Exploring the Dynamics of Logistic Equations via Python

Dania Bawab

Connor Brophy

Introduction

Overview of the fundamental dynamics formula (from Calculus II)

In calculus, particularly in dynamics, Newton's second law of motion serves as a cornerstone formula, stating that an object's acceleration is directly proportional to the net force acting on it and inversely proportional to its mass, mathematically represented as $F = ma$. This relationship elucidates how forces influence the motion of objects, with acceleration often being variable, necessitating calculus for motion description through solving differential equations relating position, velocity, and acceleration over time. One pivotal differential equation derived from Newton's second law is $F(t) = m\frac{d^2x}{dt^2}$, where $F(t)$ represents the net force as a function of time, m is the object's mass, and $\frac{d^2x}{dt^2}$ denotes the second derivative of position with respect to time, representing acceleration. This equation enables the modeling of object motion under diverse forces, finding application in physics, engineering, and astronomy, facilitating predictions of object behavior under varied conditions and constraints.

Review of Robert May's work (1976)

Robert May's work in 1976 on the dynamics of logistic equations significantly advanced our understanding of complex ecological systems. By exploring the behavior of population growth using the logistic equation, May revealed the intricate dynamics that emerge when resources are limited. His analysis demonstrated how seemingly simple equations could produce a wide range of behaviors, including stable equilibria, limit cycles, and chaotic oscillations, challenging the notion that ecological systems could be fully captured by linear models. Particularly groundbreaking was May's discovery of chaotic behavior in the logistic equation, highlighting the inherent unpredictability and sensitivity to initial conditions in ecological dynamics. This discovery paved the way for further research into chaos theory in ecology, reshaping our understanding of complex systems and inspiring new approaches to modeling and prediction.

Review of Lorenz's studies (1964)

Lorenz's paper "Deterministic Nonperiodic Flow" in 1964 marked a pivotal moment in scientific understanding, particularly in the realm of chaos theory and the dynamics of complex systems. Through his investigation of simplified models of atmospheric convection, Lorenz introduced the concept of the "butterfly effect," demonstrating how small variations in initial conditions can lead to vastly different outcomes over time. His analysis of the Lorenz equations revealed a rich tapestry of dynamic behaviors, including sensitivity to initial conditions, the emergence of strange attractors, and the manifestation of chaotic oscillations. By uncovering the presence of strange attractors in phase space, Lorenz provided a visual representation of the underlying complexity and unpredictability inherent in chaotic systems. This seminal work transcended the boundaries of meteorology, influencing diverse scientific disciplines and reshaping our understanding of nonlinear dynamics and the limitations of linear models. Lorenz's

contributions continue to reverberate through scientific inquiry, serving as a cornerstone in the study of chaos and inspiring further exploration into the dynamics of complex systems.

Application of logistic equation for Pandemic Analysis (e.g., Covid-19)

The logistic equation serves as a valuable tool in epidemiology for modeling the spread of infectious diseases such as COVID-19. While it may not capture all the intricacies of real-world pandemics, it offers a framework to understand and predict disease dynamics under specific conditions. By adapting the logistic equation, researchers can model the growth of infected individuals within a population over time, with parameters like the growth rate and carrying capacity reflecting disease characteristics and population dynamics. Initial conditions, such as the number of infected individuals at the pandemic's outset, are vital for predicting its trajectory. Through calibration using data on initial spread, the logistic equation can predict future infection peaks and assess the impact of interventions like social distancing or vaccination. However, its simplicity means it may not account for all pandemic factors, like transmission rates or human behavior. Therefore, while insightful for trend analysis, caution is warranted in interpreting its predictions, which should complement other models and empirical data. Overall, the logistic equation provides valuable insights for pandemic analysis, aiding public health decision-making amid evolving challenges like COVID-19.

Review of the sigmoid function (used as an activation function in neural networks)

The sigmoid function, a fundamental component of neural networks, plays a crucial role in deep learning. Defined as $\sigma(x) = \frac{1}{1+e^{-x}}$, it maps real-valued inputs to a range between 0 and 1, embodying distinctive characteristics beneficial for various applications. Its S-shaped curve ensures a smooth and continuous transformation, introducing crucial non-linearity vital for capturing complex input-output relationships in neural networks. Particularly advantageous for binary classification tasks, the sigmoid's bounded output yields interpretable probability scores, aiding decision-making processes. Its differentiability across all inputs facilitates efficient gradient-based optimization algorithms like gradient descent, crucial for model training. However, despite its historical prominence, the sigmoid function has seen diminished usage in hidden layers due to inherent limitations. Notably, the vanishing gradient problem arises as gradients tend towards zero with large input values, impeding effective learning in deep networks. Furthermore, output saturation at extreme input values hampers parameter updates, slowing down learning. While the sigmoid function remains pivotal in logistic regression and certain classification tasks, alternative activation functions like ReLU have gained popularity for their ability to alleviate these drawbacks. Despite its diminishing usage, the sigmoid function continues to play a foundational role in neural network architectures, underscoring the ongoing quest for optimal activation functions in deep learning frameworks.

Introduction of the butterfly effect and chaotic systems

The butterfly effect and chaotic systems are foundational concepts within the realm of chaos theory, which delves into the behavior of intricate dynamical systems highly attuned to initial conditions. The

butterfly effect, first articulated by mathematician and meteorologist Edward Lorenz, symbolizes the profound impact minute alterations in starting conditions can have on complex systems over time. Lorenz's pioneering work in weather prediction unveiled that even the subtlest changes in initial measurements, such as the flutter of a butterfly's wings, could catalyze vast divergences in subsequent weather patterns. This phenomenon highlights the sensitivity of nonlinear systems to initial states, where slight variations can exponentially amplify, yielding divergent trajectories that defy predictability.

Chaotic systems, a cornerstone of chaos theory, are characterized by their sensitive reliance on initial conditions, fostering aperiodic behavior and deterministic yet unforeseeable trajectories. Despite being governed by deterministic equations, chaotic systems exhibit intricate and seemingly random behavior over time. Key features encompass their sensitivity to initial conditions, nonlinear dynamics, intricate patterns like strange attractors, and the apparent unpredictability stemming from these complexities. These systems manifest across various natural phenomena, from weather phenomena and fluid dynamics to population dynamics and biological systems. By unraveling and analyzing chaotic systems, researchers glean invaluable insights into the underpinnings of complex phenomena, shedding light on the boundaries of predictability within deterministic frameworks.

Methodology

Mathematical Equations and Solutions

The Logistic Growth Differential Equation

Presentation of the equation: $\frac{dN}{dt} = rN(1 - \frac{N}{K})$

Where:
- N represents the population size,
- t represents time,
- r is the intrinsic growth rate of the population, and
- K is the carrying capacity, which represents the maximum population size that the environment can sustain

Analytical resolution: $\frac{dN}{dt} = rN(1 - \frac{N}{K})$

We start by separating the variables, to get $\frac{dN}{N(1-\frac{N}{K})} = rdt$.

Then we integrate both sides, $\int \frac{1}{N(1-\frac{N}{K})} dN = \int rdt$.

Using partial fraction decomposition, we can integrate, $\frac{1}{N(1-\frac{N}{K})} = \frac{A}{N} + \frac{B}{1-\frac{N}{K}}$.

Multiplying by $N(1 - \frac{N}{K})$, to clear out the denominators, $1 = A(1 - \frac{N}{K}) + BN$

Now expanding and equating the coefficients, $A + B = 0$ and $-\frac{A}{K} = r$.

Substituting back into the PFD, $\frac{1}{N(1-\frac{N}{K})} = \frac{-rK}{N} + \frac{r}{1-\frac{N}{K}}$.

Integrating, $-rK\int\frac{1}{N}dN + R\int\frac{1}{1-\frac{N}{K}}dN = \int rdt$.

$-rKln|N| + rKln|1-\frac{N}{K}| = rt + C$, where C is the integration constant.

Further simplifications lead to the final solution, $N = \frac{1}{e^{rt+C}+\frac{1}{K}}$.

Where:
- N_0 is the initial population size at $t = 0$.
- r is the intrinsic growth rate of the population.
- K is the carrying capacity, representing the maximum sustainable population size.

The Logistic Map

Derivations from the Logistic Differential Equation to the Logistic Map

The logistic map is a discrete-time dynamical system that arises from the logistic differential equation through discretization. The logistic map is widely studied in chaos theory due to its rich dynamical behavior, including bifurcations, chaos, and the emergence of complex patterns.

To derive the logistic map, we discretize the logistic differential equation by considering population size at discrete time intervals. Let N_n represent the population size at time $t = n\Delta t$, where Δt is the time step size.

Using the forward Euler method for discretization, the change in population size over one time step Δt is given by: $N_{n+1} - N_n = \Delta t \cdot \frac{dN}{dt}$.

Substituting the logistic differential equation, we get: $N_{n+1} - N_n = \Delta t \cdot rN_n(1 - \frac{N_n}{K})$.

Rearranging gives us the logistic map, $N_{n+1} = N_n + r\Delta t \cdot N_n(1 - \frac{N_n}{K})$.

This iterative equation defines how the population size at time $n + 1$ depends on the population size at time n, the growth rate r, the time step size Δt, and the carrying capacity K.

Fundamental Python Programming Concepts

Utilization of numpy library

Exploring the dynamics of logistic equations using Python typically involves utilizing the numpy library for numerical computations.

1. **Array Creation:** Numpy provides powerful array data structures that allow efficient storage and manipulation of numerical data. You can create numpy arrays using functions like `np.array()`, `np.zeros()`, `np.ones()`, `np.linspace()`, etc.
2. **Numerical Operations:** Numpy offers a wide range of mathematical functions and operations optimized for arrays. These include basic arithmetic operations (`+`, `-`, `*`, `/`), element-wise functions (`np.sin()`, `np.exp()`, `np.log()`, etc.), linear algebra operations (`np.dot()`, `np.linalg.inv()`, `np.linalg.eig()`, etc.).

Graphical representation with matplotlib.pyplot

Using matplotlib.pyplot for graphical representation of logistic equation dynamics enables visualization and insight into how population size changes over time, facilitating parameter exploration, validation of implementations, and effective communication of findings.

1. **Defining the logistic equation:** Defining a function that represents the logistic equation, taking parameters such as the intrinsic growth rate (r), carrying capacity (K), initial population size (N0), and time.
2. **Generating time values:** Creating an array of time values over which will simulate the dynamics of the logistic equation.
3. **Computing population values:** Using the logistic equation function, the population values corresponding to the time values will be computed.
4. **Plotting the dynamics:** Using matplotlib.pyplot to plot the population dynamics over time.

Application of Sympy for symbolic computations

1. **Lambdify**: The lambdify function converts symbolic expressions into callable functions, enabling efficient numerical evaluation. This is useful for accelerating computations involving complex symbolic expressions.
2. **Integrate:** The integrate function computes symbolic integrals of expressions, allowing for exact solutions to indefinite and definite integrals. This is valuable for analytical calculations in mathematics and physics.
3. **ODE Solver:** The ODE solver provides tools to solve ordinary differential equations symbolically. By specifying initial conditions and system equations, it computes symbolic solutions, facilitating analysis of dynamical systems in various fields, including physics, engineering, and biology.

Python-Driven Discovery of Key Characteristics

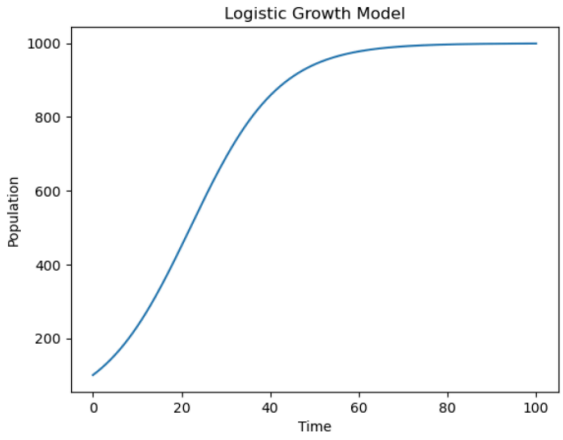
Comparison of symbolic, numerical, and graphical analyses

In [1]:
```python
#3.1.1 - symbolic analysis
import sympy as sp

# Define symbolic variables
t, P0, r, K = sp.symbols('t P0 r K')

# Define the logistic equation
P = (r * P0 * K) / (K + (P0 - K) * sp.exp(-r * t))

# Compute the derivative with respect to t
dP_dt = sp.diff(P, t)

print("Logistic Equation:", P)
print("Derivative dP/dt:", dP_dt)
```

```
Logistic Equation: K*P0*r/(K + (-K + P0)*exp(-r*t))
Derivative dP/dt: K*P0*r**2*(-K + P0)*exp(-r*t)/(K + (-K + P0)*exp(-r*t))**2
```

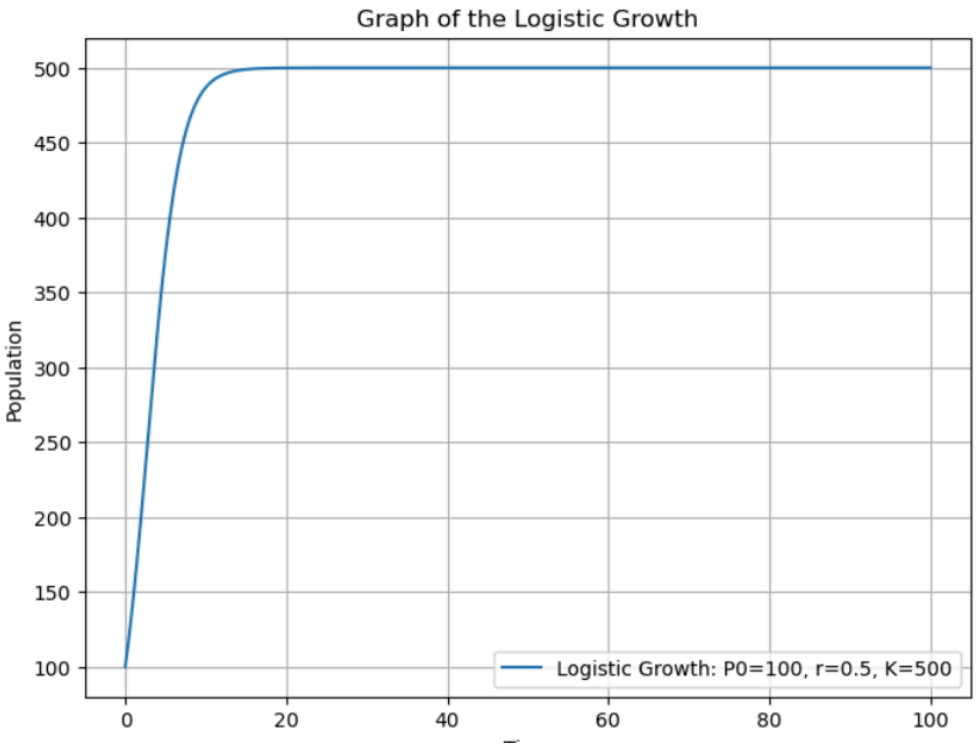
In 3.1.1, the logistic equation is in the form $P = \frac{r * P0 * K}{K + (P0 - K) * e^{-r*t}}$. Sympy is used to define symbolic variables. Unlike numerical libraries, Sympy performs exact symbolic computations and can manipulate algebraic expressions easily. Sympy automates tedious processes such as differentiation, integration, solving equations, etc. Sympy also works well with other libraries such as NumPy, SciPy, and matplotlib making it a powerful tool for analyses.

In [2]:
```python
#3.1.2 - numerical analysis
import numpy as np
from scipy.integrate import odeint

# Logistic equation as a function
def logistic(P, t, r, K):
    return r * P * (1 - P / K)

# Parameters
r = 0.1  # growth rate
K = 1000  # carrying capacity
P0 = 100  # initial population
t = np.linspace(0, 100, 200)  # time array from 0 to 100

# Solve the differential equation
solution = odeint(logistic, P0, t, args=(r, K))

# Plotting
import matplotlib.pyplot as plt
plt.plot(t, solution)
plt.title('Logistic Growth Model')
plt.xlabel('Time')
plt.ylabel('Population')
plt.show()
```



In 3.1.2 the differential form of the logistic equation is used as $\frac{dP}{dt} = rP(1 - \frac{P}{K})$. Numerical analysis has several pros and cons that make it suitable for certain applications while limiting others. Some of the pros of numerical analysis include flexibility, software availability, easy integration with data, and scalability. Some cons of numerical analysis include accuracy issues and computational cost.

In [4]:
```python
#3.1.3 - graphical analysis
import numpy as np
import matplotlib.pyplot as plt

# Correct Logistic function definition
def logistic(P0, r, K, t):
    return K * P0 / (P0 + (K - P0) * np.exp(-r * t))

# Time array
t = np.linspace(0, 100, 200)

# Parameters
P0 = 100
r = 0.5  # growth rate
K = 500  # carrying capacity

# Population values calculated with the correct logistic function
P = logistic(P0, r, K, t)

# Create the plot
plt.figure(figsize=(8, 6))
plt.plot(t, P, label=f'Logistic Growth: P0={P0}, r={r}, K={K}')
plt.title('Graph of the Logistic Growth')
plt.xlabel('Time')
plt.ylabel('Population')
plt.legend()
plt.grid(True)
plt.show()
```

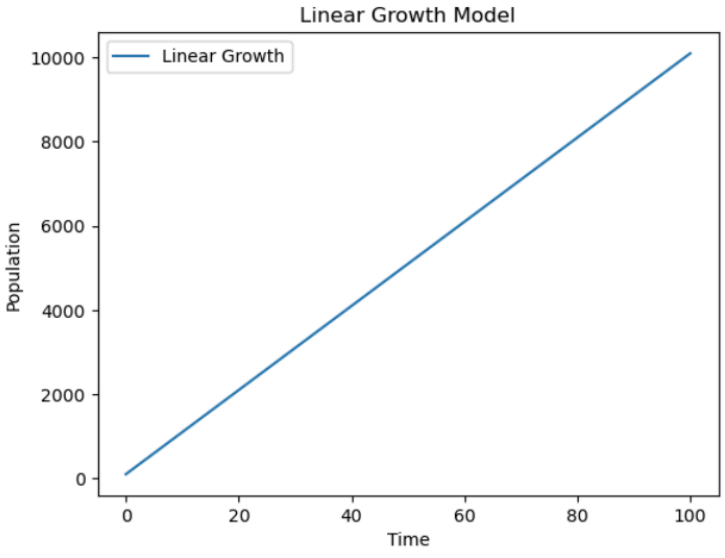


In 3.1.3, the logistic equation is defined as $P(t) = \frac{K*P0}{P0+(K-P0)*e^{-r*t}}$. The graph shows how population begins at P0 and approaches the carrying capacity K over time. Graphical analysis gives a visual representation

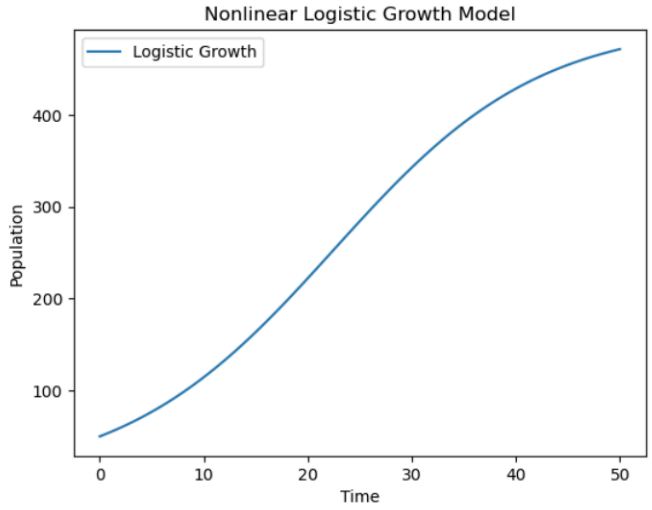
of how population changes over time given these specified conditions. Graphical analysis also provides quick insights when comparing different scenarios as you can visually see the differences in dynamics if you change parameters like r and K. Some cons of graphical analysis include potential for misinterpretation and less precision than observing numerical or symbolic values.

Implementation of finite difference schemes to compute linear and nonlinear growth rates

In [5]:
```python
#3.2
# Linear growth model
import numpy as np
import matplotlib.pyplot as plt

# Parameters
increment = 100  # constant increase per time step
P0 = 100  # initial population
dt = 1.0  # time step
T = 100   # total time

# Time array
t = np.arange(0, T + dt, dt)

# Initialize the population array
P = np.zeros(len(t))
P[0] = P0

# Finite difference scheme: Forward Euler Method for linear growth
for i in range(1, len(t)):
    P[i] = P[i-1] + increment

# Plotting
plt.plot(t, P, label='Linear Growth')
plt.title('Linear Growth Model')
plt.xlabel('Time')
plt.ylabel('Population')
plt.legend()
plt.show()
```



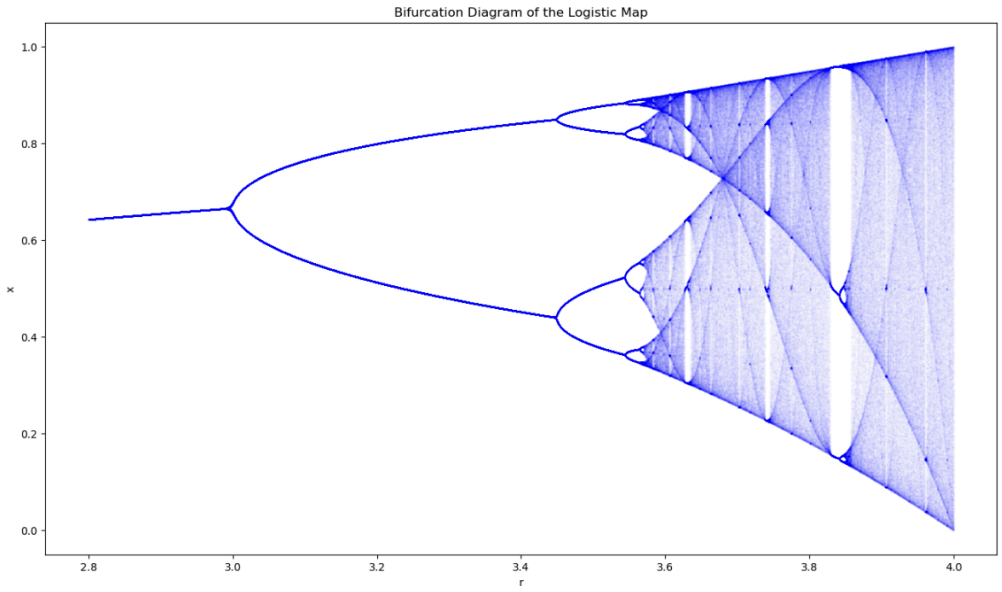
In [2]:
```
# non linear growth
# Parameters
r = 0.1  # growth rate
K = 500  # carrying capacity
P0 = 50  # initial population
dt = 0.5  # time step
T = 50   # total time

# Time array
t = np.arange(0, T+dt, dt)

# Initialize the population array
P = np.zeros(len(t))
P[0] = P0

# Finite difference scheme for logistic growth: Forward Euler Method
for i in range(1, len(t)):
    P[i] = P[i-1] + r * P[i-1] * (1 - P[i-1] / K) * dt

# Plotting
plt.plot(t, P, label='Logistic Growth')
plt.title('Nonlinear Logistic Growth Model')
plt.xlabel('Time')
plt.ylabel('Population')
plt.legend()
plt.show()
```



In the above code for linear and nonlinear growth models, finite difference schemes (like the forward Euler Method used) can be used to provide a numerical method for approximating solutions to differential equations that describe various types of growth models. For linear growth, a constant is added as the growth rate does not change. For nonlinear growth, a similar scheme is used but incorporating the carrying capacity which limits the growth as the population increases. Nonlinear growth rates often more closely model population growth.

Examination of orderly and chaotic solution patterns

In [8]:
```
import numpy as np
import matplotlib.pyplot as plt
ra=2.8
rb=4
dr = 0.0001
reps = 600 # number of repetitions
numtoplot = 200
lims = np.zeros(reps)
fig=plt.figure(figsize=(16,9))
lims[0] = np.random.rand()
for r in np.arange(ra, rb, dr):
    for i in range(reps - 1):
        lims[i + 1] = r * lims[i] * (1 - lims[i])

    plt.plot([r] * numtoplot, lims[reps - numtoplot :], "b.", markersize=0.02)
    # x, the same value of x for 200 data points
    # y, up to 200 different values (plotted at the same x spot!)
plt.xlabel("r")
plt.ylabel("x")
plt.title("Bifurcation Diagram of the Logistic Map")
plt.show()
```



In the above code, the bifurcation diagram of the logistic map shows the transition from order to chaos as r increases. The diagram shows how small changes in the parameter r (which controls growth rate) can lead to dramatically different behaviors. With growth rates less than one the system would collapse to zero, while with growth rates between one and three it settles at a stable population level. Just after a growth rate value of three, the population values split into two paths, and again around 3.4 into four paths. As r further increases, a transition between order and chaos can be seen.

Conclusions

This comprehensive report spans several mathematical concepts central to understanding dynamic systems through calculus, differential equations, and their applications in various scientific domains. Starting with Newton's second law of motion, the report underscores the foundation of dynamics, highlighting how differential equations are used to describe motion via acceleration, velocity, and position. The exploration of Robert May's work demonstrates the application of the logistic equation to ecological models, revealing complex behaviors such as chaos and unpredictability, previously unappreciated in traditional linear models. The review of Lorenz's studies extends this complexity to atmospheric sciences, where the concept of the "butterfly effect" illustrates how minuscule changes in initial conditions can lead to vastly different outcomes, introducing chaos theory to a broader scientific audience.

Further, the report applies the logistic equation to pandemic modeling, using COVID-19 as a case study to show how mathematical models can predict disease spread, though with limitations due to simplifying assumptions. The sigmoid function's role in neural networks, particularly in deep learning for binary classification, is detailed, noting both its advantages and the challenges it presents, such as the vanishing gradient problem. The introduction of chaotic systems and the butterfly effect broadens the discussion to general chaos theory, emphasizing the sensitivity of such systems to initial conditions and their broad implications across natural and scientific phenomena.

The methodology section is particularly detailed, providing a step-by-step derivation of solutions to the logistic growth differential equation through techniques like partial fraction decomposition and the forward Euler method for discretization, leading to the logistic map. This mathematical rigor is complemented by practical programming approaches using Python, where numpy and matplotlib.pyplot are utilized to simulate and visualize logistic equation dynamics, and sympy aids in symbolic computations, enhancing both the understanding and application of these complex mathematical models. This synthesis of theory, practical application, and computational modeling serves not only to advance scientific understanding but also to apply this knowledge to real-world problems in various disciplines.

This report then explored the dynamics of logistic equations using Python's computational capabilities across symbolic, numerical, and graphical analyses, each providing unique insights into dynamic systems. Symbolic analysis with SymPy demonstrated its ability to perform exact calculations and automate complex mathematical tasks, proving invaluable alongside other libraries like NumPy, SciPy, and matplotlib. Numerical methods addressed solving the logistic equation's differential form, highlighting flexibility and scalability but noting potential accuracy and computational costs. Graphical analysis effectively visualized population evolution towards

carrying capacity, offering intuitive insights though sometimes at the expense of precision. Finite difference schemes, in particular the forward Euler method, were applied to model both linear and nonlinear growth, accurately depicting natural population dynamics with considerations for constant and variable growth rates influenced by carrying capacity. The bifurcation diagram of the logistic map illustrated how small changes in the growth rate parameter r can shift system behavior from orderly to chaotic, underlining the critical transitions and sensitivity to initial conditions. Overall, this report underscores Python's robustness in analyzing and visualizing complex systems, providing comprehensive insights crucial for fields like ecology, economics, and engineering, where predicting dynamic behavior is essential.

Author Contributions

Conceptualization, D.B.; methodology, D.B.; software,C.B.; writing—original draft preparation, D.B.,C.B.; writing—review and editing, D.B.,C.B.
All authors have read and agreed to the published version of the manuscript.

Appendices

Dania Bawab (REDID 825075622)

HW10

Summary:

In this homework, I solved a logistic equation symbolically and plotted the solution in Python, then I implemented array operations using Python lists and NumPy arrays, evaluating their performance.

Methodology:

1. a) This question involves separating variables and integrating both sides of the logistic equation $\frac{dX}{dt} = \sigma X(1 - X)$, leading to a separable first-order ordinary differential equation. Through partial fraction decomposition and solving for the constants, the symbolic solution $X(t)$ is derived with the given initial condition $X(0) = 0.001$. b)The code defines a symbolic solution for the logistic equation and plots the solution for a given range of time points t and a specified value of σ. The logistic solution is calculated using the defined function `logistic_solution`, and then plotted using Matplotlib. The resulting plot shows the behavior of the logistic equation over time. You can adjust the `sigma` value to observe different behaviors of the logistic equation.
2. The code defines a logistic equation function and an Euler scheme function to numerically solve the logistic equation $\frac{dX}{dt} = \sigma X(1 - X)$ with initial condition $X(0) = 0.1$. The Euler scheme iteratively calculates $X(t)$ for different values of $\sigma \Delta t$ within a specified time range. The resulting plots visualize the behavior of $X(t)$ over time for each value of $\sigma \Delta t$, providing insights into the dynamics of the logistic equation.
3. a) Copied code b) Adaped code to include CuPy #### Results and Code below:

$dX/dt = \sigma X(1 - X)$ with the initial condition $X(0) = 0.001$.

$dX/(X(1 - X)) = \sigma dt$

$\int dX/(X(1 - X)) = \int \sigma dt$

$1/(X(1 - X)) = A/X + B/(1 - X)$, where A and B are constants.

$1/(X(1 - X)) = (A(1 - X) + BX)/(X(1 - X)) = ((A + B)X - A)/(X(1 - X))$

$1 = (A + B)X - A$

$A + B = 0 - A = 1$

$A = -1$ and $B = 1$.

$\int dX/(X(1 - X)) = \int(-1/X + 1/(1 - X))dX$

$= -\ln|X| + \ln|1 - X| + C$

$-\ln|X| + \ln|1 - X| = \sigma t + C$

$(|1 - X|/|X|) = e^{-\sigma t + C}$

$(1 - X)/X = Ce^{-\sigma t}$

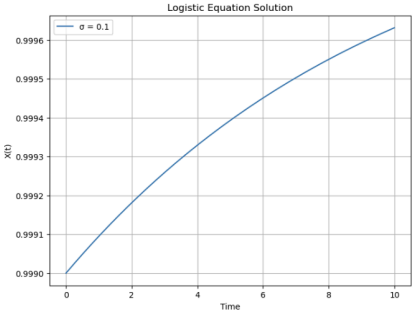
$(1 - 0.001)/0.001 = Ce^{-\sigma \cdot 0}$

$0.999/0.001 = C$

$C = 999$

$X(t) = 1/(1 + 999e^{-\sigma t})$

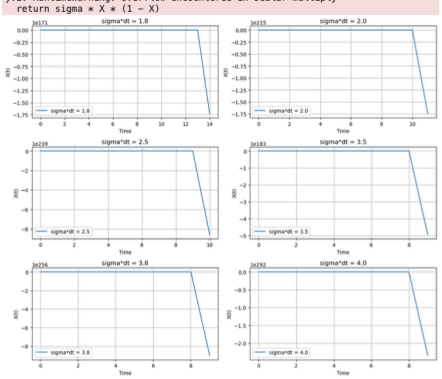
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

def logistic_solution(t, sigma):
    X0 = 0.001
    C = 1 / (1 - X0) - 1
    return 1 / (1 + C * np.exp(-sigma * t))

t = np.linspace(0, 10, 1000)

sigma = 0.1

X = logistic_solution(t, sigma)

plt.figure(figsize=(8, 6))
plt.plot(t, X, label=f"σ = {sigma}")
plt.title("Logistic Equation Solution")
plt.xlabel("Time")
plt.ylabel("X(t)")
plt.legend()
plt.grid(True)
plt.show()
```



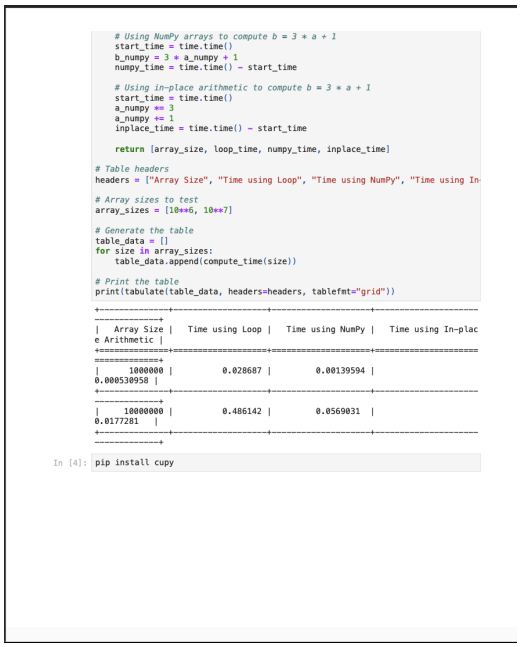
In [2]:
```
def logistic_eq(X, sigma):
    return sigma * X * (1 - X)

def euler_scheme(X0, sigma_dt, t_range):
    t_values = np.arange(t_range[0], t_range[1] + 1, 1)
    X_values = np.zeros(len(t_values))

    X_values[0] = X0

    for i in range(1, len(t_values)):
        X_values[i] = X_values[i - 1] + sigma_dt * logistic_eq(X_values[i - 1]

    return t_values, X_values

X0 = 0.1
t_range = [0, 40]
sigma_dt_values = [1.8, 2.0, 2.5, 3.5, 3.8, 4.0]

plt.figure(figsize=(12, 10))

for i, sigma_dt in enumerate(sigma_dt_values):
    t_values, X_values = euler_scheme(X0, sigma_dt, t_range)

    plt.subplot(3, 2, i+1)
    plt.plot(t_values, X_values, label=f'sigma*dt = {sigma_dt}')
    plt.title(f'sigma*dt = {sigma_dt}')
    plt.xlabel('Time')
    plt.ylabel('X(t)')
    plt.legend()
    plt.grid(True)
```

```
plt.tight_layout()
plt.show()
```

```
/var/folders/8t/345yvb2j3rb3cvr9gmt9bd_r0000gn/T/ipykernel_52559/3262841588.p
y:2: RuntimeWarning: overflow encountered in scalar multiply
  return sigma * X * (1 - X)
```



In [3]:
```
#pip install tabulate
import numpy as np
import time
from tabulate import tabulate  # Import tabulate to format the output into a ta

# Function to compute the time taken for each method
def compute_time(array_size):
    # Create a large array using Python lists
    a_list = list(range(array_size))

    # Create a large array using NumPy
    a_numpy = np.arange(array_size)

    # Using loop to compute b = 3 * a + 1
    start_time = time.time()
    b_loop = []
    for element in a_list:
        b_loop.append(3 * element + 1)
    loop_time = time.time() - start_time
```



```
    # Using NumPy arrays to compute b = 3 * a + 1
    start_time = time.time()
    b_numpy = 3 * a_numpy + 1
    numpy_time = time.time() - start_time

    # Using in-place arithmetic to compute b = 3 * a + 1
    start_time = time.time()
    a_numpy *= 3
    a_numpy += 1
    inplace_time = time.time() - start_time

    return [array_size, loop_time, numpy_time, inplace_time]

# Table headers
headers = ["Array Size", "Time using Loop", "Time using NumPy", "Time using In

# Array sizes to test
array_sizes = [10**6, 10**7]

# Generate the table
table_data = []
for size in array_sizes:
    table_data.append(compute_time(size))

# Print the table
print(tabulate(table_data, headers=headers, tablefmt="grid"))
```

```
+-------------+-------------------+--------------------+---------------------
-------------+
|   Array Size |   Time using Loop |   Time using NumPy |   Time using In-plac
e Arithmetic |
+=============+===================+====================+=====================
=============+
|      1000000 |          0.028687 |          0.00139594 |
0.000530958 |
+-------------+-------------------+--------------------+---------------------
-------------+
|     10000000 |          0.486142 |          0.0569031  |
0.0177281   |
+-------------+-------------------+--------------------+---------------------
-------------+
```

In [4]:
```
pip install cupy
```

```
Collecting cupy
  Using cached cupy-13.0.0.tar.gz (3.5 MB)
  Preparing metadata (setup.py) ... error
  error: subprocess-exited-with-error

  × python setup.py egg_info did not run successfully.
  │ exit code: 1
  ╰─> [3 lines of output]
      Error: macOS is no longer supported
      Generating cache key from header files...
      Cache key (1527 files matching /private/var/folders/8t/345yvb2j3rb3cvr9g
mt9bd_r0000gn/T/pip-install-d06so0qu/cupy_a1f8e16d82944437a9341cb42c7b1128/cup
y/_core/include/**): 3cbf38e4ed2dcd03933e60786543ab0a19bd8dd9
      [end of output]

  note: This error originates from a subprocess, and is likely not a problem w
ith pip.
error: metadata-generation-failed

× Encountered error while generating package metadata.
╰─> See above for output.

note: This is an issue with the package mentioned above, not pip.
hint: See above for details.
Note: you may need to restart the kernel to use updated packages.
```

In [5]:
```
import cupy as cp

def compute_time(array_size):
    a_cupy = cp.arange(array_size)

    start_time = time.time()
    b_cupy = 3 * a_cupy + 1
    cupy_time = time.time() - start_time

    return [array_size, cupy_time]

# Table headers
headers = ["Array Size", "Time using CuPy"]

# Array sizes to test
array_sizes = [10**6, 10**7]

# Generate the table
table_data = []
for size in array_sizes:
    table_data.append(compute_time(size))

# Print the table
print(tabulate(table_data, headers=headers, tablefmt="grid"))
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[5], line 1
----> 1 import cupy as cp
      3 def compute_time(array_size):
      4     a_cupy = cp.arange(array_size)

ModuleNotFoundError: No module named 'cupy'
```

References

Bawab, Brophy 2024: Report on Homework Assignment 8. Math 340 Programming in Mathematics. San Diego State University, 2024.

Bawab, Brophy 2024: Report on Homework Assignment 9. Math 340 Programming in Mathematics. San Diego State University, 2024.

Bawab, Brophy 2024: Report on Homework Assignment 10. Math 340 Programming in Mathematics. San Diego State University, 2024.

Butterfly effect. (2019, March 4). Wikipedia; Wikimedia Foundation.

https://en.wikipedia.org/wiki/Butterfly_effect

May, R. (n.d.). *Robert May Mathematics and Ecology*.

https://www.merton.ox.ac.uk/sites/default/files/inline-files/Robert-May.pdf

Pelinovsky, E., Kurkin, A., Kurkina, O., Kokoulina, M., & Epifanova, A. (2020). Logistic equation and COVID-19. *Chaos, Solitons & Fractals*, *140*, 110241.

https://doi.org/10.1016/j.chaos.2020.110241

Robert L. May. (2021, December 8). Wikipedia.

https://en.wikipedia.org/wiki/Robert_L._May

Saeed, M. (2021, August 24). *A Gentle Introduction To Sigmoid Function*. Machine Learning Mastery. https://machinelearningmastery.com/a-gentle-introduction-to-sigmoid-function/

Shen, B.-W. (2023, August). https://www.researchgate.net/publication/372824025_A_Review_of_Lorenz's_Models_from_1960_to_2008_published

The Decision Lab. (2023). *The Butterfly Effect*. The Decision Lab. https://thedecisionlab.com/reference-guide/economics/the-butterfly-effect

The fundamental equation of dynamics. (n.d.). Www.youtube.com. Retrieved May 1, 2024, from https://youtu.be/UPomWGm10RM?si=uI1ek6q-vaEonftE

Yanisa. (2023, July 7). *AI | Neural Networks | Sigmoid Activation Function*. Codecademy. https://www.codecademy.com/resources/docs/ai/neural-networks/sigmoid-activation-function

Wikimedia Foundation. (2024, April 8). *Logistic map*. Wikipedia. https://en.wikipedia.org/wiki/Logistic_map

Boeing, G. (2016, October 21). *Chaos theory and the logistic map*. Geoff Boeing. https://geoffboeing.com/2015/03/chaos-theory-logistic-map/