Exercise 1

We have
$$\phi''(x) = A.$$
If A is positive semidefinite, we can apply Theorem 2.9 (a) and conclude that ϕ is convex.

Now we have to show the equivalences.

(a) $\implies$ (b) If there is a global minimizer, ϕ has to be bounded below by definition.

(b) $\implies$ (c) Assume that there is an eigenvector v of A with eigenvalue 0 s.t. $b^\top v \neq 0$. Let $-C$ be a lower bound for ϕ. Then choose
$$\lambda := \frac{C + c + 1}{b^\top v}.$$
We obtain a contradiction,
$$\phi(\lambda \cdot v) = (\lambda v)\top A(\lambda v) - b^\top v + c = 0 - \frac{C + c + 1}{b^\top v} \cdot (b^\top v) + c = -C - 1.$$
Therefore, b is in the orthogonal complement of the eigenspace to the eigenvalue 0. In particular, we find α_i s.t.
$$b = \sum_i \alpha_i v_i,$$
where v_i is an eigenvector of A with eigenvalue $\lambda_i \neq 0$. With
$$x = \sum_i \frac{\alpha_i}{\lambda_i} v_i \implies Ax = \sum_i \alpha_i v_i = b$$
we have found a solution for $Ax = b$.

(c) $\implies$ (a) Let x be a solution for $Ax = b$. Then, $\nabla\phi(x) = Ax - b = 0$ and as ϕ is convex, the desired implication follows from Sheet 1, Ex. 4.

Let v be an eigenvector of A with eigenvector $\lambda < 0$. Then, using the Cauchy-Schwarz-inequality and the triangle inequality, we obtain
$$x^\top Ax - b^\top x + c \leq \lambda\|x\| + \|b\|\|x\| + c.$$
This is a quadratic polynomial with negative leading coefficient, i.e. it is not bounded below.

Exercise 2

(i) "$\supseteq$" Let $d \in$ RHS. By definition,
$$(-\nabla f(x))^\top \cdot d > 0$$
We apply BFGS for $g = (-\nabla f(x))$ and $d = d$ to the identity matrix. Thus, $M^{-1} \cdot (-\nabla f(x)) = d$ and $d \in$ LHS.

"$\subseteq$" Take $-M^{-1} \cdot \nabla f(x) \in$ LHS for any s.p.d. matrix M. Then,
$$f'(x) \cdot (-M^{-1}) \cdot \nabla f(x) = -\|\nabla f(x)\|^2_{M^{-1}} < 0.$$
Therefore, $-M^{-1} \cdot \nabla f(x) \in$ RHS.

(ii)
```python
import numpy as np
import matplotlib.pyplot as plt

def compute_gradient(derivative, preconditioner):
    m_gradient = - np.linalg.inv(preconditioner) @ derivative
    return m_gradient

#example in three dimensions:
def example():
    derivative = np.array([1,2,3])
    preconditioner = np.array([[1,2,3],[4,5,6],[7,8,9]])
    print(compute_gradient(derivative, preconditioner))

#visualization for two dimensions

def visualize(derivative):

    fix, ax = plt.subplots(1,2)

    derivative = np.array(derivative)
    r = np.arange(-1, 1, 0.01)
    theta = np.pi * r
    x = np.sin(theta)
    y = np.cos(theta)
    #d = np.array([x,y])
    #derivative: 1 x 2, d: 2 x 200
    good_x = []
    good_y = []
    for x_i, y_i in zip(x,y):
        if (np.matmul(np.array([x_i,y_i]), derivative.transpose())) <
            0:
            good_x.append(x_i)
            good_y.append(y_i)

    ax[0].scatter(good_x,good_y) #directions where gradient *
        direction < 0

    #all symmetric positive definite matrices of size 2x2 are
        symmetric, trace > 0 and det > 0
    #(and these three conditions are sufficient)
    #[[a,b],[b,c]]
    #we choose b = 1 or b = -1 because positive scalar multiples are
        boring. Then det(A) > 0 => ac > 1 and tr(A) > 0 => a + c > 0
    a = np.logspace(-4,4,100)
    c = np.logspace(-4,4,100)

    spd_matrices_p = np.array([[[a_i,1.], [1., c_i]] for a_i in a for
        c_i in c if a_i * c_i > 1], dtype=float)
    spd_matrices_n = np.array([[[a_i,-1.], [-1., c_i]] for a_i in a
        for c_i in c if a_i * c_i > 1], dtype=float)
    spd_matrices = np.concatenate((spd_matrices_p, spd_matrices_n))
```

```
    #in this step we could use our compute_gradient function, but
        this is easier:
    steepest_directions = spd_matrices @ derivative

    steepest_x = []
    steepest_y = []
    for pair in steepest_directions:
        x_i = pair[0]
        y_i = pair[1]
        norm = np.sqrt(x_i**2 + y_i**2)
        steepest_x.append(-x_i/norm)
        steepest_y.append(-y_i/norm)
    ax[1].scatter(steepest_x,steepest_y)
    #print(steepest_directions) #list of vectors with [x,y]
        coordinates after multiplication with M

    #plt.plot(steepest_directions)
    plt.savefig("2_ii_2d_plot.png")
    plt.show()

visualize([2.,.5])
```

Exercise 3

As shown in the lecture, we have the generalized spectral decomposition

$$A = MV\Lambda V^\top M,$$

s.t. $V^\top MV = \mathrm{Id}$. This implies $VV^\top = M^{-1}$. and $M^{-1}V^{-\top} = V$. Therefore we can compute

$$A^{-1} = M^{-1}V^{-\top}\Lambda^{-1}V^{-1}M^{-1} = V\Lambda^{-1}V^\top$$

Thereby we get

$$\begin{aligned}\|x^{(k)} - x^*\|_A^2 &= \|A^{-1}r^{(k)}\|_A^2 \\ &= r^{(k),\top}A^{-1}AA^{-1}r^{(k)} \\ &= r^{(k)}V\Lambda^{-1}V^\top r^{(k)}\end{aligned}$$

Using that Λ is the diagonal matric of eigenvectors, we obtain

$$\frac{1}{\beta}r^{(k),\top}V\Lambda^{-1}V^\top r^{(k)} \leq r^{(k),\top}V\Lambda^{-1}V^\top r^{(k)} \leq \frac{1}{\alpha}r^{(k),\top}VV^\top r^{(k)}$$

Analogously, we compute

$$A^{-1}MA^{-1} = V\Lambda V^\top MV\Lambda V^\top = V\Lambda^{-2}V^\top$$

and get

$$\begin{aligned}\|x^{(k)} - x^*\|_M^2 &= \|A^{-1}r^{(k)}\|_M^2 \\ &= r^{(k),\top}A^{-1}MA^{-1}r^{(k)} \\ &= r^{(k),\top}V\Lambda^{-2}V^\top r^{(k)}\end{aligned}$$

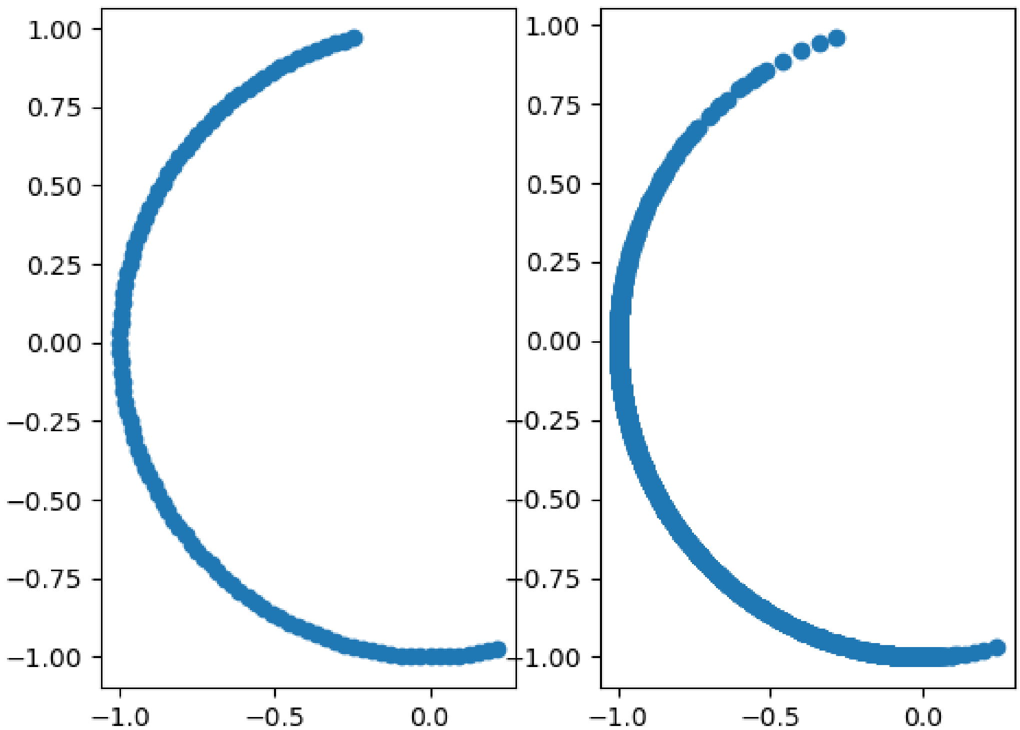

Figure 1: Here we show first the right set and then our approximation for the left set.

Using that Λ is the diagonal matric of eigenvectors, we obtain

$$\frac{1}{\beta} r^{(k),\top} V \Lambda^{-2} V^\top r^{(k)} \leq r^{(k),\top} V \Lambda^{-2} V^\top r^{(k)} \leq \frac{1}{\alpha^2} r^{(k),\top} V V^\top r^{(k)}$$

Finally, we have

$$\|r^{(k)}\|_{M^{-1}}^2 = r^{(k),\top} M^{-1} r^{(k)} = r^{(k),\top} V V^\top r^{(k)}$$

After taking square roots, we can directly see that both cases of bracket (2) hold. In order to prove bracket (1), we need to replace (k) by (0) so that we get the following inequality

$$\frac{1}{\sqrt{\beta}} \sqrt{r^{(0),\top} V \Lambda^{-1} V^\top r^{(0)}} \leq \sqrt{r^{(0),\top} V \Lambda^{-1} V^\top r^{(0)}}. \qquad (*)$$

Multiplying with $\sqrt{\beta}$, we can combine it with one of the above inequalities to obtain the desired result: First, we find an upper bound for $\|x^(k) - x^*\|_A$ by $\frac{1}{\sqrt{\alpha}} \|r^{(k)}\|_{M^{-1}}$. Then, we use the given inequality and finally apply $(*)$. The second case in bracket (1) works analogously. Brackets (3) and (4) can be shown by the exact same procedure as bracket (1).

Exercise 4

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

def gradient_descent_plot(x, b, A, M, eps, ax, steps = "cauchy", stepsize
    = 0, stepsizefunction = None):
    x_steps = [x]

    k = 0
    M_inv = np.linalg.inv(M)
    r = A @ x - b
    d = - M_inv @ r
    delta = - (r).transpose() @ d
    other_criteria = True #e.g. stop when stepsizelist is exhausted
    while delta > eps**2 and other_criteria:
        q = A @ d
        if steps == "cauchy":
            theta = q.transpose() @ d
            alpha = delta/theta
        elif steps == "constant":
            if stepsize == 0:
                raise("stepsize can't be 0 if stepsize is constant")
            else:
                alpha = stepsize
        elif steps == "custom":
            if not stepsizefunction:
                raise("stepsizefunction can't be None if steps are custom
                    ")
            else:
                try:
                    alpha = stepsizefunction(x,r,k,M,M_inv)
                except:
                    raise("stepsizefunction is ill-defined")

        x = x + alpha * d
        r = r + alpha * q
        d = - M_inv @ r
        delta = - r.transpose() @ d
        k = k + 1

        #plotting
        x_steps.append(x)

    point_xs = []
    point_ys = []
    for point in x_steps:
        point_xs.append(point[0])
        point_ys.append(point[1])

    ax.plot(point_xs, point_ys)
    return x

def contour_plot(A,b,ax):
```

```
    x_vals = np.arange(-10,10,1)
    y_vals = np.arange(-10,10,1)
    X, Y = np.meshgrid(x_vals, y_vals)
    vals = np.array([X,Y])
    Ax = np.tensordot(A, vals, axes=([1,0]))
    xAx_pre = Ax * vals
    xAx = np.sum(xAx_pre,axis = 0)

    bx = np.tensordot(b, vals, axes=([0,0]))
    #bx = np.sum(bx_pre,axis = 0)
    #print(bx_pre.shape)

    Z = xAx - bx

    ax.contour(X, Y, Z)

#example
x = np.array([9.,9.])
b = np.array([0.,0.])
A = np.array([[2.,1.],[1.,3.]])
M1 = np.array([[1.,0.],[0.,1.]])
M2 = np.array([[2.,.5],[1.,1.5]])
Ms = [M1, M2]
eps = 1e-3

def foo(x,r,k,M,M_inv):
    return 1/(k+3)

fig, ax = plt.subplots(1,2) #possibly add more subplots
for i in range(2):
    contour_plot(A,b,ax[i])
    gradient_descent_plot((9.,9.),b,A,Ms[i],eps,ax[i])
    gradient_descent_plot((-9.,9.),b,A,Ms[i],eps,ax[i])
    gradient_descent_plot((9.,-9.),b,A,Ms[i],eps,ax[i])
    gradient_descent_plot((-9.,-9.),b,A,Ms[i],eps,ax[i])
    gradient_descent_plot((-9.,-9.),b,A,Ms[i],eps,ax[i],steps="constant",
        stepsize=.01)
    gradient_descent_plot((-9.,-9.),b,A,Ms[i],eps,ax[i],steps="custom",
        stepsizefunction=foo)
plt.savefig("example.png")
```

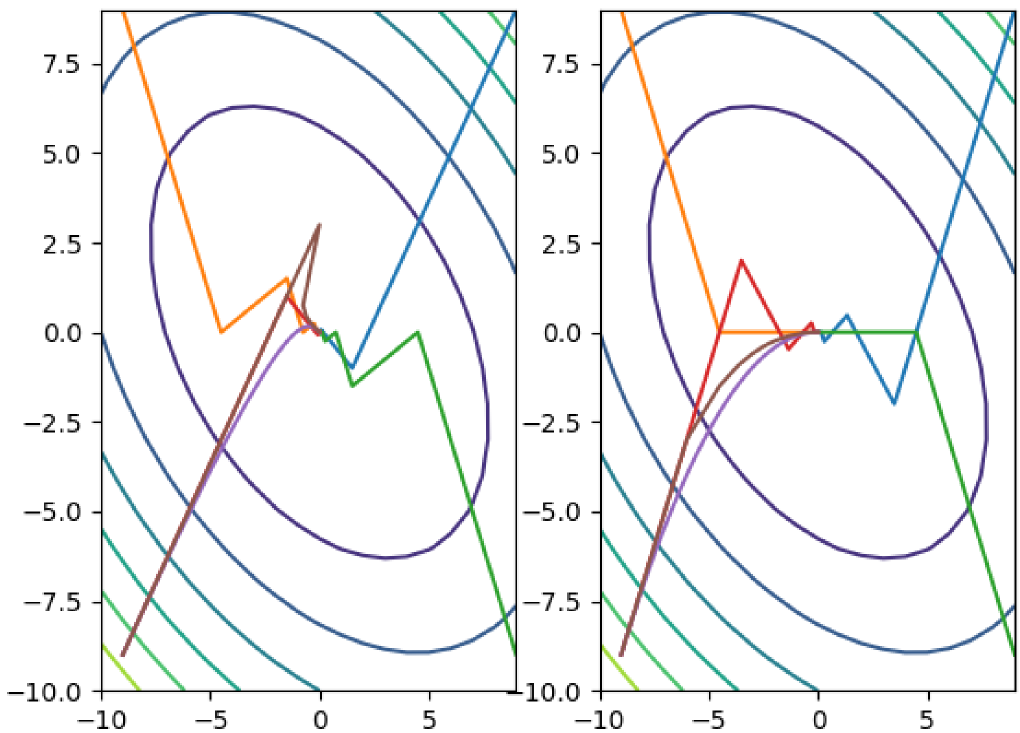

Figure 2: We clearly see how different preconditioners produce a different convergence behavior