AI505/AI801, Optimization – Exercise Sheet 05

2026-03-18

> Solutions 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 $^+$ (7.1)

Direct methods are able to use only zero-order information, that is, only 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?

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

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

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, as direct methods need not estimate the derivative or Hessian at each step.

Exercise 2

Below you find an implementation of Nelder-Mead algorithm in Python. Analyze it and use it to solve a few Benchmark functions for unconstrained continuous optimization. You can use the Python implementations.

Plot the evolution of the simplex throughout the search (you can get help from AI assistants to code the plotting facilities).

Then, compare the results with the implementation of Nelder-Mead in the scipy library. You can use the COCO test suite to carry out this part or write yourself everything you need.

```
import numpy as np

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)

    return S[np.argmin(y_arr)], simplex_history
```

For an implementation of the benchmark functions you can install the landscapes library via pip:

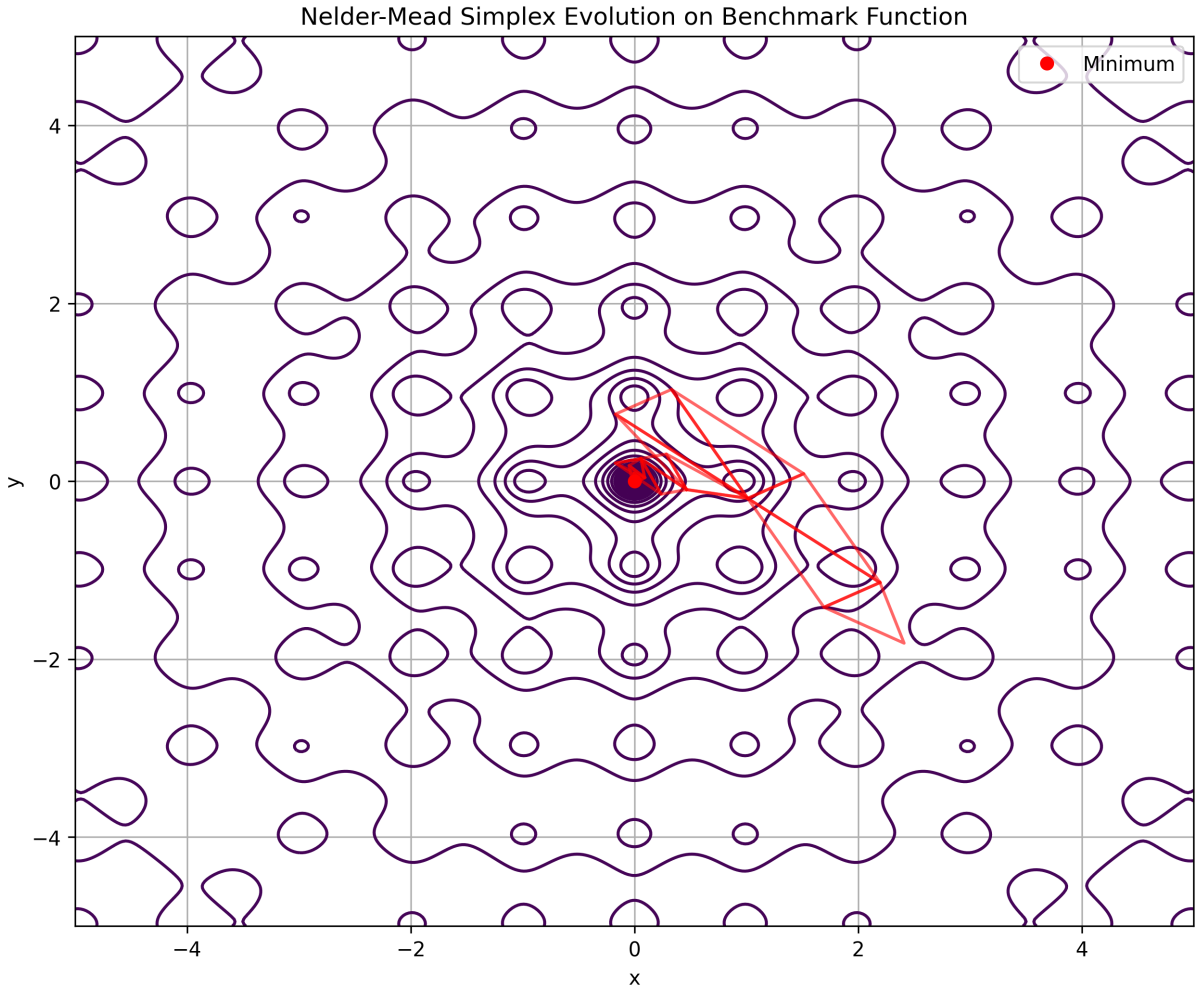
```
pip install "git+https://github.com/belzebuu/landscapes.git"
```

```python
import numpy as np
import matplotlib.pyplot as plt
from landscapes.single_objective import rosenbrock, booth, ackley

# Test function: Rosenbrock function
# #def rosenbrock(x):
#     return (1 - x[0])**2 + 100 * (x[1] - x[0]**2)**2

func = ackley
#func = rosenbrock
#func = booth

# Initial simplex
#S = np.array([[2, -2], [0, -4], [2, -4]])
# Random simplex in the quadrant [0,4]x[-4,0]
np.random.seed(0)  # For reproducibility
S = np.random.rand(3, 2) * 4 + np.array([0,-4])

epsilon = 1e-6

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

# Plotting
x = np.linspace(-5, 5, 400)
y = np.linspace(-5, 5, 400)
X, Y = np.meshgrid(x, y)
#Z = (1 - X)**2 + 100 * (Y - X**2)**2
Z = func(np.array([X, Y]))


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

for simplex in simplex_history:
    plt.plot(*zip(*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 Benchmark Function")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)
plt.show()
#plt.savefig("nelder.png")
```

```
Minimum found at: [1.13374063e-06 1.46348588e-06]
Function value at minimum: 5.23624156301139e-06
[ 2.19525402 -1.13924253  2.4110535  -1.82046727  1.6946192  -1.41642355]
[ 2.19525402 -1.13924253  1.6946192  -1.41642355  1.01270281 -0.19256459]
[ 1.01270281 -0.19256459  2.19525402 -1.13924253  1.51333763  0.08461643]
```

```
[ 1.01270281 -0.19256459  1.51333763  0.08461643  0.33078642  1.03129437]
[ 1.01270281 -0.19256459  0.33078642  1.03129437 -0.16984839  0.75411336]
[ 1.01270281 -0.19256459 -0.16984839  0.75411336  0.4667476  -0.09448561]
[ 0.4667476  -0.09448561  1.01270281 -0.19256459  0.28493841  0.30529413]
[ 0.4667476  -0.09448561  0.28493841  0.30529413  0.0574131   0.25438869]
[ 0.0574131   0.25438869  0.4667476  -0.09448561  0.23922229 -0.14539105]
[ 0.0574131   0.25438869  0.23922229 -0.14539105 -0.17011221  0.20348324]
[ 0.0574131   0.25438869 -0.17011221  0.20348324  0.09143637  0.04177246]
[ 0.09143637  0.04177246  0.0574131   0.25438869 -0.04784374  0.17578191]
[ 0.09143637  0.04177246 -0.04784374  0.17578191 -0.01382047 -0.03683432]
[-0.01382047 -0.03683432  0.09143637  0.04177246 -0.00451789  0.08912549]
[-0.01382047 -0.03683432 -0.00451789  0.08912549  0.0411336   0.03395902]
[-0.01382047 -0.03683432  0.0411336   0.03395902  0.00456934  0.04384392]
[-0.01382047 -0.03683432  0.00456934  0.04384392  0.01825402  0.01873191]
[ 0.01825402  0.01873191 -0.01382047 -0.03683432  0.00339306  0.01739635]
[ 0.00339306  0.01739635  0.01825402  0.01873191 -0.00149846 -0.0093851 ]
[-0.00149846 -0.0093851   0.00339306  0.01739635 -0.00770606 -0.00335751]
[-0.00770606 -0.00335751 -0.00149846 -0.0093851  -0.0006046   0.00551253]
[-0.0006046   0.00551253 -0.00770606 -0.00335751 -0.0028269  -0.00415379]
[-0.0028269  -0.00415379 -0.0006046   0.00551253  0.0012794   0.0026978 ]
[ 0.0012794   0.0026978  -0.0028269  -0.00415379 -0.00068918  0.00239227]
[-0.00068918  0.00239227  0.0012794   0.0026978  -0.00126589 -0.00080438]
[-0.00126589 -0.00080438 -0.00068918  0.00239227  0.00015093  0.00174587]
[-0.00126589 -0.00080438  0.00015093  0.00174587 -0.00042578 -0.00145077]
[-0.00126589 -0.00080438 -0.00042578 -0.00145077 -0.00034745  0.00030915]
[-0.00034745  0.00030915 -0.00126589 -0.00080438 -0.00061623 -0.00084919]
[-0.00034745  0.00030915 -0.00061623 -0.00084919  0.00030221  0.00026433]
[ 0.00030221  0.00026433 -0.00034745  0.00030915 -0.00031942 -0.00028123]
[ 0.00030221  0.00026433 -0.00031942 -0.00028123  0.00016082 -0.00016724]
[ 1.60819737e-04 -1.67242616e-04  3.02214744e-04  2.64334920e-04
 -4.39526387e-05 -1.16339857e-04]
[-4.39526387e-05 -1.16339857e-04  1.60819737e-04 -1.67242616e-04
  1.80324147e-04  6.12718416e-05]
[-4.39526387e-05 -1.16339857e-04  1.80324147e-04  6.12718416e-05
 -2.44482288e-05  1.12174601e-04]
[-2.44482288e-05  1.12174601e-04 -4.39526387e-05 -1.16339857e-04
  7.30618564e-05  2.95946067e-05]
[ 7.30618564e-05  2.95946067e-05 -2.44482288e-05  1.12174601e-04
 -9.82291246e-06 -2.27276266e-05]
[-9.82291246e-06 -2.27276266e-05  7.30618564e-05  2.95946067e-05
  3.58562161e-06  5.78040453e-05]
[-9.82291246e-06 -2.27276266e-05  3.58562161e-06  5.78040453e-05
  3.49716055e-05  2.35664081e-05]
[-9.82291246e-06 -2.27276266e-05  3.49716055e-05  2.35664081e-05
  8.07998406e-06  2.91117180e-05]
[-9.82291246e-06 -2.27276266e-05  8.07998406e-06  2.91117180e-05
 -1.87929990e-05 -6.99513544e-06]
[-1.87929990e-05 -6.99513544e-06 -9.82291246e-06 -2.27276266e-05
 -3.11398584e-06  7.12516852e-06]
[-3.11398584e-06  7.12516852e-06 -1.87929990e-05 -6.99513544e-06
 -1.03882024e-05 -1.13313050e-05]
[-3.11398584e-06  7.12516852e-06 -1.03882024e-05 -1.13313050e-05
  5.29081075e-06  2.78899893e-06]
```

```
[ 5.29081075e-06  2.78899893e-06 -3.11398584e-06  7.12516852e-06
 -4.64989500e-06 -3.18711065e-06]
[-4.64989500e-06 -3.18711065e-06  5.29081075e-06  2.78899893e-06
 -1.39676398e-06  3.46305633e-06]
[-1.39676398e-06  3.46305633e-06 -4.64989500e-06 -3.18711065e-06
  1.13374063e-06  1.46348588e-06]
[ 1.13374063e-06  1.46348588e-06 -1.39676398e-06  3.46305633e-06
 -2.39070334e-06 -3.61919772e-07]
[ 1.13374063e-06  1.46348588e-06 -2.39070334e-06 -3.61919772e-07
  1.39801277e-07 -2.36149021e-06]
```



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"
    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-paramet
#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

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

Comparison of different implementations of Nelder-Mead.

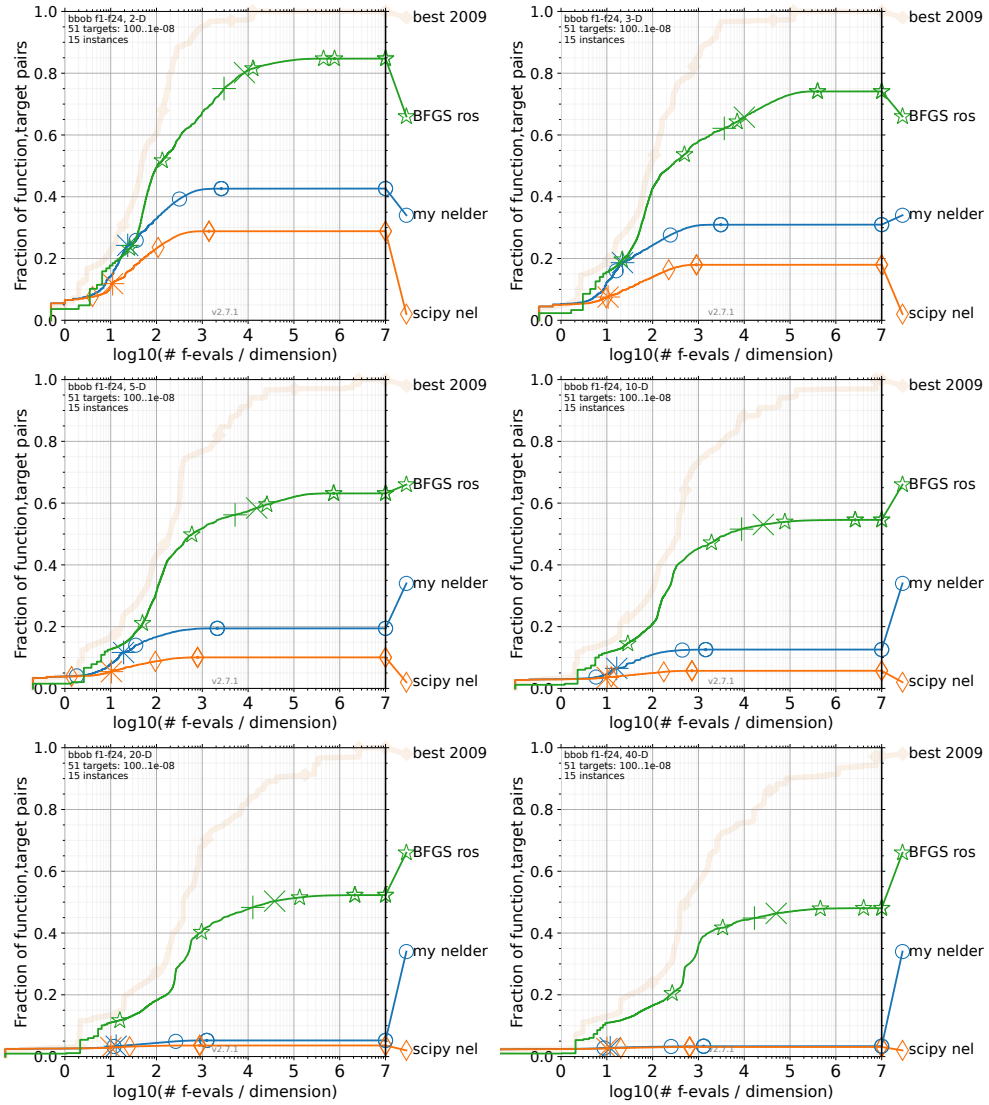


Figure 1

Exercise 3 *

The Nelder-Mead algorithm has three parameters α, β, and γ. How would you approach the problem of tuning these parameters?

The optimal value of the hyperparameters depends on the function to optimize (aka, problem instance). If we had a single instance tuning would correspond to a hyperparameter optimization problem. The objective function $h : \mathbb{R}^3 \to \mathbb{R}$ would associate to a design point (α, β, γ) the running time (or number of function evaluations) of Nelder-Mead to terminate. It is a continuous optimization problem itself. Differently from the problems we have approached so far, the objective function of this problem and its derivatives would be unknown or difficult to compute (it correspoonds to performing a full Nelder-Mead

run). By necessity it would be a black-box optimization problem. Moreover, likely two executions of the same design point would give different measurements and hence the objective function should be also regarded as stochastic. However, we are interested in the performance of the algorithm across a class of problems and to generalize this performance to unseen functions. We would need to consider a distribution of functions and the objective function would be the expected running time across this distribution. This is a more complex problem, as we would need to sample functions from the distribution and run Nelder-Mead on them to get an estimate of the expected running time for each design point. We could use techniques from machine learning, such as Bayesian optimization, to efficiently explore the hyperparameter space and find good values for α, β, and γ.

Exercise 4 *

Consider the natural evolutionary strategy for an univariate function. Assume the univariate normal distribution as proposal distribution $p(x \mid \boldsymbol{\theta}) = N(x \mid \mu, \sigma^2)$.

- Derive the update rule for θ
- If after a number of iterations the value of μ becomes equal to x^*, that is, the minimum of f, what will be the update rule for σ^2 and what will be the difficulty encountered by the algorithm?

The Gaussian distribution with mean μ and variance σ^2 has probability density function:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

The natural evolution strategy algorithm minimizes f by solving this problem:

$$\min E_{\boldsymbol{x}\sim p(\cdot|\boldsymbol{\theta})}[f(\boldsymbol{x})]$$

by gradient descent, ie, using the following update rule:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \nabla_{\boldsymbol{\theta}} E_{\boldsymbol{x}\sim p(\cdot|\boldsymbol{\theta})}[f(\boldsymbol{x})]$$

In the text book it is shown that

$$\nabla_{\boldsymbol{\theta}} E_{\boldsymbol{x}\sim p(\cdot|\boldsymbol{\theta})}[f(\boldsymbol{x})] = \frac{1}{N}\sum_{i=1}^{N} f(\boldsymbol{x}_i) \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{x}_i, \boldsymbol{\theta}_k)$$

and hence the update rule becomes

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \frac{1}{N}\sum_{i=1}^{N} f(\boldsymbol{x}_i) \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{x}_i, \boldsymbol{\theta}_k)$$

When $p(\boldsymbol{x}_i, \boldsymbol{\theta}_k)$ is the univariate Gaussian distribution with mean μ and variance σ^2 the probability density function is:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

and the log-likelihood of a single value x_i sampled from it is:

$$\log L(x_i \mid \mu, \sigma^2) = \log p(x_i \mid \mu, \sigma^2) = -\frac{1}{2}\ln 2\pi - \frac{1}{2}\ln \sigma^2 - \frac{(x_i-\mu)^2}{2\sigma^2}$$

For a set of points $\boldsymbol{x} = \{x_1, \ldots, x_N\}$ drawn conditionally independently from the distribution we have

$$p(\boldsymbol{x} \mid \mu, \sigma^2) = \prod_{i=1}^{N} p(x_i \mid \mu, \sigma^2)$$

$$\log L(\boldsymbol{x} \mid \mu, \sigma^2) = \log \prod_{i=1}^{N} p(x_i \mid \mu, \sigma^2) = \sum_{i=1}^{N} \left(-\frac{1}{2} \ln 2\pi - \frac{1}{2} \ln \sigma^2 - \frac{(x_i - \mu)^2}{2\sigma^2} \right)$$

Hence,

$$\begin{aligned} \frac{\partial \log L(x|\mu,\sigma^2)}{\partial \mu} &= x - \mu \\ \frac{\partial \log L(x|\mu,\sigma^2)}{\partial \sigma^2} &= -\frac{1}{2\sigma^2} + \frac{(x-\mu)^2}{2\sigma^4} \end{aligned}$$

The second term will be zero if the mean is already optimal. Thus, the derivative is $-1/2\sigma^2$ and decreasing σ^2 will increase the likelihood of drawing elite samples. Unfortunately, σ^2 is optimized by approaching arbitrarily close to zero. The asymptote near zero in the gradient update will lead to large step sizes, which cannot be taken as σ^2 must remain positive.

Exercise 5 * (8.4)

The maximum likelihood estimates are the parameter values that maximize the likelihood of sampling the points $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$.

In the slides it was given the analytical solution for the mean and variance of a multivariate Gaussian distribution: $N(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma)$.

Derive here those results that are used in the cross-entropy method that uses multivariate normal distributions.

The log-likelihood of a multivariate Gaussian Distribution is:

$$\log L(\boldsymbol{x} \mid \boldsymbol{\mu}, \Sigma) = -\frac{1}{2} \ln 2\pi - \frac{1}{2} \ln \Sigma - \frac{(\boldsymbol{x} - \mu)^2}{2}$$

We compute the gradient using the facts that $\nabla_{\boldsymbol{z}} \boldsymbol{z}^T A \boldsymbol{z} = (A + A^T)\boldsymbol{z}$, that $\nabla_{\boldsymbol{z}} \boldsymbol{a}^T \boldsymbol{z} = \boldsymbol{a}$, and that Σ is symmetric and positive definite, and thus Σ^{-1} is symmetric.

Given a data set $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)^{\mathrm{T}}$ in which the observations $\{\mathbf{x}_n\}$ are assumed to be drawn independently from a multivariate Gaussian distribution, we can estimate the parameters of the distribution by maximum likelihood. The log likelihood function is given by

$$\ln p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{ND}{2}\ln(2\pi) - \frac{N}{2}\ln|\boldsymbol{\Sigma}| - \frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n - \boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}). \quad (2.118)$$

By simple rearrangement, we see that the likelihood function depends on the data set only through the two quantities

$$\sum_{n=1}^{N}\mathbf{x}_n, \qquad \sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^{\mathrm{T}}. \quad (2.119)$$

These are known as the *sufficient statistics* for the Gaussian distribution. Using (C.19), the derivative of the log likelihood with respect to $\boldsymbol{\mu}$ is given by

$$\frac{\partial}{\partial\boldsymbol{\mu}}\ln p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N}\boldsymbol{\Sigma}^{-1}(\mathbf{x}_n - \boldsymbol{\mu}) \quad (2.120)$$

and setting this derivative to zero, we obtain the solution for the maximum likelihood estimate of the mean given by

$$\boldsymbol{\mu}_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n \quad (2.121)$$

which is the mean of the observed set of data points. The maximization of (2.118) with respect to $\boldsymbol{\Sigma}$ is rather more involved. The simplest approach is to ignore the symmetry constraint and show that the resulting solution is symmetric as required. Alternative derivations of this result, which impose the symmetry and positive definiteness constraints explicitly, can be found in Magnus and Neudecker (1999). The result is as expected and takes the form

$$\boldsymbol{\Sigma}_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})(\mathbf{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})^{\mathrm{T}} \quad (2.122)$$

which involves $\boldsymbol{\mu}_{\mathrm{ML}}$ because this is the result of a joint maximization with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. Note that the solution (2.121) for $\boldsymbol{\mu}_{\mathrm{ML}}$ does not depend on $\boldsymbol{\Sigma}_{\mathrm{ML}}$, and so we can first evaluate $\boldsymbol{\mu}_{\mathrm{ML}}$ and then use this to evaluate $\boldsymbol{\Sigma}_{\mathrm{ML}}$.

If we evaluate the expectations of the maximum likelihood solutions under the true distribution, we obtain the following results

$$\mathbb{E}[\boldsymbol{\mu}_{\mathrm{ML}}] = \boldsymbol{\mu} \quad (2.123)$$

$$\mathbb{E}[\boldsymbol{\Sigma}_{\mathrm{ML}}] = \frac{N-1}{N}\boldsymbol{\Sigma}. \quad (2.124)$$

We see that the expectation of the maximum likelihood estimate for the mean is equal to the true mean. However, the maximum likelihood estimate for the covariance has an expectation that is less than the true value, and hence it is biased. We can correct this bias by defining a different estimator $\widetilde{\boldsymbol{\Sigma}}$ given by

$$\widetilde{\boldsymbol{\Sigma}} = \frac{1}{N-1}\sum_{n=1}^{N}(\mathbf{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})(\mathbf{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})^{\mathrm{T}}. \quad (2.125)$$

Clearly from (2.122) and (2.124), the expectation of $\widetilde{\boldsymbol{\Sigma}}$ is equal to $\boldsymbol{\Sigma}$.

Figure 2: Log-Likelihood Function

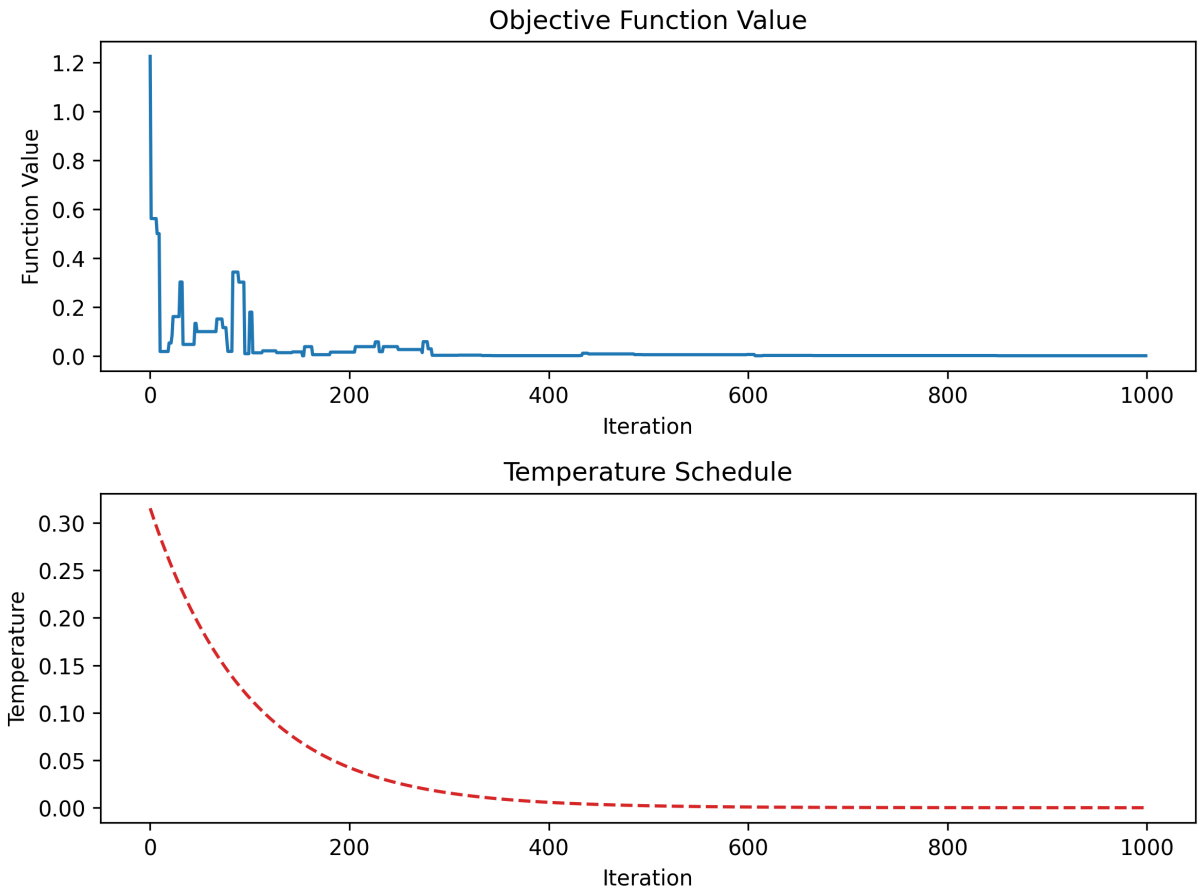
Exercise 6 *

Implement Simulated Annealing. Set the initial temperature such that the initial acceptance ratio is 0.2 and the annealing plan to exponential with cooling rate $\gamma = 0.99$. Apply the algorithm to the Rosenbrock function and plot the value of the function and the temperature throughout the iterations.

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

def compute_initial_temperature(objective, bounds):
    dim = len(bounds)

    deltas = []

    for r in range(100): # restarts
        current_solution = np.array([np.random.uniform(b[0], b[1]) for b in bounds])
        current_value = objective(current_solution)

        for i in range(100):
            # Generate a new candidate solution
            new_solution = current_solution + np.random.normal(0, 0.1, size=dim)  # Small perturba

            # Ensure the new solution is within bounds
            new_solution = np.clip(new_solution, [b[0] for b in bounds], [b[1] for b in bounds])
            new_value = objective(new_solution)

            # store deltas
            delta = new_value - current_value
            deltas.append(delta)

    return -np.mean(deltas)/np.log(0.2)

def simulated_annealing(objective, bounds, initial_temp=1000, cooling_rate=0.99, max_iter=1000):
    """
    Simulated Annealing for continuous optimization.

    Parameters:
        objective (function): The objective function to minimize.
        bounds (list of tuples): Bounds for each dimension [(min, max), ...].
        initial_temp (float): Starting temperature.
        cooling_rate (float): Cooling rate (should be between 0 and 1).
        max_iter (int): Maximum number of iterations.

    Returns:
        best_solution (numpy array): The best solution found.
        best_value (float): The corresponding objective function value.
        history (list): Values of the objective function during iterations.
        temp_history (list): Temperature values during iterations.
    """
    # Initialize solution randomly within bounds
    dim = len(bounds)
    current_solution = np.array([np.random.uniform(b[0], b[1]) for b in bounds])
    current_value = objective(current_solution)

    best_solution, best_value = current_solution, current_value

    temp = initial_temp
    history = []
    temp_history = []

    for i in range(max_iter):
        # Store history
        history.append(current_value)
        temp_history.append(temp)

        # Generate a new candidate solution
        new_solution = current_solution + np.random.normal(0, 0.1, size=dim)  # Small perturbation
```

```
bounds = [(-2, 2), (-2, 2)]  # 2D problem
initial_temperature = compute_initial_temperature(rosenbrock_function, bounds)
print("Initial temparature:", initial_temperature)
best_sol, best_val, history, temp_history = simulated_annealing(rosenbrock_function, bounds, \
    initial_temp = initial_temperature)

# Plot function value and temperature over iterations
fig, axes = plt.subplots(2, 1, figsize=(8, 6))

axes[0].plot(history, color="tab:blue")
axes[0].set_xlabel("Iteration")
axes[0].set_ylabel("Function Value")
axes[0].set_title("Objective Function Value")

axes[1].plot(temp_history, color="tab:red", linestyle="dashed")
axes[1].set_xlabel("Iteration")
axes[1].set_ylabel("Temperature")
axes[1].set_title("Temperature Schedule")

plt.tight_layout()
plt.show()
#plt.savefig("simann.png")

print(f"Best solution: {best_sol}")
print(f"Best value: {best_val}")
```

```
Initial temparature: 0.31551404681494
Best solution: [0.99091247 0.98309255]
Best value: 0.0002230096947819989
```


Objective Function Value
Function Value
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
Iteration
Temperature Schedule
Temperature
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
200
400
600
800
1000
Iteration