Chapter 1

Root finding of a convex 1D function

Let $f : \mathbb{R} \to \mathbb{R}$ be differentiable, strictly monotonously increasing and convex with some x^* such that $f(x^*) = 0$

- Since the function is strictly monotonous it has at most one root, so x^* is the unique root
- If $x_k \geq x^*$ then also $x_{k+1} \geq x^*$ due to convexity, since tangents lie below the graph, and since the gradient is positive due to strict monotonocity
- Suppose $x_0 < x^*$, then due to convexity the tangent at x_0 is below the graph and $x_1 \geq x^*$
- So after at most one iterate $x^* \leq x_k$ is a lower bound of the sequence of all x_ks
- Further, $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$ where $f(x_k) \geq 0$ since x_k is greater than the root of a strictly monotonous function and $f'(x_k) > 0$ due to strict monotonicity
- Therefore $\frac{f(x_k)}{f'(x_k)} > 0$ and $x_{k+1} < x_k$, so the sequence of x_k strictly monotonously decreases
- Since the sequence of x_k strictly monotonously decreases and is lower-bound by x^*, it converges to x^* $\square$

Regularization

$$\text{Z\!\!Z}: \lim_{\lambda\to\infty}\left(x_k - (B_k - \mathbb{I})^{-1}\nabla f(x_k)\right) = x_k - \frac{1}{\lambda}\nabla f(x_k) + \mathcal{O}\left(\frac{1}{\lambda^2}\right)$$

$$\begin{aligned}
x_{k+1} &= x_k - (B_k + \lambda\mathbb{I})^{-1}\nabla f(x_k) \\
&= x_k - \left(\lambda\frac{1}{\lambda}B_k + \lambda\mathbb{I}\right)^{-1}\nabla f(x_k) \\
&= x_k - \frac{1}{\lambda}\left(\frac{1}{\lambda}B_k + \mathbb{I}\right)^{-1}\nabla f(x_k) \\
&= x_k - \frac{1}{\lambda}\left(\mathbb{I} - \underbrace{\left(-\frac{1}{\lambda}B_k\right)}_{=:B'}\right)^{-1}\nabla f(x_k)
\end{aligned}$$

Since $\rho(B') \propto \frac{1}{\lambda}$ there exists some λ_0 such that $\forall\lambda > \lambda_0 : \rho(B') < 1$ and the geometric series expansion is applicable to the limit $\lambda \to \infty$

$$\begin{aligned}
x_{k+1} &= x_k - \frac{1}{\lambda}\left(\mathbb{I} + B' + B'^2 + \dots\right)\nabla f(x_k) \\
&= x_k - \frac{1}{\lambda}\nabla f(x_k) \underbrace{+\frac{1}{\lambda^2}B_k^2\nabla f(x_k) - \frac{1}{\lambda^3}B_k^3\nabla f(x_k) + \dots}_{\in\mathcal{O}\left(\frac{1}{\lambda^2}\right)} \\
&= x_k - \frac{1}{\lambda}\nabla f(x_k) + O\left(\frac{1}{\lambda^2}\right) \quad \square
\end{aligned}$$

Unconstrained minimization

1.

$$
\begin{aligned}
f(x,y) &= \frac{1}{2}(x-1)^2 + \frac{1}{2}y^2 + \rho\frac{1}{2}(y-\cos(x))^2 \\
\frac{\partial}{\partial x}f(x,y) &= x - 1 + \rho\sin(x)(y-\cos(x)) \\
&= x + \rho y\sin(x) - \frac{\rho}{2}\sin(2x) - 1 \\
\frac{\partial}{\partial y}f(x,y) &= y(1+\rho) - \rho\cos(x) \\
\implies \nabla f &= \begin{bmatrix} x + \rho y\sin(x) - \frac{\rho}{2}\sin(2x) - 1 \\ (1+\rho)y - \rho\cos(x) \end{bmatrix} \\
\frac{\partial^2}{\partial x^2}f &= 1 + \rho y\cos(x) - \rho\cos(2x) \\
\frac{\partial^2}{\partial y^2}f &= 1 + \rho \\
\frac{\partial}{\partial y}\left(\frac{\partial}{\partial x}f\right) &= \rho\sin(x) = \frac{\partial}{\partial x}\left(\frac{\partial}{\partial y}f\right) \\
\implies \nabla^2 f &= \begin{bmatrix} 1 + \rho y\cos(x) - \rho\cos(2x) & \rho\sin(x) \\ \rho\sin(x) & 1+\rho \end{bmatrix}
\end{aligned}
$$

2.

$$
\begin{aligned}
f(x,y) &= \frac{1}{2}(x-1)^2 + \frac{1}{2}y^2 + \rho\frac{1}{2}(y-\cos(x))^2 \\
&= \frac{1}{2}\left((x-1)^2 + y^2 + \rho(y-\cos(x))^2\right) \\
&= \frac{1}{2}\left|\left|\begin{bmatrix} (x-1) \\ y \\ \sqrt{\rho}\,(y-\cos(x)) \end{bmatrix}\right|\right|^2 \\
&=: \frac{1}{2}\,||r(x,y)||^2
\end{aligned}
$$

3.

$$
\begin{aligned}
J_r &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \sqrt{\rho}\sin(x) & \sqrt{\rho} \end{bmatrix} \\
B_k &= J_r^T J_r \\
&= \begin{bmatrix} 1 & 0 & \sqrt{\rho}\sin(x) \\ 0 & 1 & \sqrt{\rho} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \sqrt{\rho}\sin(x) & \sqrt{\rho} \end{bmatrix} \\
&= \begin{bmatrix} 1 + \rho\sin^2(x) & \rho\sin(x) \\ \rho\sin(x) & 1+\rho \end{bmatrix}
\end{aligned}
$$

4. The approximate Hessian is therefore equal to the exact Hessian when:

$$
\begin{aligned}
1 + \rho y \cos(x) - \rho \cos(2x) &= 1 + \rho \sin^2(x) && \quad | -1 \quad | \cdot \frac{1}{\rho} \\
y \cos(x) - \cos(2x) &= \sin^2(x) && \quad | \cos(2x) = 1 - 2\sin^2(x) \\
y \cos(x) + 2\sin^2(x) - 1 &= \sin^2(x) && \quad | - \sin^2(x) \\
y \cos(x) + \sin^2(x) &= 1
\end{aligned}
$$

5. see Figure 3.1 and Figure 3.2

6. All methods appear to converge to the same minimum. The Gauss-Newton converges fastest in this instance, followed by the exact Newton's method. Gradient descent makes rapid progress comparable to Gauss-newton at the beginning, but tapers off as it slowly progresses along a relatively flat ridge in the function landscape.

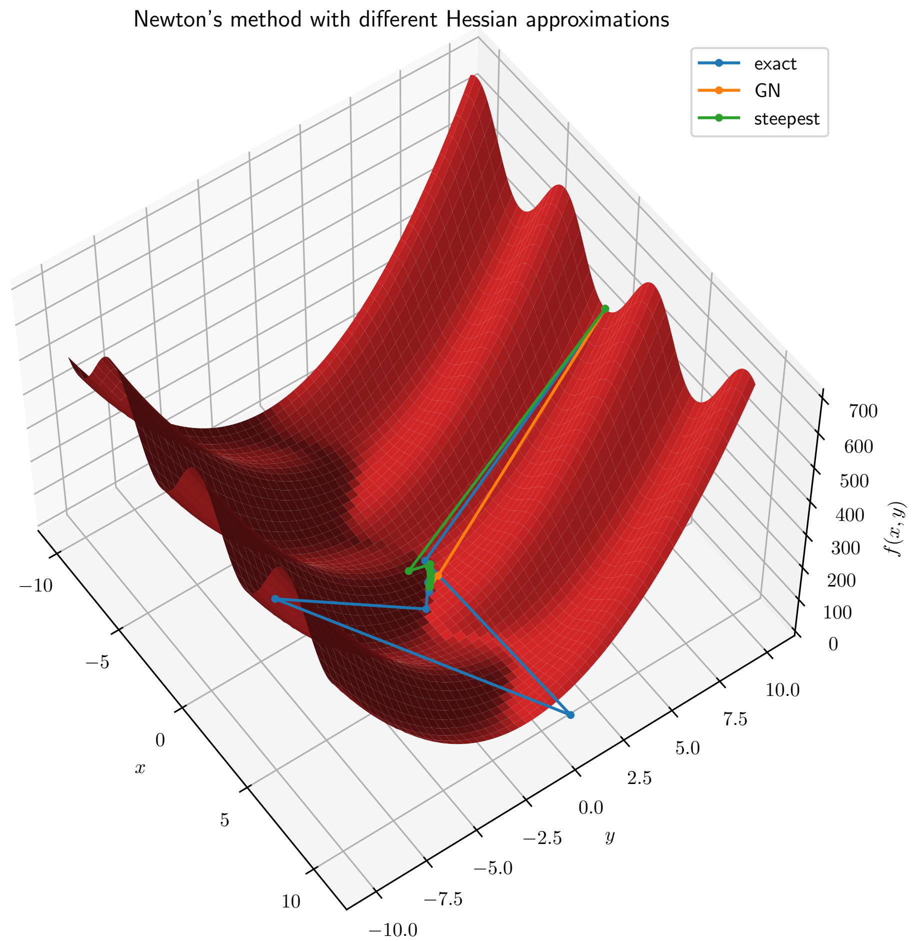

Figure 3.1: 3D surface visualization of the descent using different Hessian approximations

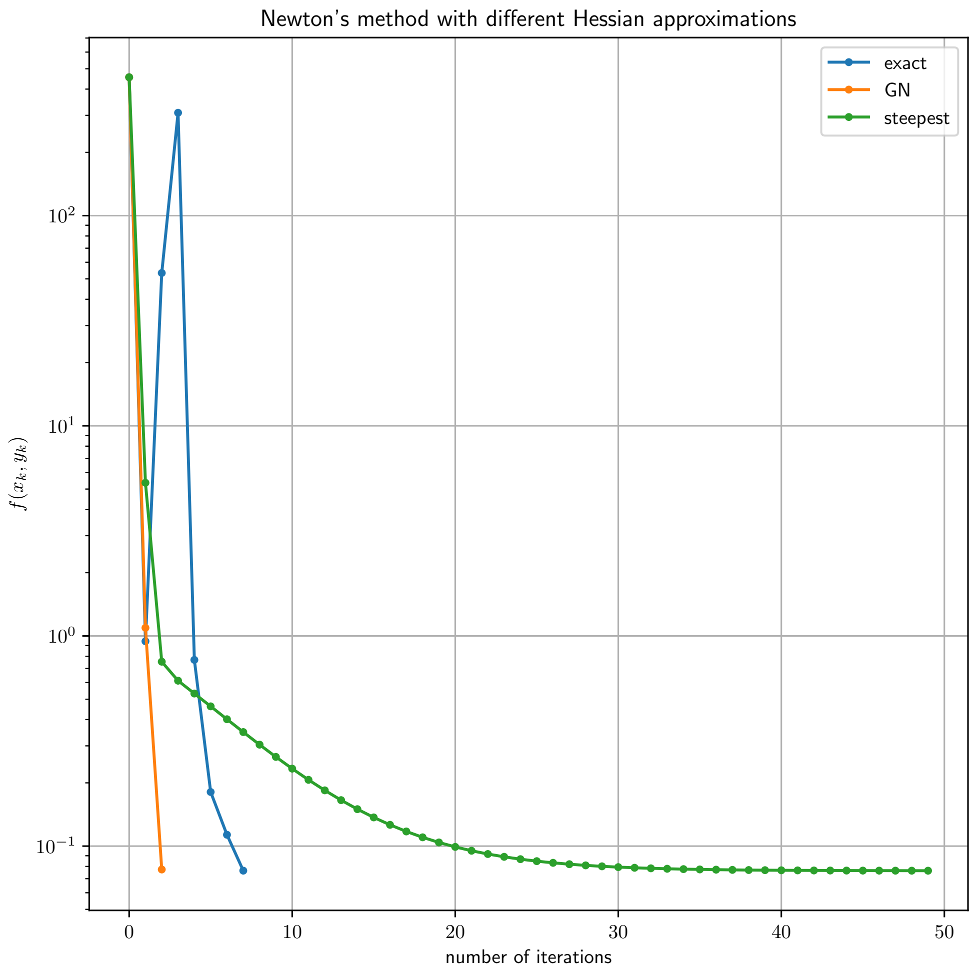

Figure 3.2: Function values per iteration of the different Hessian approximations

Hanging chain, revisited

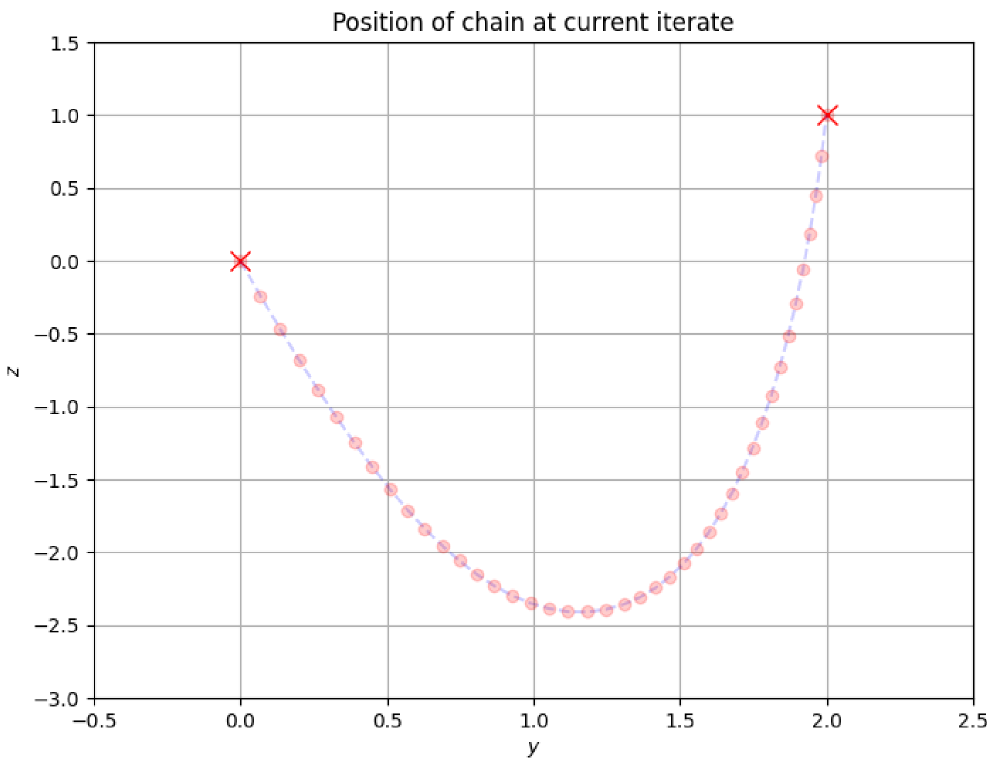

Figure 4.1: The algorithm converges to the expected solution

Source Code

5.1 Unconstrained Newton-Type Methods

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

plt.rcParams.update({"text.usetex": True})

rho = 10 # set this parameter to 5. You can also play with this parameter.

## Define the objective function
def f(x, y) -> float:
    return 0.5*( (x-1)**2 + y**2 + rho*(y-np.cos(x))**2 )

## Define the gradient:
def gradient(x, y):
   return np.array([
       x + rho*y*np.sin(x) - rho/2*np.sin(2*x) - 1,
       (1+rho)*y - rho*np.cos(x)
    ])

## Define the Hessian Approximations
def Hessian(x, y, approximation):
    """
        Approximation is a string, equal to one of the following:
         - "exact" for exact Hessian approximation
         - "GN" for Gauss-Newton hessian approximation
         - "steeepest" for alpha I with alpha= 10
    """
    if approximation == "exact":
        return np.matrix(
            [[1 + rho*y*np.cos(x) - rho*np.cos(2*x), rho*np.sin(x)],
             [rho*np.sin(x), 1+rho]]
        )
    elif approximation == "GN":
        return np.matrix(
            [[1 + rho*np.sin(x)**2, rho*np.sin(x)],
             [rho*np.sin(x), 1+rho]]
        )
    elif approximation == "steepest":
        alpha = 10
        return np.matrix(
            [[alpha, 0],
             [0, alpha]]
        )
    else:
        raise ValueError("Unknown approximation type. Choose from 'exact', 'GN', or 'steepest'.")
```

```
def Newton_step(x, y, hessian_approximation):
    """
        Perform a Newton step using the specified approximation for the Hessian.
    """
    grad = gradient(x, y)
    H = Hessian(x, y, hessian_approximation)
    step = (-np.linalg.inv(H) @ grad)
    return x+step[0,0], y+step[0,1]

def stopping_condition(x, y) -> bool:
    """
        Check the stopping condition for the Newton method.
    """
    grad = gradient(x,y)
    return np.dot(grad,grad) <= 1e-6



# Run the algorithm (nothing to do here)
hessian_approximations = ["exact", "GN", "steepest"]
all_iterates = {}
N_max = 50
for hessian_approximation in hessian_approximations:
    iterates = []
    x, y = (0,10)
    for k in range(N_max):
        iterates.append((x, y))
        x, y = Newton_step(x, y, hessian_approximation)
        if stopping_condition(x, y):
            break
    all_iterates[hessian_approximation] = iterates



# Plot the solutions
plot = "3D" # either "3D" or "value"
if plot=="3D":
    N_grid = 500
    X = np.linspace(-10, 10, N_grid)
    Y = np.linspace(-10, 10, N_grid)
    X, Y = np.meshgrid(X, Y)
    Z = f(X, Y)
    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot(projection='3d', computed_zorder=False)
    ax.grid()
    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'$y$')
    ax.set_zlabel(r'$f(x, y)$') # type: ignore


    for hessian_approximation, iterates in all_iterates.items():
        iterates = np.array(iterates)
        x_iterates, y_iterates = iterates[:, 0], iterates[:, 1]
        ax.plot(x_iterates, y_iterates, f(x_iterates, y_iterates), "-o", markersize=3, label=hessian_
    ax.plot_surface(X, Y, Z) # type: ignore
    ax.legend()
    ax.set_title("Newton's method with different Hessian approximations")
    plt.show()
```

```
elif plot=="value":
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.set_xlabel(r'number of iterations')
    ax.set_ylabel(r'$f(x_k, y_k)$')
    ax.grid()
    # ax.set_xscale('log')
    ax.set_yscale('log')
    for hessian_approximation, iterates in all_iterates.items():
        f_iterates = [f(x,y) for x,y in iterates]
        ax.plot(range(len(f_iterates)), f_iterates, "-o", markersize=3, label=hessian_approximation)
    ax.legend()
    ax.set_title("Newton's method with different Hessian approximations")
    plt.show()
```

5.2 Hanging Chain

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

from hanging_chain_functions import f, f_grad, N
from hanging_chain_animation import make_animation

"""
    You need to implement the two following functions,
    one for the BFGS update, and the other one for the globalization.

"""
with_BFGS = True # Choose to perform the BFGS update or not

def my_globalization(x, dx, grad):
    t = 1.
    gamma = 0.1
    beta = 0.9
    for i in range(1000):
        x_candidate = x + t * dx
        # check the Armijo condition of sufficient descent
        if f(x_candidate) <= f(x) + gamma*t*np.dot(grad, dx): # TODO
            return t
        else:
            t *= beta
    raise ValueError("Globalization did not finish")

def my_update(x, grad, old_x, old_grad, old_Bk):
    Bk = old_Bk.copy()
    if with_BFGS and old_grad is not None:
        # BFGS update
        s = x - old_x
        y = grad - old_grad
        Bk -= ((Bk @ s) @ (s.T @ Bk))/np.dot(s, (Bk @ s)) + np.dot(y,y)/np.dot(s,y)
    dx = -np.linalg.inv(Bk) @ grad
    t = my_globalization(x, dx, grad)
    new_x = x + t * dx
    return new_x, Bk
```

```
"""
    In this part of the file, we run the optimization algorithm using the two functions above.
"""
# initialize the optimization variables
y = np.linspace(-0.1, -0.5, N)
z = np.zeros(N)
x_opt = np.concatenate((y, z))
old_x, old_grad = None, None
Bk = 100*np.eye(2*N, 2*N)

y_list, z_list = [], []
N_max = 1_000
for i in range(N_max):
    # Compute the gradient
    grad = f_grad(x_opt)

    if np.dot(grad, grad) <= 1e-3: # type: ignore
        print("Converged !")
        break
    # Perform update
    new_x_opt, Bk = my_update(x_opt, grad, old_x, old_grad, Bk)

    # Update variables
    old_x = x_opt
    x_opt = new_x_opt
    old_grad = grad

    # Save variables
    y_list.append(x_opt[:N])
    z_list.append(x_opt[N:])

# Make animation
anim = make_animation(y_list, z_list)
plt.show()
```