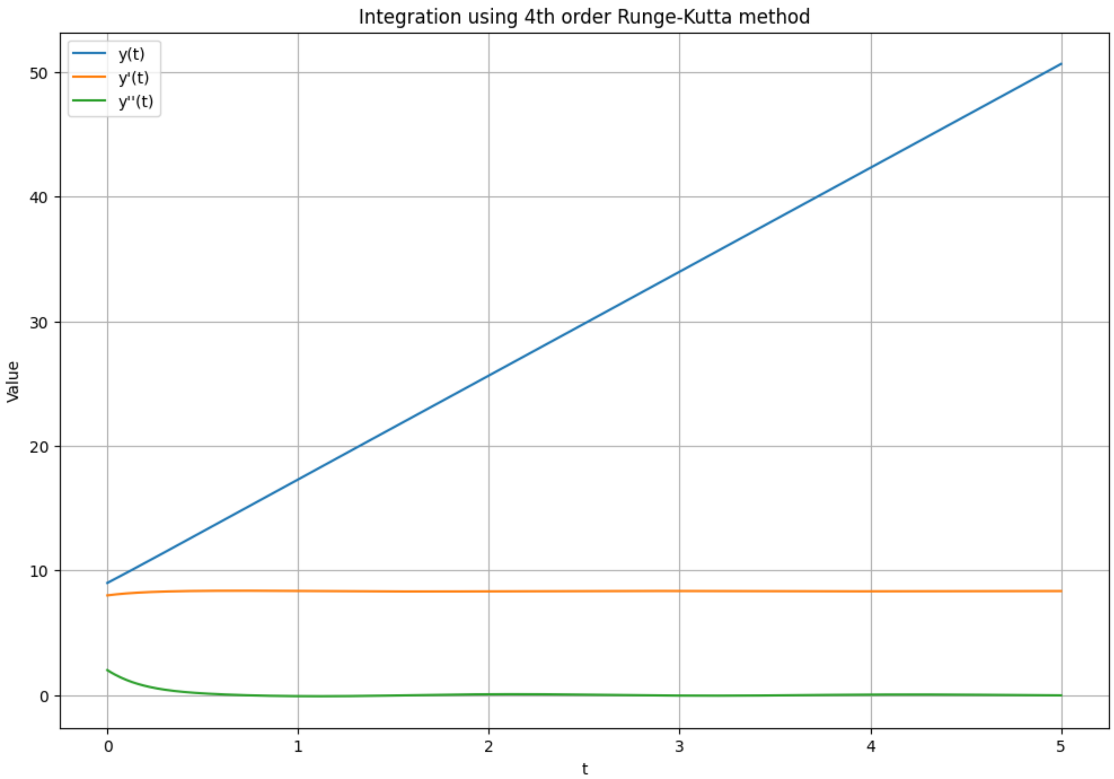
```
my TUID average: 5.555555555555555
0.6333333333333333
0.5333333333333333
```

0.5

[ ]:
```
def runge_kutta_4th_order(h, T, u0, alpha, beta):
    # The system of ODEs
    def f(t, u):
        y, y_prime, y_double_prime = u
        f1 = y_prime
        f2 = y_double_prime
        f3 = np.cos(3*t) - alpha*y_double_prime - beta*y*y_double_prime
        return [f1, f2, f3]

    t_values = np.arange(0, T+h, h)
    u_values = [u0]

    for t in t_values[:-1]:
        u = u_values[-1]

        k1 = h * np.array(f(t, u))
        k2 = h * np.array(f(t + 0.5*h, u + 0.5*k1))
        k3 = h * np.array(f(t + 0.5*h, u + 0.5*k2))
        k4 = h * np.array(f(t + h, u + k3))

        new_u = u + (1/6) * (k1 + 2*k2 + 2*k3 + k4)
        u_values.append(new_u)

    return t_values, np.array(u_values)

# Parameters
T = 5
h = 0.01

# From TUID letter mapping
A = 9
B = 8
C = 2

u0 = [A, B, C]

alpha = np.average([letter_to_int_map[letter] for letter in alpha_letters]) / 10
beta = np.average([letter_to_int_map[letter] for letter in beta_letters]) / 10

t_values, u_values = runge_kutta_4th_order(h, T, u0, alpha, beta)
```

[ ]:
```
plt.figure(figsize=(12,8))
plt.plot(t_values, u_values[:, 0], label="y(t)")
plt.plot(t_values, u_values[:, 1], label="y'(t)")
plt.plot(t_values, u_values[:, 2], label="y''(t)")
```

```
plt.legend()
plt.xlabel('t')
plt.ylabel('Value')
plt.title('Integration using 4th order Runge-Kutta method')
plt.grid(True)
plt.show()
```



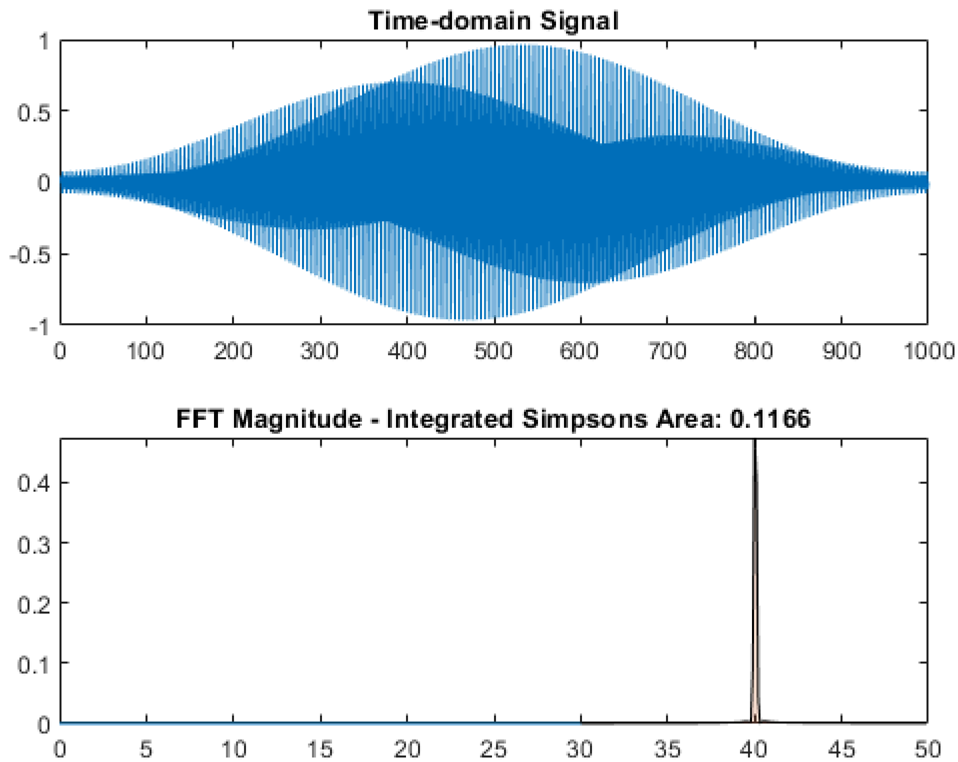
[ ]:
```
# Finding the index where y(t) is maximized
max_y_index = np.argmax(u_values[:, 0])
# Retrieving the corresponding time value
t_at_max_y = t_values[max_y_index]
# Finding the maximum value of y(t)
max_y_value = u_values[max_y_index, 0]

print(f"y(t) is maximized at t = {t_at_max_y} with a value of y(t) =␣
  ↪{max_y_value}")
```

```
y(t) is maximized at t = 5.0 with a value of y(t) = 50.65956128756216
```

4 MATLAB Outputs

```
function main_function()

    % Constants
    TIMESTEP = 0.01;
    NUMSAMPLES = 1000;
    MOD_FREQ_HZ = 40;
    CHANNEL_SEPARATION_HZ = 40;

    % Main script execution
    signal = generate_signal(TIMESTEP, NUMSAMPLES);
    visualize_signal_and_fft_simpsons(signal, TIMESTEP, MOD_FREQ_HZ,
 CHANNEL_SEPARATION_HZ);

end

function signal = generate_signal(timestep, numsamples)
    t = linspace(0, numsamples*timestep, numsamples);
    windowed_signal = sin(40.0 * 2.0 * pi * t) .* hamming(numsamples).';
    signal = windowed_signal;
end

function [xf, yf] = fft_calculate(data, timestep)
    yf = abs(fft(data));
    numsamples = length(data);
    freq = 0:1/timestep/numsamples:1/timestep - 1/timestep/numsamples;
    xf = freq(1:numsamples/2);
    yf = yf(1:numsamples/2) * 2.0 / numsamples;
end

function [idx, nearestValue] = find_nearest(array, value)
    [~, idx] = min(abs(array - value));
    nearestValue = array(idx);
end

function result = simpsons_integration(xf, yf, idx_start, idx_stop)
    n = idx_stop - idx_start;
    if mod(n, 2) ~= 0
        error('Number of intervals should be even for composite Simpson''s
 rule.');
    end

    h = (xf(idx_stop) - xf(idx_start)) / n;
    result = 0;

    for i = 0:2:n-2
        result = result + (h/3) * (yf(idx_start + i) + 4*yf(idx_start + i + 1)
 + yf(idx_start + i + 2));
    end
end

function visualize_signal_and_fft_simpsons(signal, timestep, mod_freq_hz,
 channel_separation_hz)
```

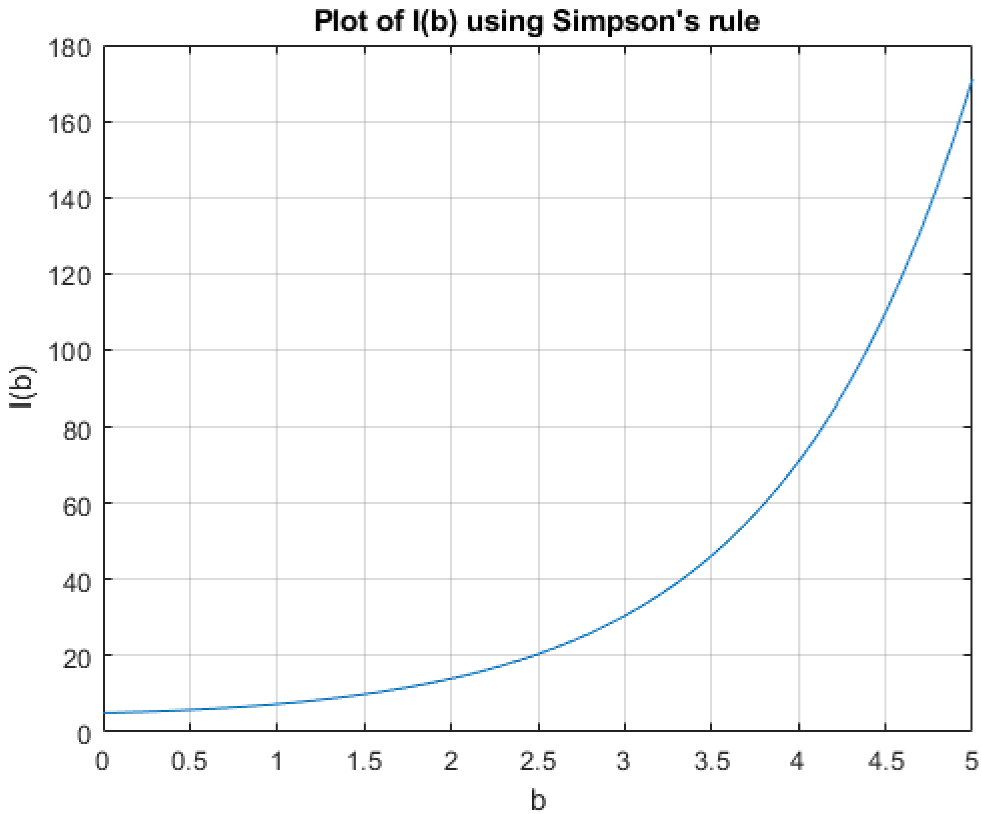
```
    [xf, yf] = fft_calculate(signal, timestep);

    freq_start = mod_freq_hz - channel_separation_hz / 2;
    freq_stop = mod_freq_hz + channel_separation_hz / 2;

    [idx_start, ~] = find_nearest(xf, freq_start);
    [idx_stop, ~] = find_nearest(xf, freq_stop);

    % Ensure an odd number of indices (even number of intervals) for Simpson's
 rule
    if mod(idx_stop - idx_start, 2) == 1
        idx_stop = idx_stop - 1;
    end

    integrated_area = simpsons_integration(xf, yf, idx_start, idx_stop);

    % Time-domain Signal plot
    subplot(2, 1, 1);
    plot(signal);
    title('Time-domain Signal');

    % FFT Magnitude plot
    subplot(2, 1, 2);
    plot(xf, yf);
    hold on;

    % Highlight Area of Interest
    freq_start = mod_freq_hz - (channel_separation_hz * 0.25);
    freq_stop = mod_freq_hz + (channel_separation_hz * 0.25);
    [idx_start, ~] = find_nearest(xf, freq_start);
    [idx_stop, ~] = find_nearest(xf, freq_stop);

    area(xf(idx_start:idx_stop), yf(idx_start:idx_stop), 'FaceAlpha', 0.2);
    title(['FFT Magnitude - Integrated Simpsons Area: ',
 num2str(integrated_area, '%.4f')]);
    hold off;
end
```



Published with MATLAB® R2023a

```
function b1()

    % Calculate I(4.6)
    num_intervals = 100; % Using 100 intervals for a good approximation
    b_value_to_shade = 4.6;
    x_values = linspace(0, 5, num_intervals + 1);
    h = x_values(2) - x_values(1);
    y_values = integrand(x_values, b_value_to_shade);
    I_4_6 = (h / 3) * (y_values(1) + 4 * sum(y_values(2:2:end-1)) + 2 *
 sum(y_values(3:2:end-1)) + y_values(end));
    fprintf('I(4.6) = %.4f\n', I_4_6);

    % Calculate I(b) for each b using Simpson's rule
    b_values = 0:0.1:5;
    I_values = zeros(1, length(b_values));

    for idx = 1:length(b_values)
        b = b_values(idx);
        y_values = integrand(linspace(0, 5, num_intervals + 1), b);
        I = (h / 3) * (y_values(1) + 4 * sum(y_values(2:2:end-1)) + 2 *
 sum(y_values(3:2:end-1)) + y_values(end));
        I_values(idx) = I;
    end

    % Plot the results
    plot(b_values, I_values)
    xlabel('b')
    ylabel('I(b)')
    title("Plot of I(b) using Simpson's rule")
    grid on

    function y = integrand(t, b)
        y = exp(-b .* cos(t));
    end
end
```

I(4.6) = 119.8920



Published with MATLAB® R2023a

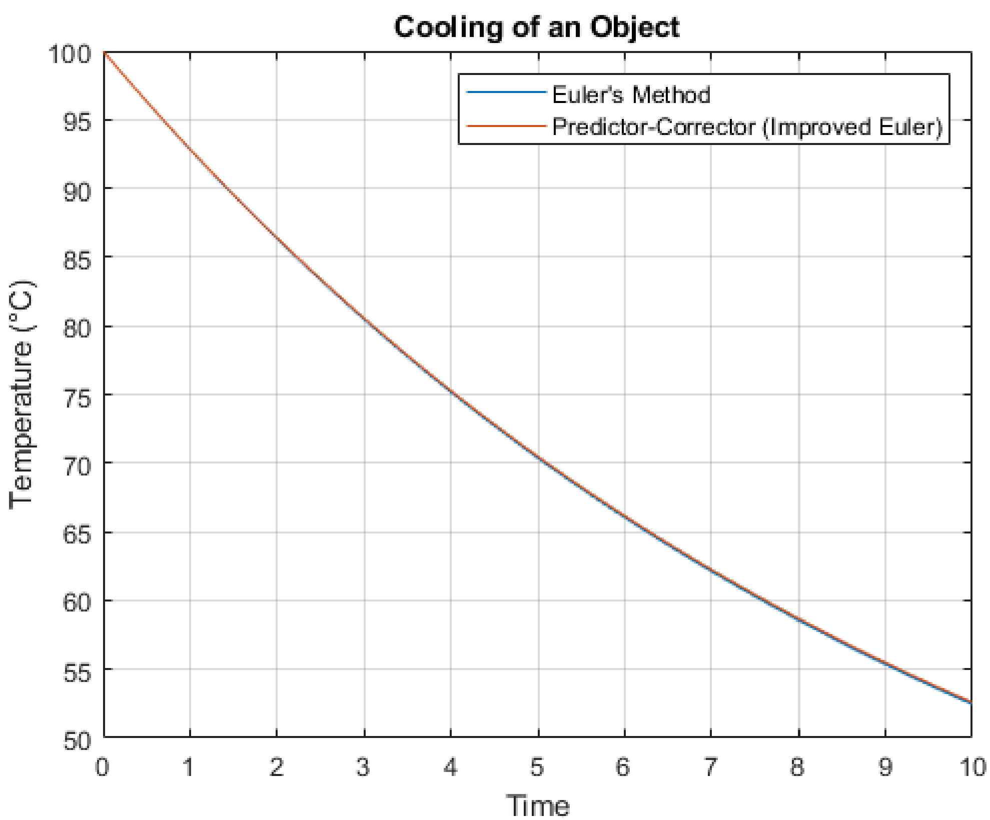
```
function main()

% Define the function to be integrated
f = @(x) 1 ./ (2 - sqrt(x));

% Integrate the function from 0 to 3.99 and from 4.01 to 5
integral_value_1 = integral(f, 0, 3.99999);
integral_value_2 = integral(f, 4.00001, 5);

% Sum the two integrals
total_integral = integral_value_1 + integral_value_2;

% Display the result
fprintf('Value of I from 0 to 5: %.5f\n', total_integral);

end
```

```
Value of I from 0 to 5: 4.07500
```

Published with MATLAB® R2023a

```matlab
function cooling_simulation()

% Parameters
k = 0.1;  % Cooling constant
Ta = 25;  % Ambient temperature (degrees Celsius)

% Initial conditions
T0 = 100;  % Initial temperature (degrees Celsius)
t0 = 0;    % Initial time
tf = 10;   % Final time

% Time step and number of steps
dt = 0.1;
num_steps = (tf - t0) / dt;

% Arrays to store results
time_euler = zeros(1, num_steps + 1);
temp_euler = zeros(1, num_steps + 1);
time_predictor_corrector = zeros(1, num_steps + 1);
temp_predictor_corrector = zeros(1, num_steps + 1);

% Euler's method
time_euler(1) = t0;
temp_euler(1) = T0;
for i = 1:num_steps
    time_euler(i + 1) = time_euler(i) + dt;
    temp_euler(i + 1) = temp_euler(i) - k * (temp_euler(i) - Ta) * dt;
end

% Predictor-Corrector (Improved Euler) method
time_predictor_corrector(1) = t0;
temp_predictor_corrector(1) = T0;
for i = 1:num_steps
    time_predictor_corrector(i + 1) = time_predictor_corrector(i) + dt;
    % Predictor step
    predictor_temp = temp_predictor_corrector(i) - k *
 (temp_predictor_corrector(i) - Ta) * dt;
    % Corrector step
    temp_predictor_corrector(i + 1) = temp_predictor_corrector(i) - 0.5 * k *
 ((temp_predictor_corrector(i) - Ta) + (predictor_temp - Ta)) * dt;
end

% Plot results
figure;
plot(time_euler, temp_euler, 'DisplayName', "Euler's Method");
hold on;
plot(time_predictor_corrector,
 temp_predictor_corrector, 'DisplayName', "Predictor-Corrector (Improved
 Euler)");
xlabel('Time');
ylabel('Temperature (°C)');
title('Cooling of an Object');
```

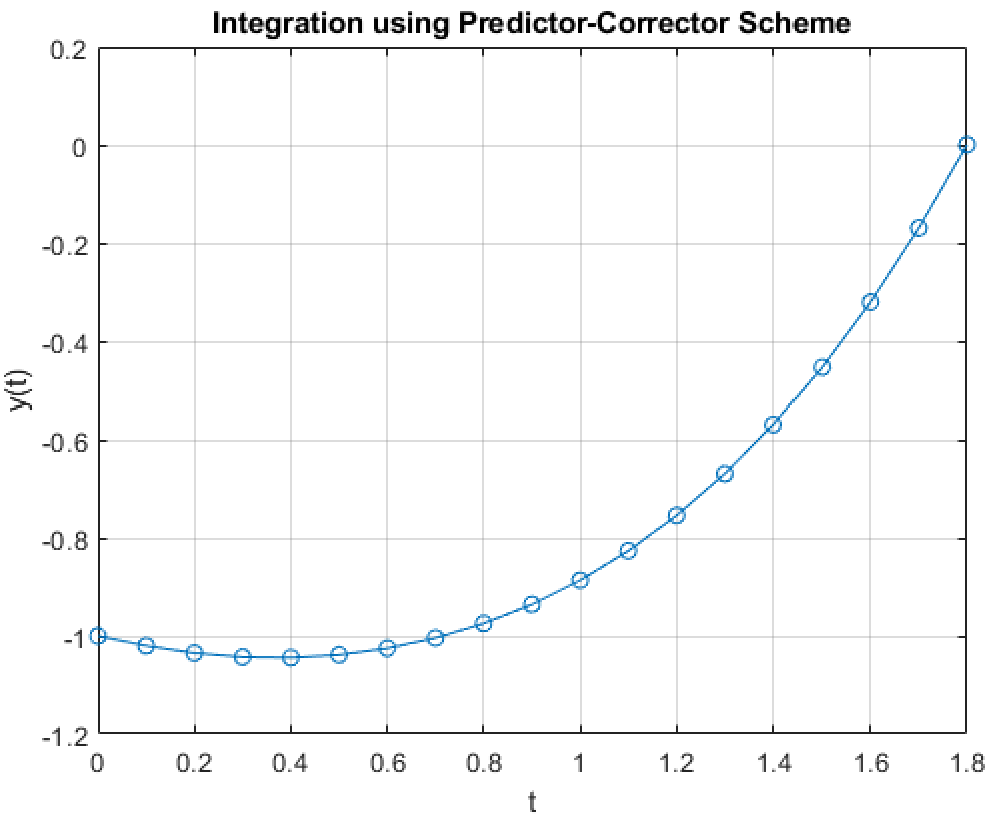
```
legend;
grid on;
hold off;

end
```



Published with MATLAB® R2023a

```
function predictor_corrector_integration()

% Compute average of TUID
TUID = [9,1,5,1,8,7,2,8,9];
alpha = mean(TUID);
fprintf('my TUID average: %.2f\n', alpha);

% Given data
t_values = 0.0:0.1:1.8;
f_values = [1.00, 0.84, 0.78, 0.73, 0.68, 0.65, 0.61, 0.58, 0.55, 0.53, 0.50,
 0.48, 0.45, 0.43, 0.41, 0.39, 0.37, 0.35, 0.33];

h = 0.1;
y = -1;  % Initial value

% Predictor-Corrector Scheme
for j = 1:length(t_values)-1
    t = t_values(j);
    % Predictor
    y_star = y(end) + h * dydt(t, y(end), f_values(j), alpha);
    % Corrector
    y_next = y(end) + (h/2) * (dydt(t, y(end), f_values(j), alpha) + dydt(t+h,
 y_star, f_values(j+1), alpha));
    y = [y, y_next];
end

% Plotting
figure;
plot(t_values, y, '-o');
xlabel('t');
ylabel('y(t)');
title('Integration using Predictor-Corrector Scheme');
grid on;

% Estimate the minimum value of y
[min_value, index] = min(y);
t_min = t_values(index);

fprintf('The minimum value of y is approximately %.4f at t = %.1f\n',
 min_value, t_min);

    function dy = dydt(t, y, f_t, alpha)
        dy = y / (y * cos(t / 3) + alpha * f_t) + t;
    end

end
```

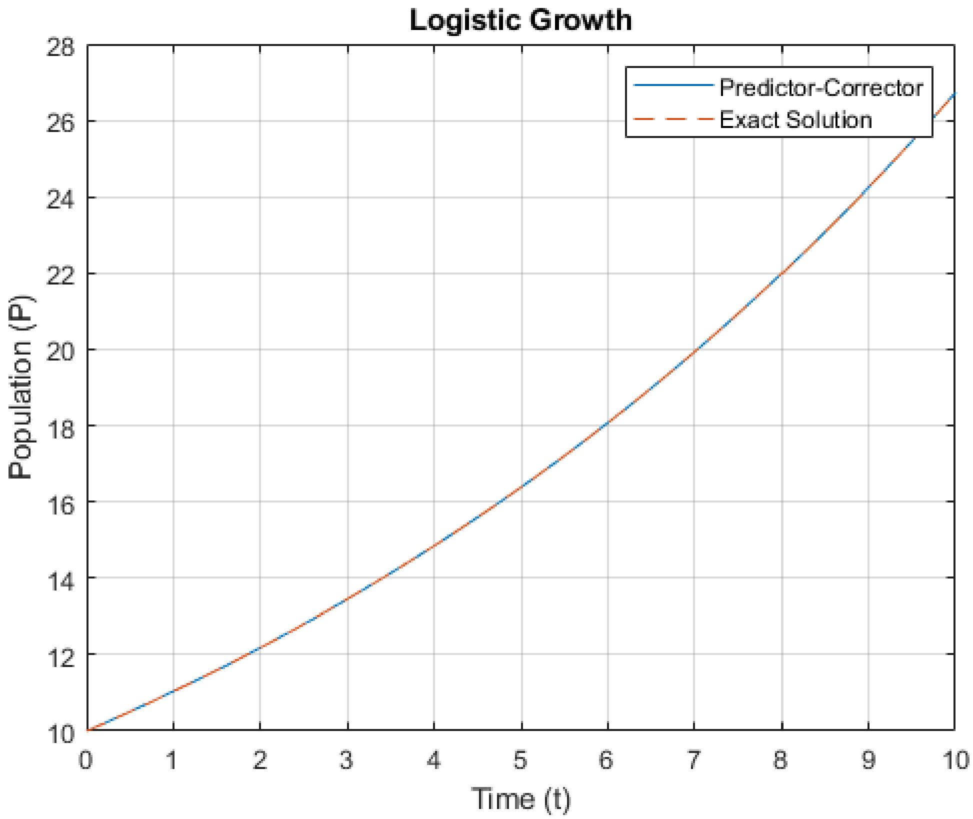
```
my TUID average: 5.56
The minimum value of y is approximately -1.0441 at t = 0.4
```



Published with MATLAB® R2023a

```
function logistic_growth_plot()

    % Parameters
    t0 = 0.0;
    P0 = 10.0;
    h = 0.1;
    N = 100;
    r = 0.1;
    K = 1000;

    [t_vals, P_vals] = predictor_corrector(P0, t0, h, N, r, K);

    % Visualization
    figure;
    plot(t_vals, P_vals, 'DisplayName', 'Predictor-Corrector');
    hold on;
    plot(t_vals, arrayfun(@(t) exact_solution(t, P0, r, K),
 t_vals), '--', 'DisplayName', 'Exact Solution');
    xlabel('Time (t)');
    ylabel('Population (P)');
    legend();
    title("Logistic Growth");
    grid on;

    function dpdt = logistic_growth(t, P, r, K)
        dpdt = r * P * (1 - P / K);
    end

    function P = exact_solution(t, P0, r, K)
        P = (K * P0 * exp(r * t)) / (K + P0 * (exp(r * t) - 1));
    end

    function [t, P] = predictor_corrector(y0, t0, h, N, r, K)
        t = t0;
        P = y0;

        % Bootstrap using 4th order Runge-Kutta
        for i = 1
            k1 = h * logistic_growth(t(end), P(end), r, K);
            k2 = h * logistic_growth(t(end) + 0.5 * h, P(end) + 0.5 * k1, r,
 K);
            k3 = h * logistic_growth(t(end) + 0.5 * h, P(end) + 0.5 * k2, r,
 K);
            k4 = h * logistic_growth(t(end) + h, P(end) + k3, r, K);

            P(end+1) = P(end) + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
            t(end+1) = t(end) + h;
        end

        for i = 2:N
            % Predictor
```

```
            P_predict = P(end) + h * (1.5 * logistic_growth(t(end), P(end), r,
 K) - 0.5 * logistic_growth(t(end-1), P(end-1), r, K));
            t_predict = t(end) + h;

            % Corrector
            P_correct = P(end) + h/2 * (logistic_growth(t(end), P(end), r, K)
 + logistic_growth(t_predict, P_predict, r, K));

            P(end+1) = P_correct;
            t(end+1) = t_predict;
        end
    end

end
```

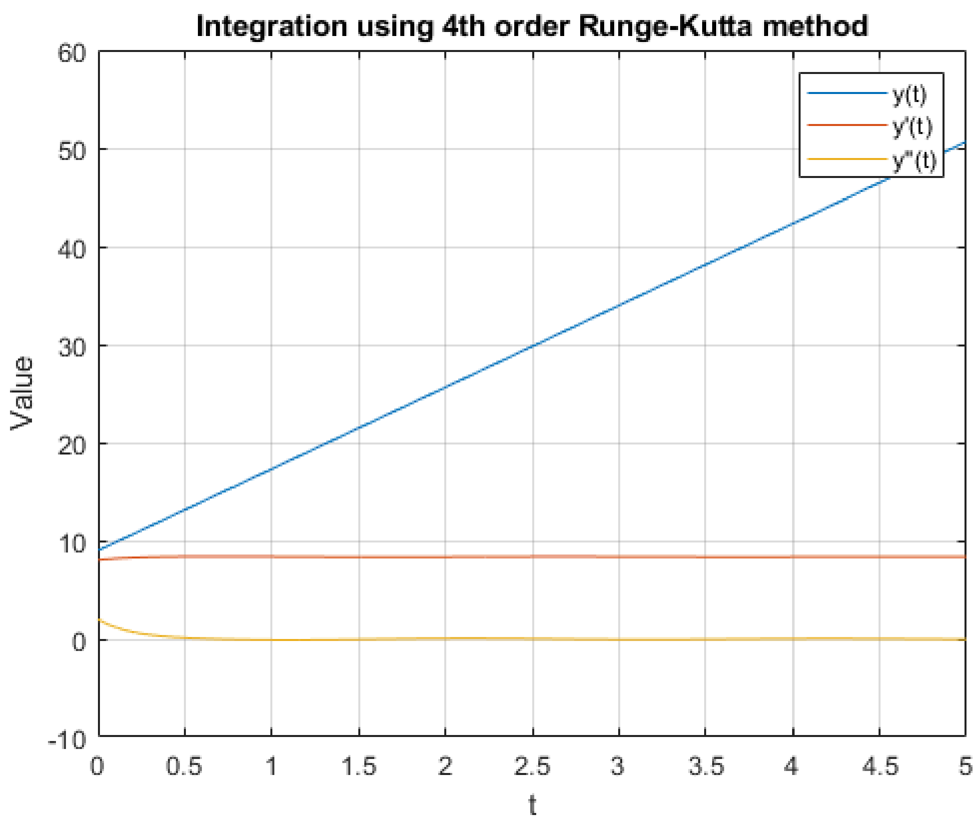


Published with MATLAB® R2023a

```
function runge_kutta_plot()

    TUID = [9, 1, 5, 1, 8, 7, 2, 8, 9];
    LETTER_MAP = {'I', 'H', 'G', 'F', 'E', 'D', 'C', 'B', 'A'};

    total_sum = sum(TUID);
    average = total_sum / length(TUID);

    fprintf('my TUID average: %.2f\n', average);

    % Create a map (in MATLAB, we use containers.Map) to map letters to
 integers
    letter_to_int_map = containers.Map(LETTER_MAP, TUID);

    % Calculate the averages of the letters of interest
    alpha_letters = {'A', 'B', 'C'};
    beta_letters = {'D', 'E', 'F'};
    gamma_letters = {'G', 'H', 'I'};

    alpha = mean(cellfun(@(x) letter_to_int_map(x), alpha_letters)) / 10;
    beta = mean(cellfun(@(x) letter_to_int_map(x), beta_letters)) / 10;
    gamma = mean(cellfun(@(x) letter_to_int_map(x), gamma_letters)) / 10;

    % Parameters
    T = 5;
    h = 0.01;

    % From TUID letter mapping
    A = 9;
    B = 8;
    C = 2;

    u0 = [A, B, C];

    [t_values, u_values] = runge_kutta_4th_order(h, T, u0, alpha, beta);

    figure;
    plot(t_values, u_values(1,:), 'DisplayName', 'y(t)');
    hold on;
    plot(t_values, u_values(2,:), 'DisplayName', "y'(t)");
    plot(t_values, u_values(3,:), 'DisplayName', "y''(t)");
    legend();
    xlabel('t');
    ylabel('Value');
    title('Integration using 4th order Runge-Kutta method');
    grid on;

    function [t_values, u_values] = runge_kutta_4th_order(h, T, u0, alpha,
 beta)
        % The system of ODEs
        f = @(t, u) [u(2); u(3); cos(3*t) - alpha*u(3) - beta*u(1)*u(3)];
```

```
        t_values = 0:h:T;
        u_values = zeros(3, length(t_values));
        u_values(:, 1) = u0;

        for i = 1:length(t_values)-1
            u = u_values(:, i);

            k1 = h * f(t_values(i), u);
            k2 = h * f(t_values(i) + 0.5*h, u + 0.5*k1);
            k3 = h * f(t_values(i) + 0.5*h, u + 0.5*k2);
            k4 = h * f(t_values(i) + h, u + k3);

            u_values(:, i+1) = u + (1/6) * (k1 + 2*k2 + 2*k3 + k4);
        end
    end
end
```

my TUID average: 5.56



Published with MATLAB® R2023a