Nonlinear Optimization – Sheet 07

Exercise 1

(i) Choose $M = H$. Since M is assumed to be s.p.d., we conclude that $H + \mu H = (1 + \mu)H$ is s.p.d.

Condition (6.18b) gives $s = -(1 + \mu)^{-1} H^{-1} \nabla f(x)$ for some μ that we will determine using condition (6.18a). We have

$$\begin{aligned}\|s\|_H &= \|(1+\mu)^{-1} H^{-1} \nabla f(x)\|_H \\ &= (1+\mu)^{-2} f'(x) H^{-1} H H^{-1} \nabla f(x) \\ &= (1+\mu)^{-2} \|\nabla f(x)\|_H \,.\end{aligned}$$

Condition (6.18a) requires either $\mu = 0$ (and $\|\nabla f(x)\|_H = \|s\|_H \le \Delta$) or $\|s\|_H = \Delta$, which solves to

$$\mu = -1 + \sqrt{\frac{\|f'(x)\|_H}{\Delta}} \,.$$

We make our choice according to whether $\|\nabla f(x)\|_H$ is greater or smaller than Δ.

Define $\mu(\Delta) := \min(0, \max(0, -1 + \sqrt{\frac{\|f'(x)\|_H}{\Delta}}))$. We obtain

$$S_{\overline{\Delta}} = \{-(1 + \mu(\Delta))^{-1} H^{-1} \nabla f(x) \mid \Delta \in [0, \overline{\Delta}]\} \,.$$

(ii) The code for this exercise is in 1.py

```python
import numpy as np
import matplotlib.pyplot as plt
from example_functions import rosenbrock

def get_s_inner(b, H_inverse, delta):
    s = H_inverse @ b
    b_norm_sq = b.T @ s
    b_norm = np.sqrt(b_norm_sq)
    if b_norm <= delta:
        return s, True
    else:
        #correction = np.sqrt(delta) / b_norm
        correction = delta / b_norm
        return correction * s, False


def get_S_delta_from_delta_bar(b, H_inverse, delta_bar, num_deltas)
    :
    deltas = np.linspace(0,delta_bar,num_deltas)
    S_delta = []
    for delta in deltas:
        s, inner = get_s_inner(b, H_inverse, delta)
        S_delta.append((s, inner))
    return S_delta

num = 100
delta = 100
xs = [(0,-1), (0,.5)]
b_H_tupels = []
for x in xs:
    b = rosenbrock(1,100,x)[1]
```

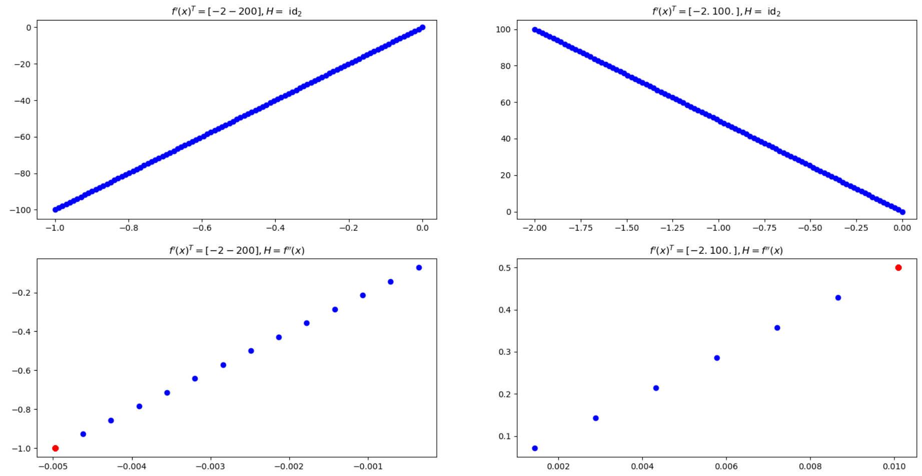


Abbildung 1: S_Δ for various parameters

```
    H = rosenbrock(1,100,x)[2]
    H_inverse = np.linalg.inv(H)
    b_H_tupels.append((b, np.eye(2)))
    b_H_tupels.append((b, H_inverse))

solutions = []
for b_H_tupel in b_H_tupels:
    S_delta = get_S_delta_from_delta_bar(*b_H_tupel, delta,
        num_deltas=num)
    solutions.append((S_delta, b_H_tupel))

fig, ax = plt.subplots(2,2)

for i, solution_tupel in enumerate(solutions):
    (S_delta, b_H_tupel) = solution_tupel
    (b, H) = b_H_tupel
    current_ax = ax[i%2,int(i>1)]
    for s in S_delta:
        (s, inner) = s
        if inner:
            c = 'red'
        else:
            c = 'blue'
        current_ax.scatter(*s,c=c)
    if (H==np.eye(2)).all():
        current_ax.title.set_text(f"$f'(x)^T={b}, H = $ id$_2$")
    else:
        current_ax.title.set_text(f"$f'(x)^T={b}, H = f''(x)$")

plt.show()
```

Exercise 2

We apply Remark 6.15 (ii) (a) and see that the operation

$$q(s^{l+1}) = q(s) - \frac{1}{2}\alpha^l \delta^l$$

requires exactly three floating point operations. In the case that the Steihaug-Toint-CG algorithm terminates with $\|s\|_M < \Delta$, we obtain the desired result.

Exercise 3

To check positive (semi) definiteness, it suffices to consider products $x^t(H + \mu M)x$ with vectors $x \in S^n = \{x \in \mathbb{R}^n \mid \|x\| = 1\}$. Consider the continuous maps

$$\begin{aligned} \varphi_M &: S^n \to \mathbb{R}, \qquad x \mapsto x^t M x\,, \\ \varphi_H &: S^n \to \mathbb{R}, \qquad x \mapsto x^t H x\,. \end{aligned}$$

S^n is a compact metric space, therefore there are scalars $\delta_M \in \mathbb{R}_{>0}$ and $\delta_H \in \mathbb{R}$ such that $\delta_M \le \varphi_M(x)$ and $\delta_H \le \varphi_H(x)$ holds for all $x \in S^n$. Then for all $x \in S^n$ we have

$$x^t(H + \mu M)x = x^t H x + \mu x^t M x \ge \delta_H + \mu \delta_M\,.$$

Therefore, if we choose μ greater (equal) than $-\delta_H/\delta_M$, positive (semi-) definiteness of $H + \mu M$ is guaranteed.

For any lesser values $H + \mu M$ might be indefinite. Consider for example the matrices

$$H = \begin{pmatrix} -2 & 0 \\ 0 & -2 \end{pmatrix}, \qquad M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Parameterizing vectors in S^2 via $(\sin\alpha, \cos\alpha)$, $\alpha \in [0, 2\pi]$, we that φ_H and φ_M are constant function of value -2 and 1, respectively. Therefore $\delta_H = -2$, $\delta_M = 1$ and $-\delta_H/\delta_M = 2$. If we choose $\mu < 2$, we have

$$(1,1)(H + \mu M)(1,1)^t = \varphi_H(1,1) + \mu\varphi_M(1,1) = -2 + \mu < 0$$

and the matrix $H + \mu M$ is indefinite.

Exercise 4

We use the same example_functions.py file as last week. Then, we implement the Steihaug-Toint-CG algorithm in steihaug_Toint_CG.py.

```
import numpy as np

def steihaug_Toint_CG(H, b, Minv, eps_rel, Delta):
    """return approximate solution of Hs = b with ||s|| <= Delta"""
    #l = 0
    s = np.array([0,0])
    zeta = -b
    p = -Minv @ zeta
    delta_0 = - np.dot(zeta, p)
    delta = delta_0
    gamma = delta
    xi = 0
    omega = 0
    while delta >= eps_rel**2 * delta_0:
        #TODO: edit
        q = H @ p
        theta = np.dot(q, p)
        M = np.linalg.inv(Minv)
```

```
        if theta > 0:
            alpha = delta/theta
            s_new = s + alpha * p
            omega_new = omega + 2*alpha*xi + alpha**2*gamma
            if np.sqrt(omega_new) > Delta:
                alpha_star =- xi/(2 * gamma) + 1/(2*gamma) * np.sqrt((
                    xi**2 - 4*gamma*(omega - Delta**2)))
                s = s + alpha_star * p #in the 0-th iteration,
                    alpha_star will be Delta/sqrt(gamma)
                #at this point, the norm of s should be Delta
                #print("here", alpha_star, s, p, s.T @ M @ s, Delta)
                pred = b @ s - .5 * s.T @ H @ s

                return s, pred #add s_m_norm
            else:
                s = s_new
                omega = omega_new
                zeta = zeta + alpha * q
                new_p =- Minv @ zeta
                new_delta =- np.dot(zeta, new_p)
                beta = new_delta/delta
                delta = new_delta

                xi = beta * (xi + alpha * gamma)
                gamma = delta + beta**2 * gamma

                p = new_p + beta*p

        else:
            alpha_star =- xi/(2 * gamma) + xi/(2*gamma) * np.sqrt((1 -
                4*gamma*(omega - Delta**2)))
            s = s + alpha_star * p
            pred = b @ s - .5 * s.T @ H @ s
            return s, pred #add s_m_norm
    pred = b @ s - .5 * s.T @ H @ s
    return s, pred #add s_m_norm
```

and the trust region algorithm using Steihaug-Toint-CG for computing the steps in steihaug_Toint_CG.py.

```
import numpy as np

def steihaug_Toint_CG(H, b, Minv, eps_rel, Delta):
    """return approximate solution of Hs = b with ||s|| <= Delta"""
    #l = 0
    s = np.array([0,0])
    zeta = -b
    p = -Minv @ zeta
    delta_0 =- np.dot(zeta, p)
    delta = delta_0
    gamma = delta
    xi = 0
    omega = 0
    while delta >= eps_rel**2 * delta_0:
        #TODO: edit
        q = H @ p
        theta = np.dot(q, p)
        M = np.linalg.inv(Minv)
```

```
        if theta > 0:
            alpha = delta/theta
            s_new = s + alpha * p
            omega_new = omega + 2*alpha*xi + alpha**2*gamma
            if np.sqrt(omega_new) > Delta:
                alpha_star = - xi/(2 * gamma) + 1/(2*gamma) * np.sqrt((
                    xi**2 - 4*gamma*(omega - Delta**2)))
                s = s + alpha_star * p #in the 0-th iteration,
                    alpha_star will be Delta/sqrt(gamma)
                #at this point, the norm of s should be Delta
                #print("here", alpha_star, s, p, s.T @ M @ s, Delta)
                pred = b @ s - .5 * s.T @ H @ s

                return s, pred #add s_m_norm
            else:
                s = s_new
                omega = omega_new
                zeta = zeta + alpha * q
                new_p = - Minv @ zeta
                new_delta = - np.dot(zeta, new_p)
                beta = new_delta/delta
                delta = new_delta

                xi = beta * (xi + alpha * gamma)
                gamma = delta + beta**2 * gamma

                p = new_p + beta*p

        else:
            alpha_star = - xi/(2 * gamma) + xi/(2*gamma) * np.sqrt((1 -
                4*gamma*(omega - Delta**2)))
            s = s + alpha_star * p
            pred = b @ s - .5 * s.T @ H @ s
            return s, pred #add s_m_norm
    pred = b @ s - .5 * s.T @ H @ s
    return s, pred #add s_m_norm
```

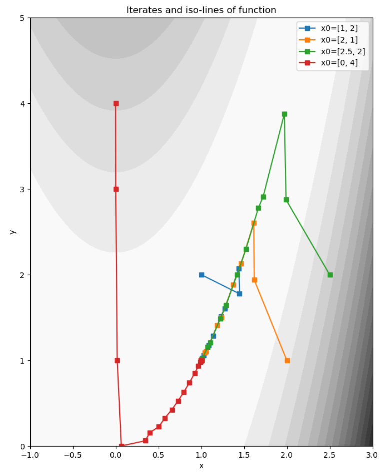
Our results and plots are generated in 4.py.

```
import numpy as np
from generic_trust_region import generic_trust_region
from example_functions import rosenbrock
import matplotlib.pyplot as plt
from visualization_functions import plot_2d_iterates_contours

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]

#x0_list = [[5, 5], [0, 0], [5, 0], [0, 5]]
x0_list = [[1, 2], [2, 1], [2.5, 2], [0, 4]]

histories = []

for x0 in x0_list:
```

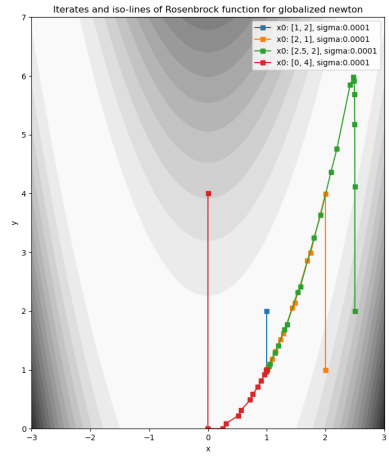
```
    _, history = generic_trust_region(
        x0,
        rosenbrock_f,
        rosenbrock_prime,
        rosenbrock_two_prime,
        np.eye(2),
        Delta_0=1,
        eta_1=1e-2,
        eta_2=0.6,
        gamma_1=0.5,
        gamma_2=2,
    )
    histories.append(history)


plot_2d_iterates_contours(
    rosenbrock_f, histories, labels=[f"x0={x0}" for x0 in x0_list],
        xlims=[-1, 3], ylims=[0, 5]
)
plt.savefig("iterates_contours_rosenbrock")
plt.show()

for history in histories:
    deltas = history["Delta"]
    plt.plot(range(len(deltas)), deltas)
plt.savefig("trust_region_size_rosenbrock")
plt.show()



for history in histories:
    areds = history["ared"]
    plt.plot(range(len(areds)), areds)
    plt.yscale('log')
plt.savefig("reduction_rosenbrock")
plt.show()
```

The method works really well for the rosenbrock function. For all 4 example start points, it takes less than iterations until the method reaches the global minimizer.

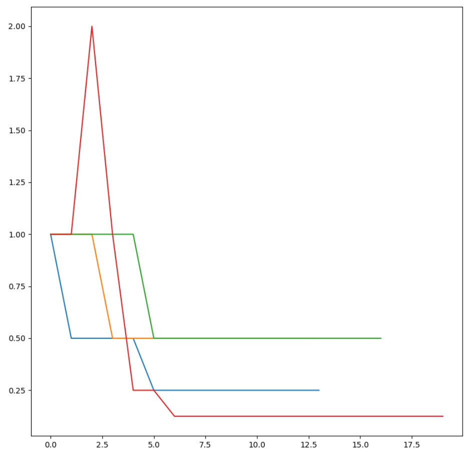
Below are some plots to visualize the results.



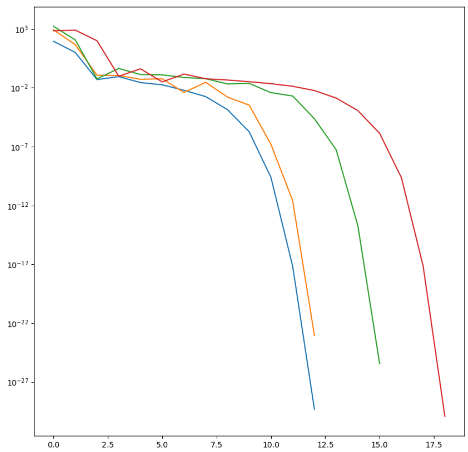
(a) Iterates for the trust region method



(b) For comparison: Iterates for the globalized newton method



(c) trust region radius for successful iterates



(d) actual reduction for successful iterates