Problem 1

a. LLS regression
    i. The design matrix X is 11 x 3
    ii. The parameter vector w is 3 x 1
    iii. The product X'X is 3 x 3
b. For 1000 point dataset
    i. The design matrix is 1000 x 3
    ii. The parameter vector w is 3 x 1
    iii. The size of the product X'X is 3 x 3
c. False: The model complexity is determined by the number of model parameters.
    True: The number of data points used for training would help in increasing the model accuracy.

Problem 2

Since the function slope changes twice, a 3rd order polynomial ($w_1x_1^3 + w_2x_1^2 + w_3x_1 + w_4$) model would be able to describe these points quite well.

Problem 3

$$x_{new} = x_{old} + \eta \cdot \frac{\partial \text{obj}}{\partial x}$$

Since we want to reach the top of the mountain, we must follow the path that leads us up the slope of the mountain. Therefore, our next step should be in the direction of positive gradient/slope. Therefore, the formula attached above should lead us up the slope.

Problem 4

1. Increasing the batch size does not have any direct effect on the oscillations while minimizing the objective function.
2. Switching to stochastic gradient descent can help partially as it picks random samples and this might cause the gradient to smoothen out over time.
3. Yes, reducing the learning rate can help in reducing the oscillations as it prevents the update from overshooting.

Problem 5

On reducing the L1_ratio term to 0, the formulant becomes equivalent to an LLS with L2 regularization problem. The ($1 - L1_{ratio}$) coefficient becomes 1, and the associated norm is $||w||_2^2$, which is an L2 norm.

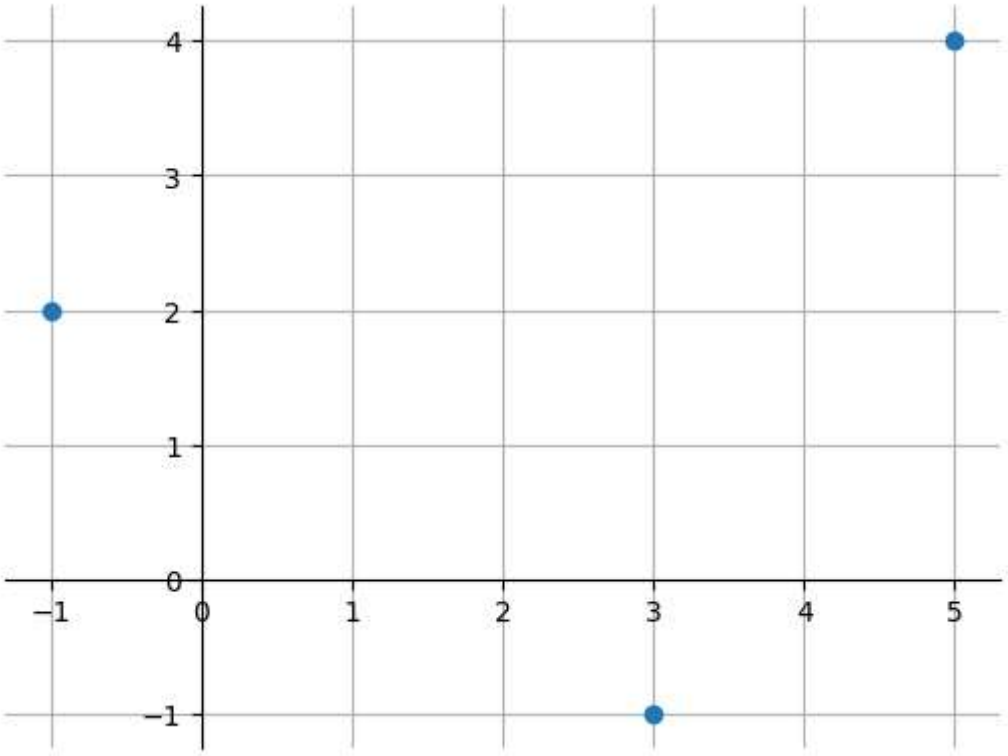
M2-L1 Problem 1 (5 points)

In this question you will perform linear least squares regression on a very small dataset of 3 points. First, load and plot the data by running the following cell.

The variables provided are:

- x: 3x1 input data
- y: 3x1 output data

In [8]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

x = np.array([[-1, 3, 5]]).T
y = np.array([[2, -1, 4]]).T

fig, ax = plt.subplots()
plt.plot(x, y,'o')
ax.spines['left'].set_position(('data', 0))
ax.spines['bottom'].set_position(('data', 0))
sns.despine()
plt.grid()
plt.show()
```



Construct a design matrix

For 1-D linear regression, the design matrix must contain not only a column of input x-values, but also a 'bias column' -- a column of ones (to allow the regression line to have an intercept).

The next step is to construct the design matrix `X` by concatenating a column of ones to the given input `x`. This has been done for you below:

In [9]:
```
bias = np.ones_like(x)

X = np.concatenate([x,bias],1)

print("Design Matrix:\n",X)
```

```
Design Matrix:
 [[-1  1]
 [ 3  1]
 [ 5  1]]
```

Solving for regression coefficients

Now that we have the design matrix X and the output y, we can solve for the coefficients w such that $Xw \approx y$ using:

$$w = (X' X)^{-1} X' y$$

Note that you can use the following in Python:

- `@` for matrix multiplication
- `np.linalg.inv(A)` for inversion of matrix `A`
- `A.T` for transpose of a matrix `A`
- `b.reshape(-1,1)` to treat 1D array `b` as a column (you will need to do this for `y`)

Your line's slope should be ≈ 0.18 and your y-intercept should be ≈ 1.25.

In [10]:
```
# YOUR CODE GOES HERE
# Get coefficients w
w = np.linalg.inv(X.T @ X) @ X.T @ y

print("Linear Coefficients:\n", w)
```

```
Linear Coefficients:
 [[0.17857143]
 [1.25      ]]
```

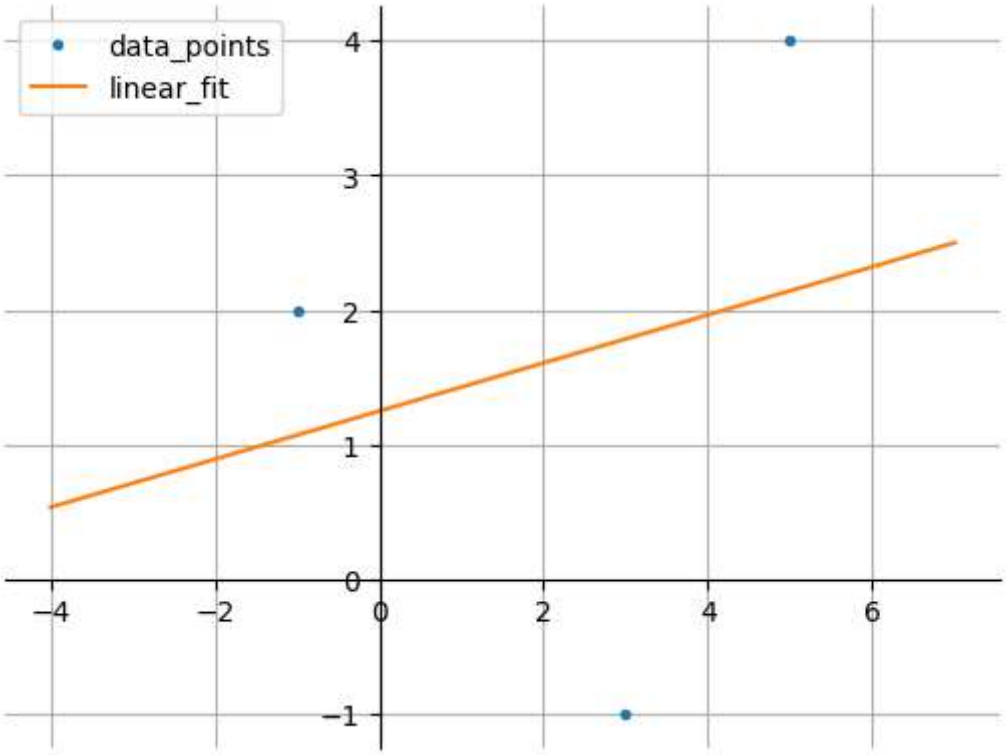
Making predictions

Now that we have the coefficients, we can make predictions on new data with the model.

Do the following steps:

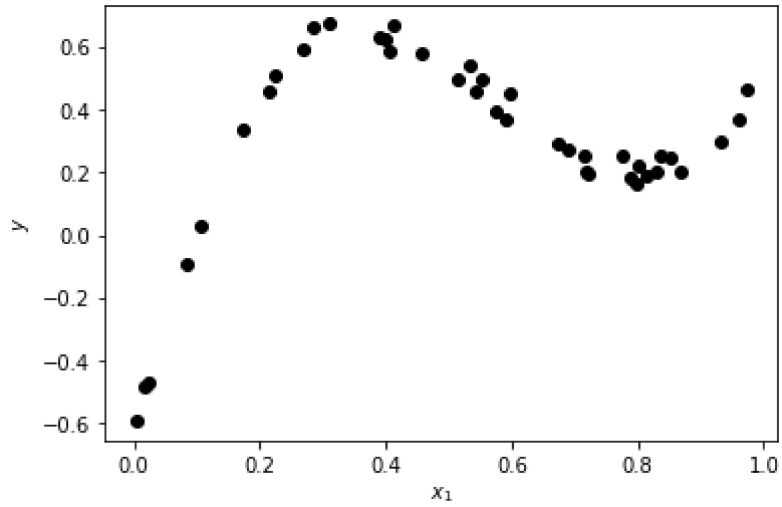
- [Given] Sample 40 points on the interval [-3,7], such as by using `np.linspace()` (Append `.reshape(-1,1)` to convert to a column)
- [Given] Create a design matrix by adding a column of ones as done previously
- Make a prediction by multiplying your new design matrix by `w`. You can do matrix multiplication with the `@` symbol
- [Given] Add a line to the plot showing these predictions

In [14]:
```
n = 40
x_test = np.linspace(-4,7,n).reshape(-1,1)
bias_test = np.ones_like(x_test)
X_test = np.concatenate([x_test, bias_test], 1)

# YOUR CODE GOES HERE
# Predict y_test
y_test = X_test @ w

fig, ax = plt.subplots()
plt.plot(x, y, '.')
plt.plot(x_test, y_test)
ax.spines['left'].set_position(('data', 0))
ax.spines['bottom'].set_position(('data', 0))
plt.legend(['data_points', 'linear_fit'])
sns.despine()
plt.grid()
plt.show()
```



M2-L1 Problem 2 (5 points)

Consider the following data:



We will perform these steps:

1. Load the data
2. Generate a design matrix
3. Solve for the regression coefficients
4. Create and plot a curve using these coefficients

First, we demonstrate the above for a linear (1st order) model. First, run each cell in the demo and follow along with each step. Your job is then to make 2nd and 3rd order models for the same data using similar techniques.

Demonstration: Linear fit

First, we load the data in to x and y:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([0.2855, 0.0033, 0.8307, 0.9606, 0.8153, 0.5539, 0.5152, 0.7761,
       0.5763, 0.2697, 0.6744, 0.7998, 0.1052, 0.8674, 0.598 , 0.3985,
       0.0171, 0.1732, 0.7976, 0.4137, 0.7161, 0.7225, 0.3892, 0.0834,
       0.9733, 0.3097, 0.8509, 0.0226, 0.6901, 0.2235, 0.5914, 0.5436,
       0.7189, 0.4558, 0.8366, 0.534 , 0.214 , 0.9314, 0.4065, 0.788 ])

y = np.array([ 0.6603, -0.5925,  0.2045,  0.3698,  0.191 ,  0.496 ,  0.4986,
        0.2516,  0.3905,  0.5932,  0.2924,  0.219 ,  0.0287,  0.2024,
        0.4489,  0.6237, -0.4857,  0.3384,  0.162 ,  0.6694,  0.2539,
        0.1936,  0.6322, -0.0953,  0.4632,  0.6721,  0.2464, -0.4672,
```

```
        0.2746,  0.5087,  0.3691,  0.4559,  0.2021,  0.5797,  0.2531,
        0.5417,  0.4577,  0.2952,  0.5856,  0.1818])
```

Now we generate a linear design matrix for the data:

In [2]:
```
# Function to get a linear design matrix for 1D data
def get_linear_design_matrix(x):
    x = x.reshape(-1, 1)                # Turn x into a column array
    columns = [x, np.ones_like(x)]      # Linear design matrix has a column of x an
    X = np.concatenate(columns, axis=1) # Combine each column horizontally to make
    return X
```

In [3]:
```
X = get_linear_design_matrix(x)
print("First four rows of X:")
print(X[:4,:])
```

```
First four rows of X:
[[0.2855 1.    ]
 [0.0033 1.    ]
 [0.8307 1.    ]
 [0.9606 1.    ]]
```

Now that we have the design matrix X and the output y, we can solve for the coefficients w such that $Xw \approx y$ using:

$$w = (X' X)^{-1} X' y$$

Note the use of the following in Python:

- `@` for matrix multiplication
- `np.inv(A)` for inversion of matrix `A`
- `A.T` for transpose of a matrix `A`
- `b.reshape(-1,1)` to treat 1D array `b` as a column

In [4]:
```
# Get coefficients
w = np.linalg.inv(X.T @ X) @ X.T @ y.reshape(-1,1)
print("Linear Coefficients:", w.flatten())
```

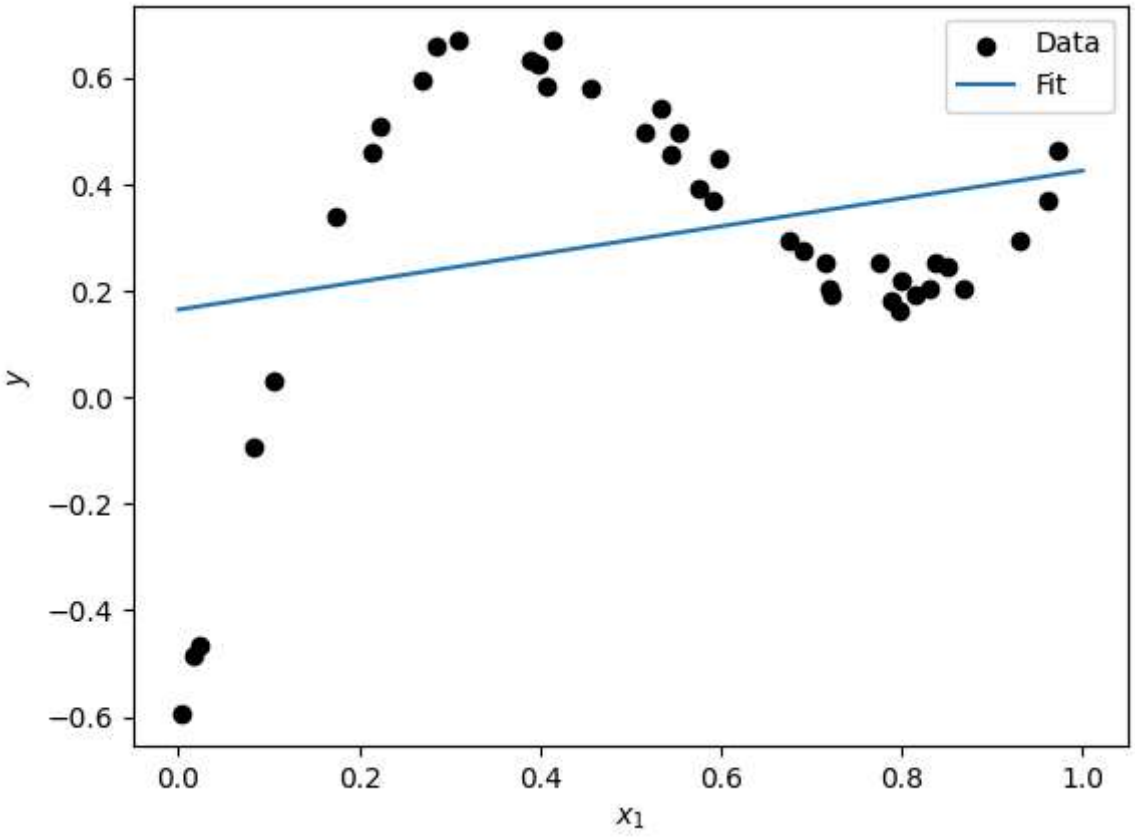
```
Linear Coefficients: [0.26080739 0.16441038]
```

Next, we write a plotting function to plot data and our least squares regression on the same axes. To apply our regression we can call the `get_linear_design_matrix()` function on a new set of x values and multiply the result by our coefficients `w`.

In [7]:
```
def plot_data_with_regression(x_data, y_data, x_reg, y_reg):
    plt.figure()

    plt.scatter(x_data, y_data, label="Data", c="black")
    plt.plot(x_reg, y_reg, label="Fit")

    plt.legend()
    plt.xlabel(r"$x_1$")
```

```
    plt.ylabel(r"$y$")
    plt.show()
```

In [8]:
```
x_fit = np.linspace(0, 1, 100)                # x values for fit line
y_fit = get_linear_design_matrix(x_fit) @ w   # y values for fit line

plot_data_with_regression(x, y, x_fit, y_fit)
```



Your turn: Second order polynomial

Now you will solve the same problem, but with a quadratic polynomial instead of linear. We need to write a new function to generate a design matrix.

Replace the commented code below:

In [9]:
```
def get_quadratic_design_matrix(x):
    # YOUR CODE GOES HERE
    # GENERATE A DESIGN MATRIX WITH 2ND ORDER FEATURES: X
    x = x.reshape(-1, 1)                # Turn x into a column array
    columns = [x ** 2, x, np.ones_like(x)]      # Linear design matrix has a column
    X = np.concatenate(columns, axis=1) # Combine each column horizontally to make
    return X
```

In [10]:
```
X = get_quadratic_design_matrix(x)
print("First four rows of X:")
print(X[:4,:])
```

```
First four rows of X:
[[8.1510250e-02 2.8550000e-01 1.0000000e+00]
 [1.0890000e-05 3.3000000e-03 1.0000000e+00]
 [6.9006249e-01 8.3070000e-01 1.0000000e+00]
 [9.2275236e-01 9.6060000e-01 1.0000000e+00]]
```

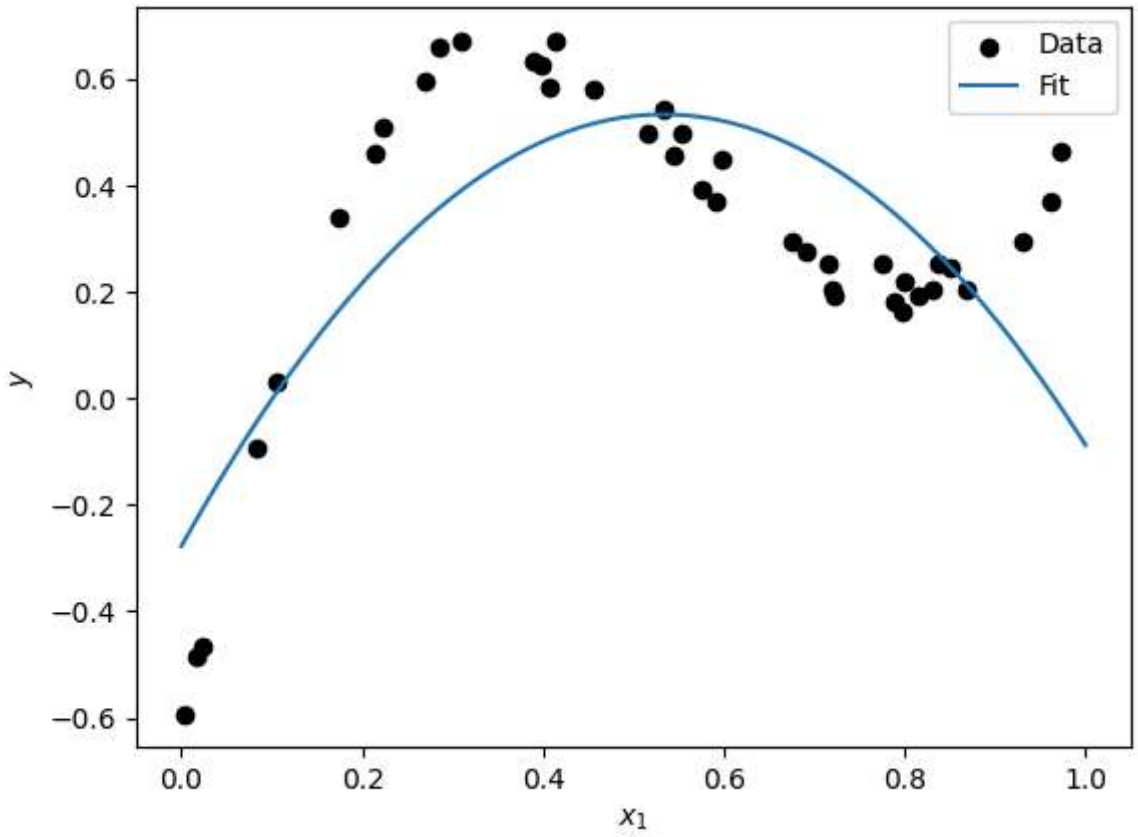
Compute the `w` coefficients with a pseudo-inverse as in the example:

In [11]:
```python
# YOUR CODE GOES HERE
# COMPUTE COEFFICIENTS w
w = np.linalg.inv(X.T @ X) @ X.T @ y.reshape(-1,1)

print("Quadratic Coefficients:", w.flatten())
```

```
Quadratic Coefficients: [-2.84999306  3.03972629 -0.27780147]
```

Now plot by generating a regression curve and calling the `plot_data_with_regression()` function:

In [12]:
```python
# YOUR CODE GOES HERE
# PLOT
x_fit = np.linspace(0, 1, 100)                # x values for fit line
y_fit = get_quadratic_design_matrix(x_fit) @ w   # y values for fit line

plot_data_with_regression(x, y, x_fit, y_fit)
```



3rd Order Polynomial

Here we go through the same steps as above, but this time we perform a cubic polynomial fit. Once again, fill in the necessary code:

In [13]:
```
def get_cubic_design_matrix(x):
    # YOUR CODE GOES HERE
    # GENERATE A DESIGN MATRIX WITH 3RD ORDER FEATURES: X
    x = x.reshape(-1, 1)                # Turn x into a column array
    columns = [x ** 3, x ** 2, x, np.ones_like(x)]      # Linear design matrix has
    X = np.concatenate(columns, axis=1) # Combine each column horizontally to make
    return X
```

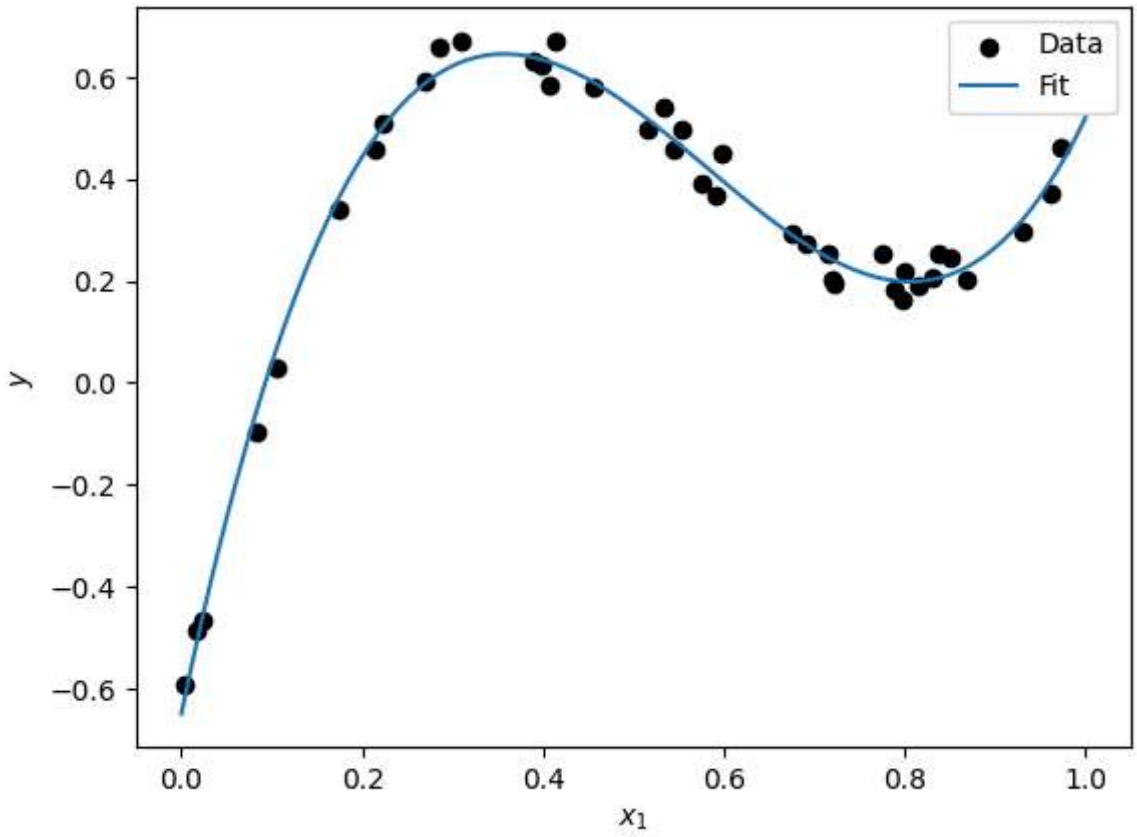
In [14]:
```
X = get_cubic_design_matrix(x)
print("First four rows of X:")
print(X[:4,:])
```

```
First four rows of X:
[[2.32711764e-02 8.15102500e-02 2.85500000e-01 1.00000000e+00]
 [3.59370000e-08 1.08900000e-05 3.30000000e-03 1.00000000e+00]
 [5.73234910e-01 6.90062490e-01 8.30700000e-01 1.00000000e+00]
 [8.86395917e-01 9.22752360e-01 9.60600000e-01 1.00000000e+00]]
```

In [15]:
```
# YOUR CODE GOES HERE
# COMPUTE COEFFICIENTS w
w = np.linalg.inv(X.T @ X) @ X.T @ y.reshape(-1,1)
print("Cubic Coefficients:", w.flatten())
```

```
Cubic Coefficients: [  9.86043282 -17.20179622   8.51092348  -0.64960983]
```

In [14]:
```
# YOUR CODE GOES HERE
# PLOT
x_fit = np.linspace(0, 1, 100)                # x values for fit line
y_fit = get_cubic_design_matrix(x_fit) @ w   # y values for fit line

plot_data_with_regression(x, y, x_fit, y_fit)
```


Data
Fit
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
y
0.0
0.2
0.4
0.6
0.8
1.0
x_1

M2-L1 Problem 3 (5 points)

In this question you will perform regression on 2D data. A linear fit will be demonstrated, and afterward you will extend the code to perform a second-order fit. First, run the Setup cells.

Setup

Generating data

In [1]:
```
# Generating data for the problem
import numpy as np
import matplotlib.pyplot as plt

def gaussian2d(A, mx, my, sx, sy):
    F = lambda xy: A*np.exp(-((xy[:,0]-mx)**2/(2*sx*sx)
                            + (xy[:,1]-my)**2/(2*sy*sy)))
    return F

def get_data_function():
    f1 = gaussian2d(A=0.7, mx = 0.25,my=0.25,sx=0.25,sy=0.25)
    f2 = gaussian2d(A=0.7, mx = 0.75,my=0.75,sx=0.25,sy=0.45)
    f = lambda xy: f1(xy) + f2(xy)
    return f

np.random.seed(0)
x = np.random.rand(60,2)
f = get_data_function()
y = f(x)
```

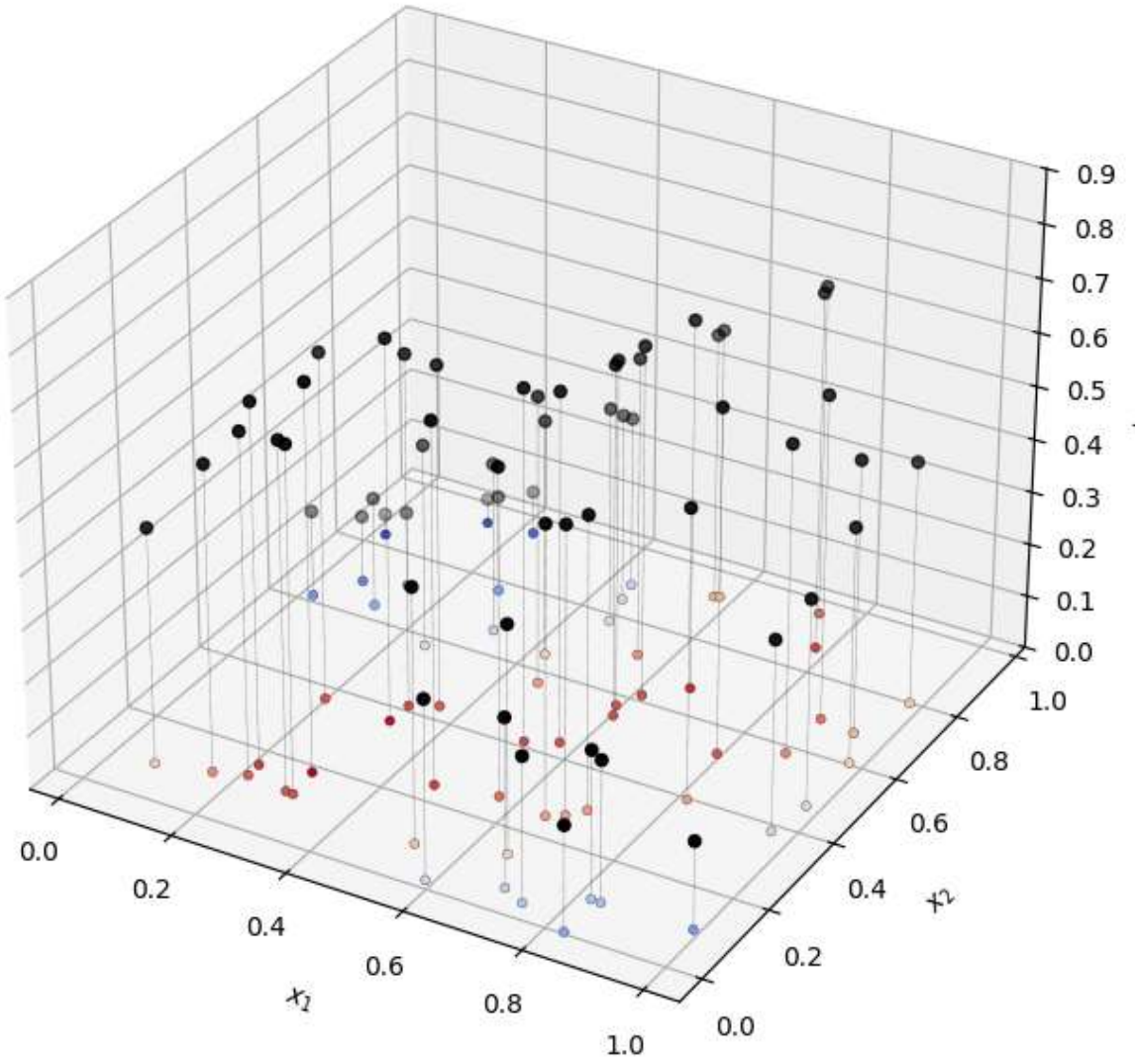
Function for 3D plotting

In [2]:
```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

# Writing a 3D Plotting function. Inputs data points and regression function
def plot_data_with_regression(x_data, y_data, regfun=None):
    plt.figure(figsize=(8,8))
    fig = plt.gcf()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(x_data[:,0], x_data[:,1],0*y_data,s=13,c=y_data,zorder=-1,cmap="cool
    ax.scatter(x_data[:,0], x_data[:,1],y_data,s=20,c="black",zorder=-1)
    for i in range(len(y_data)):
        ax.plot([x_data[i,0],x_data[i,0]],[x_data[i,1],x_data[i,1]],[0,y_data[i]],'

    ax.set_xlabel('\n' + r"$x_1$")
    ax.set_ylabel('\n' + r"$x_2$")
    ax.set_zlabel('\n'+r"$y$")
```

```
    ax.set_zlim(0,0.9)

    if regfun is not None:
        vals = np.linspace(0, 1, 100)
        x1grid, x2grid = np.meshgrid(vals, vals)
        y = regfun(np.concatenate((x1grid.reshape(-1,1),x2grid.reshape(-1,1)),1)).r
        ax.plot_surface(x1grid, x2grid, y.reshape(x1grid.shape), alpha = 0.8, cmap
        plt.show()
```

Data visualized

In [3]:
```
plot_data_with_regression(x,y)
```



Demonstration: 2D Linear Regression

First, I generate a design matrix within a function called `get_linear_design_matrix()`

In [4]:
```
def get_linear_design_matrix(x):
    x1 = x[:,0].reshape(-1, 1)
    x2 = x[:,1].reshape(-1, 1)
    columns = [x1, x2, np.ones_like(x1)]   # Linear design matrix has a column of x
    X = np.concatenate(columns, axis=1)   # Combine each column horizontally to mak
    return X
```

In [5]:
```
X = get_linear_design_matrix(x)
print("First four rows of X:")
print(X[:4,:])
```

```
First four rows of X:
[[0.5488135  0.71518937 1.        ]
 [0.60276338 0.54488318 1.        ]
 [0.4236548  0.64589411 1.        ]
 [0.43758721 0.891773   1.        ]]
```

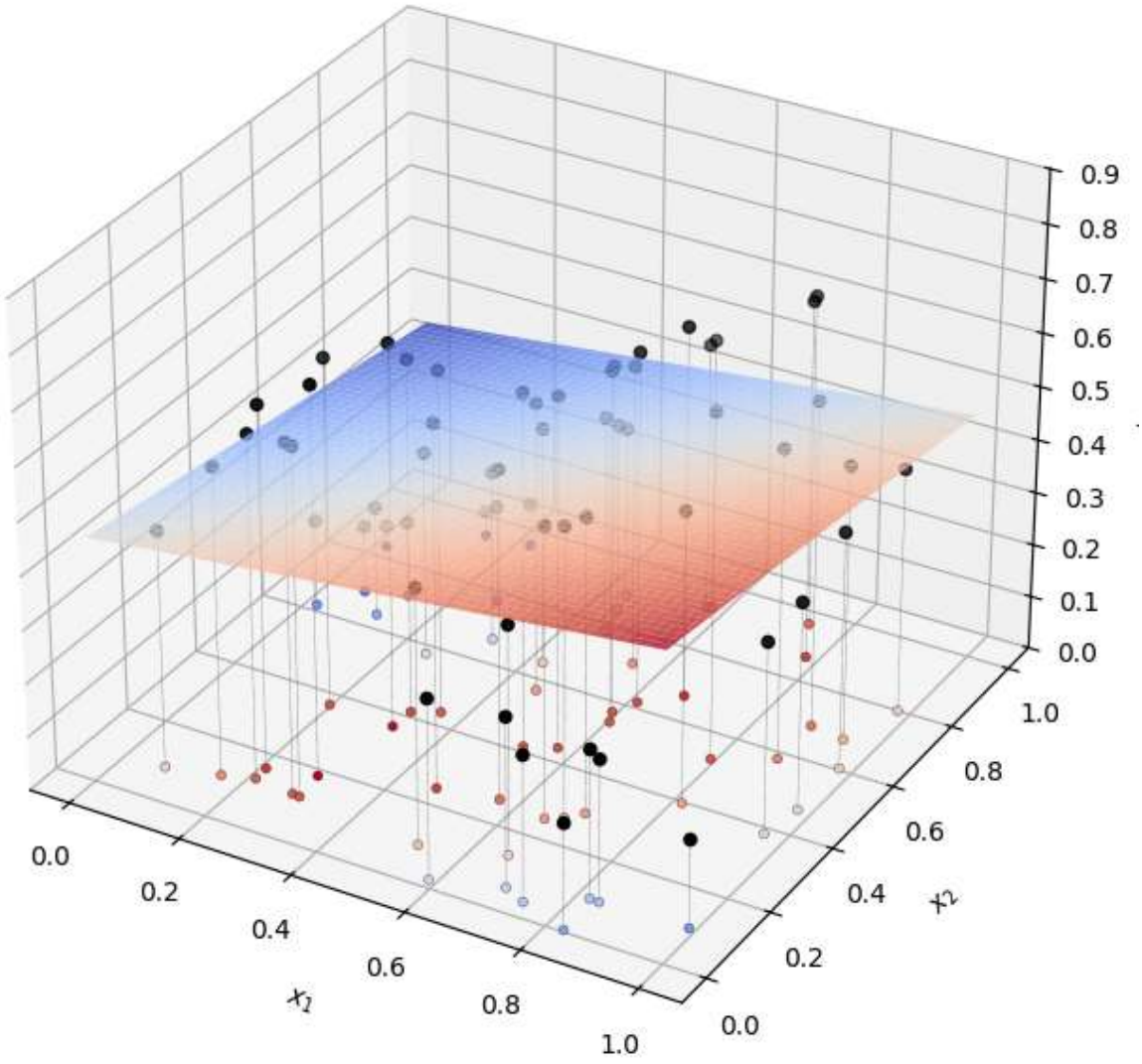
Next, get the coefficients of the regression:

In [6]:
```
# Get coefficients
w1 = np.linalg.inv(X.T @ X) @ X.T @ y.reshape(-1,1)
print("Linear Coefficients:", w1.flatten())
```

```
Linear Coefficients: [ 0.11233939 -0.10638434  0.45720916]
```

Finally, we plot the result. Here, `plot_data_with_regression()` takes as input the x input data, y output data, and a function which performs the desired regression. Therefore I first define said regression function, and plug it in as an argument to the plotting function:

In [7]:
```
def do_2d_linear_regression(x):
    y_fit = get_linear_design_matrix(x) @ w1
    return y_fit

plot_data_with_regression(x, y, do_2d_linear_regression)
```



Your Turn: 2D Quadratic Regression

The linear regression results are clearly not a great fit. You will see if a 2nd order fit can do any better. Fill in the missing code below to generate a quadratic design matrix and plot the results:

In [8]:
```
def get_quadratic_design_matrix(x):
    x1 = x[:,0].reshape(-1, 1)
    x2 = x[:,1].reshape(-1, 1)

    # YOUR CODE GOES HERE
    # 2ND ORDER, 2-D DESIGN MATRIX NEEDS 6 TOTAL COLUMNS
    columns = [x1 ** 2, x2 ** 2, np.multiply(x1, x2), x1, x2, np.ones_like(x1)]   #
    X = np.concatenate(columns, axis=1)   # Combine each column horizontally to mak

    return X
```

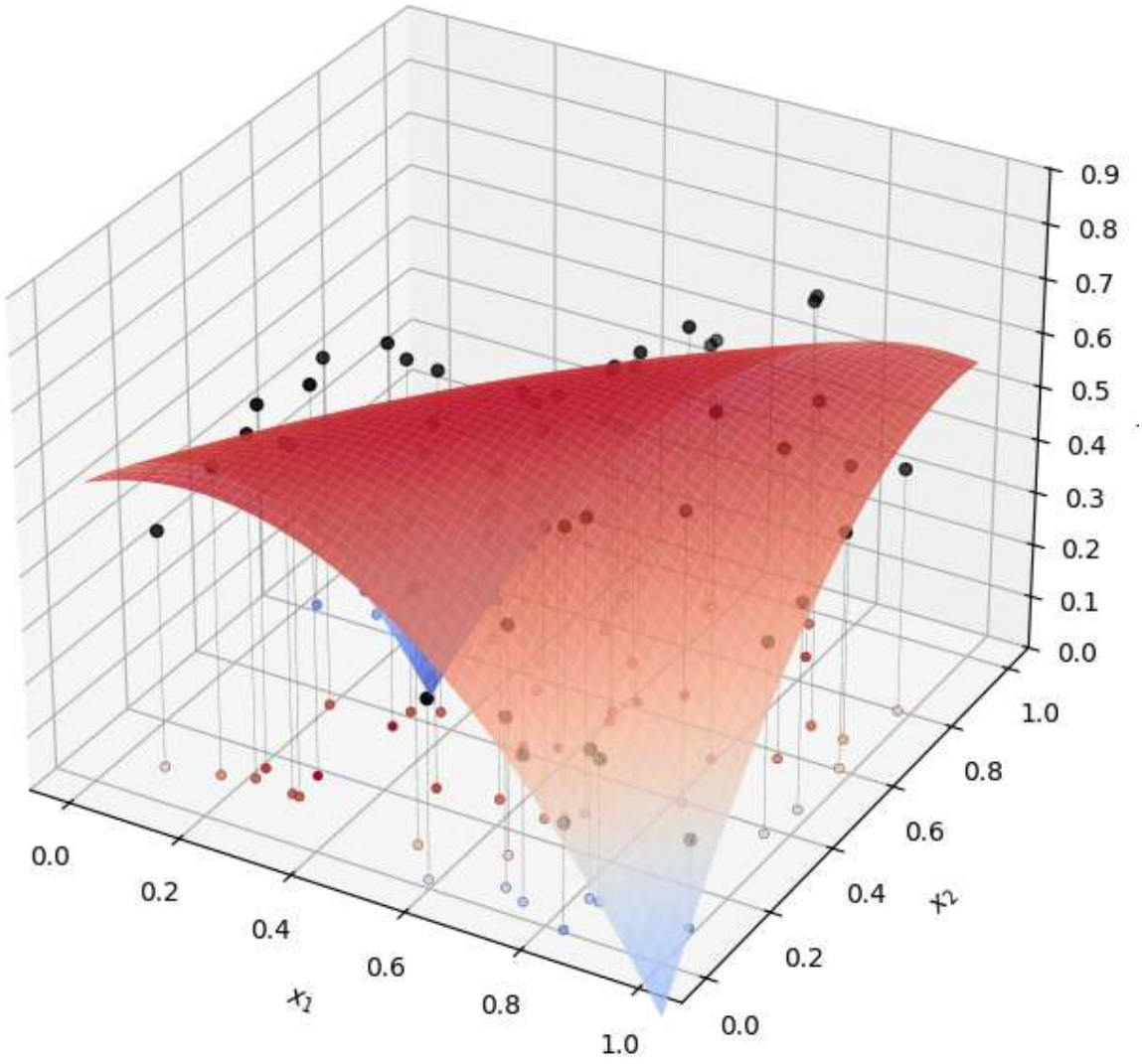
In [9]:
```
X = get_quadratic_design_matrix(x)
print("First four rows of X:")
print(X[:4,:])
```

```
First four rows of X:
[[0.30119626 0.51149583 0.39250558 0.5488135  0.71518937 1.        ]
 [0.36332369 0.29689768 0.32843563 0.60276338 0.54488318 1.        ]
 [0.17948339 0.41717921 0.27363614 0.4236548  0.64589411 1.        ]
 [0.19148257 0.79525908 0.39022846 0.43758721 0.891773   1.        ]]
```

In [10]:
```
# Get coefficients
w2 = np.linalg.inv(X.T @ X) @ X.T @ y.reshape(-1,1)
print("Quadratic Coefficients:", w2.flatten())
```

```
Quadratic Coefficients: [-1.09949493 -0.78655383  1.62592273  0.44193704 -0.17753776
0.55677679]
```

In [16]:
```
def do_2d_quadratic_regression(x):
    y_fit = get_quadratic_design_matrix(x) @ w2
    return y_fit

plot_data_with_regression(x, y, do_2d_quadratic_regression)
```


0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.0
0.2
0.4
0.6
0.8
1.0
x_1
0.0
0.2
0.4
0.6
0.8
1.0
x_2

M2-L2 Problem 1 (5 points)

In this problem you will perform cubic least squares regression using gradient descent. Starting with an initial guess, you will plot the regression curve as you perform many iterations of gradient descent, and observe how the regression performance improves.

Start by running the following cell to load data and define helpful functions. Data is loaded into 'x' and 'y', with a cubic design matrix as 'X'.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

def plot_data_with_regression(x_data, y_data, x_reg, y_reg, title=""):
    plt.figure()

    plt.scatter(x_data.flatten(), y_data.flatten(), label="Data", c="black")
    plt.plot(x_reg.flatten(), y_reg.flatten(), label="Fit")

    plt.legend()
    plt.xlabel(r"$x_1$")
    plt.ylabel(r"$y$")
    plt.xlim(-2,2)
    plt.ylim(-2,2)
    plt.title(title)
    plt.show()

x = np.array([-1.52362349, -1.60576489, -1.34827768, -1.45340266, -1.42652973, -1.2
y = np.array([ 1.65517515,  1.33249684,  1.38328432,  1.1531808 ,  0.89478436,0.667
X = np.vstack([x*x*x, x*x, x, np.ones_like(x)]).T

xreg = np.linspace(-2,2)
Xreg = np.vstack([xreg*xreg*xreg, xreg*xreg, xreg, np.ones_like(xreg)]).T
```

Gradient function

We need to define a function for the gradient of our objective function. In the lecture, this gradient was:

$$\frac{\partial \text{obj}}{\partial w} = 2X'(Xw - y)$$

Please complete the function below to compute the gradient. Recall that the design matrix is the global variable X, and you may need to reshape the output y as a column vector.

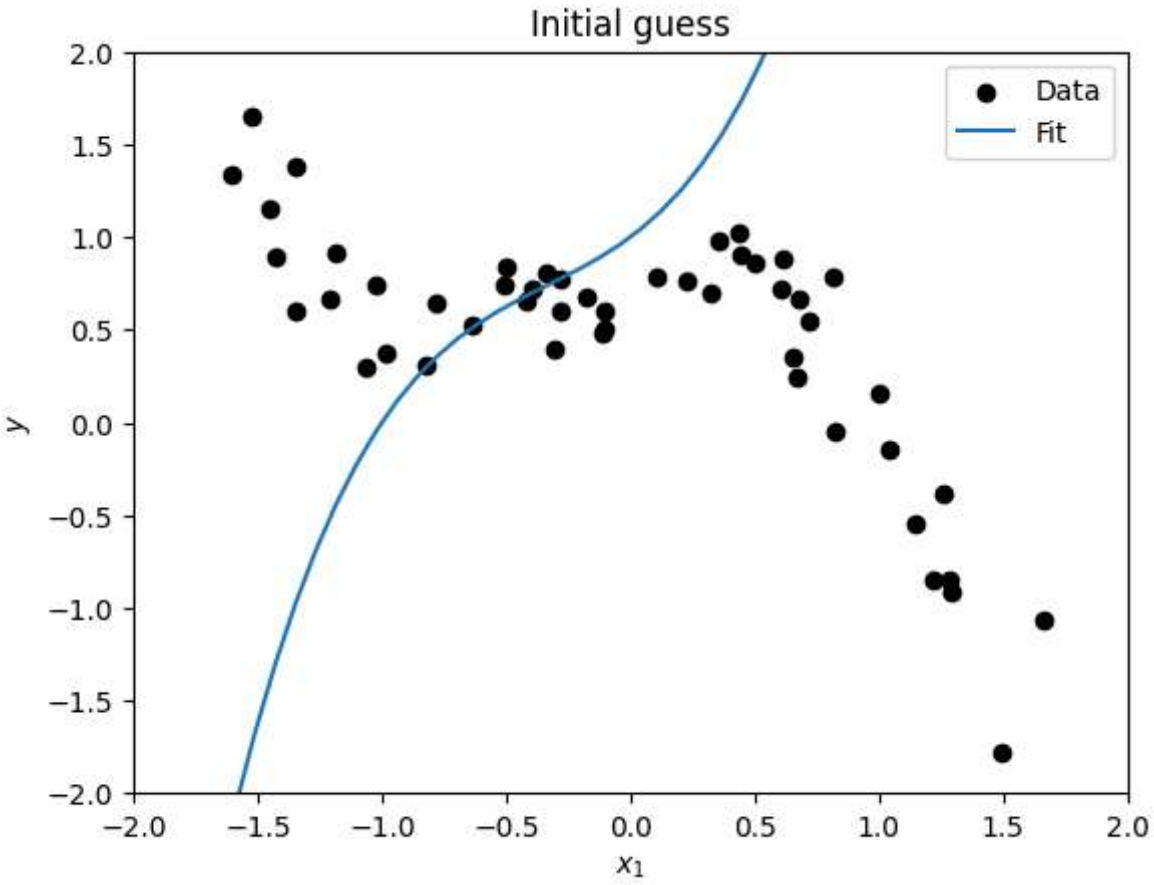
In [2]:
```
def grad(w):
    # YOUR CODE GOES HERE
    # Return the gradient of the objective function
```

```
    grad = 2 * X.T @ (X @ w - y.reshape(-1, 1))
    return grad
```

Initial guess

Let's define an initial guess for w and visualize the resulting curve:

In [3]:
```
w = np.array([[1,1,1,1]]).reshape(-1, 1)
yreg = Xreg @ w
plot_data_with_regression(x,y,xreg, yreg, "Initial guess")
```



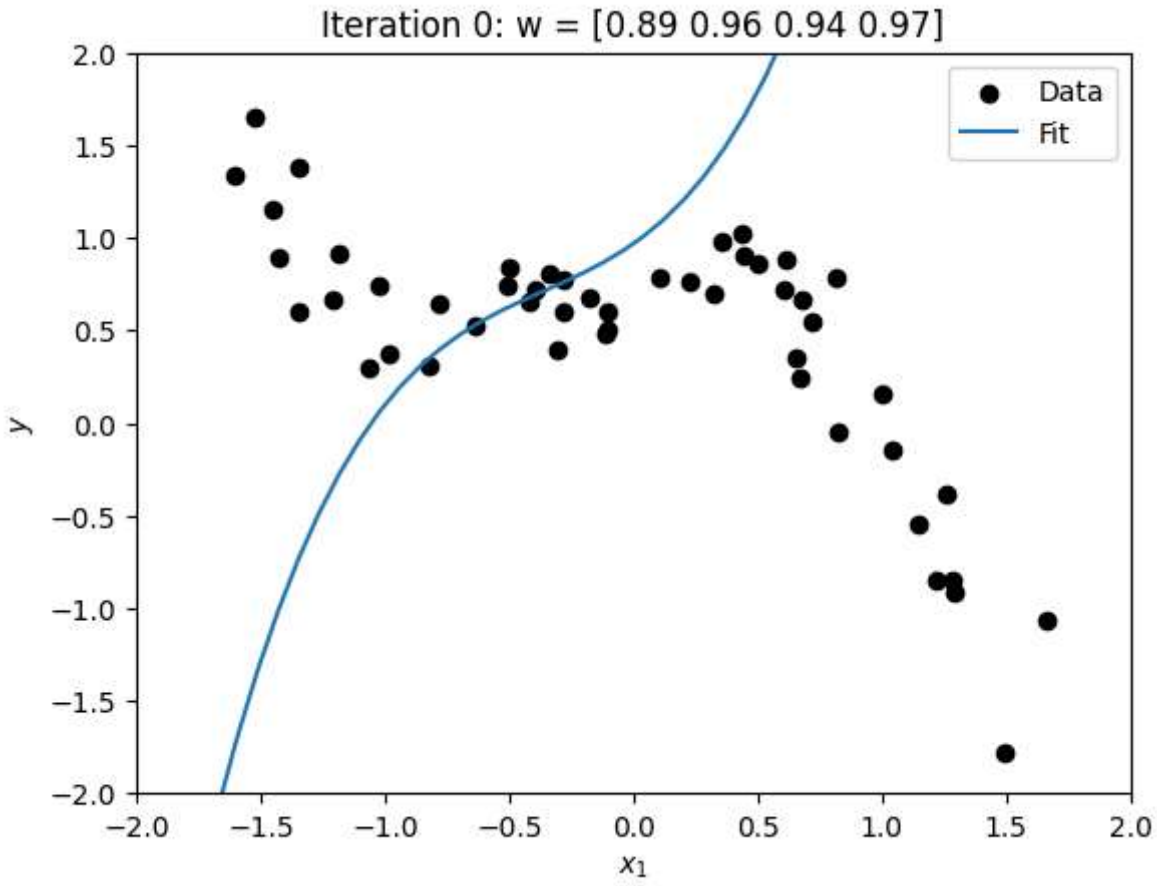
Gradient Descent Loop

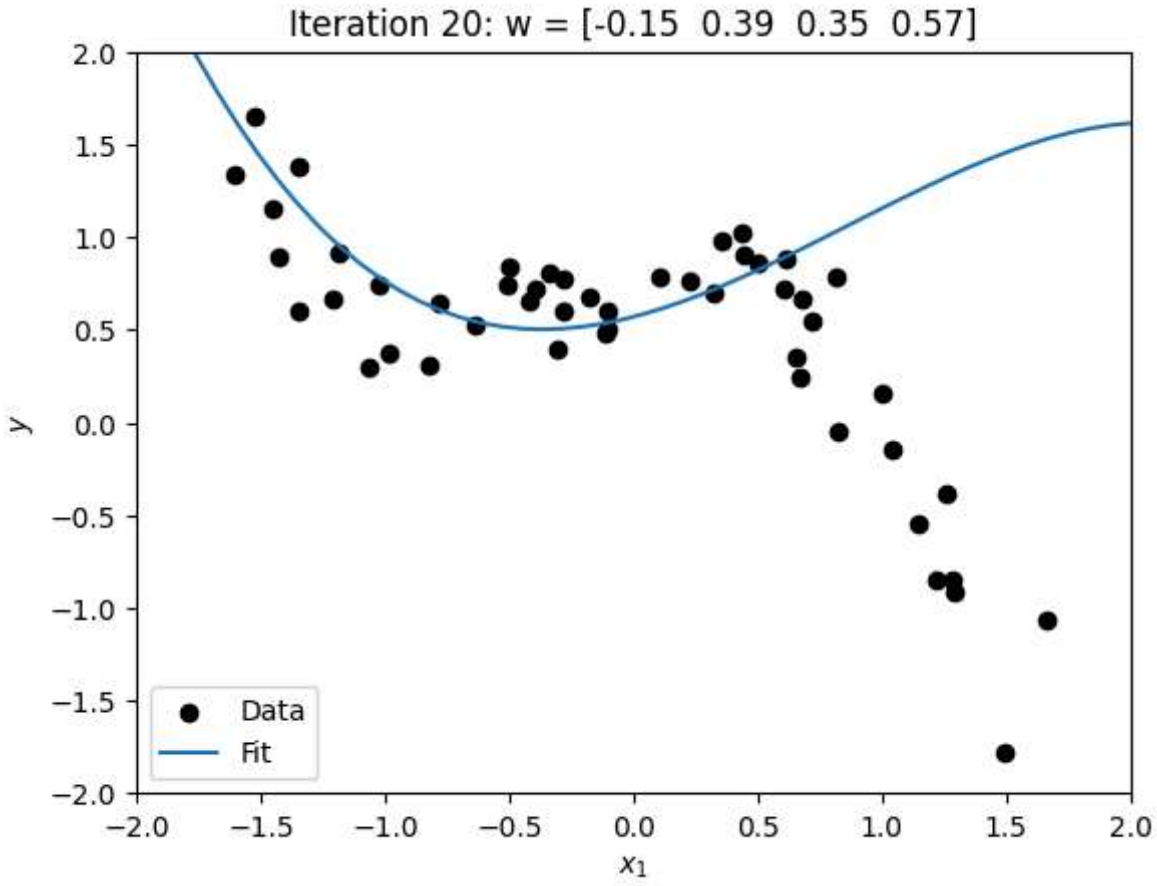
Now complete the gradient descent loop below. You should call your gradient function, update w by subtracting eta (the step size) times the gradient.

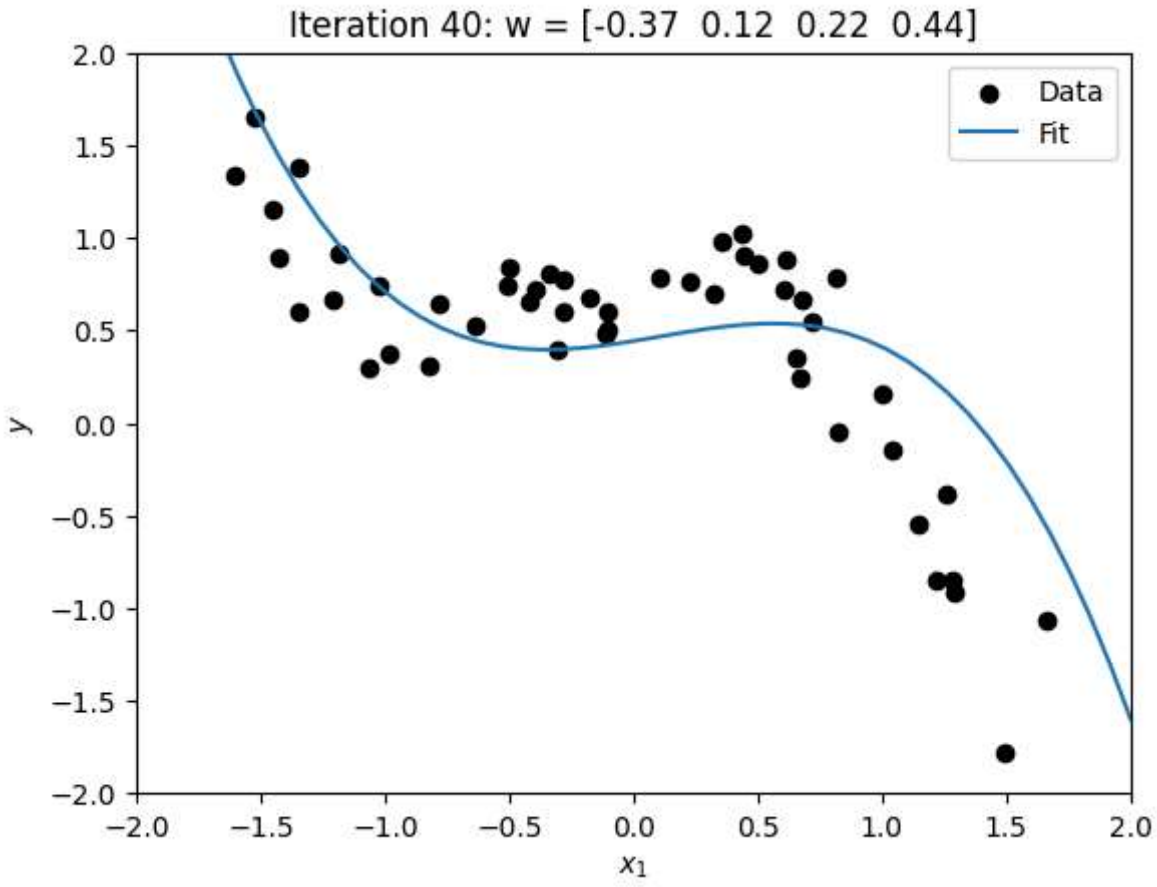
Every several iterations, the result is plotted. You should observe the gradual improvement of the model.

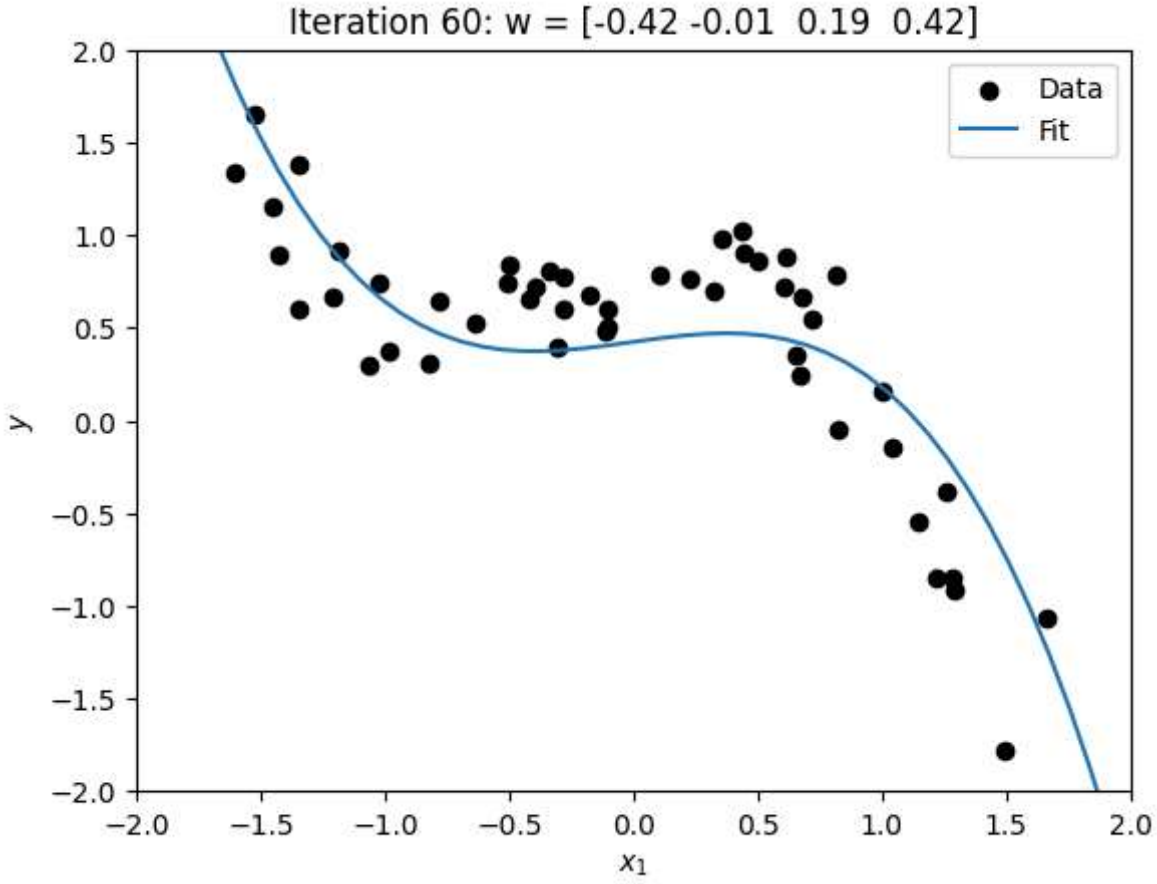
In [4]:
```
eta = 0.00025
for i in range(101):

    # YOUR CODE GOES HERE
```

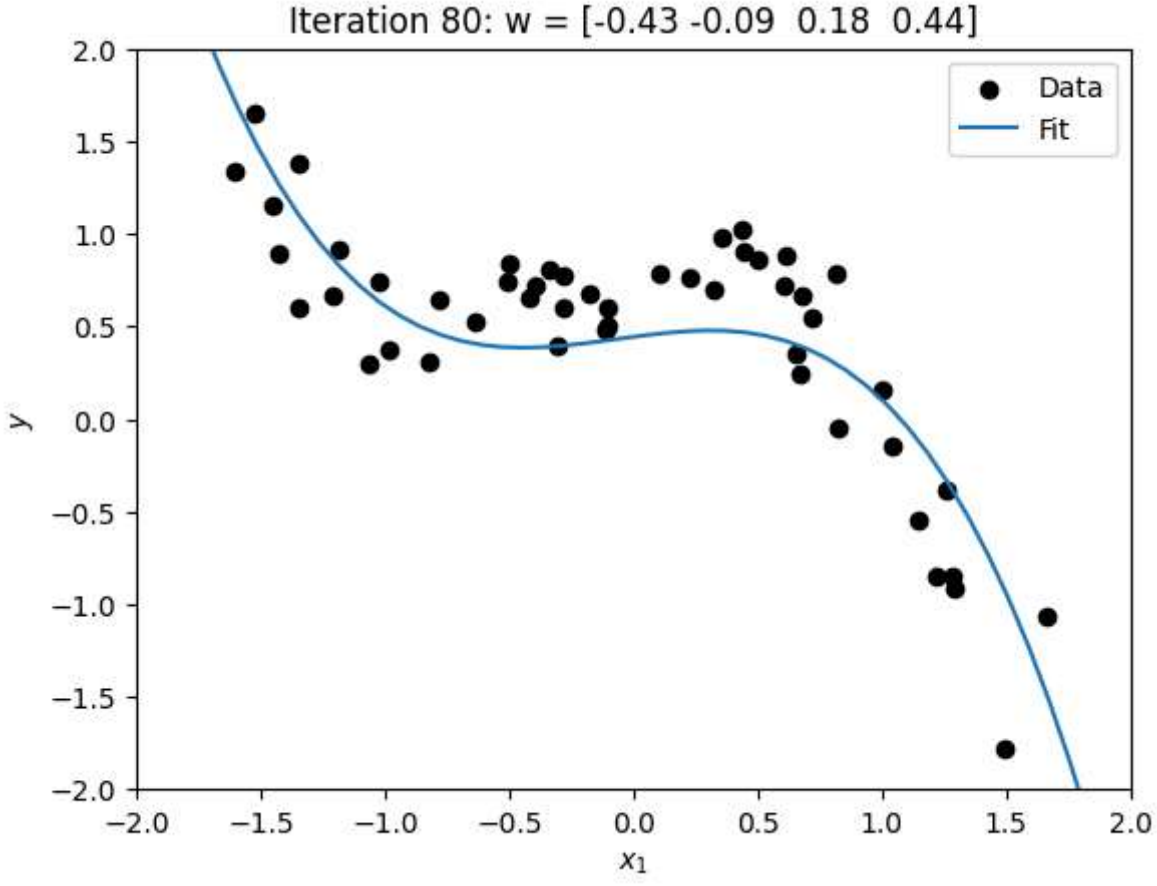
```
    # Update w
    w = w - eta * grad(w)
    if 0 == i%20:
        yreg = Xreg @ w
        plot_data_with_regression(x,y,xreg, yreg, f"Iteration {i}: w = {np.round(w.
```

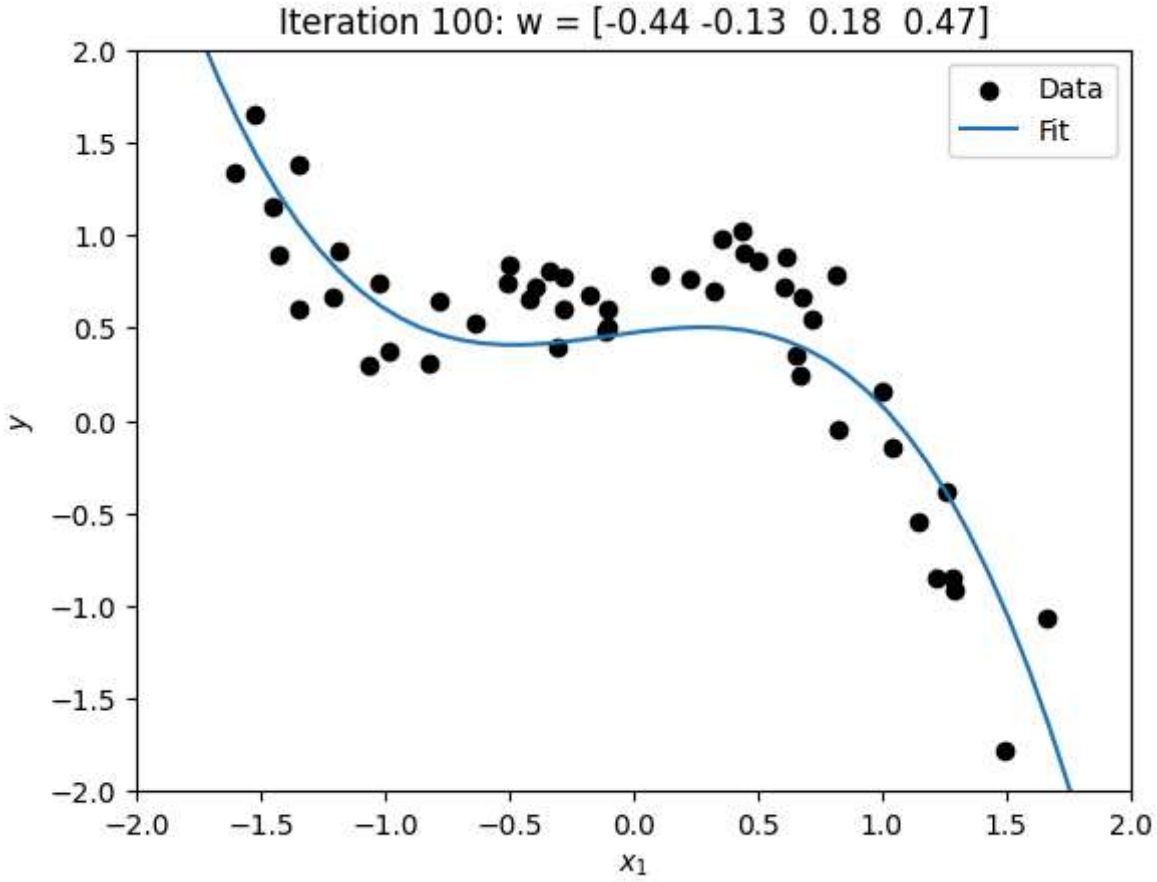



Iteration 20: w = [-0.15 0.39 0.35 0.57]
Data
Fit
y
x_1


Iteration 40: w = [-0.37 0.12 0.22 0.44]
Data
Fit
y
x_1


Iteration 60: w = [-0.42 -0.01 0.19 0.42]
Data
Fit
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x_1


Iteration 80: w = [-0.43 -0.09 0.18 0.44]
Data
Fit
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x_1


Iteration 100: w = [-0.44 -0.13 0.18 0.47]
Data
Fit
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x_1

M2-L2 Problem 2 (5 points)

In this problem we will perform least-squares regression using sklearn's built-in tools.

First, you will generate a standard linear least squares regression model with `LinearRegression`.

Next, you will use stochastic gradient descent to train another model with `SGDRegressor`.

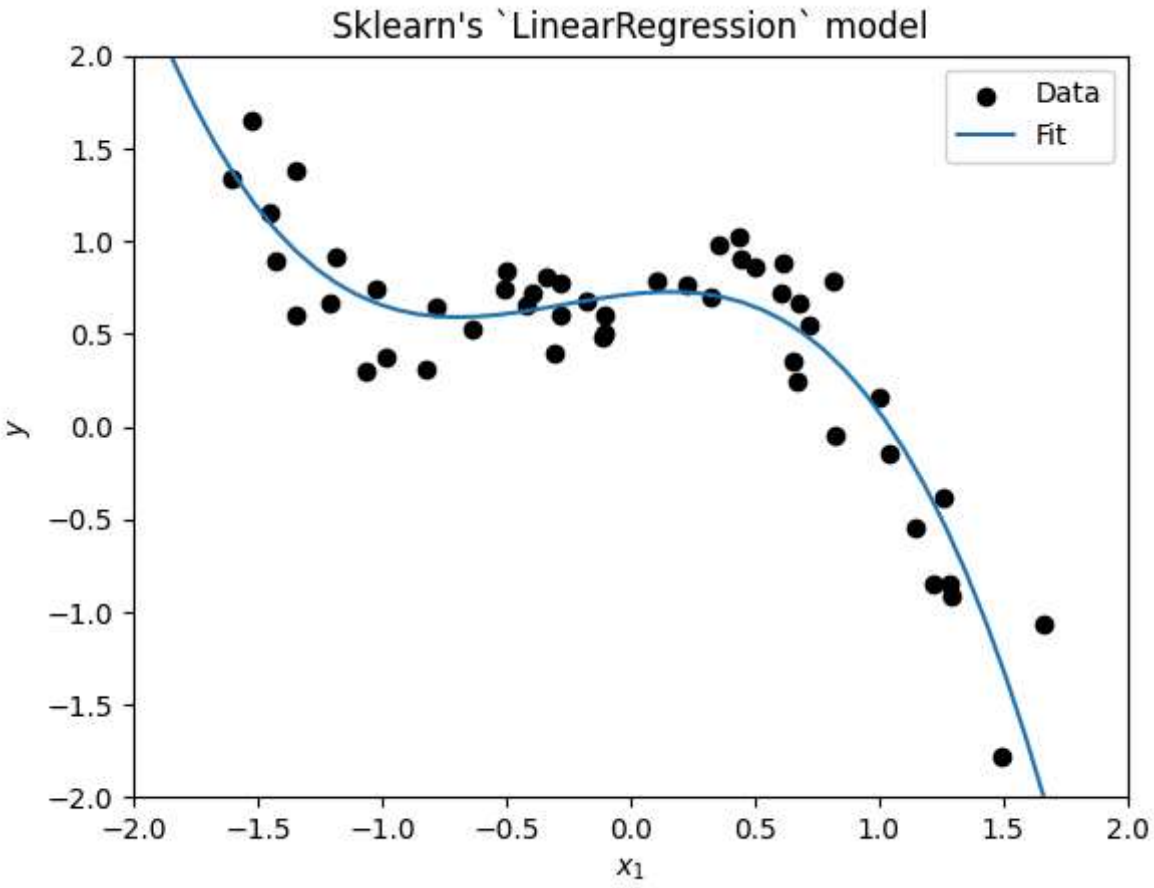
Run this cell to perform the required imports and load the data:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import SGDRegressor

def plot_data_with_regression(x_data, y_data, x_reg, y_reg, title=""):
    plt.figure()

    plt.scatter(x_data.flatten(), y_data.flatten(), label="Data", c="black")
    plt.plot(x_reg.flatten(), y_reg.flatten(), label="Fit")

    plt.legend()
    plt.xlabel(r"$x_1$")
    plt.ylabel(r"$y$")
    plt.xlim(-2,2)
    plt.ylim(-2,2)
    plt.title(title)
    plt.show()

x = np.array([-1.52362349, -1.60576489, -1.34827768, -1.45340266, -1.42652973, -1.2
y = np.array([ 1.65517515,  1.33249684,  1.38328432,  1.1531808 ,  0.89478436,0.667
X = np.vstack([x*x*x, x*x, x, np.ones_like(x)]).T

xreg = np.linspace(-2,2)
Xreg = np.vstack([xreg*xreg*xreg, xreg*xreg, xreg, np.ones_like(xreg)]).T
```

Least Squares Regression

We have provided a demonstration of least squares regression using sklearn:

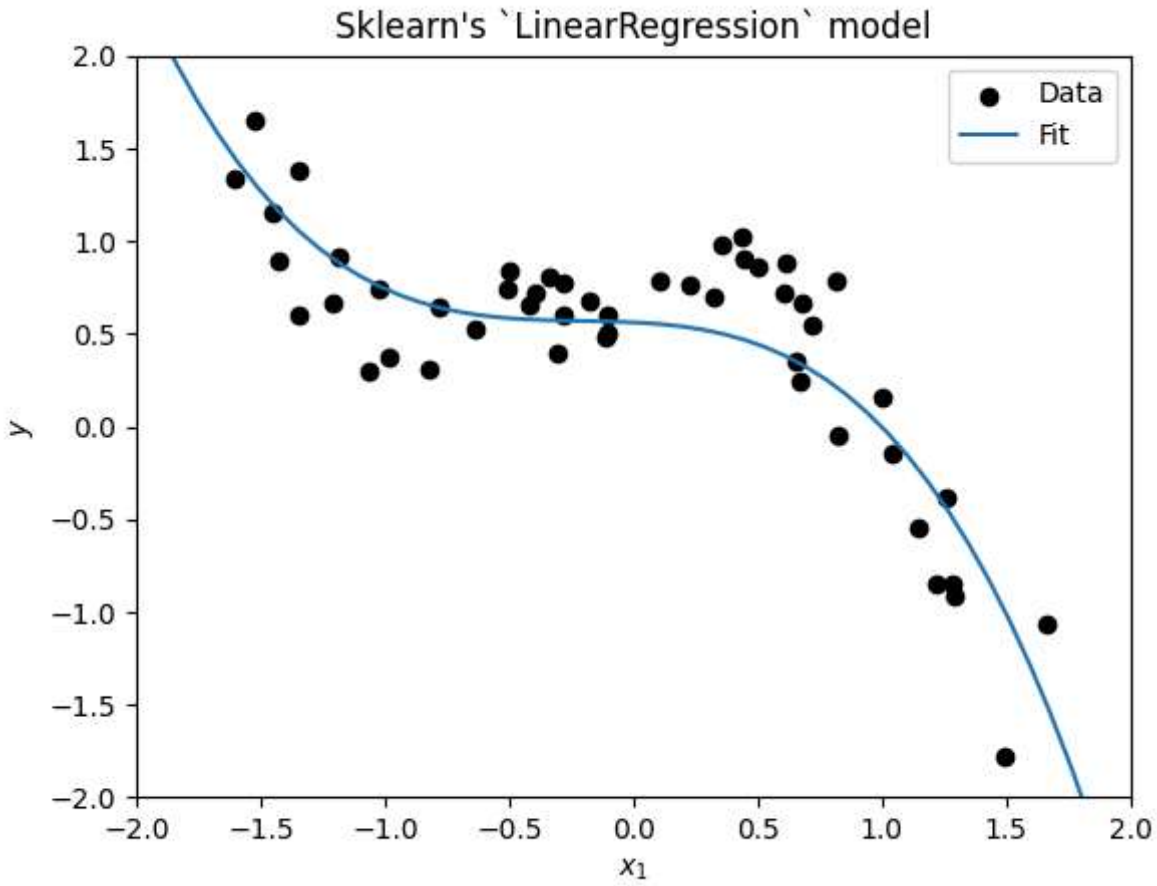
In [2]:
```
model = LinearRegression()
model.fit(X,y)

yreg = model.predict(Xreg)
plot_data_with_regression(x, y, xreg, yreg, "Sklearn's `LinearRegression` model")
```



Using SGD

Now use stochastic gradient descent to solve the same problem and make a similar plot, but for a `SGDRegressor` model instead of `LinearRegression`:

In [3]:
```
# YOUR CODE GOES HERE
model = SGDRegressor()
model.fit(X,y)

yreg = model.predict(Xreg)
plot_data_with_regression(x, y, xreg, yreg, "Sklearn's `LinearRegression` model")
```


Sklearn's `LinearRegression` model
Data
Fit
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x_1

M2-L2 Problem 3 (5 points)

When flow is directed across a pin fin heat sink, increasing fluid velocity can improve the heat transfer, making the heat sink more effective.

You have been given a dataset containing measurements for such a scenario, which contains the following:

- Input: Reynolds Number of air flowing past the heat sink
- Output: Heat transfer coefficient of the heat sink, in W/(m ^{2}) K

Your job is to train a model on this data to predict the heat transfer coefficient, given Reynolds number as input. You will use a high-order polynomial

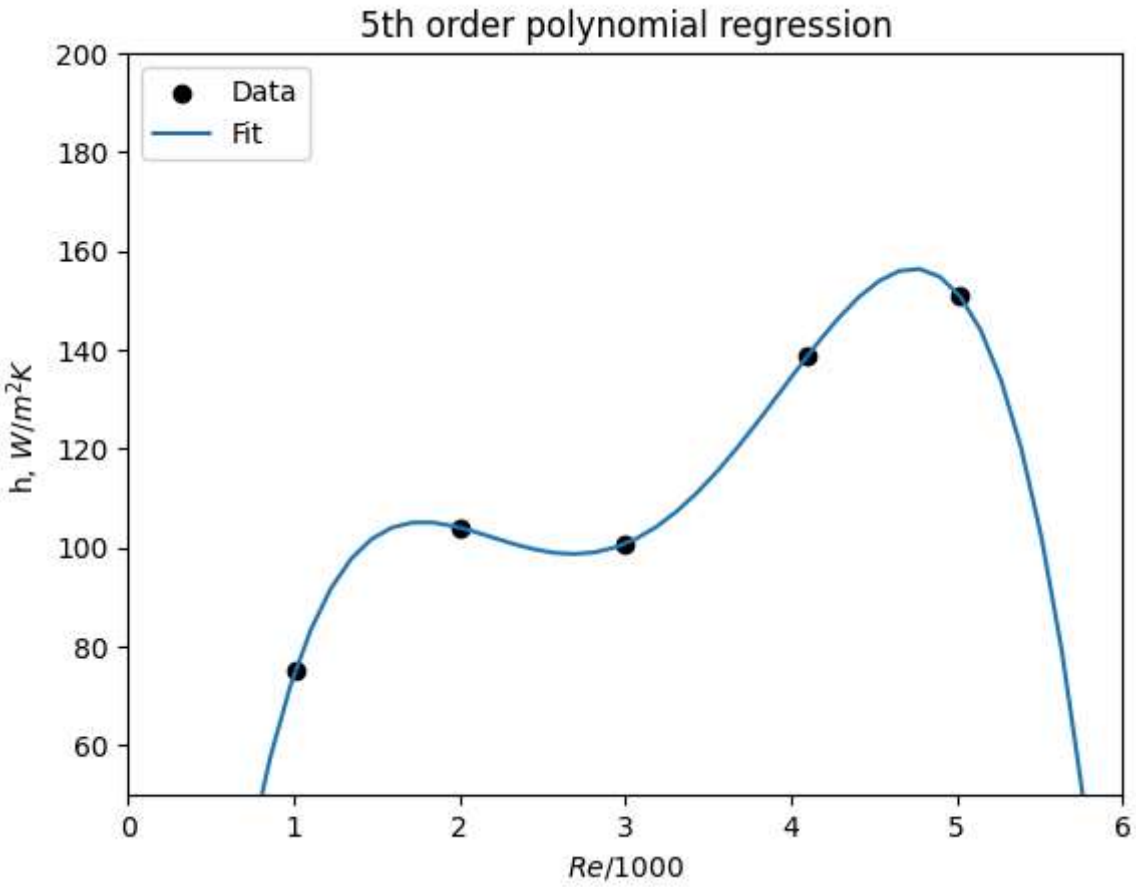
Start by loading the data in the following cell:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

def plot_data_with_regression(x_data, y_data, x_reg, y_reg, title=""):
    plt.figure()

    plt.scatter(x_data.flatten(), y_data.flatten(), label="Data", c="black")
    plt.plot(x_reg.flatten(), y_reg.flatten(), label="Fit")

    plt.legend(loc="upper left")
    plt.xlabel(r"$Re / 1000$")
    plt.ylabel(r"h, $W/m^2 K$")
    plt.xlim(0,6)
    plt.ylim(50,200)
    plt.title(title)
    plt.show()

deg = 5
x = np.array([1.010,2.000,2.990,4.100,5.020])
y = np.array([75.1, 104.0,100.6,138.8,150.8])
X = np.vander(x,deg)

xreg = np.linspace(0,6)
Xreg = np.vander(xreg,deg)
```

Least Squares Regression

As we have done for previous problems, we can do least squares regression by computing the pseudo-inverse of the design matrix. Notice how the model performs beyond the training data.

In [2]:
```
w = np.linalg.inv(X.T @ X) @ X.T @ y.reshape(-1,1)
yreg = Xreg @ w
plot_data_with_regression(x, y, xreg, yreg, "5th order polynomial regression")
```



L2 Regularization

Notice that the plot above reveals that our fifth-order model is overfitting to the data. Let's try applying L2 regularization to fix this. In the lecture, the closed-form solution to least squares with L2 regularization was:

$$w = (X'X + \lambda I_m)^{-1} X'y$$

where I_m is the identity matrix, but with zero in the bias row/column instead of 1; λ is regularization strength; X' is the design matrix and y column vector output.

Complete the function below to compute this w for a given lambda:

In [3]:
```
def get_regularized_w(L):
    I_m = np.eye(deg)
    I_m[-1,-1] = 0

    # YOUR CODE GOES HERE
    # return regularized w
```

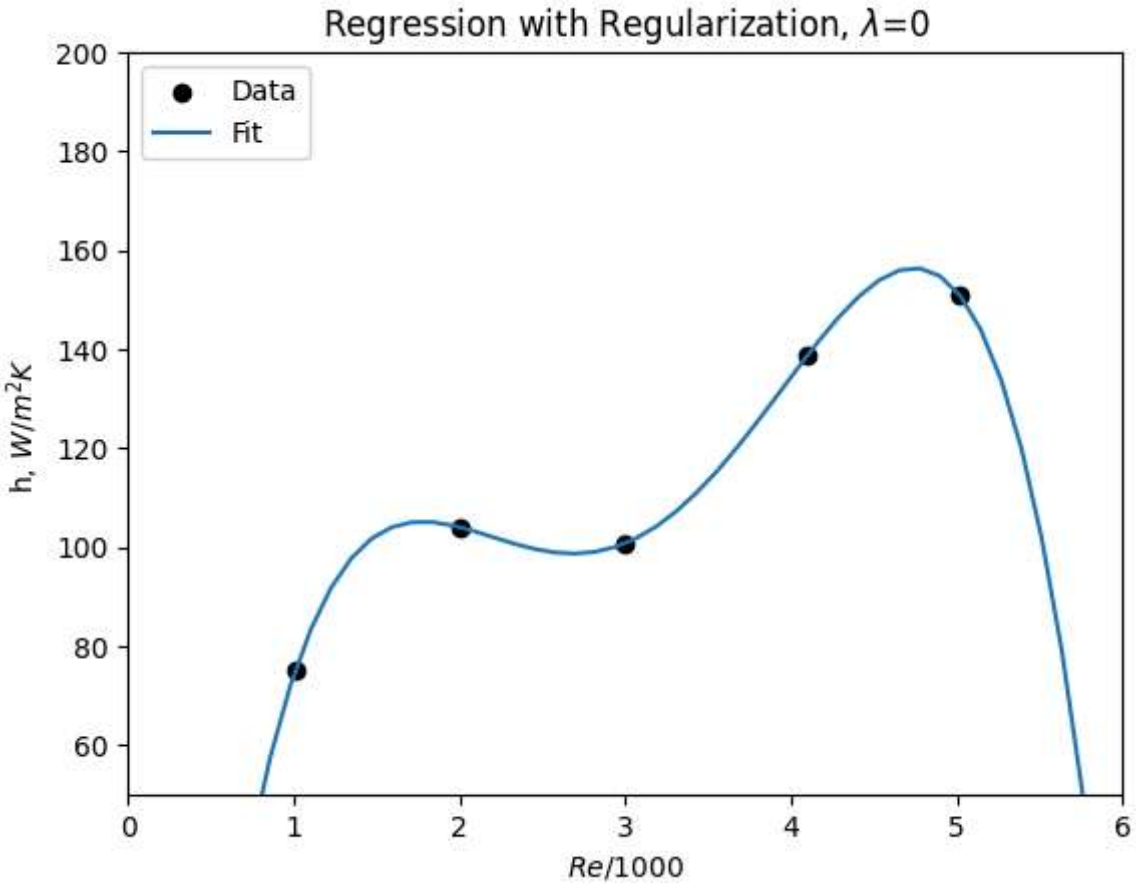
```
    w = np.linalg.inv(X.T @ X + L * I_m) @ X.T @ y.reshape(-1,1)
    return w
```

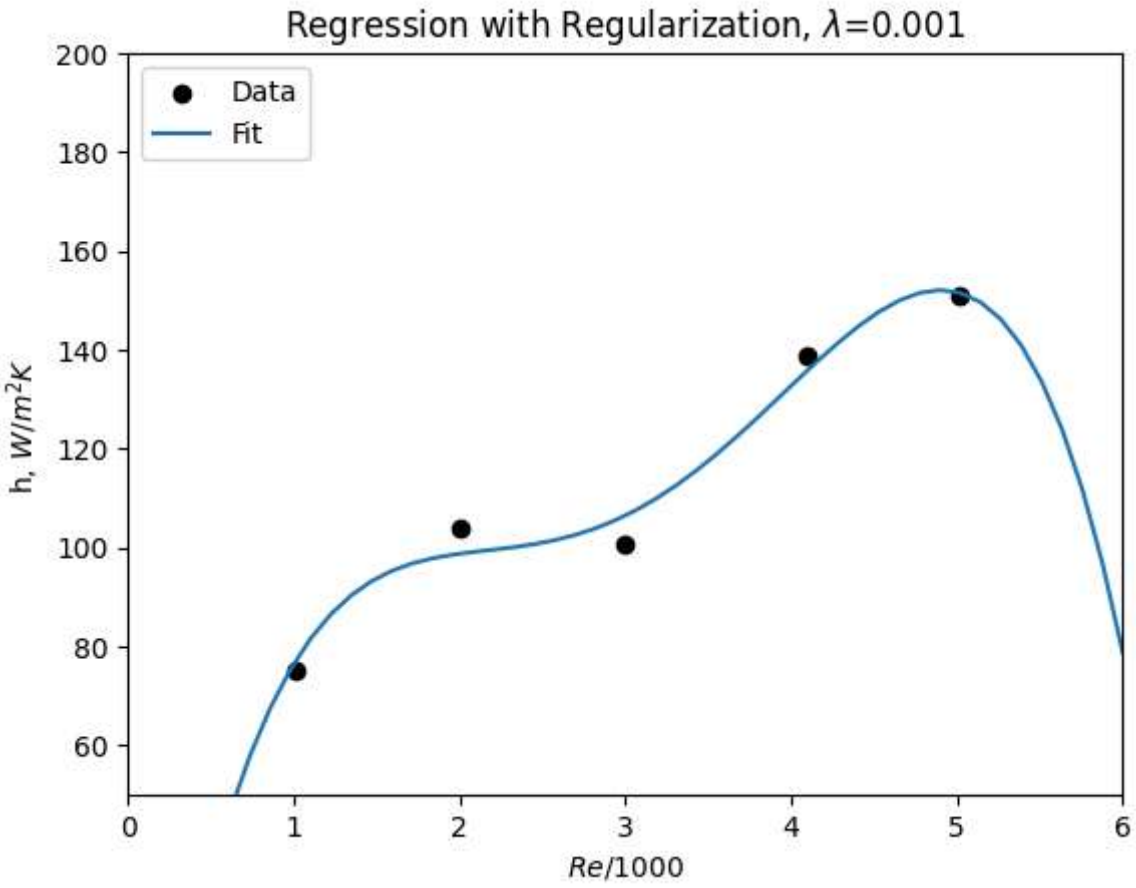
Testing different lambda values

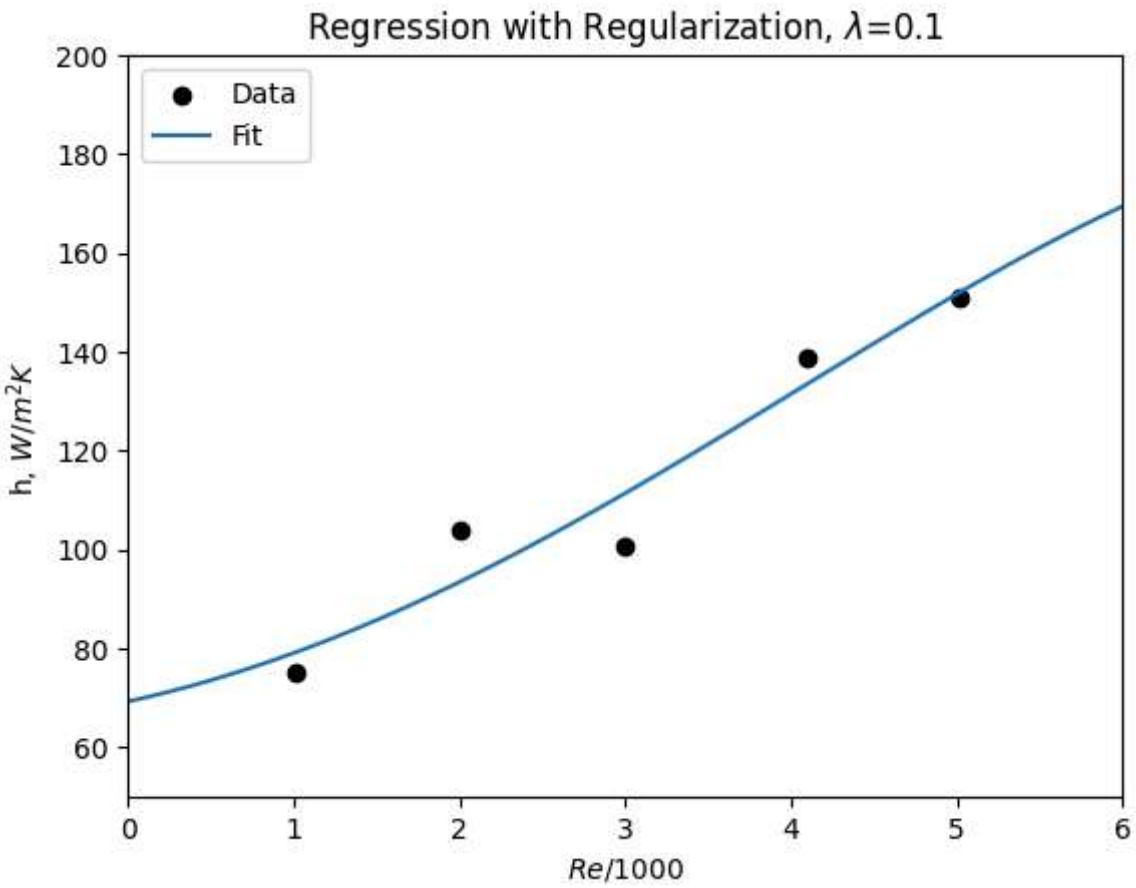
With the above function written, we can compute w for some different values of lambda and decide which is qualitatively best.

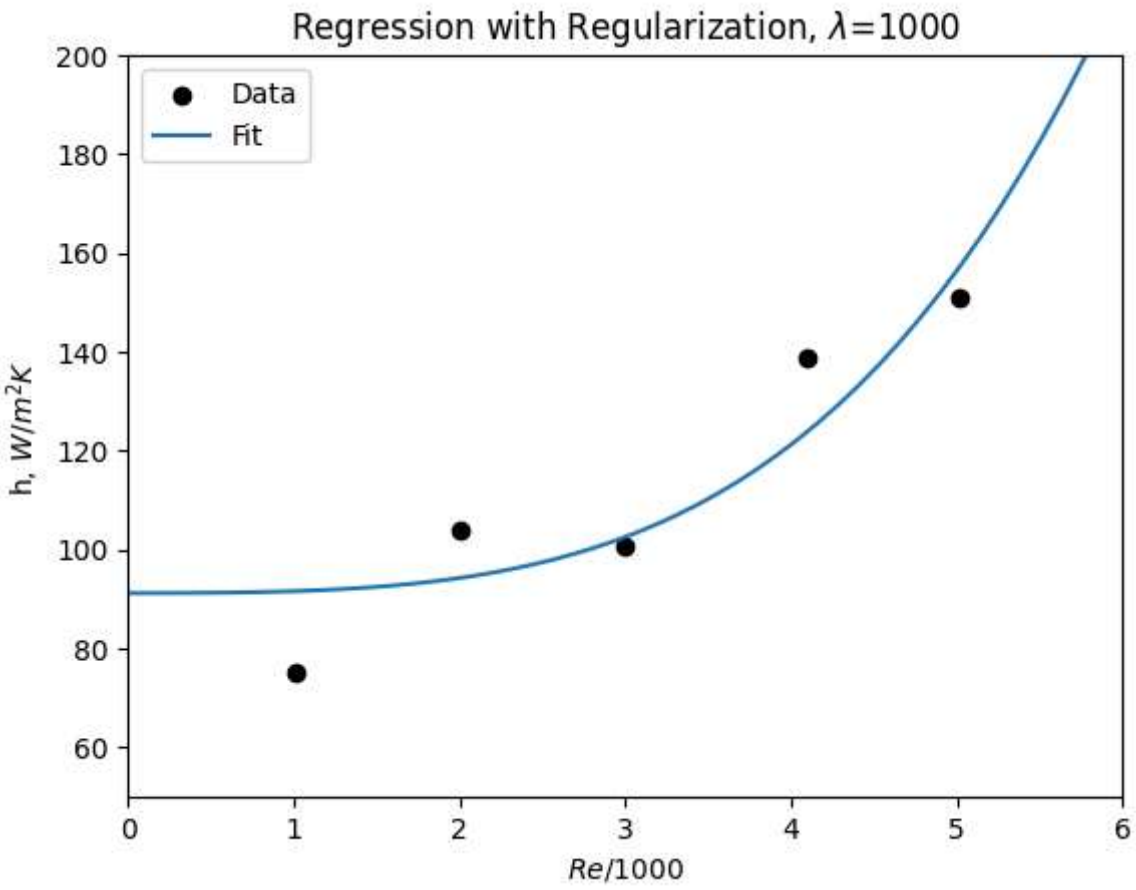
In [4]:
```
for L in [0,.001,0.1,1000]:
    w = get_regularized_w(L)
    yreg = Xreg @ w
    plot_data_with_regression(x, y, xreg, yreg, f"Regression with Regularization, $
```

```
<>:4: SyntaxWarning: invalid escape sequence '\l'
<>:4: SyntaxWarning: invalid escape sequence '\l'
C:\Users\barat\AppData\Local\Temp\ipykernel_20756\3821688261.py:4: SyntaxWarning: in
valid escape sequence '\l'
  plot_data_with_regression(x, y, xreg, yreg, f"Regression with Regularization, $\la
mbda$={L}")
```




Regression with Regularization, λ=0.001
Data
Fit
h, W/m^2K
Re/1000
200
180
160
140
120
100
80
60
0
1
2
3
4
5
6


Regression with Regularization, λ=0.1
Data
Fit
h, W/m^2K
Re/1000
200
180
160
140
120
100
80
60
0
1
2
3
4
5
6



Model Selection

Which value of lambda appears to yield the "best" model?

The model seems to be 'best fit' when the value of lambda is 0.1. This is probably because the value of lambda is high enough to penalize overfitting, while not so high that it eliminates complexity in the model entirely.

Problem 1 (30 points)

Problem Description

In this problem you will implement polynomial linear least squares regression on two datasets, with and without regularization. Additionally, you will use gradient descent to optimize for the model parameters.

Fill out the notebook as instructed, making the requested plots and printing necessary values.

You are welcome to use any of the code provided in the lecture activities.

Summary of deliverables:

Results:

- Print fitted model parameters `w` for the 4 models requested without regularization
- Print fitted model parameters `w` for the 2 models requested *with* L_2 regularization
- Print fitted model parameters `w` for the one model solved via gradient descent

Plots:

- 2 plots of each dataset along with the ground truth function
- 4 plots of the fitted function along with the respective data and ground truth function for LLS without regularization
- 2 plots of the fitted function along with the respective data and ground truth function for LLS with L_2 regularization
- 1 plot of the fitted function along with the respective data and ground truth function for LLS with L_2 regularization solved via gradient descent

Discussion:

- Discussion of challenges fitting complex models to small datasets
- Discussion of difference between the L_2 regularized model versus the standard model
- Discussion of whether gradient descent could get stuck in a local minimum
- Discussion of gradient descent results versus closed form results

Imports and Utility Functions:

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

def gt_function():
    xt = np.linspace(0,1,101)
```

```
    yt = np.sin(2 *np.pi*xt)
    return xt, yt

def plot_data(x,y,xt,yt,title = None):
    # Provide title as a string e.g. 'string'
    plt.plot(x,y,'bo',label = 'Data')
    plt.plot(xt,yt,'g-', label = 'Ground Truth')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid()
    plt.legend()
    if title:
        plt.title(title)
    plt.show()

def plot_model(x,y,xt,yt,xr,yr,title = None):
    # Provide title as a string e.g. 'string'
    plt.plot(x,y,'bo',label = 'Data')
    plt.plot(xt,yt,'g-', label = 'Ground Truth')
    plt.plot(xr,yr,'r-', label = 'Fitted Function')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid()
    plt.legend()
    if title:
        plt.title(title)
    plt.show()
```

Load and visualize the data

The data is contained in `d10.npy` and `d100.npy` and can be loaded with `np.load()`.

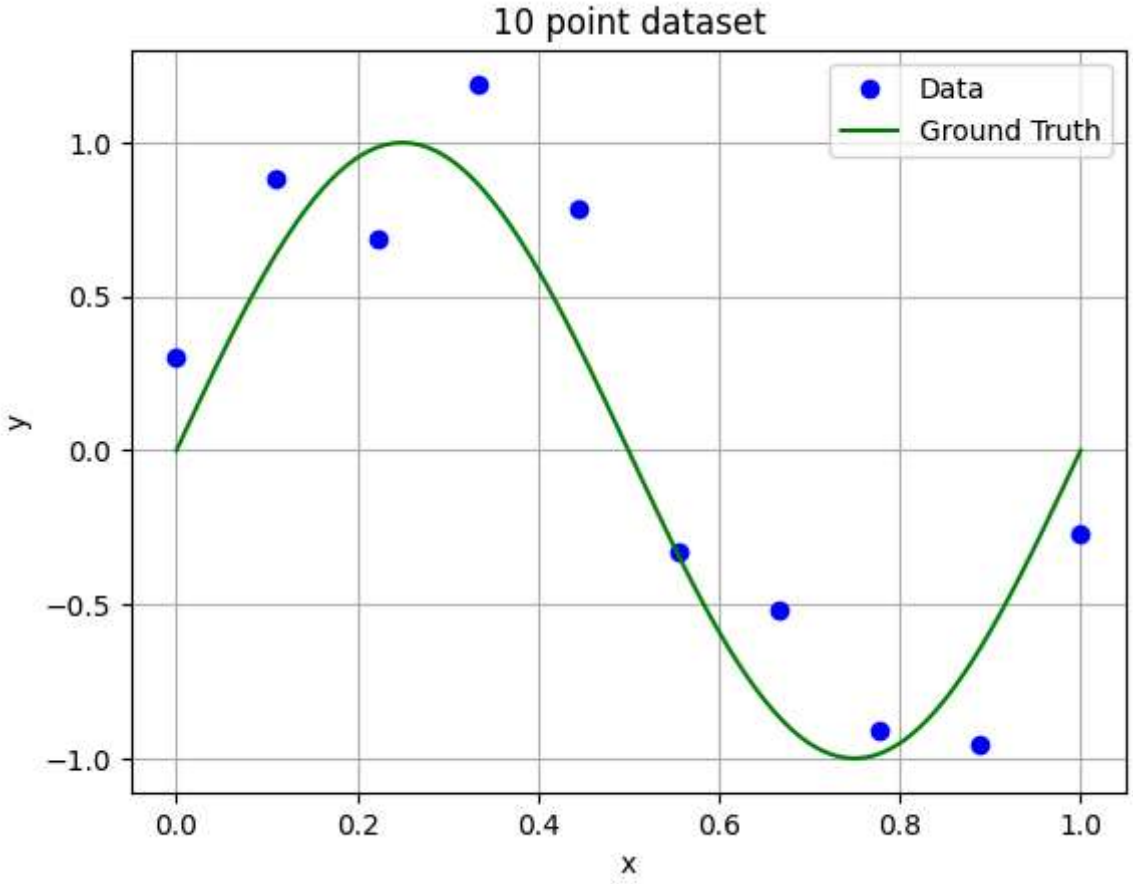
Store the data as:

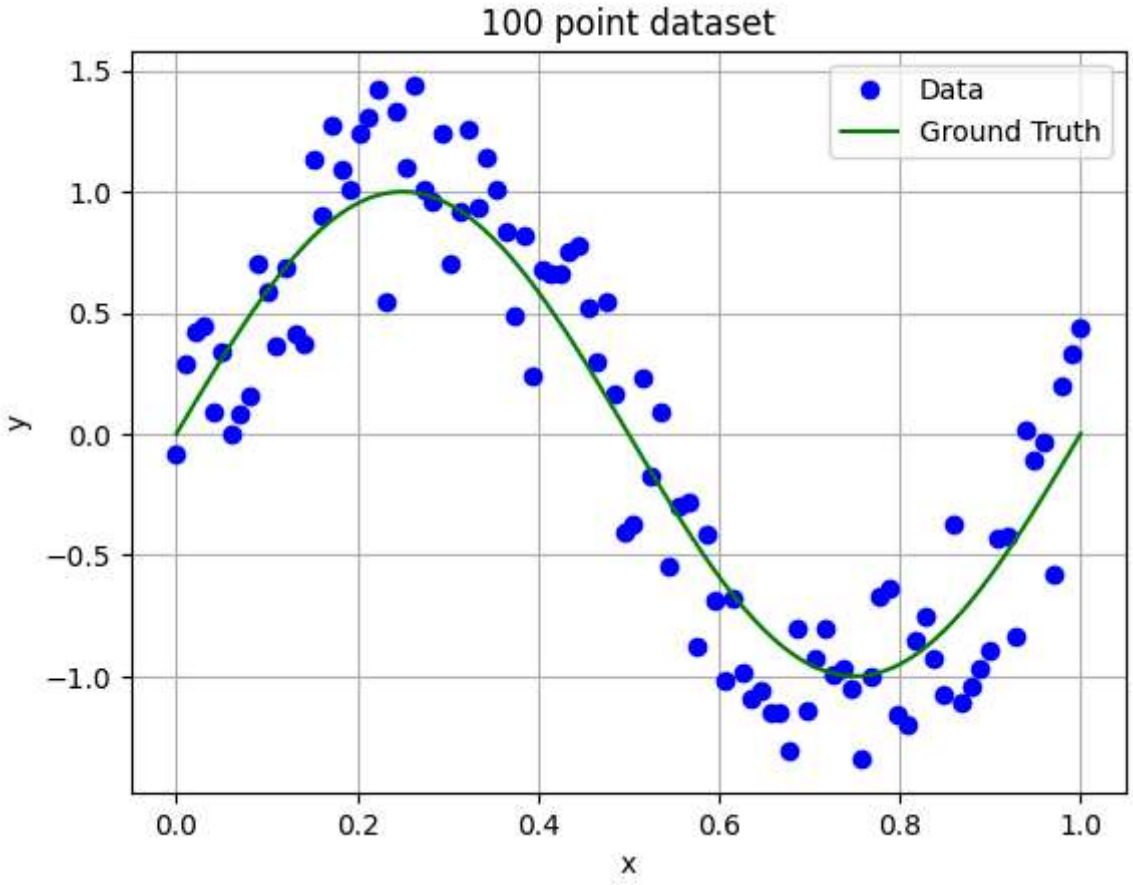
- `x10` and `x100` (the first column of the data)
- `y10` and `y100` (the second column of the data)

Generate the ground truth function $f(x) = \sin(2\pi x)$ using `xt, yt = gt_function()`.

Then visualize the each dataset with `plotxy(x,y,xt,yt,title)` with an appropriate title. You should generate two plots.

In [3]:
```
# YOUR CODE GOES HERE
x10 = np.load('d10.npy')[:, 0]
y10 = np.load('d10.npy')[:, 1]
x100 = np.load('d100.npy')[:, 0]
y100 = np.load('d100.npy')[:, 1]

xt, yt = gt_function()
plot_data(x10, y10, xt, yt, '10 point dataset')
plot_data(x100, y100, xt, yt, '100 point dataset')
```


10 point dataset
Data
Ground Truth
1.0
0.5
0.0
−0.5
−1.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x


100 point dataset
Data
Ground Truth
1.5
1.0
0.5
0.0
−0.5
−1.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x

Implement polynomial linear regression

Now you will implement polynomial linear least squares regression without regularization using the closed form solution from lecture to compute the model parameters. You will consider the following 4 cases:

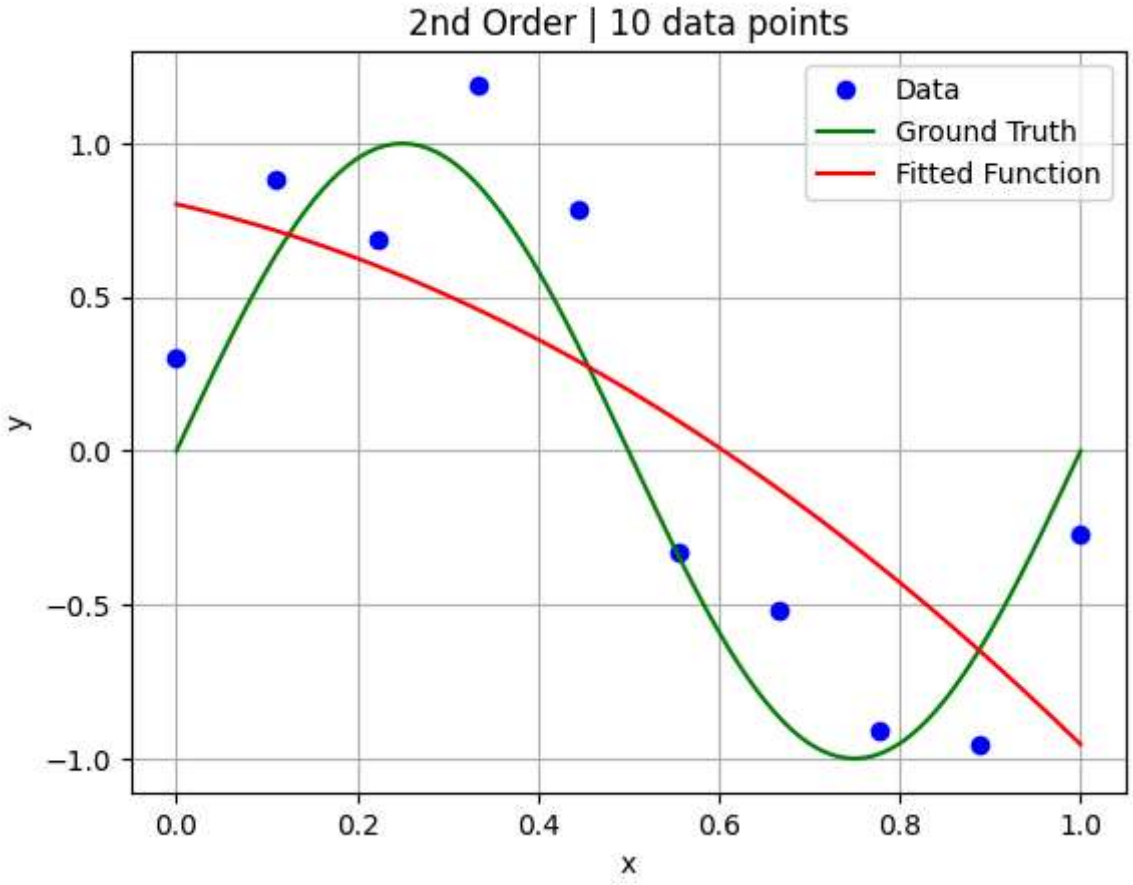
1. Data: data10.txt, Model: 2nd order polynomial (highest power of x in your regression model = 2)
2. Data: data100.txt, Model: 2nd order polynomial (highest power of x in your regression model = 2)
3. Data: data10.txt, Model: 9th order polynomial (highest power of x in your regression model = 9)
4. Data: data100.txt, Model: 9th order polynomial (highest power of x in your regression model = 9)

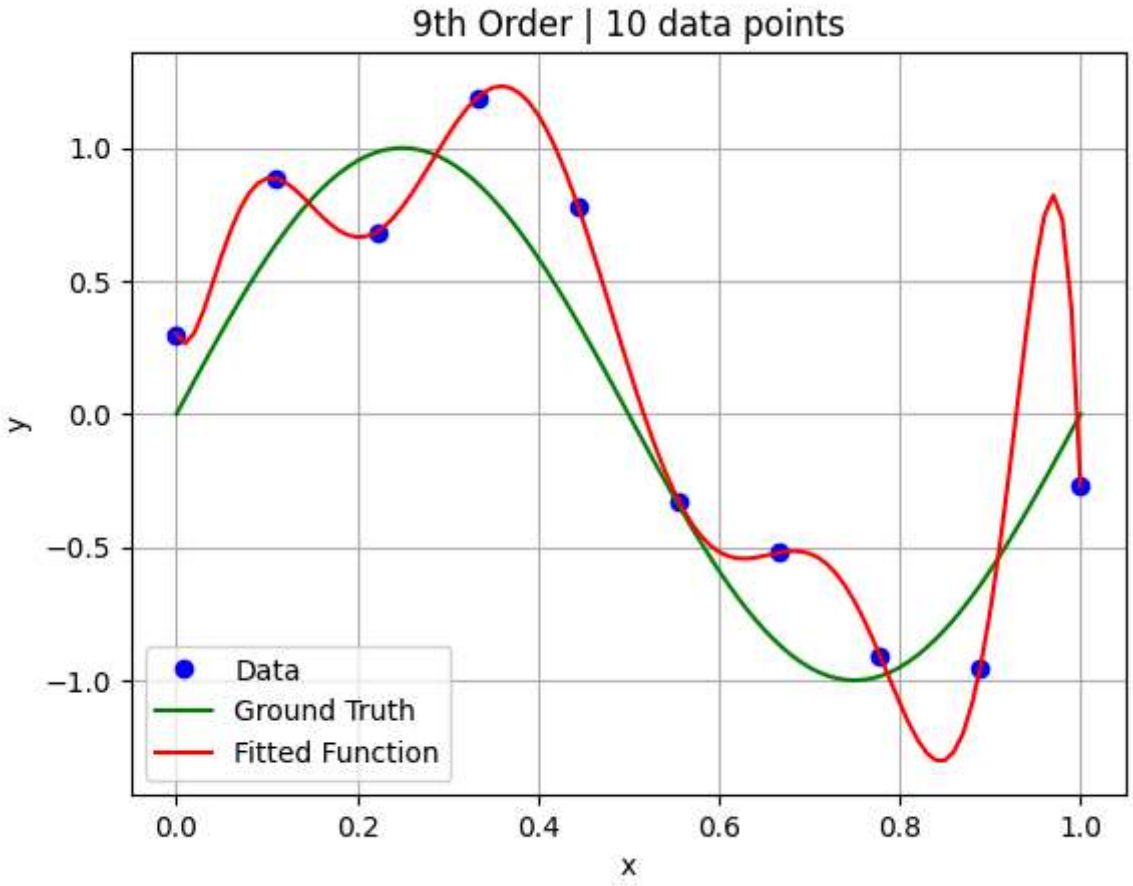
For each model:

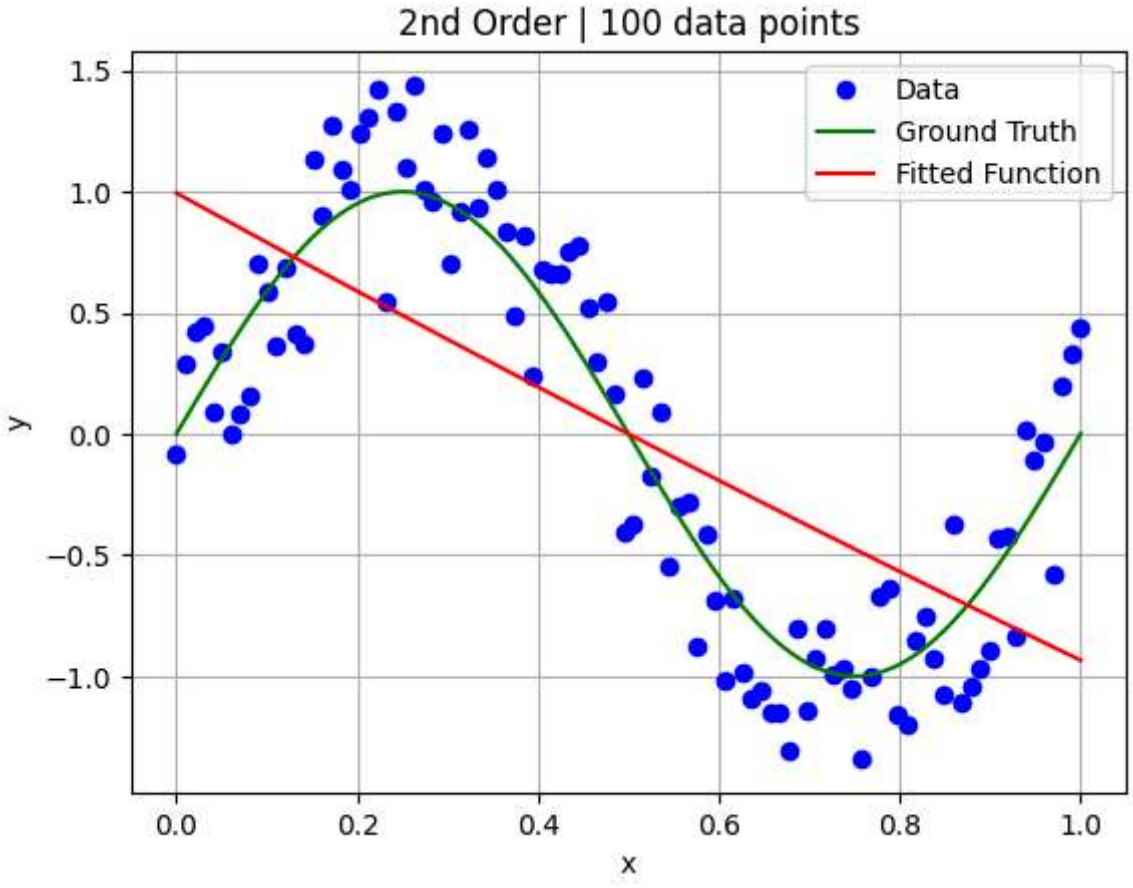
- Print the learned model parameters `w`
- Use the model parameters `w` to predict `yr` values over a range of x values given by `xr = np.linspace(0,1,101)`
- Plot the data, ground truth function, and regressed model using `plot_model(x,y,xt,yt,xr,yr,title)` with an appropriate title.

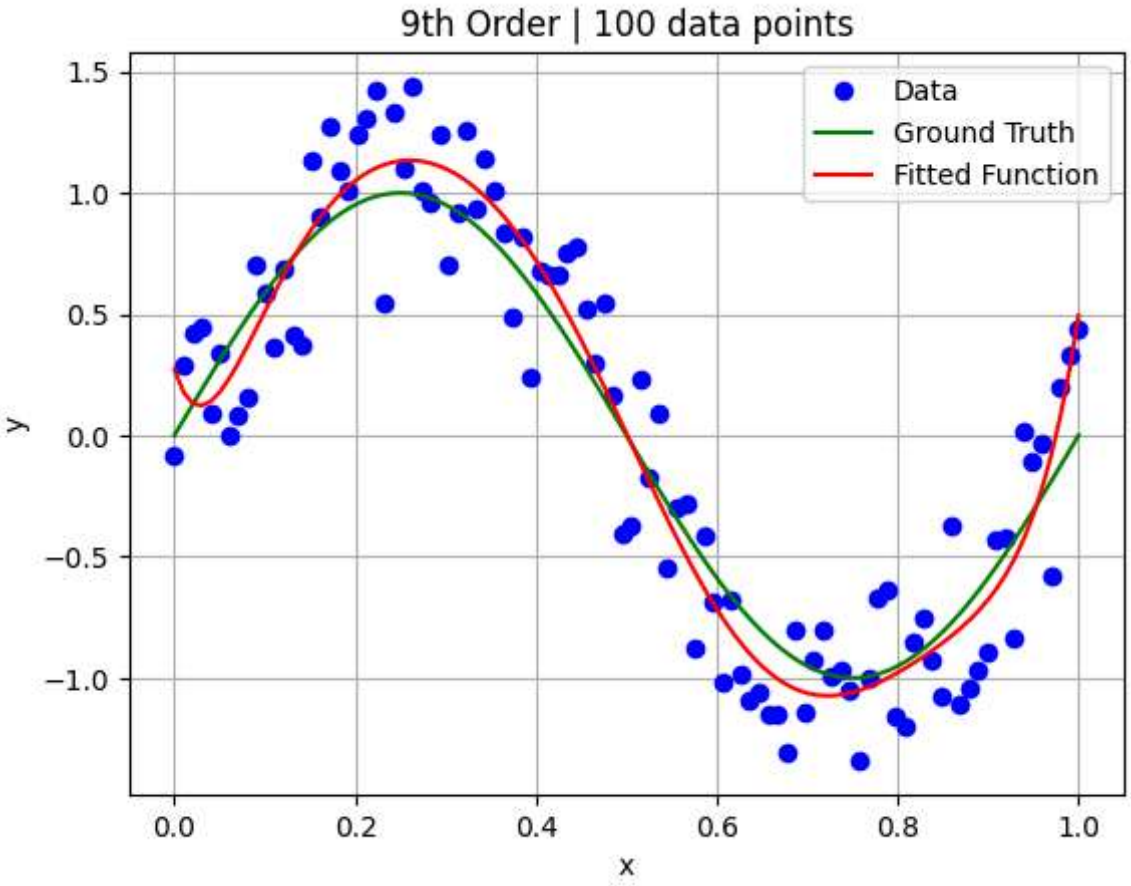
In [4]:
```
# YOUR CODE GOES HERE

# function to compute the weights given the data points and the order of the polyno
def compute_w(x, y, k):
    x_design = design_matrix(x, k)
    w = np.linalg.inv(x_design.T @ x_design) @ x_design.T @ y
    return w

# function to construct the design matrix from the data points and the order of the
def design_matrix(x, k):
    x_design = np.zeros([np.shape(x)[0], k + 1])
    for i in range(k + 1):
        x_design[:, i] = x ** i
    return x_design

# computing weights and making model predictions for 10 point data set and 2nd orde
w_data10_2 = compute_w(x10, y10, 2)
xr = np.linspace(0, 1, 101)
yr = design_matrix(xr, 2) @ w_data10_2
plot_model(x10, y10, xt, yt, xr, yr, title = '2nd Order | 10 data points')

# computing weights and making model predictions for 10 point data set and 9th orde
w_data10_9 = compute_w(x10, y10, 9)
yr = design_matrix(xr, 9) @ w_data10_9
plot_model(x10, y10, xt, yt, xr, yr, title = '9th Order | 10 data points')
```

```python
# computing weights and making model predictions for 100 point data set and 2nd ord
w_data100_2 = compute_w(x100, y100, 2)
yr = design_matrix(xr, 2) @ w_data100_2
plot_model(x100, y100, xt, yt, xr, yr, title = '2nd Order | 100 data points')

# computing weights and making model predictions for 100 point data set and 9th ord
w_data100_9 = compute_w(x100, y100, 9)
yr = design_matrix(xr, 9) @ w_data100_9
plot_model(x100, y100, xt, yt, xr, yr, title = '9th Order | 100 data points')
```




9th Order | 10 data points
1.0
0.5
0.0
−0.5
−1.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x
Data
Ground Truth
Fitted Function


2nd Order | 100 data points
1.5
1.0
0.5
0.0
−0.5
−1.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x
Data
Ground Truth
Fitted Function



Discussion:

When the sample size (number of data points) is small, what issues or tendencies do you see with complex models?

Since the number of data points is less, the model complexity allows the function too be overfit to the sample. This causes the function to be more complex than necessary, resulting in incorrect predictions.

Implement polynomial linear regression with L_2 regularization

You will repeat the previous section, but this time using L_2 regularization. Your regularization term should be $\lambda w' \mathbb{I}_m w$, where $\lambda = e^{-10}$, and $\mathbb{I}_m$ is the modified identity matrix that masks out the bias term from regularization.

You will consider only two cases:

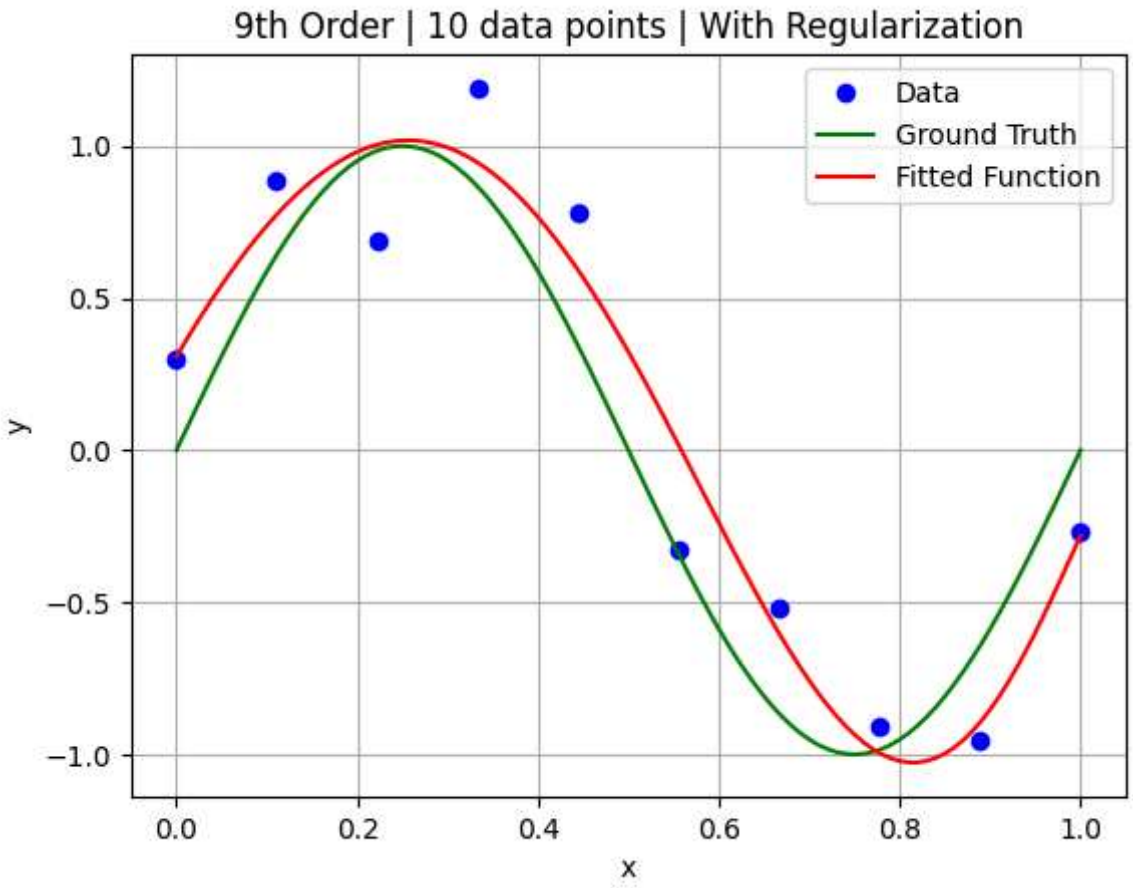
1. Data: data10.txt, Model: 9th order polynomial (highest power of x in your regression model = 9)

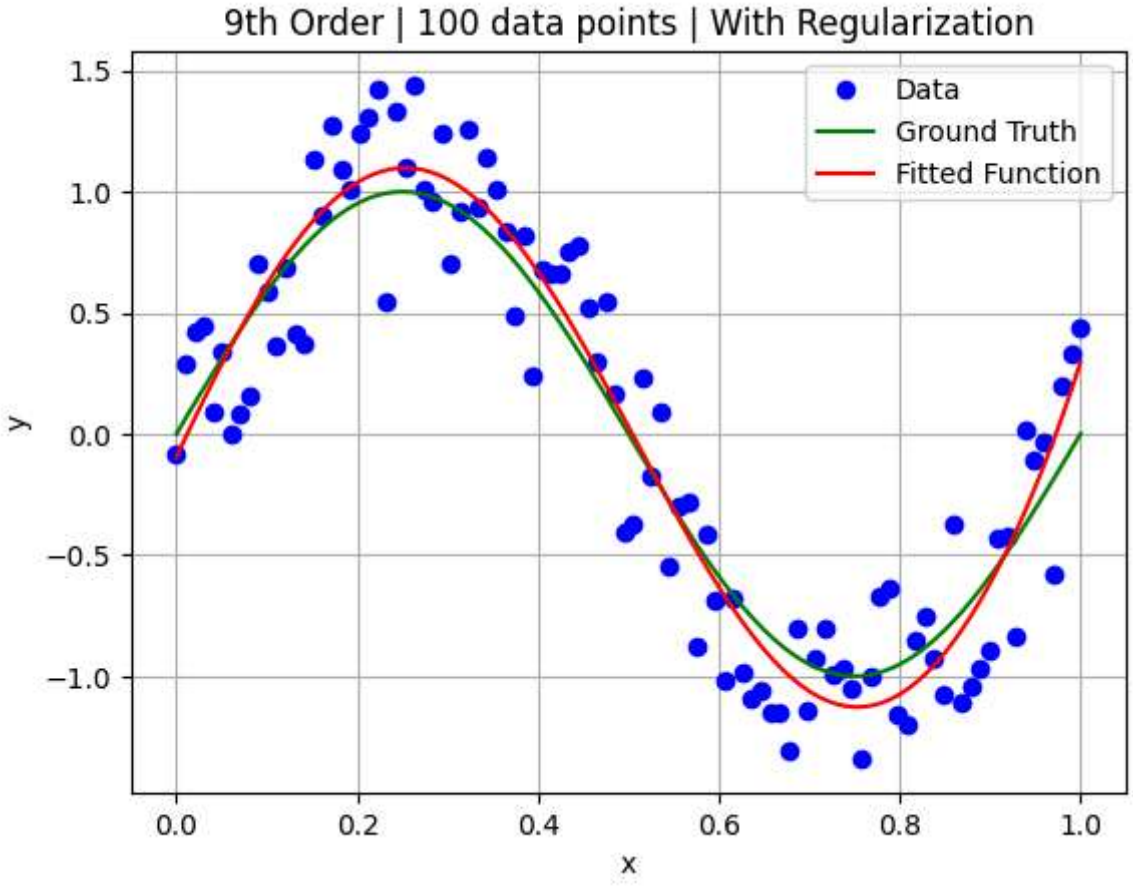
2. Data: data100.txt, Model: 9th order polynomial (highest power of x in your regression model = 9)

For each model:

- Print the learned model parameters `w`
- Use the model parameters `w` to predict `yr` values over a range of x values given by `xr = np.linspace(0,1,101)`
- Plot the data, ground truth function, and regressed model using `plot_model(x,y,xt,yt,xr,yr,title)` with an appropriate title.

In [5]:
```
# YOUR CODE GOES HERE
from math import exp

# function to compute the weights with L2 regularization given the data points and
def compute_w_reg(x, y, k, lam):
    x_design = design_matrix(x, k)
    I_prime = np.identity(np.shape(x_design)[1])
    I_prime[-1, -1] = 0

    w = np.linalg.inv(x_design.T @ x_design + lam*I_prime) @ x_design.T @ y
    return w

lam = exp(-10)

# computing weights and making model predictions for 10 point data set and 9th orde
w_data10_9_reg = compute_w_reg(x10, y10, 9, lam)
yr = design_matrix(xr, 9) @ w_data10_9_reg
plot_model(x10, y10, xt, yt, xr, yr, title = '9th Order | 10 data points | With Reg

# computing weights and making model predictions for 100 point data set and 9th ord
w_data100_9_reg = compute_w_reg(x100, y100, 9, lam)
yr = design_matrix(xr, 9) @ w_data100_9_reg
plot_model(x100, y100, xt, yt, xr, yr, title = '9th Order | 100 data points | With
```


9th Order | 10 data points | With Regularization
Data
Ground Truth
Fitted Function
1.0
0.5
0.0
−0.5
−1.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x


9th Order | 100 data points | With Regularization
Data
Ground Truth
Fitted Function
1.5
1.0
0.5
0.0
−0.5
−1.0
y
0.0
0.2
0.4
0.6
0.8
1.0
x

Discussion:

What differences between the regularized and standard 9th order models fit to `d10` do you notice? How does regularization affect the fitted function?

The regularization function does a much better job in the 10 data points case, as it penalizes model complexity and reduces overfitting to small fluctuations etc.

LLS with L_2 regularization and gradient descent

For complex models, the size of $X'X$ can be large, making matrix inversion computationally demanding. Instead, one can use gradient descent to compute w. In our notes, we derived the gradient descent approach both for unregularized as well as L_2 regularized linear regression. The formula for the gradient descent approach with L_2 regularization is:

$$\frac{\partial obj}{\partial w} = X'Xw - X'y + \lambda \mathbb{I}_m w$$

$$w^{new} \leftarrow w^{cur} - \alpha \frac{\partial obj}{\partial w}$$

In this problem, could gradient descent get stuck in a local minimum? Explain why / why not?

Your answer goes here

You will consider just a single case in the following question:

1. Data: data10.txt, Model: 9th order polynomial.

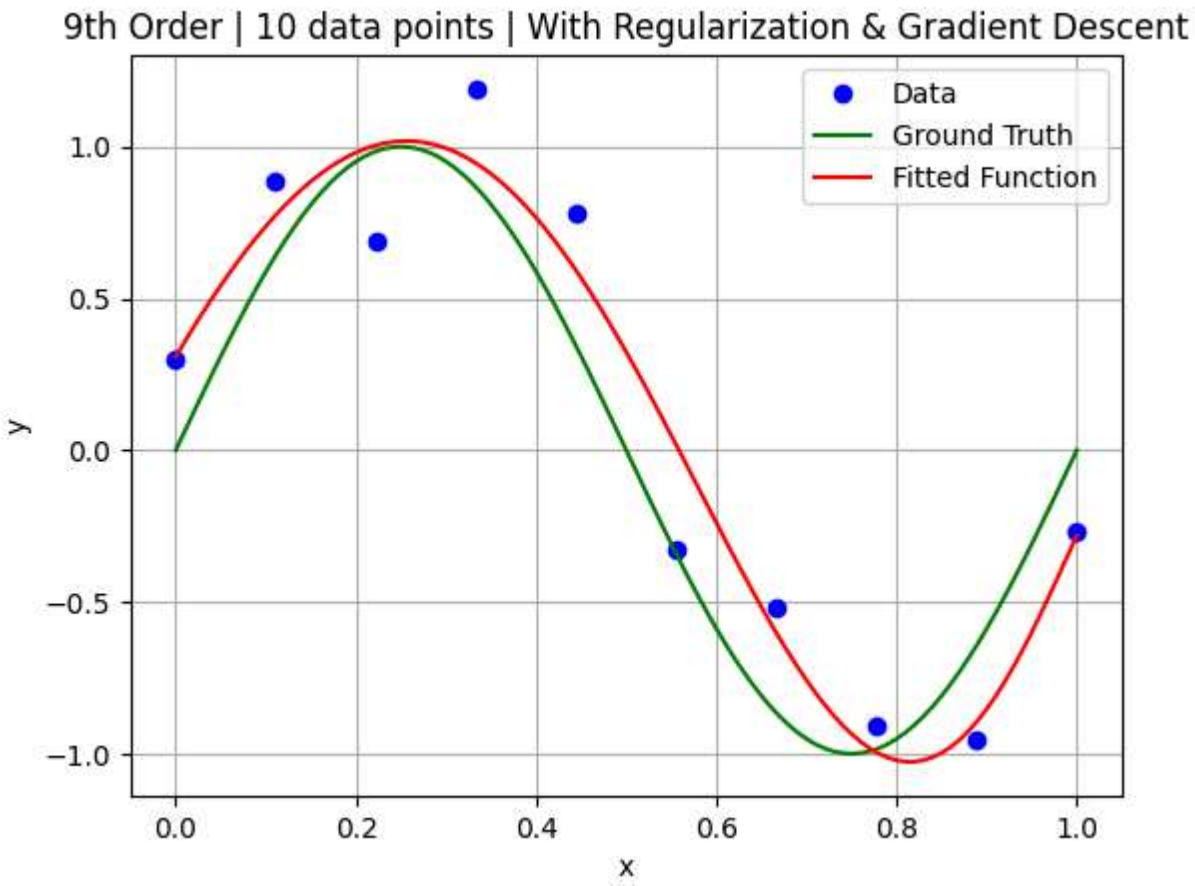
Starting with a weight vector of zeros as the initial guess, and $\lambda = e^{-10}$, $\alpha = 0.075$, apply 50000 iterations of gradient descent to find the optimal model parameters. In practice, when you train your own models you will have to determine these parameters yourself!

For the trained model:

- Print the learned model parameters `w`
- Use the model parameters `w` to predict `yr` values over a range of x values given by `xr = np.linspace(0,1,101)`
- Plot the data, ground truth function, and regressed model using `plot_model(x,y,xt,yt,xr,yr,title)` with an appropriate title.

In [10]:
```
# YOUR CODE GOES HERE
from math import exp

# function to execute gradient descent to arrive to a better estimation of the weig
def grad_desc(w, x_design, yr, alpha, lam, iter):
```

```
    Im = np.identity(np.shape(w)[0])
    Im[-1, -1] = 0
    for i in range(iter):
        grad = x_design.T @ (x_design @ w - yr) + lam * Im @ w
        w_new = w - alpha * grad
    return w_new

w = compute_w_reg(x10, y10, 9, lam)
alpha = 0.075
iter = 50000
x_design = design_matrix(xr, 9)
yr = x_design @ w

w = grad_desc(w, x_design, yr, alpha, lam, iter)

plot_model(x10, y10, xt, yt, xr, yr, title = '9th Order | 10 data points | With Reg
```



Discussion:

Visually compare the result you just obtained to the same 9th order polynomial model with L_2 regularization where you solved for `w` directly in the previous section. They should be very similar. Comment on whether gradient descent has converged.

Visually, there's not much of a difference between the plots with and without gradient descent. The gradient descent does not seem to have converged, especially close to the final data point.

Porblem 2 (30 points)

Problem Description

In this problem you will use linear least squares to fit a linear function to a 3D temperature field, with x,y,z locations and an associated temperature T.

Fill out the notebook as instructed, making the requested plots and printing necessary values.

You are welcome to use any of the code provided in the lecture activities.

Summary of deliverables:

Results:

- Predicted temperature `T(5,5,5)` using a hand-coded LLS squares model with a linear function
- Direction of travel from `(5,5,5)` to experience the greatest decrease in temperature

Discussion:

- Reasoning for how we can use our fitted function to determine the direction of greatest decrease in temperature

Imports and Utility Functions:

In [1]:
```
import numpy as np
```

Load the data

The data is contained in `tempfield.npy` and can be loaded with `np.load(tempfield.npy)`. The first three columns correspond to the `x`, `y`, and `z` locations of the data points, and the 4th column corresponds to the temperature `T` at the respective point. Store the data as you see fit.

In [2]:
```
# YORU CODE GOES HERE
data = np.load('tempfield.npy')
```

LLS Regression in 3D

Now fit a linear function to the data using the closed for of LLS regression. Use your fitted function to report the predicted temperature at `x = 5, y = 5, z = 5`. You are free to

add regularization to your model, but this is not required and will not be graded.

In [3]:
```python
# YOUR CODE GOES HERE

# function to construct design matrix given the data points
def design_matrix(x):
    x_design = np.zeros([np.shape(x)[0], np.shape(x)[1] + 1])
    x_design[:, :-1] = x_design[:, :-1] + x
    x_design[:, -1] = np.ones(np.shape(x)[0])
    return x_design

# function to compute the weights for the different features
def compute_w(data):
    y = np.zeros([np.shape(data)[0], 1])
    x_design = design_matrix(data[:, :-1])
    y = data[:, -1].reshape(-1, 1)
    w = np.linalg.inv(x_design.T @ x_design) @ x_design.T @ y
    return w

# function to make a prediction at a given data point
def preds(x, data):
    w = compute_w(data)
    x_design = design_matrix(x)
    predict = x_design @ w
    return predict

# query point
x = 5 * np.ones([1, 3])

# prediction at query point
print("Prediction: ", preds(x, data))
```

```
Prediction:  [[21.38907338]]
```

Gradient Intuition

Using the function you fit in the previous part, in which direction should one move from the point `p = (5,5,5)` to experience the largest decrease in temperature in the immediate neighborhood of the point? Report the specific direction, along with your reasoning.

Looking at the data, it seems like taking a step down in the 'y' value has a significant effect on the value of the temperature reading

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sympy import symbols, solve

data = pd.read_csv('m02_bonus.csv')
x = data['x'].values
y = data['y'].values
sigma = data['sigma'].values
```

In [3]:
```
X = np.column_stack([x, y])
model = LinearRegression().fit(X, sigma)

print('Fitted equation: x * (', model.coef_[0], ') + y * (', model.coef_[1], ') + (
```

```
Fitted equation: x * ( 0.13851116932147264 ) + y * ( -0.22081029743649447 ) + ( 9.08
1655840821126 ) = sigma
```

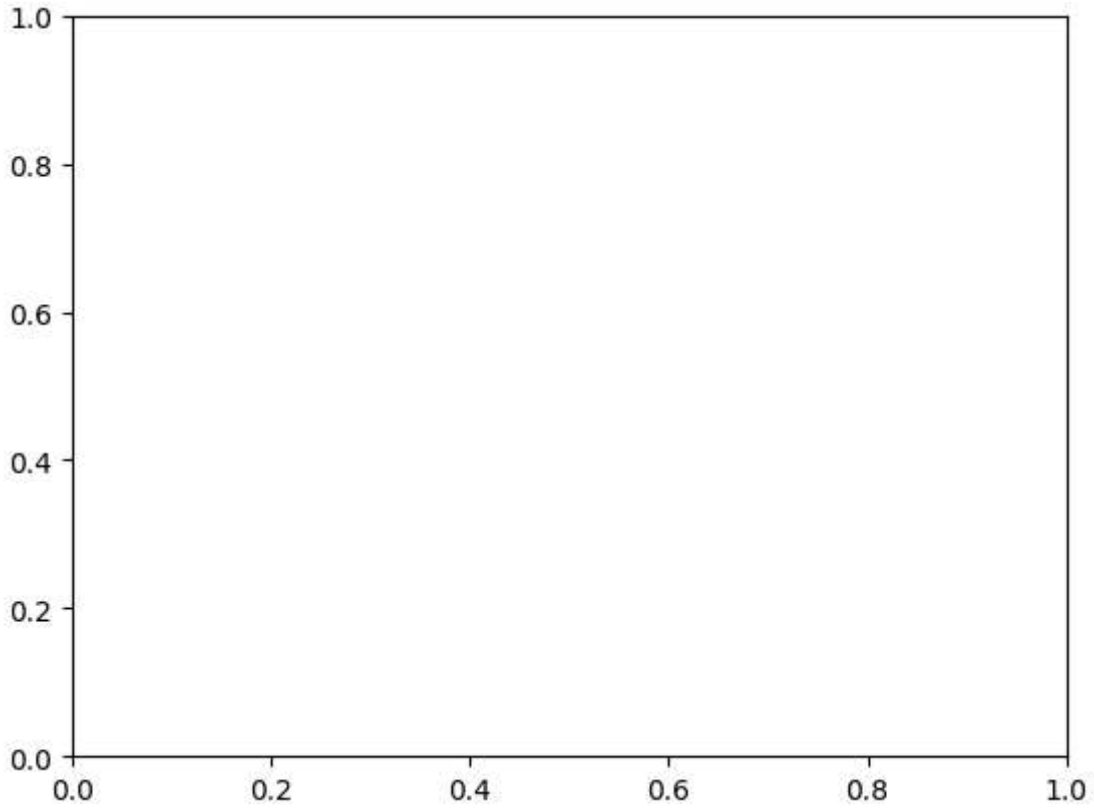
In [4]:
```
# using two data points to infer the values of sigma, R

# for x[0], y[0], calculating r and theta, we get
r0 = np.sqrt(x[0] ** 2 + y[0] ** 2)
theta0 = np.arctan2(y[0], x[0])

# for x[1], y[1]
r1 = np.sqrt(x[0] ** 2 + y[0] ** 2)
theta1 = np.arctan2(y[1], x[1])

R = symbols('R')

expr = sigma[1] * (1 - R ** 2 / r1 ** 2 + np.cos(2 * theta1) * (1 - 4 * R ** 2 / r1
R_sol = solve(expr)
R_vals = np.array([R_sol[2], R_sol[3]])

sigma_sol = sigma[0] * 2 / (1 - np.power(R_vals, 2) / r1 ** 2 + np.cos(2 * theta1)
print('R: ', R_vals.T, '\nsigma_0: ', sigma_sol.T)
```

```
R:  [4.63643150995978 6.09582686727525]
sigma_0:  [-0.137888698292661 -0.0450673227364446]
```

In [32]:
```
# for R = 4.64
x = np.linspace(-5 * 4.64, 5 * 4.64, 101).reshape(-1, 1)
y = np.linspace(-5 * 4.64, 5 * 4.64, 101).reshape(-1, 1)
X = np.hstack([x, y])

preds = np.zeros([len(x), len(y)])
preds = model.predict(X)

predictions = np.meshgrid(preds)
plt.pcolormesh(X, predictions)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[32], line 10
      7 preds = model.predict(X)
      9 predictions = np.meshgrid(preds)
---> 10 plt.pcolormesh(X, predictions)

File c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\matplo
tlib\pyplot.py:3697, in pcolormesh(alpha, norm, cmap, vmin, vmax, shading, antialias
ed, data, *args, **kwargs)
   3684 @_copy_docstring_and_deprecators(Axes.pcolormesh)
   3685 def pcolormesh(
   3686     *args: ArrayLike,
   (...)
   3695     **kwargs,
   3696 ) -> QuadMesh:
-> 3697     __ret = gca().pcolormesh(
   3698         *args,
   3699         alpha=alpha,
   3700         norm=norm,
   3701         cmap=cmap,
   3702         vmin=vmin,
   3703         vmax=vmax,
   3704         shading=shading,
   3705         antialiased=antialiased,
   3706         **({"data": data} if data is not None else {}),
   3707         **kwargs,
   3708     )
   3709     sci(__ret)
   3710     return __ret

File c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\matplo
tlib\__init__.py:1473, in _preprocess_data.<locals>.inner(ax, data, *args, **kwargs)
   1470 @functools.wraps(func)
   1471 def inner(ax, *args, data=None, **kwargs):
   1472     if data is None:
-> 1473         return func(
   1474             ax,
   1475             *map(sanitize_sequence, args),
   1476             **{k: sanitize_sequence(v) for k, v in kwargs.items()})
   1478     bound = new_sig.bind(ax, *args, **kwargs)
   1479     auto_label = (bound.arguments.get(label_namer)
   1480                   or bound.kwargs.get(label_namer))

File c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\matplo
tlib\axes\_axes.py:6428, in Axes.pcolormesh(self, alpha, norm, cmap, vmin, vmax, sha
ding, antialiased, *args, **kwargs)
   6425 shading = shading.lower()
   6426 kwargs.setdefault('edgecolors', 'none')
-> 6428 X, Y, C, shading = self._pcolorargs('pcolormesh', *args,
   6429                                      shading=shading, kwargs=kwargs)
   6430 coords = np.stack([X, Y], axis=-1)
   6432 kwargs.setdefault('snap', mpl.rcParams['pcolormesh.snap'])

File c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\matplo
tlib\axes\_axes.py:5953, in Axes._pcolorargs(self, funcname, shading, *args, **kwarg
```

```
s)
   5951     nrows, ncols = C.shape[:2]
   5952 else:
-> 5953     raise _api.nargs_error(funcname, takes="1 or 3", given=len(args))
   5955 Nx = X.shape[-1]
   5956 Ny = Y.shape[0]

TypeError: pcolormesh() takes 1 or 3 positional arguments but 2 were given
```



In [ ]: