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

March 9, 2025
Marco Chiarandini

AI505 – Optimization

Sheet 03, Spring 2025

Solution:

Included.

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?

Solution:

Second-order information can guarantee that one is at a local minimum, whereas a gradient of zero is necessary but insufficient to guarantee local optimality.

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?

Solution:

We would prefer Newtons method if we start sufficiently close to the root and can compute derivatives analytically. Newton's method enjoys a better rate of convergence.

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.

Solution:

Newton's method updates design points (solutions) as follows:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - H_k^{-1}\nabla f(\mathbf{x}_k)$$

and substituting $\nabla f(\mathbf{x}_k) = H\mathbf{x}_k$ yields:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{x}_k = \mathbf{0}$$

Hence

$$\mathbf{x}_1 = \mathbf{0}$$

and the search finishes in one step because the following ones would not yield any change.

Gradient Descent update rule is as follows for unnormalized case:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla f(\mathbf{x}_k)$$

and subtituting $\nabla f(\mathbf{x}_k) = H\mathbf{x}_k$ for the specific case and setting $\alpha = 1$ yields:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - H\mathbf{x}_k$$

$$\mathbf{x}_1 = \begin{bmatrix}1\\1\end{bmatrix} - \begin{bmatrix}1 & 0\\0 & 1000\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}1\\-999\end{bmatrix}$$

$$\mathbf{x}_2 = \begin{bmatrix}0\\-999\end{bmatrix} - \begin{bmatrix}1 & 0\\0 & 1000\end{bmatrix}\begin{bmatrix}0\\-999\end{bmatrix} = \begin{bmatrix}0\\998111\end{bmatrix}$$

$$\mathbf{x}_3 = \begin{bmatrix}0\\998001\end{bmatrix} - \begin{bmatrix}1 & 0\\0 & 1000\end{bmatrix}\begin{bmatrix}0\\998001\end{bmatrix} = \begin{bmatrix}0\\-997002999\end{bmatrix}$$

Hence, the gradient descent method diverges. It must be due to the selection of the initial point.

Conjugate gradient

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k \qquad \mathbf{d}_k = -\mathbf{r}_k + \beta_k \mathbf{d}_{k-1}$$

$$\mathbf{r}_k = \nabla f(\mathbf{x}_k) = H\mathbf{x}_k \qquad \beta_k = \frac{\mathbf{r}_k^T H \mathbf{d}_{k-1}}{\mathbf{d}_{k-1}^T H \mathbf{d}_{k-1}} \qquad \alpha_k = -\frac{\mathbf{r}_k \mathbf{d}_k}{\mathbf{d}_k^T H \mathbf{d}_k}$$

$$\mathbf{x}_1 = \mathbf{x}_0 + \alpha_0 \mathbf{d}_0$$

We choose the first search direction $\mathbf{d}_0$ to be the steepest descent direction at the initial point $\mathbf{x}_0$. Hence, $\mathbf{d}_0 = -\nabla f(\mathbf{x}_0) = -H\mathbf{x}_0$ and

$$\alpha_0 = -\frac{(H\mathbf{x}_0)^T(-H\mathbf{x}_0)}{(-H\mathbf{x}_0)^T H(-H\mathbf{x}_0)} = \frac{10001}{1000001}$$

The first iteration then yields:

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha_0 \mathbf{d}_0 = \begin{bmatrix}1\\1\end{bmatrix} - 0.010\begin{bmatrix}1\\1000\end{bmatrix} = \begin{bmatrix}0.99\\11\end{bmatrix}$$

The second iteration:

$$\mathbf{x}_2 = \mathbf{x}_1 + \alpha_1 \mathbf{d}_1$$

Setting $r_1 = H\mathbf{x}_1$:

$$\beta_1 = \frac{\mathbf{r}_1^T H \mathbf{d}_0}{\mathbf{d}_0^T H \mathbf{d}_0} = \frac{(H\mathbf{x}_1)^T H(H\mathbf{d}_0)}{\mathbf{d}_0^T H \mathbf{d}_0} = \ldots$$

$$\mathbf{d}_1 = -\mathbf{r}_1 + \beta_1 \mathbf{d}_0 = \ldots$$

$$\alpha_1 = -\frac{(H\mathbf{x}_1)^T \mathbf{d}_0}{\mathbf{d}_0^T H \mathbf{d}_0} = \ldots$$

$$\mathbf{x}_2 \approx \begin{bmatrix}0\\0\end{bmatrix}$$

The matrix H has eigenvalues 1 and 100 that are both positive, hence the matrix is positive definite (and hence symmetric). In this case, the conjugate gradient method needs n iterations to find the local optimum. So $\mathbf{x}_2$ should be the optimal solution.

The function:

$$f(\mathbf{x}) = (x_1 + 1)^2 + (x_2 + 3)^2 + 4 = x_1^2 + x_2^2 + 2x_1 + 6x_2 + 9 + 4 + 1$$

Can be obtained by

$$f(\mathbf{x}) = 1/2\mathbf{x}^T A\mathbf{x} + \mathbf{b}^T\mathbf{x} + c$$

where $\mathbf{x} = [x_1, x_2]^T$, $\mathbf{b} = [2, 6]$, $c = 14$ and

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

The matrix A is positive definite since its eigenvalues are positive. Hence, this is a quadratic function where the Netwon's method finds the optimal solution in one iteration.

$$\nabla f(\mathbf{x}) = [2(x_1 + 1), 2(x_2 + 3)]$$

$$H = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

$$\mathbf{x}_1 = \mathbf{x}_0 - H^{-1}\nabla(\mathbf{x}_0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}^{-1} \begin{bmatrix} 2 \\ 6 \end{bmatrix} = \begin{bmatrix} -1 \\ -3 \end{bmatrix}$$

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?

Solution:

Note that Newton's method is used both for finding roots and for finding minima. When used for finding roots it is also known as Newton-Raphson method. Compare:

Method	Goal	Needs	Update rule
Newton-Raphson method	Solve $f(x) = 0$	First derivative $f'(x)$	$x_{k+1} = x_k - \frac{f(x)}{f'(x)}$
Newton's method	Minimize $f(x) = 0$	First and second derivative $f'(x), f''(x)$	$x_{k+1} = x_k - \frac{f'(x)}{f''(x)}$

Its first derivative is $f'(x) = 2x + 4x^3$.
Its second derivative is $f'(x) = 2 + 12x^2$.
Newton's method updates the estimate x_k using the formula:

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

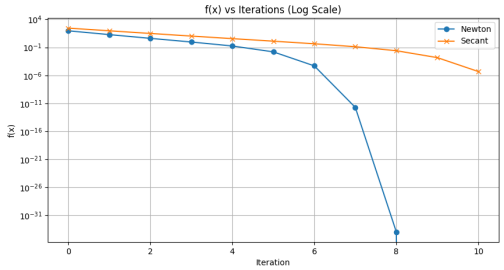
The secant method updates the estimate x_n using the formula:

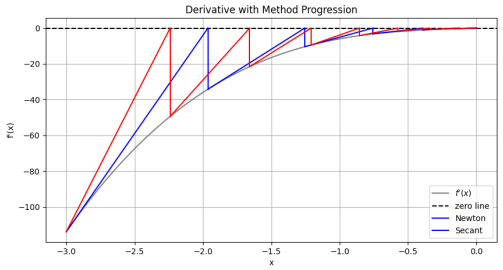
$$x_{k+1} = x_k - f'(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

Note that the secant method requires two initial guesses, x_0 and x_1.

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

# Define the function and its derivative
def f(x):
    return x**2 + x**4
```

```
def df(x):
    return 2*x + 4*x**3

def ddf(x):
    return 2 + 12*x**2


# Newton's method
def newton_method(x0, iterations):
    xs = [x0]
    for _ in range(iterations):
        x0 = x0 - df(x0) / ddf(x0)
        xs.append(x0)
    return np.array(xs)

# Secant method
def secant_method(x0, x1, iterations):
    xs = [x0, x1]
    for _ in range(iterations - 1):
        x_new = xs[-1] - df(xs[-1]) * (xs[-1] - xs[-2]) / (df(xs[-1]) - df(xs[-2]))
        xs.append(x_new)
    return np.array(xs)

# Plot f(x) vs iteration
def plot_function_vs_iteration():
    iterations = 10
    x_newton = newton_method(-3, iterations)
    x_secant = secant_method(-4, -3, iterations)

    plt.figure(figsize=(10, 5))
    plt.plot(range(iterations + 1), f(x_newton), 'o-', label="Newton")
    plt.plot(range(iterations + 1), f(x_secant), 'x-', label="Secant")
    plt.yscale("log")
    plt.xlabel("Iteration")
    plt.ylabel("f(x)")
    plt.title("f(x) vs Iterations (Log Scale)")
    plt.legend()
    plt.grid()
    #plt.show()
    plt.savefig("secant_function.png")


# Plot f'(x) vs x with transitions
def plot_derivative_with_transitions():
    x_vals = np.linspace(-3, 0, 1000)
    plt.figure(figsize=(10, 5))
    plt.plot(x_vals, df(x_vals), label="$f'(x)$", color='grey')
    plt.axhline(0, color='black', linestyle='--')

    # Newton's method
    x_newton = newton_method(-3, 10)
    for i in range(len(x_newton) - 1):
        plt.plot([x_newton[i], x_newton[i+1]], [df(x_newton[i]), 0], 'b-')
        plt.plot([x_newton[i+1], x_newton[i+1]], [0, df(x_newton[i+1])], 'b-')

    # Secant method
    x_secant = secant_method(-4, -3, 10)
    for i in range(1,len(x_secant) - 1):
        plt.plot([x_secant[i], x_secant[i+1]], [df(x_secant[i]), 0], 'r-')
        plt.plot([x_secant[i+1], x_secant[i+1]], [0, df(x_secant[i+1])], 'r-')

    plt.xlabel("x")
    plt.ylabel("f'(x)")
```






```
    plt.title("Derivative with Method Progression")
    plt.legend(["$f'(x)$", "zero line", "Newton", "Secant"])
    plt.grid()
    #plt.show()
    plt.savefig("secant_derivative.png")

# Run the plots
plot_function_vs_iteration()
```

The first plot illustrates how quickly each method converges to a local optimum.
The second plot visualizes how each method approaches the root in the context of the derivative's behavior.
Conclusions from the comparison:

- Convergence Rate: Newton's method typically exhibits quadratic convergence, meaning the error decreases exponentially with each iteration when close to the root. The secant method generally has a convergence rate of approximately 1.618 (the golden ratio), which is superlinear but slower than Newton's method.

- Initial Guess Sensitivity: Newton's method requires a good initial guess to ensure convergence, as poor initial guesses can lead to divergence or convergence to an unintended root. The secant method, while generally more robust to initial guesses, can still fail to converge if the initial guesses are not chosen appropriately.

- Computational Efficiency: If derivative computation is costly or impractical, the secant method may be preferred despite its slower convergence rate. However, if the derivative is readily available and computationally inexpensive, Newton's method's faster convergence can lead to quicker results.

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?

Solution:

The derivative has n terms whereas the Hessian has n^2 terms. Each derivative term requires two evaluations when using finite difference methods: $f(\mathbf{x})$ and $f(\mathbf{x} + h\mathbf{e}_i)$. Each Hessian term requires 4 evaluations when using finite difference methods (in fact less than 4 as some computations can be cached):

$$\frac{\partial^2 f}{\partial x_i \, \partial x_j} \approx \frac{f(\mathbf{x} + h\mathbf{e}(i) + h\mathbf{e}(j)) - f(\mathbf{x} + h\mathbf{e}(i)) - f(\mathbf{x} + h\mathbf{e}(j)) + f(\mathbf{x})}{h^2}$$

Thus, to approximate the gradient, we need $2n$ evaluations, and to approximate the Hessian we need on the order of $4n^2$ evaluations. Approximating the Hessian is prohibitively expensive for large n. Direct methods can take comparatively more steps using the same budget of evaluations n^2, as direct methods need not estimate the derivative or Hessian at each step.

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 $[-1, -1, \ldots, -1]$.
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.

Solution:

We use the COCO test suite (see article) to carry out this exercise. The advantages of the platform is that it provides:

- a set of problem instances to use, about 1000 to 5000 problems (number of functions × number of dimensions × number of instances)
- a collection of results from the literature
- tools to launch and analyze the experiments

The COCO framework considers functions divided in suites. Functions, f_i, within suites are distinguished by their identifier $i = 1, 2, \ldots$. They are further parametrized by the (input) dimension, n, and the instance number, j. We can think of j as an index to a continuous parameter vector setting. It parametrizes, among other things, search space translations and rotations. In practice, the integer j identifies a single instantiation of these parameters. We then have:

$$f_i^j \equiv f[n, i, j] : \mathbb{R}^n \to \mathbb{R} \qquad \mathbf{x} \mapsto f_i^j(\mathbf{x}) = f[n, i, j](\mathbf{x}).$$

Varying n or j leads to a variation of the same function i of a given suite. Fixing n and j of function f_i defines an optimization problem instance $(n, i, j) \equiv (f_i, n, j)$ that can be presented to the solver. Each problem receives again an index within the suite, mapping the triple (n, i, j) to a single number.
Varying the instance parameter j represents a natural randomization for experiments in order to:

- generate repetitions on a single function for deterministic solvers, making deterministic and non-deterministic solvers directly comparable (both are benchmarked with the same experimental setup)
- average away irrelevant aspects of the function definition,
- alleviate the problem of overfitting, and
- prevent exploitation of artificial function properties

We focus here only on the comparison between different implementations of BFGS. In particular, we compare a freshly run execution of `scipy.optimize.minimize(method='BFGS')` against the values collected in the suite.

```
import cocoex # experimentation module
import cocopp # post-processing module (not strictly necessary)
import scipy # to define the solver to be benchmarked

### input: define suite and solver (see also "input" below where fmin is called)
suite_name = "bbob"



budget_multiplier = 7 # increase to 3, 10, 30, ... x dimension

### prepare
```

```
suite = cocoex.Suite(suite_name, "", "") # see https://numbbo.github.io/coco-doc/C/#suite-
    parameters
# suite = cocoex.Suite(suite_name, "instances: 1-5", "dimensions: 2,3,5,10,20")
output_folder = '{}_of_{}_{}D_on_{}'.format(
       "bfgs", scipy.optimize.minimize.__module__ or '', int(budget_multiplier+0.499),
           suite_name)
observer = cocoex.Observer(suite_name, "result_folder: " + output_folder)
repeater = cocoex.ExperimentRepeater(budget_multiplier) #, min_successes=4) # x dimension
minimal_print = cocoex.utilities.MiniPrint()

### go
while not repeater.done(): # while budget is left and successes are few
   for problem in suite: # loop takes 2-3 minutes x budget_multiplier
       if repeater.done(problem):
           continue
       problem.observe_with(observer) # generate data for cocopp
       problem(problem.dimension * [0]) # for better comparability

       ### input: the next three lines need to be adapted to the specific fmin
       # xopt = fmin(problem, repeater.initial_solution_proposal(problem), disp=False) #
           could depend on budget_multiplier
       xopt = scipy.optimize.minimize(problem, repeater.initial_solution_proposal(problem)
           , method="BFGS", options={"disp":False}) # could depend on budget_multiplier
       problem(xopt.x) # make sure the returned solution is evaluated

       repeater.track(problem) # track evaluations and final_target_hit
       minimal_print(problem) # show progress

### post-process data
dsl = cocopp.main(observer.result_folder + ' bfgs-scipy*') # re-run folders look like
    "...-001" etc
```

An aggregate comparison is carried out by means of empirical distribution functions (ECDF), to be explained in class. The analysis is visualized in Figure 1. The plots show that the three algorithms peform very close to each others as expected. However, for the smaller dimensions it seems that the freshly tested version has some troubles. Why?

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)
```

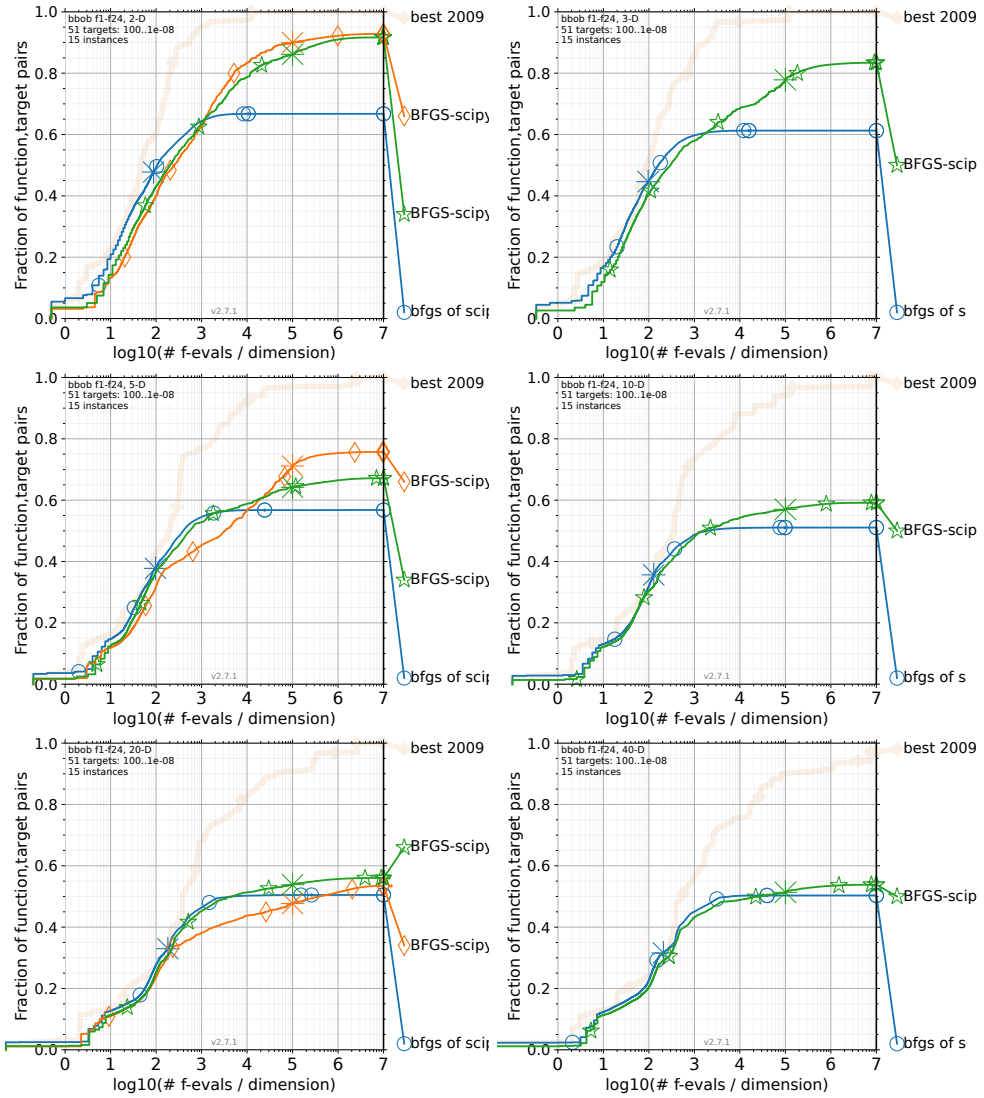


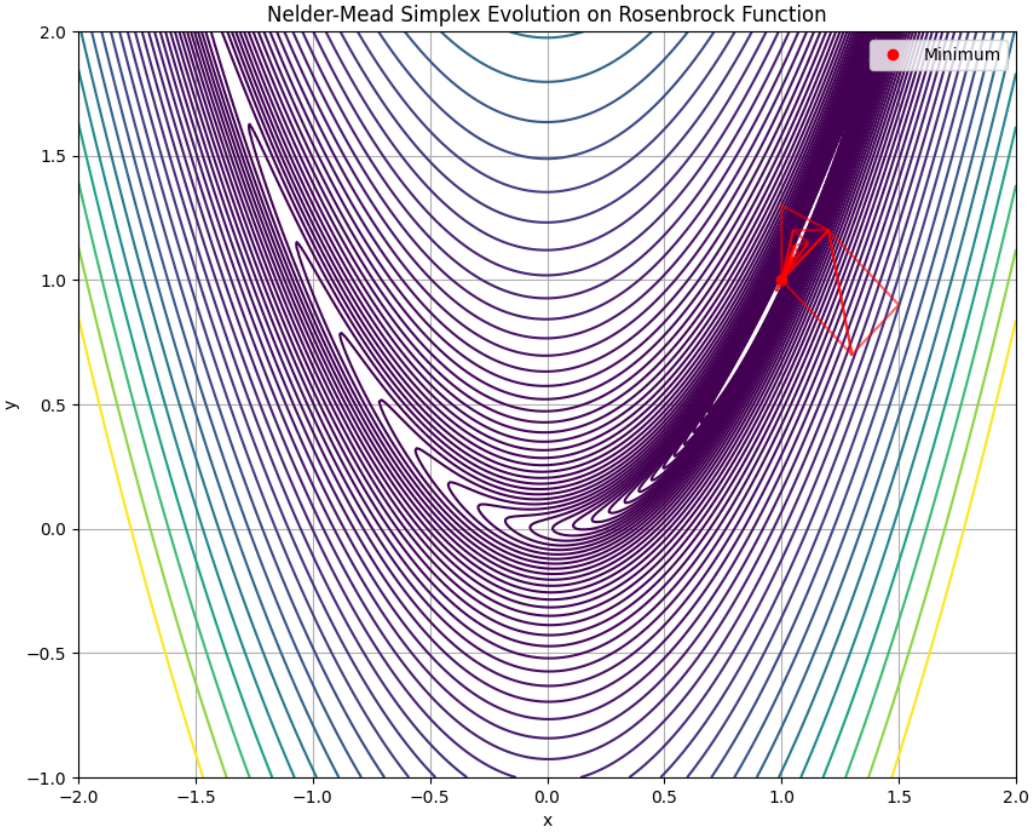
Figure 1: Comparison of BFGS implementaiton in COCO by means of aggregated ECDFs.

```
        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)
```

Solution:

```
if __name__ == "__main__":
    # Test function: Rosenbrock function
    def rosenbrock(x):
        return (1 - x[0])**2 + 100 * (x[1] - x[0]**2)**2

    # Initial simplex
    S = np.array([[1.3, 0.7], [1.5, 0.9], [1.2, 1.2]])
    epsilon = 1e-6

    result, simplex_history = nelder_mead(rosenbrock, S, epsilon, 100)
    print("Minimum found at:", result)
    print("Function value at minimum:", rosenbrock(result))

    # Plotting
    x = np.linspace(-2, 2, 400)
    y = np.linspace(-1, 2, 400)
    X, Y = np.meshgrid(x, y)
    Z = (1 - X)**2 + 100 * (Y - X**2)**2

    plt.figure(figsize=(10, 8))
    plt.contour(X, Y, Z, levels=np.logspace(-1, 3, 50), cmap="viridis")

    for simplex in simplex_history:
        plt.plotzip(*np.vstack([simplex, simplex[0]])), "r-",
            alpha=0.6)print(simplex.ravel())plt.plot(result[0], result[1], "ro",
            label="Minimum")plt.title("Nelder-Mead Simplex Evolution on Rosenbrock
            Function")plt.xlabel("x")plt.ylabel("y")plt.legend()plt.grid(True)plt.savefig("nelder.png")
```

```
Minimum found at: [1. 1.]
Function value at minimum: 1.232595164407831e-30
[1.3 0.7 1.5 0.9 1.2 1.2]
```




```
[1.2 1.2 1.3 0.7 1. 1. ]
[1. 1. 1.2 1.2 1. 1.3]
[1. 1. 1.2 1.2 1.05 1.2 ]
[1. 1. 1.05 1.2 1.1125 1.15 ]
[1. 1. 1.1125 1.15 1.053125 1.1375 ]
[1. 1. 1.053125 1.1375 1.06953125 1.109375 ]
[1. 1. 1.053125 1.1375 1.04804687 1.0890625 ]
[1. 1. 1.04804687 1.0890625 1.03857422 1.09101562]
[1. 1. 1.04804687 1.0890625 1.03129883 1.06777344]
[1. 1. 1.03129883 1.06777344 1.03184814 1.06147461]
[1. 1. 1.03184814 1.06147461 1.02361145 1.04925537]
[1. 1. 1.02361145 1.04925537 1.00178452 1.00620422]
[1. 1. 1.00178452 1.00620422 0.97817307 0.95694885]
[1. 1. 0.97817307 0.95694885 0.99543552 0.99233932]
[1. 1. 0.99543552 0.99233932 1.00749011 1.01578007]
[1. 1. 1.00749011 1.01578007 1.00789982 1.01566539]
[1. 1. 1.00789982 1.01566539 1.00217981 1.00385901]
[1. 1. 1.00217981 1.00385901 0.99768495 0.99506156]
[1. 1. 0.99768495 0.99506156 0.99717381 0.99436667]
[1. 1. 0.99717381 0.99436667 0.99903788 0.99824422]
[1. 1. 0.99903788 0.99824422 1.00069151 1.00149983]
[1. 1. 1.00069151 1.00149983 1.00099969 1.00200276]
```

The output above shows the coordinates of the points of the simplex throughout the iterations. The matrix 3×2 `simplex` has been flattened. There are 3 points in the simplex, each with two coordinates. Let's compare this implementation of Nelder-Mead simplex algorithm with the one from the scipy library using the test platform COCO. Below you find the script used and in Figure 2 the results.

The results seem to indicate that the version reported here performs better than the one in scipy. It remains to be understood whether the reason for the flattening of the cure is the computational budget.

```
import cocoex # experimentation module
import cocopp # post-processing module (not strictly necessary)
import scipy # to define the solver to be benchmarked
import numpy as np
from nelder import nelder_mead

def nelder_mead_my(func, x0, max_iterations, epsilon = 1e-6):
    N = x0.shape[0]
    S0 = np.zeros((N + 1, N))
    sign = -1
    for i in range(N):
        sign = np.zeros_like(x0)
        sign[i]=0.2 if i % 2 == 0 else -0.2
        S0[i,]=x0 + sign
    sign = np.zeros_like(x0)
    for j in range(N):
        sign[j]=0.2 if j % 2 == 0 else -0.2
    S0[N,]=x0 + sign

    x, simplex_history = nelder_mead(func, S0, eps=epsilon, max_iterations=max_iterations)
    return x

def nelder_mead_scipy(func, x0, maxfev):
    res = scipy.optimize.minimize(func, x0, method="Nelder-Mead", options={"disp":False, "
        maxfev":maxfev})
    return res.x


### input: define suite and solver (see also "input" below where fmin is called)
suite_name = "bbob"
# List of algorithms to test
algorithms = {
    "my_nelder_mead": nelder_mead_my,
    "scipy_nelder_mead": nelder_mead_scipy,
}

budget_multiplier = 10 # increase to 3, 10, 30, ... x dimension

### prepare
suite = cocoex.Suite(suite_name, "", "") # see https://numbbo.github.io/coco-doc/C/#suite-
    parameters
#suite = cocoex.Suite(suite_name, "instances: 1-5", "dimensions: 2,3,5,10,20")

minimal_print = cocoex.utilities.MiniPrint()

result_folders = []
### go
for algo_name, algo in algorithms.items():
    output_folder = f'{algo_name}_{int(budget_multiplier+0.499)}D_on_{suite_name}'
    observer = cocoex.Observer(suite_name, "result_folder: " + output_folder)
    repeater = cocoex.ExperimentRepeater(budget_multiplier) #, min_successes=4) # x
        dimension
    while not repeater.done(): # while budget is left and successes are few
        for problem in suite: # loop takes 2-3 minutes x budget_multiplier
            #print(f"Running {algo_name} on problem {problem.id}")

            if repeater.done(problem):
                continue
            problem.observe_with(observer) # generate data for cocopp
            problem(problem.dimension * [0]) # for better comparability
```

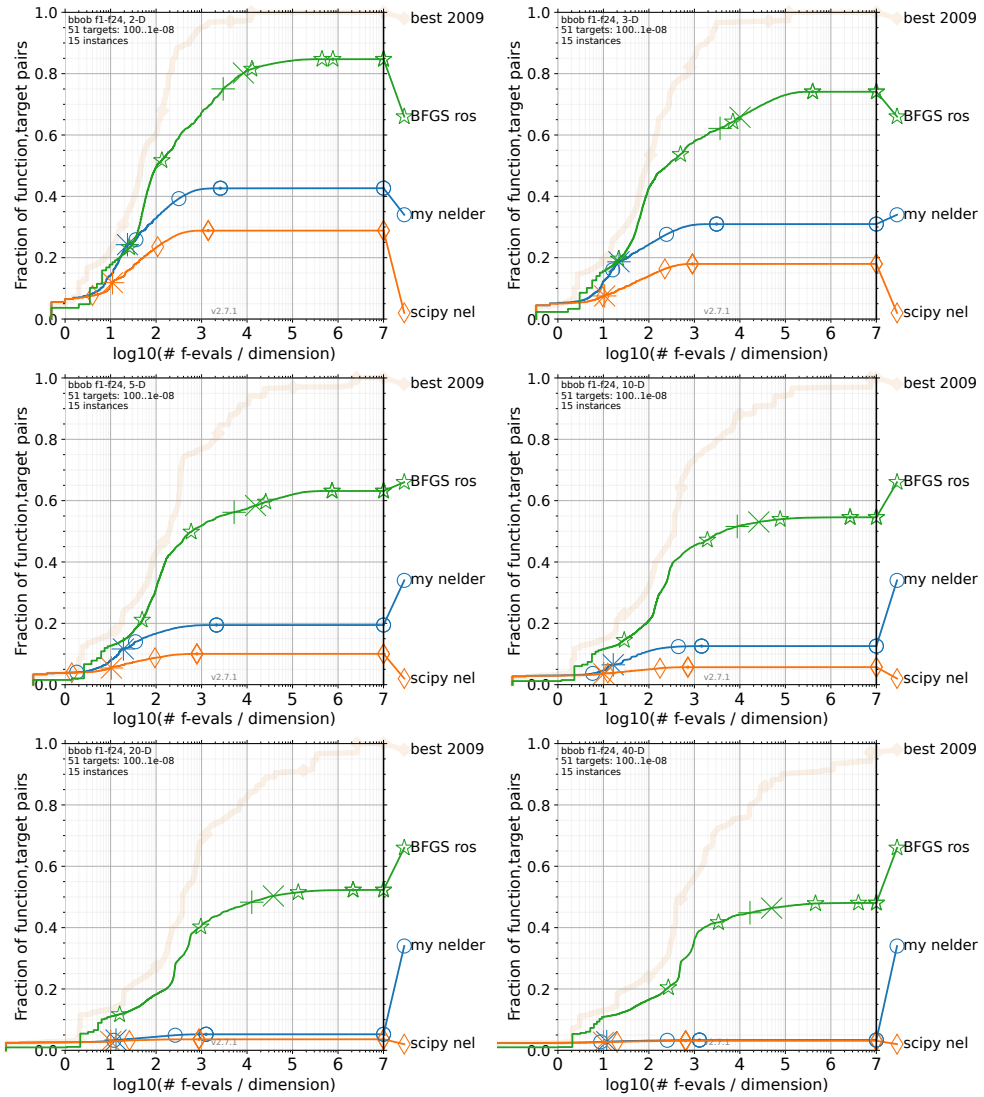


Figure 2: Comparison of different implementations of Nelder-Mead.

```
            ### input: the next three lines need to be adapted to the specific fmin
            # xopt = fmin(problem, repeater.initial_solution_proposal(problem), disp=False)
                 # could depend on budget_multiplier
            xopt = algo(problem, repeater.initial_solution_proposal(problem), problem.
                dimension * budget_multiplier) # could depend on budget_multiplier
            problem(xopt) # make sure the returned solution is evaluated

            repeater.track(problem) # track evaluations and final_target_hit
            minimal_print(problem) # show progress
    result_folders += [observer.result_folder]

### post-process data
dsl = cocopp.main(" ".join(result_folders) + ' bfgs!') # re-run folders look like
    "...-001" etc
```