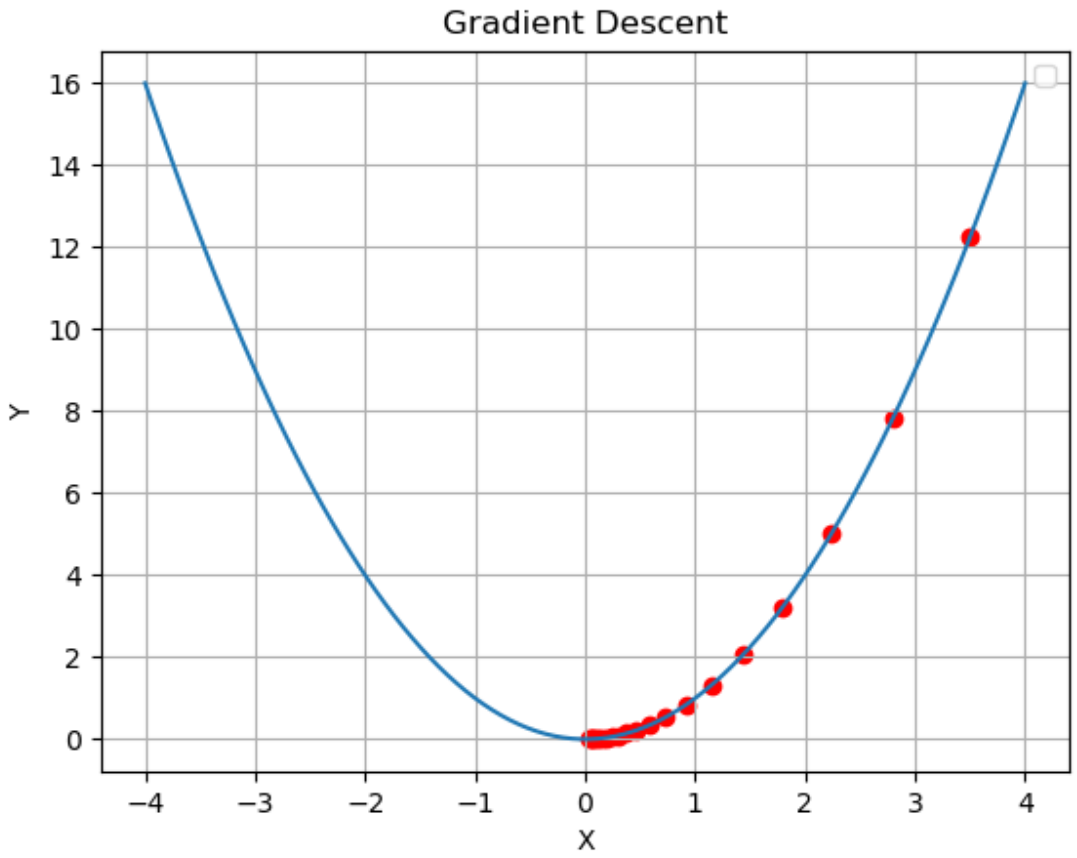
Nicholas Yegon
Matrikel Number: 1748461
Exercise 04

1. **Implement Vanilla Gradient Descent (Not parallel)**

Implement the algorithm from slide 8 As function use f (x) := x2 with ∇f (x) := 2x Run the algorithm for 20 iterations with learning rate η := 0.1 and starting point 3.5 Plot the function in the interval [−4, 4] and overlay the point that x visits in the 20 epochs The result should look like slide 9

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


def square_fn(x):
   return x**2


def gradient(x):
   return 2*x


def gradient_descent(square_fn, gradient, x_init, learning_rate, epsilon,
max_iterations):
   x = x_init
   x_values = [x]

   for i in range(max_iterations):
       gradient_x = gradient(x)
       x_next = x - learning_rate * gradient_x

       if np.abs(x_next - x) < epsilon:
           break

       x = x_next
       x_values.append(x)

   return x, x_values
```

```
learning_rate = 0.1
x_init = 3.5
num_iterations = 20
epsilon = 1e-6

x_optimal, x_values = gradient_descent(
   square_fn, gradient, x_init, learning_rate, epsilon,
max_iterations=num_iterations)

x_axis = np.linspace(-4, 4, 100)
y_axis = square_fn(x_axis)
plt.plot(x_axis, y_axis)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Gradient Descent')
plt.grid(True)


x_values_vals = np.array(x_values)
y_history_vals = square_fn(x_values_vals)
plt.scatter(x_values_vals, y_history_vals, c='red')
plt.legend()
plt.show()
```



2. Ramp Up

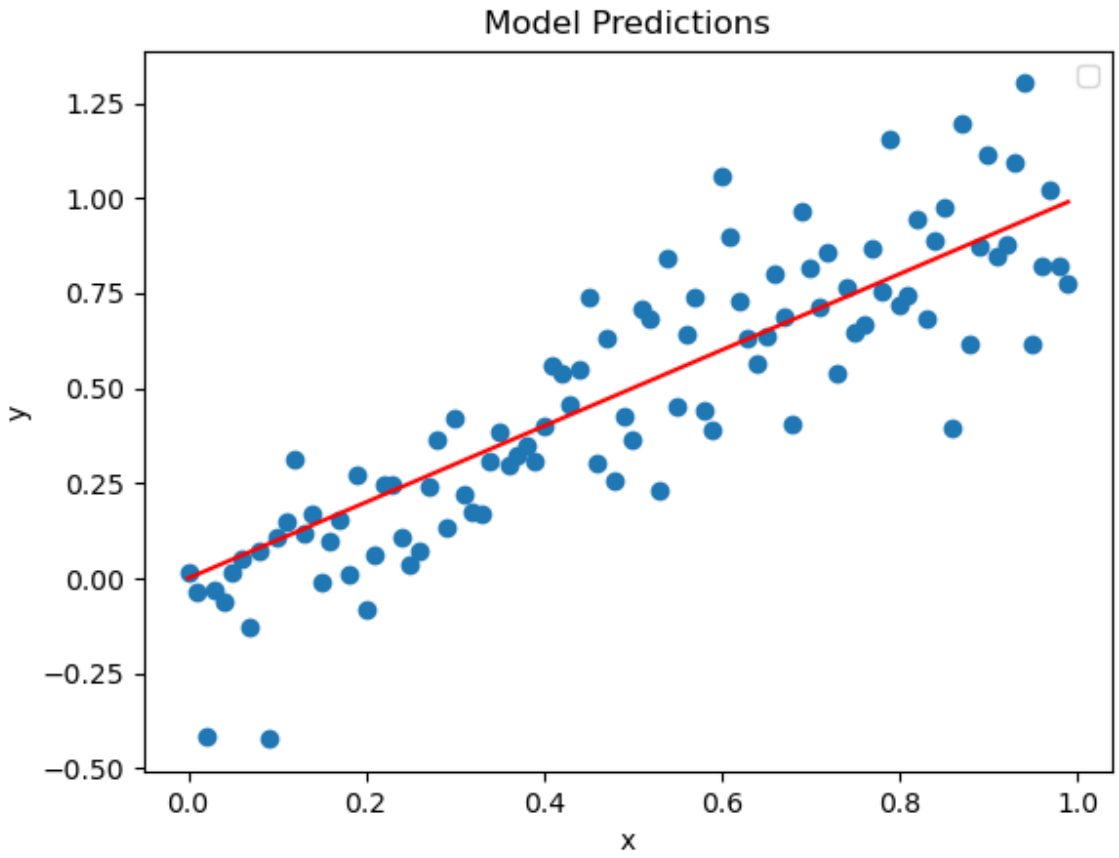
Generate data with the following code:

```
X = np . arange (0 , 1 , 0.01)
Y = X + np . random . normal (0 , 0.2 , len ( X ) )
```

Write a python function `y hat(x, theta1, theta2)` that represents your model. Plot the data with `plt.scatter` and plot the models predictions with `plt.plot` using `θ1 = 1.0` and `θ2 = 0.0`. (These are the optimal parameters)

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

def y_hat(x, theta1, theta2):
    return theta1 * x + theta2

# Generate data
X = np.arange(0, 1, 0.01)
```

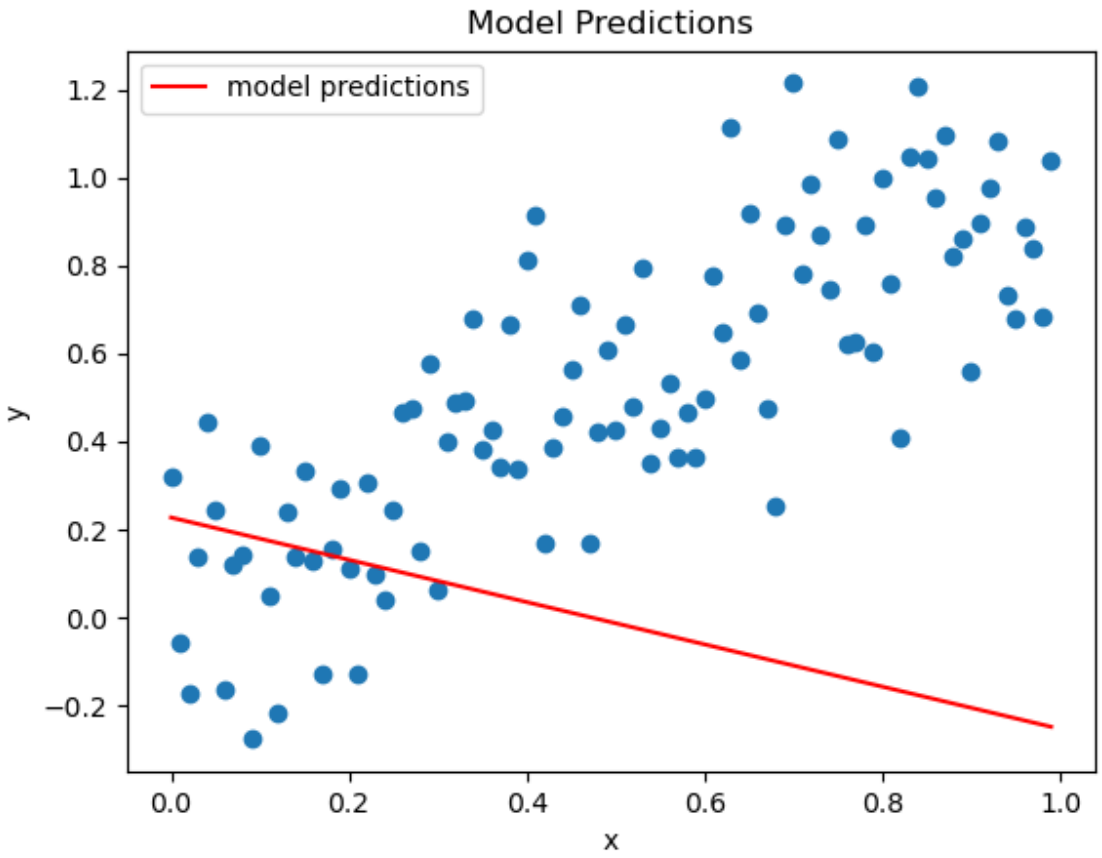
```
Y = X + np.random.normal(0, 0.2, len(X))

# Plot the data
plt.scatter(X, Y)

# Plot the model's predictions
theta1 = 1.0
theta2 = 0.0
predictions = y_hat(X, theta1, theta2)
plt.plot(X, predictions, color='red')

plt.xlabel('x')
plt.ylabel('y')
plt.title('Model Predictions')
plt.legend()
plt.show()
```



Implement the algorithm from slide 13(non-parallel)
Initialize your parameters to θ1 = −0.5 and θ2 = 0.2, set the learning rate η = 0.01 and run for Emax = 5 iterations.

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

def y_hat(x, theta1, theta2):
    return theta1*x+theta2

#generate data
X = np.arange(0, 1, 0.01)
Y = X + np.random.normal(0, 0.2, len(X))

# Initialize parameters
theta1 = -0.5
theta2 = 0.2
```

```
# set learning rate and max iterations
learning_rate = 0.01
Emax = 5



# perform gradient descent
for i in range(Emax):

#    compute gradients
   grad_theta1 = np.mean((y_hat(X, theta1, theta2)-Y)*X)
   grad_theta2 = np.mean(y_hat(X, theta1, theta2)-Y)

   #Update params
   theta1 -= learning_rate * grad_theta1
   theta2 -= learning_rate * grad_theta2

# plot the data
plt.scatter(X,Y)
#  plot the final model's predictions
predictions = y_hat(X, theta1, theta2)
plt.plot(X, predictions, color='red', label='model predictions')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Model Predictions')
plt.legend()
plt.show()
```



3. Distributed Gradient Descent

Implement a distributed version of gradient descent by following slide 15
Test your code with different numbers of ranks and experiment with the amount of iterations needed to approach the optimal parameters θ1 = 1.0 and θ2 = 0.0.

```
import numpy as np
import matplotlib.pyplot as plt
from mpi4py import MPI

def y_hat(x, theta1, theta2):
    return theta1 * x + theta2

# Generate data
X = np.arange(0, 1, 0.01)
Y = X + np.random.normal(0, 0.2, len(X))

# Set the desired optimal parameters
```

```
desiredParam1 = 1.0
desiredParam2 = 0.0

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
ranksSize = comm.Get_size()

# Partition the data
samples = len(X) // ranksSize
start= rank * samples
end = start+ samples
varX = X[start:end]
varY = Y[start:end]

# Initialize parameters
theta1 = -0.5
theta2 = 0.2

# Set learning rate and max iterations
learning_rate = 0.01
Emax = 1000  # Increase the max iterations

# Perform gradient descent
for i in range(Emax):

   # Compute gradients locally
   gradientTheta1 = np.mean((y_hat(varX, theta1, theta2) - varY) * varX)
   gradientTheta2 = np.mean(y_hat(varX, theta1, theta2) - varY)

   # Sum gradients across ranks
   gradientTheta1_sum = comm.allreduce(gradientTheta1, op=MPI.SUM)
   gradientTheta2_sum = comm.allreduce(gradientTheta2, op=MPI.SUM)

   # Update parameters
   theta1 -= learning_rate * gradientTheta1_sum
   theta2 -= learning_rate * gradientTheta2_sum

   # Share the updated parameters with ranks
   comm.Bcast([theta1, MPI.DOUBLE], root=0)
```

```
    comm.Bcast([theta2, MPI.DOUBLE], root=0)

    # Check convergence
    if np.abs(theta1 - desiredParam1) < 1e-6 and np.abs(theta2 -
desiredParam2) < 1e-6:
        break

# Gather the final model parameters to rank 0
finalTheta1 = comm.gather(theta1, root=0)
finalTheta2 = comm.gather(theta2, root=0)

# Plot the data
if rank == 0:
    plt.scatter(X, Y)

    # Plot the final model's predictions
    if len(finalTheta1) > 0 and len(finalTheta2) > 0:
        lastTheta1 = finalTheta1[-1]
        lastTheta2 = finalTheta2[-1]
        predictions = y_hat(X, lastTheta1, lastTheta2)
```

Do the needed iterations change with the amount of workers change? Yes, it is possible the iterations can change with the number of workers
Repeat this experiment with different model initializations