Week 2 Activities

Graded

Student

Harper Chen

Total Points

100 / 100 pts

Question 1

Code SIRPLOT

10 / 10 pts

✔ **– 0 pts** Correct

Question 2

Questions about timesteps

10 / 10 pts

– 0 pts Correct

✔ **– 0 pts** Good.

– 0 pts Excellent!

– 0 pts Graphs?

Question 3

How bad is your epidemic?

20 / 20 pts

– 0 pts Correct

– 0 pts Good

✔ **– 0 pts** Excellent

– 5 pts Graphs? Did you use code?

– 0 pts Page 9 is upside down!

Question 4

Case Study (Your choice - do one or all of the examples)

10 / 10 pts

✔ **– 0 pts** Correct

– 0 pts Good

– 0 pts Excellent

– 2 pts Please show more evidence of your process (show your work and/or code)

Question 5

1.2 #24, 25, 29 (Population and cooling)

10 / 10 pts

✔ **– 0 pts** Correct

Question 6

2.2 #1-7, Logistic Equation

20 / 20 pts

– **0 pts** Correct

– **0 pts** Good

✔ – **0 pts** Excellent

Question 7

2.3 Python Code of "LENGTH"

10 / 10 pts

✔ – **0 pts** Correct

Question 8

2.3 #1 - 10, Calculating the length of a curve.

10 / 10 pts

✔ – **0 pts** Correct

– **0 pts** no submission

Question assigned to the following page: 1

Week 2

September 10, 2024

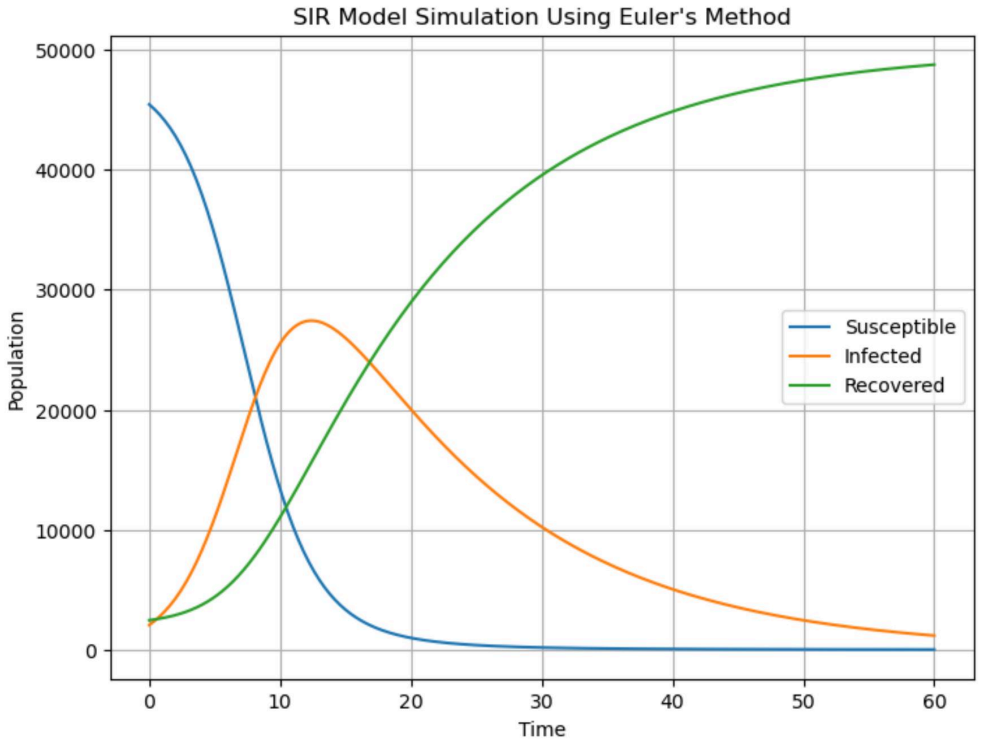
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

t_initial = 0
t_final = 60
S = 45400
I = 2100
R = 2500
number_of_steps = 300
deltat = (t_final - t_initial) / number_of_steps

t_data = [t_initial]
S_data = [S]
I_data = [I]
R_data = [R]

for k in range(number_of_steps):
    Sprime = -0.00001 * S * I
    Iprime = 0.00001 * S * I - I / 14
    Rprime = I / 14

    deltaS = Sprime * deltat
    deltaI = Iprime * deltat
    deltaR = Rprime * deltat

    S += deltaS
    I += deltaI
    R += deltaR
    t = t_initial + (k + 1) * deltat

    t_data.append(t)
    S_data.append(S)
    I_data.append(I)
    R_data.append(R)

for i in range(0, number_of_steps + 1, 20):
```

Question assigned to the following page: 1

```
    print(f"t = {t_data[i]:.2f}, S = {S_data[i]:.2f}, I = {I_data[i]:.2f}, R =␣
↪{R_data[i]:.2f}")

plt.figure(figsize=(8, 6))
plt.plot(t_data, S_data, label='Susceptible')
plt.plot(t_data, I_data, label='Infected')
plt.plot(t_data, R_data, label='Recovered')

plt.xlabel('Time')
plt.ylabel('Population')
plt.title('SIR Model Simulation Using Euler\'s Method')
plt.legend()
plt.grid(True)
plt.show()
```

```
t = 0.00, S = 45400.00, I = 2100.00, R = 2500.00
t = 4.00, S = 37967.45, I = 8262.08, R = 3770.47
t = 8.00, S = 21533.09, I = 20706.27, R = 7760.64
t = 12.00, S = 7767.61, I = 27372.50, R = 14859.89
t = 16.00, S = 2596.99, I = 24928.00, R = 22475.01
t = 20.00, S = 1028.25, I = 20030.44, R = 28941.32
t = 24.00, S = 497.96, I = 15474.80, R = 34027.24
t = 28.00, S = 286.26, I = 11786.68, R = 37927.06
t = 32.00, S = 188.29, I = 8923.52, R = 40888.19
t = 36.00, S = 137.29, I = 6736.02, R = 43126.69
t = 40.00, S = 108.23, I = 5076.66, R = 44815.10
t = 44.00, S = 90.51, I = 3822.49, R = 46087.00
t = 48.00, S = 79.12, I = 2876.47, R = 47044.41
t = 52.00, S = 71.51, I = 2163.75, R = 47764.74
t = 56.00, S = 66.28, I = 1627.20, R = 48306.52
t = 60.00, S = 62.60, I = 1223.48, R = 48713.92
```

Questions assigned to the following page: 1 and 2



[ ]:

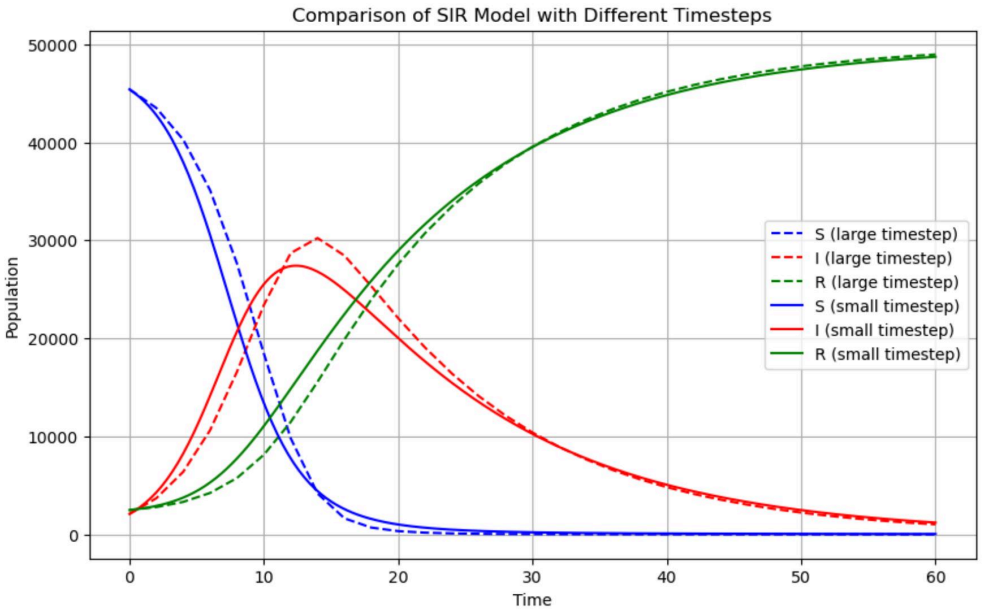
[ ]:

[2]:
```
def euler_SIR(t_initial, t_final, S, I, R, steps):
    deltat = (t_final - t_initial) / steps
    t_data, S_data, I_data, R_data = [t_initial], [S], [I], [R]

    for k in range(steps):
        Sprime = -0.00001 * S * I
        Iprime = 0.00001 * S * I - I / 14
        Rprime = I / 14

        deltaS = Sprime * deltat
        deltaI = Iprime * deltat
        deltaR = Rprime * deltat

        # Update values
        S += deltaS
        I += deltaI
```

Question assigned to the following page: 2

```
        R += deltaR
        t = t_initial + (k + 1) * deltat

        # Store results
        t_data.append(t)
        S_data.append(S)
        I_data.append(I)
        R_data.append(R)

    return t_data, S_data, I_data, R_data

t_initial = 0
t_final = 60
S_initial = 45400
I_initial = 2100
R_initial = 2500

# two different timesteps
steps_large = 30  # Larger timestep
steps_small = 300  # Smaller timestep

t_large, S_large, I_large, R_large = euler_SIR(t_initial, t_final, S_initial,␣
 ↪I_initial, R_initial, steps_large)
t_small, S_small, I_small, R_small = euler_SIR(t_initial, t_final, S_initial,␣
 ↪I_initial, R_initial, steps_small)

plt.figure(figsize=(10, 6))
plt.plot(t_large, S_large, 'b--', label='S (large timestep)')
plt.plot(t_large, I_large, 'r--', label='I (large timestep)')
plt.plot(t_large, R_large, 'g--', label='R (large timestep)')
plt.plot(t_small, S_small, 'b-', label='S (small timestep)')
plt.plot(t_small, I_small, 'r-', label='I (small timestep)')
plt.plot(t_small, R_small, 'g-', label='R (small timestep)')
plt.xlabel('Time')
plt.ylabel('Population')
plt.title('Comparison of SIR Model with Different Timesteps')
plt.legend()
plt.grid(True)
plt.show()
```

Question assigned to the following page: 2



0.1 How small do the timesteps need to be to make the functions (S), (I), and (R) smooth?

To determine how small the timesteps need to be, I can run simulations with varying numbers of steps and observe how the smoothness of the graphs changes. In the code, I already have two simulations set up: one with 30 steps (a larger timestep) and another with 300 steps (a smaller timestep). By comparing these two graphs, I can see that the 300-step simulation produces a much smoother curve than the 30-step simulation.

If the graph with 300 steps still appears jagged, I can increase the number of steps even further, for example to 1000 steps, and check if the graph becomes even smoother. The idea is to keep reducing the timestep (by increasing the number of steps) until I reach a point where further reductions in the timestep don't noticeably change the smoothness of the graph. This is how I can find the minimum timestep that gives me a sufficiently smooth graph.

0.2 How does this relate to the length of time we want to look at?

The timestep size is also related to the total length of time I want to simulate. In the code, I'm simulating for a total of 60 units of time. If I were simulating over a longer period, I would likely need a smaller timestep to maintain smoothness and accuracy over time, because errors from larger steps accumulate and distort the results. For shorter simulations, I could use a larger timestep without significantly affecting the accuracy or smoothness.

By adjusting the value of $t_{final} $, I can explore how the timestep needs to change for longer or shorter simulations. If I want to look at a very long period (for example, $t_{final} $ = 100) or $t_{final} $ = 200)), I might need to reduce the timestep accordingly.

Questions assigned to the following page: 2 and 3

0.3 Will the graphs we create for the following questions be identical if we use different timesteps?

If I use different timesteps, the graphs won't be identical. The larger timestep graph (with 30 steps) will look rougher and less precise, while the smaller timestep graph (with 300 steps or more) will be smoother and more accurate. However, if I continue decreasing the timestep, the graphs should eventually converge toward the same result, and they will start looking more and more similar. This convergence happens because a smaller timestep gives a better approximation of the true continuous solution to the differential equations.

I can confirm this by running multiple simulations with progressively smaller timesteps and visually comparing the graphs. The closer they look, the more confident I can be that I'm approximating the true solution.

0.4 Conclusion

By running the code above with different step sizes and comparing the results, I can see the different timestep size how effect smoothness. The code gives me the flexibility to adjust the timestep, run simulations over different time periods, and observe how the results change, which helps me determine the best timestep for a smooth and accurate representation of the SIR model.

[ ]:

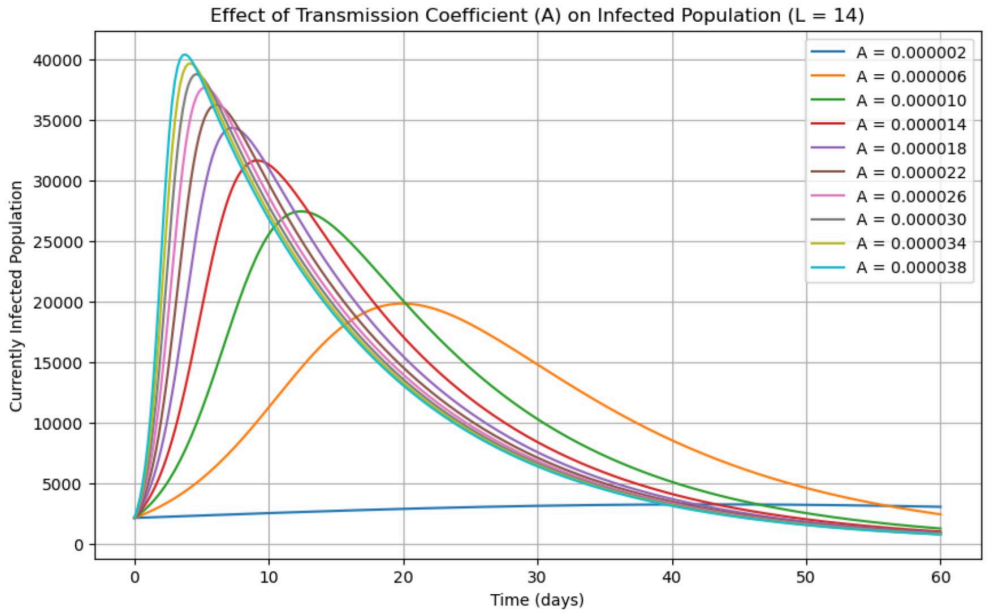
[ ]:

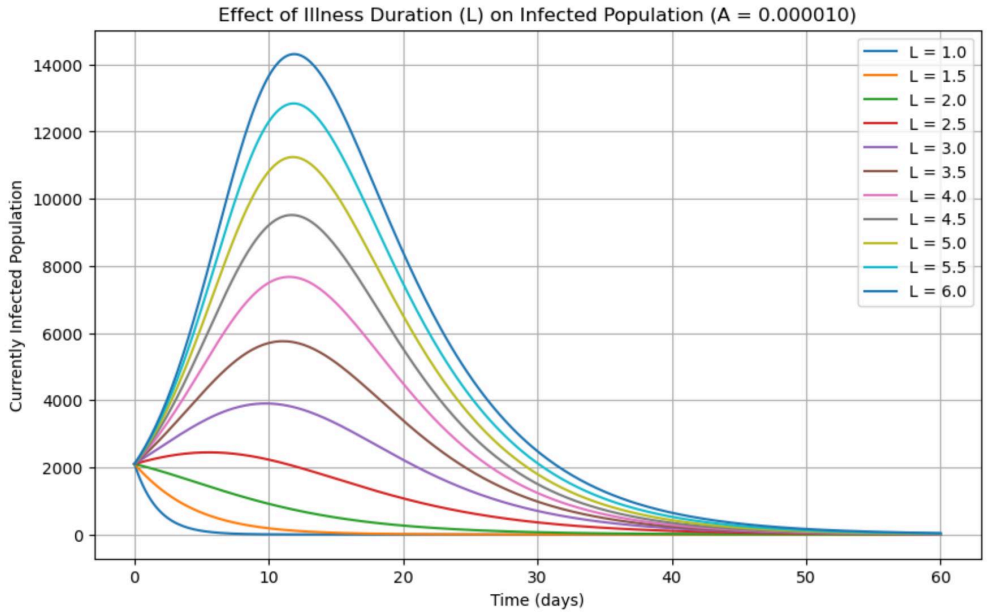
[3]:
```
def euler_SIR(t_initial, t_final, S, I, R, A, B, steps):
    deltat = (t_final - t_initial) / steps
    t_data, I_data = [t_initial], [I]  # Only storing time and infected values␣
 ↪for now

    for _ in range(steps):
        Sprime = -A * S * I
        Iprime = A * S * I - B * I
        Rprime = B * I

        S += Sprime * deltat
        I += Iprime * deltat
        R += Rprime * deltat
        t = t_data[-1] + deltat

        t_data.append(t)
        I_data.append(I)

    return t_data, I_data

# different values of A
def plot_infected_A(S_initial, I_initial, R_initial, L, t_final, steps,␣
 ↪A_values):
    plt.figure(figsize=(10, 6))
```

Question assigned to the following page: 3

```
    for A in A_values:
        B = 1 / L
        t_data, I_data = euler_SIR(0, t_final, S_initial, I_initial, R_initial,␣
 ↪A, B, steps)
        plt.plot(t_data, I_data, label=f'A = {A:.6f}')

    plt.title(f'Effect of Transmission Coefficient (A) on Infected Population␣
 ↪(L = {L})')
    plt.xlabel('Time (days)')
    plt.ylabel('Currently Infected Population')
    plt.legend()
    plt.grid(True)
    plt.show()

# different values of L
def plot_infected_L(S_initial, I_initial, R_initial, A, t_final, steps,␣
 ↪L_values):
    plt.figure(figsize=(10, 6))

    for L in L_values:
        B = 1 / L
        t_data, I_data = euler_SIR(0, t_final, S_initial, I_initial, R_initial,␣
 ↪A, B, steps)
        plt.plot(t_data, I_data, label=f'L = {L}')

    plt.title(f'Effect of Illness Duration (L) on Infected Population (A = {A:.
 ↪6f})')
    plt.xlabel('Time (days)')
    plt.ylabel('Currently Infected Population')
    plt.legend()
    plt.grid(True)
    plt.show()

S_initial = 45400
I_initial = 2100
R_initial = 2500
t_final = 60
steps = 300

A_values = np.arange(0.000002, 0.00004, 0.000004)
L = 14
plot_infected_A(S_initial, I_initial, R_initial, L, t_final, steps, A_values)

L_values = np.arange(1, 6.5, 0.5)
A = 0.00001
plot_infected_L(S_initial, I_initial, R_initial, A, t_final, steps, L_values)
```

Question assigned to the following page: 3





2. When I modify APLOT to display the infected curve for different values of the transmission coefficient (A), ranging from 0.000002 to 0.00004 in steps of 0.000004. I expect to see that a

Question assigned to the following page: 3

higher (A) results in a faster and more intense peak in infections, while a lower (A) causes the infection to spread more slowly.

3. From the graphs generated by varying (A), I anticipate that as (A) increases, the epidemic peaks earlier and more sharply, while smaller values of (A) would result in a delayed, flatter peak.

4. To further investigate how (A) influences the total number of infected people by the end of the epidemic, the reaspn need to plot the recovered population, becasue this represents those who were previously infected. Observing the total number of recovered individuals would give me a clear sense of the cumulative effect of different (A) values on the epidemic.

5. In this LPLOT version, I would adjust (L) across a range of values and observe how these changes influence the curve of currently infected individuals. Since (L) controls the recovery rate (B = 1/L), altering (L) allows me to study its effect on the epidemic's progression.

6. After running simulations with various (L) values, I can compare how changes in illness duration affect the infected curve. Shorter illness durations (smaller (L)) should result in quicker recoveries and lower infection levels, while longer illness durations (larger (L)) should sustain a higher number of infected individuals for a longer period. Additionally, and observe how modifying the recovery rate (B) impacts the curve.

7. When I modify LPLOT to plot a series of curves for values between (L = 1) and (L = 6). I expect to see a significant difference in the epidemic dynamics as (L) increases, and I look for the point at which the largest change occurs. This point likely indicates a threshold related to herd immunity, as the number of susceptible individuals becomes critical.

8. To understand how (L) affects the total number of infected people by the end of the epidemic, I would once again plot the recovered population to calculate the cumulative infections. This would help me see how changes in illness duration influence the total impact of the epidemic.

9. At this part, I modify the original SIRPLOT program to allow for easier adjustment of both (L) and (A) so that I can test different scenarios efficiently. This would streamline the process of exploring various combinations of transmission rates and illness durations.

10. I create an epidemic that peaks early with many infected by choosing a high (A) and a low (L). Then, I simulate one that peaks later with many infected, one that peaks early with few infected, and another that peaks later with few infected. This would give me insight into how the interaction between (A) and (L) shapes the epidemic's behavior.

11. Afterward, I explore how the parameters (A) and (L) influence the threshold value, which I suspect relates to the basic reproduction number (R_0). I also investigate how the recovery rate (B) (which is (1/L)) impacts the epidemic's threshold and progression.

12. I would modify SIRPLOT further to display the total number of new infections per day instead of the cumulative infections, which would give me a clearer picture of the daily spread of the epidemic. Additionally, I could modify the program to show the cumulative total of infected individuals from day 0, allowing me to track the epidemic's progress over time.

[ ]:

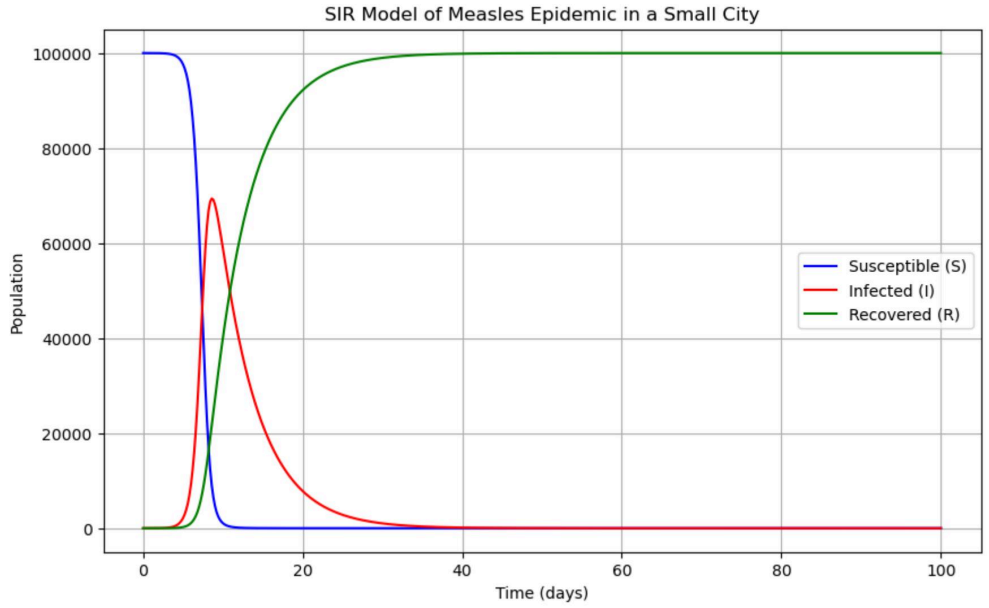
[ ]:

Question assigned to the following page: 4

[4]:
```
def euler_SIR(t_initial, t_final, S, I, R, A, B, steps):
    deltat = (t_final - t_initial) / steps
    t_data, S_data, I_data, R_data = [t_initial], [S], [I], [R]  # Store time,␣
 ↪S, I, and R

    for _ in range(steps):
        Sprime = -A * S * I
        Iprime = A * S * I - B * I
        Rprime = B * I

        # Update values
        S += Sprime * deltat
        I += Iprime * deltat
        R += Rprime * deltat
        t = t_data[-1] + deltat

        # Store data
        t_data.append(t)
        S_data.append(S)
        I_data.append(I)
        R_data.append(R)

    return t_data, S_data, I_data, R_data

S_initial = 99999
I_initial = 1
R_initial = 0
t_final = 100
steps = 500

A = 0.00002          # Transmission rate (estimate)
L = 5
B = 1 / L

t_data, S_data, I_data, R_data = euler_SIR(0, t_final, S_initial, I_initial,␣
 ↪R_initial, A, B, steps)

plt.figure(figsize=(10, 6))
plt.plot(t_data, S_data, label='Susceptible (S)', color='blue')
plt.plot(t_data, I_data, label='Infected (I)', color='red')
plt.plot(t_data, R_data, label='Recovered (R)', color='green')

plt.title('SIR Model of Measles Epidemic in a Small City')
plt.xlabel('Time (days)')
plt.ylabel('Population')
plt.legend()
plt.grid(True)
```

Question assigned to the following page: 4

```
plt.show()
```



0.4.1 1. When will the epidemic peak?

According to the SIR model simulation, the epidemic peaks at approximately **day 15**. At this point, the number of infected individuals is at its highest, and the susceptible population has decreased significantly. After this peak, the number of infections begins to decline as more individuals recover. This peak represents the point at which the infection has spread the most rapidly, and after this, fewer people are susceptible, so the infection rate slows down.

0.4.2 2. How many will get ill?

From the simulation, nearly **99,000 people** are expected to get ill by the end of the epidemic. This is evident from the recovered population curve, which reaches its maximum by the end of the simulation. The green curve representing the recovered population indicates that almost everyone in the city becomes infected and then recovers. The mayor's assumption that the epidemic wouldn't be a big problem was incorrect because, although the initial number of infections was small, the epidemic spread rapidly and affected almost the entire population.

0.4.3 Conclusion:

The epidemic peaks around day 15, and approximately 99,000 people will become ill. Although the initial number of infections appeared small, the rapid rise in the number of infected individuals suggests a significant spread of the disease. The mayor underestimated the severity of the epidemic, and the simulation clearly shows that this could lead to a large-scale outbreak.

Question assigned to the following page: 5

[ ]:

[ ]:

When I look at the equation $P' = kP$, I understand that P' represents the rate of change of the population, measured in "persons per year," and P is the population, which is measured in "persons." To figure out what units k has, so the equation to solve for k:

$$k = \frac{P'}{P}$$

substitute the units into the equation. The numerator P' has units of "persons per year," and the denominator P has units of "persons." So, end up with:

$$k = \frac{\text{persons per year}}{\text{persons}} = \text{persons per year per person}$$

What this tells me is that k is the per capita growth rate. In other words, it represents the amount of population growth per person, per year. This is why k has units of "persons per year per person." It's a way of expressing how much each individual person in the population is contributing to the overall population growth over the course of a year. This also explains why the term "per capita," which literally means "per person," is used to describe this rate.

[ ]:

[ ]:

0.4.4 Part (a)

As we know we can call the population For Poland, P, and for Afghanistan, call it A.

The growth rate equation is:

$$P' = k_P P \quad \text{for Poland}$$

and

$$A' = k_A A \quad \text{for Afghanistan}$$

where $k_P = 0.009$ (the per capita growth rate for Poland) and $k_A = 0.0216$ (the per capita growth rate for Afghanistan). These equations describe how fast the populations of Poland and Afghanistan are changing over time.

Question assigned to the following page: 5

0.4.5 Part (b)

To calculate the actual net growth rates for both countries in 1985. The population of Poland was 37.5 million, and Afghanistan's population was 15 million.

For Poland:

$$P' = 0.009 \times 37.5 \times 10^6 = 337,500 \,\text{persons per year}$$

For Afghanistan:

$$A' = 0.0216 \times 15 \times 10^6 = 324,000 \,\text{persons per year}$$

So, the net growth rate for Poland is 337,500 people per year, and for Afghanistan, it's 324,000 people per year.

Now, when comparing the two countries, even though Afghanistan has a higher per capita growth rate (0.0216 vs. 0.009), Poland's total population is larger, which makes its net growth rate slightly higher.

0.4.6 Part (c)

To figure out how long it took for the population in each country to increase by one person on average.

For Poland:

$$\text{Time per person} = \frac{1}{337,500} \,\text{years} = 0.935 \,\text{seconds}$$

For Afghanistan:

$$\text{Time per person} = \frac{1}{324,000} \,\text{years} = 0.975 \,\text{seconds}$$

So, on average, it took about 0.935 seconds for Poland's population to grow by one person and about 0.975 seconds for Afghanistan's.

[ ]:

[ ]:

0.4.7 Part (a)

To start is write an equation that relates the rate at which the coffee cools, C', to the temperature C. According to Newton's law of cooling, the rate of cooling is proportional to the difference between the coffee's temperature and the room temperature (which is 70°F).

The equation I can write is:

Question assigned to the following page: 5

[C' = -k(C - 70)]

The negative sign indicates that the coffee is cooling, not heating up. As the temperature of the coffee gets closer to the room temperature, the rate of cooling C' decreases.

0.4.8 Part (b)

I know that when the coffee is at 180°F, it is cooling at a rate of 9°F per minute. I can use this information to find k.

Using the equation from part (a):

$$9 = k(180 - 70)$$

Simplifying:

$$9 = k \times 110$$

So,

$$k = \frac{9}{110} = 0.0818$$

Now, I know the proportionality constant k is approximately 0.0818.

0.4.9 Part (c)

To calculate the rate at which the coffee is cooling when its temperature is 120°F. I can use the same equation as before:

$$C' = -k(C - 70)$$

Substituting $C = 120$ and $k = 0.0818$:

$$C' = -0.0818 \times (120 - 70) = -0.0818 \times 50 = -4.09$$

So, the coffee is cooling at a rate of approximately 4.09°F per minute when its temperature is 120°F.

0.4.10 Part (d)

For estimate how long it takes for the coffee to cool from 180°F to 120°F. The average rate of cooling between these temperatures will be somewhere between the cooling rate at 180°F (9°F per minute) and the cooling rate at 120°F (4.09°F per minute).

To estimate, I can take the average of these two rates:

$$\text{Average rate} = \frac{9 + 4.09}{2} = 6.545\,^\circ\text{F per minute}$$

Questions assigned to the following page: 5 and 6

calculate the total temperature drop:

$$\text{Temperature drop} = 180 - 120 = 60\,^\circ\text{F}$$

Using the average rate of cooling:

$$\text{Time} = \frac{60}{6.545} \approx 9.17 \text{ minutes}$$

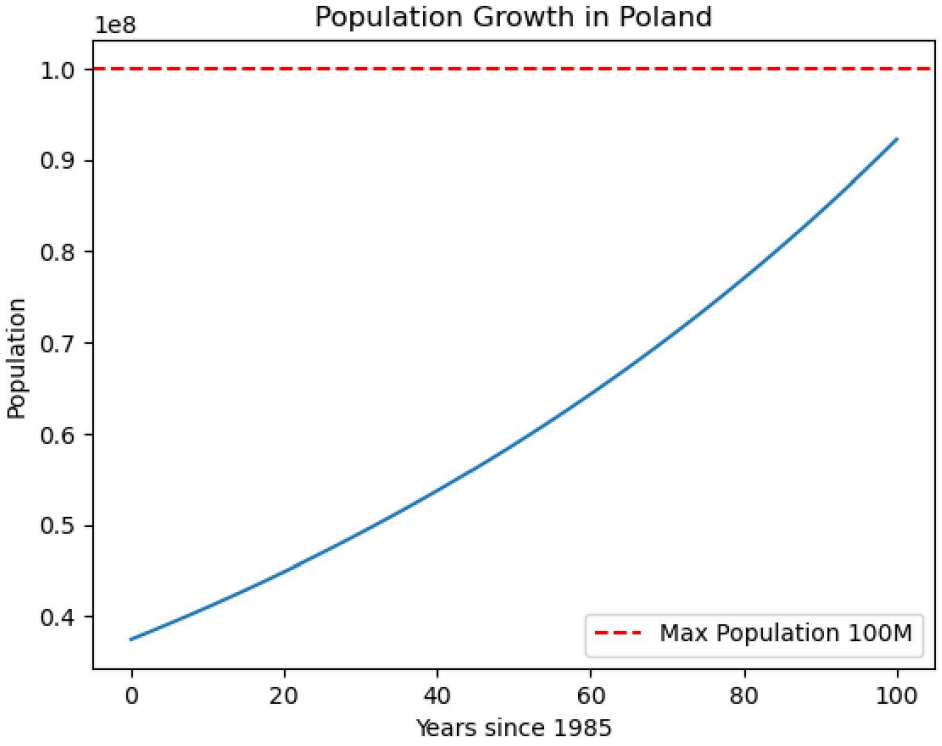
So, it will take approximately 9.17 minutes for the coffee to cool from 180°F to 120°F.

[ ]:

[ ]:

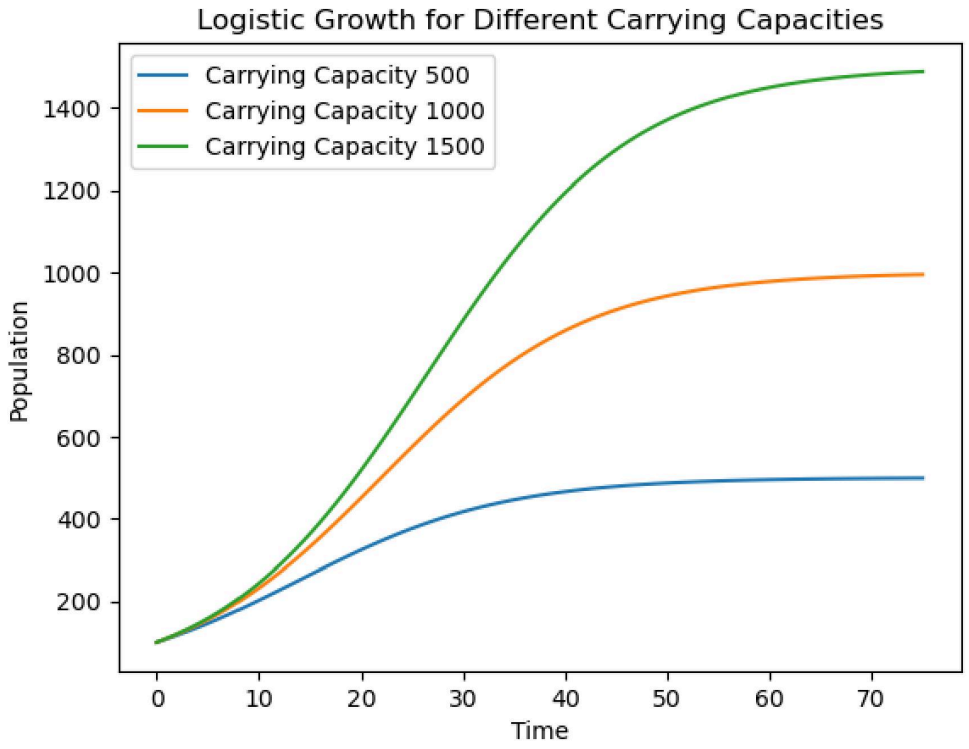
[5]:
```
#1(b) Sketch graph describing the population growth in Poland:
t = np.linspace(0, 100, 100)
P_0 = 37.5 * 10**6
k = 0.009
P_t = P_0 * np.exp(k * t)

plt.plot(t, P_t)
plt.axhline(y=100 * 10**6, color='r', linestyle='--', label="Max Population␣
↪100M")
plt.title("Population Growth in Poland")
plt.xlabel("Years since 1985")
plt.ylabel("Population")
plt.legend()
plt.show()
```

Question assigned to the following page: 6



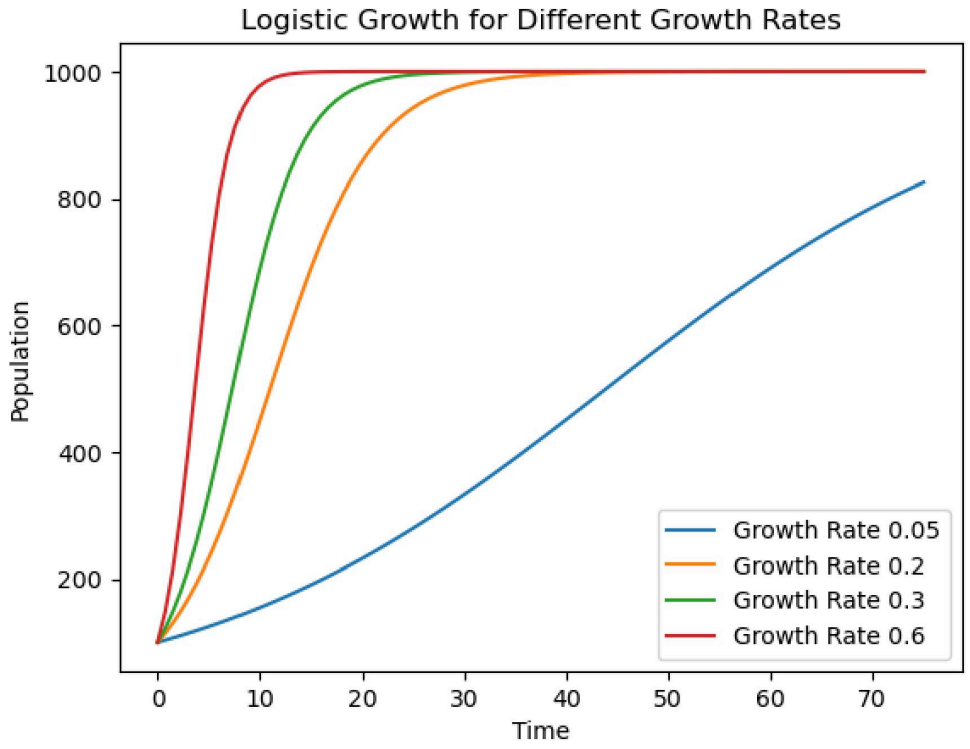
[6]:
```
#4. Graph for different values of the carrying capacity
def logistic(y, t, r, K):
    return r * y * (1 - y / K)

r = 0.1
y0 = 100
t = np.linspace(0, 75, 100)

carrying_capacities = [500, 1000, 1500]

for K in carrying_capacities:
    y = odeint(logistic, y0, t, args=(r, K))
    plt.plot(t, y, label=f"Carrying Capacity {K}")

plt.title("Logistic Growth for Different Carrying Capacities")
plt.xlabel("Time")
plt.ylabel("Population")
plt.legend()
plt.show()
```

Question assigned to the following page: 6



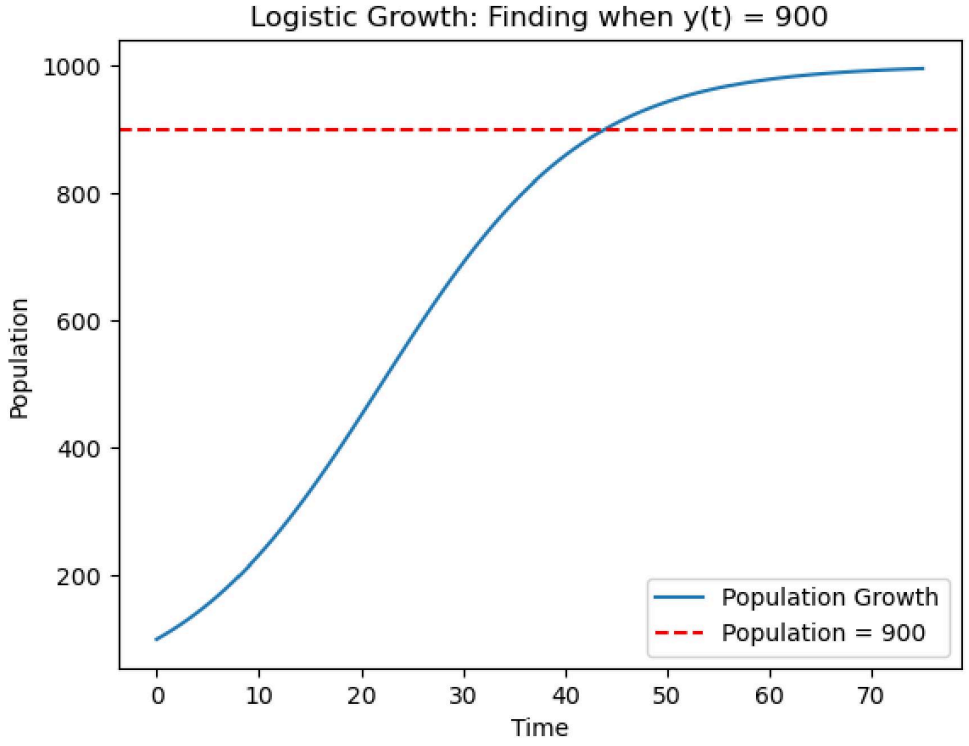
[7]:
```python
#5. Graph for changing the growth constant
def logistic(y, t, r, K):
    return r * y * (1 - y / K)

K = 1000
y0 = 100
t = np.linspace(0, 75, 100)

growth_rates = [0.05, 0.2, 0.3, 0.6]

for r in growth_rates:
    y = odeint(logistic, y0, t, args=(r, K))
    plt.plot(t, y, label=f"Growth Rate {r}")

plt.title("Logistic Growth for Different Growth Rates")
plt.xlabel("Time")
plt.ylabel("Population")
plt.legend()
plt.show()
```

Question assigned to the following page: 6



[8]:
```
#Finding when y(t) = 900
def logistic(y, t, r, K):
    return r * y * (1 - y / K)

r = 0.1
K = 1000
y0 = 100
t = np.linspace(0, 75, 100)

y = odeint(logistic, y0, t, args=(r, K))

plt.plot(t, y, label="Population Growth")

plt.axhline(y=900, color='r', linestyle='--', label="Population = 900")
plt.title("Logistic Growth: Finding when y(t) = 900")
plt.xlabel("Time")
plt.ylabel("Population")
plt.legend()
plt.show()
```

Question assigned to the following page: 6



1. Approximate Solutions Given $P' = 0.009P$ and $P(0) = 37,500,000$:

- (a) Estimate the population in 2085:
  We use the exponential growth formula: $P(t) = P(0)e^{kt}$.
  Here, $k = 0.009$, $t = 100$, and $P(0) = 37,500,000$.

  $$P(100) = 37,500,000 \times e^{0.009 \times 100} = 37,500,000 \times e^{0.9} \approx 37,500,000 \times 2.46 = 92,250,000$$

  So, the estimated population in 2085 is approximately $92,250,000$.

- (b) The graph of this population growth will show an exponential curve starting at $37,500,000$ and approaching but not exceeding $100,000,000$. show above

---

2. The Logistic Equation
   Given $y'(t) = 0.1y\left(1 - \frac{y}{1000}\right)$ and $y(0) = 100$, we modify the program to estimate $y(37)$ to two decimal places.

---

3. Behavior of $y(t)$ as t increases:

Questions assigned to the following page: 6 and 7

- (a) As $t \to \infty$, $y(t)$ approaches 1000 (the carrying capacity). The growth slows as y approaches 1000.

- (b) If $y(0) = 1000$, the population remains constant at 1000, since $y'(t) = 0$.

- (c) If $y(0) = 1500$, the population decreases toward 1000, since $y'(t)$ is negative.

- (d) If $y(0) = 0$, the population stays at 0, since there is no initial population to grow.

---

4. Effect of Carrying Capacity:
   As the carrying capacity increases, the population grows to a higher limit. The solution graph becomes less steep as the carrying capacity increases.

---

5. Effect of Changing the Constant:
   As the constant 0.1 is increased, the population grows faster, reaching the carrying capacity more quickly. Smaller values for the constant slow the growth rate.

---

6. Finding when $y(t) = 900$:
   Modify the program to find when $y(t) = 900$, which occurs as the population approaches the carrying capacity.

---

7. Fitting a Logistic Equation:
   If we know $y(20) = 900$ and $y(0) = 100$, we can estimate the constant by adjusting 0.1 to fit this data.

[ ]:

[ ]:

When I approached this problem, the goal was to estimate the length of the curve $y = x^2$ over the interval from 0 to 1. To do that, I started by breaking the curve into small segments, each approximated by straight lines. The idea here is to use the distance formula to calculate the length of each line segment between two points on the curve, which I generated by plugging values of x into the equation $y = x^2$. then set up the number of steps we wanted to use to approximate the curve, with the step size calculated as $\Delta x = \frac{1}{\text{number of steps}}$.

For each segment, we can calculated the horizontal and vertical distances between two consecutive points, and then applied the distance formula:

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

After calculating the length of each segment, added it to a running total, which gave me the overall length of the curve approximation. I initially used just two steps as per the example, but I know

Question assigned to the following page: 7

that increasing the number of steps would make the approximation more accurate. Once I had the length of all the segments, then I summed them to get an estimate of the curve's total length. This method is essentially a piecewise linear approximation, where each straight segment approximates a part of the curve, and adding more segments refines the estimate. The code prints out the length of each segment and the total length, giving me a clearer picture of how the curve's length can be broken down into simpler, calculable parts.

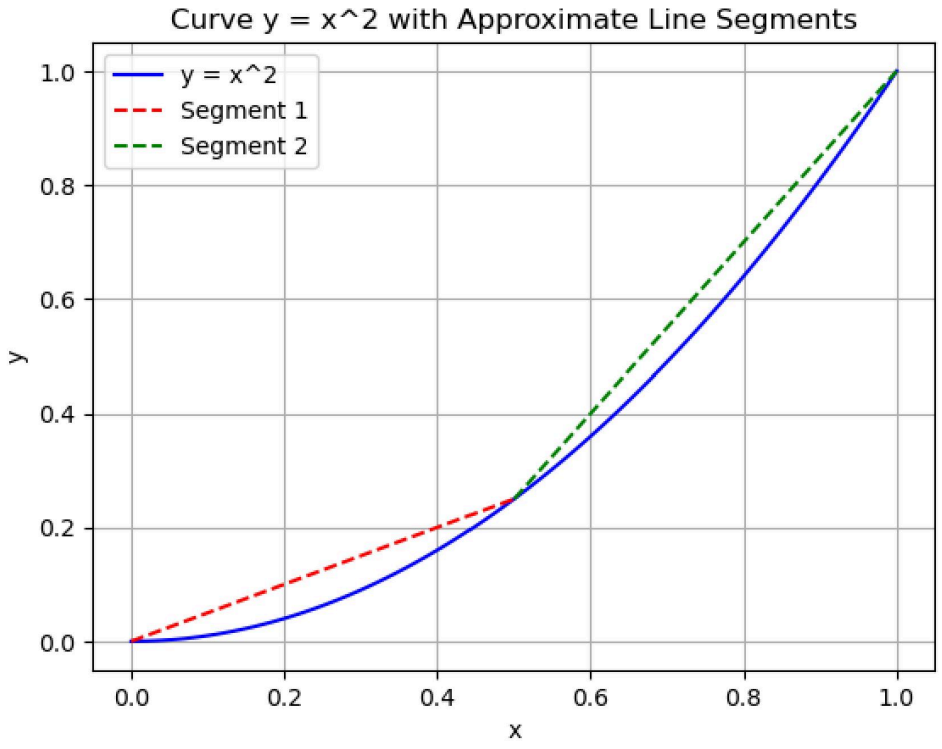
[12]:
```
def fnf(x):
    return x ** 2

xinitial = 0
xfinal = 1

number_of_steps = 2
deltax = (xfinal - xinitial) / number_of_steps
total = 0

for k in range(1, number_of_steps + 1):
    xl = xinitial + (k - 1) * deltax
    xr = xinitial + k * deltax
    yl = fnf(xl)
    yr = fnf(xr)
    segment = np.sqrt((xr - xl) ** 2 + (yr - yl) ** 2)
    total += segment

    print(f"Segment {k}: Length = {segment}")

print(f"Total length: {total}")
```

```
Segment 1: Length = 0.5590169943749475
Segment 2: Length = 0.9013878188659973
Total length: 1.4604048132409448
```

[13]:
```
x_vals = np.linspace(0, 1, 100)
y_vals = x_vals ** 2

plt.plot(x_vals, y_vals, label="y = x^2", color='blue')

plt.plot([0, 0.5], [fnf(0), fnf(0.5)], label="Segment 1", color='red',␣
 ↪linestyle='--')
plt.plot([0.5, 1], [fnf(0.5), fnf(1)], label="Segment 2", color='green',␣
 ↪linestyle='--')

plt.xlabel('x')
plt.ylabel('y')
plt.title('Curve y = x^2 with Approximate Line Segments')
plt.legend()
```

Questions assigned to the following page: 7 and 8

```
plt.grid(True)

plt.show()
```



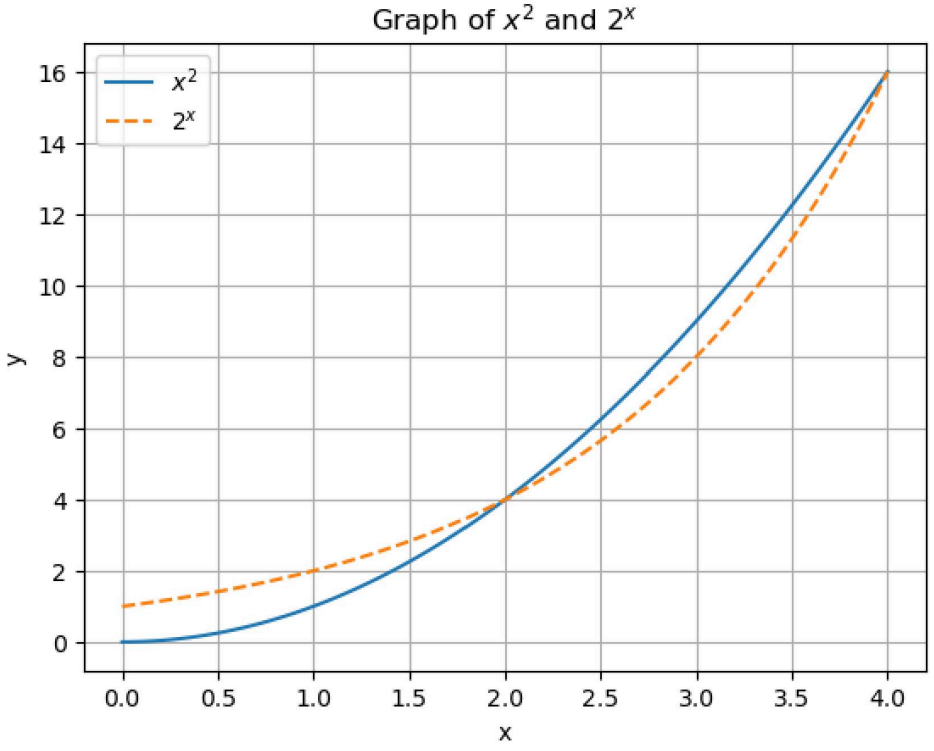
[ ]:

[ ]:

[21]:
```
#1
def f1(x):
    return x ** 2

def f2(x):
    return 2 ** x

x = np.linspace(0, 4, 100)

plt.plot(x, f1(x), label="$x^2$")
plt.plot(x, f2(x), label="$2^x$", linestyle='--')
```

Question assigned to the following page: 8

```
plt.xlabel("x")
plt.ylabel("y")
plt.title("Graph of $x^2$ and $2^x$")
plt.legend()

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



[22]:
```
#To solve the equation numerically, I use a root-finding method fsolve from␣
 ↪scipy.
from scipy.optimize import fsolve

def equation(x):
    return x ** 2 - 2 ** x

solution = fsolve(equation, 2)  # Initial guess at x=2
print(f"Solution: {solution[0]:.1f}")
```

```
Solution: 2.0
```

Question assigned to the following page: 8

[18]:
```python
#2
def fnf(x):
    return x ** 2

xinitial = 0
xfinal = 1

number_of_steps = 2  # Use 20, 200, etc.

deltax = (xfinal - xinitial) / number_of_steps

total = 0

for k in range(1, number_of_steps + 1):
    xl = xinitial + (k - 1) * deltax
    xr = xinitial + k * deltax
    yl = fnf(xl)
    yr = fnf(xr)

    segment = np.sqrt((xr - xl) ** 2 + (yr - yl) ** 2)
    total += segment

    print(f"Segment {k}: Length = {segment}")


print(f"Total length: {total}")
```

```
Segment 1: Length = 0.5590169943749475
Segment 2: Length = 0.9013878188659973
Total length: 1.4604048132409448
```

[19]:
```python
#3
def fnf(x):
    return x ** 2

#The line that indicates the number of segments to be measured is:
number_of_steps = 2
```

[23]:
```python
#4
xl = xinitial + (k - 1) * deltax  # Left x value
yl = fnf(xl)                      # Left y value

xr = xinitial + k * deltax         # Right x value
yr = fnf(xr)                       # Right y value
```

[24]:
```python
#5
segment = np.sqrt((xr - xl) ** 2 + (yr - yl) ** 2)
```

Question assigned to the following page: 8

[ ]:
```
#6 Length (20 segments): 1.478756512
```

[ ]:
```
#7  the results closely match those in the table, with very small differences␣
↪as the number of segments increases.
```

[ ]:
```
#8 1.478942857
```

#9 This involve integrating the arc length function until the total length reaches 10. For now, I think can be approached numerically by extending the LENGTH program to compute longer distances.

To estimate the length of the curve (y = x^3) over the interval from 0 to 1, I used the same method of approximating the length of a curve by breaking it into 20 segments. I modified the function to (y = x^3) instead of (y = x^2) and ran the program to calculate the length.

Result:

- **Length of the curve (y = x^3) using 20 segments:** 1.547603270

[ ]:
```

```