# Exercise 4.10: Linear Interpolation

Asdf

## Part 1

# if t = 3.2 then i = 3 will satisfy the requirement:

# t\_3 <= 3.2 <= t\_4

## Part 2

# A mathematical expression of a function of a line between two points:

# Given that y(x) is a function that is discontinuous and only defined at integer values of x.

# f(x) = y(i) + (x-i)\*(y(i+1)-y(i))/(i+1-i)

# -> f(x,i) = y(i) + (x-i)\*(y(i+1)-y(i))

## Part 3

# The "y value" calculated by the user's time value x and i being the floor of x, can be calculated as f(x,i):

# i = 3, x = 3.2

# -> f(3.2,3) = y(3) + (3.2-3)\*(y(3+1)-y(3))

# -> f(3.2,3) = y(3) + (0.2)\*(y(4)-y(3))

## Part 3 a

# Implementation of linear interpolation

def lin\_interpolate(dataset: list[float], floatIndex: float) -> float:

    """Function that finds a float number between entries of a list with linear interpolation

    Args:

        dataset (list[float]): any list of numbers

        floatIndex (float): a number representing a space between entries in dataset above

    Returns:

        float: linearly interpolated float

    """

    if(floatIndex <= len(dataset)-1 and floatIndex >= 0):

        index\_1 = int(math.floor(floatIndex))

        index\_2 = index\_1+1

        value\_1 = dataset[index\_1]

        value\_2 = dataset[index\_2]

        fraction = floatIndex-index\_1

        interpolated\_number = value\_1 + (value\_2-value\_1)\*fraction

        return interpolated\_number

## Part 3 b

# Function that prints interpolated values of y at times requested by the user

def find\_y(dataset: list[float]) -> None:

    """Function that prints interpolated values of y at times requested by the user

    Args:

        dataset (list[float]): Requires a set of data to perform interpolation on

    Returns:

        [type]: No return

    """

    run = True

    while(run):

        print(dataset)

        val = input(

            f'Enter a pseudo index (float), between 0 and {len(dataset)-1}, and I will return a linearly interpolated point from above dataset:')

        try:

            val = float(val)

            print(val)

            if(val < 0):

                run = False

            elif(val >= 0):

                print(

                    f'The interpolated point is: y = {lin\_interpolate(dataset, val)} at x = {val}')

        except:

            print('An exception occurred')

    else:

        print("End of function.")

## Part 3 c

y = [4.4, 2.0, 11.0, 21.5, 7.5]

print(lin\_interpolate(y, 2.5)) #Result: 16.25

print(lin\_interpolate(y, 3.1)) #Result: 20.099999999999998

# Exercise 4.12: Fit Straight Line to Data

asdf

## Part a

def error\_sum\_points\_and\_line(dataset: list[float], a: float, b: float) -> float:

    """sums the square of error between points in a list and a linear function.

    Assumes that x increases by 1 per value of y.

    Args:

        dataset (list[float]): list of datapoints.

        a (float): coefficient of a straight line function.

        b (float): constant part of a straight line function.

    Returns:

        float: sum of squared error.

    """

    error = 0

    for num, y in enumerate(dataset):

        f\_x = a\*num+b

        error += (f\_x-y)\*\*2

    return error

## Part b

def user\_interaction(dataset: dict[list[float], list[float]]) -> None:

    """Takes a dictionary of datapoints, queries the user for values of a and b to create a linear function, and prints the datapoints and the function to a plot.

    Args:

        dataset (dict[list[float], list[float]]): Takes a dictionary with lists of measurements of x and y values

    """

    run = True

    while(run):

        print(f'Enter values of a and b for the function f(x) = a\*x + b to calculate the summed square of error compared to the dataset.')

        print(f'dataset: {dataset["y"]}')

        print(type(dataset["x"]))

        a = input("a: ")

        b = input("b: ")

        print(

            f'Sum of squared error: {error\_sum\_points\_and\_line(dataset["y"],float(a),float(b))}')

        test = list(map(lambda x: float(a)\*x+float(b), dataset["x"]))

        print(test)

        figure = plt.figure()

        ax = figure.add\_subplot(1, 1, 1)

        ax.scatter(dataset["x"], dataset["y"],

                   color='tab:blue', label='Measurements')

        ax.plot(dataset["x"], test, color='tab:orange',

                label='Regression attempt')

        ax.legend()

        plt.show()

        cont = input("Continue [Y/N]?\n")

        if(cont.lower() == "n"):

            run = False

Result:

Chart

Description automatically generated

## Part c

Found a smaller error: Chart, scatter chart

Description automatically generated

# Exercise 5.6: Area of a Polygon

## Function:

# Shoelace algorithm

def polyarea(x: list[float], y: list[float]) -> float:

    if(len(x) == len(y)):

        area = 0

        for i in range(len(x)):

            index = i

            index\_plus\_one = i+1 < len(y) and i+1 or 0

            area += x[index]\*y[index\_plus\_one]

            area -= y[index]\*x[index\_plus\_one]

        area = (1/2)\*abs(area)

        return area

    else:

        print("inputs are required to be of the same length.")

## Results:

# Square with area of 25 -> function gives expected output

""" x = [0,5,5,0]

y = [0,0,5,5] """

# Triangle with area of 12.5 -> function gives expected output

""" x = [0,5,5]

y = [0,0,5] """

# quadrilateral with area of 57 -> function gives expected output

""" x = [2, 11, 11, 4]

y = [2, 2, 8, 10]

 """

# polygon of five vertices with an area of 30 -> function gives expected output

x = [3, 5, 9, 12, 5]

y = [4, 6, 5, 8, 11]

print(polyarea(x, y))

# Exercise 6.10: Definite integral of xx between 0 and 4

from matplotlib.pyplot import step

import numpy as np

# integrate x\*\*x from 0 to 4 with four decimal precision

# middlepoint method

def numeric\_integration\_middlepoint(integrand: str, step\_amount: int, start: float, stop: float) -> float:

    """Calculates definite integral of x\*\*x from amount of steps (resolution), start and stop values.

    Args:

        integrand (str): UNUSED (string representation of integrand)

        step\_amount (int): Amounts of steps or slices to divide the function into

        start (float): start value of definite integral

        stop (float): stop value of definite integral

    Returns:

        float: numeric result of integration

    """

    steps = np.linspace(start, stop, step\_amount)

    half\_step = (steps[1]-steps[0])/2

    definite\_integral = 0;

    for i in range(len(steps)-1):

        x = steps[i]+half\_step

        definite\_integral += (x\*\*x)\*(2\*half\_step)

    return definite\_integral

print(f'{numeric\_integration\_middlepoint("x\*\*x", 1000, 0, 4) = }')

My result:

Text

Description automatically generated

Calculation took a few seconds, maybe a minute

Wolfram Alpha:

Graphical user interface, text, application, email, Teams

Description automatically generated

Maple:

Text

Description automatically generated with medium confidence

# Exercise 7.7: Fixed Point Iteration

## a)

def newtons\_method(x\_initial: float, lower\_bound: float = -2, upper\_bound: float = 2, iterations: int = 10):

    x = np.zeros(iterations+1)

    x[0] = x\_initial

    for n in range(iterations):

        x[n+1] = x[n] - (function(x[n])/diff\_function(x[n]))

        if x[n+1] > upper\_bound:

            x[n+1] = upper\_bound

        elif x[n+1] < lower\_bound:

            x[n+1] = lower\_bound

    return x

def function(x: float) -> float:

    return (x\*\*3)+2\*x-math.e\*\*(-x)

def diff\_function(x: float) -> float:

    return 3\*(x\*\*2)+2+math.e\*\*(-x)

result:

Chart, line chart

Description automatically generated

## b)

def fixed\_point\_iteration(x\_initial: float = 1, lower\_bound: float = -2, upper\_bound: float = 2, iterations: int = 10, tolerance\_limit: float = 0.05):

    x\_n = np.zeros(iterations+1)

    x\_n[0] = x\_initial

    for i in range(iterations):

        x\_n[i+1] = function\_fixed\_point\_form(x\_n[i])

    print(f'{x\_n = }')

    return x\_n

result

Chart, line chart

Description automatically generated

# Exercise 8.20

import numpy as np

import matplotlib.pyplot as plt

*# Time unit: 1 h*

β = 10./(40\*8\*24)

γ = 3./(15\*24)

dt = 10 *# 6 min*

D = 30 *# Simulate for D days*

N\_t = int(D\*24/dt) *# Corresponding no. of time steps*

t = np.linspace(0, N\_t\*dt, N\_t+1)

S = np.zeros(N\_t+1)

I = np.zeros(N\_t+1)

R = np.zeros(N\_t+1)

S\_heun = np.zeros(N\_t+1)

I\_heun = np.zeros(N\_t+1)

R\_heun = np.zeros(N\_t+1)

*# Initial conditions*

S[0] = 50

S\_heun[0] = 50

I[0] = I\_heun[0] = 1

R[0] = R\_heun[0] = 0

*#differential equations*

def diff\_susceptible(S, I):

    return -β\*S\*I

def diff\_infectious(S,I):

    return β\*S\*I - γ\*I

def diff\_recovered(I):

    return γ\*I

*# Step equations forward in time*

for n in range(N\_t):

    S[n+1] = S[n] + dt\*diff\_susceptible(S[n], I[n])

    I[n+1] = I[n] + dt\*diff\_infectious(S[n], I[n])

    R[n+1] = R[n] + dt\*diff\_recovered(I[n])

*# Step equations heun*

for n in range(N\_t):

*#next y = current y + half\_timestep \* (differentiated\_current+approx\_differentiated\_next)*

*#y[i+1] = y[i] + (h/2)\*(y'(x[i],y[i]) + y'(x[i]+h,y[i]+h\*y'(x[i],y[i])))*

    S\_diff\_n = diff\_susceptible(S\_heun[n], I\_heun[n])

    I\_diff\_n = diff\_infectious(S\_heun[n], I\_heun[n])

    R\_diff\_n = diff\_recovered(I\_heun[n])

    S\_diff\_next = diff\_susceptible(S\_heun[n]+dt\*S\_diff\_n, I\_heun[n]+dt\*I\_diff\_n)

    I\_diff\_next = diff\_infectious(S\_heun[n]+dt\*S\_diff\_n, I\_heun[n]+dt\*I\_diff\_n)

    R\_diff\_next = diff\_recovered(I\_heun[n]+dt\*I\_diff\_n)

    S\_heun[n+1] = S\_heun[n] + (dt/2)\*(S\_diff\_n+S\_diff\_next)

    I\_heun[n+1] = I\_heun[n] + (dt/2)\*(I\_diff\_n+I\_diff\_next)

    R\_heun[n+1] = R\_heun[n] + (dt/2)\*(R\_diff\_n+R\_diff\_next)

fig = plt.figure()

l1, l2, l3, l4, l5, l6 = plt.plot(t, S, t, I, t, R, t, S\_heun, t, I\_heun, t, R\_heun)

fig.legend((l1, l2, l3, l4,l5,l6), ('S', 'I', 'R', 'S\_heun', 'I\_heun', 'R\_heun'), 'center right')

plt.xlabel('hours')

plt.savefig('tmp.svg')

plt.show()

import numpy as np

import matplotlib.pyplot as plt

*# Time unit: 1 h*

β = 10./(40\*8\*24)

γ = 3./(15\*24)

dt = 10 *# 6 min*

D = 30 *# Simulate for D days*

N\_t = int(D\*24/dt) *# Corresponding no. of time steps*

t = np.linspace(0, N\_t\*dt, N\_t+1)

S = np.zeros(N\_t+1)

I = np.zeros(N\_t+1)

R = np.zeros(N\_t+1)

S\_heun = np.zeros(N\_t+1)

I\_heun = np.zeros(N\_t+1)

R\_heun = np.zeros(N\_t+1)

*# Initial conditions*

S[0] = 50

S\_heun[0] = 50

I[0] = I\_heun[0] = 1

R[0] = R\_heun[0] = 0

*#differential equations*

def diff\_susceptible(S, I):

    return -β\*S\*I

def diff\_infectious(S,I):

    return β\*S\*I - γ\*I

def diff\_recovered(I):

    return γ\*I

*# Step equations forward in time*

for n in range(N\_t):

    S[n+1] = S[n] + dt\*diff\_susceptible(S[n], I[n])

    I[n+1] = I[n] + dt\*diff\_infectious(S[n], I[n])

    R[n+1] = R[n] + dt\*diff\_recovered(I[n])

*# Step equations heun*

for n in range(N\_t):

*#next y = current y + half\_timestep \* (differentiated\_current+approx\_differentiated\_next)*

*#y[i+1] = y[i] + (h/2)\*(y'(x[i],y[i]) + y'(x[i]+h,y[i]+h\*y'(x[i],y[i])))*

    S\_diff\_n = diff\_susceptible(S\_heun[n], I\_heun[n])

    I\_diff\_n = diff\_infectious(S\_heun[n], I\_heun[n])

    R\_diff\_n = diff\_recovered(I\_heun[n])

    S\_diff\_next = diff\_susceptible(S\_heun[n]+dt\*S\_diff\_n, I\_heun[n]+dt\*I\_diff\_n)

    I\_diff\_next = diff\_infectious(S\_heun[n]+dt\*S\_diff\_n, I\_heun[n]+dt\*I\_diff\_n)

    R\_diff\_next = diff\_recovered(I\_heun[n]+dt\*I\_diff\_n)

    S\_heun[n+1] = S\_heun[n] + (dt/2)\*(S\_diff\_n+S\_diff\_next)

    I\_heun[n+1] = I\_heun[n] + (dt/2)\*(I\_diff\_n+I\_diff\_next)

    R\_heun[n+1] = R\_heun[n] + (dt/2)\*(R\_diff\_n+R\_diff\_next)

fig = plt.figure()

l1, l2, l3, l4, l5, l6 = plt.plot(t, S, t, I, t, R, t, S\_heun, t, I\_heun, t, R\_heun)

fig.legend((l1, l2, l3, l4,l5,l6), ('S', 'I', 'R', 'S\_heun', 'I\_heun', 'R\_heun'), 'center right')

plt.xlabel('hours')

plt.savefig('tmp.svg')

plt.show()

Result:

A picture containing diagram

Description automatically generated

Figure Step size dt=0.1

A picture containing diagram

Description automatically generated

Figure Step size dt = 5

Chart

Description automatically generated with low confidence

Figure Step size dt = 10

# Exercise 8.24

**Exercise 8.24: Solving a Nonlinear ODE with Backward Euler**

Let *y* be a scalar function of time *t* and consider the nonlinear ODE

*Logo, company name

Description automatically generated*

a) Assume you want to solve this ODE numerically by the Backward Euler method.

Derive the computational scheme and show that (contrary to the Forward Euler

scheme) you have to *solve a nonlinear algebraic equation* for each time step

when using this scheme.

b) Implement the scheme in a program that also solves the ODE by a Forward

Euler method.With Backward Euler, use Newton’s method to solve the algebraic

equation. As your initial guess, you have one good alternative, which one?

Let your program plot the two numerical solutions together with the exact

solution, which is known (e.g., fromWolfram Alpha) to be

Diagram, schematic, box and whisker chart

Description automatically generated with medium confidence

Filename: nonlinBE.py.

## Part a

## Part b