Computational Physics 2 Lab 2

Zachary Wojcik

February 2026

1 Abstract

This program searches for the global minimum of an oscillatory graph using two different numerical methods, gradient descent, and simulated annealing. The simulated annealing appeared to find the global minimum while the gradient descent simply found the nearest local minimum. The simulated annealing technique was then used to simulate the formation of crystalline structures.

2 Introduction

The beginning of this lab attempts to find the global minimum of the equation

$$E(x) = x^2 - 4\cos(4\pi x)$$

Which looks like this:

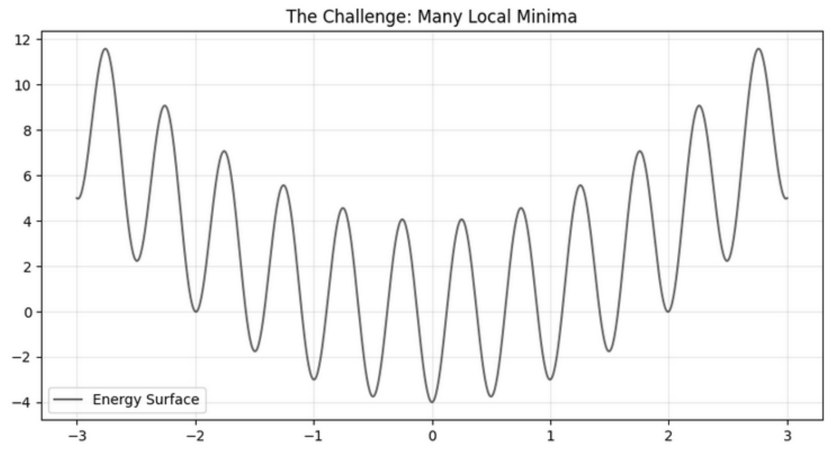

Figure 1: Graph of function

The gradient dissent method follows the following differential equation to direct it from point to point until a local minimum is found:

$$x_n = x_{n-1} - \eta \frac{dE}{dx}$$

Where η is the "Learning rate" or the step size of each iteration.
This was then repeated using simulated annealing which changes the x position slightly in order to decrease the chance of the algorithm being compressed into a single local minimum.
The results of these two algorithms showed that the gradient descent simply converged the local minimum closest to where it started, but the jumps from the simulated annealing caused the estimations to jump between energy wells until it reached the lowest energy state.
The simulated annealing algorithm was then used to simulate where the certain molecules will converge to in a compressed space to show the minimum energy states form a crystalline structure.

3 Methods

After plotting the convergence paths of the two numerical methods, gradient descent in red and simulated annealing in blue, we see

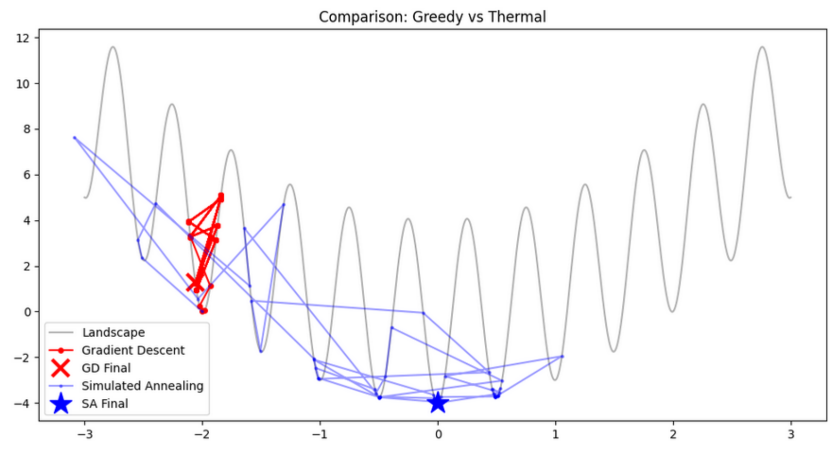

Figure 2: Plot of convergence paths

From these graphs, we can see that the convergence path of the gradient descent gets stuck in its current energy well, while the simulated annealing allows a global search, jumping between energy wells to find the true global minimum energy state.
For the second part of this experiment, we simulate the crystallization of Argon by randomly placing n Argon atoms in a confined area and using simulated

annealing to converge the atoms to the lowest energy state. This creates a pattern where the argon atoms bond into a crystalline structure as shown:

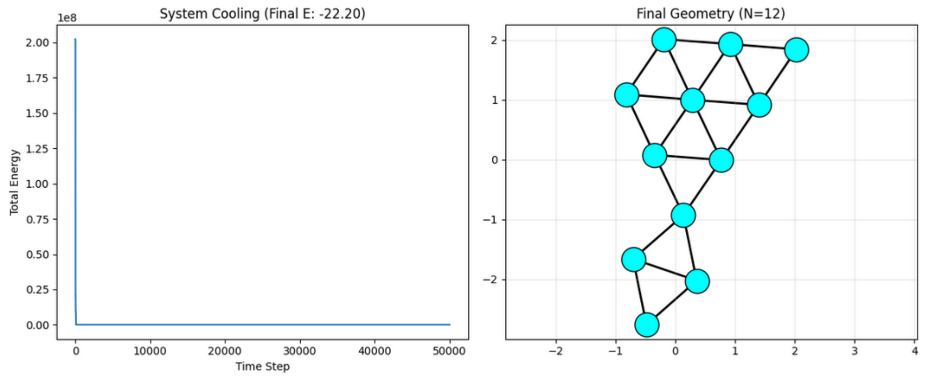

Figure 3: Position of atoms in the structure

This shows how crystals form from rapidly cooling liquids.

4 Discussion

The simulated annealing allows for a better way to find the global minimum of an energy distribution as it allows the convergence path to jump between energy wells to find the lowest possible value rather than the local minimum of the current energy well. This then allows us to better simulate the convergence of certain particles to the lowest energy state in its vicinity, creating a crystalline structure.

5 Appendix A: Questions

Question 1: Why does the line stop in the hole at $x \approx -2$, while the blue line is able to escape to $x \approx 0$.
This occurs because the simulated annealing applies a shift in x which at times flattens the slope and allows the line to pass through to another energy well.
Question 2: Run the code for $n = 4$. What shape does it form? Does it ever form a square? Why or why not?
It very rarely forms a square, but it will commonly form a parallelogram or two will bond separately. This is most likely because the space is too big to compress the atoms tight enough to force it into a square.

6 Appendix B: Code

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

def energy_function(x):
    """The potential energy landscape."""
    return x**2 - 4 * np.cos(4 * np.pi * x)

# Visualize
x_vals = np.linspace(-3, 3, 1000)
plt.figure(figsize=(10, 5))
plt.plot(x_vals, energy_function(x_vals), 'k-', alpha=0.6, label='Energy Surface')
plt.title("The Challenge: Many Local Minima")
plt.grid(True, alpha=0.3)
plt.legend()
plt.show()

def run_gradient_descent(x_start, steps=50, learning_rate=0.005):
    path = [x_start]
    x = x_start

    for _ in range(steps):
        # Numerical Derivative (Slope)
        dx = 0.001
        slope = (energy_function(x + dx) - energy_function(x - dx)) / (2*dx)

        # TODO: Update x using the Gradient Descent rule
        x = x - learning_rate * slope

        path.append(x)

    return np.array(path)

# Start at x = -2 (A bad spot!)
x_start = -2.0
gd_path = run_gradient_descent(x_start)

print(f"Gradient Descent started at {x_start} and finished at {gd_path[-1]:.3f}")
print(f"True Global Minimum is at 0.0")

def run_annealing(x_start, T_start=10.0, cooling_rate=0.99, steps=1000):
    x = x_start
    E = energy_function(x)
    T = T_start

    path = [x]

    for i in range(steps):
        # 1. Propose a random small move
        dx = np.random.uniform(-0.5, 0.5)
        x_new = x + dx

        # 2. Calculate Energy Change
        E_new = energy_function(x_new)
        delta_E = E_new - E

        # 3. Metropolis Logic
        # TODO: Decide whether to accept the move
        accept = False

        # HINT: If delta_E < 0, always accept.
        #       If delta_E > 0, accept with probability exp(-delta_E / T).

        if (delta_E < 0):
          accept = True
        else:
          prob = np.exp(-delta_E / T)
          if (np.random.rand() < prob):
            accept = True

        if accept:
            x = x_new
            E = E_new

        # 4. Cool down
        T = T * cooling_rate
        path.append(x)

    return np.array(path)

# --- VISUALIZATION ---
# We compare the two methods side-by-side

# 1. Run Gradient Descent
gd_path = run_gradient_descent(-2.0, steps=50, learning_rate=0.005)

# 2. Run Simulated Annealing (Uses solution function for demo)
sa_path = run_annealing(-2.0, T_start=10.0, cooling_rate=0.99, steps=1000)

plt.figure(figsize=(12, 6))
plt.plot(x_vals, energy_function(x_vals), 'k-', alpha=0.3, label='Landscape')

# Plot Gradient Descent (Red)
plt.plot(gd_path, energy_function(gd_path), 'o-', color='red', label='Gradient Descent', mar
plt.plot(gd_path[-1], energy_function(gd_path[-1]), 'rx', markersize=15, markeredgewidth=3,

# Plot Simulated Annealing (Blue)
# We only plot every 10th step so the graph isn't messy
plt.plot(sa_path[::10], energy_function(sa_path)[::10], 'o-', color='blue', alpha=0.4, label
plt.plot(sa_path[-1], energy_function(sa_path[-1]), 'b*', markersize=20, label='SA Final')

plt.title("Comparison: Greedy vs Thermal")
plt.legend()
plt.show()

def lennard_jones_energy(positions):
    """
    Calculates total energy of N atoms.
    positions: An array of shape (N, 2) containing (x, y) for each atom.
    """
    N = len(positions)
    total_energy = 0

    # Loop over every unique pair of atoms
    for i in range(N):
        for j in range(i + 1, N):
            # Distance formula
            dist_vector = positions[i] - positions[j]
            r = np.linalg.norm(dist_vector)

            # Prevent division by zero if atoms overlap perfectly
            if r < 0.01:
                r = 0.01

            # Lennard-Jones formula (epsilon=1, sigma=1)
            # Term 1: Repulsion (r^-12)
            # Term 2: Attraction (r^-6)
            energy = 4 * ((1/r)**12 - (1/r)**6)
            total_energy += energy

    return total_energy

def solve_cluster(N_atoms, steps=50000):
    # 1. Initialize random positions
    positions = np.random.uniform(-1, 1, size=(N_atoms, 2))
    curr_E = lennard_jones_energy(positions)

    T = 20
    cooling_rate = 0.999
    history_E = []

    for i in range(steps):
        # 2. Propose a move
        # TODO: Pick one random atom and move it by a small random amount
        atom_choice = np.random.randint(0, N_atoms)
        dr = np.random.normal(0, 0.1, size = 2)
        new_positions = positions.copy()
        new_positions[atom_choice] = new_positions[atom_choice] + dr




        # 3. Calculate Energy Change
        new_E = lennard_jones_energy(new_positions)
        delta_E = new_E - curr_E


        # 4. Metropolis Criterion
        # TODO: Copy your logic from Exercise 1.2 to accept/reject here
        accept = False

        if (delta_E < 0):
          accept = True
        else:
          prob = np.exp(-delta_E / T)
          if (np.random.rand() < prob):
            accept = True

        if accept:
            positions = new_positions
            curr_E = new_E

        T *= cooling_rate
        history_E.append(curr_E)

    return positions, history_E

# --- RUN THE EXPERIMENT ---
N = 4 # Try N=3 (triangle), N=4 (diamond), N=7 (hexagon with center)
final_pos, energy_log = solve_cluster(N)

# Plotting
plt.figure(figsize=(12, 5))

# Plot 1: The Energy Drop (Cooling)
```

```
plt.subplot(1, 2, 1)
plt.plot(energy_log)
plt.title(f"System Cooling (Final E: {energy_log[-1]:.2f})")
plt.xlabel("Time Step")
plt.ylabel("Total Energy")

# Plot 2: The Final Crystal Shape
plt.subplot(1, 2, 2)
plt.scatter(final_pos[:,0], final_pos[:,1], s=500, c='cyan', edgecolors='black', zorder=2)

# Draw bonds between close atoms to visualize structure
from scipy.spatial.distance import pdist, squareform
dists = squareform(pdist(final_pos))
for i in range(N):
    for j in range(i+1, N):
        if dists[i,j] < 1.5: # If close enough to bond
            plt.plot([final_pos[i,0], final_pos[j,0]], [final_pos[i,1], final_pos[j,1]], 'k-

plt.title(f"Final Geometry (N={N})")
plt.axis('equal')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```