

# BME 6310 – Homework 3

## Problem 1

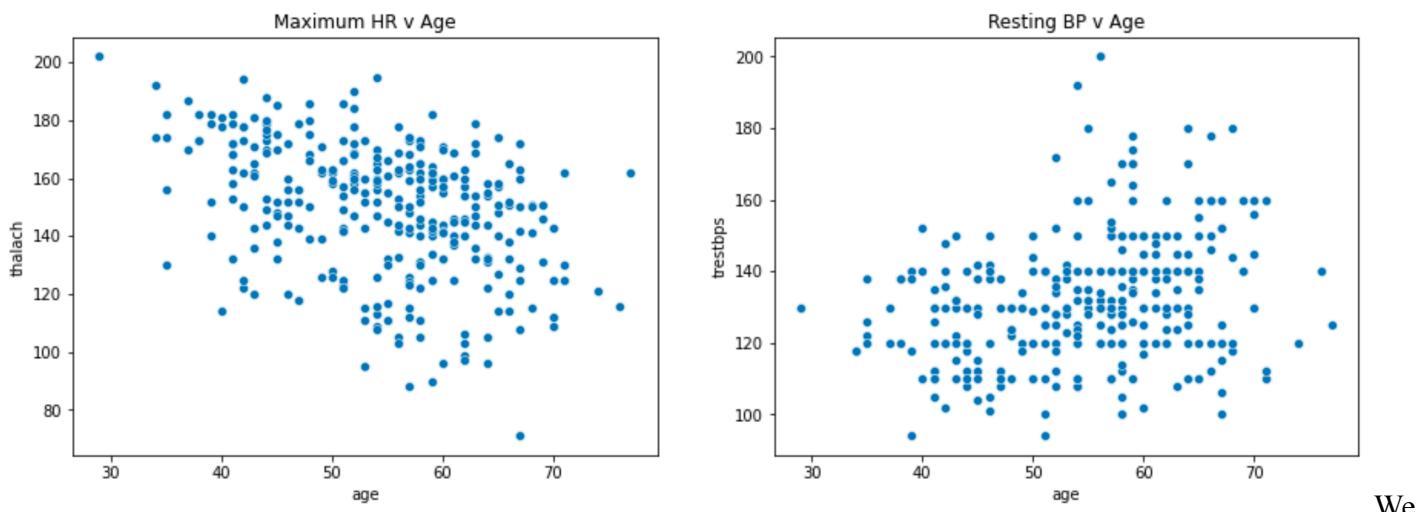
Multiple linear regression is a great tool to use when you have a data set that contains a large set of **independent variables** and one **dependent variable**. It works well to capture mathematical relationships that exist between predictors and outcomes that might not necessarily exist in singular regression analysis alone. Two pitfalls that can be made, however, are non-linear relationships between predictors and outcome variables and colinearities that exist between predictor variables – that is predictor variables that depend on each other. Plots should be made between predictor's to examine possible colinearities. As well, residual plots in the predictor space can indicate the existence of nonlinear relationships in the data. Many robust statistical techniques exist to formalize these processes.

## Problem 2

I was able to find a relatively small sample ( $n=304$ ) of cardiac vital datapoints for a patient population. For each patient the following metrics were collected:

- Age
- Sex
- Resting blood pressure (mmHg)
- Cholesterol level (mg/dL)
- Maximum heart rate (bps)
- ECH Peak value

I plotted both the **resting blood pressure** and the **cholesterol level** against the patient age:



We can see that there seems to be at least *some* linear correlation between both variables in relation to a patients age. Using the `sklearn` python package we can fit a multilinear regression to the data using a patients **Resting BP**, **age**, and **cholesterol level** as *predictor* variables, with an attempt to predict the patients **maximum heart rate**. Our regression equation will take on the form:

$$maxHR = c_0 + c_1X_1 + c_2X_2 + c_3X_3$$

Where  $X_1$ ,  $X_2$ , and  $X_3$  are the patients Resting BP in mmHg, Age in years, and Cholesterol level in mg/dL respectively.

The following code was used to generate a multi-linear regression:

```
from sklearn import linear_model

# init the regression class
regr = linear_model.LinearRegression()

X = heart_data
Y = heart_data['thalach']

# fit data to multilinear regression
regr.fit(X, Y)
```

We can extract out the intercept ( $c_0$ ) and coefficients to our model like so:

```
# print out our coefficients
print(f"Regression intercept: {regr.intercept_}")
print(f"Regression coeffs: {regr.coef_}")
print(f"Equation: MaxHR = {regr.intercept_} + X1*{round(regr.coef_[0],2)} + \
      X2*{round(regr.coef_[1],2)}, where X1 = Resting BP, X2 = Age, and X3 = Cholesterol level")
```

Which produces the following output:

```
Regression intercept: 189.68457189894912
Regression coeffs: [ 0.08485324 -1.09099419  0.03291974]
Equation: MaxHR = 189.68 + X1*0.08 + X2*-1.09 + X3*30.03,
           where X1 = Resting BP, X2 = Age, and X3 = Cholesterol level
```

Thus, our multi-linear model becomes the equation:

$$maxHR = 189.68 + 0.08X_1 - 1.09X_2 + 30.03X_3$$

The value, now, of our equation being that we can take a patients vitals (namely Resting BP, Age, and Cholesterol) and predict their Maximum Heart Rate value. This is done programmatically like so:

```

# using the first data point in our set (maxHR = 150)
maxHR = regr.predict([[  

    145, # trestbps  

    63, # age  

    233 # chol  

]])

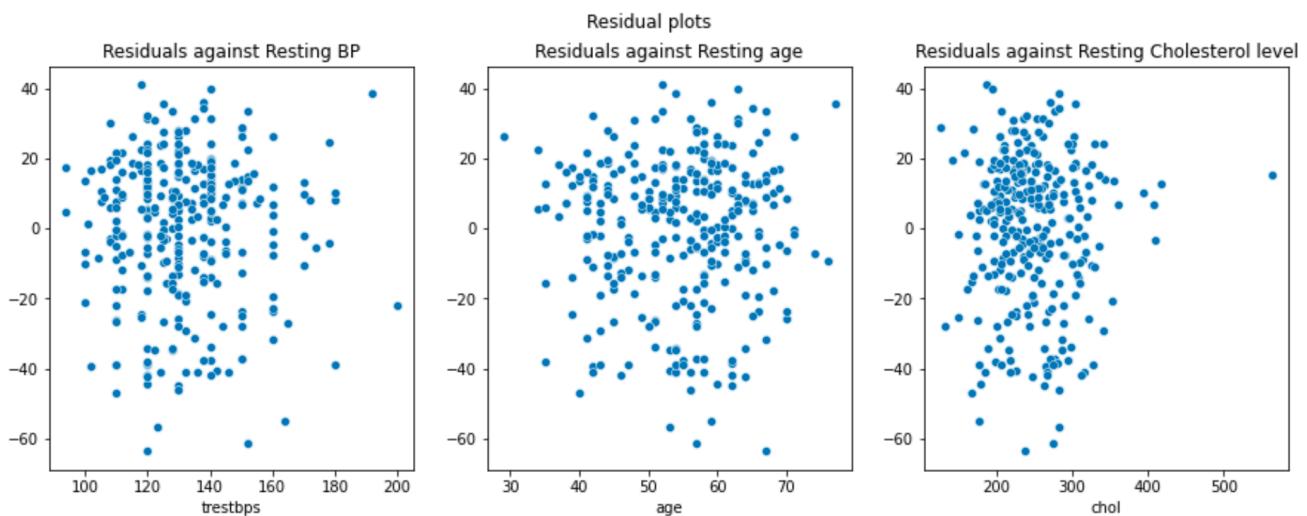
print(f"Predicted max heart rate: {round(maxHR[0])} bps")

```

Which produces:

Predicted max heart rate: 141 bps

This is not too far off from the actual value of 150bps demonstrating that our model **could potentially** be valuable and accurate. A lot more testing would be required, however. Here are some residual plots:



**Figure 1:** Residual plots for our generated multi-linear regression model

We can see that the model is definitely not perfect.. and we occasionally get pretty far from our true max heart rate value.

## Problem 3

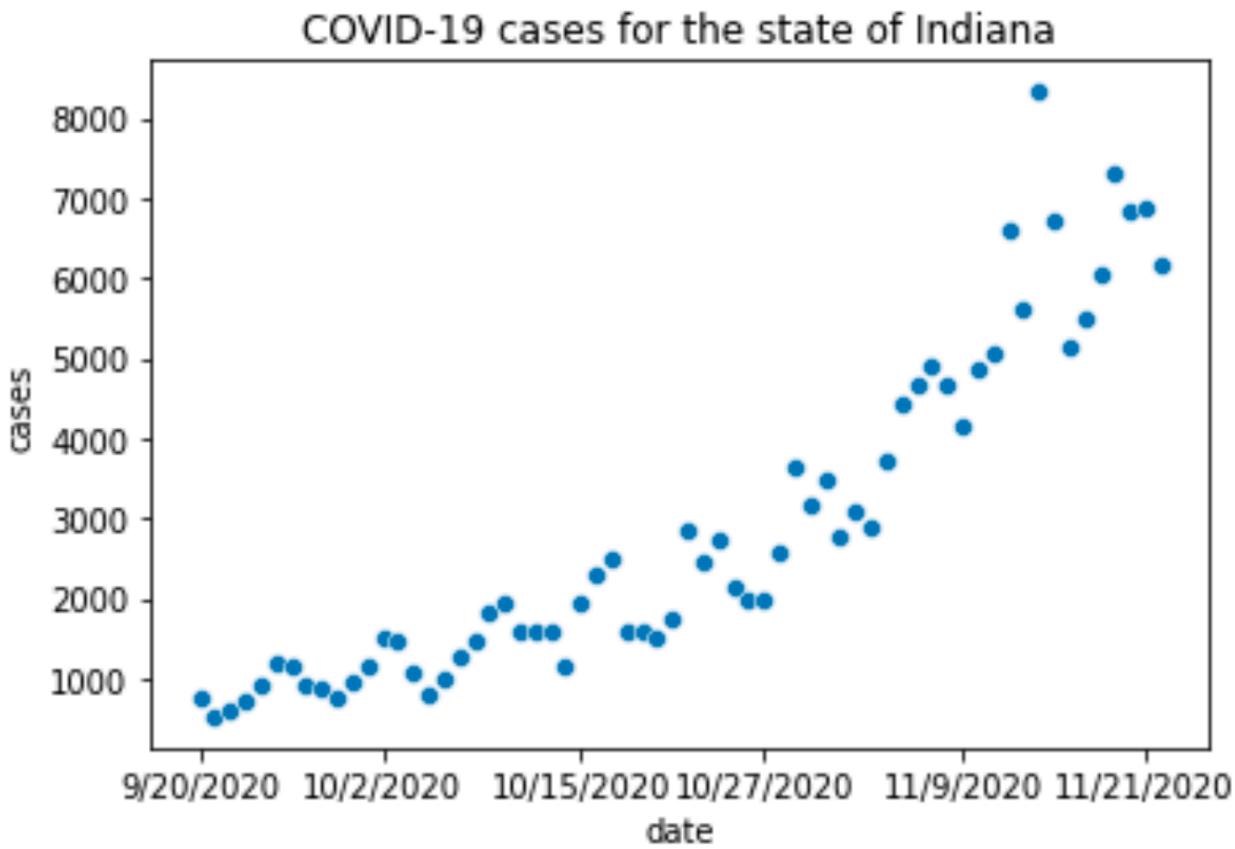
### The data

A classic (and now extremely relevant) example of a naturally occurring exponential phenomenon is the spread of a virus through a population. A timely example is the spread of SARS-CoV-2 throughout the world. I happen to have some data on COVID-19 cases in the state of Indiana and I'll use that to fit an exponential function. For this dataset, I extracted out a small timeline between **September 20th, 2020** and

**November 22nd, 2020** in order for the exponential fitting to properly work. (i.e. neglecting any pandemic “waves”). Here is the data plotted:

```
# read in the data
covid_data = pd.read_csv('data/covid.csv')

# plot the data
sns.scatterplot(
    data=covid_data,
    x="date",
    y="cases"
).set(title="COVID-19 cases for the state of Indiana")
```



**Figure 2:** New COVID-19 cases between September, 20th, 2020 and November 22nd, 2020 in the state of Indiana

### Fitting to an exponential

An exponential function takes on the following form:

$$f(x) = ab^x$$

A common clever technique used to fit data to this type of function is to first **linearize** the function like so:

$$\log_{10}(f(x)) = \log_{10}(ab^x) = \log_{10} a + x \log_{10} b$$

Which, taking  $f(x) = y$ , can be rearrange into:

$$\log_{10} y = (\log_{10} b)x + \log_{10} a$$

This looks identical to the equation of a line:

$$y = mx + b$$

And thus, if we plot  $\log_{10} y$  against  $x$  and fit the data using linear regression, we can back calculate the parameters for the original exponential function. I.e.:

$$m = (\log_{10} b) \rightarrow b = 10^m$$

$$b = \log_{10} a \rightarrow a = 10^b$$

In python, lets linearize our data and plot the result. Note that, in my model, the value of  $x$  is the days since September 20th, 2020. Thus, some calculations are done to reconcile this as our data is given in date format:

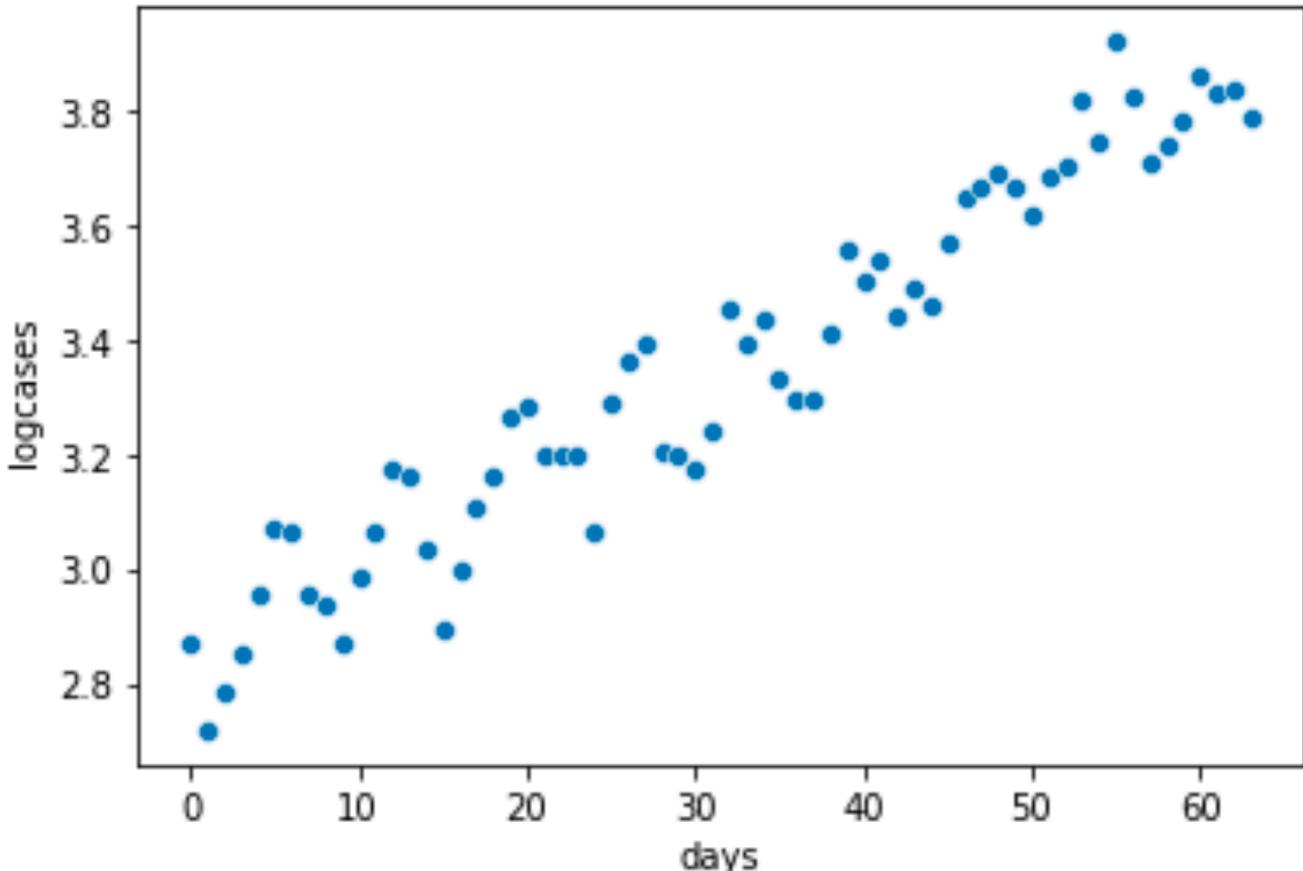
```
from datetime import datetime
date_format = "%m/%d/%Y"

day_zero = datetime.strptime('09/20/2020', date_format)

# linearize data
covid_data['logcases'] = np.log10(covid_data['cases'])

# init a new column
covid_data['days'] = np.zeros(len(covid_data))
covid_data['days'] = [(datetime.strptime(row['date'], date_format) -
                     day_zero).days for _, row in covid_data.iterrows()]

# plot to show linear relationship
sns.scatterplot(
    data=covid_data,
    x="days",
    y="logcases"
)
```



We

can see now that our data looks **very** linear! Lets use `scipy` again to fit a single variable linear regression to the data:

```
# init the regression class
covid_lin_regr = linear_model.LinearRegression()

# extract data to fit
X = covid_data
y = covid_data['logcases']

# fit
covid_lin_regr.fit(X, y)

# extract params
print(f"Intercept: {covid_lin_regr.intercept_}")
print(f"Slope: {covid_lin_regr.coef_[0]}")
```

Intercept: 2.8330092724282654

Slope: 0.016452348797853567

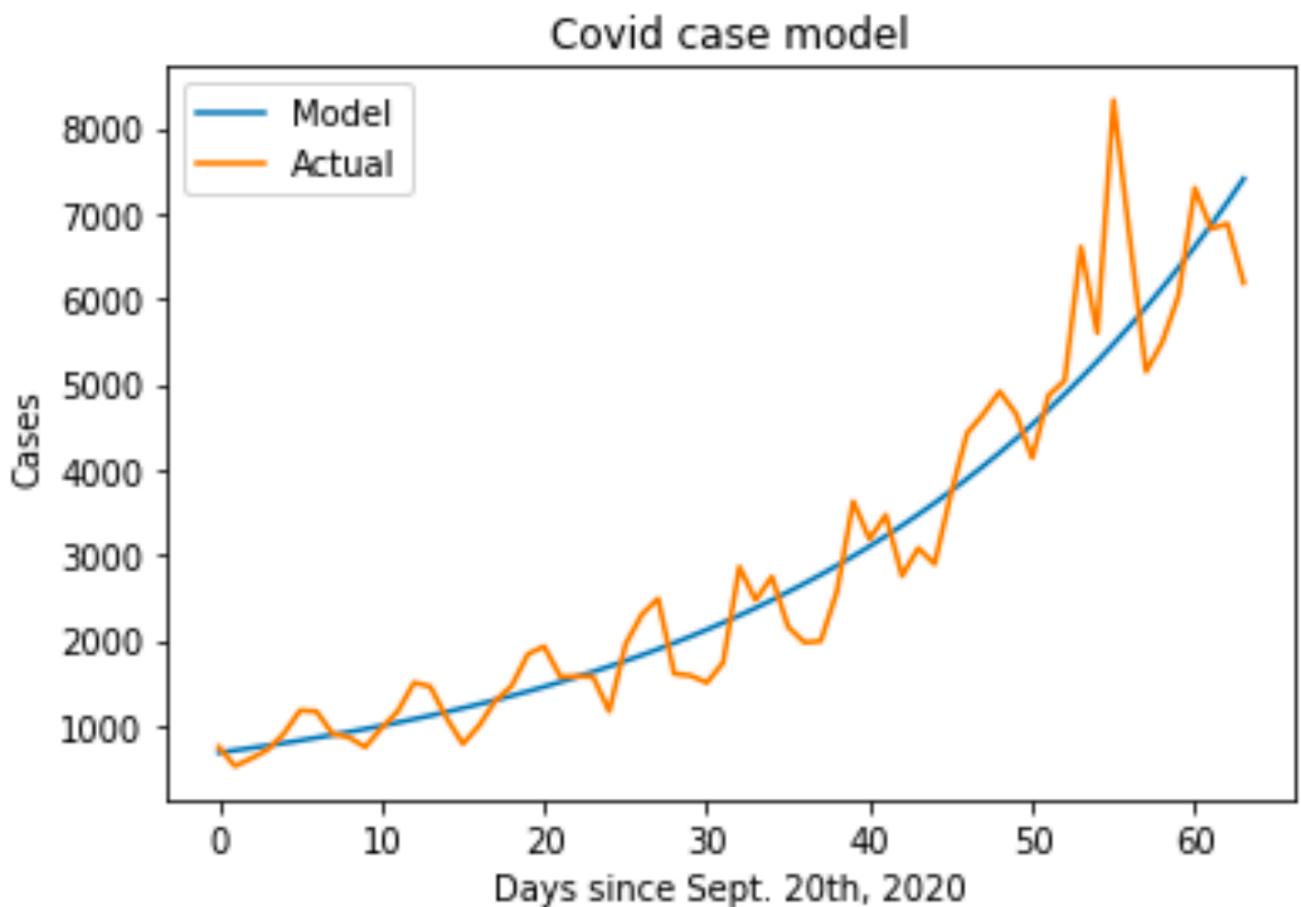
Lets use these values to back calculate the parameters  $a$  and  $b$  for our exponential function. As well, lets plot our fitted function against the original, unlinearized, data:

```

# calc original params
a = 10**covid_lin_regr.intercept_
b = 10**covid_lin_regr.coef_[0]

# create model
def covid_model(x: int, a:float, b:float) -> float:
    return a*b**x

```



**Figure 3:** COVID case data fit to an exponential

We can see from the above plot that our model is a very nice fit to our data! This model, obviously, breaks down once we reach a peak or new wave of the pandemic.

## Problem 4

To run the decimation and interpolation process, I will be using an image of my cat, Karen:



**Figure 4:** Reference image for decimation – Karen the Cat

First we must load in the image, and create a new matrix to store our decimated “sparse” image”

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate

DIM = 4000
DECIMATION_FACTOR = ...

# load in image and convert to grayscale and crop
cat = plt.imread('data/karen.png')
cat = np.dot(cat[...,:3], [0.299, 0.587, 0.114])
cat = cat[:DIM,:DIM] # crop image

# create the sparse image
cat_sparse = np.zeros((int(DIM/DECIMATION_FACTOR), int(DIM/DECIMATION_FACTOR)))
```

To make things simpler, I am defining the image dimensions and then also assigning the decimation factor to a variable later. This value will be varied to assess how the decimation and interpolation changes as we alter this value. I will use decimation factors of **10, 40, and 80**. Here is the code to decimate the image:

```
# decimate the image
for i in range(int(DIM/DECIMATION_FACTOR)):
    for j in range(int(DIM/DECIMATION_FACTOR)):
        cat_sparse[i,j] = cat[i*DECIMATION_FACTOR][j*DECIMATION_FACTOR]
```

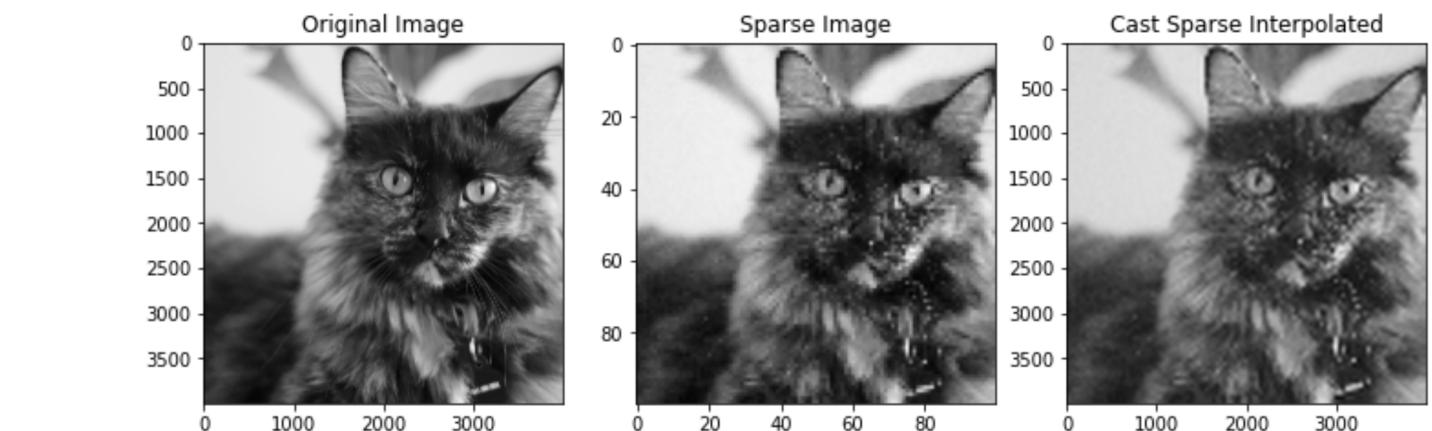
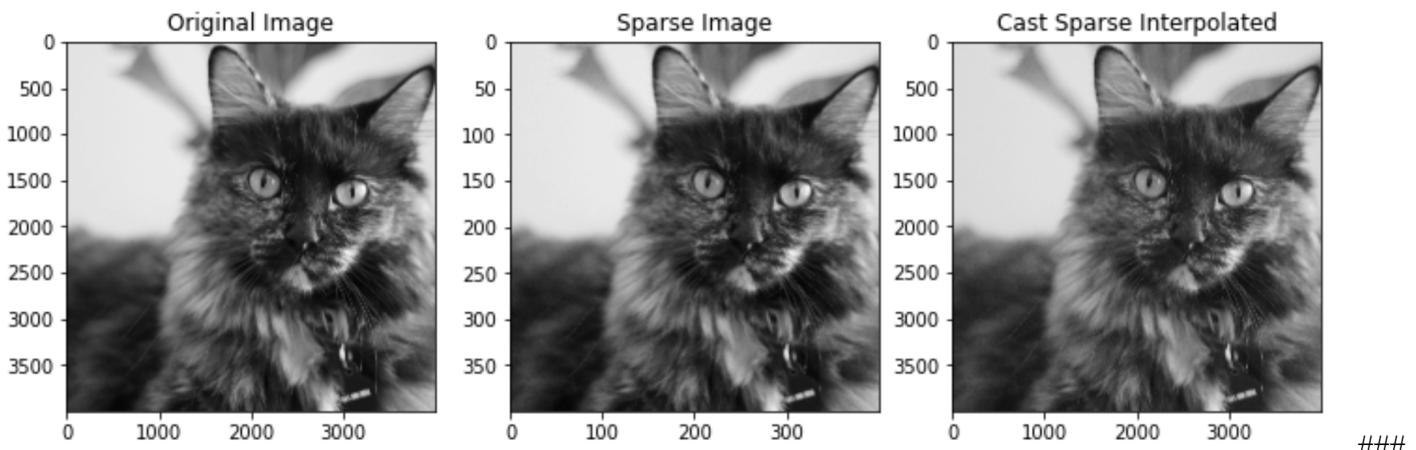
Once the image is decimated, we can then interpolate backwards using the `scipy` package `interpolate`:

```
# Interpolate
Xi, Yi = np.arange(0, DIM, 1), np.arange(0, DIM, 1)
X, Y = np.arange(0, DIM, DECIMATION_FACTOR), np.arange(0, DIM, DECIMATION_FACTOR)

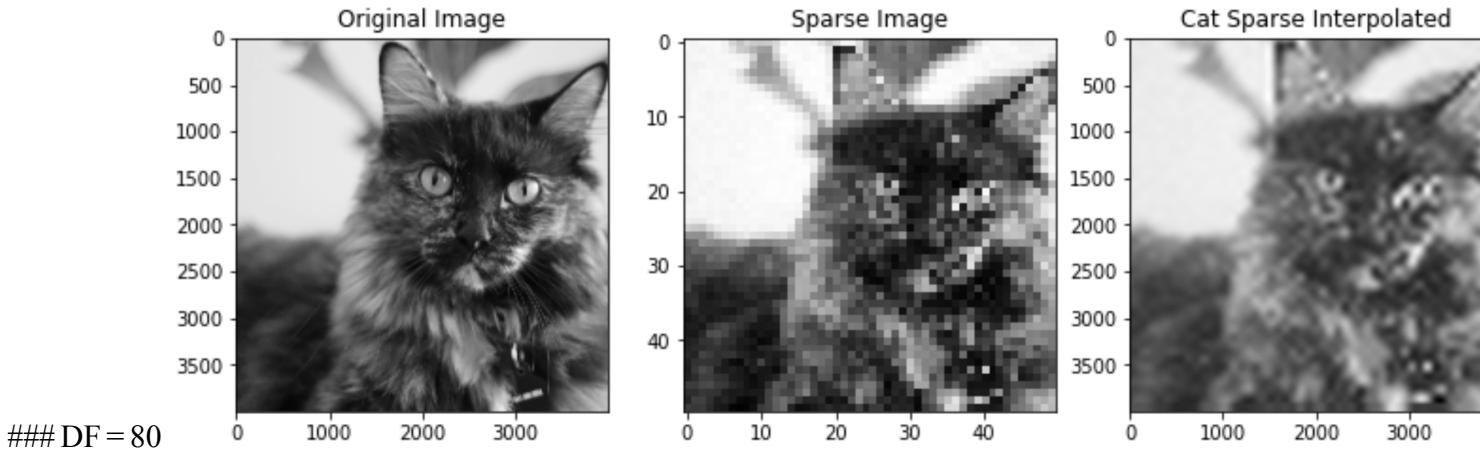
f = interpolate.interp2d(X, Y, cat_sparse, kind='cubic')
cat_sparse_i = f(Xi, Yi)
```

Now we can plot the results at three different decimation levels or decimation factors (**DF**):

**DF = 10**



DF=40



As expected, as the decimation factor increases, the picture of Karen becomes more and more pixelated. As well, the final interpolated image is less discernible and more blurry. A decimation factor of 80 showed the biggest relative decrease in quality and you can even see it picking out pixels from area of fur that shouldn't be where they are.

## Problem 5

To calculate the steepest descent, lets first calculate the gradient of the function  $f(x)$ :

$$f(x) = x_1^2 + 2x_2^2 - x_1 - 6x_2$$

$$\nabla f(x) = \frac{\partial f}{\partial x_1} i + \frac{\partial f}{\partial x_2} j = \begin{bmatrix} 2x_1 - 1 \\ 4x_2 - 6 \end{bmatrix}$$

$$x_0 = (0, 0)$$

### Step 1

First, evaluate the gradient at  $x_0$ :

$$\nabla f(x)|_{x=(0,0)} = \begin{bmatrix} -1 \\ -6 \end{bmatrix}$$

Then we define our new  $x_0$  values as the following:

$$x_{1_{new}} = x_0 + \frac{\partial f}{\partial x_1} h = 0 - 1h = -h$$

$$x_{2_{new}} = x_0 + \frac{\partial f}{\partial x_2} h = 0 - 6h = -6h$$

We can plug this into the original equation  $f(x)$  to optimize the value of  $h$ , the amount we take a step:

$$g(h) = f(x_1(h), x_2(h)) = (-h)^2 + 2(-6h)^2 - (-h) - 6(-6h)$$

$$g(h) = h^2 + 72h^2 + h + 36h = 73h^2 + 37h$$

We can then take the derivative of  $g(h)$  to optimize the step value:

$$\frac{dg}{dh} = 146h + 37 = 0$$

Thus,

$$h = \frac{-37}{146} \approx -0.25$$

We can back calculate using our value of  $h$  to get the new values for  $x_1$  and  $x_2$ :

$$x_{1_{new}} = x_0 + \frac{\partial f}{\partial x_1}h = 0 - 1h = -h = 0.25$$

$$x_{2_{new}} = x_0 + \frac{\partial f}{\partial x_2}h = 0 - 6h = -6(-0.25) = 1.5$$

## Step 2

With our new values, we can repeat this entire process:

$$\nabla f(x)|_{x=(0.25, 1.5)} = \begin{bmatrix} 2(0.25) - 1 \\ 4(1.5) - 6 \end{bmatrix} = \begin{bmatrix} -0.5 \\ 0 \end{bmatrix}$$

And then, we can define our new values using the same process as step 1:

$$x_{1_{new}} = x_0 + \frac{\partial f}{\partial x_1}h = 0.25 - 0.5h$$

$$x_{2_{new}} = x_0 + \frac{\partial f}{\partial x_2}h = 1.5 + 0h = 1.5$$

And then create an update  $g(h)$ :

$$g(h) = f(x_1(h), x_2(h)) = (0.25 - 0.5h)^2 + 2(1.5)^2 - (0.25 - 0.5h) - 6(1.5)$$

$$g(h) = 0.25h^2 + 0.25h + C$$

I am neglecting to calculate the constant values as they get dropped anyways in the derivative calculation.  
Lets now optimize  $h$ :

$$\frac{dg}{dh} = 0.5h + 0.25$$

$$0 = 0.5h + 0.25$$

$$h = -0.5$$

And, finally, we can plug this value of  $h$  back into our original equation's's for the new  $x$  values:

$$x_{1_{new}} = x_0 + \frac{\partial f}{\partial x_1} h = 0.25 - 0.5(-0.5) = 0.5$$

$$x_{2_{new}} = x_0 + \frac{\partial f}{\partial x_2} h = 1.5 + 0h = 1.5$$

Plugging into our original function,

$$f(x) = (0.5)^2 + 2(1.5)^2 - 0.5 - 6(1.5) = -4.75$$

We can verify this result using an optimization package in python:

```
from scipy import optimize

# define function to optimize
def f(x):
    return (x[0]**2) + (2*(x[1]**2)) - x[0] - (6*x[1])

optimize.fmin(f, [0,0])
```

Which produces the following output:

```
Optimization terminated successfully.
    Current function value: -4.750000
    Iterations: 63
    Function evaluations: 120
array([0.50001601, 1.50002605])
```

Which we can see is almost precisely the values we achieved with steepest descent.

## Problem 6

I am going to use Wolfram Alpha's constrained optimization web app to optimize the following function:

$$f(x) = 2x_1 + 3x_2 + x_3$$

Subject to the following constraints,

$$x_1 + x_2 + x_3 \leq 4.8$$

$$10x_1 + x_3 \leq 9.9$$

$$x_2 - x_3 \leq 0.2$$

Wolfram alpha found the global maximum to be **9.8** at **(x,y,z) = (0, 2.5, 2.3)**. We can see that this satisfies the constraints:

$$0 + 2.5 + 2.3 = 4.8 \leq 4.8$$

$$10(0) + 2.3 = 2.3 \leq 9.9$$

$$2.5 - 2.3 = 0.2 \leq 0.2$$

We can also try to perturb the system variables within the constraints to demonstrate it moves us away from the maximum:

### Perturb $x_1$

Let  $x_1 = -1$

We are still within the constraints:

$$-1 + 2.5 + 2.3 = 3.8 \leq 4.8$$

$$10(-1) + 2.3 = -7.7 \leq 9.9$$

$$2.5 - 2.3 = 0.2 \leq 0.2$$

The new function value:

$$f(x) = 2(-1) + 3(2.5) + 2.3 = 7.8$$

which is lower than the previous maximum.

### Perturb $x_2$

Let  $x_2 = -1$ :

We are still within the constraints:

$$0 + -1 + 2.3 = 1.3 \leq 4.8$$

$$10(0) + 2.3 = 2.3 \leq 9.9$$

$$0 - 2.3 = -3.3 \leq 0.2$$

The new function value:

$$f(x) = 2(0) + 3(-1) + 2.3 = 0.7$$

which is lower than the previous maximum.

## Problem 7

To solve this constrained optimization problem in Python, I am using the `scipy.optimize.linprog` function. The following code was used to run the optimization:

```
# function coeffs
f = -1*np.array()

# constraint coeffs
constraint_coeffs = np.array([
    [1, 1, 1],
    [10, 0, 1],
    [0, 1, -1]
])

# upper bounds
# ... <= bound
upper_bounds = np.array([
    4.8,
    9.9,
    0.2
])

res = linprog(f, A_ub=constraint_coeffs, b_ub=upper_bounds)
```

The following output is achieved:

```
The maximum value was found to be 9.799999999985156
Our function values were found to be: [0.66754125 2.16622938 1.96622938\]
```

The maximum is almost identical to the original value we saw using **Wolfram Alpha**.

We can now iterate over the optimization function code to survey how our maximum value changes as we vary the constraints. The following loop was made to vary the first constraint from **1 to 10**:

```
for i in range(1,11):
    upper_bounds[0] = i
    res = linprog(f, A_ub=constraint_coeffs, b_ub=upper_bounds)
    print(f"First constraint <= {i}")
    print(f"The maximum value was found to be {abs(res.fun)}")
    print(f"Our function values were found to be: {-1*res.x}")
    print("\n---*20)
```

Which produces the (very long) output:

```
First constraint <= 1
The maximum value was found to be 2.200000001701729
Our function values were found to be: [-0.61566885 -0.29216557 -0.09216557]
-----
First constraint <= 2
The maximum value was found to be 4.20000000018904
Our function values were found to be: [-0.83611454 -0.68194273 -0.48194273]
-----
First constraint <= 3
The maximum value was found to be 6.20000000007946
Our function values were found to be: [-0.77811795 -1.21094102 -1.01094102]
-----
First constraint <= 4
The maximum value was found to be 8.20000000006241
Our function values were found to be: [-0.7166718 -1.7416641 -1.5416641]
-----
First constraint <= 5
The maximum value was found to be 10.19999999979102
Our function values were found to be: [-0.65528067 -2.27235967 -2.07235967]
-----
```

```

First constraint <= 6
The maximum value was found to be 12.199999999950952
Our function values were found to be: [-0.59314323 -2.80342838 -2.60342838]
=====

```

The maximum value steadily increases as we run this constraint.

## Problem 8

We can use the optimization techniques to maximize profit given cost and demand functions for any given price that we set for a product. We determine the **total cost** ( $TC$ ) of our product to be defined by the following function:

$$TC(y) = 200y + 15y^2$$

Where  $y$  is the price of our product. Then we can determine the demand curve for our product given any price  $y$ :

$$D = 1200 - 10y^3$$

The total revenue we make then, as a function of price is going to be:

$$TR(y) = (\text{Demand}) \times (\text{Price}) = Dy = (1200 - 10y^3)y = 1200y - 10y^4$$

We can then arrange a total **profit** function to be:

$$TP(y) = TR(y) - TC(y) = (1200y - 10y^4) - (200y + 15y^2) = 1000y - 15y^3 - 10y^4$$

We can take the derivative of the revenue function to maximize it:

$$\frac{dP}{dy} = 1000 - 45y^2 - 40y^3 = 0$$

Plugging this into my pre-written newton-raphson algorithm:

```

from nr import *

def marg_profit(x: float) -> float:
    return 1000 - 45*x**2 - 40*x**3

```

```

res = newton_raphson(marg_profit, 4)
print(f"The monopolist should price his magic beans at {res} schmeckles per bean")

```

We receive the following output:

We should price our product at 2.59 schmeckles per unit

The **optimal price** that maximizes our profit is **2.59** USD/EUR/Schmeckles

## Problem 9

Below is my Newton-Raphson algorithm implemented in Python:

```

def newton_raphson(func, x: float, tolerance: float = 0.001) -> float:
    """
    Use newton raphson method to estimate the root of an equation
    """
    # estimate zero / calculate error
    h = func(x) / _derivative(func, x)

    # while error greater than tolerance
    while abs(h) > tolerance:
        h = func(x) / _derivative(func, x)
        x = x - h

    return x

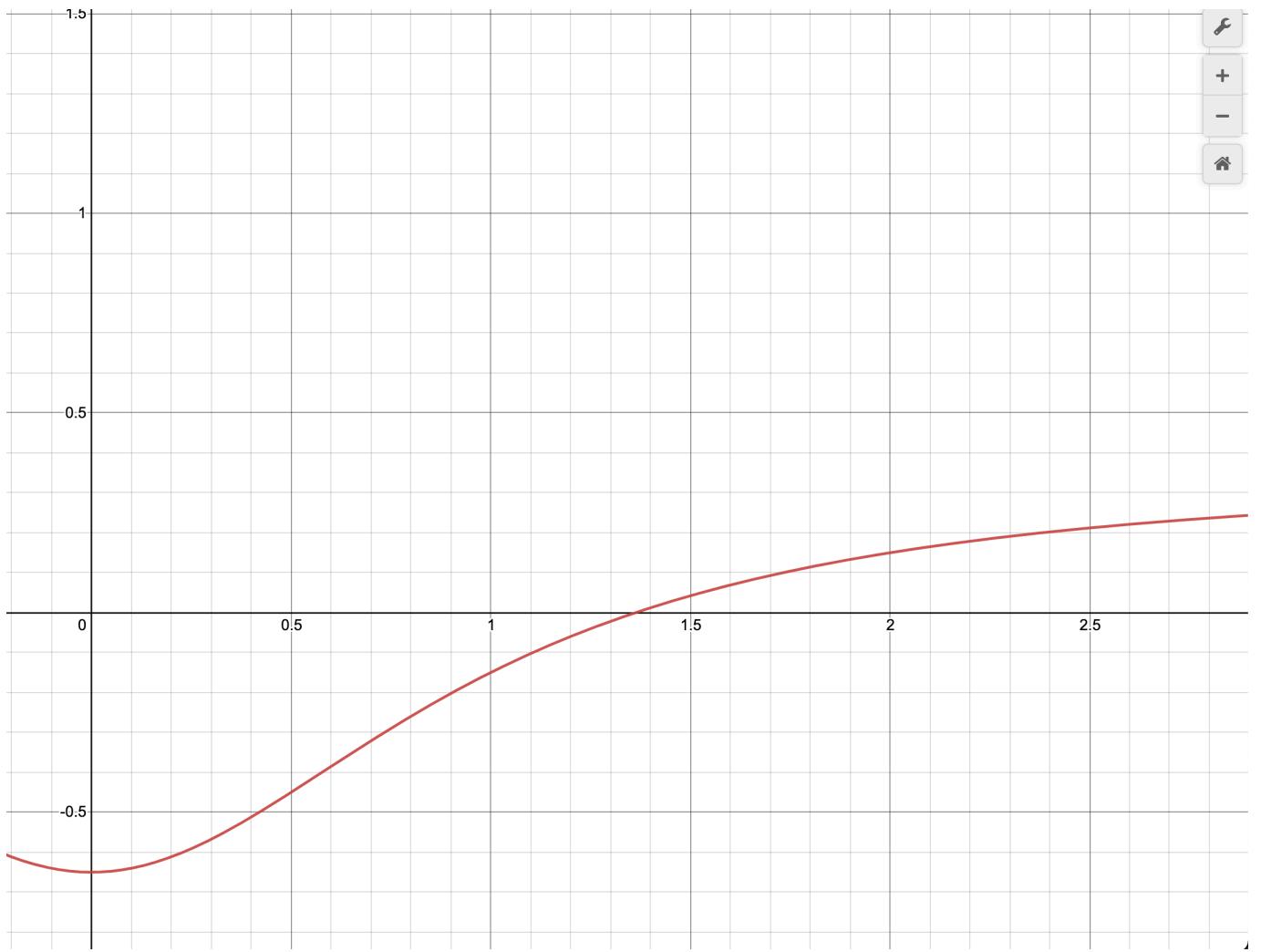
```

I decided to test my algorithm on the **Hill Equation**:

$$Y = \frac{AX^n}{K^n + X^n}$$

I am going to take  $A = 1$  and  $K = 1$  and  $n = 2$  and shift the function down 0.65 units in order to set up a nice function for root calculation. Here is the function:

$$Y = \frac{X^2}{1 + X^2} - 0.65$$



**Figure 5:** Made-up hill function plotted using desmos.

Here is a python implementation of my made-up hill equation:

```
def hill(x: float, n: int = 2) -> float:
    """
    Evaluate the hill equation at certain x
    """
    return ((x**n)/(1+x**n) - 0.65)
```

Plugging this into my newton-raphson function, we get the following output:

```
from nr import *
root = newton_raphson(hill, 1.5)
```

The root was found to be: 1.36

By eye, we can see that this is pretty close to the actual root value!

## Problem 10

Below is my **Golden Section Search** algorithm implemented in Python:

```
def gss(func, x_low: float, x_up: float, tolerance: float = 0.0001, loc='max') -> float:
    """
    Implement the golden section search algorithm
    """

    # calculate the distance
    # multiplied by the golden
    # ratio
    dist = (x_up - x_low)*GOLDEN_RATIO

    # remove this calculated value
    # from the upper value
    # and add to the lower value
    x_2 = x_up - dist
    x_1 = x_low + dist

    while abs(x_up - x_low) > tolerance:
        if func(x_1) > func(x_2):
            x_low = x_2
        else:
            x_up = x_1

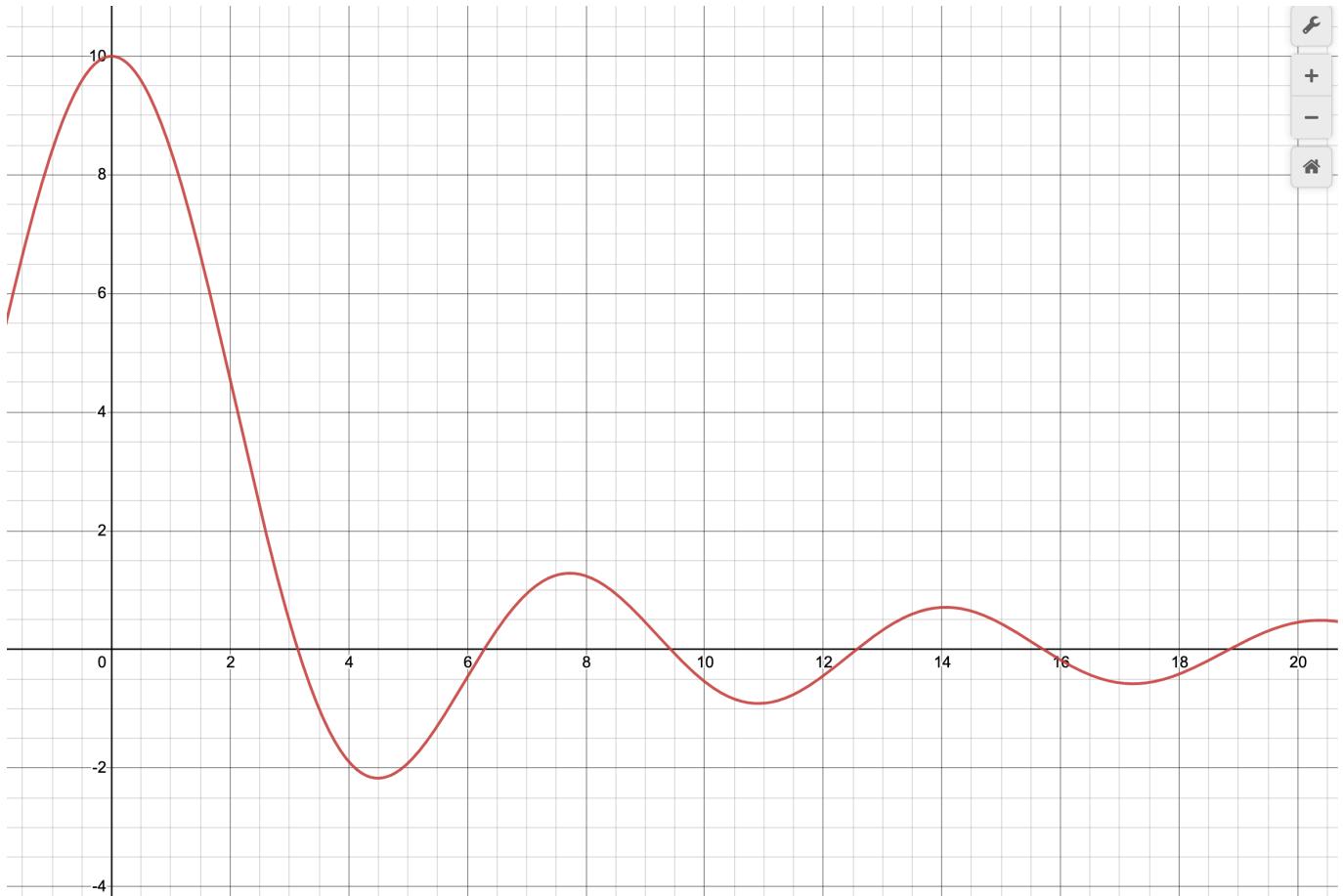
        # calculate the distance
        # multiplied by the golden
        # ratio again
        dist = (x_up - x_low)*GOLDEN_RATIO

        # recalculate the bounds
        x_2 = x_up - dist
        x_1 = x_low + dist

    # return average between
    # the two bounds
    return (x_up + x_low) / 2
```

To test this algorithm, I am using the following made-up equation:

$$f(x) = \frac{10}{x} \sin(x)$$



**Figure 6:** Our made up function plotted on desmos.

In order to solve for the maximum of that first hump, I am taking an initial x value of 7:

```
from gss import *\n\n# implement gss algorithm\nmaximum = gss(f, 7, 9)\nprint(f"Maximum found: {round(maximum, 2)}")
```

Which produces the following output:

```
Maximum found: 7.73
```

By eye, we can see that this is pretty close to the x-value that corresponds to our maximum!