Objectives

```
Understand the difference between closed-form and iterative
optimization
Implement basic gradient descent by hand and in Python
Visualize how optimization algorithms converge to minima
```

Part 1: Hand Calculations (50 points) Problem 1.1: Closed-Form Solution (10 points)

Find the minimum of $f(x) = x^2 - 4x + 7$ using calculus (closed-form solution).

Hints:

```
Find f'(x) and solve f'(x) = 0
```

Problem 1.2: Gradient Descent by Hand (20 points)

Use gradient descent to find the minimum of the same function $f(x) = x^2 - 4x + 7$.

Starting from $x_0 = 0$ with learning rate $\alpha = 0.1$:

```
Perform 5 iterations of gradient descent by hand
Create a table showing: iteration, x_n, f'(x_n), f(x_n)
How close did you get to the true minimum after 5 iterations?
The sign of f'(x_n) tells the direction to move. Why is the magnitude
of the value of f'(x_n) used as the step size?
```

Hints:

```
Use the update rule: x_{n+1} = x_n - α × f'(x_n)
```

Problem 1.3: Introducing Cost Functions - Fence Optimization (20 points)

A farmer wants to build a rectangular fence to enclose exactly 1 acre of land (43,560 ft^2). Fencing costs $8 per foot. The farmer wants to minimize the total fencing cost.

Since the area must be exactly 1 acre, if we choose the length L in feet, then the width W is determined by: W = 43,560/L Derive the Cost Function

```
Write an expression for the perimeter of the rectangle in terms of
length L only
Write the cost function C(L) that gives the total fencing cost in
dollars as a function of length L
What is the domain of this function? (What range of values of L make
sense?)
```

Find the Minimum Cost (Closed-Form)

```
Find C'(L) and solve C'(L) = 0 to find the optimal length
What is the corresponding optimal width?
What is the minimum total cost?
What do you notice about the relationship between the optimal length
and width?
```

Note for Students: This introduces the idea of a cost or loss function that we want to minimize. Here, our cost function C(L) represents the total expense we want to minimize, subject to the constraint that we must enclose exactly 1 acre. In machine learning, we'll minimize functions that represent how "bad" our model's predictions are. As the model learns, the predictions will get less bad. Part 2: Python Implementation (50 points) Problem 2.1: Implement Gradient Descent (25 points)

For the following problem, use the given base function stump and extend it to accomplish the required functionality. Do not change the inputs or the return statement.

Don't use an LLM! Homework is graded based on a good faith effort (not for correctness) so there's no value to getting a perfect answer from an LLM. The goal of this is for you to gain personal understanding of how to code gradient descent. Using an LLM will hurt you because you will be expected to write this code (or very similar) on a test/quiz.

def function1(x): return x**2 -4*x + 7

def derivative1(x): return ...

def gradient_descent(f, df, x_start, learning_rate, num_iterations): """ f: function to minimize df: derivative of f x_start: starting point learning_rate: step size num_iterations: number of iterations to run

```
Returns: (x_history, f_history) - lists of x values and f(x) values
"""
# Your implementation here
# delete this and the following line in your implementation. (do not
delete the return)
pass
return x_history, f_history
```

Example call of gradient descent.

gradient_descent(function1, derivative1, 0.01, 100)

Test your function on $f(x) = x^2 - 4x + 7$ with:

```
Starting point: x₀ = 0
Learning rate: α = 0.1
50 iterations
```

Part 3: Analysis and Visualization (20 points) Problem 3.1: Convergence Visualization

Create two plots using matplotlib:

```
Function Plot: Plot the cost function of the fence from question 1.3
for lengths from l = 100 to l = 21500, with the minimum point clearly
marked
Convergence Plot: For α = 0.1, use your gradient descent function to
show how x_n approaches the minimum over 1000 iterations with a
convergence plot (x-axis: iteration, y-axis: cost).
How many iterations did it take to get within 0.01 of the true minimum
with α = 0.1?
```

Problem 3.2: Reflection Questions

```
The closed form derivative optimization solution is obviously much
faster than an iterative approach. When might iterative methods be
necessary?
What role does the learning rate play in convergence speed and
stability?
```

Submission Requirements

```
A single PDF containing
    Hand calculations (can be scanned/photographed if handwritten)
    Answers to any questions
    Python code
    Plots (save as .png files)

%pip install uv --quiet
%uv pip install numpy pandas matplotlib --quiet

Note: you may need to restart the kernel to use updated packages.
Note: you may need to restart the kernel to use updated packages.

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

f(x_n+1) = f(x_n) - (learning_rate * f'(x_n))

for n in len(range(num_iterations))

```
def function1(x):
    return x**2 -4*x + 7

def derivative1(x):
    return 2*x - 4

def gradient_descent(f, df, x_start, learning_rate, num_iterations):
    """
    f: function to minimize
    df: derivative of f
    x_start: starting point
    learning_rate: step size
    num_iterations: number of iterations to run

    Returns: (x_history, f_history) - lists of x values and f(x)
values
    """

    x = x_start
    x_history = [x]
    f_history = [f(x)]
    for n in range(num_iterations):
        x = x - learning_rate * df(x)
        x_history.append(x)
        f_history.append(f(x))


    return x_history, f_history

results = gradient_descent(function1, derivative1, 0, 0.10, 5)
print(results)
```

```
([0, 0.4, 0.7200000000000001, 0.976, 1.1808, 1.34464], [7,
5.5600000000000005, 4.6384, 4.048576000000001, 3.67108864,
3.4294967296])
```

Problem 2.2: Explore Different Learning Rates (25 points)

Using your gradient_descent function, test the following learning rates on function 2:

```
function2(x) = (x-1)² - 10x + 3
α = 0.01, 0.1, 0.5, 0.9
```

For each learning rate:

```
Run for 100 iterations
Plot the convergence using matplotlib (x-axis: iteration, y-axis: f(x)
value)
    Matplotlib quick reference here
Report the final x value and f(x) value
```

Questions to answer:

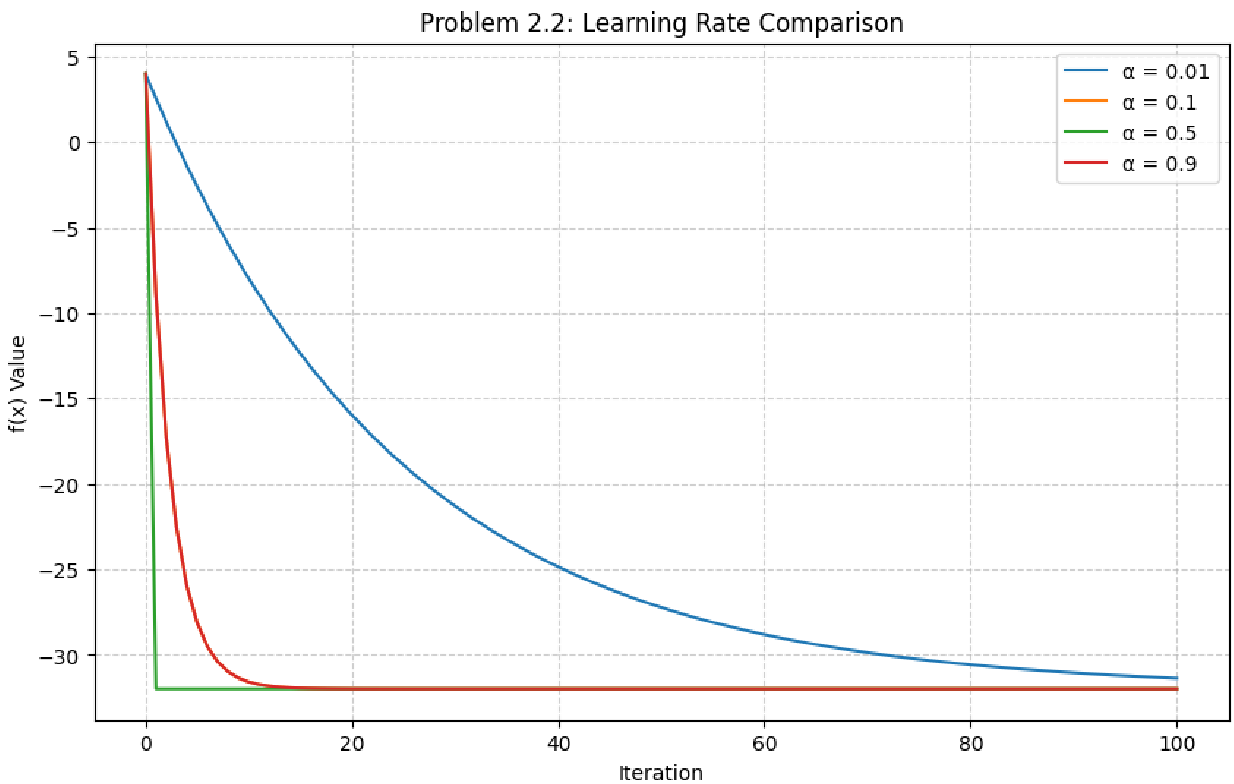
```
Which learning rate converges fastest?
What happens with α = 0.9? Why?
What would you expect with α = 1.1? (Don't implement, just reason
about it)
How many iterations did it take to get within 0.01 of the true minimum
with α = 0.1
```

Hints:

```
Expand and simplify function 2 to find its derivative more easily.

def function2(x):
    return (x-1)**2 - 10*x + 3

def derivative2(x):
    return 2*x -12

def initialize_convergence_plot(title="Optimization Convergence"):
    """Initializes a matplotlib figure for tracking cost over
iterations."""
    plt.figure(figsize=(10, 6))
    plt.title(title)
    plt.xlabel('Iteration')
    plt.ylabel('f(x) Value')
    plt.grid(True, linestyle='--', alpha=0.6)

def finalize_and_show_plot(filename="convergence_plot.png"):
    """Adds legend, saves the figure, and displays it."""
    plt.legend()
    plt.savefig(filename)
    plt.show()

initialize_convergence_plot(title="Problem 2.2: Learning Rate
Comparison")

a = [0.01, 0.1, 0.5, 0.9]
for i in range(len(a)):
    results = gradient_descent(function2, derivative2, 0, a[i], 100)
    x_history, f_history = results

    print(f"{a[i]} learning rate. Final x: {x_history[-1]:.4f}, Final
```

```
f(x): {f_history[-1]:.4f}")

    plt.plot(f_history, label=f'α = {a[i]}')

finalize_and_show_plot("learning_rate_analysis.png")
```

```
0.01 learning rate. Final x: 5.2043, Final f(x): -31.3668
0.1 learning rate. Final x: 6.0000, Final f(x): -32.0000
0.5 learning rate. Final x: 6.0000, Final f(x): -32.0000
0.9 learning rate. Final x: 6.0000, Final f(x): -32.0000
```



Questions to answer:

```
Which learning rate converges fastest?
    We can see an instantaneous snap to 6.0 from alpha=0.5.
What happens with α = 0.9? Why?
    It's jumping past the correct value either size. With such a large
training rate, there is too much chance of jumping past the correct
answer with the modificaitons to x.
What would you expect with α = 1.1? (Don't implement, just reason
about it)
    It would never converge since it would always just balloon the x
value like crazy and it would be worthless.
How many iterations did it take to get within 0.01 of the true minimum
```

```
with α = 0.1
    30 epochs, a very slow rate compared to the anomaly of 0.5
```

```
len([0, 1.2000000000000002, 2.16, 2.928, 3.5423999999999998, 4.03392,
4.427136, 4.7417088, 4.99336704, 5.194693632, 5.3557549056,
5.48460392448, 5.587683139584, 5.6701465116672, 5.73611720933376,
5.788893767467008, 5.831115013973607, 5.864892011178886,
5.891913608943108, 5.9135308871544865, 5.9308247097235895,
5.944659767778871, 5.955727814223097, 5.964582251378478,
5.971665801102782, 5.977332640882226, 5.981866112705781,
5.9854928901646245, 5.9883943121317, 5.99071544970536])
```

```
30
```

Part 3: Analysis and Visualization (20 points) Problem 3.1: Convergence Visualization

Create two plots using matplotlib:

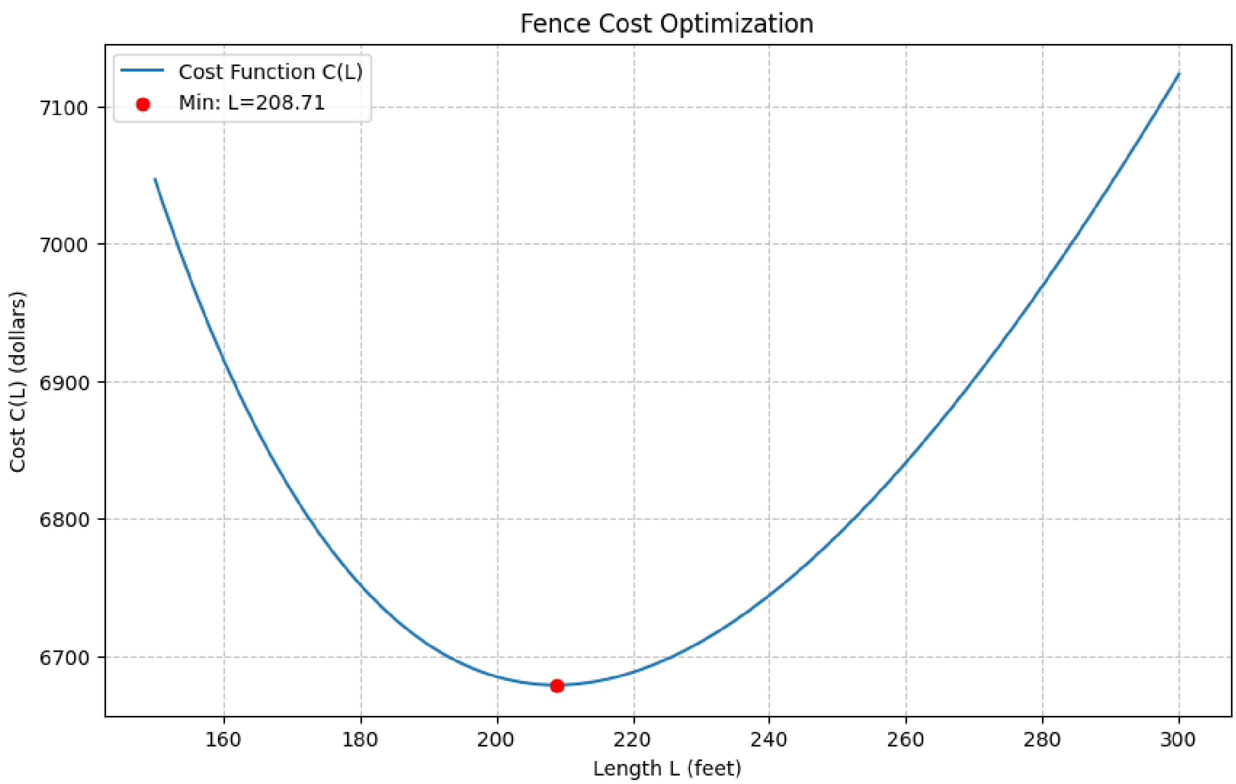
```
Function Plot: Plot the cost function of the fence from question 1.3
for lengths from l = 100 to l = 21500, with the minimum point clearly
marked
Convergence Plot: For α = 0.1, use your gradient descent function to
show how x_n approaches the minimum over 1000 iterations with a
convergence plot (x-axis: iteration, y-axis: cost).
How many iterations did it take to get within 0.01 of the true minimum
with α = 0.1?
```

Problem 3.2: Reflection Questions

```
The closed form derivative optimization solution is obviously much
faster than an iterative approach. When might iterative methods be
necessary?
What role does the learning rate play in convergence speed and
stability?
```

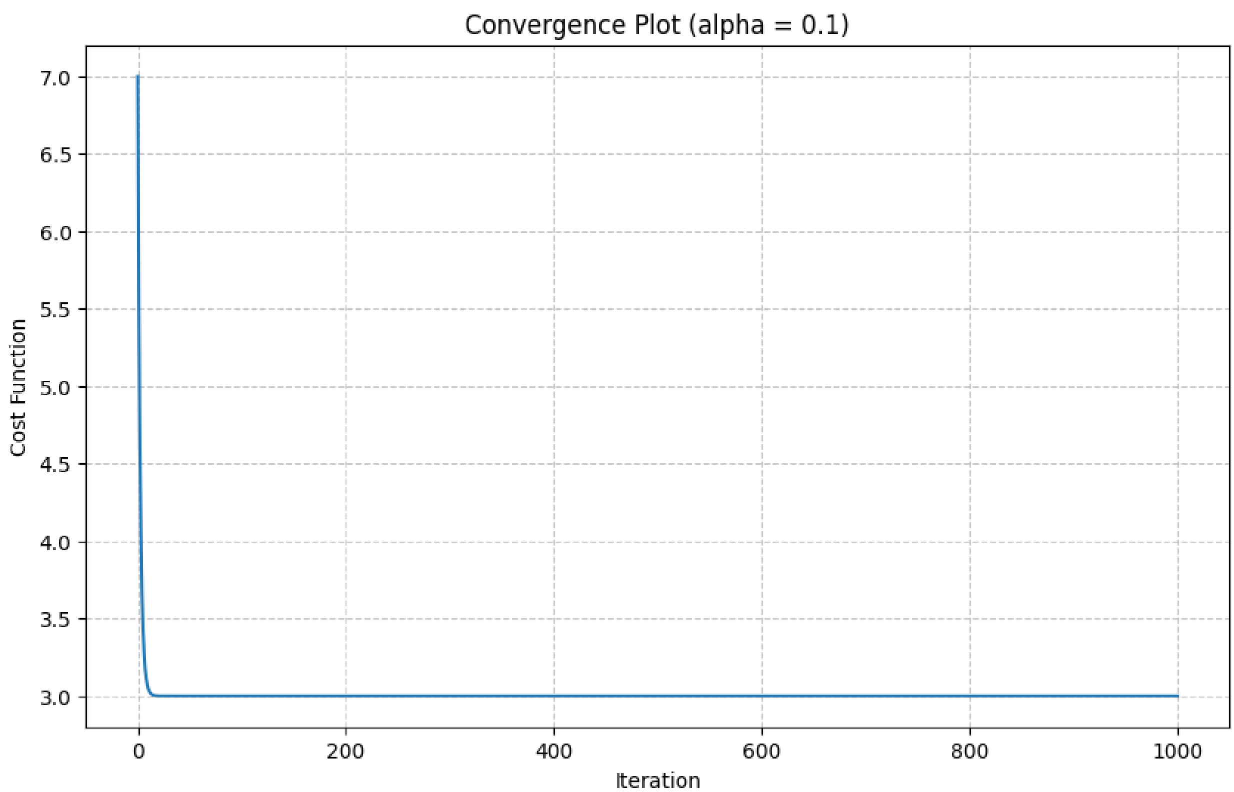
```
def fence_cost(L):
    return 16 * L + 696960 / L

def fence_derivative(L):
    return 16 - 696960 / (L**2)

# Plot for Fence Cost
optimal_L = np.sqrt(43560)
min_cost = fence_cost(optimal_L)

# Use a narrow range around the optimal L
L = np.linspace(150, 300, 1000)
C = fence_cost(L)

plt.figure(figsize=(10, 6))
```

```
plt.plot(L, C, label='Cost Function C(L)')
plt.scatter([optimal_L], [min_cost], color='red', zorder=5,
label=f'Min: L={optimal_L:.2f}')
plt.title('Fence Cost Optimization')
plt.xlabel('Length L (feet)')
plt.ylabel('Cost C(L) (dollars)')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.savefig('fence_cost.png')
```




```
x_history, f_history = gradient_descent(function1, derivative1, 0,
0.1, 1000)

# How many iterations to get within 0.01 of true minimum (x=2)?
true_min_x = 2
iterations_needed = -1
for i, x in enumerate(x_history):
    if abs(x - true_min_x) < 0.01:
        iterations_to_target = i
        break

print(f"Iterations to get within 0.01 of x=2: {iterations_to_target}")

plt.figure(figsize=(10, 6))
plt.plot(f_history)
```

```
plt.title('Convergence Plot (alpha = 0.1)')
plt.xlabel('Iteration')
plt.ylabel('Cost Function')
plt.grid(True, linestyle='--', alpha=0.7)
plt.savefig('convergence_plot.png')

print(f"Optimal L: {optimal_L}")
print(f"Minimum Cost: {min_cost}")

Iterations to get within 0.01 of x=2: 24
Optimal L: 208.71032557111303
Minimum Cost: 6678.730418275617
```



Problem 3.2: Reflection Questions

```
The closed form derivative optimization solution is obviously much
faster than an iterative approach. When might iterative methods be
necessary?
What role does the learning rate play in convergence speed and
stability?
```

1. We need the iterative approach whenever the function is transcendental such that we cannot use the closed form approach.
2. Too low of a rate can cause it to never converge in the given iterations, too much can cause it to explode, bouncing back and forth past the true value and grow rapidly.