Nonlinear Optimization – Sheet 03

Exercise 1

(i) Comparison between Theorem 4.18 and Theorem 4.7: For the steepest descent method (GD), we have a worst case convergence result for every single step. This is due to the fact that the method has no memory and each step is therefore independent of the other steps. The conjugate gradient method (CG), on the other hand, gives a result that holds only globally, for k steps. Therefore, we have Q-linear convergence for GD, while we have only R-linear convergence for the CG. In both cases, however, the objective values and, as a result, the norm of the error are monotonically decreasing. For k steps, the convergence factor for CG (considering the objective values) is

$$2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^{2k},$$

i.e. in order to reduce the initial error by a factor of ε, it takes

$$k \leq \left\lceil \frac{\sqrt{\kappa}}{4} \ln\left(\frac{2}{\varepsilon}\right) \right\rceil$$

steps. For k steps, the convergence factor for GD (considering the objective values) is

$$(\frac{\kappa-1}{\kappa+1})^{2k},$$

i.e. in order to reduce the initial error by a factor of ε, it takes

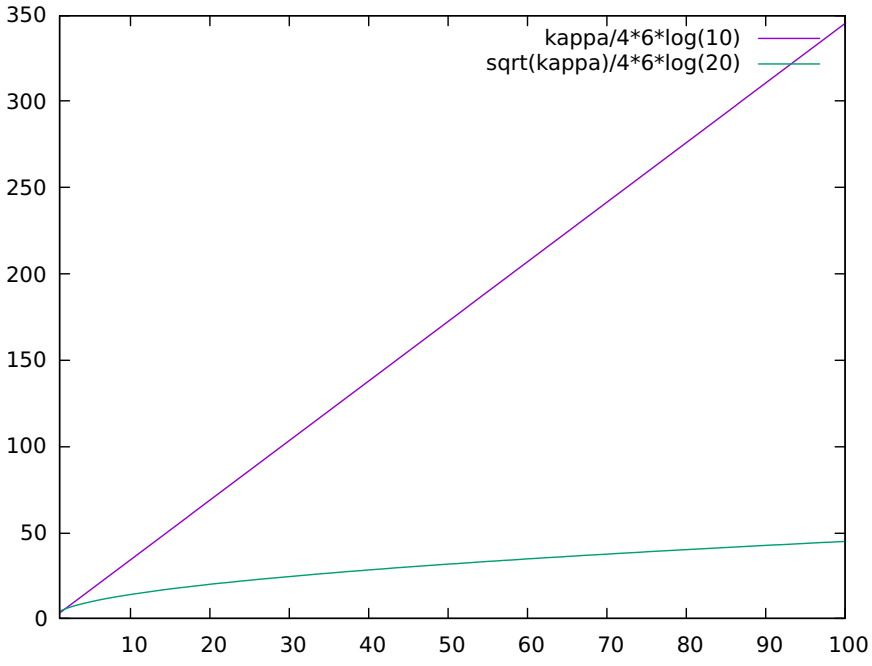
$$k \leq \left\lceil \frac{\kappa}{4} \ln\left(\frac{1}{\varepsilon}\right) \right\rceil$$

steps. CG is better than GD already for very small κ (definitely for all $\kappa \geq 2$, see (ii) for a plot.

(ii) We use gnuplot,

```
plot  [1:100]  kappa/4*6*log(10), sqrt(kappa)/4*6*log(20)
```

and get



Exercise 2

Let $A \in \mathbb{R}^{n\times n}$ be s.p.d. $b \in \mathbb{R}^n$, $c \in \mathbb{R}$. Let $x^{(k)}$ be an iterate of the CG method for $\varphi(x) = \frac{1}{2}x^tAx - b^tx + c$. Further let $x^{(k+1)}$ be computed by an additional step of the CG method and $\tilde{x}^{(k+1)}$ be computed by a steepest descent step with Cauchy stepsize starting from $x^{(k)}$. Show:

$$\|x^{(k+1)} - x^*\|_A \leq \|\tilde{x}^{(k+1)} - x^*\|_A$$

Proof. We know that

$$\frac{1}{2}\|x^{(k+1)} - x^*\|_A^2 = \varphi(x^{(k+1)}) - \varphi(x^*) \tag{1}$$

and by Lemma 4.13 and due to the A-conjugacy of the CG directions we know that $x^{(k+1)}$ minimizes φ over the affine subspace $x^{(0)} + \operatorname{span}(d^{(0)}, \dots, d^{(k)})$ where the $d^{(i)}$ are the CG directions for $0 \leq i \leq k$. Furthermore

$$\tilde{x}^{(k+1)} = \underbrace{x^{(k)} - \alpha^{(k)}M^{-1}r^{(k)} + \alpha^{(k)}\beta^{(k)}d^{(k)}}_{=x^{(k+1)}} - \alpha^{(k)}\beta^{(k)}d^{(k)} \in x^{(0)} + \operatorname{span}(d^{(0)}, \dots, d^{(k)})$$

because already $x^{(k+1)} \in x^{(0)} + \operatorname{span}(d^{(0)}, \dots, d^{(k)})$ holds. So the claim follows by (1). □

Exercise 3

Since the eigenvectors of the matrix A span the whole space, for any vector in $\mathbb{R}^n$ there is a decomposition into n distinct eigenvectors of A. Since any linear combination of two eigenvectors of the same eigenvalue is again an eigenvector of that value, condition (i) therefore implies condition (ii) and (iii).

Assume condition (iii). Then there is a decomposition

$$M^{-1}r^{(0)} = \sum_{j=1}^{k} \gamma^{(j)}v^{(j)}$$

for some $\gamma^{(j)} \in \mathbb{R}$ and some distinct generalized eigenvectors $v^{(j)} \in \mathbb{R}^n$ with eigenvalues $\lambda^{(j)}$. If v is a generalized eigenvector of (A, M) and fulfils $Av = \lambda Mv$, then $A^{-1}Mv = \lambda^{-1}v$ is again a generalized eigenvector of (A, M). Therefore the decomposition

$$x^{(0)} - x^* = A^{-1}r^{(0)} = A^{-1}MM^{-1}r^{(0)} = \sum_{j=1}^{k} \gamma^{(j)}A^{-1}Mv^{(j)}$$

is again a decomposition into k distinct generalized eigenvectors.

We see that both condition (i) and (iii) imply condition (ii), therefore it is enough to prove the assertion when assuming (ii).

Assume condition (ii). Set $e^{(j)} := x^{(j)} - x^*$ for $j \in \mathbb{N}_0$. By assumption, there is a decomposition

$$e^{(0)} = \sum_{j=1}^{k} \gamma^{(j)}v^{(j)}$$

for some $\gamma^{(j)} \in \mathbb{R}$ and some distinct generalized eigenvectors $v^{(j)} \in \mathbb{R}^n$ with eigenvalues $\lambda^{(j)}$. By the proof of Theorem 4.19, we have

$$\|e^{(k)}\|_A = \min_p \|p(M^{-1}A)e^{(0)}\|_A\,,$$

where p runs through the polynomials of degree $\leq k$ with $p(0) = 1$. By the same arguments as presented in the proof, we get

$$p(M^{-1}A)e^{(0)} = \sum_{j=1}^{k} \gamma^{(j)}p(\lambda^{(j)})v^{(j)}$$

for all such polynomials p. Consider the polynomial

$$p(T) := \prod_{j=1}^{k} \frac{\lambda^{(j)} - T}{\lambda^{(j)}} .$$

Then p is a polynomial of degree k, fulfils $p(0) = 1$ and minimizes

$$\|p(M^{-1}A)e^{(0)}\|_A = \|\sum_{j=1}^{k} \gamma^{(j)} p(\lambda^{(j)}) v^{(j)}\|_A = \|0\|_A = 0 .$$

We conclude $\|e^{(k)}\|_A = 0$, whence $e^{(k)} = 0$.

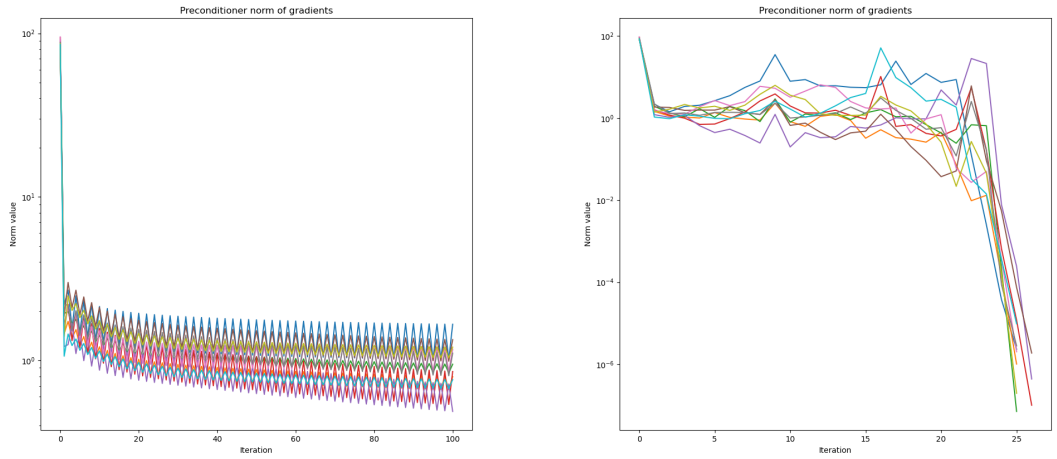
Exercise 4

The code in 4.py defines the main function and the convergence 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 rand_problem import rand_problem

max_iter = 100

def quadratic_minimization(x, A,b,c, M_inv, eps, conjugacy = True):
    f = lambda x: 0.5*x.T@(A@x) - b@x + c

    k = 0
    r = A @ x - b
    d = - M_inv @ r
    delta = - (r).transpose() @ d
    history = {
                        "iterates" : [],
                        "objective_values" : [],
                        "gradient_norms" : [],
                        "step_lengths" : []
                }
    while delta > eps**2 and k < max_iter:
        history["iterates"].append(x)
        history["objective_values"].append(f(x))
        history["gradient_norms"].append(np.sqrt(delta))
        q = A @ d
        theta = q.transpose() @ d
        alpha = delta/theta
        history["step_lengths"].append(alpha)
        x = x + alpha * d
        r = A@x - b
        #r = r + alpha * q
        d_old = d
        d = - M_inv @ r
        delta_old = delta
        delta = - r.transpose() @ d
        if conjugacy:
            beta = delta/delta_old
            d = d + beta * d_old
```

```
            k = k + 1

        history["iterates"].append(x)
        history["objective_values"].append(f(x))
        history["gradient_norms"].append(np.sqrt(delta))

        return history

#compare gradient norms

eps = 1e-5
N = 10
problems = []

for i in range(N):
    problems.append(rand_problem(20))

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

histories = []
conj_histories = []
#labels = []
for problem in problems:
    histories.append(quadratic_minimization(problem.x0, problem.A,
        problem.b, problem.c, problem.Pinv, eps, conjugacy=False))
    conj_histories.append(quadratic_minimization(problem.x0, problem.A,
        problem.b, problem.c, problem.Pinv, eps, conjugacy=True))

plot_grad_norms(histories=histories,labels=[])
plot_grad_norms(histories=conj_histories,labels=[])

plt.show()
```

This code generates the following plots Even though the gradient norm for the conjugate



(a) gradient norms for gradient descent method (b) gradient norms for conjugate gradient method

gradient method clearly doesn't converge monotonically, a plot of absolute values shows that they

do converge monotonically.

The code to generate random examples is very simple:

```
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.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 : 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)
```

For visualization, we use the code provided in the solution for exercise 1:

```
# This module implements various plotting functions to visualize the
    results
# of iterative optimization schemes and more.

import numpy as np

import matplotlib.pyplot as plt
import matplotlib.colors as mcolors

from mpl_toolkits import mplot3d

from scipy.spatial import HalfspaceIntersection
from scipy.spatial import ConvexHull

from itertools import cycle

plt.rcParams["figure.figsize"] = (10, 10)

def plot_2d_iterates_contours(f, histories, labels, title = 'Iterates␣
    and␣iso-lines␣of␣function', xlims = None, ylims = None):
        """
        Plot 2d iterates provided in histories of iterative solvers in
            the iterate
        space

        Accepts:
                                    f: the function that was minimized by
                                        the iterative solvers
```

```
                histories: list of history dictionaries created
                    from the iterative solvers
                    labels: list of label strings for the plot
                     title: title of the plot (default supplied)

    Returns:
                figure handle
    """
    if labels is None:
            labels = ['dummy']*len(histories)
            pltlabels = False
    else:
            pltlabels = True


    if xlims is None or ylims is None:
            # Determine a quadratic bounding box for the iterates
            all_iterates = np.vstack(list(np.vstack(history["
                iterates"]) for history in histories))

            min_x, max_x = np.min(all_iterates[0]), np.max(
                all_iterates[0])
            min_y, max_y = np.min(all_iterates[1]), np.max(
                all_iterates[1])

            dx = np.abs(max_x - min_x)
            dy = np.abs(max_y - min_y)

            alph = 0.5

            xlims, ylims = (min_x - alph*dx, max_x + alph*dx), (
                min_y - alph*dy, max_y + alph*dy)

    x_disc = np.linspace(xlims[0], xlims[1], num = 251)
    y_disc = np.linspace(ylims[0], ylims[1], num = 251)
    X, Y = np.meshgrid(x_disc, y_disc)

    # Get the function values of f on the grid for contour plots
    Z = np.zeros(X.shape)
    for i, x_comp in enumerate(x_disc):
            for j, y_comp in enumerate(y_disc):
                    Z[i,j] = f(np.array([x_comp,y_comp]))

    fig, ax = plt.subplots()

    for i, history in enumerate(histories):
            # Plot contour lines at iterate function values levels
                (max of 20)
            #contour_vals = sorted(set(history["objective_values
                "][20::-1]))

            # Set relevant contour lines (max of 20)
            #contour_vals = sorted(set(history["objective_values
                "][20::-1]))
            contour_vals = np.linspace(min(history["
                objective_values"]), max(history["objective_values"
```

```
                ]), 20)
            contour_vals = np.linspace(np.min(Z)-0.2*np.abs(np.min(
                Z)), np.max(Z)+0.2*np.abs(np.max(Z)), 20)

            # plot
            plt.contourf(X, Y, Z.T, cmap='gray_r', levels =
                contour_vals)

            # Plot 2d iterates
            plt.plot(np.vstack(history["iterates"])[:,0], np.vstack
                (history["iterates"])[:,1], 's-', label = labels[i
                ])

    plt.gca().set_aspect('equal','box')
    plt.title(title)
    if pltlabels:
            plt.legend()
    plt.xlabel('x')
    plt.ylabel('y')
    ax.set_xlim(xlims)
    ax.set_ylim(ylims)

    return fig

def plot_f_val_diffs(histories, reference_values, condition_numbers =
    None, methods = None, labels = None, title = 'Difference of
    functional values'):
        """
        Plot difference of objective function values provided in
            histories of
        iterative solvers and user supplied reference values (times 2
            and taken the square root) for each history.

        If the user supplied reference values are the optimal value and
             the
        the problem is quadratic, then this equals the energy norm
        of the error. Otherwise this method may be used to provide
            approximative
        information.

        Optional plot is the expected upper bound for convergence
        speed for quadratic problems provided the generalized condition
             number.

        Accepts:
                                histories: list of history dictionaries
                                    created from the iterative solvers
                        reference_values: list of reference values
                        condition_numbers: list of generalized condition
                            numbers if the problems in histories
                                        were quadratic (default None)
                                labels: list of label strings for the
                                    plot
                                 title: title of the plot (default
                                    supplied)
```

```
    Returns:
                No return values.
    """
    if labels is None:
            labels = ['dummy']*len(histories)
            pltlabels = False
    else:
            pltlabels = True

    # Evaluate and plot the differences of the objective values and
        reference values
    fig = plt.figure()

    for history, reference_value, label in zip(histories,
        reference_values, labels):
            history["f_val_diffs"] = np.sqrt(2*(np.abs(history["
                objective_values"] - reference_value)))
            plt.semilogy(history["f_val_diffs"], label = label)

    # If condition_numbers were supplied, assume that f was
        quadratic and plot
    # upper bounds on convergence speed
    bound_labels = []

    if methods is None:
            methods = ['']*len(labels)

    try:
            plt.gca().set_prop_cycle(None) # Restart the color
                cycle

            for history, reference_value, condition_number, method,
                label in \
                    zip(histories, reference_values,
                        condition_numbers, methods, labels):
                    # Evaluate the linear convergence factor from
                        condition quotient
                    contraction_CG = (np.sqrt(condition_number) -
                        1) / (np.sqrt(condition_number) + 1)
                    contraction_SD = (condition_number - 1) / (
                        condition_number + 1)

                    # Evaluate initial energy norm of error squared
                         from the function value diffs
                    initial_error = history["f_val_diffs"][0]

                    # Evaluate the predicted upper bound at each
                        iteration starting from the initial error
                        norm
                    upper_error_bound_SD = initial_error * (
                        contraction_SD**(np.arange(0,len(history["
                        objective_values"]))))
                    upper_error_bound_CG = 2*initial_error * (
                        contraction_CG**(np.arange(0,len(history["
                        objective_values"])))))
```

```
                        if method == 'CG':
                                plt.semilogy(upper_error_bound_CG, '--'
                                    , label = label + ' bound')
                        else:
                                plt.semilogy(upper_error_bound_SD, '--'
                                    , label = label + ' bound')

        except Exception as exx:
                #print("No upper bound plotable. " + exx.__str__())
                #print(exx)
                pass

        plt.title(title)
        plt.xlabel('Iteration')
        plt.ylabel(r'$||x-x^*||_A$')
        if pltlabels:
                plt.legend()

        return fig

def plot_step_sizes(histories,labels):
        """Plot the step lengths of a list of histories."""
        fig = plt.figure()
        for history in histories:
                plt.plot(history["step_lengths"])

        plt.title('Step lengths')
        plt.xlabel('Iteration')
        plt.ylabel('Step length')
        if labels:
                plt.legend(labels)

        return fig

def plot_grad_norms(histories,labels):
        """Plot the gradient norms of a list of histories."""
        fig = plt.figure()
        for history in histories:
                plt.semilogy(history["gradient_norms"])

        plt.title('Preconditioner norm of gradients')
        plt.xlabel('Iteration')
        plt.ylabel('Norm value')
        if labels:
                plt.legend(labels)

        return fig
```