Exercise 4.10: Linear Interpolation

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: (i, y_i) and (i+1, y_i+1)
# Such function is usually expressed on the form f(x) = ax + b
# a is the average change between the points: a = (y_i+1-y_i)/(i+1-i) = y_i+1-y_i
# b is found by inserting all the known information into the function f(x) = ax + b:
# (i, y_i) => f(i) = (y_i+1-y_i)*i + b => b = y_i - (y_i+1-y_i)*i
# finally giving: f(x) = y_i + (x-i)*(y_i+1-y_i)
# or more generally: 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 us
er

    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, va
l)} 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]
```

```python
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

Part a

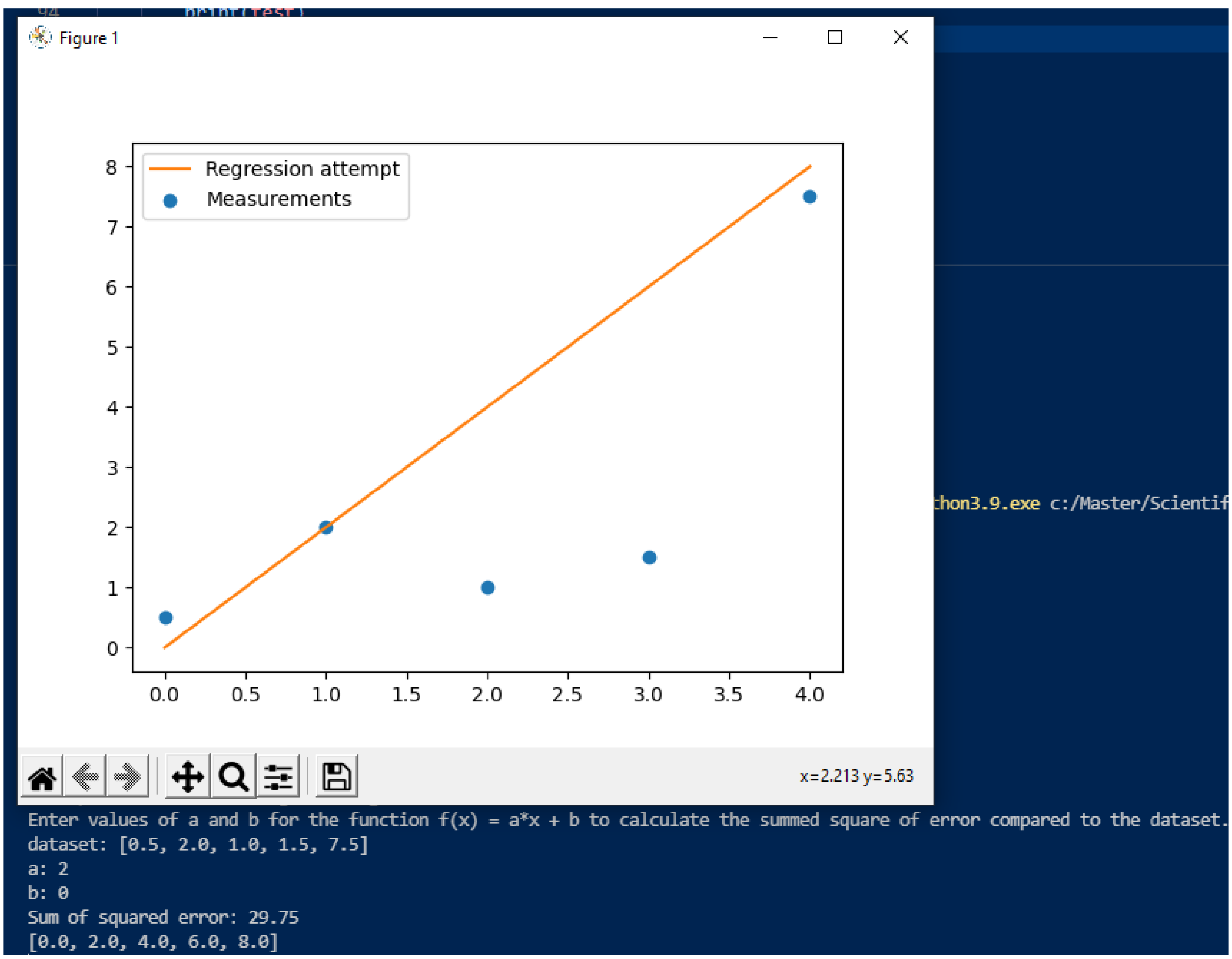
```python
def error_sum_points_and_line(dataset: list[float], a: float, b: float) -> float:
    """sums the square of error between points in a list and a linear function.
    Assumes that x increases by 1 per value of y.

    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

```python
def user_interaction(dataset: dict[list[float], list[float]]) -> None:
    """Takes a dictionary of datapoints, queries the user for values of a and b t
o 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 o
f 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 calcul
ate 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))}')
```

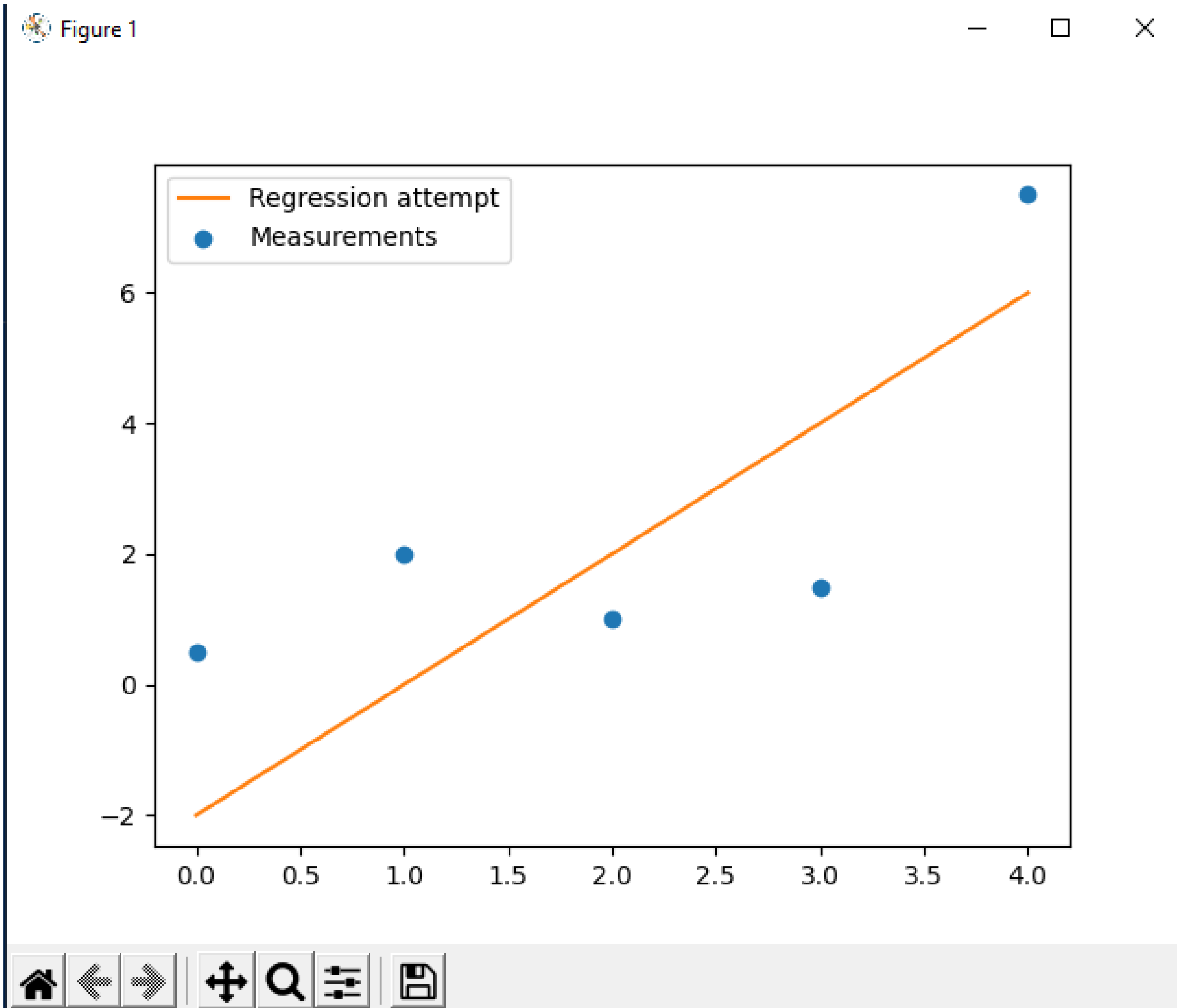
```
    test = list(map(lambda x: float(a)*x+float(b), dataset["x"]))
    print(test)

    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:



Part c

Found a smaller error:




```
Enter values of a and b for the function f(x) = a*x + b to calculate the summed square of e
dataset: [0.5, 2.0, 1.0, 1.5, 7.5]
a: 2
b: -2
Sum of squared error: 19.75
[-2.0, 0.0, 2.0, 4.0, 6.0]
```

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 x^x 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: floa
t, stop: float) -> float:
    """Calculates definite integral of x**x from amount of steps (resolution), st
art 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:

```
PS C:\Master\Scientific-computing> & C:/Users/larsr/AppData/Local/Microsoft/WindowsApps/python3.9.exe c:/Master/Scientific-computing/Assignment/integrate_x2x.py
numeric_integration_middlepoint("x**x", 100_000_000, 0, 4)
 = 114.11906219403006
```

Calculation took a few seconds, maybe a minute.

Wolfram Alpha:

Definite integral | More digits

$$\int_0^4 x^x \, dx \approx 114.119\ldots$$

Download Page

POWERED BY THE **WOLFRAM LANGUAGE**

Maple:

$$int\left(x^{x}, x=0\,..4,\ numeric\right)$$

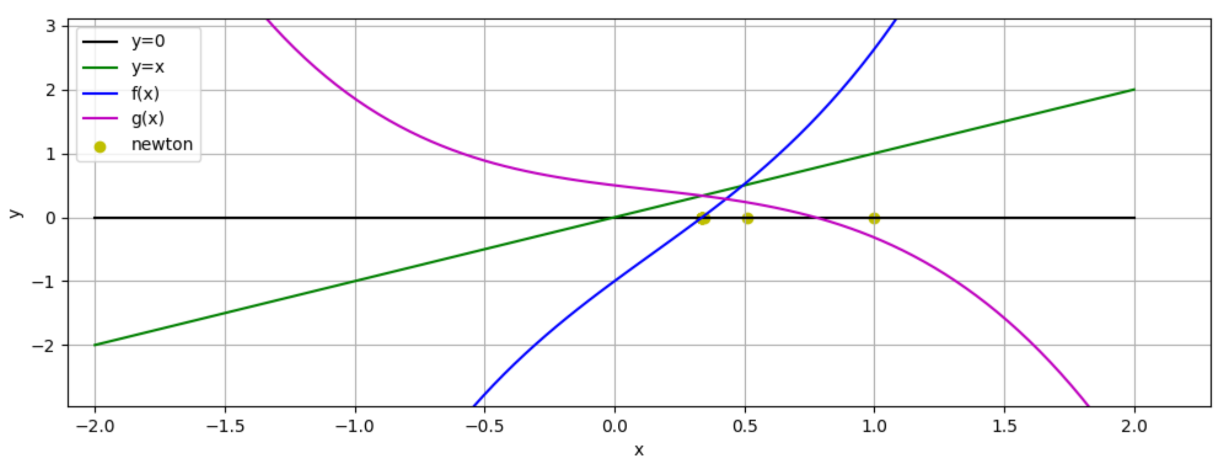
$$114.1190622 \qquad (1)$$

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)
```

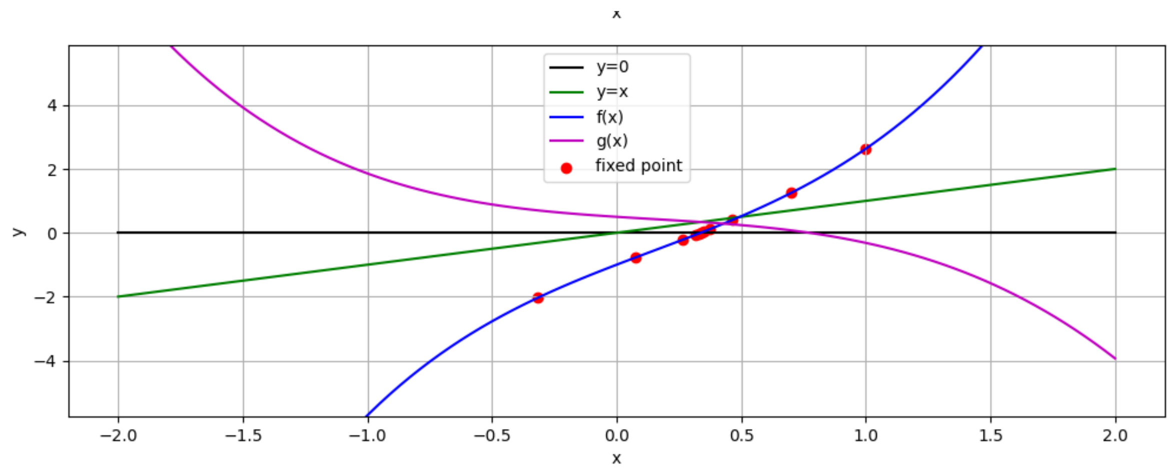
result:



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



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
```

```python
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
```

```python
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:

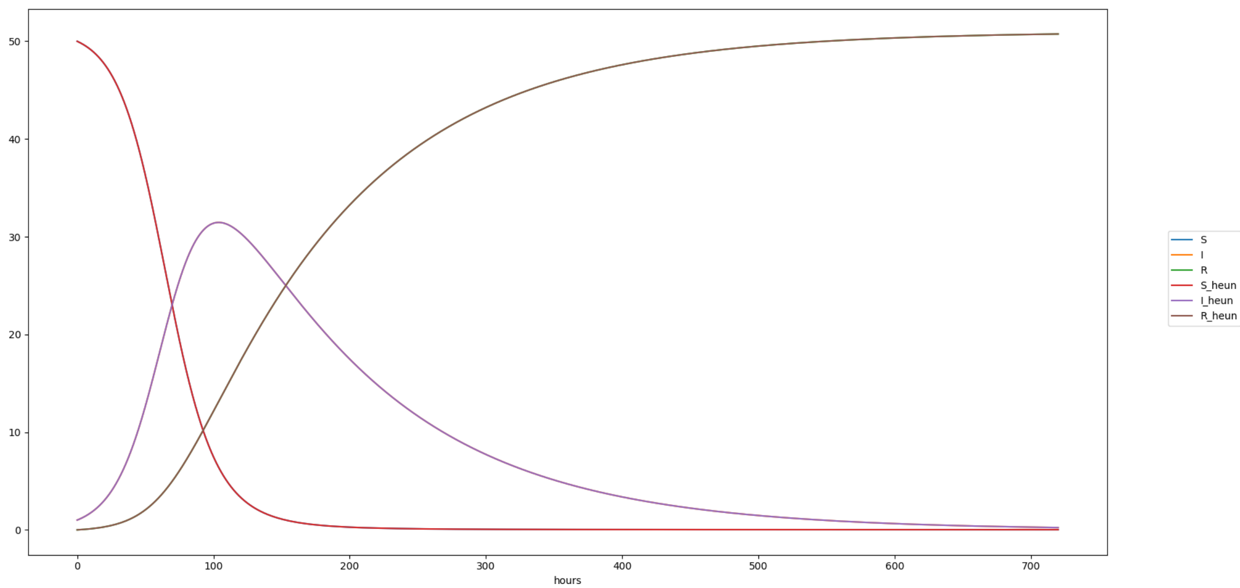


Figure 1 Step size dt=0.1

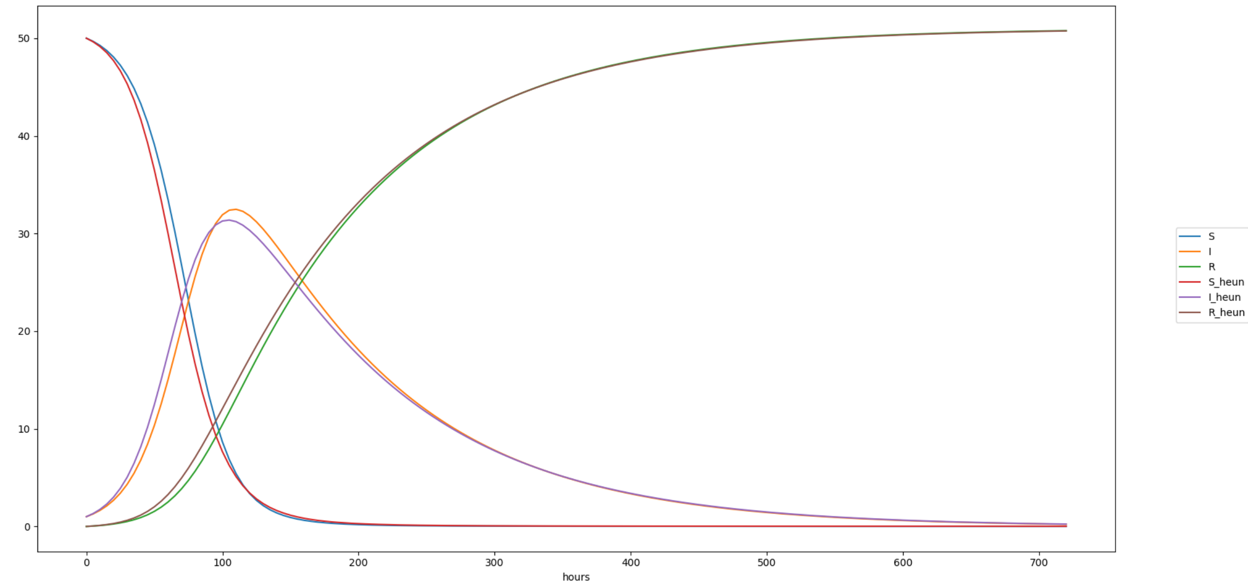


Figure 2 Step size dt = 5

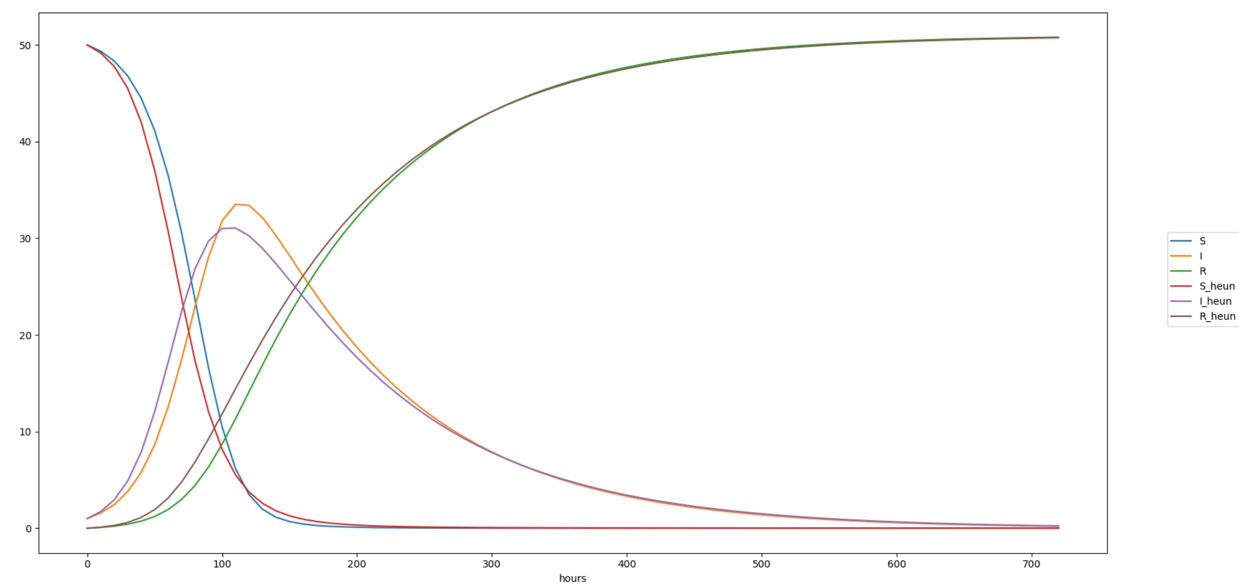


Figure 3 Step size dt = 10

Exercise 8.24

Part a

We have the ODE:

$$y' + y = t y^3, \quad t \in (0, 4), \quad y(0) = \frac{1}{2}$$

Bacward Euler gives:

$$y_{i+1} = y_i + \Delta t \left(t_{i+1} y_{i+1}^3 - y_{i+1} \right)$$

y_{i+1} on both sides, so the eq. needs to be solved for y_{i+1} for each t_{i+1}.

Newton's method can be used for this.

Part b

Code

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

start = 0
end = 4
dt = 0.25
resolution = int((end-start)/dt)
timeline = np.linspace(start, end, resolution+1)
y_BE = np.zeros(resolution+1)
```

```
y_BE[0] = 1/2
y_FE = np.zeros(resolution+1)
y_FE[0] = 1/2
y_Analytical = np.zeros(resolution+1)
y_Analytical[0] = 1/2


def analytic_function(t):
    return np.sqrt(2)/(np.sqrt(7*(np.exp(2*t))+2*t+1))

def function(y_new, y_old, t):
    return y_new - (y_old + dt*(t*(y_new**3) - y_new))

def diff_function(y_new, y_old, t):
    return 1 - (y_old + dt*(3*t*(y_new**2)-1))

def newtons_method(x_initial: float = 2, y_old: float = 1/2, t: float = 0,
lower_bound: float = -2000, upper_bound: float = 2000, iterations: int = 20):
    x = np.zeros(iterations+1)
    x[0] = x_initial
    for n in range(iterations):
        x[n+1] = x[n] - (function(x[n],y_old,t)/diff_function(x[n],y_old,t))
        if x[n+1] > upper_bound:
            x[n+1] = upper_bound
        elif x[n+1] < lower_bound:
            x[n+1] = lower_bound
    return x

for n in range(resolution):
    why = newtons_method(y_BE[n], y_BE[n], timeline[n+1])
    print(f'{why[-1] = }')
    y_BE[n+1] = y_BE[n]+dt*(timeline[n+1]*(why[-1]**3)-why[-1])

for n in range(resolution):
    y_FE[n+1] = y_FE[n]+dt*(timeline[n]*(y_FE[n]**3)-y_FE[n])

for n in range(resolution):
    y_Analytical[n+1] = analytic_function(timeline[n+1])

fig = plt.figure()
l1, l2, l3 = plt.plot(timeline, y_BE, timeline, y_FE, timeline, y_Analytical)
fig.legend((l1, l2, l3), ('BE', 'FE', 'Analytical'))
plt.xlabel('time')

plt.show()
```

Plot