Department of Mathematics and Computer Science
University of Southern Denmark, Odense

March 9, 2025
Marco Chiarandini

AI505 – Optimization

Sheet 03, Spring 2025

Exercises with the symbol $^+$ are to be done at home before the class. Exercises with the symbol * will be tackled in class. The remaining exercises are left for self training after the exercise class. Some exercises are from the text book and the number is reported. They have the solution at the end of the book.

Exercise 1$^+$ (6.1)

What advantage does second-order information provide about the point of convergence that first-order information lacks?

Exercise 2$^+$ (6.2)

When would we use Newton's method instead of the bisection method for the task of finding roots in one dimension?

Exercise 3* (6.4, 6.9)

Apply Newton's method to $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T H\mathbf{x}$ starting from $\mathbf{x}_0 = [1, 1]$. What have you observed? Use H as follows:

$$H = \begin{bmatrix} 1 & 0 \\ 0 & 1000 \end{bmatrix}$$

Next, apply gradient descent to the same optimization problem by stepping with the unnormalized gradient. Do two steps of the algorithm. What have you observed? Finally, apply the conjugate gradient method. How many steps do you need to converge?
Repeat the exercise for:

$$f(\mathbf{x}) = (x_1 + 1)^2 + (x_2 + 3)^2 + 4.$$

starting at the origin.
Note that $H = A$, hence we could have derived A also by calculating the Hessian.

Exercise 4$^+$ (6.5)

Compare Newton's method and the secant method on $f(x) = x^2 + x^4$, with $x_1 = -3$ and $x_0 = -4$. Run each method for 10 iterations. Make two plots:

1. Plot f vs. the iteration for each method.
2. Plot f' vs. x. Overlay the progression of each method, drawing lines from $(x_i, f'(x_i))$ to $(x_{i+1}, 0)$ to $(x_{i+1}, f'(x_{i+1}))$ for each transition.

What can we conclude about this comparison?

Exercise 5$^+$ (7.1)

Direct methods are able to use only zero-order information—evaluations of f. How many evaluations are needed to approximate the derivative and the Hessian of an n-dimensional objective function using finite difference methods? Why do you think it is important to have zero-order methods?

Exercise 6*

Implement the extended Rosenbrock function

$$f(x) = \sum_{n/2}^{i=1} \left[a(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2\right]$$

where a is a parameter that you can vary (for example, 1 or 100). The minimum is $\mathbf{x}^* = [1, 1, \ldots, 1]$, $f(\mathbf{x}^*) = 0$. Consider as starting point $[-1, -1, ..., -1]$.
Solve the minimization problem with scipy.optimize using all methods seen in class that are suitable for this task. Observe the behavior of the calls for various values of parameters, for example, for the L-BFGS algorithm the memory parameter m.

Exercise 7

Below you find an implementation of Nelder-Mead algorithm in Python. Analyze it and use it to solve the Rosenbrock function in 2D. Plot the evolution of the simplex throughout the search (you can get help from chatGPT to code the plottig facilities).

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

def nelder_mead(f, S, eps, max_iterations, alpha=1.0, beta=2.0, gamma=0.5):
    delta = float("inf")
    y_arr = np.array([f(x) for x in S])
    simplex_history = [S.copy()]
    iterations=0
    while delta > eps and iterations <= max_iterations:
        iterations+=1
        # Sort by objective values (lowest to highest)
        p = np.argsort(y_arr)
        S, y_arr = S[p], y_arr[p]
        xl, yl = S[0], y_arr[0] # Lowest
        xh, yh = S[-1], y_arr[-1] # Highest
        xs, ys = S[-2], y_arr[-2] # Second-highest
        xm = np.mean(S[:-1], axis=0) # Centroid

        # Reflection
        xr = xm + alpha * (xm - xh)
        yr = f(xr)

        if yr < yl:
            # Expansion
            xe = xm + beta * (xr - xm)
            ye = f(xe)
            S[-1], y_arr[-1] = (xe, ye) if ye < yr else (xr, yr)
        elif yr >= ys:
            if yr < yh:
                xh, yh = xr, yr
                S[-1], y_arr[-1] = xr, yr
            # Contraction
            xc = xm + gamma * (xh - xm)
            yc = f(xc)
            if yc > yh:
                # Shrink
                for i in range(1, len(S)):
                    S[i] = (S[i] + xl) / 2
                    y_arr[i] = f(S[i])
            else:
                S[-1], y_arr[-1] = xc, yc
        else:
            S[-1], y_arr[-1] = xr, yr

        simplex_history.append(S.copy())
        delta = np.std(y_arr, ddof=0)
```