Nonlinear Optimization – Sheet 03

Exercise 1

Exercise 2

Exercise 3

Exercise 4

In order to use the line search method, we implement the Armijo procedures.

```
"""
define armijo algorithms to find a suitable step size
"""

def armijo_condition(alpha, phi_alpha, phi_0, phi_prime_0, sigma):
    """evaluate armijo condition"""
    return phi_alpha <= phi_0 + sigma * alpha * phi_prime_0


def armijo_backtracking(alpha_0, phi, phi_0, phi_prime_0, sigma, beta):
    """
    return a step size alpha that satisfies the armijo condition
    using the simple backtracking approach
    """
    alpha = alpha_0
    while not armijo_condition(alpha, phi(alpha), phi_0, phi_prime_0,
        sigma):
        alpha = beta * alpha
    return alpha

def armijo_interpolation(alpha_0, phi, phi_0, phi_prime_0, sigma,
    beta_lower, beta_upper):
    """
    return a step size alpha that satisfies the armijo condition
    use the interpolation approach
    """
    alpha = alpha_0
    phi_alpha = phi(alpha)
    while not armijo_condition(alpha, phi_alpha, phi_0, phi_prime_0,
        sigma):
        alpha_star = (-phi_prime_0 * alpha**2)/2*(phi_alpha - phi_0 -
            phi_prime_0 * alpha)
        alpha = min(max(alpha_star, beta_lower * alpha), beta_upper *
            alpha) #clip alpha to interval
        phi_alpha = phi(alpha)
    return alpha
```

Now we can implement the whole algorithm

```
import numpy as np
from armijo_procedures import armijo_backtracking, armijo_interpolation

def gradient_descent_UP(
    x_0, f, f_prime, M_inv, sigma, alpha_lower_bound, beta, beta_upper=
        None, eps=1e-5, max_iter=100
):
```

```
    """
    if beta_upper is given, we use the interpolating armijo algorithm
        with beta as beta_lower,
    otherwise the original armijo algorithm
    """

    interpolate = False
    if beta_upper:
        interpolate = True
        beta_lower = beta

    k = 0
    f_new = f(x_0)
    r = f_prime(x_0)
    d = -M_inv @ r
    delta = -r.transpose() @ d
    history = {
        "iterates": [x_0],
        "objective_values": [f_new],
        "gradient_norms": [np.sqrt(delta)],
        "step_lengths": [],
    }
    x = x_0
    while delta > eps**2 and k < max_iter:
        if k == 0:
            alpha_0 = alpha_lower_bound
        else:
            alpha_0 = max(alpha_lower_bound, (f_new - f_old) / delta)
        phi = lambda alpha: f(x + alpha * d)
        phi_0 = f_new  # f(x + 0*d) = f(x)
        phi_prime_0 = -delta
        if interpolate:
            alpha = armijo_interpolation(
                alpha_0, phi, phi_0, phi_prime_0, sigma, beta_lower,
                    beta_upper
            )
        else:
            alpha = armijo_backtracking(alpha_0, phi, phi_0,
                phi_prime_0, sigma, beta)

        x = x + alpha * d
        f_old = f_new
        f_new = f(x)
        r = f_prime(x)
        d = -M_inv @ r
        delta = -r.transpose() @ d
        k = k + 1

        history["step_lengths"].append(alpha)
        history["iterates"].append(x)
        history["objective_values"].append(f_new)
        history["gradient_norms"].append(np.sqrt(delta))

    return history
```

As examples we use the rosenbrock function from example_functions.py and some random quadratic problems from rand_problem.py.

```
import numpy as np

def rosenbrock(a,b,x):
        """
        Implements the rosenbrock function
        Accepts:
                        a,b: scalar parameters
                        x: array of length 2
        Returns:
                        f:  function value
                        df: derivative value
        """
        f  = (a - x[0])**2 + b * (x[1] - x[0]**2)**2
        df = np.array(
                [2*(a - x[0]) + 2*b*(x[1] - x[0]**2)*(-2*x[0]),
                 2*b*(x[1] - x[0]**2)
                ]
                )
        return f, df

def himmelblau(x):
        f = (x[0]**2 + x[1]-11)**2 + (x[0]+x[1]**2-7)**2
        df = 0 #if I have some spare time maybe I'll compute it

import numpy as np

class rand_problem():
    def __init__(self, n) -> None:
        self.n = n
        self.A = self.create_random_A()
        self.b = np.random.rand(n)
        self.c = np.random.rand()
        self.f = self.quadratic_function()
        self.f_prime = lambda x : self.A @ x - self.b
        self.Pinv = np.identity(n)

        self.x0 = np.random.rand(n)

    def create_random_A(self):
        """create random spd matrix in dimension n x n"""
        M = np.random.rand(self.n,self.n)
        return np.dot(M, M.T)

    def quadratic_function(self):
        f = lambda x : 0.5 * x.T @ self.A @ x - self.b.T @ x + self.c
        return f

if __name__ == "main":
    x = rand_problem(4)
    print(x.A, x.b, x.c)
```

Finally, we can put all of it together to create some plots

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

from visualization_functions import (
```

```
    plot_2d_iterates_contours,
    plot_f_val_diffs,
    plot_step_sizes,
    plot_grad_norms,
)
from gradient_descent_UP import gradient_descent_UP
from rand_problem import rand_problem
from example_functions import rosenbrock


# compare gradient norms
N = 10
problems = []

for i in range(N):
    problems.append(rand_problem(2))

# normalize norm of random start point
for problem in problems:
    problem.x0 = problem.x0 * (np.linalg.norm(problem.x0)) ** (-1)

histories = []
# labels = []
for problem in problems:
    histories.append(
        gradient_descent_UP(
            problem.x0,
            problem.f,
            problem.f_prime,
            problem.Pinv,
            sigma=1e-3,
            alpha_lower_bound=10,
            beta=0.5,
            #beta_upper=0.9,
        )
    )

plot_grad_norms(
   histories=histories, labels=range(len(histories))
)  # Gradient norms - gradient descent algorithm

# for i, problem in enumerate(problems):
#     plot_2d_iterates_contours(problem.f, histories=[histories[i]],
    labels=[str(i)], xlims=[-10,10], ylims=[-10,10])

a = 1
b = 100
rosenbrock_f = lambda x: rosenbrock(a, b, x)[0]
rosenbrock_prime = lambda x: rosenbrock(a, b, x)[1]

rosenbrock_histories = []
rosenbrock_labels = []
configurations = [
    ([2, 2], 1e-2, 0.01, 0.5),
    ([2, 2.5], 1e-2, 0.1, 0.5),
    ([2.5, 2], 1e-4, 0.01, 0.5),
```

```
    ([2.5, 2.5], 1e-4, 0.1, 0.5),
]
for configuration in configurations:
    rosenbrock_histories.append(
        gradient_descent_UP(
            configuration[0],
            rosenbrock_f,
            rosenbrock_prime,
            np.identity(2),
            sigma=configuration[1],
            alpha_lower_bound=configuration[2],
            beta=configuration[3],
            max_iter=100,
        )
    )
    rosenbrock_labels.append(
        f"x0: {configuration[0]}, sigma: {configuration[1]}, alpha: {
            configuration[2]}, beta: {configuration[3]}"
    )


plot_grad_norms(
    histories=rosenbrock_histories,
    labels=rosenbrock_labels,
)

plot_2d_iterates_contours(
    rosenbrock_f,
    histories=rosenbrock_histories,
    labels=rosenbrock_labels,
    xlims=[-3, 3],
    ylims=[-2,4],
    title="Iterates and iso-lines of Rosenbrock function"
)

plt.show()
```

First, we look at the rosenbrock function. We use $\beta = .5$ and vary some other parameters, see 1. Then, we consider some random quadratic problems with varying parameters.

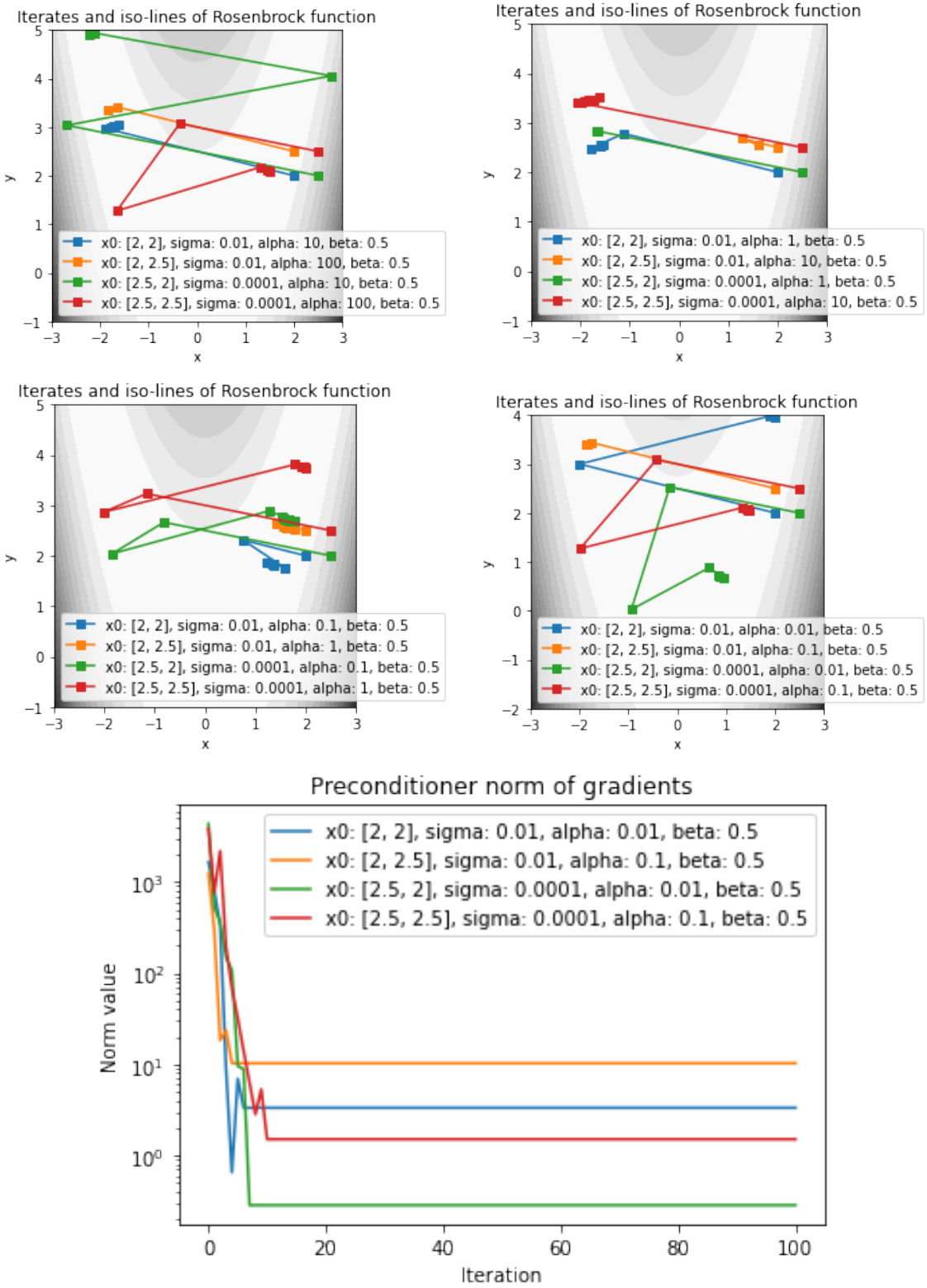
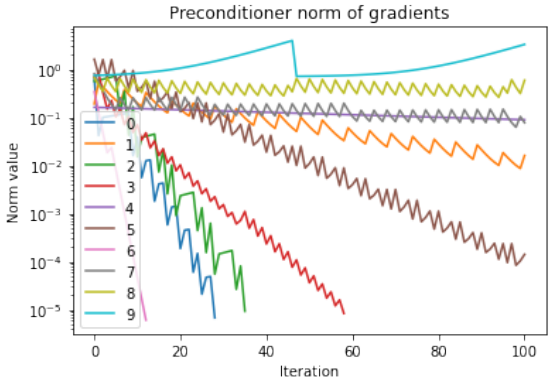
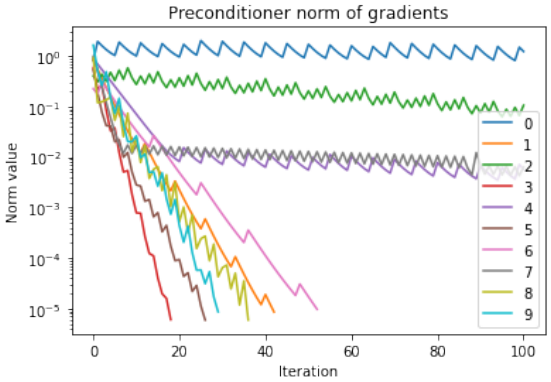


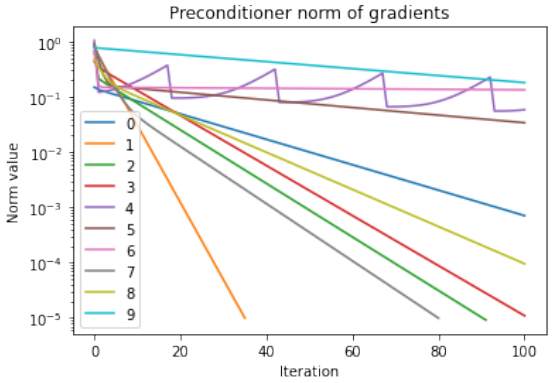
Abbildung 1: (a) - (d): Varying α and σ for the rosenbrock function, (e): gradient norms displayed for the configurations from plot (d)



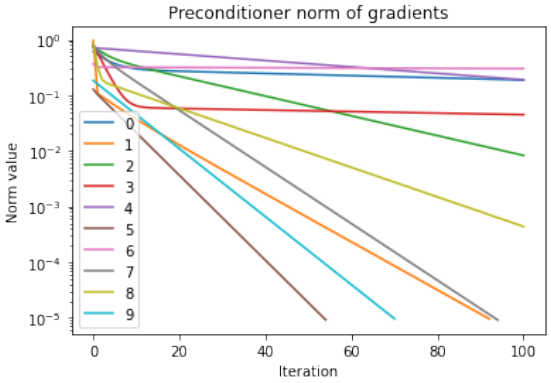
(a) $\alpha = 10, \sigma = 10^{-3}$



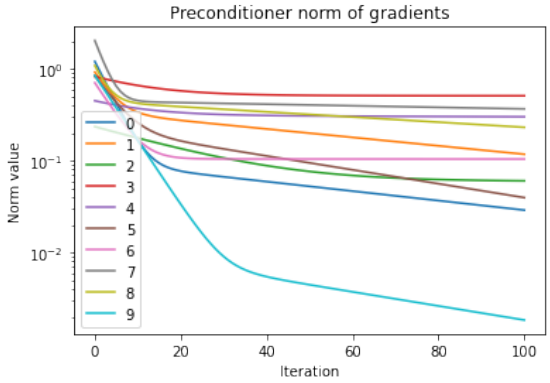
(b) $\alpha = 10, \sigma = 10^{-1}$



(c) $\alpha = 1, \sigma = 10^{-2}$



(d) $\alpha = 1, \sigma = 10^{-1}$



(e) $\alpha = 0.1, \sigma = 10^{-2}$

Abbildung 2: Varying α and σ for 10 random quadratic problems (the random problems differ for each plot)