Lab 8: Quadratic Programming and Simulated Annealing

This lab continues the optimization theme from linear programming. In the previous lab, the objective and the constraints were linear. Here we study two extensions:

- **quadratic programming**, where the objective may contain squares and products of variables, while the constraints are still linear;
- **simulated annealing**, a heuristic method for searching through very large solution spaces.

Goals

By the end of the lab, you should be able to:

- recognize a quadratic programming problem;
- rewrite a quadratic objective in matrix form;
- use `quadprog` to solve a constrained quadratic problem;
- explain what a state, neighborhood, move, and energy function mean in simulated annealing;
- implement a simple simulated annealing solution for the 0–1 knapsack problem.

Part 1: Quadratic Programming

Problem Type

A quadratic programming problem has the form:

$$\min \frac{1}{2} x^T D x - d^T x$$

subject to linear constraints:

$$A^T x \geq b.$$

The matrix D describes the quadratic part of the objective, and the vector d describes the linear part. The constraints must still be linear.

The `quadprog` Package

In this lab we use the Python package `quadprog`. It is a solver for quadratic programming problems with linear constraints. Just like `PuLP` allowed us to describe a linear program and send it to a solver, `quadprog` lets us describe a quadratic program by giving it the matrices and vectors D, d, A, and b.

`quadprog` expects the problem in a specific format:

$$\min\left(\frac{1}{2}x^T Dx - d^T x\right) \qquad \text{subject to} \qquad A^T x \geq b.$$

This means that part of the work is translating the mathematical problem into the exact input format required by the package.

Worked Example

Solve the following quadratic problem:

$$\min\left(a - 2b + 4c + a^2 + 2b^2 + 3c^2 + ac\right)$$

subject to:

$$3a + 4b - 2c \leq 10,$$
$$-3a + 2b + c \geq 2,$$
$$2a + 3b + 4c = 5,$$
$$0 \leq a \leq 5, \qquad 1 \leq b \leq 5, \qquad 0 \leq c \leq 5.$$

Matrix Form

The package `quadprog` solves problems in the form

$$\min\left(\frac{1}{2}x^T Dx - d^T x\right).$$

For this example:

$$x = \begin{bmatrix} a \\ b \\ c \end{bmatrix}.$$

The quadratic part is

$$a^2 + 2b^2 + 3c^2 + ac.$$

Because `quadprog` uses the factor $\frac{1}{2}$, we use

$$D = \begin{bmatrix} 2 & 0 & 1 \\ 0 & 4 & 0 \\ 1 & 0 & 6 \end{bmatrix}.$$

The linear part is

$$a - 2b + 4c.$$

Since the package uses $-d^T x$, we set

$$d = -\begin{bmatrix} 1 \\ -2 \\ 4 \end{bmatrix}.$$

Python Code

```
import numpy as np
import quadprog

# Minimize: a - 2b + 4c + a^2 + 2b^2 + 3c^2 + ac

Dmat = np.array([
    [2, 0, 1],
    [0, 4, 0],
    [1, 0, 6]
], dtype="float64")

dvec = -np.array([1, -2, 4], dtype="float64")

# quadprog uses constraints of the form A^T x >= b.
# Equalities must be placed first.
Amat = np.array([
    [2, 3, 4],      # equality: 2a + 3b + 4c = 5
    [-3, -4, 2],    # 3a + 4b - 2c <= 10 becomes -3a - 4b + 2c >= -10
    [-3, 2, 1],     # -3a + 2b + c >= 2
    [1, 0, 0],      # a >= 0
    [-1, 0, 0],     # a <= 5 becomes -a >= -5
    [0, 1, 0],      # b >= 1
    [0, -1, 0],     # b <= 5 becomes -b >= -5
    [0, 0, 1],      # c >= 0
    [0, 0, -1]      # c <= 5 becomes -c >= -5
], dtype="float64").T

bvec = np.array([5, -10, 2, 0, -5, 1, -5, 0, -5], dtype="float64")

result = quadprog.solve_qp(Dmat, dvec, Amat, bvec, meq=1)

print("Optimal solution:", result[0])
print("Objective function value:", result[1])
```

Exercise 1

Solve the following quadratic programming problem using `quadprog`:

$$\min \left(x + y + 2x^2 + 2xy + 2y^2\right)$$

subject to:

$$-x + y \leq 0,$$
$$x + 2y \geq 6,$$
$$3x - 4y = 10,$$
$$4 \leq x \leq 20, \qquad 2 \leq y \leq 22.$$

Tasks.

a) Write the vector of variables.

b) Build *Dmat* and *dvec*.

c) Rewrite every constraint in the form required by `quadprog`.

d) Solve the problem and report the optimal x, y, and objective value.

Part 2: Simulated Annealing

Main Idea

Simulated annealing is a heuristic optimization method. It is useful when the search space is large and we cannot easily try every possible solution.

The method works with:

- a **state**, meaning one candidate solution;
- a **move**, meaning a small random change to the current state;
- an **energy function**, meaning the value we want to minimize;
- a **temperature schedule**, which controls how willing the method is to accept worse solutions while searching.

At high temperature, the algorithm is more willing to accept worse moves. This helps it escape local minima. At low temperature, it becomes more conservative and behaves more like local search.

Python Packages Used in This Part

The code in this part uses:

- `numpy` for arrays and random data generation;
- `matplotlib` for plotting functions and routes;
- `scipy.optimize.dual_annealing` for simulated annealing on a continuous function;
- `scipy.spatial.distance_matrix` for computing distances between points;
- `simanneal.Annealer` for building custom simulated annealing solvers.

If a package is missing, install it before running the code:

```
pip install numpy matplotlib scipy simanneal
```

Exercise 2: Continuous Optimization With the Rastrigin Function

Problem statement. Use simulated annealing to minimize the two-dimensional Rastrigin function

$$f(x) = \sum_i \left(x_i^2 - 10\cos(2\pi x_i) + 10\right)$$

on the domain

$$-5.12 \le x_i \le 5.12.$$

The Rastrigin function is useful for testing optimization algorithms because it has many local minima. The global minimum is at the origin, where the function value is 0. Your goal is to use `dual_annealing` to find a point close to this global minimum.

Starting code.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import dual_annealing

def rastrigin(x):
    return sum([d**2 - 10 * np.cos(2 * np.pi * d) + 10 for d in x])

x = np.linspace(-5.12, 5.12, 100)
y = np.linspace(-5.12, 5.12, 100)
X, Y = np.meshgrid(x, y)
Z = rastrigin([X, Y])

fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
ax.plot_surface(X, Y, Z, cmap="viridis")
plt.show()
```

```
bounds = [(-5.12, 5.12), (-5.12, 5.12)]
result = dual_annealing(rastrigin, bounds=bounds)

print("Minimum value found:", result.fun)
print("At the point:", result.x)
```

Tasks.

a) Run the code and inspect the surface plot.

b) Report the minimum value found.

c) Report the point where the minimum was found.

d) Compare the result with the known global optimum $f(0,0) = 0$.

Exercise 3: Traveling Salesman Problem

Problem statement. We are given a set of cities represented by points in the plane. The goal is to find a short cyclic route that visits each city exactly once and returns to the starting city.

In the traveling salesman problem, a state is an ordering of cities. A move can be made by swapping two cities. The energy is the total route length.

Starting code.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import distance_matrix
from simanneal import Annealer

np.random.seed(123)
cities = np.random.rand(10, 2)
dist_matrix = distance_matrix(cities, cities)
initial_state = list(range(len(cities)))

class TravelingSalesmanProblem(Annealer):
    def move(self):
        a, b = np.random.choice(len(self.state), 2, replace=False)
        self.state[a], self.state[b] = self.state[b], self.state[a]

    def energy(self):
        return sum(
            dist_matrix[self.state[i], self.state[(i + 1) % len(self.
               state)]]
            for i in range(len(self.state))
        )
```

```
tsp = TravelingSalesmanProblem(initial_state)
tsp.set_schedule(tsp.auto(minutes=0.1))
state, e = tsp.anneal()

print("Path:", state)
print("Path length:", e)

plt.plot(
    [*cities[state, 0], cities[state[0], 0]],
    [*cities[state, 1], cities[state[0], 1]],
    "o-"
)
plt.show()
```

Tasks.

a) Explain what the state represents.

b) Explain what the `move()` function does.

c) Explain what the `energy()` function computes.

d) Run the code and report the path length found by simulated annealing.

Exercise 4: 0–1 Knapsack With Simulated Annealing

Problem statement. You are given $n = 50$ items. Each item has a price and a weight. You must choose a subset of the items. The goal is to maximize total price while keeping total weight at most `maxWeight`.

In the 0–1 knapsack problem, each item can either be selected or not selected. We want to maximize total price while keeping total weight below a maximum allowed weight.

Starting code.

```
import numpy as np

np.random.seed(123)

n = 50
prices = np.random.uniform(10, 40, size=n)
weights = np.random.uniform(5, 10, size=n)
maxWeight = sum(weights) * 0.6

initial_state = np.array([0] * (n // 2) + [1] * (n // 2))
np.random.shuffle(initial_state)
```

State. A state is a vector of 0's and 1's:

$$s_i = \begin{cases} 1, & \text{if item } i \text{ is selected,} \\ 0, & \text{otherwise.} \end{cases}$$

Tasks.

a) Define a simple neighborhood: flip one item from 0 to 1, or from 1 to 0.

b) Write code that generates all neighbors in this simple neighborhood.

c) Define a wider neighborhood: flip 2, 3, or more items at once.

d) Implement an annealer for the knapsack problem.

e) Report the best state found, its total price, and its total weight.

Hint. Since `simanneal` minimizes energy, a natural choice is:

$$\text{energy} = -\text{total price}$$

for feasible states. For infeasible states, add a large penalty for exceeding the maximum weight.