Nonlinear Optimization – Sheet 05

Exercise 1

Let $A \in \mathbb{R}^{n\times n}$ be regular, $b \in \mathbb{R}^n$ and consider a sequence $(x^{(k)})_{k\geq 0}$ generated by Newtons method starting from $x^{(0)} \in \mathbb{R}^n$. Prove that

(i) Newons method for the function

$$G : \mathbb{R}^n \to \mathbb{R}^n, \quad G(y) = F(Ay + b)$$

with initial value $y^{(0)} \in \mathbb{R}^n$ s.t. $x^{(0)} = Ay^{(0)} + b$ is well defined and produces a sequence $(y^{(k)})_{k\geq 0}$ s.t.

$$x^{(k)} = Ay^{(k)} + b. \tag{$*$}$$

Proof. We show $(*)$ by induction on k. For $k = 0$ the assertion holds by assumption. A calculation shows

$$\begin{aligned} y^{(k+1)} &= y^{(k)} - DG(y^{(k)})^{-1}G(y^{(k)}) = y^{(k)} - A^{-1}DF(Ay^{(k)} + b)^{-1}F(Ay^{(k)} + b) \\ &= y^{(k)} - A^{-1}DF(x^{(k)})^{-1}F(x^{(k)}) \\ \implies Ay^{(k+1)} &= x^{(k)} - b - DF(x^{(k)})^{-1}F(x^{(k)}) = x^{(k+1)} - b. \end{aligned}$$

Because $DG(y^{(k)}) = DF(Ay^{(k)} + b)A = DF(x^{(k)})A$ and $\det A \neq 0$ we see that Newtons method is well-defined. □

(ii) Newtons method for the function

$$H : \mathbb{R}^n \to \mathbb{R}^n, \quad y \mapsto AF(y)$$

with inital value $y^{(0)} \in \mathbb{R}^n$ s.t. $x^{(0)} = y^{(0)}$ is well defined and produces a sequence of iterates $(y^{(k)})_{k\geq 0}$ s.t. $x^{(k)} = y^{(k)}$.

Proof. Suppose $y^{(k)} = x^{(k)}$ for some $k \geq 0$ (for $k = 0$ by assumption). Then

$$y^{(k+1)} = y^{(k)} - DH(y^{(k)})^{-1}H(y^{(k)}) = x^{(k)} - DF(x^{(k)})^{-1}A^{-1}AF(x^{(k)}) = x^{(k+1)}.$$

As A is regular $\implies$ $DH(y^{(k)}) = A \cdot DF(x^{(k)}$ is regular for all $k \geq 0$. □

(iii) Explain why we can not expect a similar transformation result to hold for the iterates of the Newton method when we expand the transformation in (ii) as in

$$H : \mathbb{R}^n \to \mathbb{R}^n, \quad H(y) = AF(y) + b.$$

Explanation. If we evaluate for $y \in \mathbb{R}^n$ with $\det DH(y) \neq 0$ the Newton step direction we get

$$DH(y^{-1})H(y) = DF(y)^{-1}A^{-1}(AF(y) + b) = DF(y)^{-1}F(y) + DF(y)^{-1}A^{-1}b$$

and the latter term causes the issue because it also stacks with every further iteration step. □

Exercise 2

(i) is an easy consequence of basic complex analysis.

(ii) see code below

(iii) see code below

```
import numpy as np
import matplotlib.pyplot as plt
def local_newton(x_0, F, F_prime, eps = 1e-4, max_iter = 100):
    x = x_0
    k = 0
    while np.linalg.norm(F(x)) > eps and k < max_iter:
        d = - np.linalg.inv(F_prime(x)) @ F(x)
        x = x + d
    return x

def f(X):
    x = X[0]; y = X[1]
    return np.array([x**3 - 3*x*y**2 - 1, 3*x**2*y - y**3])

def f_prime(X):
    x = X[0]; y = X[1]
    return np.array([[3*x**2 - 3*y**2, -6*x*y], [6*x*y, 3*x**2 - 3*y
        **2]])

third_roots_of_unity = [[1,0],[-.5,np.sqrt(3)/2],[-.5,-np.sqrt(3)]] #
    all three complex roots of z**3 - 1
index_to_color = {0: (255,0,0), 1: (0,255,0), 2: (0,0,255), 3:(0,0,0)}

N = 500
x = np.linspace(-1,1,N)
y = np.linspace(-1,1,N)

xv, yv = np.meshgrid(x,y)
data = np.zeros((N,N,3),dtype=np.uint8)

for index in np.ndindex(N,N):
    x_0 = xv[index]
    y_0 = yv[index]
    X_0 = np.array([x_0, y_0])
    X = local_newton(X_0, f, f_prime)
    min_distance = .1 #sanity check
    zero_index = 3
    for i, zero in enumerate(third_roots_of_unity):
        distance = np.linalg.norm(X - zero)
        if distance < min_distance:
            min_distance = distance
            zero_index = i
    data[index] = index_to_color[zero_index]

plt.imshow(data, interpolation='nearest')
plt.show()
```

Exercise 3

(i) Show that the step length $\alpha^{(k)} = 1$ statisfies the Armijo condition for the Newton direction $d^{(k)} \neq 0$ for the quadratic function

$$f(x) = \frac{1}{2}x^T Ax + b^T x + c$$

with s.p.d. $A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n, c \in \mathbb{R}$ iff $\sigma \leq \frac{1}{2}$.

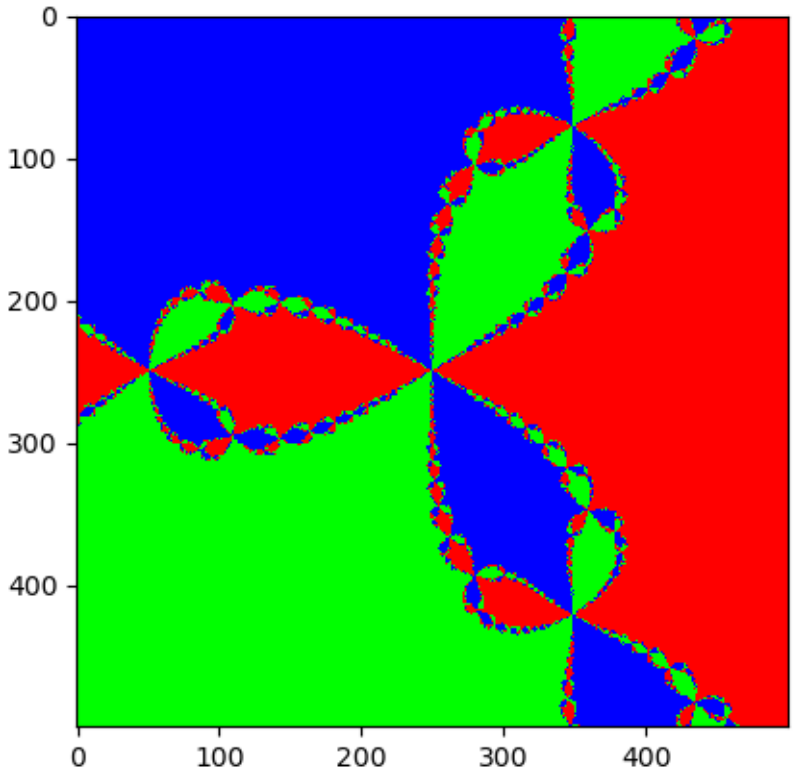


Abbildung 1: Grid from (-1,-1) to (1,1) divided into 500 steps in each dimension. Red is 1, green is $\frac{-1+\sqrt{3}}{2}$ and blue is $\frac{-1-\sqrt{3}}{2}$

Proof. We omit all iteration indices. Armijo condition is equivalent to

$$\begin{aligned} f(x+d) - f(x) = \frac{1}{2}d^T Ad + (Ax+b)^T d = \frac{1}{2}d^t Ad + x^T Ad + b^T d \\ \leq \sigma f'(x)d = \sigma(x^T Ad + b^T d), \end{aligned}$$

where $d = -f''(x)^{-1}\nabla f(x) = -A^{-1}(Ax+b)$, thus $x = -A^{-1}b - d$. Plugging this into the above inequality we see that the Armijo conidtion is equivalent to

$$-\frac{1}{2}d^T Ad \leq \sigma(-d^T Ad) \iff \sigma \leq \frac{1}{2}$$

because A is spd and $d \neq 0$. □

Exercise 4

We use the same code for visualization_functions.py, armijo_procedures.py and gradient_descent_UP.py as in sheet 4. We add the second derivative to the rosenbrock function:

```
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

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)]
          )
        ddf = np.array(
```

```
            [[2 - 4*b*x[1] + 12*b*x[0]**2, -4*b*x[0]],
             [-4*b*x[0], 2*b]]
                )
        return f, df, ddf

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
```

Also, we implement algorithm 5.30:

```
import numpy as np
from armijo_procedures import armijo_backtracking

def globalized_newton_UP(
    x_0, f, f_prime, f_two_prime, M, sigma, eta, rho, p, beta, eps=1e
        -5, max_iter=100
):
    """
    Implements Algorithm 5.30
    """
    debug = False
    M_inv = np.linalg.inv(M)

    k = 0
    f_k = f(x_0)
    r = f_prime(x_0)
    d_G = -M_inv @ r
    delta = -r.transpose() @ d_G
    history = {
        "iterates": [x_0],
        "objective_values": [f_k],
        "gradient_norms": [np.sqrt(delta)],
        "step_lengths": [],
    }
    x = x_0
    while delta > eps**2 and k < max_iter:
        solvable = True
        try:
            if debug == True:
                print(f'solve {f_two_prime(x)} * x = {-r}')
            d_N = np.linalg.solve(f_two_prime(x), -r)
        except:
            solvable = False
            d = d_G
        if solvable:
            if np.dot(f_prime(x), d_N) <= - min(eta, rho * np.linalg.
                norm(d_G)**p) * np.sqrt(delta) * np.sqrt(d_N.T @ M @
                d_N): #Mnorm of d_N
                d = d_N
            else:
                d = d_G

        phi = lambda alpha: f(x + alpha * d)
        phi_0 = f_k  # f(x + 0*d) = f(x)
        phi_prime_0 = -delta
```

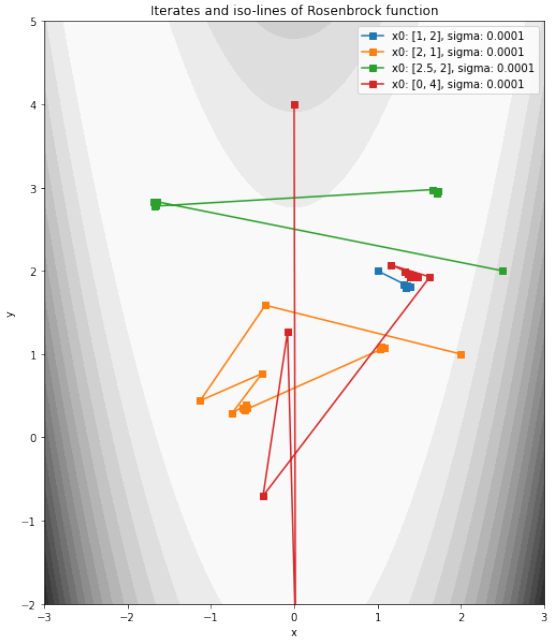
```
        if debug == True:
            case = ''
            if not solvable:
                case = 'unsolvable -> gradient'
            elif np.array_equiv(d, d_G):
                case = 'solvable -> gradient'
            elif np.array_equiv(d, d_N):
                case = 'solvable -> newton'
            else:
                case = 'bug!'
            print(f'{case}, phi_0={phi_0}, phi_prime_0={phi_prime_0} at
                x={x} and d={d}')
        alpha = armijo_backtracking(1, phi, phi_0, phi_prime_0, sigma,
            beta) #initial trial step size is 1

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

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

    return history
```

In 4.py, we implement the code to test algorithm 5.30 on the rosenbrock function an to generate plots (see end of exercise sheet).

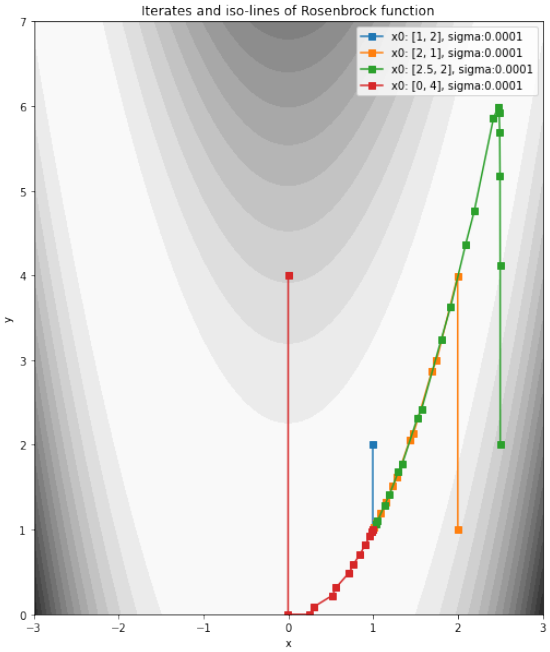
```
import numpy as np
import matplotlib.pyplot as plt

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 globalized_newton_UP import globalized_newton_UP
from example_functions import rosenbrock

a = 1
b = 100
rosenbrock_f = lambda x: rosenbrock(a, b, x)[0]
rosenbrock_prime = lambda x: rosenbrock(a, b, x)[1]
rosenbrock_two_prime = lambda x: rosenbrock(a, b, x)[2]

configurations = [
    ([1, 2], 1e-4),
    ([2, 1], 1e-4),
    ([2.5, 2], 1e-4),
    ([0, 4], 1e-4),
]
```

```
rosenbrock_histories_n = []
rosenbrock_labels_n = []

for configuration in configurations:
    rosenbrock_histories_n.append(
        globalized_newton_UP(
            configuration[0],
            rosenbrock_f,
            rosenbrock_prime,
            rosenbrock_two_prime,
            np.identity(2),
            sigma=1e-4,
            eta=.5,
            rho=1e-6,
            p=.1,
            beta=.5,
            eps=1e-10,
            max_iter=100,
        )
    )
    rosenbrock_labels_n.append(f"x0: {configuration[0]}, sigma:{
        configuration[1]}")

rosenbrock_histories_g = []
rosenbrock_labels_g = []
for configuration in configurations:
    rosenbrock_histories_g.append(
        gradient_descent_UP(
            configuration[0],
            rosenbrock_f,
            rosenbrock_prime,
            np.identity(2),
            sigma=configuration[1],
            alpha_lower_bound=1,
            beta=.5,
            max_iter=100,
        )
    )
    rosenbrock_labels_g.append(f"x0: {configuration[0]}, sigma: {
        configuration[1]}")

plot_grad_norms(
    histories=rosenbrock_histories_n,
    labels=rosenbrock_labels_n,
)
plot_grad_norms(
    histories=rosenbrock_histories_g,
    labels=rosenbrock_labels_g
)

plot_2d_iterates_contours(
    rosenbrock_f,
    histories=rosenbrock_histories_n,
    labels=rosenbrock_labels_n,
    xlims=[-3, 3],
    ylims=[0,7],
```

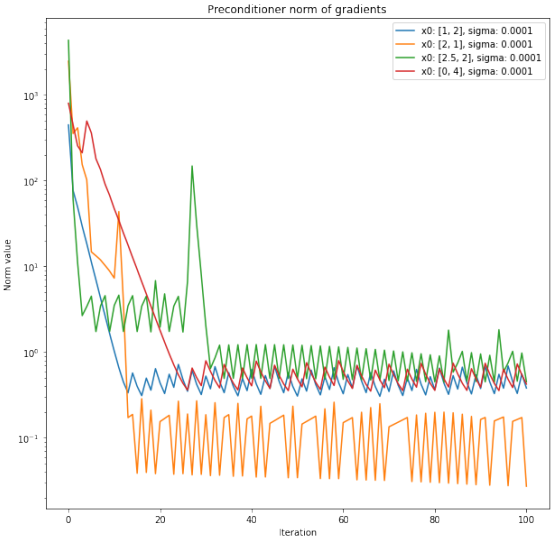
```
    title="Iterates and iso-lines of Rosenbrock function"
)
plot_2d_iterates_contours(
    rosenbrock_f,
    histories=rosenbrock_histories_g,
    labels=rosenbrock_labels_g,
    xlims=[-3, 3],
    ylims=[-2,5],
    title="Iterates and iso-lines of Rosenbrock function"
)
plt.show()
```



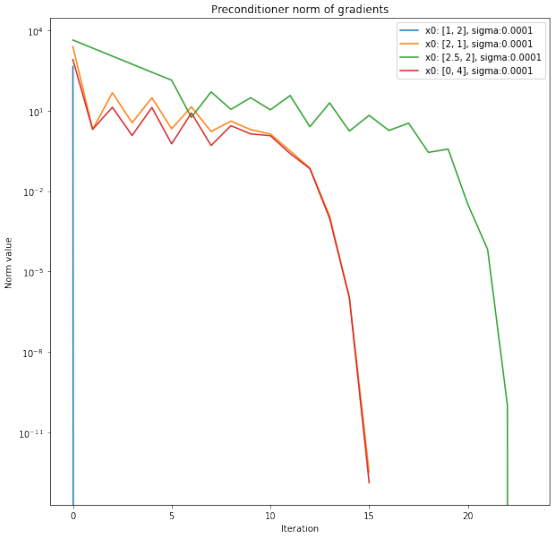
(a) Iterates for the gradient descent method



(b) Iterates for the globalized newton method



(c) gradient norms for the gradient descent method



(d) gradient norms for the globalized newton method