Homework 5: Modeling, Estimation and Gradient Descent

Due Date: Saturday 11/23, 11:59 PM

We will explore modeling through estimation and prediction. Along the way, we will get experience with an iterative method for guess-and-check called gradient descent. These approaches are helpful throughout data science particularly the field of machine learning. By completing Homework 5, you should take away...

- Practice reasoning about a model and building intuition for loss functions.
- Determining the gradient of a loss function with respect to model parameters and using the calculations for gradient descent.

We will apply these skills in Homework 6 to a real-world dataset.

Submission Instructions

For this assignment, you will submit a copy to Gradescope. Follow these steps

1. Download as HTML (`File->Download As->HTML(.html)`).
2. Open the HTML in the browser. Print to .pdf
3. Upload to Gradescope. Tag your answers.

Note that

- Please map your answers to our questions. Otherwise you may lose points. Please see the rubric below.
- You should break long lines of code into multiple lines. Otherwise your code will extend out of view from the cell. Consider using `\` followed by a new line.
- For each textual response, please include relevant code that informed your response. For each plotting question, please include the code used to generate the plot.
- You should not display large output cells such as all rows of a table. Instead convert the input cell from Code to Markdown back to Code to remove the output cell.

Moreover you will submit a copy on Jupyter Hub under Assignments Tab. You cannot access the extension in JupyterLab. So if the URL ends with lab, then please change it to tree

```
https://pds-f19.jupyter.hpc.nyu.edu/user/[Your NetID]/tree
```

Consult the instructional video

`https://nbgrader.readthedocs.io/en/stable/_images/student_assignment.gif`

for steps to...

1. fetch
2. modify
3. optionally validate
4. submit your project

Failure to follow these guidelines for submission could mean the deduction of two points. See the rubric below.

Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the homework, we ask that you **write your solutions individually**. If you do discuss the assignments with others please **include their names** at the top of your solution.

Rubric

Question	Points
Gradescope	2
Question 1a	1
Question 1b	1
Question 1c	1
Question 1d	1
Question 1e	1
Question 2a	2
Question 2b	1
Question 2c	1
Question 2d	1
Question 2e	1
Question 2f	1
Question 3a	1
Question 3b	3

Question	Points
Question 3c	2
Question 4a	3
Question 4b	1
Question 4c	1
Question 4d	1
Question 4e	1
Question 5a	2
Question 5b	1
Question 5c	1
Question 5d	0
Total	37

Getting Started

In [6]:
```
from IPython.display import display, Markdown

# Import standard packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import csv
import re
import seaborn as sns

# Set some parameters
plt.rcParams['figure.figsize'] = (12, 9)
plt.rcParams['font.size'] = 16
np.set_printoptions(4)
```

In [7]:
```
# Import a specialized plotting package useful for 3d
import plotly
import plotly.graph_objs as go
plotly.offline.init_notebook_mode(connected=True)
from hw5_utils import plot_3d

# Note that you may need to install plotly to import these packages
# Either execucte the next cell or complete the assignment on JupyterHub
```

In [8]:
```
# !pip install plotly
# !conda install -c conda-forge jupyterlab-plotly-extension
```

Load Data

Load the data.csv file into a pandas dataframe.
Note that we are reading the data directly from the URL address.

In [9]:
```
# Run this cell to load our sample data
data = pd.read_csv("data.csv", index_col=0)
data.head()
```

Out[9]:

	x	y
0	-5.000000	-8.262369
1	-4.966555	-7.692411
2	-4.933110	-7.358698
3	-4.899666	-8.067488
4	-4.866221	-7.834256

1: A Simple Model

Let's start by examining our data and creating a simple model that can represent this data.

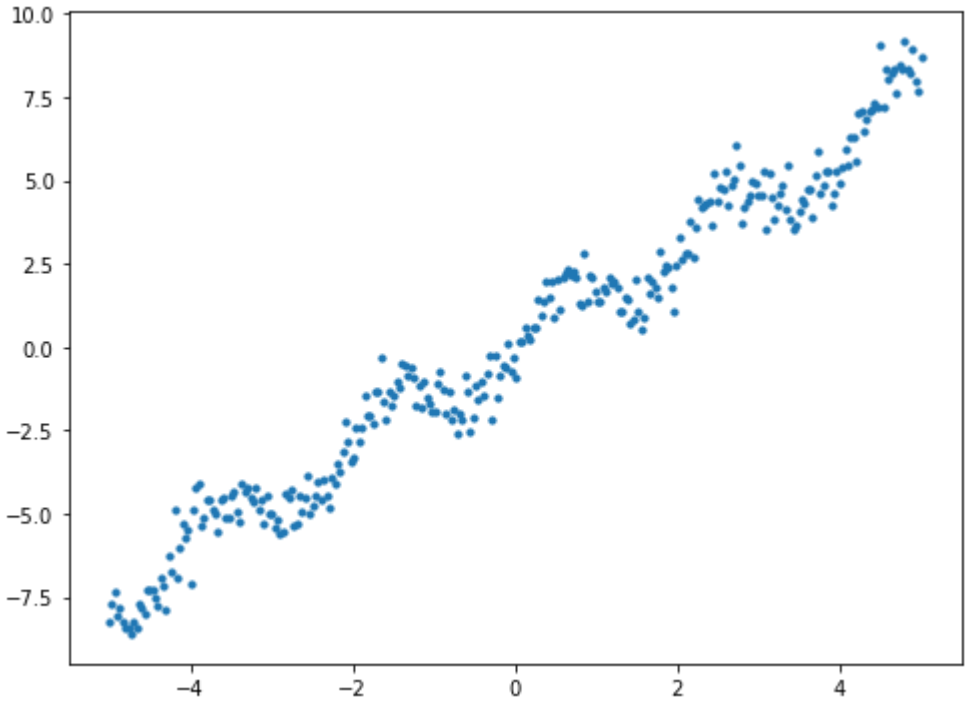
Question 1

Question 1a

First, let's visualize the data in a scatter plot. After implementing the `scatter` function below, you should see something like this:

 scatter

In [11]:
```
def scatter(x, y):
    """
    Generate a scatter plot using x and y

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    """
    plt.figure(figsize=(8, 6))
    plt.scatter(x, y, s=10)
    # YOUR CODE HERE
    raise NotImplementedError()

x = data['x']
y = data['y']
scatter(x,y)
```



Question 1b

Describe any significant observations about the distribution of the data. How can you describe the relationship between x and y?

In [14]:
```
# YOUR CODE HERE
'''x and y are said to be vectors of data points. Based on the scatter plot, there seems to be a strong posit
ive linear
correlation between x and y, where as x increases, y increases. A significant observation about the distribut
ion of the data
is that for x and y, the data seems to be concentrated heavily towards the negative values of x and y at firs
t.
As said before, the data is positive linear, and there is no outliers, as well as no gaps.
The data seems to be tighly compact, meaning there is no deviation, or loss from other adjacent data points.
In essence, it's very possible to calculate the r as .90, and graph a linear regression line, given the data.
Another important factor is that the x axis values range from ~-2 to 6, whereas the y axis values range from
 ~-8.0 to 10,
meaning there is variability within the data ranges.

raise NotImplementedError()'''
```

Out[14]:
```
"x and y are said to be vectors of data points. Based on the scatter plot, there seems to be a strong positiv
e linear \ncorrelation between x and y, where as x increases, y increases. A significant observation about th
e distribution of the data \nis that for x and y, the data seems to be concentrated heavily towards the negat
ive values of x and y at first. \nAs said before, the data is positive linear, and there is no outliers, as w
ell as no gaps. \nThe data seems to be tighly compact, meaning there is no deviation, or loss from other adja
cent data points. \nIn essence, it's very possible to calculate the r as .90, and graph a linear regression l
ine, given the data. \nAnother important factor is that the x axis values range from ~-2 to 6, whereas the y
axis values range from ~-8.0 to 10,\nmeaning there is variability within the data ranges. \n\nraise NotImplem
entedError()"
```

Question 1c

The data looks roughly linear. However it moves up and down away from the line. For now, let's assume that the data follows some underlying linear model. We define the underlying linear model that predicts the value y using the value x as: $f_{\theta^*}(x) = \theta^* \cdot x$

Since we cannot find the value of the population parameter θ^* exactly, we will assume that our dataset approximates our population and use our dataset to estimate θ^*. We denote our estimation with θ, our fitted estimation with $\hat{\theta}$, and our model as:

$$f_\theta(x) = \theta \cdot x$$

Based on this equation, define the linear model function `linear_model` below to estimate $\mathbf{y}$ (the y-values) given $\mathbf{x}$ (the x-values) and θ. This model is similar to the model you defined in Lab 5: Modeling and Estimation.

In [15]:
```
def linear_model(x, theta):
    """
    Returns the estimate of y given x and theta

    Keyword arguments:
    x -- the vector of values x
    theta -- the scalar theta
    """
    y = x * theta
    # YOUR CODE HERE
    return y
    raise NotImplementedError()
```

In [16]:
```
assert linear_model(0, 1) == 0
assert np.sum(linear_model(np.array([3, 5]), 3)) == 24
```

In [ ]:
```

```

Question 1d

In class, we learned that the square loss, sometimes called the L^2 loss. Let's use L^2 loss to evaluate our estimate θ, which we will use later to identify an optimal θ, represented as $\hat{\theta}$. Define the L^2 loss function `l2_loss` below.

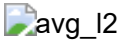
In [17]:
```
def l2_loss(y, y_hat):
    """
    Returns the average l2 loss given y and y_hat

    Keyword arguments:
    y -- the vector of true values y
    y_hat -- the vector of predicted values y_hat
    """
    if type(y) == int:
        return (y-y_hat)**2
    return (sum((y - y_hat)** 2))/len(y)
    # YOUR CODE HERE
    raise NotImplementedError()
```

In [18]:
```
assert l2_loss(2, 1) == 1
assert l2_loss(2, 0) == 4
assert l2_loss(np.array([5, 6]), np.array([1, 1])) == 20.5
```

In [ ]:
```

```

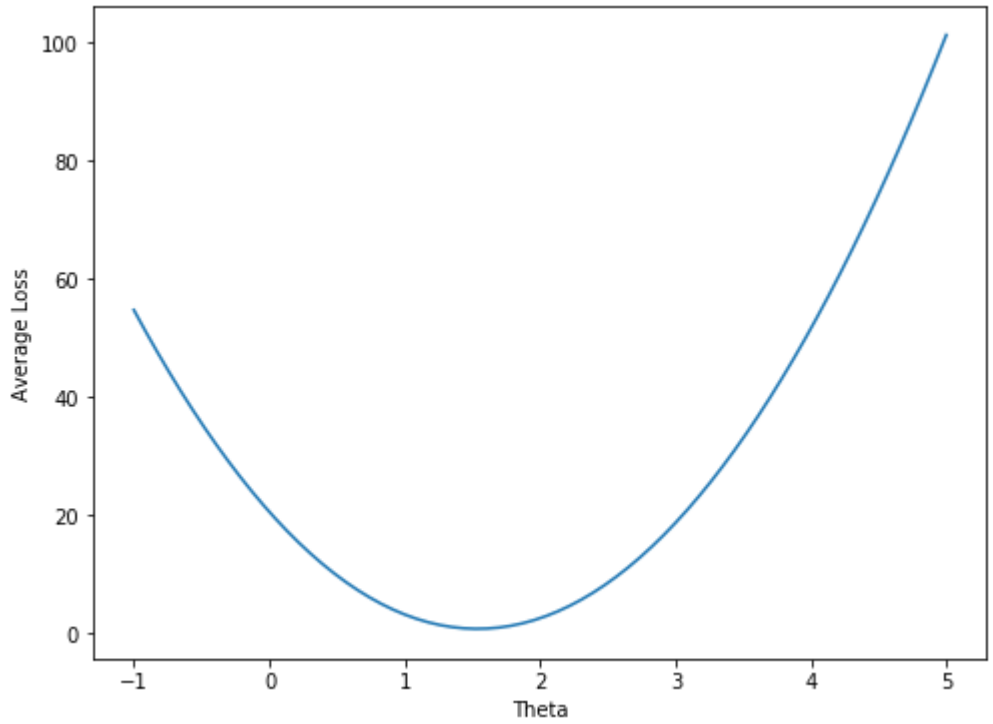
Question 1e

First, visualize the L^2 loss as a function of θ, where several different values of θ are given. Be sure to label your axes properly. You plot should look something like this:



What looks like the optimal value, $\hat{\theta}$, based on the visualization? Set `theta_star_guess` to the value of θ that appears to minimize our loss.

In [20]:
```
def visualize(x, y, thetas):
    """
    Plots the average l2 loss for given x, y as a function of theta.
    Use the functions you wrote for linear_model and l2_loss.

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    thetas -- an array containing different estimates of the scalar theta
    """
    avg_loss = np.array([l2_loss(linear_model(theta,x),y) for theta in thetas])# Calculate the loss here for
 each value of theta

    plt.figure(figsize=(8,6))

    plt.plot(thetas, avg_loss)
    plt.xlabel(r"Theta")
    plt.ylabel(r"Average Loss") # Create your plot here
#     raise NotImplementedError()

thetas = np.linspace(-1, 5, 70)
visualize(x, y, thetas)

theta_star_guess = 1.55
# YOUR CODE HERE
raise NotImplementedError()
```



In [21]:
```
assert l2_loss(3, 2) == 1
assert l2_loss(0, 10) == 100
```

In [ ]:
```

```

2: Fitting our Simple Model

Now that we have defined a simple linear model and loss function, let's begin working on fitting our model to the data.

Question 2

Let's confirm our visual findings for optimal $\hat{\theta}$.

Question 2a

First, find the analytical solution for the optimal $\hat{\theta}$ for average L^2 loss. Write up your solution in the cell below using LaTex.

Hint: notice that we now have $\mathbf{x}$ and $\mathbf{y}$ instead of x and y. This means that when writing the loss function $L(\mathbf{x}, \mathbf{y}, \theta)$, you'll need to take the average of the squared losses for each y_i, $f_\theta(x_i)$ pair.

For tips on getting started, see chapter 10 (https://www.textbook.ds100.org/ch/10/modeling_loss_functions.html) of the textbook. Note that if you check the source (https://github.com/DS-100/textbook/tree/master/content/ch/10), you can access the LaTeX code of the book chapter, which you might find handy for typing up your work. Show your work, i.e. don't just write the answer.

In [22]:
```
# YOUR CODE HERE
'''To find the optimal  θ  for average  L2  loss, we now have  x  and  y  instead of  x  and  y . This means that when
writing the loss function  L(x,y,θ) , we'll need to take the average of the squared losses for each  yi ,  fθ(xi)  pair.

To do that we assume: (1/n)Σxi == (1/n)Σyi == 0. Then we do (dL/d θ 1) (1/n)( θ 1xi-yi) == (2/n) ( θ 1Σ (x)2 i-yi)xi.
Using the chain rule, this simplifies to (2/n)( θ 1Σ (x)2 i - Σyi*xi == 0 where the summation is i=1 to n.
So, essentially  θ  == (Σxi * yi) / (Σ (x)2 i) where summation is i=1 to n.
'''
```

Out[22]:
```
"To find the optimal  θ  for average  L2  loss, we now have  x  and  y  instead of  x  and  y . This means t
hat when \nwriting the loss function  L(x,y,θ) , we'll need to take the average of the squared losses for eac
h  yi ,  fθ(xi)  pair.\n\nTo do that we assume: (1/n)Σxi == (1/n)Σyi == 0. Then we do (dL/d θ 1) (1/n)( θ 1xi
-yi) == (2/n) ( θ 1Σ (x)2 i-yi)xi. \nUsing the chain rule, this simplifies to (2/n)( θ 1Σ (x)2 i - Σyi*xi ==
0 where the summation is i=1 to n. \nSo, essentially  θ  == (Σxi * yi) / (Σ (x)2 i) where summation is i=1 t
o n.\n"
```

Question 2b

Now that we have the analytic solution for $\hat{\theta}$, implement the function `find_theta` that calculates the numerical value of $\hat{\theta}$ based on our data $\mathbf{x}$, $\mathbf{y}$.

In [23]:
```
def find_theta(x, y):
    """
    Find optimal theta given x and y

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    """
    theta_opt = sum(x * y) / sum (x**2)
    # YOUR CODE HERE
    return theta_opt
    raise NotImplementedError()
```

In [24]:
```
t_hat = find_theta(x, y)
print(f'theta_opt = {t_hat}')

assert 1.4 <= t_hat <= 1.6
```

```
theta_opt = 1.5372874874787728
```

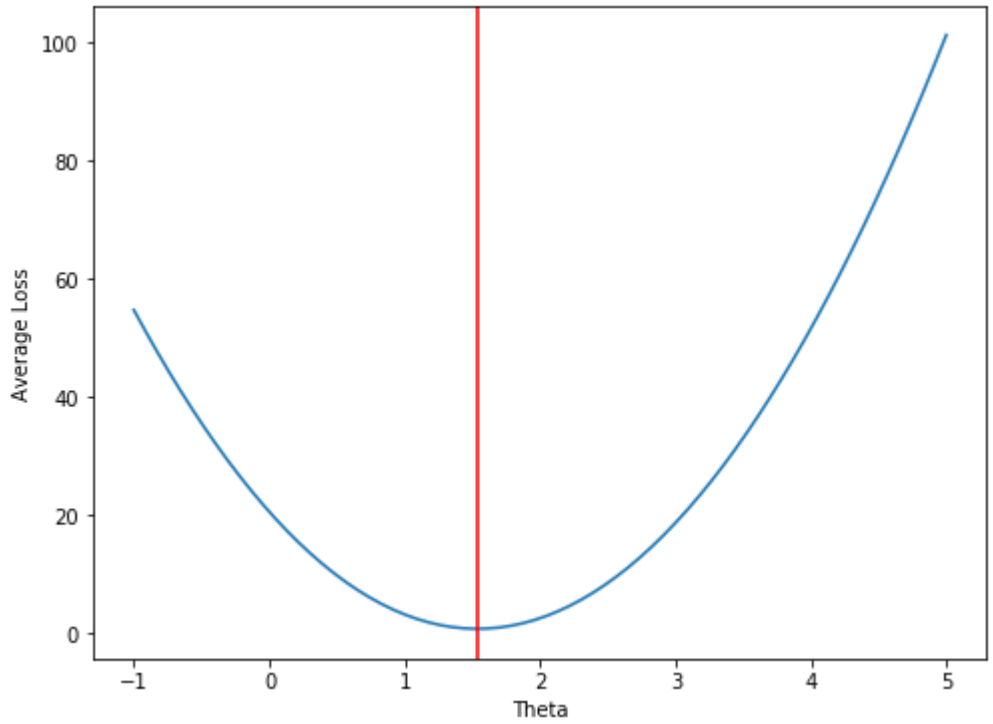
Question 2c

Now, let's plot our loss function again using the `visualize` function. But this time, add a vertical line at the optimal value of theta (plot the line $x = \hat{\theta}$). Your plot should look something like this:

 vertical_linear

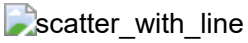
In [27]:
```
theta_opt = 1.5372874874787728
thetas = np.linspace(-1, 5, 70)
visualize(x, y, thetas)
plt.axvline(x=theta_opt, color='red')
# YOUR CODE HERE
raise NotImplementedError()
```

Out[27]: <matplotlib.lines.Line2D at 0x2adc22c3de80>



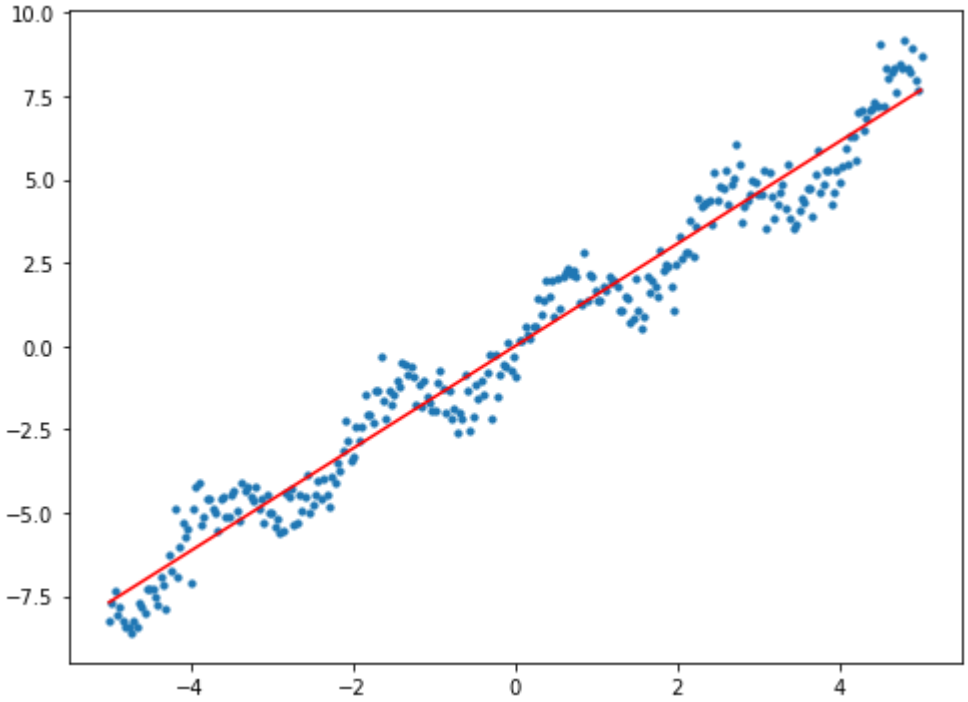
Question 2d

We now have an optimal value for θ that minimizes our loss. In the cell below, plot the scatter plot of the data from Question 1a (you can reuse the `scatter` function here). But this time, add the line $f_{\hat{\theta}}(x) = \hat{\theta} \cdot \mathbf{x}$ using the $\hat{\theta}$ you computed above. Your plot should look something like this:


scatter_with_line

In [28]:
```
theta_opt = 1.5372874874787728
x = data['x']
y = data['y']
scatter(x,y)
X = np.linspace(-5,5,300)
Y = theta_opt * x
plt.plot(X, Y, '-r')
# YOUR CODE HERE
raise NotImplementedError()
```

Out[28]:
```
[<matplotlib.lines.Line2D at 0x2adc22ec7b70>]
```

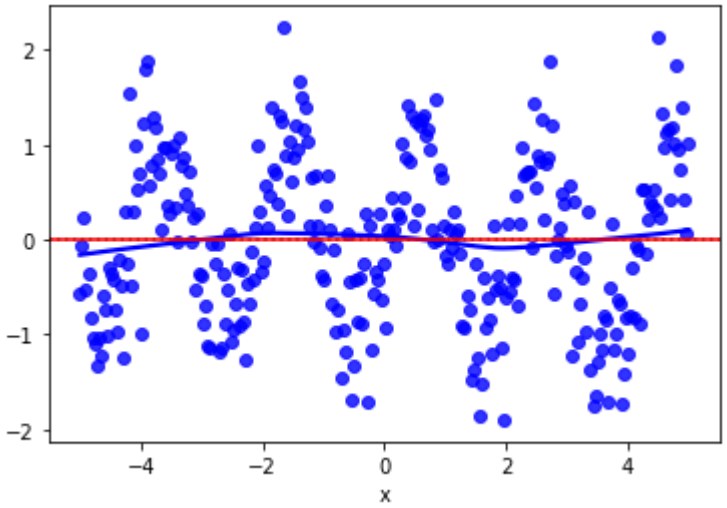


Question 2e

Great! It looks like our estimator $f_{\hat{\theta}}(x)$ is able to capture a lot of the data with a single parameter θ. Now let's try to remove the linear portion of our model from the data to see if we missed anything.

The remaining data is known as the residual, $\mathbf{r} = \mathbf{y} - \hat{\theta} \cdot \mathbf{x}$. Below, write a function to find the residual and plot the residuals corresponding to x in a scatter plot. Plot a horizontal line at $y = 0$ to assist visualization.

In [30]:
```
def visualize_residual(x, y):
    """
    Plot a scatter plot of the residuals, the remaining
    values after removing the linear model from our data.

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    """
    r = y - theta_opt * x
    sns.residplot(x, r, lowess=True, color="b")
    plt.axhline(y=0, color='r')
    # YOUR CODE HERE
    raise NotImplementedError()

visualize_residual(x, y)
```



Question 2f

What does the residual look like? Do you notice a relationship between x and r?

In [34]:
```
'''
### BEGIN SOLUTION

After using the estimator fθ (x) to capture a lot of the data with a single parameter θ and removing the line
ar portion,
the residual (r = y - theta_opt * x) looks like a zigzap line, since the relationship is that as x increases,
the r seems
to increase steadily, then decrease steadily. This is because you are plotting the values of subtracting x *
 optimal theta
from y, meaning that no matter how much x increases, depending on y, the r could be very high or very low. Fo
r example, with
x == 4 and y == -1, the r value is very low, since -1 -  θ  *(4) equals a low value of r.

### END SOLUTION

'''
# YOUR CODE HERE
# raise NotImplementedError()
```

Out[34]:
```
'       \n### BEGIN SOLUTION\n\nAfter using the estimator fθ (x) to capture a lot of the data with a single p
arameter θ and removing the linear portion, \nthe residual (r = y - theta_opt * x) looks like a zigzap line,
since the relationship is that as x increases, the r seems \nto increase steadily, then decrease steadily. Th
is is because you are plotting the values of subtracting x * optimal theta \nfrom y, meaning that no matter h
ow much x increases, depending on y, the r could be very high or very low. For example, with \nx == 4 and y =
= -1, the r value is very low, since -1 -  θ  *(4) equals a low value of r.\n\n### END SOLUTION\n\n\n'
```

3: Increasing Model Complexity

It looks like the remaining data is sinusoidal, meaning our original data follows a linear function and a sinusoidal function. Let's define a new model to address this discovery and find optimal parameters to best fit the data:

$$f_{\boldsymbol{\theta}}(x) = \theta_1 x + sin(\theta_2 x)$$

Now, our model is parameterized by both θ_1 and θ_2, or composed together, $\boldsymbol{\theta}$.

Note that a generalized sine function $a\sin(bx+c)$ has three parameters: amplitude scaling parameter a, frequency parameter b and phase shifting parameter c. Looking at the residual plot above, it looks like the residual is zero at x = 0, and the residual swings between -1 and 1. Thus, it seems reasonable to effectively set the scaling and phase shifting parameter (a and c in this case) to 1 and 0 respectively. While we could try to fit a and c, we're unlikely to get much benefit. When you're done with the homework, you can try adding a and c to our model and fitting these values to see if you can get a better loss.

Question 3a

As in Question 1, fill in the `sin_model` function that predicts $\mathbf{y}$ (the y-values) using $\mathbf{x}$ (the x-values), but this time based on our new equation.

Hint: Try to do this without using for loops. The `np.sin` function may help you.

In [35]:
```
def sin_model(x, theta_1, theta_2):
    """
    Predict the estimate of y given x, theta_1, theta_2

    Keyword arguments:
    x -- the vector of values x
    theta_1 -- the scalar value theta_1
    theta_2 -- the scalar value theta_2
    """
    y = np.array((theta_1 * x) + np.sin(theta_2*x))
    # YOUR CODE HERE
    return y
    raise NotImplementedError()
```

In [36]:
```
assert np.isclose(sin_model(1, 1, np.pi), 1.0000000000000002)
# Check that we accept x as arrays
assert len(sin_model(x, 2, 2)) > 1
```

Question 3b

Use the average L^2 loss to compute $\frac{\partial L}{\partial \theta_1}$, $\frac{\partial L}{\partial \theta_2}$.

First, we will use LaTex to write $L(\mathbf{x}, \mathbf{y}, \theta_1, \theta_2)$, $\frac{\partial L}{\partial \theta_1}$, and $\frac{\partial L}{\partial \theta_2}$ given $\mathbf{x}$, $\mathbf{y}$, $\boldsymbol{\theta}$.

You don't need to write out the full derivation. Just the final expression is fine.

In [37]:
```
# YOUR CODE HERE
'''After finding the optimal θ for average  L2  loss each  yi ,  fθ(xi)  pair, we have (1/n)Σxi == (1/n)Σyi ==
0. Then we do
(dL/d θ 1) (1/n)( θ 1xi-yi) == (2/n). ( θ 1Σ (x)2 i-yi)xi. Using the chain rule, this simplifies to (2/n)( θ
 1Σ (x)2 i - Σyi*xi
== 0 where the summation is i=1 to n. So, essentially  θ  == (Σxi * yi) / (Σ (x)2 i) where summation is i=1
 to n.

Given  x ,  y ,  θ , and using  L(x,y,θ1,θ2) ,  ∂L∂θ1  == (2/n)( θ 1Σ (x)2 i - Σyixi == 0 where the summation
is i=1 to n.
∂L∂θ2  == -2np.mean(xnp.cos(theta[1]x)(y-theta[0]x-np.sin(theta[1]*x))) == 0 where the summation is i=1 to n.

L(x,y,θ1,θ2)  == (1/n) Σ( θ 1xi + sin( θ 2xi)-yi) 2

raise NotImplementedError()'''
```

Out[37]:
```
'After finding the optimal θ for average  L2  loss each  yi ,  fθ(xi)  pair, we have (1/n)Σxi == (1/n)Σyi ==
0. Then we do \n(dL/d θ 1) (1/n)( θ 1xi-yi) == (2/n). ( θ 1Σ (x)2 i-yi)xi. Using the chain rule, this simpli
fies to (2/n)( θ 1Σ (x)2 i - Σyi*xi\n== 0 where the summation is i=1 to n. So, essentially  θ  == (Σxi * yi)
/ (Σ (x)2 i) where summation is i=1 to n.\n\nGiven  x ,  y ,  θ , and using  L(x,y,θ1,θ2) ,  ∂L∂θ1  == (2/n)(
θ 1Σ (x)2 i - Σyixi == 0 where the summation is i=1 to n.  \n∂L∂θ2  == -2np.mean(xnp.cos(theta[1]x)(y-theta
[0]x-np.sin(theta[1]*x))) == 0 where the summation is i=1 to n.\n\nL(x,y,θ1,θ2)  == (1/n) Σ( θ 1xi + sin( θ 2
xi)-yi) 2\n\nraise NotImplementedError()'
```

Question 3c

Now, implement the functions `dt1` and `dt2`, which should compute $\frac{\partial L}{\partial \theta_1}$ and $\frac{\partial L}{\partial \theta_2}$ respectively. Use the formulas you wrote for $\frac{\partial L}{\partial \theta_1}$ and $\frac{\partial L}{\partial \theta_2}$ in the previous exercise. In the functions below, the parameter `theta` is a vector that looks like (θ_1, θ_2).

Note: To keep your code a bit more concise, be aware that `np.mean` does the same thing as `np.sum` divided by the length of the numpy array.

In [38]:
```
def dt1(x, y, theta):
    """
    Compute the numerical value of the partial of l2 loss with respect to theta_1

    Keyword arguments:
    x -- the vector of all x values
    y -- the vector of all y values
    theta -- the vector of values theta
    """
    return 2*np.mean((theta[0]*x - y) * x)
    # YOUR CODE HERE
    raise NotImplementedError()
```

In [39]:
```
def dt2(x, y, theta):
    """
    Compute the numerical value of the partial of l2 loss with respect to theta_2

    Keyword arguments:
    x -- the vector of all x values
    y -- the vector of all y values
    theta -- the vector of values theta
    """
    return -2*np.mean((np.cos(theta[1]*x)*x)*(y-theta[0]*x-np.sin(theta[1]*x)))
    # YOUR CODE HERE
    raise NotImplementedError()
```

In [40]:
```
# This function calls dt1 and dt2 and returns the gradient dt. It is already implemented for you.
def dt(x, y, theta):
    """
    Returns the gradient of l2 loss with respect to vector theta

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    theta -- the vector of values theta
    """
    return np.array([dt1(x,y,theta), dt2(x,y,theta)])
```

In [41]:
```
assert np.isclose(dt1(x, y, [0, np.pi]), -25.376660670924529, rtol=0.5)
```

In [ ]:
```

```

4: Gradient Descent

We cannot try to solve for the optimal $\hat{\theta}$ exactly. We resort to an algorithm for guess-and-check to iteratively find an approximate solution. So let's try implementing gradient descent.

Question 4

Question 4a

Implement the `grad_desc` function that performs gradient descent for a finite number of iterations. This function takes in an array for $\mathbf{x}$ (`x`), an array for $\mathbf{y}$ (`y`), and an initial value for θ (`theta`). `alpha` will be the learning rate (or step size, whichever term you prefer). In this part, we'll use a static learning rate that is the same at every time step.

At each time step, use the gradient and `alpha` to update your current `theta`. Also at each time step, be sure to save the current `theta` in `theta_history`, along with the L^2 loss (computed with the current `theta`) in `loss_history`.

Hints:

- Write out the gradient update equation (1 step). What variables will you need for each gradient update? Of these variables, which ones do you already have, and which ones will you need to recompute at each time step?
- You may need a loop here to update `theta` several times
- Recall that the gradient descent update function follows the form:
$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \alpha \left(\nabla_{\boldsymbol{\theta}} \mathbf{L}(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}^{(t)}) \right)$

In [42]:
```
# Run me
def init_t():
    """Creates an initial theta [0, 0] of shape (2,) as a starting point for gradient descent"""
    return np.zeros((2,))
```

In [43]:
```
def grad_desc(x, y, theta, num_iter=20, alpha=0.1):
    """
    Run gradient descent update for a finite number of iterations and static learning rate

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    theta -- the vector of values theta to use at first iteration
    num_iter -- the max number of iterations
    alpha -- the learning rate (also called the step size)

    Return:
    theta -- the optimal value of theta after num_iter of gradient descent
    theta_history -- the series of theta values over each iteration of gradient descent
    loss_history -- the series of loss values over each iteration of gradient descent
    """
    theta_history = []
    loss_history = []

    for i in range(num_iter):
        gradient = dt(x, y, theta)
        theta = theta - alpha/(i+1)*gradient
        theta_history.append(theta)
        loss_history.append(l2_loss(sin_model(x, theta[0], theta[1]), y))

    # YOUR CODE HERE

    return theta, theta_history, loss_history
    raise NotImplementedError()
```

In [44]:
```
t = init_t()
t_est, ts, loss = grad_desc(x, y, t, num_iter=20, alpha=0.1)

assert len(ts) == len(loss) == 20 # theta history and loss history are 20 items in them
assert ts[0].shape == (2,) # theta history contains theta values
assert np.isscalar(loss[0]) # loss history is a list of scalar values, not vector
assert loss[1] - loss[-1] > 0 # loss is decreasing
```

In [ ]:
```

```

Question 4b

Now, let's try using a decaying learning rate. Implement `grad_desc_decay` below, which performs gradient descent with a learning rate that decreases slightly with each time step. You should be able to copy most of your work from the previous part, but you'll need to tweak how you update `theta` at each time step.

By decaying learning rate, we mean instead of just a number α, the learning should be now $\frac{\alpha}{i+1}$ where i is the current number of iteration. (Why do we need to add '+ 1' in the denominator?)

In [45]:
```
def grad_desc_decay(x, y, theta, num_iter=20, alpha=0.1):
    """
    Run gradient descent update for a finite number of iterations and decaying learning rate

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    theta -- the vector of values theta
    num_iter -- the max number of iterations
    alpha -- the learning rate

    Return:
    theta -- the optimal value of theta after num_iter of gradient descent
    theta_history -- the series of theta values over each iteration of gradient descent
    loss_history -- the series of loss values over each iteration of gradient descent
    """
    theta_history = []
    loss_history = []

    for i in range(num_iter):
        gradient = dt(x, y, theta)
        theta_history.append(theta)
        loss_history.append(l2_loss(sin_model(x, theta[0], theta[1]), y))
        theta = theta - (alpha * (gradient))


    # YOUR CODE HERE

    return theta, theta_history, loss_history
    raise NotImplementedError()
```

In [46]:
```
t = init_t()
t_est_decay, ts_decay, loss_decay = grad_desc_decay(x, y, t, num_iter=20, alpha=0.1)

assert len(ts_decay) == len(loss_decay) == 20 # theta history and loss history are 20 items in them
assert ts_decay[0].shape == (2,) # theta history contains theta values
assert np.isscalar(loss[0]) # loss history should be a list of values, not vector
assert loss_decay[1] - loss_decay[-1] > 0 # loss is decreasing
```

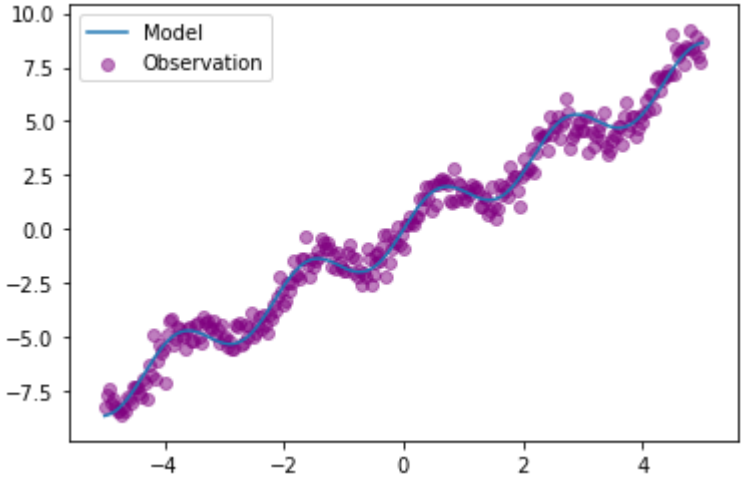
In [ ]:
```

```

Question 4c

Let's visually inspect our results of running gradient descent to optimize θ. Plot our x-values with our model's predicted y-values over the original scatter plot. Did gradient descent successfully optimize θ?

In [47]:
```
# Run me
t = init_t()
t_est, ts, loss = grad_desc(x, y, t)

t = init_t()
t_est_decay, ts_decay, loss_decay = grad_desc_decay(x, y, t)
```

In [48]:
```
y_pred = sin_model(x, t_est[0], t_est[1])

plt.plot(x, y_pred, label='Model')
plt.scatter(x, y, alpha=0.5, label='Observation', color='purple')
plt.legend();
```



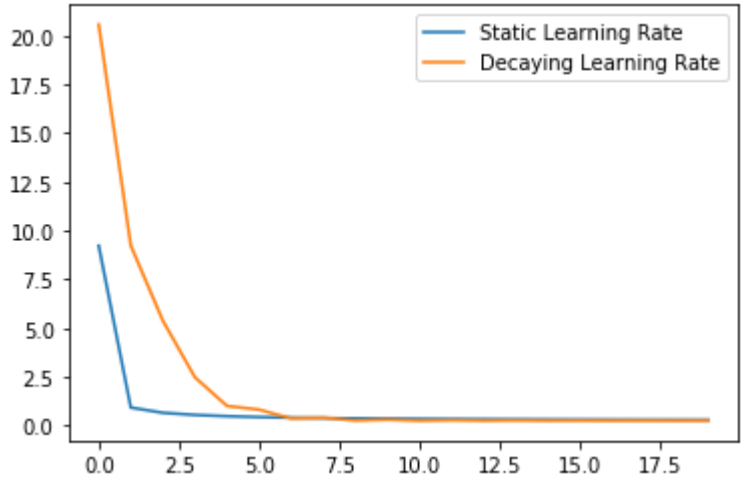
In [49]:
```
# YOUR CODE HERE
print("The gradient descent successfully optimized  θ since the actual values are close to the model's predicted y -values")
```

```
The gradient descent successfully optimized  θ since the actual values are close to the model's predicted y -values
```

Question 4d

Let's compare our two gradient descent methods and see how they differ. Plot the loss values over each iteration of gradient descent for both static learning rate and decaying learning rate.

In [50]:
```
plt.plot(loss, label='Static Learning Rate') # Plot of loss history for static learning rate
plt.plot(loss_decay, label='Decaying Learning Rate') # Plot of loss history for decaying learning rate
plt.legend();

# YOUR CODE HERE
# raise NotImplementedError()
```



Question 4e

Compare and contrast the performance of the two gradient descent methods. Which method begins to converge more quickly?

In [54]:
```
# YOUR CODE HERE
'''If we were to compare and contrast the performance of the two gradient descent methods, we would find that for both methods
the num_iter=20 and alpha=0.1. For the Static learning rate, at the lowest x value, the y value is around 9.5, whereas for the
Decaying Learning rate, you have the y value be 21 at the lowest x. So, the Decaying learning rate gradient descent method
seems to more slowly have lower loss history, whereas the static learning method more quickly lowers its loss history, possibly
due to the static learning rate.

The method that begins to converge more quickly is the static learning method, since it has a static learning rate every iteration, as opposed to the decaying learning rate which takes longer to result in a lower loss history.

'''
raise NotImplementedError()
```

Out[54]:
```
'If we were to compare and contrast the performance of the two gradient descent methods, we would find that for both methods \nthe num_iter=20 and alpha=0.1. For the Static learning rate, at the lowest x value, the y value is around 9.5, whereas for the \nDecaying Learning rate, you have the y value be 21 at the lowest x. So, the Decaying learning rate gradient descent method \nseems to more slowly have lower loss history, whereas the static learning method more quickly lowers its loss history, possibly \ndue to the static learning rate. \n\nThe method that begins to converge more quickly is the static learning method, since it has a static learning rate every iteration, as opposed to the decaying learning rate which takes longer to result in a lower loss history. \n\n'
```

5: Visualizing Loss

Question 5:

Let's visualize our loss functions and gain some insight as to how gradient descent and stochastic gradient descent are optimizing our model parameters.

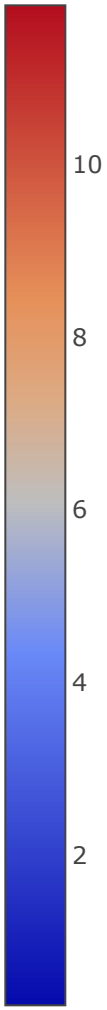
Question 5a:

In the previous plot is about the loss decrease over time, but what exactly is path the theta value? Run the following three cells.

In [55]:
```
# Run me
ts = np.array(ts).squeeze()
ts_decay = np.array(ts_decay).squeeze()
loss = np.array(loss)
loss_decay = np.array(loss_decay)
```

In [56]:
```
# Run me to see a 3D plot (gradient descent with static alpha)
plot_3d(ts[:, 0], ts[:, 1], loss, l2_loss, sin_model, x, y)
```

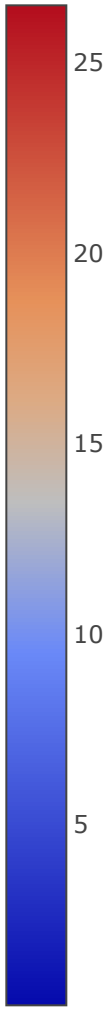
Gradient Descent



Export to plot.ly »

In [57]:
```
# Run me to see another 3D plot (gradient descent with decaying alpha)
plot_3d(ts_decay[:, 0], ts_decay[:, 1], loss_decay, l2_loss, sin_model, x, y)
```

Gradient Descent



Export to plot.ly »

In the following cell, write 1-2 sentences about the differences between using a static learning rate and a learning rate with decay for gradient descent. Use the loss history plot as well as the two 3D visualization to support your answer.

The two gradient descent methods differs in ...

In [58]:
```
# YOUR CODE HERE
'''The two gradient descent methods of using a static learning rate and a learning rate with decay for gradie
nt descent differ
based on how the static learning rate curve coverges faster due to alpha maintaining its value, meaning it ha
s a lower loss
history than the decaying learning rate curve. In terms of the two 3D visualization representations, for the
 static learning
rate model, it seems that due to static alpha, it takes less time for the loss to decrease, based on the colo
r scheme of red
to immediately dark blue, whereas, for the decaying alpha learning curve, it seems to take longer for the los
s to decrease,
based on the color scheme of red to light blue to lighter blue to dark blue, however, with a decaying learnin
g rate trend,
you can decrease your size so you don't overshoot your minimum.
'''


raise NotImplementedError()
```

Out[58]:
```
"The two gradient descent methods of using a static learning rate and a learning rate with decay for gradient
descent differ \nbased on how the static learning rate curve coverges faster due to alpha maintaining its val
ue, meaning it has a lower loss \nhistory than the decaying learning rate curve. In terms of the two 3D visua
lization representations, for the static learning \nrate model, it seems that due to static alpha, it takes l
ess time for the loss to decrease, based on the color scheme of red \nto immediately dark blue, whereas, for
the decaying alpha learning curve, it seems to take longer for the loss to decrease, \nbased on the color sch
eme of red to light blue to lighter blue to dark blue, however, with a decaying learning rate trend, \nyou ca
n decrease your size so you don't overshoot your minimum.\n"
```

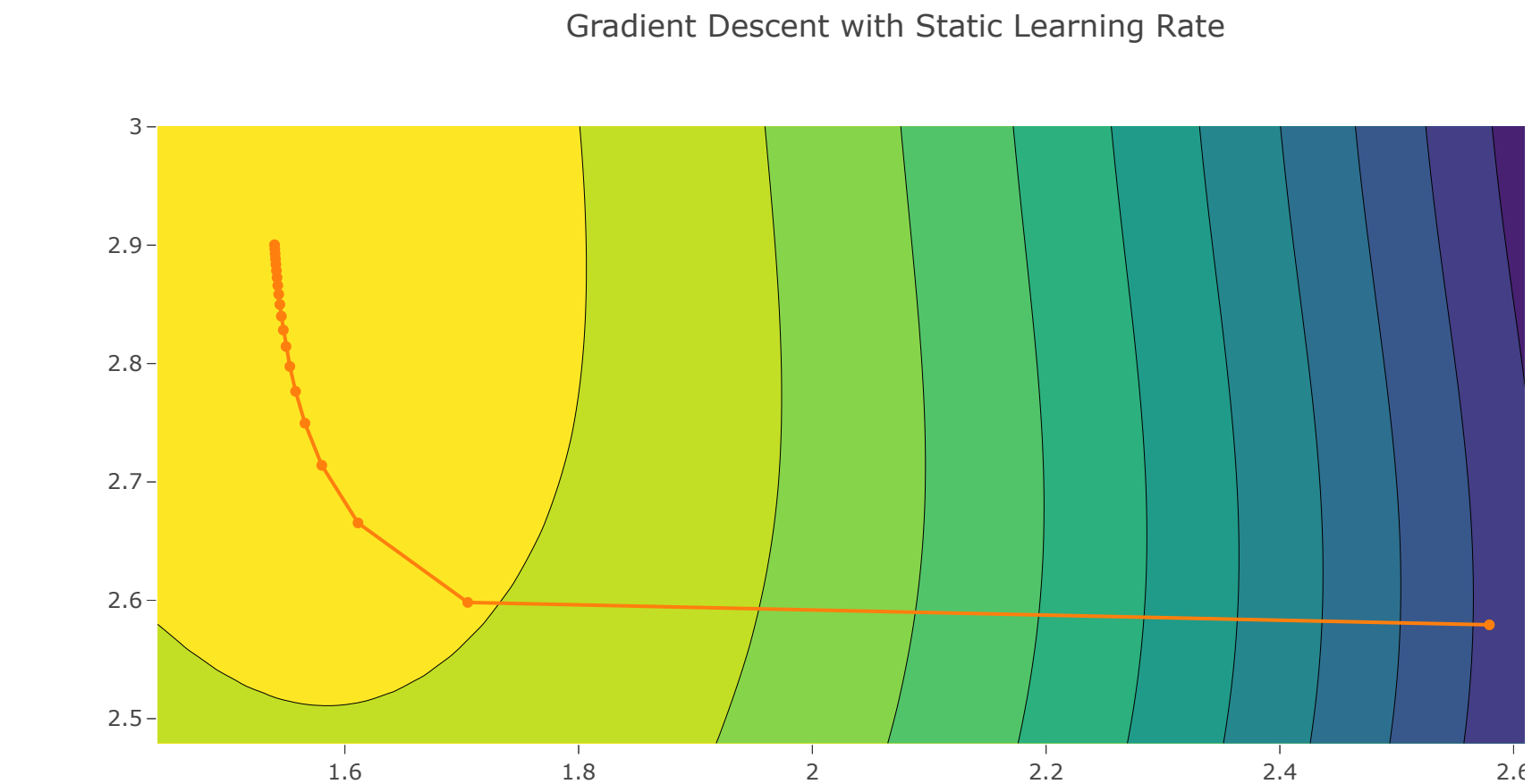
Question 5b:

Another common way of visualizing 3D dynamics is with a *contour* plot. Please run the following cell.

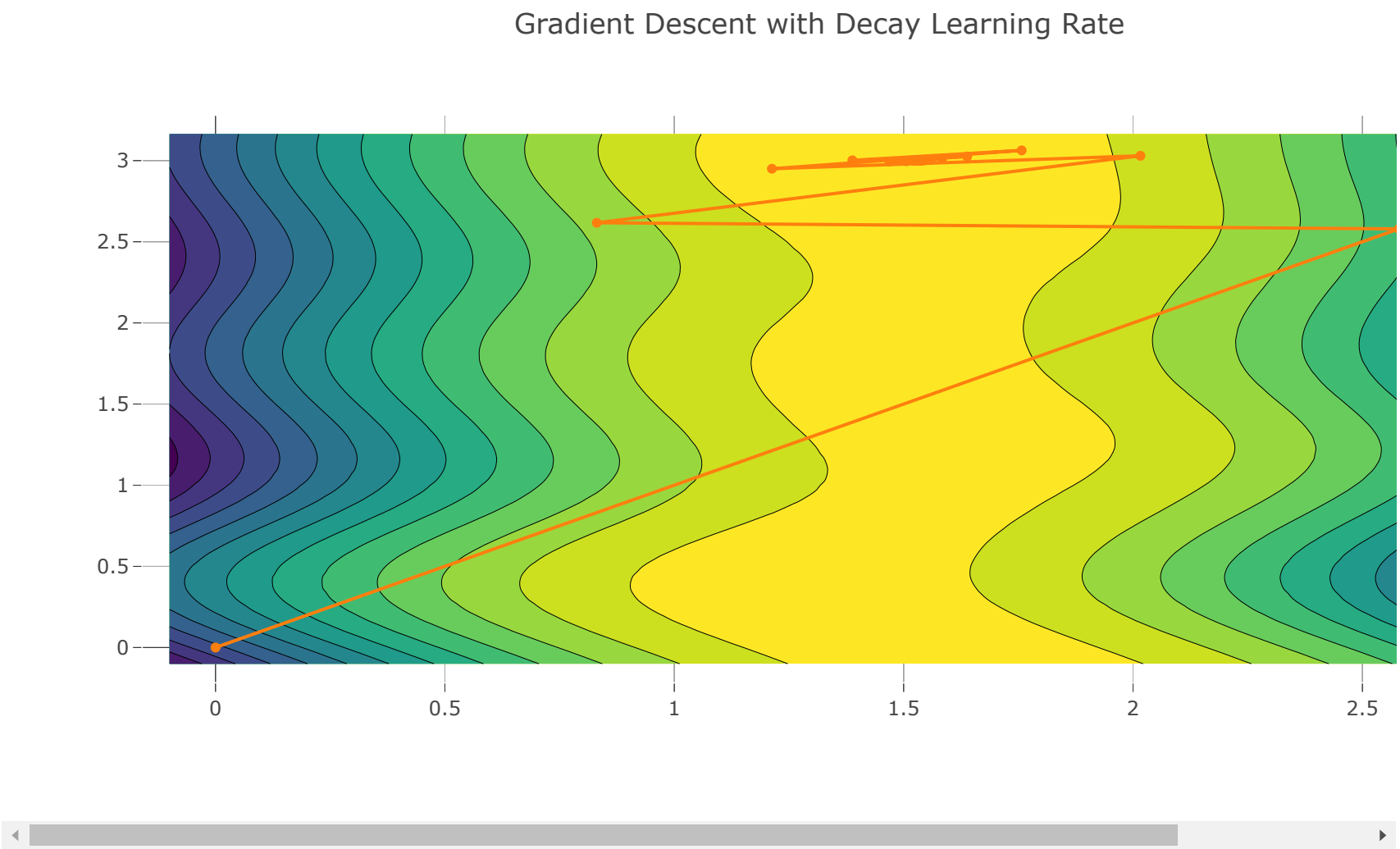
In [59]:
```
# YOUR CODE HERE
def contour_plot(title, theta_history, loss_function, model, x, y):
    """
    The function takes the following as argument:
        theta_history: a (N, 2) array of theta history
        loss: a list or array of loss value
        loss_function: for example, l2_loss
        model: for example, sin_model
        x: the original x input
        y: the original y output
    """
    theta_1_series = theta_history[:,0] # a list or array of theta_1 value
    theta_2_series = theta_history[:,1] # a list or array of theta_2 value

    # Create trace of theta point
    # Uncomment the following lines and fill in the TODOS
#     thata_points = go.Scatter(name="Theta Values",
#                               x=..., #TODO
#                               y=..., #TODO
#                               mode="lines+markers")

    ## In the following block of code, we generate the z value
    ## across a 2D grid
    t1_s = np.linspace(np.min(theta_1_series) - 0.1, np.max(theta_1_series) + 0.1)
    t2_s = np.linspace(np.min(theta_2_series) - 0.1, np.max(theta_2_series) + 0.1)

    x_s, y_s = np.meshgrid(t1_s, t2_s)
    data = np.stack([x_s.flatten(), y_s.flatten()]).T
    ls = []
    for t1, t2 in data:
        l = loss_function(model(x, t1, t2), y)
        ls.append(l)
    z = np.array(ls).reshape(50, 50)

    # Create the contour
    # Uncomment the following lines and fill in the TODOS
#     lr_loss_contours = go.Contour(x=..., #TODO
#                                   y=..., #TODO
#                                   z=..., #TODO
#                                   colorscale='Viridis', reversescale=True)
```

```
    thata_points = go.Scatter(name="Theta Values",
                              x=theta_1_series,
                              y=theta_2_series,
                              mode="lines+markers")
    lr_loss_contours = go.Contour(x=t1_s,
                                  y=t2_s,
                                  z=z,
                                  colorscale='Viridis', reversescale=True)

    plotly.offline.iplot(go.Figure(data=[lr_loss_contours, thata_points], layout={'title': title}))

raise NotImplementedError()
```

In [60]:
```
# Run this
contour_plot('Gradient Descent with Static Learning Rate', ts, l2_loss, sin_model, x, y)
```



In [61]:
```
## Run me
contour_plot('Gradient Descent with Decay Learning Rate', ts_decay, l2_loss, sin_model, x, y)
```



In the following cells, write down the answer to the following questions:

- How do you interpret the two contour plots?
- Compare contour plot and 3D plot, what are the pros and cons of each?

From reading these two contour plots, I can see that...

In [63]:
```
# YOUR CODE HERE
''' A contour plot is an isoline of a function of two variables creating a curve along which the function has
a constant value.
From reading these two contour plots, I can interpret the contour plot representing the Gradient Descent with
Static Learning
Rate as the sine model on the x values compared to the y-values. In terms of the gradient descent, there seem
s to be a sharp
decline in the y-value, and an immediate shift from the dark blue color to the yellow color, meaning that the
re is lower l2
loss (and faster learning to reduce loss) in terms of a increase in theta, but decrease in x. While the color
scheme/legend is
the same as the Gradient Descent with Decaying Learning Rate contour plot, the wave frequency seems to be les
s varied, and the
amplitude seems lower. Also, the waves for both are uniform and nearly symmetric for each wave color, except
 that the darker
colors in both seem to have thinner waves. Regarding the Gradient Descent with Decaying Learning Rate contour
plot, there seems
to be an erratic transition between different wave colors, from dark blue to bluish green to dark green to li
ghtish green to
yellow. This means that it takes longer to reduce the loss, and longer to learn how to reduce the loss.

Both the contour plots and the 3D plots both show how the gradient descent and stochastic gradient descent ar
e optimizing our
model parameters, and show the loss decrease over time as well as the path of the theta value. Lastly, both a
llow you to
interact with the plot, testing the values of x and y, and in some cases, theta. However, neither of these pl
ots seems to
determine the phase and amplitude for the four parameter model. The pros of the contour plots are that you ca
n see a more
clear transition from higher loss (represented in darker colors) to less loss (represented by lighter color
s), which shows how
using the Gradient Descent with Learning type Rate methods can help reduce loss, and demostrate the speed of
 loss reduction
learning. The Cons of contour plots are that the contour plots don't seem to always have the same wave freque
ncies and
amplitude for each colors, and some contour plots have only sided color changes, while others show color chan
ges on both sides.
Lastly, the contour plots vary in their x and y value ranges, some starting at x = 0, and others at x=1.5, wh
ich doesn't show
the progression of x and theta on the effect of y. The pros of the 3D plots are that you can easily show the
 change in the
color (representing the loss), so, and see the effect of x on y, as well as on z (if there was one). For 3D p
```

```
lots, you can test
the effect of three variables on each other. The 3D plot adds a little bit of a buffer to each edge, meaning
 that it will be
easier to estimate the loss amount based on the color hue/shade. Lastly, depending on the concavity of the 3D
plot, you can
tell how quickly the loss reduction can be. For example, with the stochastic gradient descent method there is
more of a folding
of the plot, since it takes less time for the loss to be reduced. This explains why the method that begins to
converge more
quickly is the static learning method, since it has a static learning rate every iteration, as opposed to the
decaying learning
rate which takes longer to result in a lower loss history.

The cons are that the 3D plots don't seem to have an indicator of what the theta values are. The dimensions c
ould also be very
skewed and warped, such that it is hard to follow how the color changes and the model itself. Speaking of, th
e color hue
changes are too subtle, so both of the visually representations of the static gradient descent and stochastic
gradient descent
methods seem too similar, which masks the slower color change of the stochastic (decaying) gradient descent m
ethod that is
shown in the contour plots.


'''
raise NotImplementedError()
```

Out[63]: " A contour plot is an isoline of a function of two variables creating a curve along which the function has a constant value.\n\nFrom reading these two contour plots, I can interpret the contour plot representing the Gradient Descent with Static Learning \nRate as the sine model on the x values compared to the y-values. In terms of the gradient descent, there seems to be a sharp \ndecline in the y-value, and an immediate shift from the dark blue color to the yellow color, meaning that there is lower l2 \nloss (and faster learning to reduce loss) in terms of a increase in theta, but decrease in x. While the color scheme/legend is \nthe same as the Gradient Descent with Decaying Learning Rate contour plot, the wave frequency seems to be less varied, and the \namplitude seems lower. Also, the waves for both are uniform and nearly symmetric for each wave color, except that the darker \ncolors in both seem to have thinner waves. Regarding the Gradient Descent with Decaying Learning Rate contour plot, there seems\nto be an erratic transition between different wave colors, from dark blue to bluish green to dark green to lightish green to \nyellow. This means that it takes longer to reduce the loss, and longer to learn how to reduce the loss.\n\nBoth the contour plots and the 3D plots both show how the gradient descent and stochastic gradient descent are optimizing our \nmodel parameters, and show the loss decrease over time as well as the path of the theta value. Lastly, both allow you to \ninteract with the plot, testing the values of x and y, and in some cases, theta. However, neither of these plots seems to \ndetermine the phase and amplitude for the four parameter model. The pros of the contour plots are that you can see a more \nclear transition from higher loss (represented in darker colors) to less loss (represented by lighter colors), which shows how \nusing the Gradient Descent with Learning type Rate methods can help reduce loss, and demostrate the speed of loss reduction \nlearning. The Cons of contour plots are that the contour plots don't seem to always have the same wave frequencies and \namplitude for each colors, and some contour plots have only sided color changes, while others show color changes on both sides.\nLastly, the contour plots vary in their x and y value ranges, some starting at x = 0, and others at x=1.5, which doesn't show \nthe progression of x and theta on the effect of y. The pros of the 3D plots are that you can easily show the change in the \ncolor (representing the loss), so, and see the effect of x on y, as well as on z (if there was one). For 3D plots, you can test\nthe effect of three variables on each other. The 3D plot adds a little bit of a buffer to each edge, meaning that it will be \neasier to estimate the loss amount based on the color hue/shade. Lastly, depending on the concavity of the 3D plot, you can \ntell how quickly the loss reduction can be. For example, with the stochastic gradient descent method there is more of a folding \nof the plot, since it takes less time for the loss to be reduced. This explains why the method that begins to converge more \nquickly is the static learning method, since it has a static learning rate every iteration, as opposed to the decaying learning \nrate which takes longer to result in a lower loss history.\n\nThe cons are that the 3D plots don't seem to have an indicator of what the theta values are. The dimensions could also be very \nskewed and warped, such that it is hard to follow how the color changes and the model itself. Speaking of, the color hue \nchanges are too subtle, so both of the visually representations of the static gradient descent and stochastic gradient descent \nmethods seem too similar, which masks the slower color change of the stochastic (decaying) gradient descent method that is \nshown in the contour plots.\n\n\n"

Question 5c

Try adding the two additional model parameters for phase and amplitude that we ignored (see 3a). What are the optimal phase and amplitude values for your four parameter model? Do you get a better loss?

I think...

In [64]:
```
# YOUR CODE HERE
''' We represented the original data in a linear function and a sinusoidal function and found a new model to
 address this
discovery and find optimal parameters to best fit the data: fθ(x)=θ1x+sin(θ2x) where the model is parameteriz
ed by both θ1 and
θ2 , or composed together, θ . In our sine function asin(bx+c) we have three parameters: amplitude scaling pa
rameter a ,
frequency parameter b and phase shifting parameter c . We originally set the scaling and phase shifting param
eter ( a and c in
this case) to 1 and 0 respectively. As stated: "The coefficient b and the period of the sine curve have an in
verse relationship,
so as b gets smaller, the length of one cycle of the curve gets bigger. Likewise, as you increase b, the peri
od will decrease.
What is the phase shift of a sine curve? The phase shift of a sine curve is how much the curve shifts from ze
ro."

I think if you add the two additional model parameters for phase (c) and amplitude (a) that we ignored, the a
mplitude would
certainly increase or decrease, meaning the gradient descent to appear more "steep" depending on the value. T
he phase shift
would just mean the the curve shifts more from zero.

The optimal phase and amplitude values for the four parameter model would be the same as the original gradien
t descent graph,
but the graph would be slightly elevated and the gradient descent graph is slightly steeper.

Do you get a better loss? yes, because if you make the gradient descent curve "steeper" and shift the graph o
ver, you will
make the same trend, but the loss starts at 14, and plateus at 6, whereas in the original static learning lea
rning alpha
gradient descent curve starts at 9 and plateaus at 1. So, this essentially gets you less loss, which means th
at there is a
better loss.


'''
raise NotImplementedError()
```

Out[64]: ' We represented the original data in a linear function and a sinusoidal function and found a new model to address this \ndiscovery and find optimal parameters to best fit the data: $f\boldsymbol{\theta}(x)=\theta 1x+sin(\theta 2x)$ where the model is parameterized by both $\theta 1$ and \n$\theta 2$, or composed together, $\boldsymbol{\theta}$. In our sine function $a\sin(bx+c)$ we have three parameters: amplitude scaling parameter a , \nfrequency parameter b and phase shifting parameter c . We originally set the scaling and phase shifting parameter (a and c in \nthis case) to 1 and 0 respectively. As stated: "The coefficient b and the period of the sine curve have an inverse relationship,\nso as b gets smaller, the length of one cycle of the curve gets bigger. Likewise, as you increase b, the period will decrease. \nWhat is the phase shift of a sine curve? The phase shift of a sine curve is how much the curve shifts from zero."\n\nI think if you add the two additional model parameters for phase (c) and amplitude (a) that we ignored, the amplitude would \ncertainly increase or decrease, meaning the gradient descent to appear more "steep" depending on the value. The phase shift \nwould just mean the the curve shifts more from zero.\n\nThe optimal phase and amplitude values for the four parameter model would be the same as the original gradient descent graph, \nbut the graph would be slightly elevated and the gradient descent graph is slightly steeper.\n\nDo you get a better loss? yes, because if you make the gradient descent curve "steeper" and shift the graph over, you will \nmake the same trend, but the loss starts at 14, and plateus at 6, whereas in the original static learning learning alpha \ngradient descent curve starts at 9 and plateaus at 1. So, this essentially gets you less loss, which means that there is a \nbetter loss.\n\n\n'

Question 5d (optional)

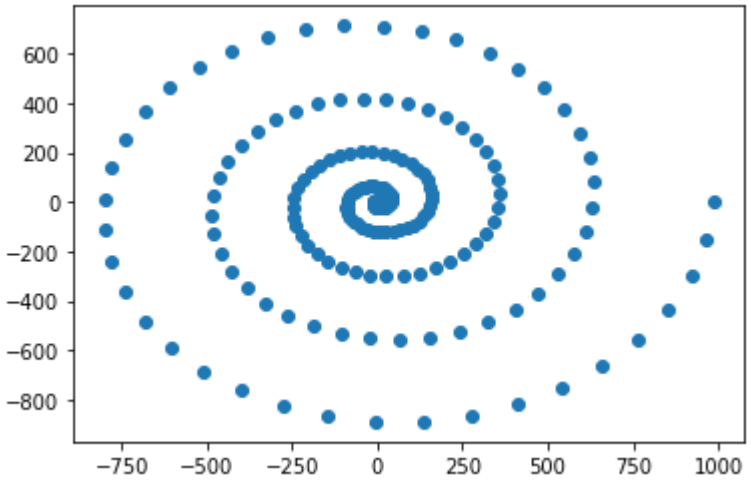
It looks like our basic two parameter model, a combination of a linear function and sinusoidal function, was able to almost perfectly fit our data. It turns out that many real world scenarios come from relatively simple models.

At the same time, the real world can be incredibly complex and a simple model wouldn't work so well. Consider the example below; it is neither linear, nor sinusoidal, nor quadratic.

Optional: Suggest how we could iteratively create a model to fit this data and how we might improve our results.

Extra optional: Try and build a model that fits this data.

In [65]:
```
x = []
y = []
for t in np.linspace(0,10*np.pi, 200):
    r = ((t)**2)
    x.append(r*np.cos(t))
    y.append(r*np.sin(t))

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



With our two parameter model, a combination of a linear function and sinusoidal function, was able to almost perfectly fit our data, we can suggest how to iteratively create a model to fit this data and how we might improve our results.

I suggest the following model of exponential decay which will improve the results by showing more of a exponential decay or growth of the data.

Homework 5: Modeling, Estimation and Gradient Descent

Due Date: Saturday 11/23, 11:59 PM

We will explore modeling through estimation and prediction. Along the way, we will get experience with an iterative method for guess-and-check called gradient descent. These approaches are helpful throughout data science particularly the field of machine learning. By completing Homework 5, you should take away...

- Practice reasoning about a model and building intuition for loss functions.
- Determining the gradient of a loss function with respect to model parameters and using the calculations for gradient descent.

We will apply these skills in Homework 6 to a real-world dataset.

Submission Instructions

For this assignment, you will submit a copy to Gradescope. Follow these steps

1. Download as HTML (`File->Download As->HTML(.html)`).
2. Open the HTML in the browser. Print to .pdf
3. Upload to Gradescope. Tag your answers.

Note that

- Please map your answers to our questions. Otherwise you may lose points. Please see the rubric below.
- You should break long lines of code into multiple lines. Otherwise your code will extend out of view from the cell. Consider using `\` followed by a new line.
- For each textual response, please include relevant code that informed your response. For each plotting question, please include the code used to generate the plot.
- You should not display large output cells such as all rows of a table. Instead convert the input cell from Code to Markdown back to Code to remove the output cell.

Moreover you will submit a copy on Jupyter Hub under Assignments Tab. You cannot access the extension in JupyterLab. So if the URL ends with lab, then please change it to tree

```
https://pds-f19.jupyter.hpc.nyu.edu/user/[Your NetID]/tree
```

Consult the instructional video

https://nbgrader.readthedocs.io/en/stable/_images/student_assignment.gif

for steps to...

1. fetch
2. modify
3. optionally validate
4. submit your project

Failure to follow these guidelines for submission could mean the deduction of two points. See the rubric below.

Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the homework, we ask that you **write your solutions individually**. If you do discuss the assignments with others please **include their names** at the top of your solution.

Rubric

Question	Points
Gradescope	2
Question 1a	1
Question 1b	1
Question 1c	1
Question 1d	1
Question 1e	1
Question 2a	2
Question 2b	1
Question 2c	1
Question 2d	1
Question 2e	1
Question 2f	1
Question 3a	1
Question 3b	3

Question	Points
Question 3c	2
Question 4a	3
Question 4b	1
Question 4c	1
Question 4d	1
Question 4e	1
Question 5a	2
Question 5b	1
Question 5c	1
Question 5d	0
Total	37

Getting Started

In [6]:
```
from IPython.display import display, Markdown

# Import standard packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import csv
import re
import seaborn as sns

# Set some parameters
plt.rcParams['figure.figsize'] = (12, 9)
plt.rcParams['font.size'] = 16
np.set_printoptions(4)
```

In [7]:
```
# Import a specialized plotting package useful for 3d
import plotly
import plotly.graph_objs as go
plotly.offline.init_notebook_mode(connected=True)
from hw5_utils import plot_3d

# Note that you may need to install plotly to import these packages
# Either execucte the next cell or complete the assignment on JupyterHub
```

In [8]:
```
# !pip install plotly
# !conda install -c conda-forge jupyterlab-plotly-extension
```

Load Data

Load the data.csv file into a pandas dataframe.
Note that we are reading the data directly from the URL address.

In [9]:
```
# Run this cell to load our sample data
data = pd.read_csv("data.csv", index_col=0)
data.head()
```

Out[9]:

	x	y
0	-5.000000	-8.262369
1	-4.966555	-7.692411
2	-4.933110	-7.358698
3	-4.899666	-8.067488
4	-4.866221	-7.834256

1: A Simple Model

Let's start by examining our data and creating a simple model that can represent this data.

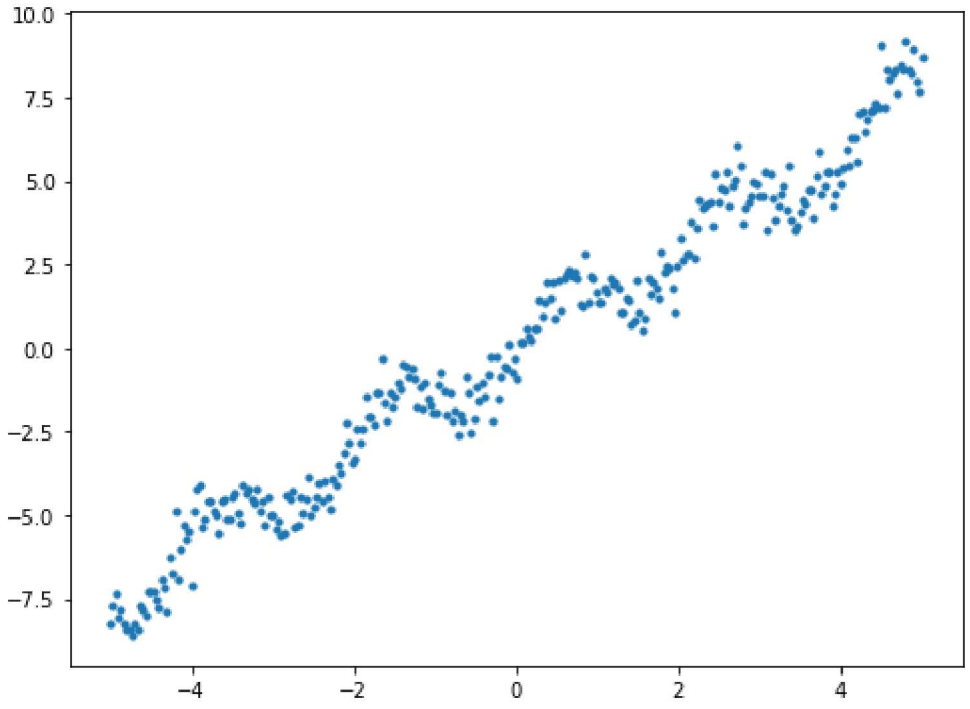
Question 1

Question 1a

First, let's visualize the data in a scatter plot. After implementing the `scatter` function below, you should see something like this:

 scatter

In [11]:
```
def scatter(x, y):
    """
    Generate a scatter plot using x and y

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    """
    plt.figure(figsize=(8, 6))
    plt.scatter(x, y, s=10)
    # YOUR CODE HERE
    raise NotImplementedError()

x = data['x']
y = data['y']
scatter(x,y)
```



Question 1b

Describe any significant observations about the distribution of the data. How can you describe the relationship between x and y?

In [14]:
```
# YOUR CODE HERE
'''x and y are said to be vectors of data points. Based on the scatter plot, there seems to be a strong posit
ive linear
correlation between x and y, where as x increases, y increases. A significant observation about the distribut
ion of the data
is that for x and y, the data seems to be concentrated heavily towards the negative values of x and y at firs
t.
As said before, the data is positive linear, and there is no outliers, as well as no gaps.
The data seems to be tighly compact, meaning there is no deviation, or loss from other adjacent data points.
In essence, it's very possible to calculate the r as .90, and graph a linear regression line, given the data.
Another important factor is that the x axis values range from ~-2 to 6, whereas the y axis values range from
 ~-8.0 to 10,
meaning there is variability within the data ranges.

raise NotImplementedError()'''
```

Out[14]:
```
"x and y are said to be vectors of data points. Based on the scatter plot, there seems to be a strong positiv
e linear \ncorrelation between x and y, where as x increases, y increases. A significant observation about th
e distribution of the data \nis that for x and y, the data seems to be concentrated heavily towards the negat
ive values of x and y at first. \nAs said before, the data is positive linear, and there is no outliers, as w
ell as no gaps. \nThe data seems to be tighly compact, meaning there is no deviation, or loss from other adja
cent data points. \nIn essence, it's very possible to calculate the r as .90, and graph a linear regression l
ine, given the data. \nAnother important factor is that the x axis values range from ~-2 to 6, whereas the y
axis values range from ~-8.0 to 10,\nmeaning there is variability within the data ranges. \n\nraise NotImplem
entedError()"
```

Question 1c

The data looks roughly linear. However it moves up and down away from the line. For now, let's assume that the data follows some underlying linear model. We define the underlying linear model that predicts the value y using the value x as: $f_{\theta^*}(x) = \theta^* \cdot x$

Since we cannot find the value of the population parameter θ^* exactly, we will assume that our dataset approximates our population and use our dataset to estimate θ^*. We denote our estimation with θ, our fitted estimation with $\hat{\theta}$, and our model as:

$$f_\theta(x) = \theta \cdot x$$

Based on this equation, define the linear model function `linear_model` below to estimate $\mathbf{y}$ (the y-values) given $\mathbf{x}$ (the x-values) and θ. This model is similar to the model you defined in Lab 5: Modeling and Estimation.

In [15]:
```
def linear_model(x, theta):
    """
    Returns the estimate of y given x and theta

    Keyword arguments:
    x -- the vector of values x
    theta -- the scalar theta
    """
    y = x * theta
    # YOUR CODE HERE
    return y
    raise NotImplementedError()
```

In [16]:
```
assert linear_model(0, 1) == 0
assert np.sum(linear_model(np.array([3, 5]), 3)) == 24
```

In [ ]:
```

```

Question 1d

In class, we learned that the square loss, sometimes called the L^2 loss. Let's use L^2 loss to evaluate our estimate θ, which we will use later to identify an optimal θ, represented as $\hat{\theta}$. Define the L^2 loss function `l2_loss` below.

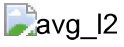
In [17]:
```
def l2_loss(y, y_hat):
    """
    Returns the average l2 loss given y and y_hat

    Keyword arguments:
    y -- the vector of true values y
    y_hat -- the vector of predicted values y_hat
    """
    if type(y) == int:
        return (y-y_hat)**2
    return (sum((y - y_hat)** 2))/len(y)
    # YOUR CODE HERE
    raise NotImplementedError()
```

In [18]:
```
assert l2_loss(2, 1) == 1
assert l2_loss(2, 0) == 4
assert l2_loss(np.array([5, 6]), np.array([1, 1])) == 20.5
```

In [ ]:
```

```

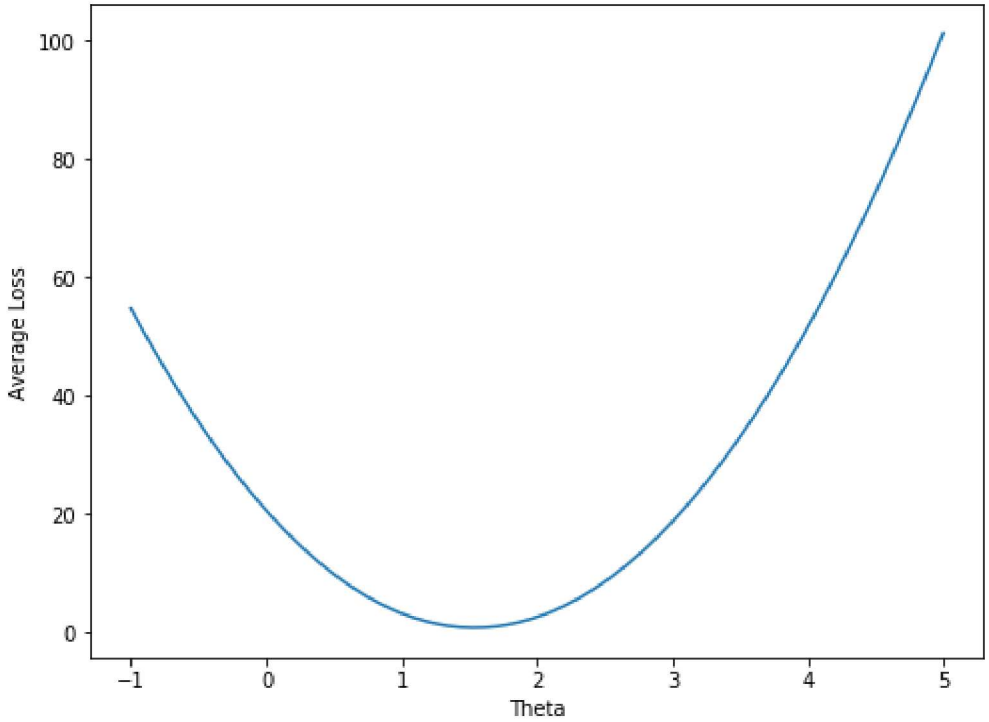
Question 1e

First, visualize the L^2 loss as a function of θ, where several different values of θ are given. Be sure to label your axes properly. You plot should look something like this:

 avg_l2

What looks like the optimal value, $\hat{\theta}$, based on the visualization? Set `theta_star_guess` to the value of θ that appears to minimize our loss.

In [20]:
```
def visualize(x, y, thetas):
    """
    Plots the average l2 loss for given x, y as a function of theta.
    Use the functions you wrote for linear_model and l2_loss.

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    thetas -- an array containing different estimates of the scalar theta
    """
    avg_loss = np.array([l2_loss(linear_model(theta,x),y) for theta in thetas])# Calculate the loss here for
 each value of theta

    plt.figure(figsize=(8,6))

    plt.plot(thetas, avg_loss)
    plt.xlabel(r"Theta")
    plt.ylabel(r"Average Loss") # Create your plot here
#     raise NotImplementedError()

thetas = np.linspace(-1, 5, 70)
visualize(x, y, thetas)

theta_star_guess = 1.55
# YOUR CODE HERE
raise NotImplementedError()
```



In [21]:
```
assert l2_loss(3, 2) == 1
assert l2_loss(0, 10) == 100
```

In [ ]:
```

```

2: Fitting our Simple Model

Now that we have defined a simple linear model and loss function, let's begin working on fitting our model to the data.

Question 2

Let's confirm our visual findings for optimal $\hat{\theta}$.

Question 2a

First, find the analytical solution for the optimal $\hat{\theta}$ for average L^2 loss. Write up your solution in the cell below using LaTex.

Hint: notice that we now have $\mathbf{x}$ and $\mathbf{y}$ instead of x and y. This means that when writing the loss function $L(\mathbf{x}, \mathbf{y}, \theta)$, you'll need to take the average of the squared losses for each y_i, $f_\theta(x_i)$ pair.

For tips on getting started, see chapter 10 (https://www.textbook.ds100.org/ch/10/modeling_loss_functions.html) of the textbook. Note that if you check the source (https://github.com/DS-100/textbook/tree/master/content/ch/10), you can access the LaTeX code of the book chapter, which you might find handy for typing up your work. Show your work, i.e. don't just write the answer.

In [22]:
```
# YOUR CODE HERE
'''To find the optimal  θ  for average  L2  loss, we now have  x  and  y  instead of  x  and  y . This means that when
writing the loss function  L(x,y,θ) , we'll need to take the average of the squared losses for each  yi ,  fθ(xi)  pair.

To do that we assume: (1/n)Σxi == (1/n)Σyi == 0. Then we do (dL/d θ 1) (1/n)( θ 1xi-yi) == (2/n) ( θ 1Σ (x)2 i-yi)xi.
Using the chain rule, this simplifies to (2/n)( θ 1Σ (x)2 i - Σyi*xi == 0 where the summation is i=1 to n.
So, essentially  θ  == (Σxi * yi) / (Σ (x)2 i) where summation is i=1 to n.
'''
```

Out[22]:
```
"To find the optimal  θ  for average  L2  loss, we now have  x  and  y  instead of  x  and  y . This means t
hat when \nwriting the loss function  L(x,y,θ) , we'll need to take the average of the squared losses for eac
h  yi ,  fθ(xi)  pair.\n\nTo do that we assume: (1/n)Σxi == (1/n)Σyi == 0. Then we do (dL/d θ 1) (1/n)( θ 1xi
-yi) == (2/n) ( θ 1Σ (x)2 i-yi)xi. \nUsing the chain rule, this simplifies to (2/n)( θ 1Σ (x)2 i - Σyi*xi ==
0 where the summation is i=1 to n. \nSo, essentially  θ  == (Σxi * yi) / (Σ (x)2 i) where summation is i=1 t
o n.\n"
```

Question 2b

Now that we have the analytic solution for $\hat{\theta}$, implement the function `find_theta` that calculates the numerical value of $\hat{\theta}$ based on our data $\mathbf{x}$, $\mathbf{y}$.

In [23]:
```
def find_theta(x, y):
    """
    Find optimal theta given x and y

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    """
    theta_opt = sum(x * y) / sum (x**2)
    # YOUR CODE HERE
    return theta_opt
    raise NotImplementedError()
```

In [24]:
```
t_hat = find_theta(x, y)
print(f'theta_opt = {t_hat}')

assert 1.4 <= t_hat <= 1.6
```

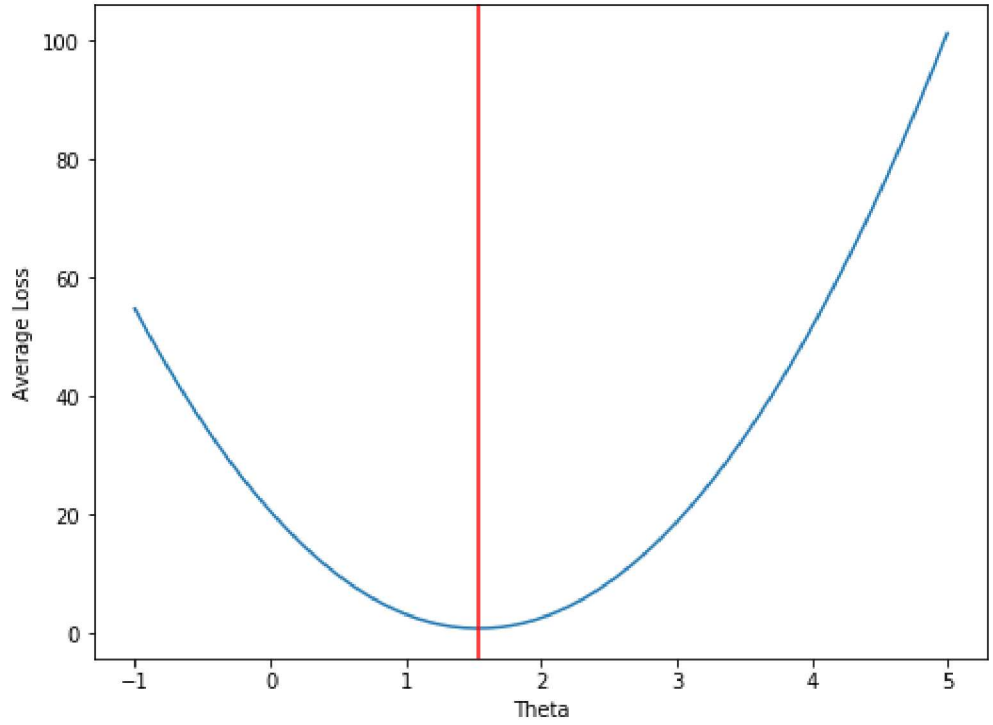
```
theta_opt = 1.5372874874787728
```

Question 2c

Now, let's plot our loss function again using the `visualize` function. But this time, add a vertical line at the optimal value of theta (plot the line $x = \hat{\theta}$). Your plot should look something like this:

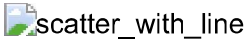
In [27]:
```
theta_opt = 1.5372874874787728
thetas = np.linspace(-1, 5, 70)
visualize(x, y, thetas)
plt.axvline(x=theta_opt, color='red')
# YOUR CODE HERE
raise NotImplementedError()
```

Out[27]: <matplotlib.lines.Line2D at 0x2adc22c3de80>



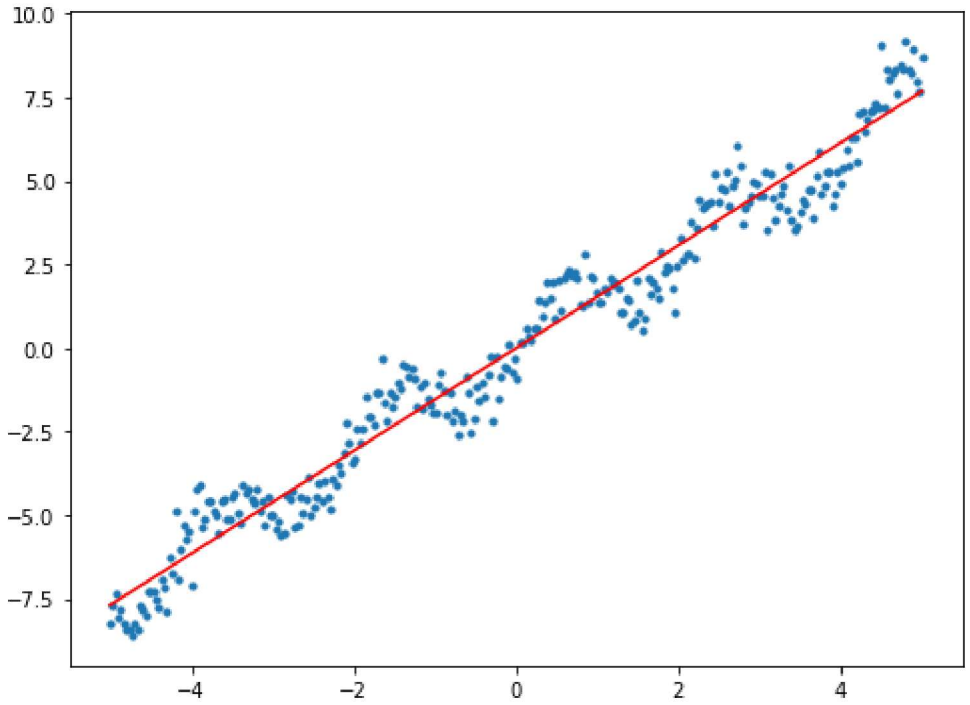
Question 2d

We now have an optimal value for θ that minimizes our loss. In the cell below, plot the scatter plot of the data from Question 1a (you can reuse the `scatter` function here). But this time, add the line $f_{\hat{\theta}}(x) = \hat{\theta} \cdot \mathbf{x}$ using the $\hat{\theta}$ you computed above. Your plot should look something like this:


scatter_with_line

In [28]:
```
theta_opt = 1.5372874874787728
x = data['x']
y = data['y']
scatter(x,y)
X = np.linspace(-5,5,300)
Y = theta_opt * x
plt.plot(X, Y, '-r')
# YOUR CODE HERE
raise NotImplementedError()
```

Out[28]: [<matplotlib.lines.Line2D at 0x2adc22ec7b70>]

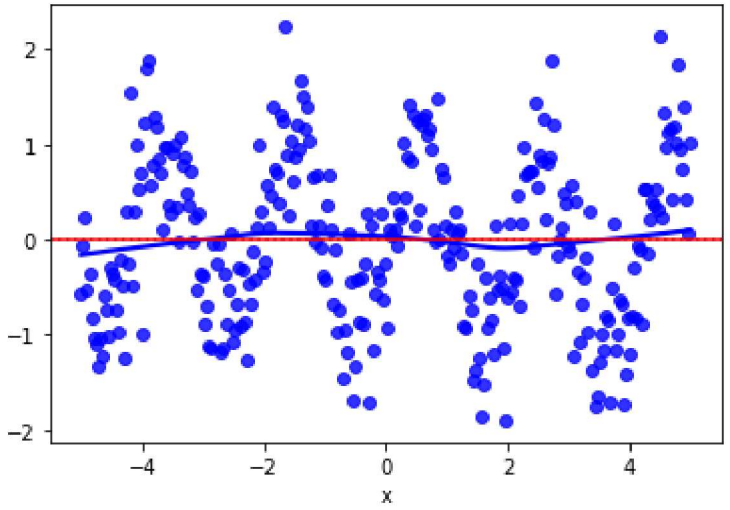


Question 2e

Great! It looks like our estimator $f_{\hat{\theta}}(x)$ is able to capture a lot of the data with a single parameter θ. Now let's try to remove the linear portion of our model from the data to see if we missed anything.

The remaining data is known as the residual, $\mathbf{r} = \mathbf{y} - \hat{\theta} \cdot \mathbf{x}$. Below, write a function to find the residual and plot the residuals corresponding to x in a scatter plot. Plot a horizontal line at $y = 0$ to assist visualization.

In [30]:
```
def visualize_residual(x, y):
    """
    Plot a scatter plot of the residuals, the remaining
    values after removing the linear model from our data.

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    """
    r = y - theta_opt * x
    sns.residplot(x, r, lowess=True, color="b")
    plt.axhline(y=0, color='r')
    # YOUR CODE HERE
    raise NotImplementedError()

visualize_residual(x, y)
```



Question 2f

What does the residual look like? Do you notice a relationship between x and r?

In [34]:
```
'''
### BEGIN SOLUTION

After using the estimator fθ (x) to capture a lot of the data with a single parameter θ and removing the line
ar portion,
the residual (r = y - theta_opt * x) looks like a zigzap line, since the relationship is that as x increases,
the r seems
to increase steadily, then decrease steadily. This is because you are plotting the values of subtracting x *
 optimal theta
from y, meaning that no matter how much x increases, depending on y, the r could be very high or very low. Fo
r example, with
x == 4 and y == -1, the r value is very low, since -1 -  θ  *(4) equals a low value of r.

### END SOLUTION

'''
# YOUR CODE HERE
# raise NotImplementedError()
```

Out[34]:
```
'       \n### BEGIN SOLUTION\n\nAfter using the estimator fθ (x) to capture a lot of the data with a single p
arameter θ and removing the linear portion, \nthe residual (r = y - theta_opt * x) looks like a zigzap line,
since the relationship is that as x increases, the r seems \nto increase steadily, then decrease steadily. Th
is is because you are plotting the values of subtracting x * optimal theta \nfrom y, meaning that no matter h
ow much x increases, depending on y, the r could be very high or very low. For example, with \nx == 4 and y =
= -1, the r value is very low, since -1 -  θ  *(4) equals a low value of r.\n\n### END SOLUTION\n\n\n'
```

3: Increasing Model Complexity

It looks like the remaining data is sinusoidal, meaning our original data follows a linear function and a sinusoidal function. Let's define a new model to address this discovery and find optimal parameters to best fit the data:

$$f_{\boldsymbol{\theta}}(x) = \theta_1 x + sin(\theta_2 x)$$

Now, our model is parameterized by both θ_1 and θ_2, or composed together, $\boldsymbol{\theta}$.

Note that a generalized sine function $a\sin(bx+c)$ has three parameters: amplitude scaling parameter a, frequency parameter b and phase shifting parameter c. Looking at the residual plot above, it looks like the residual is zero at x = 0, and the residual swings between -1 and 1. Thus, it seems reasonable to effectively set the scaling and phase shifting parameter (a and c in this case) to 1 and 0 respectively. While we could try to fit a and c, we're unlikely to get much benefit. When you're done with the homework, you can try adding a and c to our model and fitting these values to see if you can get a better loss.

Question 3a

As in Question 1, fill in the `sin_model` function that predicts $\mathbf{y}$ (the y-values) using $\mathbf{x}$ (the x-values), but this time based on our new equation.

Hint: Try to do this without using for loops. The `np.sin` function may help you.

In [35]:
```
def sin_model(x, theta_1, theta_2):
    """
    Predict the estimate of y given x, theta_1, theta_2

    Keyword arguments:
    x -- the vector of values x
    theta_1 -- the scalar value theta_1
    theta_2 -- the scalar value theta_2
    """
    y = np.array((theta_1 * x) + np.sin(theta_2*x))
    # YOUR CODE HERE
    return y
    raise NotImplementedError()
```

In [36]:
```
assert np.isclose(sin_model(1, 1, np.pi), 1.0000000000000002)
# Check that we accept x as arrays
assert len(sin_model(x, 2, 2)) > 1
```

Question 3b

Use the average L^2 loss to compute $\frac{\partial L}{\partial \theta_1}$, $\frac{\partial L}{\partial \theta_2}$.

First, we will use LaTex to write $L(\mathbf{x}, \mathbf{y}, \theta_1, \theta_2)$, $\frac{\partial L}{\partial \theta_1}$, and $\frac{\partial L}{\partial \theta_2}$ given $\mathbf{x}$, $\mathbf{y}$, $\boldsymbol{\theta}$.

You don't need to write out the full derivation. Just the final expression is fine.

In [37]:
```
# YOUR CODE HERE
'''After finding the optimal θ for average  L2  loss each  yi ,  fθ(xi)  pair, we have (1/n)Σxi == (1/n)Σyi ==
0. Then we do
(dL/d θ 1) (1/n)( θ 1xi-yi) == (2/n). ( θ 1Σ (x)2 i-yi)xi. Using the chain rule, this simplifies to (2/n)( θ
 1Σ (x)2 i - Σyi*xi
== 0 where the summation is i=1 to n. So, essentially  θ  == (Σxi * yi) / (Σ (x)2 i) where summation is i=1
 to n.

Given  x ,  y ,  θ , and using  L(x,y,θ1,θ2) ,  ∂L∂θ1  == (2/n)( θ 1Σ (x)2 i - Σyixi == 0 where the summation
is i=1 to n.
∂L∂θ2  == -2np.mean(xnp.cos(theta[1]x)(y-theta[0]x-np.sin(theta[1]*x))) == 0 where the summation is i=1 to n.

L(x,y,θ1,θ2)  == (1/n) Σ( θ 1xi + sin( θ 2xi)-yi) 2

raise NotImplementedError()'''
```

Out[37]:
```
'After finding the optimal θ for average  L2  loss each  yi ,  fθ(xi)  pair, we have (1/n)Σxi == (1/n)Σyi ==
0. Then we do \n(dL/d θ 1) (1/n)( θ 1xi-yi) == (2/n). ( θ 1Σ (x)2 i-yi)xi. Using the chain rule, this simpli
fies to (2/n)( θ 1Σ (x)2 i - Σyi*xi\n== 0 where the summation is i=1 to n. So, essentially  θ  == (Σxi * yi)
/ (Σ (x)2 i) where summation is i=1 to n.\n\nGiven  x ,  y ,  θ , and using  L(x,y,θ1,θ2) ,  ∂L∂θ1  == (2/n)(
θ 1Σ (x)2 i - Σyixi == 0 where the summation is i=1 to n.  \n∂L∂θ2  == -2np.mean(xnp.cos(theta[1]x)(y-theta
[0]x-np.sin(theta[1]*x))) == 0 where the summation is i=1 to n.\n\nL(x,y,θ1,θ2)  == (1/n) Σ( θ 1xi + sin( θ 2
xi)-yi) 2\n\nraise NotImplementedError()'
```

Question 3c

Now, implement the functions `dt1` and `dt2`, which should compute $\frac{\partial L}{\partial \theta_1}$ and $\frac{\partial L}{\partial \theta_2}$ respectively. Use the formulas you wrote for $\frac{\partial L}{\partial \theta_1}$ and $\frac{\partial L}{\partial \theta_2}$ in the previous exercise. In the functions below, the parameter `theta` is a vector that looks like (θ_1, θ_2).

Note: To keep your code a bit more concise, be aware that `np.mean` does the same thing as `np.sum` divided by the length of the numpy array.

In [38]:
```
def dt1(x, y, theta):
    """
    Compute the numerical value of the partial of l2 loss with respect to theta_1

    Keyword arguments:
    x -- the vector of all x values
    y -- the vector of all y values
    theta -- the vector of values theta
    """
    return 2*np.mean((theta[0]*x - y) * x)
    # YOUR CODE HERE
    raise NotImplementedError()
```

In [39]:
```
def dt2(x, y, theta):
    """
    Compute the numerical value of the partial of l2 loss with respect to theta_2

    Keyword arguments:
    x -- the vector of all x values
    y -- the vector of all y values
    theta -- the vector of values theta
    """
    return -2*np.mean((np.cos(theta[1]*x)*x)*(y-theta[0]*x-np.sin(theta[1]*x)))
    # YOUR CODE HERE
    raise NotImplementedError()
```

In [40]:
```
# This function calls dt1 and dt2 and returns the gradient dt. It is already implemented for you.
def dt(x, y, theta):
    """
    Returns the gradient of l2 loss with respect to vector theta

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    theta -- the vector of values theta
    """
    return np.array([dt1(x,y,theta), dt2(x,y,theta)])
```

In [41]:
```
assert np.isclose(dt1(x, y, [0, np.pi]), -25.376660670924529, rtol=0.5)
```

In [ ]:
```

```

4: Gradient Descent

We cannot try to solve for the optimal $\hat{\theta}$ exactly. We resort to an algorithm for guess-and-check to iteratively find an approximate solution. So let's try implementing gradient descent.

Question 4

Question 4a

Implement the `grad_desc` function that performs gradient descent for a finite number of iterations. This function takes in an array for $\mathbf{x}$ (`x`), an array for $\mathbf{y}$ (`y`), and an initial value for θ (`theta`). `alpha` will be the learning rate (or step size, whichever term you prefer). In this part, we'll use a static learning rate that is the same at every time step.

At each time step, use the gradient and `alpha` to update your current `theta`. Also at each time step, be sure to save the current `theta` in `theta_history`, along with the L^2 loss (computed with the current `theta`) in `loss_history`.

Hints:

- Write out the gradient update equation (1 step). What variables will you need for each gradient update? Of these variables, which ones do you already have, and which ones will you need to recompute at each time step?
- You may need a loop here to update `theta` several times
- Recall that the gradient descent update function follows the form:
$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \alpha \left(\nabla_{\boldsymbol{\theta}} \mathbf{L}(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}^{(t)}) \right)$

In [42]:
```
# Run me
def init_t():
    """Creates an initial theta [0, 0] of shape (2,) as a starting point for gradient descent"""
    return np.zeros((2,))
```

In [43]:
```
def grad_desc(x, y, theta, num_iter=20, alpha=0.1):
    """
    Run gradient descent update for a finite number of iterations and static learning rate

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    theta -- the vector of values theta to use at first iteration
    num_iter -- the max number of iterations
    alpha -- the learning rate (also called the step size)

    Return:
    theta -- the optimal value of theta after num_iter of gradient descent
    theta_history -- the series of theta values over each iteration of gradient descent
    loss_history -- the series of loss values over each iteration of gradient descent
    """
    theta_history = []
    loss_history = []

    for i in range(num_iter):
        gradient = dt(x, y, theta)
        theta = theta - alpha/(i+1)*gradient
        theta_history.append(theta)
        loss_history.append(l2_loss(sin_model(x, theta[0], theta[1]), y))

    # YOUR CODE HERE

    return theta, theta_history, loss_history
    raise NotImplementedError()
```

In [44]:
```
t = init_t()
t_est, ts, loss = grad_desc(x, y, t, num_iter=20, alpha=0.1)

assert len(ts) == len(loss) == 20 # theta history and loss history are 20 items in them
assert ts[0].shape == (2,) # theta history contains theta values
assert np.isscalar(loss[0]) # loss history is a list of scalar values, not vector
assert loss[1] - loss[-1] > 0 # loss is decreasing
```

In [ ]:
```

```

Question 4b

Now, let's try using a decaying learning rate. Implement `grad_desc_decay` below, which performs gradient descent with a learning rate that decreases slightly with each time step. You should be able to copy most of your work from the previous part, but you'll need to tweak how you update `theta` at each time step.

By decaying learning rate, we mean instead of just a number α, the learning should be now $\frac{\alpha}{i+1}$ where i is the current number of iteration. (Why do we need to add '+ 1' in the denominator?)

In [45]:
```
def grad_desc_decay(x, y, theta, num_iter=20, alpha=0.1):
    """
    Run gradient descent update for a finite number of iterations and decaying learning rate

    Keyword arguments:
    x -- the vector of values x
    y -- the vector of values y
    theta -- the vector of values theta
    num_iter -- the max number of iterations
    alpha -- the learning rate

    Return:
    theta -- the optimal value of theta after num_iter of gradient descent
    theta_history -- the series of theta values over each iteration of gradient descent
    loss_history -- the series of loss values over each iteration of gradient descent
    """
    theta_history = []
    loss_history = []

    for i in range(num_iter):
        gradient = dt(x, y, theta)
        theta_history.append(theta)
        loss_history.append(l2_loss(sin_model(x, theta[0], theta[1]), y))
        theta = theta - (alpha * (gradient))


    # YOUR CODE HERE

    return theta, theta_history, loss_history
    raise NotImplementedError()
```

In [46]:
```
t = init_t()
t_est_decay, ts_decay, loss_decay = grad_desc_decay(x, y, t, num_iter=20, alpha=0.1)

assert len(ts_decay) == len(loss_decay) == 20 # theta history and loss history are 20 items in them
assert ts_decay[0].shape == (2,) # theta history contains theta values
assert np.isscalar(loss[0]) # loss history should be a list of values, not vector
assert loss_decay[1] - loss_decay[-1] > 0 # loss is decreasing
```

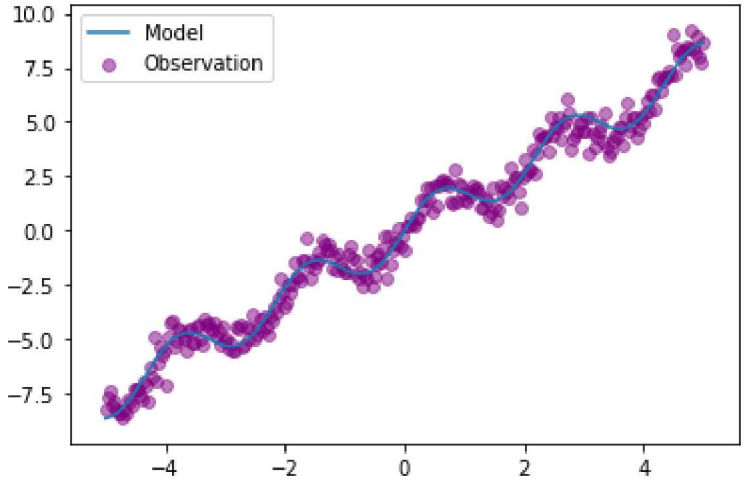
In [ ]:
```

```

Question 4c

Let's visually inspect our results of running gradient descent to optimize $\boldsymbol{\theta}$. Plot our x-values with our model's predicted y-values over the original scatter plot. Did gradient descent successfully optimize $\boldsymbol{\theta}$?

In [47]:
```
# Run me
t = init_t()
t_est, ts, loss = grad_desc(x, y, t)

t = init_t()
t_est_decay, ts_decay, loss_decay = grad_desc_decay(x, y, t)
```

In [48]:
```
y_pred = sin_model(x, t_est[0], t_est[1])

plt.plot(x, y_pred, label='Model')
plt.scatter(x, y, alpha=0.5, label='Observation', color='purple')
plt.legend();
```



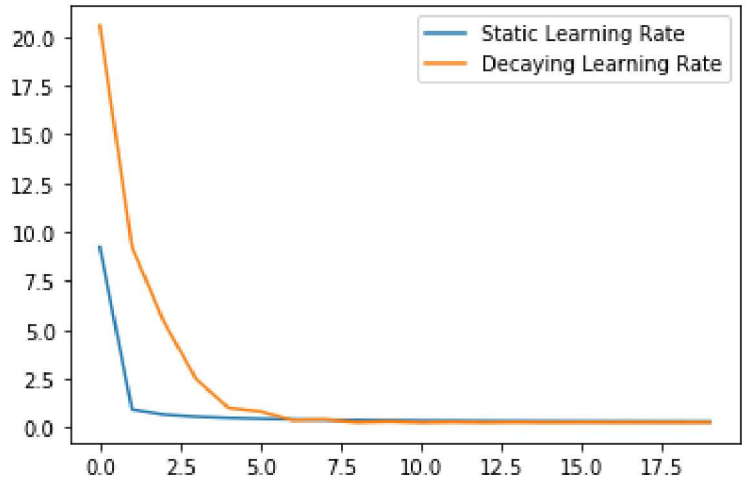
In [49]:
```
# YOUR CODE HERE
print("The gradient descent successfully optimized  θ since the actual values are close to the model's predicted y -values")
```

```
The gradient descent successfully optimized  θ since the actual values are close to the model's predicted y -values
```

Question 4d

Let's compare our two gradient descent methods and see how they differ. Plot the loss values over each iteration of gradient descent for both static learning rate and decaying learning rate.

In [50]:
```
plt.plot(loss, label='Static Learning Rate') # Plot of loss history for static learning rate
plt.plot(loss_decay, label='Decaying Learning Rate') # Plot of loss history for decaying learning rate
plt.legend();

# YOUR CODE HERE
# raise NotImplementedError()
```



Question 4e

Compare and contrast the performance of the two gradient descent methods. Which method begins to converge more quickly?

In [54]:
```
# YOUR CODE HERE
'''If we were to compare and contrast the performance of the two gradient descent methods, we would find that for both methods
the num_iter=20 and alpha=0.1. For the Static learning rate, at the lowest x value, the y value is around 9.5, whereas for the
Decaying Learning rate, you have the y value be 21 at the lowest x. So, the Decaying learning rate gradient descent method
seems to more slowly have lower loss history, whereas the static learning method more quickly lowers its loss history, possibly
due to the static learning rate.

The method that begins to converge more quickly is the static learning method, since it has a static learning rate every iteration, as opposed to the decaying learning rate which takes longer to result in a lower loss history.

'''
raise NotImplementedError()
```

Out[54]:
```
'If we were to compare and contrast the performance of the two gradient descent methods, we would find that for both methods \nthe num_iter=20 and alpha=0.1. For the Static learning rate, at the lowest x value, the y value is around 9.5, whereas for the \nDecaying Learning rate, you have the y value be 21 at the lowest x. So, the Decaying learning rate gradient descent method \nseems to more slowly have lower loss history, whereas the static learning method more quickly lowers its loss history, possibly \ndue to the static learning rate. \n\nThe method that begins to converge more quickly is the static learning method, since it has a static learning rate every iteration, as opposed to the decaying learning rate which takes longer to result in a lower loss history. \n\n'
```

5: Visualizing Loss

Question 5:

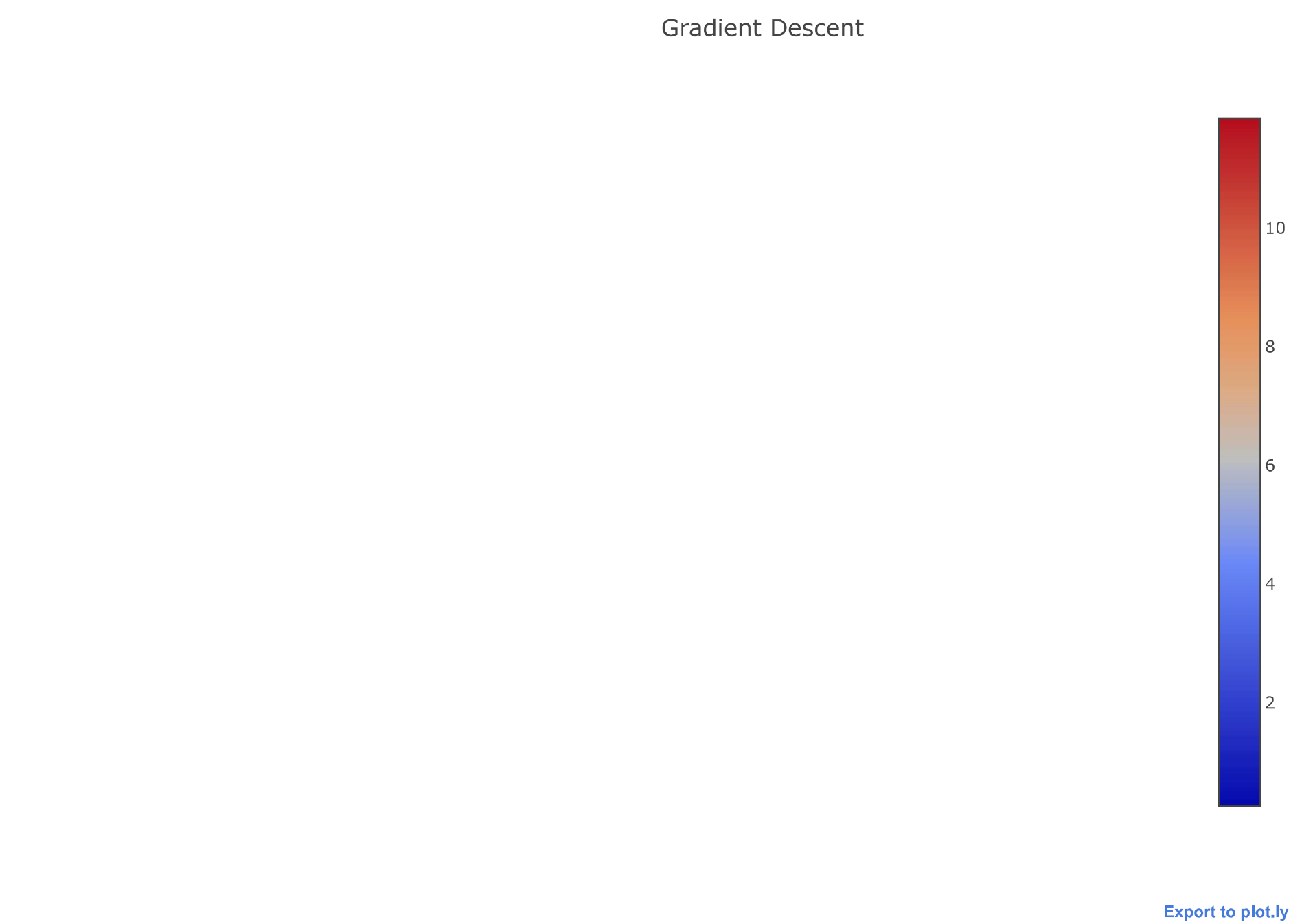
Let's visualize our loss functions and gain some insight as to how gradient descent and stochastic gradient descent are optimizing our model parameters.

Question 5a:

In the previous plot is about the loss decrease over time, but what exactly is path the theta value? Run the following three cells.

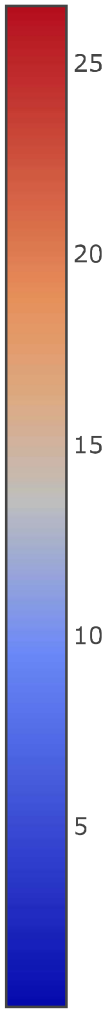
In [55]:
```
# Run me
ts = np.array(ts).squeeze()
ts_decay = np.array(ts_decay).squeeze()
loss = np.array(loss)
loss_decay = np.array(loss_decay)
```

In [56]:
```
# Run me to see a 3D plot (gradient descent with static alpha)
plot_3d(ts[:, 0], ts[:, 1], loss, l2_loss, sin_model, x, y)
```


Gradient Descent
10
8
6
4
2
Export to plot.ly »

In [57]:
```
# Run me to see another 3D plot (gradient descent with decaying alpha)
plot_3d(ts_decay[:, 0], ts_decay[:, 1], loss_decay, l2_loss, sin_model, x, y)
```

Gradient Descent



Export to plot.ly »

In the following cell, write 1-2 sentences about the differences between using a static learning rate and a learning rate with decay for gradient descent. Use the loss history plot as well as the two 3D visualization to support your answer.

The two gradient descent methods differs in ...

In [58]:
```
# YOUR CODE HERE
'''The two gradient descent methods of using a static learning rate and a learning rate with decay for gradie
nt descent differ
based on how the static learning rate curve coverges faster due to alpha maintaining its value, meaning it ha
s a lower loss
history than the decaying learning rate curve. In terms of the two 3D visualization representations, for the
 static learning
rate model, it seems that due to static alpha, it takes less time for the loss to decrease, based on the colo
r scheme of red
to immediately dark blue, whereas, for the decaying alpha learning curve, it seems to take longer for the los
s to decrease,
based on the color scheme of red to light blue to lighter blue to dark blue, however, with a decaying learnin
g rate trend,
you can decrease your size so you don't overshoot your minimum.
'''


raise NotImplementedError()
```

Out[58]:
```
"The two gradient descent methods of using a static learning rate and a learning rate with decay for gradient
descent differ \nbased on how the static learning rate curve coverges faster due to alpha maintaining its val
ue, meaning it has a lower loss \nhistory than the decaying learning rate curve. In terms of the two 3D visua
lization representations, for the static learning \nrate model, it seems that due to static alpha, it takes l
ess time for the loss to decrease, based on the color scheme of red \nto immediately dark blue, whereas, for
the decaying alpha learning curve, it seems to take longer for the loss to decrease, \nbased on the color sch
eme of red to light blue to lighter blue to dark blue, however, with a decaying learning rate trend, \nyou ca
n decrease your size so you don't overshoot your minimum.\n"
```

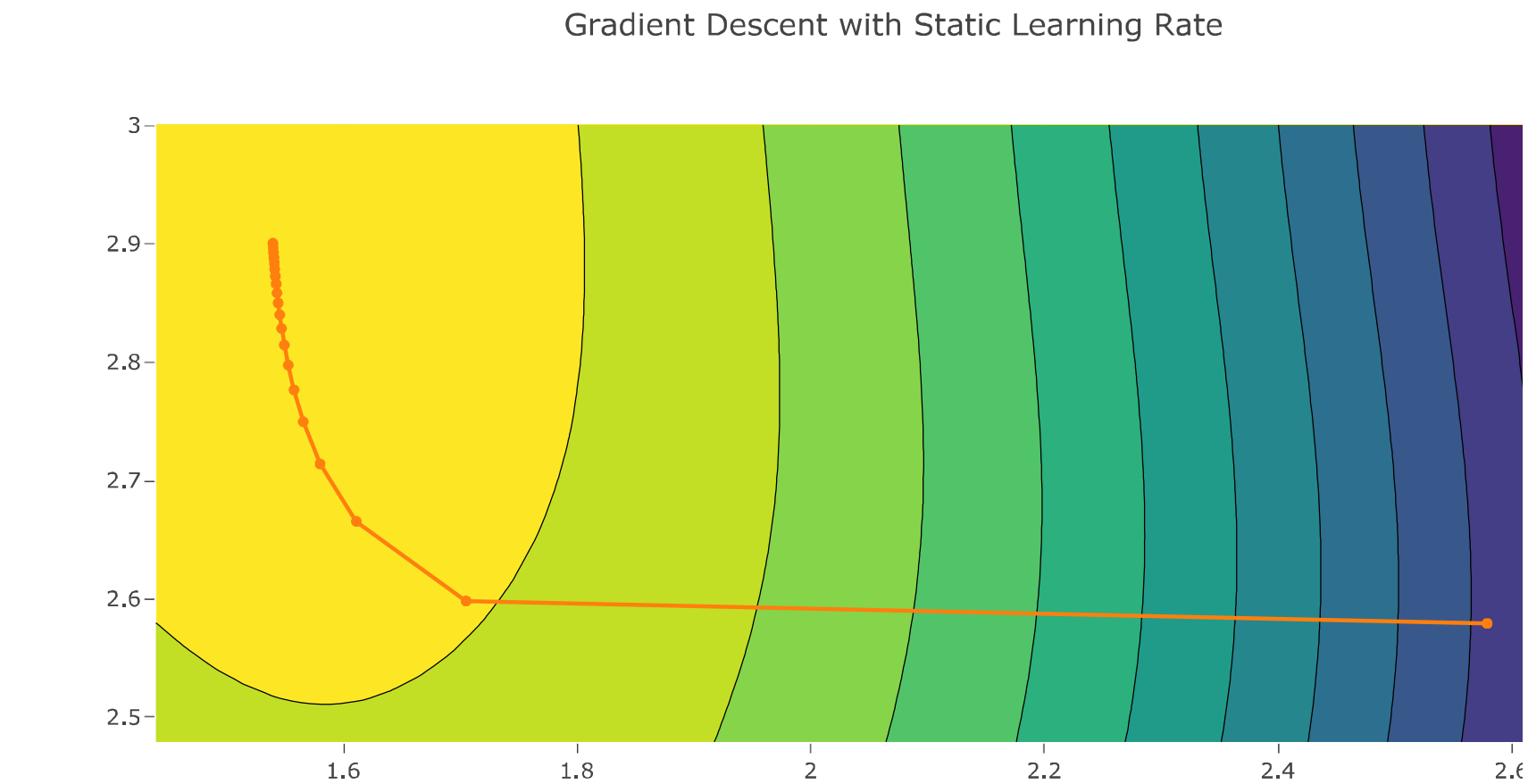
Question 5b:

Another common way of visualizing 3D dynamics is with a *contour* plot. Please run the following cell.

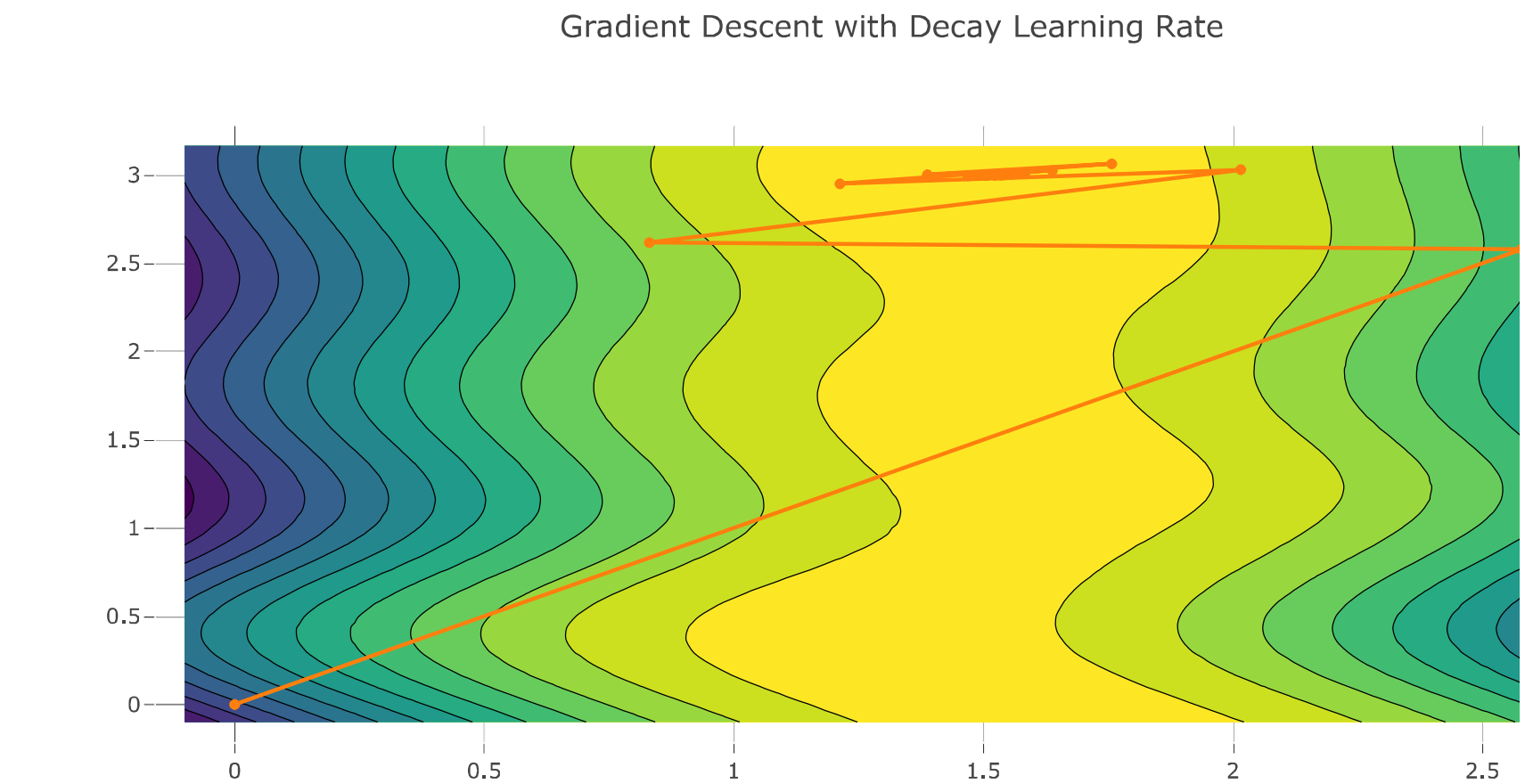
In [59]:
```
# YOUR CODE HERE
def contour_plot(title, theta_history, loss_function, model, x, y):
    """
    The function takes the following as argument:
        theta_history: a (N, 2) array of theta history
        loss: a list or array of loss value
        loss_function: for example, l2_loss
        model: for example, sin_model
        x: the original x input
        y: the original y output
    """
    theta_1_series = theta_history[:,0] # a list or array of theta_1 value
    theta_2_series = theta_history[:,1] # a list or array of theta_2 value

    # Create trace of theta point
    # Uncomment the following lines and fill in the TODOS
#     thata_points = go.Scatter(name="Theta Values",
#                               x=..., #TODO
#                               y=..., #TODO
#                               mode="lines+markers")

    ## In the following block of code, we generate the z value
    ## across a 2D grid
    t1_s = np.linspace(np.min(theta_1_series) - 0.1, np.max(theta_1_series) + 0.1)
    t2_s = np.linspace(np.min(theta_2_series) - 0.1, np.max(theta_2_series) + 0.1)

    x_s, y_s = np.meshgrid(t1_s, t2_s)
    data = np.stack([x_s.flatten(), y_s.flatten()]).T
    ls = []
    for t1, t2 in data:
        l = loss_function(model(x, t1, t2), y)
        ls.append(l)
    z = np.array(ls).reshape(50, 50)

    # Create the contour
    # Uncomment the following lines and fill in the TODOS
#     lr_loss_contours = go.Contour(x=..., #TODO
#                                   y=..., #TODO
#                                   z=..., #TODO
#                                   colorscale='Viridis', reversescale=True)
```

```
    thata_points = go.Scatter(name="Theta Values",
                              x=theta_1_series,
                              y=theta_2_series,
                              mode="lines+markers")
    lr_loss_contours = go.Contour(x=t1_s,
                                  y=t2_s,
                                  z=z,
                                  colorscale='Viridis', reversescale=True)

    plotly.offline.iplot(go.Figure(data=[lr_loss_contours, thata_points], layout={'title': title}))

raise NotImplementedError()
```

In [60]:
```
# Run this
contour_plot('Gradient Descent with Static Learning Rate', ts, l2_loss, sin_model, x, y)
```



In [61]:
```
## Run me
contour_plot('Gradient Descent with Decay Learning Rate', ts_decay, l2_loss, sin_model, x, y)
```



In the following cells, write down the answer to the following questions:

- How do you interpret the two contour plots?
- Compare contour plot and 3D plot, what are the pros and cons of each?

From reading these two contour plots, I can see that...

In [63]:
```
# YOUR CODE HERE
''' A contour plot is an isoline of a function of two variables creating a curve along which the function has
a constant value.
From reading these two contour plots, I can interpret the contour plot representing the Gradient Descent with
Static Learning
Rate as the sine model on the x values compared to the y-values. In terms of the gradient descent, there seem
s to be a sharp
decline in the y-value, and an immediate shift from the dark blue color to the yellow color, meaning that the
re is lower l2
loss (and faster learning to reduce loss) in terms of a increase in theta, but decrease in x. While the color
scheme/legend is
the same as the Gradient Descent with Decaying Learning Rate contour plot, the wave frequency seems to be les
s varied, and the
amplitude seems lower. Also, the waves for both are uniform and nearly symmetric for each wave color, except
 that the darker
colors in both seem to have thinner waves. Regarding the Gradient Descent with Decaying Learning Rate contour
plot, there seems
to be an erratic transition between different wave colors, from dark blue to bluish green to dark green to li
ghtish green to
yellow. This means that it takes longer to reduce the loss, and longer to learn how to reduce the loss.

Both the contour plots and the 3D plots both show how the gradient descent and stochastic gradient descent ar
e optimizing our
model parameters, and show the loss decrease over time as well as the path of the theta value. Lastly, both a
llow you to
interact with the plot, testing the values of x and y, and in some cases, theta. However, neither of these pl
ots seems to
determine the phase and amplitude for the four parameter model. The pros of the contour plots are that you ca
n see a more
clear transition from higher loss (represented in darker colors) to less loss (represented by lighter color
s), which shows how
using the Gradient Descent with Learning type Rate methods can help reduce loss, and demostrate the speed of
 loss reduction
learning. The Cons of contour plots are that the contour plots don't seem to always have the same wave freque
ncies and
amplitude for each colors, and some contour plots have only sided color changes, while others show color chan
ges on both sides.
Lastly, the contour plots vary in their x and y value ranges, some starting at x = 0, and others at x=1.5, wh
ich doesn't show
the progression of x and theta on the effect of y. The pros of the 3D plots are that you can easily show the
 change in the
color (representing the loss), so, and see the effect of x on y, as well as on z (if there was one). For 3D p
```

```
lots, you can test
the effect of three variables on each other. The 3D plot adds a little bit of a buffer to each edge, meaning
 that it will be
easier to estimate the loss amount based on the color hue/shade. Lastly, depending on the concavity of the 3D
plot, you can
tell how quickly the loss reduction can be. For example, with the stochastic gradient descent method there is
more of a folding
of the plot, since it takes less time for the loss to be reduced. This explains why the method that begins to
converge more
quickly is the static learning method, since it has a static learning rate every iteration, as opposed to the
decaying learning
rate which takes longer to result in a lower loss history.

The cons are that the 3D plots don't seem to have an indicator of what the theta values are. The dimensions c
ould also be very
skewed and warped, such that it is hard to follow how the color changes and the model itself. Speaking of, th
e color hue
changes are too subtle, so both of the visually representations of the static gradient descent and stochastic
gradient descent
methods seem too similar, which masks the slower color change of the stochastic (decaying) gradient descent m
ethod that is
shown in the contour plots.


'''
raise NotImplementedError()
```

Out[63]: " A contour plot is an isoline of a function of two variables creating a curve along which the function has a constant value.\n\nFrom reading these two contour plots, I can interpret the contour plot representing the Gradient Descent with Static Learning \nRate as the sine model on the x values compared to the y-values. In terms of the gradient descent, there seems to be a sharp \ndecline in the y-value, and an immediate shift from the dark blue color to the yellow color, meaning that there is lower l2 \nloss (and faster learning to reduce loss) in terms of a increase in theta, but decrease in x. While the color scheme/legend is \nthe same as the Gradient Descent with Decaying Learning Rate contour plot, the wave frequency seems to be less varied, and the \namplitude seems lower. Also, the waves for both are uniform and nearly symmetric for each wave color, except that the darker \ncolors in both seem to have thinner waves. Regarding the Gradient Descent with Decaying Learning Rate contour plot, there seems\nto be an erratic transition between different wave colors, from dark blue to bluish green to dark green to lightish green to \nyellow. This means that it takes longer to reduce the loss, and longer to learn how to reduce the loss.\n\nBoth the contour plots and the 3D plots both show how the gradient descent and stochastic gradient descent are optimizing our \nmodel parameters, and show the loss decrease over time as well as the path of the theta value. Lastly, both allow you to \ninteract with the plot, testing the values of x and y, and in some cases, theta. However, neither of these plots seems to \ndetermine the phase and amplitude for the four parameter model. The pros of the contour plots are that you can see a more \nclear transition from higher loss (represented in darker colors) to less loss (represented by lighter colors), which shows how \nusing the Gradient Descent with Learning type Rate methods can help reduce loss, and demostrate the speed of loss reduction \nlearning. The Cons of contour plots are that the contour plots don't seem to always have the same wave frequencies and \namplitude for each colors, and some contour plots have only sided color changes, while others show color changes on both sides.\nLastly, the contour plots vary in their x and y value ranges, some starting at x = 0, and others at x=1.5, which doesn't show \nthe progression of x and theta on the effect of y. The pros of the 3D plots are that you can easily show the change in the \ncolor (representing the loss), so, and see the effect of x on y, as well as on z (if there was one). For 3D plots, you can test\nthe effect of three variables on each other. The 3D plot adds a little bit of a buffer to each edge, meaning that it will be \neasier to estimate the loss amount based on the color hue/shade. Lastly, depending on the concavity of the 3D plot, you can \ntell how quickly the loss reduction can be. For example, with the stochastic gradient descent method there is more of a folding \nof the plot, since it takes less time for the loss to be reduced. This explains why the method that begins to converge more \nquickly is the static learning method, since it has a static learning rate every iteration, as opposed to the decaying learning \nrate which takes longer to result in a lower loss history.\n\nThe cons are that the 3D plots don't seem to have an indicator of what the theta values are. The dimensions could also be very \nskewed and warped, such that it is hard to follow how the color changes and the model itself. Speaking of, the color hue \nchanges are too subtle, so both of the visually representations of the static gradient descent and stochastic gradient descent \nmethods seem too similar, which masks the slower color change of the stochastic (decaying) gradient descent method that is \nshown in the contour plots.\n\n\n"

Question 5c

Try adding the two additional model parameters for phase and amplitude that we ignored (see 3a). What are the optimal phase and amplitude values for your four parameter model? Do you get a better loss?

I think...

In [64]:
```
# YOUR CODE HERE
''' We represented the original data in a linear function and a sinusoidal function and found a new model to
 address this
discovery and find optimal parameters to best fit the data: fθ(x)=θ1x+sin(θ2x) where the model is parameteriz
ed by both θ1 and
θ2 , or composed together, θ . In our sine function asin(bx+c) we have three parameters: amplitude scaling pa
rameter a ,
frequency parameter b and phase shifting parameter c . We originally set the scaling and phase shifting param
eter ( a and c in
this case) to 1 and 0 respectively. As stated: "The coefficient b and the period of the sine curve have an in
verse relationship,
so as b gets smaller, the length of one cycle of the curve gets bigger. Likewise, as you increase b, the peri
od will decrease.
What is the phase shift of a sine curve? The phase shift of a sine curve is how much the curve shifts from ze
ro."

I think if you add the two additional model parameters for phase (c) and amplitude (a) that we ignored, the a
mplitude would
certainly increase or decrease, meaning the gradient descent to appear more "steep" depending on the value. T
he phase shift
would just mean the the curve shifts more from zero.

The optimal phase and amplitude values for the four parameter model would be the same as the original gradien
t descent graph,
but the graph would be slightly elevated and the gradient descent graph is slightly steeper.

Do you get a better loss? yes, because if you make the gradient descent curve "steeper" and shift the graph o
ver, you will
make the same trend, but the loss starts at 14, and plateus at 6, whereas in the original static learning lea
rning alpha
gradient descent curve starts at 9 and plateaus at 1. So, this essentially gets you less loss, which means th
at there is a
better loss.


'''
raise NotImplementedError()
```

Out[64]: ' We represented the original data in a linear function and a sinusoidal function and found a new model to address this \ndiscovery and find optimal parameters to best fit the data: $f\boldsymbol{\theta}(x)=\theta 1x+sin(\theta 2x)$ where the model is parameterized by both $\theta 1$ and \n$\theta 2$, or composed together, $\boldsymbol{\theta}$. In our sine function $a\sin(bx+c)$ we have three parameters: amplitude scaling parameter a , \nfrequency parameter b and phase shifting parameter c . We originally set the scaling and phase shifting parameter (a and c in \nthis case) to 1 and 0 respectively. As stated: "The coefficient b and the period of the sine curve have an inverse relationship,\nso as b gets smaller, the length of one cycle of the curve gets bigger. Likewise, as you increase b, the period will decrease. \nWhat is the phase shift of a sine curve? The phase shift of a sine curve is how much the curve shifts from zero."\n\nI think if you add the two additional model parameters for phase (c) and amplitude (a) that we ignored, the amplitude would \ncertainly increase or decrease, meaning the gradient descent to appear more "steep" depending on the value. The phase shift \nwould just mean the the curve shifts more from zero.\n\nThe optimal phase and amplitude values for the four parameter model would be the same as the original gradient descent graph, \nbut the graph would be slightly elevated and the gradient descent graph is slightly steeper.\n\nDo you get a better loss? yes, because if you make the gradient descent curve "steeper" and shift the graph over, you will \nmake the same trend, but the loss starts at 14, and plateus at 6, whereas in the original static learning learning alpha \ngradient descent curve starts at 9 and plateaus at 1. So, this essentially gets you less loss, which means that there is a \nbetter loss.\n\n\n'

Question 5d (optional)

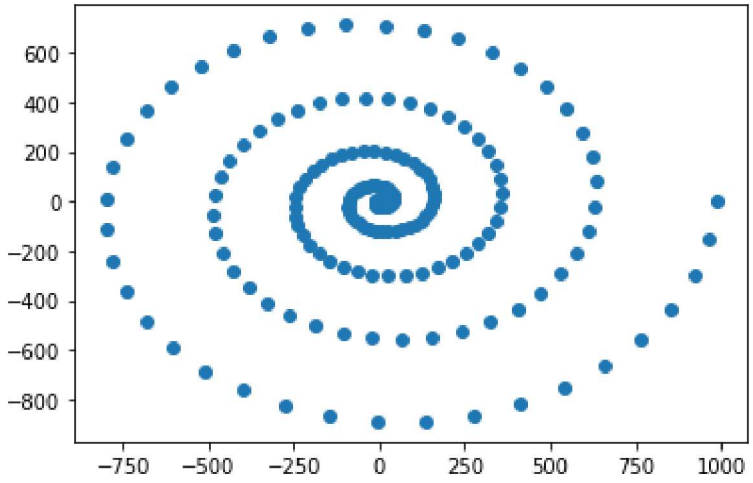
It looks like our basic two parameter model, a combination of a linear function and sinusoidal function, was able to almost perfectly fit our data. It turns out that many real world scenarios come from relatively simple models.

At the same time, the real world can be incredibly complex and a simple model wouldn't work so well. Consider the example below; it is neither linear, nor sinusoidal, nor quadratic.

Optional: Suggest how we could iteratively create a model to fit this data and how we might improve our results.

Extra optional: Try and build a model that fits this data.

In [65]:
```
x = []
y = []
for t in np.linspace(0,10*np.pi, 200):
    r = ((t)**2)
    x.append(r*np.cos(t))
    y.append(r*np.sin(t))

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



With our two parameter model, a combination of a linear function and sinusoidal function, was able to almost perfectly fit our data, we can suggest how to iteratively create a model to fit this data and how we might improve our results.

I suggest the following model of exponential decay which will improve the results by showing more of a exponential decay or growth of the data.