A Report on Computational Physics - Spring 2023

Sudeepta Saha,

[a] *Independent University, Bangladesh (IUB)*
Dhaka 1229, Bangladesh

E-mail: 2010331@iub.edu.bd

Abstract: Computational Physics (PHY-433) Has been taught by Dr M Arshad Momen in Sprint 2023. This file contains a comprehensive report on that course.

[1] Corresponding author.

Contents

sadfas adsfahsdkfjasdf

1 Some examples and best-practices

For internal references use label-refs: see section 1. Bibliographic citations can be done with cite: refs. [**?** **?** **?**]. When possible, align equations on the equal sign. The package `amsmath` is already loaded. See (1.1).

$$\begin{aligned} x &= 1\,, \qquad y = 2\,, \\ z &= 3\,. \end{aligned} \tag{1.1}$$

Also, watch out for the punctuation at the end of the equations.

If you want some equations without the tag (number), please use the available starred-environments. For example:

$$x = 1$$

The amsmath package has many features. For example, you can use use `subequations` environment:

$$a = 1 \tag{1.2a}$$

$$b = 2 \tag{1.2b}$$

and it will continue to operate across the text also.

$$c = 3 \tag{1.2c}$$

The references will work as you'd expect: (1.2a), (1.2b) and (1.2c) are all part of (1.2).

A similar solution is available for figures via the `subfigure` package (not loaded by default and not shown here). All figures and tables should be referenced in the text and should be placed at the top of the page where they are first cited or in subsequent pages. Positioning them in the source file after the paragraph where you first reference them usually yield good results. See figure **??** and table **??**.

2 Problem 1

Write a python code that implements the Euler method, which is used to solve first-order ordinary differential equations and plot the value.

2.1 Introduction

The task of this problem is tom do a simple implementation of the Euler method to numerically solve a differential equation and plot the solution using the matplotlib library. The Euler method is a first-order numerical method for approximating the solution of a differential equation, and it involves iteratively updating the solution using a linear approximation. This code defines a function euler method that implements the Euler method to solve a first-order ordinary differential equation (ODE) of the form dy/dx = f(x,y), where f is a function of x and y. The method takes as input the derivative function f, the initial and final values of the independent variable x, the initial value of the dependent variable y, and the number of steps N to take.

2.2 Algorithm

1)The euler method function takes as input the derivative function f, the initial and final values of the independent variable x, the initial value of the dependent variable y, and the number of steps N to take. The function returns arrays of x and y values representing the solution to the differential equation.

2)The algorithm calculates the step size h by dividing the range of x values by the number of steps N. It then creates an array of x values using the linspace function from NumPy. The zeros function is used to initialize an array of y values, and the initial value of y is set to y0.

3)The algorithm then loops over the N steps, updating y at each step using the Euler method. The updated value of y is calculated by multiplying the step size h by the value of the derivative function f at the current x and y values, and adding it to the previous value of y.

4)The algorithm returns the arrays of x and y values representing the solution to the differential equation.

listings

2.3 Code

Listing 1. Euler Method Plotting

```
import numpy as np
import matplotlib.pyplot as plt

def f(x, y):
    return -y

def euler_method(f, x0, y0, xf, N): #f derivative function,
                                    #x0 and xf are the initial and final values
                                    #y0 initial value of the dependent variable
                                    #N= the number of steps
    h = (xf - x0) / N
    x = np.linspace(x0, xf, N+1)
    y = np.zeros(N+1)
    y[0] = y0
    for i in range(1, N+1):
        y[i] = y[i-1] + h * f(x[i-1], y[i-1])
    return x, y

x0 = 0
y0 = 1
```

```
xf = 10
N = 1000
x, y = euler_method(f, x0, y0, xf, N)

plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Solution of the differential equation using the Euler method")
plt.show()
```

2.4 Plot

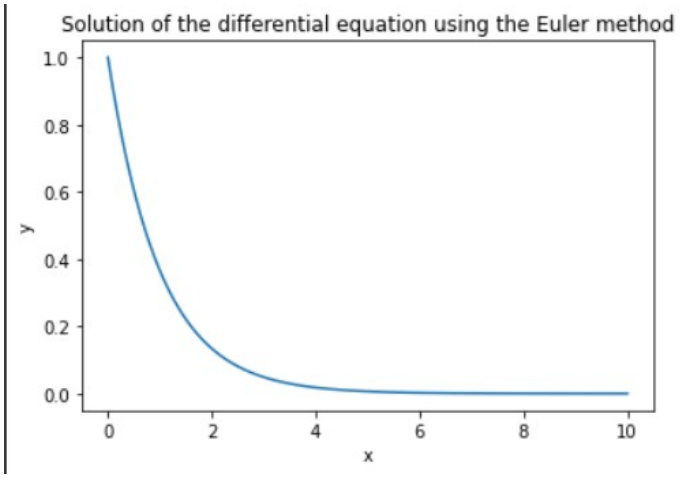

Figure 1. Plot of Euler method

2.5 Conclusion

The Euler method is a simple and widely used numerical method for approximating solutions to first-order ordinary differential equations. This method uses a finite difference approximation to the derivative of the dependent variable with respect to the independent variable to iteratively update the value of the dependent variable at each step. The euler method function presented in the code above implements this method using NumPy and Matplotlib libraries to solve the differential equation dy/dx = -y for a given range of x values. The algorithm produces a solution for y that approximates the true solution and can be visualized using a plot.

3 Problem 2

Write a python code that plots the Harmonic oscillator solving using the Euler method.

3.1 Introduction

The task is to implement the Euler method for solving a system of first-order ordinary differential equations, specifically, the equations for a harmonic oscillator. A harmonic oscillator is a physical system that oscillates with a characteristic frequency. It can be described mathematically as a second-order linear differential equation, which can be converted to a system of two first-order differential equations. The Euler method is a numerical method for solving differential equations. It is a simple and widely used method that approximates the solution at discrete time steps by using the slope at the previous time step. It is a first-order method, meaning that its error decreases linearly with the step size.

3.2 Algorithm

The algorithm for this code is summarized in the following steps:
1) Import the necessary libraries (numpy and matplotlib).
2) Define the system of differential equations to be solved (f).
3) Define the Euler method function, which takes as input the differential equations, the initial conditions, the final time, and the number of steps.
4) Calculate the step size h by dividing the time interval by the number of steps.
5) Create arrays to store the time and solution values.
6) Initialize the arrays with the initial conditions.
7) Use a for loop to iterate over the time steps, using the Euler method to approximate the solution at each step.
8) Return the arrays with the time and solution values.
9) Set the initial conditions and simulation parameters.
10) Call the Euler method function with the parameters, and store the returned time and solution arrays. Plot the solution using matplotlib.

3.3 Code

```
import numpy as np
import matplotlib.pyplot as plt

def f(t, y):
    return np.array([y[1], -y[0]])

def euler_method(f, t0, y0, tf, N):
    h = (tf - t0) / N
    t = np.linspace(t0, tf, N+1)
    y = np.zeros((N+1, 2))
    y[0, :] = y0
    for i in range(1, N+1):
        y[i, :] = y[i-1, :] + h * f(t[i-1], y[i-1, :])
    return t, y

# Set the initial conditions and parameters for the simulation
t0 = 0
y0 = np.array([1.0, 0.0])
tf = 50
N = 1000
t, y = euler_method(f, t0, y0, tf, N)

plt.plot(t, y[:, 0])
plt.xlabel("Time(t)")
plt.ylabel("Amplitude")
plt.title("Harmonic oscillation using the Euler method")
plt.show()
```

3.4 Plot

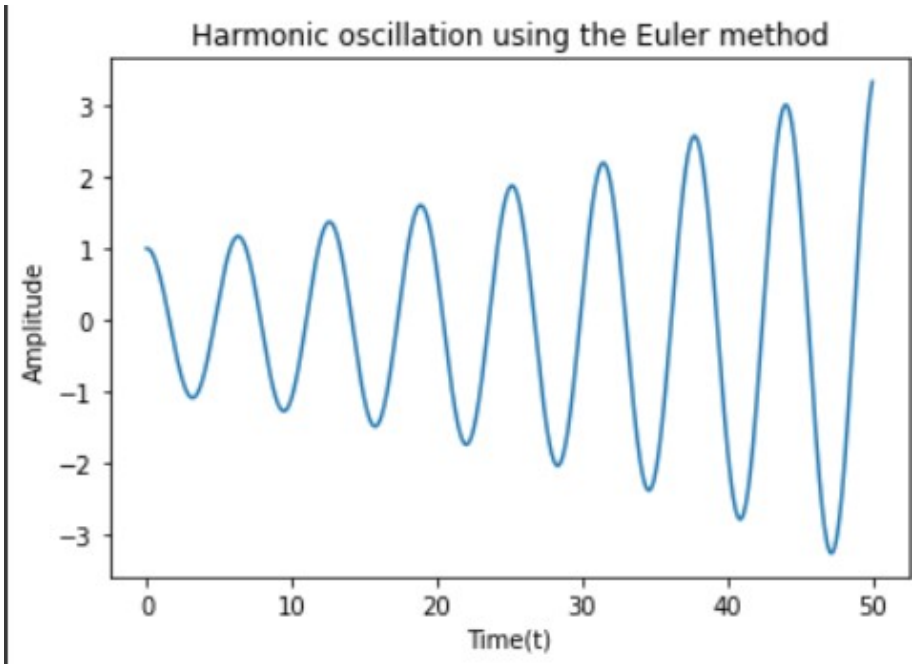

Figure 2. Plot of Harmonic Oscillation using the Euler Method

3.5 Conclusion

In this code, it has been demonstrated the implementation of the Euler method for solving a system of first-order ordinary differential equations, specifically, the equations for a harmonic oscillator. The Euler method is a simple and widely used numerical method that approximates the solution at discrete time steps by using the slope at the previous time step. This code shows how Python libraries such as numpy and matplotlib can be used to perform scientific computing and data visualization. By solving the equations of motion for a harmonic oscillator, we can visualize the expected oscillation of the system.

4 Problem 3

Use the Monte Carlo method to approximate the value of pi.

4.1 Introduction

The task is use the Monte Carlo method that relies on random sampling to obtain numerical results. The method can be used to estimate the value of mathematical constants, such as pi. The Monte Carlo method is particularly useful when analytical solutions are difficult or impossible to obtain, and it can provide a numerical approximation with a known level of uncertainty.

In the case of estimating pi, the Monte Carlo method works by generating random points and checking whether they fall inside or outside a unit circle centered at the origin. The probability that a randomly generated point falls inside the circle is proportional to the area of the circle, which is pi times the radius squared. By counting the number of points inside the circle and dividing by the total number of points generated, we can estimate the value of pi as 4 times the ratio of the number of points inside the circle to the total number of points.

4.2 Algorithm

The algorithm for this code is summarized in the following steps:
1) Set the number of points to use in the Monte Carlo method to num points.
2) Initialize variables to keep track of the number of points inside and outside the circle to inside circle and outside circle.
3) Generate num points random points, each with x and y coordinates between -1 and 1.
4) For each point, calculate its distance from the origin using the Pythagorean theorem.
5) If the distance is less than or equal to 1 (which means the point is inside the unit circle), increment inside circle by 1. Otherwise, increment outside circle by 1.
6) Calculate the value of pi using the formula pi = 4 * (number of points inside the circle / total number of points).
7) Print the result.

4.3 Code

```
import random
import math
import matplotlib.pyplot as plt

# Set the number of points to use in the Monte Carlo method
num_points = 100000

# Initialize variables to keep track of the number of points inside and outside
inside_circle = 0
outside_circle = 0

# Generate random points and check if they are inside the circle
for i in range(num_points):
    x = random.uniform(-1, 1)
    y = random.uniform(-1, 1)
    distance = math.sqrt(x**2 + y**2)
    if distance <= 1:
        inside_circle += 1
    else:
        outside_circle += 1

# Calculate the value of pi using the Monte Carlo method
pi_approx = 4 * (inside_circle / num_points)

# Print the result
print("Approximate value of pi:", pi_approx)
```

4.4 Conclusion

The code gives a result near the actual pi value. The accuracy of the estimation depends on the number of samples generated, and the Monte Carlo method can be combined with other techniques to improve its efficiency and reduce its error.

5 Problem 4

Approximates the volume of a tetrahedron using the Monte Carlo method

5.1 Introduction

The task of this code is to approximates the volume of a tetrahedron using the Monte Carlo method and visualizes the volume and the random points used to calculate the approximation.
The tetrahedron is defined by the points (0,0,0), (1,0,0), (0,1,0), and (0,0,1) in a 3D coordinate system. The Monte Carlo method randomly generates points within the bounding cube
(0 <= x,y,z <= 1) and checks if the point is inside the tetrahedron by calculating if the sum of the x,y,z coordinates is less than or equal to 1. The approximation of the volume is calculated by dividing the number of points inside the tetrahedron by the total number of generated points.

5.2 Algorithm

The algorithm of this problem can be described in following steps:
1) Set the number of points to use in the Monte Carlo method.
2) Initialize a variable to keep track of the number of points inside the volume.
3) Generate random points within the bounding cube (0 <= x,y,z <= 1) using the random.uniform function from Python's random module.
4) For each point, check if it is inside the tetrahedron by calculating if the sum of the x,y,z coordinates is less than or equal to 1.
5) After all points have been generated and checked, approximate the volume by dividing the number of points inside the tetrahedron by the total number of generated points.
6)Plot the tetrahedron using the plot surface function from Matplotlib.
7)Plot the volume with the x,y,z coordinates from the 2D grid and the corresponding z values as the height.

5.3 Code

```
import random
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

# Set the number of points to use in the Monte Carlo method
num_points = 1000

# Initialize variable to keep track of the number of points inside the volume
inside_volume = 0
# Generate random points and check if they are inside the volume
for i in range(num_points):
    x = random.uniform(0, 1)
    y = random.uniform(0, 1-x)
    z = random.uniform(0, 1-x-y)
    if x + y + z <= 1:
        inside_volume += 1

# Calculate the value of the volume using the Monte Carlo method
volume_approx = inside_volume / num_points

# Print the result
print("Approximate value of the volume:", volume_approx)

# Plot and shade the volume generated
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlim([0, 1])
ax.set_ylim([0, 1])
ax.set_zlim([0, 1])
x, y = np.linspace(0, 1, 20), np.linspace(0, 1, 20)
X, Y = np.meshgrid(x, y)
Z = 1 - X - Y
ax.plot_surface(X, Y, Z, alpha=0.2, color='#4169e1')
for i in range(num_points):
    x = random.uniform(0, 1)
    y = random.uniform(0, 1-x)
    z = random.uniform(0, 1-x-y)
    if x + y + z <= 1:
        ax.scatter(x, y, z, color='#4169e1', alpha=0.2)
    else:
```

```
        ax.scatter(x, y, z, color='#ff4500', alpha=0.2)
plt.show()
```

5.4 Result and Plot

Approximate value of the volume: 1.0

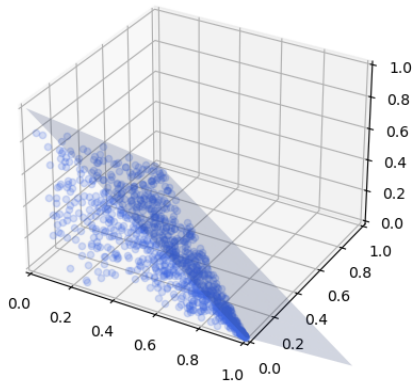

Figure 3. Volume approximation

5.5 Conclusion

In conclusion, this code provides a clear example of how the Monte Carlo method can be used to approximate the volume of a complex 3D shape such as a tetrahedron. By generating random points within the bounding cube and checking if they fall within the tetrahedron, the code can calculate an estimate of the volume with reasonable accuracy. The visualizations produced by the code make it easy to understand and interpret the results. Overall, the Monte Carlo method is a powerful tool for approximating complex shapes and can be applied to a wide range of scientific and engineering problems.

6 Problem 5

Use the Box-Muller transform to generate pairs of random numbers from a standard normal distribution

6.1 Introduction

The task is to illustrate how to use the Box-Muller transform to generate pairs of random numbers from a standard normal distribution and visualize them using scatter plots.
Sometimes we need to generate random numbers from other distributions, such as the normal (Gaussian) distribution, which is widely used in statistics and probability theory. In this context, the Box-Muller transform is a commonly used method to generate pairs of independent random numbers from a standard normal distribution.

6.2 Algorithm

The algorithm for the following problem is described in following steps:
1)Generate n pairs of independent random numbers u and v from a uniform distribution between 0 and 1.
2)For each pair of u and v, compute the following:
a. z1 = sqrt(-2 * ln(u)) * cos(2 * pi * v)
b. z2 = sqrt(-2 * ln(u)) * sin(2 * pi * v)
3)The pairs (z1, z2) are random numbers from a standard normal distribution.

Note: The Box-Muller transform uses the polar form of the Box-Muller transform, which transforms two independent uniform random variables into two independent normal random variables with zero mean and unit variance.

6.3 Code

```
import random
import matplotlib.pyplot as plt
import math

n= 16384
u=[]
v=[]
x=[]
y=[]
for i in range(n):
    u0= random.uniform(0,1)
    v0= random.uniform(0,1)
    u.append(u0)
    v.append(v0)
    x0= math.sqrt(-2 * math.log(u0))*math.cos(2 * math.pi * v0)
```

```
    y0= math.sqrt(-2 * math.log(u0))*math.sin(2 * math.pi * v0)


    x.append(x0)
    y.append(y0)

#Plot Graphs
plt.figure(figsize=(10,4))

plt.scatter(x,y)
plt.show()
```

6.4 Plot

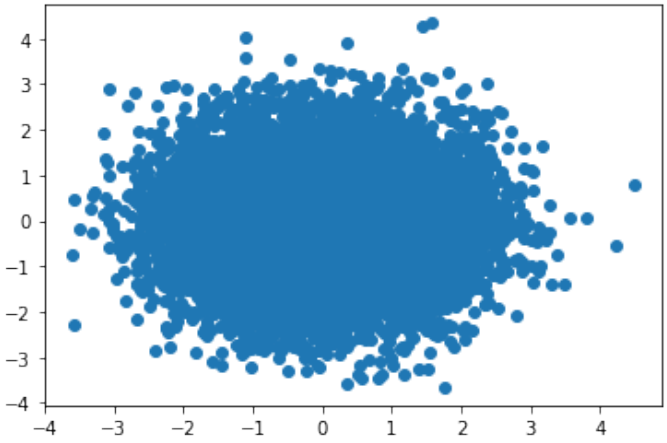

Figure 4. Visualization of Box-muller Using Scatter Plot

6.5 Conclusion

It is demonstrated how to use the Box-Muller transform to generate pairs of random numbers from a standard normal distribution and visualize them using scatter plots. The code shows that the pairs of uniform random numbers generated are transformed into pairs of normal random numbers with mean 0 and variance 1, which is a key property of the standard normal distribution.

7 Problem 6

Use the bisection method to find the root of a function.

7.1 Introduction

The task is to use the bisection method to find the root of a function. The function is defined as $f(x) = tanh(x) + x - x^3 - 1$. It is need to set a lower and upper bounds of the range to search for the root. The method works by repeatedly bisecting in this interval and selecting a subinterval in which the function changes sign. The process is repeated until a small enough interval containing the root is obtained.

7.2 Algorithm

The algorithm for the following problem is described in following steps:
1)Define the function f(x) that you want to find the root of.

2) Give the lower and upper bounds of the range to search for the root.

3) Check if the product of the function values at the two endpoints is positive. If it is, print an error message indicating invalid input and exit the program.

4) Set the tolerance level to a desired value.

5) Set the maximum number of iterations to a desired value.

6) Initialize the iteration counter to 0.

7) Start the bisection loop.

8) Calculate the midpoint c of the interval [a,b] as the average of the lower and upper bounds.

9) Evaluate the function at the lower bound f(a) and the midpoint f(c).

10) If the product of the function values at the lower bound and midpoint is negative, then the root is in the lower half of the range and the upper bound is set to c.

11) Otherwise, the root is in the upper half of the range and the lower bound is set to c.

12) Increment the iteration counter.

13) If the absolute difference between the function values at the two endpoints is less than the tolerance level or the maximum number of iterations has been reached, exit the

loop.

14) Otherwise, repeat steps 8 to 13.

15) Print the root of the equation.

7.3 Code

```
import math

def f(x):
    return math.tanh(x) + x - x**3 - 1

a = float(input("Enter the lower range: "))
b = float(input("Enter the upper range: "))

# Check if the product of the function values at the two endpoints is positive
if f(a) * f(b) >= 0:
    print("Invalid input")
else:
    # Set the tolerance level to 0.001
    tol = 0.001
    # Set the maximum number of iterations to 1000
    max_iter = 1000
    # Initialize the iteration counter to 0
    i = 0
    # Start the bisection loop
    while abs(f(a) - f(b)) >= tol and i < max_iter:
        c = (a + b) / 2
        if f(a) * f(c) < 0:
            b = c
        else:
            a = c
        i += 1
    # Print the root of the equation
    print("The root of the equation is: ", c)
```

7.4 Result

The root of the equation is: -1.5054.....
Result varies for different lower and upper range.

7.5 Conclusion

In conclusion, the bisection method is a simple and reliable numerical algorithm for finding the root of a function. It is an important tool in numerical analysis and is widely used in applications that involve finding roots of nonlinear equations. The method is easy to implement and does not require any knowledge of the derivative of the function, making it particularly useful for functions that are difficult to differentiate. However, the bisection method can be slower than other methods and may require a large number of iterations to achieve a desired level of accuracy.

8 Problem 7

Use the Newton-Raphson method to find the root of the function

8.1 Introduction

The task is to implements the Newton-Raphson method to find the root of the function $f(x) = tanh(x) + x - x^3 - 1$ within the given interval, which is obtained from the user input. The Newton-Raphson method is an iterative method that starts with an initial guess and iteratively refines this guess to get closer to the actual root of the function. In each iteration, it computes the slope of the function at the current guess (i.e., the derivative) and uses this to update the guess using x - f(x)/f'(x), where f'(x) is the derivative of the function. This process is repeated until the difference between two successive guesses is smaller than a given tolerance value.

8.2 Algorithm

Here is the algorithm for the following problem:
1) Import the math module.
2) Define the function f(x) that takes a real number x as input and returns the value of the function $tanh(x) + x - x^3 - 1$.
3)Define the function df(x) that takes a real number x as input and returns the value of the derivative of the function f(x), which is 1 - 3*x**2 + 1/math.cosh(x)**2.
4)Define the function newton raphson(x0, eps) that takes two arguments, x0 (initial guess) and eps (tolerance), and returns the approximate root of the function f(x) using the Newton-Raphson method.
5)Inside the newton raphson function, initialize x to x0.
6)While True, do the following:
a. Compute the value of f(x) using the function f(x) defined in step 2.
b. Compute the value of f'(x) (derivative of f(x)) using the function df(x) defined in step 3.
c. Compute the next guess x-next using the formula x - f(x)/f'(x).
d. If the absolute value of x-next - x is less than eps, return x-next as the approximate root.
e. Otherwise, update x to x-next and continue with the next iteration.
7)Get the lower and upper bounds of the root from the user input, and store them in the variables a and b, respectively.
8)Calculate the midpoint of the bounds as the initial guess for the root, and store it in the variable x0.
9)Set the tolerance eps to 0.001.
10) Call the function with x0 and eps as arguments, and store the result in the variable root.
11) Print the approximate root of the function f(x) using the Newton-Raphson method, which is stored in the variable root.

8.3 Code

```
import math

# Define the function f
def f(x):
    return math.tanh(x) + x - x**3 - 1

# Define the derivative of the function f
def df(x):
    return 1 - 3*x**2 + 1/math.cosh(x)**2

# Define the Newton-Raphson method to find the root of f
def newton_raphson(x0, eps):
    x = x0
    while True:
        fx = f(x)
        dfx = df(x)
        x_next = x - fx / dfx
        if abs(x_next - x) < eps:
            return x_next
        x = x_next

# Get the user input for the lower and upper bounds of the root
a = float(input("Enter lower bound: "))
b = float(input("Enter upper bound: "))

# Use the midpoint of the bounds as the initial guess for the root
x0 = (a + b) / 2

# Set the tolerance for the Newton-Raphson method
eps = 0.001

# Find the root using the Newton-Raphson method
root = newton_raphson(x0, eps)

# Print the root of the function
print("The root of the function is: ", root)
```

8.4 Result

The root of the equation is: -1.5053.....
Result varies for different lower and upper bound.

8.5 Conclusion

The code is the implementation of the Newton-Raphson method to find the root of a given function within a specified tolerance. The code imports the math module and defines the function f(x) and its derivative df(x), which are used in the newton raphson function to compute the root. The newton raphson function iteratively updates the guess for the root until the specified tolerance is met. The code prompts the user for the lower and upper bounds of the root, calculates the midpoint of the bounds as the initial guess, and calls the newton raphson function to find the root.

9 Problem 8

Use the the Metropolis-Hastings algorithm for sampling from a 1D harmonic oscillator distribution.

9.1 Introduction

In this task the Metropolis-Hastings algorithm, which is a popular Markov Chain Monte Carlo (MCMC) method used for sampling from probability distributions that are difficult to sample from directly. The algorithm is based on the idea of constructing a Markov chain whose stationary distribution is the target probability distribution.
The target distribution is a 1D harmonic oscillator distribution, which is defined by the potential energy function $0.5 * K * x^2$, where K is the spring constant. The algorithm generates a sequence of samples from this distribution by starting from an initial state and iteratively generating new states using a proposal distribution. In this case, the proposal distribution is a Gaussian distribution with mean equal to the current state and standard deviation .
The decision to accept or reject a new state is based on a probabilistic criterion that ensures that the Markov chain converges to the target distribution. This criterion is based on the relative probability of the proposed state compared to the current state, and is given by the Metropolis-Hastings acceptance probability. The algorithm uses this probability to make a random decision to either accept or reject the proposed state.
The acceptance ratio, which is the ratio of accepted samples to the total number of samples generated, is used as a measure of the efficiency of the algorithm. A high acceptance ratio indicates that the proposal distribution is a good match for the target distribution, while a low acceptance ratio indicates that the proposal distribution needs to be modified.

9.2 Algorithm

The algorithm for the Metropolis-Hastings algorithm can be described as follows:

1)Define the target probability distribution that we want to sample from, which is the 1D harmonic oscillator distribution in this case.

2) Choose an initial state and calculate its potential energy

3)Set the number of steps for the MCMC algorithm.

4)Choose a proposal distribution that is used to generate new states.Here The proposal distribution is a Gaussian distribution with mean and standard deviation .

5)For each steps do the following:

a. Generate a new state by sampling from the proposal distribution.

b. Calculate the potential energy of the new state.

c. Calculate the Metropolis-Hastings acceptance probability A.

d. Generate a uniform random number r in the range [0, 1].

e. If r < A, accept the new state. Otherwise, reject the new state by setting

6)Calculate the acceptance ratio, which is the ratio of accepted samples to the total number of samples generated.

7) Output the acceptance ratio.

9.3 Code

```
import numpy as np
import random
K = 1.0
beta = 1.0

def calculate_energy(x):
    return 0.5 * K * x**2
#Define initial state
x_i = random.uniform(0,1)
E_i = calculate_energy(x_i)

#Define final state
sigma = 1.0
final = lambda x: np.random.normal(x, sigma)

n_steps = 10000
reject=0

for i in range(n_steps):
    # Generate a final state
    x_f = final(x_i)
    E_f = calculate_energy(x_f)

    # Calculate the relative probability of picking the final state over the cu
    X = np.exp(-beta * (E_f - E_i))

    # Accept or reject the final state
    if E_f < E_i:
        reject +=1
```

```
    else:
        r = np.random.uniform()
        if X > r:
            reject +=1
accept= (n_steps-reject)

accept_ratio= (n_steps-reject)/n_steps

print("Total Accept:",accept)
print("Accept_Ratio:",accept_ratio)
```

9.4 Result

As this code works with random number generation, result varies. But acceptance ratio is near 0.3. Which is expected.

9.5 Conclusion

This code implements the algorithm to sample from a 1D harmonic oscillator distribution, with the aim of generating a sequence of states that converge to the target distribution. The algorithm uses a proposal distribution to generate new states, and the Metropolis-Hastings acceptance probability to determine whether to accept or reject the new state. The acceptance ratio is used to measure the efficiency of the algorithm, and can be used to tune the parameters of the proposal distribution for optimal performance.

10 Problem 9

Generates a random walk plot.

10.1 Introduction

A random walk is a mathematical object that describes a path taken by an object that moves randomly. In this task, the random walk is simulated by generating random x and y coordinates using the Python random module, which are then plotted using Matplotlib. The resulting plot shows the path taken by the random walk over a certain number of steps, in this case 1000.

10.2 Algorithm

Here's the algorithm for the code that generates a random walk plot:

1)Set the number of steps to n = 1000.
2)Create empty lists for x and y coordinates.
3) Set x-new and y-new to 0.
4) For each step i in range(n):
a. Generate a random number between -1 and 1 for x and y coordinates using the Python random.uniform() function.
b. Add the random number to the current x and y coordinates.
c. Append the new x and y coordinates to their respective lists.
5) Plot the x and y coordinates using the Matplotlib plot() function.
6) Display the plot using the Matplotlib show() function.

10.3 Code

```
import random
import matplotlib.pyplot as plt

n=1000

x=[]
y=[]
x_new=0
y_new=0
for i in range(n):
    x_new= x_new+random.uniform(-1,1)
    y_new= y_new+random.uniform(-1,1)

    x.append(x_new)
    y.append(y_new)
```

```
plt.plot(x,y)
```

10.4 Plot

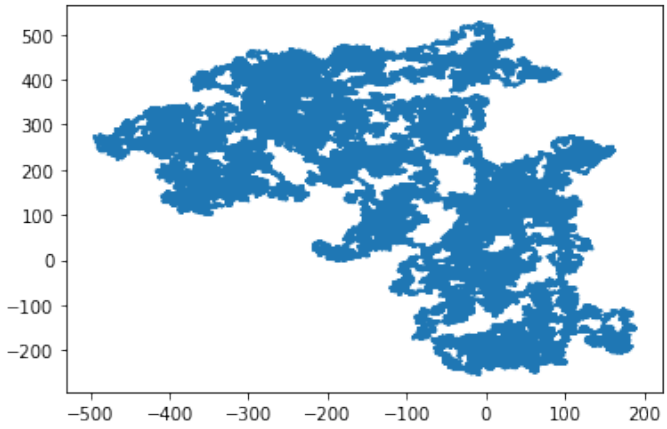

Figure 5. Visualization of Random walk

10.5 Conclusion

In conclusion, the code generates a random walk plot using Python and Matplotlib. The resulting plot shows the path taken by an object that moves randomly over a certain number of steps. Random walks have numerous applications in various fields and can be modified to suit different scenarios.

11 Problem 10

Use the shooting method to solve a system of ordinary differential equations.

11.1 Introduction

The task is to solves a system of ordinary differential equations using the Runge-Kutta method and finds the value of a variable at a specific point using root-finding algorithms. In numerical analysis, the shooting method is a method for solving a boundary value problem by reducing it to an initial value problem. It involves finding solutions to the initial value problem for different initial conditions until one finds the solution that also satisfies the boundary conditions of the boundary value problem.

11.2 Algorithm

Here is the algorithm for the shooting method:

1) Import the required modules: NumPy for numerical computing, and root scalar from SciPy for root-finding algorithms, and Matplotlib for plotting the solution.
2) Define the system of ordinary differential equations in the function.
a. The function takes two arguments, x and y, which represent the independent variable and the dependent variable, respectively.
b. The function returns an array dydx that represents the rate of change of y with respect to x.
3) Define the Runge-Kutta method (4th order) in the function rk4.
a. The function takes five arguments: a function f that defines the ODE system, two initial points x0 and y0, a final point x1, and a step size h.
b. The function returns two arrays x and y that represent the points at which the solution is computed and the solution itself.
4)Define the residual function in residual.
a. The function takes one argument, y2 guess, which represents the initial guess for the value of a dependent variable at a specific point.
b. The function uses the rk4 function to solve the ODE system with the given initial conditions and step size, and returns the residual between the value of a dependent variable at a specific point and a specific value.
5)Guess an initial value for the dependent variable at a specific point.
6)Use the root scalar function from the SciPy module to find the value of the dependent variable that makes the residual equal to zero.
a. The function takes two arguments: the residual function and a bracket that defines the range of the root.
b. The function returns the root of the residual function.
7)Retrieve the final solution to the ODE system using the rk4 function and the found value of the dependent variable.
8)Plot the solution to the ODE system using Matplotlib.

11.3 Code

```
import numpy as np
from scipy.optimize import root_scalar

# Define the system of ODEs
def ode_system(x, y):
    dydx = np.zeros_like(y)
    dydx[0] = y[1]
    dydx[1] = -y[0]*(1-y[0]**2)
    return dydx

# Define the Runge-Kutta method (4th order)
def rk4(f, x0, x1, y0, h):
    n = int((x1 - x0)/h)
    x = x0 + np.arange(n + 1)*h
    y = np.zeros((n + 1, len(y0)))
    y[0] = y0
    for i in range(n):
        k1 = h*f(x[i], y[i])
        k2 = h*f(x[i] + h/2, y[i] + k1/2)
        k3 = h*f(x[i] + h/2, y[i] + k2/2)
        k4 = h*f(x[i] + h, y[i] + k3)
        y[i+1] = y[i] + (k1 + 2*k2 + 2*k3 + k4)/6
    return x, y

# Define the residual function to find root for y1(10) = 0
def residual(y2_guess):
    # Initial conditions
    x0 = 0
    y0 = [0, y2_guess]
    # Solve the ODE system using a Runge-Kutta method
    x, y = rk4(ode_system, x0, 10, y0, h=0.01)
    # Return the residual
    return y[-1, 0]-1

# Guess an initial value for y2(0)
y2_guess = 0.1

# Use the root-finding algorithm to adjust y2(0) until y1(10) = 0
sol = root_scalar(residual, bracket=[0.1, 1.0], method='brentq')

# Retrieve the final solution
```

```
y2_final = sol.root
x_final = 10
y_final = [0, y2_final]

# Solve the ODE system using a Runge-Kutta method
x, y = rk4(ode_system, 0, x_final, y_final, h=0.01)

# Plot the solution
import matplotlib.pyplot as plt
plt.plot(x, y[:, 0])
plt.xlabel('x')
plt.ylabel('  (x)')
plt.show()
```

11.4 Plot

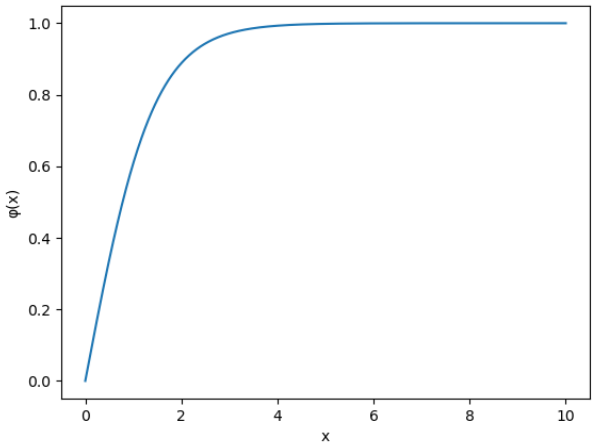

Figure 6. Shooting Method

11.5 Conclusion

In this code, numerical methods and root-finding algorithms to solve a system of ordinary differential equations. Specifically, we used the fourth-order Runge-Kutta method to approximate the solution to the system of ODEs, and the Brent algorithm from SciPy to find the value of a dependent variable that makes the residual equal to zero.

The code successfully finds a numerical solution to the system of ODEs, and the resulting plot shows the behavior of the dependent variable over the range of the independent variable.

12 Problem 11

Simulate the Ising model in 1D

12.1 Introduction

The task for this problem is to simulate the Ising model in 1D using the Metropolis algorithm and measures the specific heat as a function of temperature. The Ising model is a model in statistical physics that describes the behavior of magnetic materials. It consists of a lattice of spins, where each spin can be either up or down. The energy of the system depends on the orientation of the neighboring spins. The Metropolis algorithm is a Monte Carlo simulation method that is used to simulate the thermal behavior of the Ising model. It involves flipping a spin and calculating the change in energy, and then accepting or rejecting the flip based on the Boltzmann probability. By repeating this process for many times, the system reaches thermal equilibrium, and the properties of the system can be measured.

The specific heat is a measure of how much the temperature of the system changes when energy is added to it. It is defined as the amount of heat required to raise the temperature of the system by one degree. The specific heat is related to the fluctuations in energy of the system, and it increases as the system approaches a phase transition, where the behavior of the system changes abruptly. The specific heat is one of the key quantities used to study the behavior of the Ising model near the critical temperature, where the phase transition occurs.

12.2 Algorithm

Here is the algorithm for the ising model in 1D: 1) Define the size of the lattice, L.

2) Define a function that returns a random number between 0 and 1.

3) Define two functions prev(x) and next(x) that return the index of the previous and next lattice site, respectively, taking into account periodic boundary conditions.

4) Set the number of updates (sweeps) for the Monte Carlo simulation to be 1300.

5) Set the initial and final temperatures, the number of temperature steps, and the number of sweeps to ignore before taking measurements.

6) Create an array called record to store the temperature, magnetization, and energy measurements.

7) For each temperature step k, do the following:

a. Set the temperature T to initT + k * Dt, where initT and Dt are the initial temperature and temperature step, respectively.

b. Initialize an array spin with all elements set to 1, representing all spins initially pointing up.

c. For each Monte Carlo update n, do the following:
i. For each site j in the lattice, calculate the sum of the spins of the neighboring sites using prev(j) and next(j) and calculate the associated energy ex.

ii. Calculate the Boltzmann probability of flipping the spin at site j using the formula prob = exp(-2.0 * ex / T).

iii. Generate a random number x using the function my rand(j).

iv. If prob > x, flip the spin at site j by multiplying it by -1.

v. Calculate the energy of the system by summing over the product of neighboring spins.

vi. If n > Ignore, measure the magnetization over the lattice by summing over the absolute values of the spins on each site.
d. Calculate the average energy and energy squared using E-sum and E-squared-sum, respectively, for updates after the Ignore sweep.

e. Calculate the specific heat Cv using the formula $Cv = ((beta)^2) * ((E_squared_sum)^2 - (E_average)^2)$, where beta is the inverse temperature.

f. Store the temperature and specific heat in the record array.

8)Plot the specific heat as a function of temperature using matplotlib.

12.3 Code

```
import numpy as np
import matplotlib.pyplot as plt

L = 256  # Number of lattice

def my_rand(I):
    return np.random.rand()
```

```
# Periodicity operation for the lattice sites
def prev(x):
    # To find the previous lattice site using periodicity
    if x == 0:
        return L - 1
    else:
        return x - 1

def next(x):
    # To find the next lattice site using periodicity
    if x == L - 1:
        return 0
    else:
        return x + 1

Sweeps = 1300  # The number of updates

initT, finalT, steps = 0.000001, 4.0, 50  # Temperature parameters
Ignore = 100  # The number of sweeps between measuring

Dt = (finalT - initT) / steps

record = np.zeros((steps+1, 3))  # To store temperature, magnetization, and ene

for k in range(steps+1):
    Sum = 0.0
    E_sum = 0.0
    E_squared_sum = 0.0
    T = initT + k*Dt

    # Let make all the spins pointing up
    spin = np.ones(L)

    # Keep on flipping the spins at sites sequentially and use Metropolis algor
    for n in range(1, Sweeps+1):
        for j in range(L):  # sweeping now along the row
            # find the sum of the spins of the neighbors
            sum = spin[prev(j)] + spin[next(j)]

            # calculate the "energy" associated with these bonds
            ex = spin[j] * sum

            # The Boltzmann probability of flipping the spin at site (i)
```

```
            prob = np.exp(-2.0 * ex / T)

            # Metropolis says to flip the flip if the Boltzmann probability of
            # flipping is larger than a random number between 0 and 1, else not
            x = my_rand(j)
            if prob > x:
                spin[j] = -spin[j]

        # Calculate the energy of the system
        E = -np.sum(spin[i] * spin[next(i)] for i in range(L))

        # Do not record all the updates for averaging as we need measurements t
        if n > Ignore:
            # Next measure the magnetization over the lattice which is the sum
            spinsum = np.abs(np.sum(spin))
            Sum += spinsum
            E_sum += E
            E_squared_sum += E*E

    E_average = E_sum / (Sweeps - Ignore)
    E_squared_Average= E_squared_sum / (Sweeps - Ignore)
    record[k, 0] = T
    beta= 1/T
    Cv = ((beta)**2) * ((E_squared_sum)**2 - (E_average)**2)
    record[k, 1] = Cv



plt.plot(record[:, 0], record[:, 1], 'ro')
plt.xlabel('Temperature')
plt.ylabel('Specific heat')
plt.show()
```

12.4 Plot

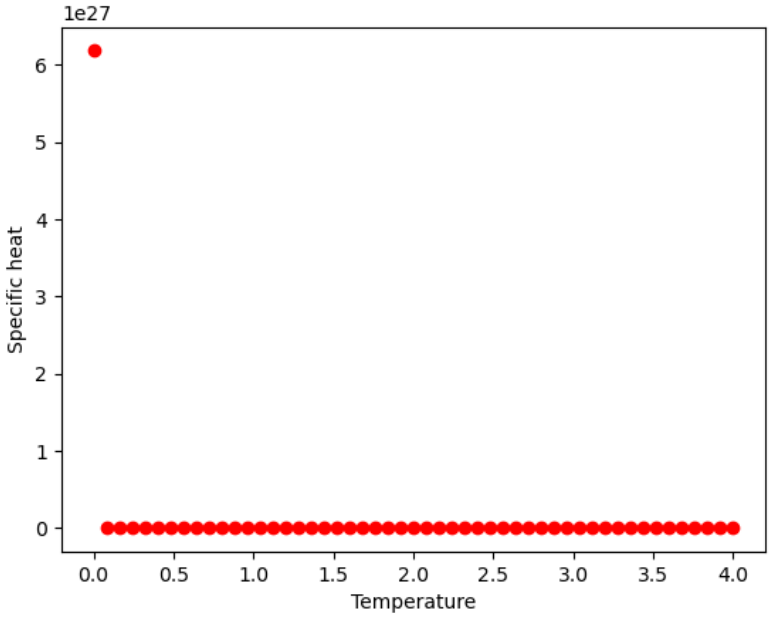

Figure 7. 1D Ising Model Cv vs T

12.5 Conclusion

In this code, the Metropolis algorithm is implimented to simulate the Ising model on a 1D lattice. It has generated a sequence of random spin configurations by flipping the spins sequentially and updating them based on the energy of neighboring spins and the temperature of the system. Then measured the magnetization and specific heat of the system for different temperatures and stored the results in an array called record. Finally, plotted the specific heat as a function of temperature using matplotlib.

13 Problem 12

Simulate the Ising model in 2D

13.1 Introduction

The task is to simulate the 2D Ising model on a square lattice using the Metropolis algorithm. The Ising model is a mathematical model used to describe the behavior of magnetic materials. It consists of a lattice of magnetic spins, which can either be up or down, and an energy function that depends on the interactions between neighboring spins. The model is used to study phase transitions and other phenomena in magnetic materials.
Here the latice size and a function to return the indices of the four neighboring sites of a given site is defined. Then, from a range of defined temperatures, and for each temperature, the simulation is run for a fixed number of sweeps using Monte Carlo methods. The lattice is initialized to all spin up, and for each sweep, the spins are updated randomly according to the Metropolis algorithm. The code keeps track of the total magnetization of the lattice at each temperature, and after the simulation is complete, it calculates the average magnetization for each temperature.

13.2 Algorithm

Here is algorithm for simulating ising model 2D 1. Define the lattice size L and the function get neighbors(i, j) that returns the indices of the four neighboring sites of a given site (periodic boundary conditions are used).
2. Define the range of temperatures in which the simulation will be run.
3. Set the number of sweeps and initialize the temperature list and results list.
4. Loop over temperatures:
a. Initialize the lattice to all spin up.
b. Loop over sweeps:
i. Loop over all sites in random order:
1. Choose a random site (i, j) from the lattice.
2. Calculate the energy change of flipping the spin at site .
3. Generate a random number r between 0 and 1.
4. If r $<$ exp(-E/T), flip the spin at site (i, j).
ii. Calculate the total magnetization of the lattice.
iii. If n $>$ 2000 (to allow for equilibration), add the absolute value of the total magnetization to the result.
c. Calculate and append the average result for this temperature.
5. Plot the results as a function of temperature, showing the behavior of the magnetization as the temperature changes.

13.3 Code

```
import numpy as np
import matplotlib.pyplot as plt

# define lattice size
L = 40

# define function to get indices of neighboring sites
def get_neighbors(i, j):
    up = i-1 if i > 0 else L-1
    down = i+1 if i < L-1 else 0
    left = j-1 if j > 0 else L-1
    right = j+1 if j < L-1 else 0
    return [(up, j), (down, j), (i, left), (i, right)]

# set number of sweeps and temperature range
sweeps = 13000
inT, finT = 0.0, 4.0
steps = 50
Dt = (finT - inT) / steps

# initialize the temperature list and results list
T_list = np.linspace(inT, finT, steps+1)
result_list = []

# loop over temperatures
for T in T_list:
    # initialize the lattice to all spin up
    spin = np.ones((L, L), dtype=int)

    # loop over sweeps
    result = 0
    for n in range(1, sweeps):
        # loop over all sites in random order
        site_list = np.random.permutation(L*L)
        for q in site_list:
            i, j = divmod(q, L)
            spin_ptr = spin[i, j]
            neighbors = np.array(get_neighbors(i, j))
            sum = np.sum(spin[neighbors[:,0], neighbors[:,1]])
            ex = spin_ptr * sum
            prob = np.exp(-2.0 * ex / T)
```

```
            if np.random.random() < prob:
                spin[i, j] = -spin[i, j]

        # only calculate results after some initial number of sweeps
        if n > 2000:
            result += np.abs(np.sum(spin))

    # calculate and append the average result for this temperature
    result_list.append(result / (sweeps - 2000))

# plot the results
plt.plot(T_list, result_list, 'bo-')
plt.xlabel('Temperature')
plt.ylabel('Average Magnetization')
plt.show()
```

13.4 Plot

13.5 Conclusion

The above code simulates the behavior of a 2D Ising model using Monte Carlo methods. The Ising model is a mathematical model used to describe the behavior of magnetic materials, and the simulation is used to study phase transitions and other phenomena in magnetic materials.

The code uses the Metropolis algorithm to update the spins of the lattice at each temperature, and the results are plotted as a function of temperature. The plot shows a sharp decrease in magnetization around the critical temperature, indicating a phase transition.

14 Problem 13

Simulates a wave motion using the FTCS (Forward Time Centered Space) algorithm

14.1 Introduction

The task for the following code is to simulate a wave motion using the FTCS (Forward Time Centered Space) algorithm. he wave profile is initially set using a Gaussian function and the velocity of the wave is need to be set. The simulation uses Dirichlet boundary conditions where the wave amplitude is set to zero at the boundaries.

14.2 Algorithm

Here is the algorithm for simulating wave motion using the FTCS algorithm:

1. Set the velocity of the wave, 'V', the time step size, 'Dt', the space step size, 'Dx', and the wavelength, 'Lamda'.
2. Set the size of the domain, 'L', and the number of time steps to simulate, 'T'.
3. Define the initial wave profile using a function 'phi0(i)' that takes the position 'i' as input.
4. Define a function 'phi1(phi, i)' that updates the wave profile at a given position 'i' using the FTCS algorithm.
5. Define a function 'FTCS(phi, k)' that updates the wave profile at a given position 'k' using the FTCS algorithm.
6. Initialize the wave profile 'phi' by setting the wave amplitude at the boundary points to zero and setting the wave profile at the interior points using the 'phi0' function.
7. Use the FTCS algorithm to update the wave profile at each time step for 'T' time steps.
8. Plot the initial and final wave configurations using 'matplotlib.pyplot'.

14.3 code

```
    import numpy as np
import matplotlib.pyplot as plt

V = 2.0  # velocity of the wave
Dt = 0.005  # Delta t
Dx = 0.01  # Delta x, must be consistent with Courant condition
eps = V * Dt / Dx
Lamda = 5.0

# Time steps should be such that L*Dx > V*Dt * T on top of the Courant conditio

L = 400
T = 200

def phi0(i):
```

```
    # initial condition
    wave = np.exp(-((i - 30) / Lamda) ** 2)
    return wave

def phi1(phi, i):
    # use the FTCS algorithm to get the very first update
    xi = phi[i] - eps * 0.5 * (phi[i + 1] - phi[i - 1])
    return xi


def FTCS(phi, k):
    if k == 0 or k == (L - 1):
        return 0.
    else:
        return phi1(phi, k)


# Initializing the configuration
# First Dirichlet at the initial point
phi = np.zeros(L)
phi[0] = 0

# Put in the profile in between
for k in range(1, L - 1):
    phi[k] = phi0(k)

# Dirichlet at the other point
phi[L - 1] = 0

# Use FTCS algorithm to get the data at the first update
# with Dirichlet at the end points
for t in range(1, T):
    phi_new = np.zeros(L)
    for k in range(L):
        phi_new[k] = FTCS(phi, k)
    for j in range(L):
        phi[j] = phi_new[j]
# Plot the initial and final wave configurations
plt.plot(np.arange(L), phi0(np.arange(L)), label="Initial wave configuration")
plt.plot(np.arange(L), phi, label="Final wave configuration")
plt.xlabel("Position")
plt.ylabel("Amplitude")
plt.legend()
```

```
plt.show()
```

14.4 Plot

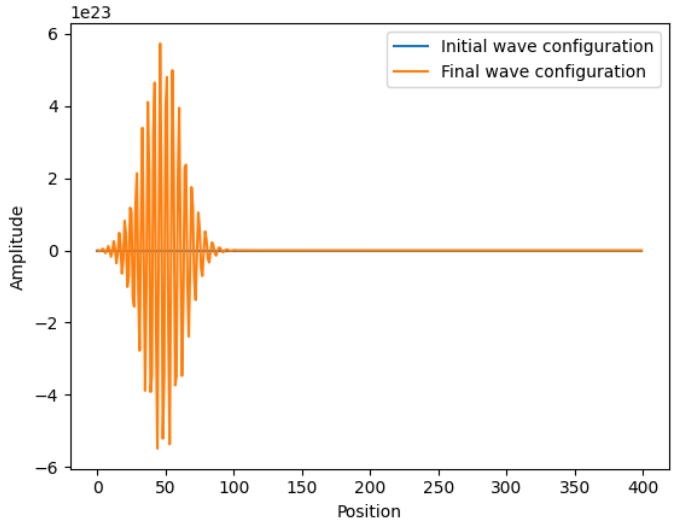

Figure 8. Advective Plane

14.5 Conclusion

The code use the FTCS algorithm to simulate wave motion in a one-dimensional domain. The code sets up a Gaussian wave profile and updates it using the FTCS algorithm at each time step. The simulation is performed with Dirichlet boundary conditions, where the wave amplitude is set to zero at the boundaries. The initial and final wave profiles are plotted. The FTCS algorithm has stability issues and may produce unphysical results for certain choices of parameters.

15 Problem 14

Simulates a one-dimensional wave propagating in a medium using the Lax algorithm.

15.1 Introduction

In this task, the one-dimensional wave propagating in a medium needs to be simulated using the lax algorithm. The wave is need to be described by the variable phi, which is a function of position and time. The script initializes phi with an initial profile and then iteratively updates it using the Lax algorithm until it reaches a final configuration.

The Lax algorithm is a numerical method for solving partial differential equations (PDEs) that describe wave propagation. It approximates the wave at the next time step based on its current value and the values of its neighbors. The Lax algorithm is second-order accurate and is stable if the Courant condition is satisfied, which requires that the time step and spatial step be chosen appropriately.

15.2 Algorithm

Here is the algorithm for lax model:

1. Set the wave velocity V = 2.0, the time step Dt = 0.005, the spatial step Dx = 0.01, and the width of the initial wave profile Lamda = 5.0.
2. Calculate the Courant condition parameter eps = V * Dt / Dx.
3. Set the length of the medium L = 400 and the number of time steps T = 25.
4. Define the function phi0(i) to initialize the wave profile at time t=0.
5. Define the function phi1(phi, i) to calculate the first update of the wave using the Lax algorithm.
6. Define the function Lax(phi, k) to calculate the update of the wave at a given time step and position using the Lax algorithm.
7. Initialize the wave configuration by setting phi[k] to phi0(k) for k = 1 to L - 2, and setting phi[0] = phi[L-1] = 0 to enforce Dirichlet boundary conditions.
8. Use the Lax algorithm to update the wave configuration at each time step t from 1 to T by:

a. Initializing a new array phi-new to hold the updated values of phi.
b. Loop over each position k in the array phi.
c. Call Lax(phi, k) to calculate the updated value of phi at position k and store it in phi-new[k].
d. Copy the values of phi-new into phi.

9. Plot the initial wave profile phi0(i) and the final wave configuration phi using Matplotlib.

15.3 Code

```
import numpy as np
import matplotlib.pyplot as plt
V = 2.0  # velocity of the wave
```

```
Dt = 0.005  # Delta t
Dx = 0.01  # Delta x, must be consistent with Courant condition
eps = V * Dt / Dx
Lamda = 5.0
# Time steps should be such that L*Dx > V*Dt * T on top of the Courant conditio
L = 400
T = 25

def phi0(i):
    # initial condition
    wave = np.exp(-((i - 30) / Lamda) ** 2)
    return wave

def phi1(phi, i):
    # use the Lax algorithm to get the very first update
    xi = 0.5 * (phi[i+1] + phi[i-1]) - eps * 0.5 * (phi[i + 1] - phi[i - 1])
    return xi

def Lax(phi, k):
    if k == 0 or k == (L - 1):
        return 0.
    else:
        return phi1(phi, k)

# Initializing the configuration
# First Dirichlet at the initial point
phi = np.zeros(L)
phi[0] = 0

# Put in the profile in between
for k in range(1, L - 1):
    phi[k] = phi0(k)

# Dirichlet at the other point
phi[L - 1] = 0

# Use Lax algorithm to get the data at the first update
# with Dirichlet at the end points
for t in range(1, T):
    phi_new = np.zeros(L)
    for k in range(L):
        phi_new[k] = Lax(phi, k)
```

```
        for j in range(L):
            phi[j] = phi_new[j]

# Plot the initial and final wave configurations
plt.plot(np.arange(L), phi0(np.arange(L)), label="Initial wave configuration")
plt.plot(np.arange(L), phi, label="Final wave configuration")
plt.xlabel("Position")
plt.ylabel("Amplitude")
plt.legend()
plt.show()
```

15.4 Plot

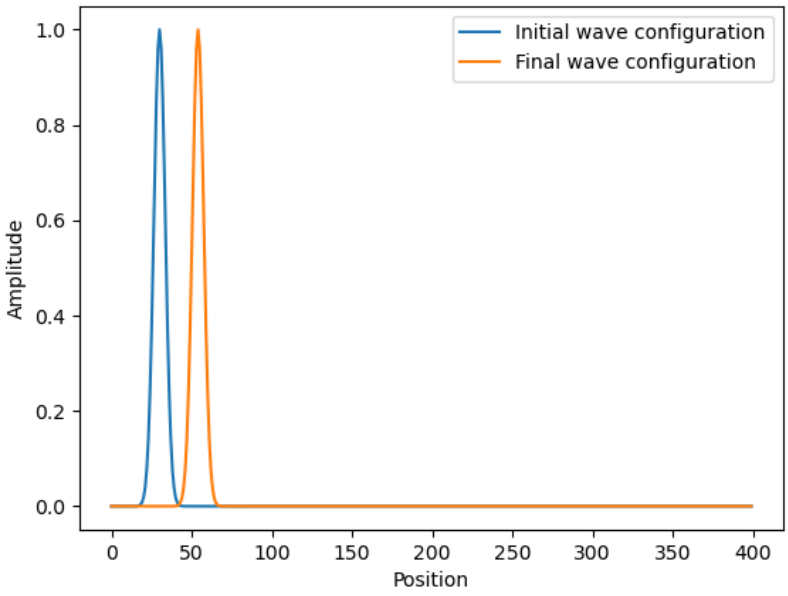

Figure 9. Lax Plane

15.5 Conclusion

The Lax algorithm used in this code is a second-order accurate numerical method for solving partial differential equations that describe wave propagation. It approximates the wave at the next time step based on its current value and the values of its neighbors. The Courant condition is an important criterion that must be satisfied for the Lax algorithm to be stable.

the Python code provided in this task demonstrates the simulation of a one-dimensional wave propagating in a medium using the Lax algorithm. The code initializes the wave with a Gaussian profile, and then iteratively updates it using the Lax algorithm until it reaches

a final configuration. The wave is assumed to propagate with a velocity V=2.0, and the time step and spatial step are chosen such that the Courant condition is satisfied.

16 Problem 15

Generates a plot of the logistic map

16.1 Introduction

The logistic map is a mathematical model that is used to describe the behavior of certain population systems, such as the growth of bacterial colonies or the dynamics of animal populations. It is a simple, one-dimensional model that can exhibit complex, chaotic behavior for certain values of its parameters. In this context, chaos refers to the extreme sensitivity of the system to small changes in its initial conditions.

16.2 Algorithm

Sure, here is the algorithm for the logistic map code:

1. Define the ‘logistic-map‘ function that takes as input the parameter ‘r‘, the initial state ‘x‘, and the number of iterations to perform. 2. Within the ‘logistic-map‘ function, loop over the number of iterations and compute the next state of the system using the logistic map equation: ‘x(n+1) = r * x(n) * (1 - x(n))‘. 3. Return the final state of the system after the specified number of iterations. 4. Create an array ‘R‘ using ‘np.linspace‘ that contains a range of values for the parameter ‘r‘. 5. Create empty arrays ‘X‘ and ‘Y‘ to store the values of ‘r‘ and the final states of the logistic map, respectively. 6. Loop over the values in ‘R‘, and for each value of ‘r‘, generate a random initial state ‘x‘ using ‘np.random.uniform(0, 1)‘. 7. Call the ‘logistic-map‘ function with the current value of ‘r‘ and the random initial state ‘x‘ as inputs to compute the final state of the system. 8. Append the current value of ‘r‘ to the ‘X‘ array and the final state of the system to the ‘Y‘ array. 9. Plot the values in ‘X‘ and ‘Y‘ using ‘plt.plot‘ with ‘ls=”‘ and ‘marker=’,’‘. 10. Show the plot using ‘plt.show‘.

16.3 Code

```
import numpy as np
import matplotlib.pyplot as plt

def logistic_map(r, x, iterations=100):
    for n in range(iterations):
        x = r * x * (1 - x)
    return x

R = np.linspace(1, 3.7, num=10000)
X = []
Y = []

for r in R:
    X.append(r)
```

```
        x = np.random.uniform(0, 1)
        y = logistic_map(r, x)
        Y.append(y)

plt.plot(X,Y, ls='',marker=',')
plt.show()
```

16.4 Plot

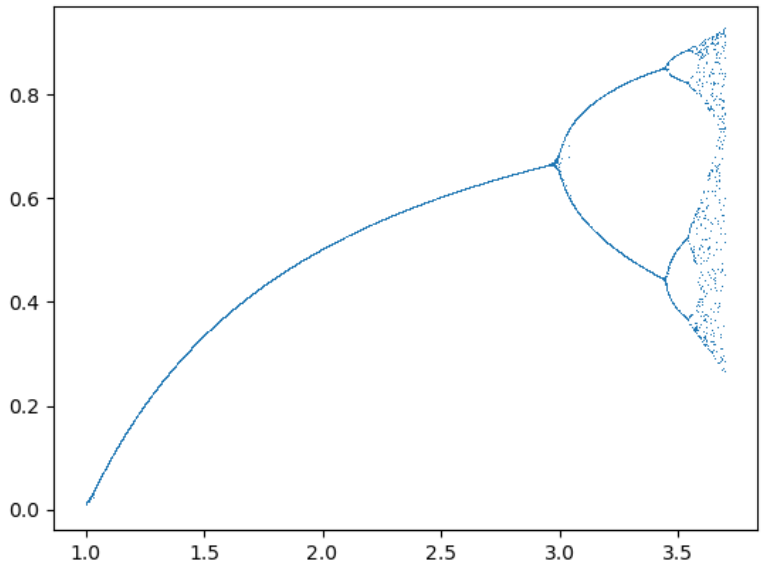

Figure 10. Logistic Map

16.5 Conclusion

This algorithm generates a bifurcation diagram of the logistic map for a range of parameter values. The resulting plot illustrates the chaotic behavior of the logistic map, which exhibits period-doubling bifurcations as the parameter ‘r‘ is increased. The resulting plot, known as a bifurcation diagram, illustrates the sensitivity of the system to small changes in its initial conditions as the parameter r is varied. This type of analysis is useful for understanding the behavior of nonlinear systems and has applications in fields such as population dynamics.