

# hw5\_2023

December 6, 2023

## 1 Homework set 5

Before you turn this problem in, make sure everything runs as expected (in the menubar, select Kernel → Restart Kernel and Run All Cells...).

Please **submit this Jupyter notebook through Canvas** no later than **Mon Dec. 4, 9:00**. **Submit the notebook file with your answers (as .ipynb file) and a pdf printout. The pdf version can be used by the teachers to provide feedback. A pdf version can be made using the save and export option in the Jupyter Lab file menu.**

Homework is in **groups of two**, and you are expected to hand in original work. Work that is copied from another group will not be accepted.

## 2 Exercise 0

Write down the names + student ID of the people in your group.

- Amir Sahrani (12661651)
- Jonas Schäfer (14039982)

## 3 Exercise 1 (6 points)

A bacterial population  $P$  grows according to the geometric progression

$$P_t = rP_{t-1}$$

Where  $r$  is the growth rate. The following population counts  $P_1, \dots, P_8$  (in billions) are observed:

```
[ ]: import numpy as np
from typing import Callable
import matplotlib.pyplot as plt
import scipy as sp

data = np.array([0.19, 0.36, 0.69, 1.3, 2.5, 4.7, 8.5, 14])
```

## 4 (a)

Read chapter 6.6 on Nonlinear Least squares. Use the Gauss-Newton Method to fit the model function  $f(t, x_1, x_2) = x_1 \cdot x_2^t$  to the data. Find estimates for the initial population  $P_0 = x_1$  and the growth rate  $r = x_2$ . Implement the Gauss-Newton method yourself. You may use linear algebra functions from `scipy` and `numpy`. Plot the datapoints and the curve fitted to the data in a semilogarithmic plot.

It is best if you define your function for Gauss-Newton separately from the definitions associated with the bacterial model.

```
[ ]: def jacobian(x1: float, x2: float, t: np.ndarray, sigma: np.ndarray) -> np.
      ndarray:
          dx1 = np.power(x2, t) / sigma
          dx2 = t * x1 * np.power(x2, t-1) / sigma
          return np.array([dx1, dx2]).T

def gauss_newton(x1: float, x2: float, t: np.ndarray,
                 f: Callable, data: np.ndarray,
                 jac: Callable, weight = None, max_iter: int = 100, tol: float =
      1e-8) -> np.ndarray:
    if weight is not None:
        sigma = data * weight
    else:
        sigma = 1.0
    for n in range(max_iter):
        J = jac(x1, x2, t, sigma)
        res = data - f(x1, x2, t)
        R = 1/2 * res / sigma
        hessian = J.T @ J
        gradient = J.T @ R
        delta = np.linalg.solve(hessian, gradient)
        x1 += delta[0]
        x2 += delta[1]
        if np.linalg.norm(delta) < tol:
            break
    return np.array([x1, x2])

def model(x1: float, x2: float, t: np.ndarray) -> np.ndarray:
    return x1*x2**t
```

```
[ ]: x1_guess = 1.0
      x2_guess = 3.0
      t = np.arange(len(data))

      x1, x2 = gauss_newton(x1_guess, x2_guess, t, model, data, jacobian)
      pred = model(x1, x2, t)
```

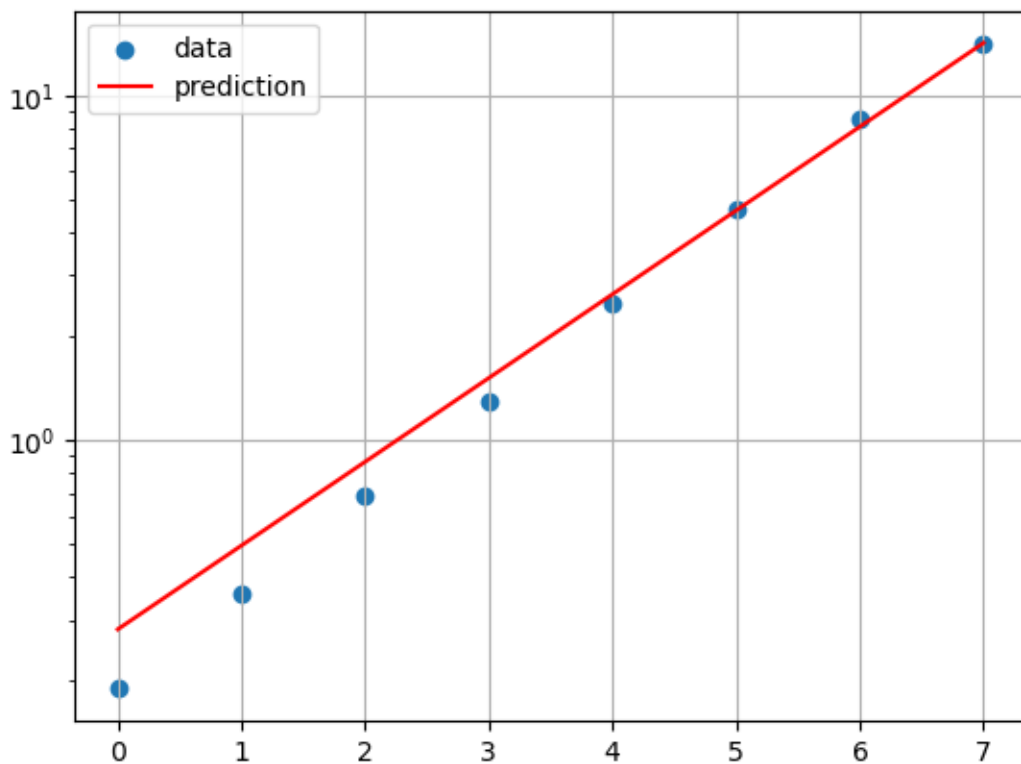
```

res = np.linalg.norm(data - pred)
list_res = []
for i in range(len(data)):
    list_res.append(abs(data[i] - pred[i]))
print(f'Unweighted parameters found: x1 = {x1:.3f}, x2 = {x2:.3f}')
print(f'Residual: {res:.3f}')

plt.scatter(range(len(data)), data, label='data')
plt.plot(range(len(data)), pred, label='prediction', color='red')
plt.legend()
plt.yscale('log')
plt.grid()
plt.show()

```

Unweighted parameters found: x1 = 0.283, x2 = 1.749  
Residual: 0.564



## 5 (b)

Let  $f$  be a vector valued function  $f = [f_1, \dots, f_m]^T$ . In weighted least squares one aims to minimize the objective function

$$\phi(x) = \frac{1}{2} \sum_{i=1}^m W_{ii} (y_i - f_i(x))^2, \quad W_{ii} = \frac{1}{\sigma_i^2},$$

where  $\sigma_i$  is an estimate of the standard deviation in the data point  $y_i$ . This is equivalent to the standard least squares problem

$$\min_x \frac{1}{2} \|Y - F(x)\|_2^2$$

with  $F_i(x) = \frac{1}{\sigma_i} f_i(x)$ ,  $Y_i = \frac{1}{\sigma_i} y_i$ . Assume that for each data point  $y_i$  in the list above, the estimate for the standard deviation is given by

$$\sigma_i = 0.05 y_i.$$

Perform a weighted least squares fit to obtain estimates for  $P_0$  and  $r$ .

Plot the datapoints and the curve fitted to the data again in a semilogarithmic plot.

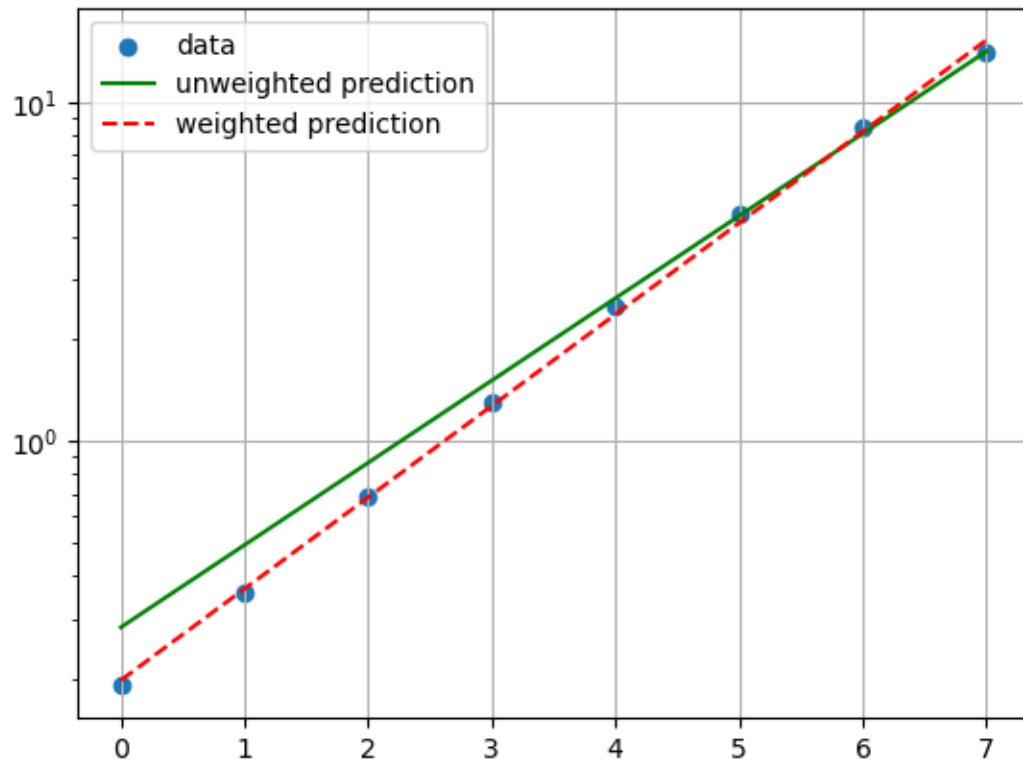
Compare the residuals, i.e. the values of  $y_i - f_i(x)$  obtained in (a) and (b), and discuss the differences between the results of the weighted and the unweighted optimization.

```
[ ]: x_guess = np.array([x1_guess, x2_guess])
x1, x2 = gauss_newton(x1_guess, x2_guess, t, model, data, jacobian, weight=0.05)
pred_weighted = model(x1, x2, t)
res_weighted = np.linalg.norm(data - pred_weighted)
list_res_weighted = []
for i in range(len(data)):
    list_res_weighted.append(abs(data[i] - pred_weighted[i]))
print(f'Weighted parameters found: x1 = {x1:.3f}, x2 = {x2:.3f}')
print(f'Residual: {res_weighted:.3f}')

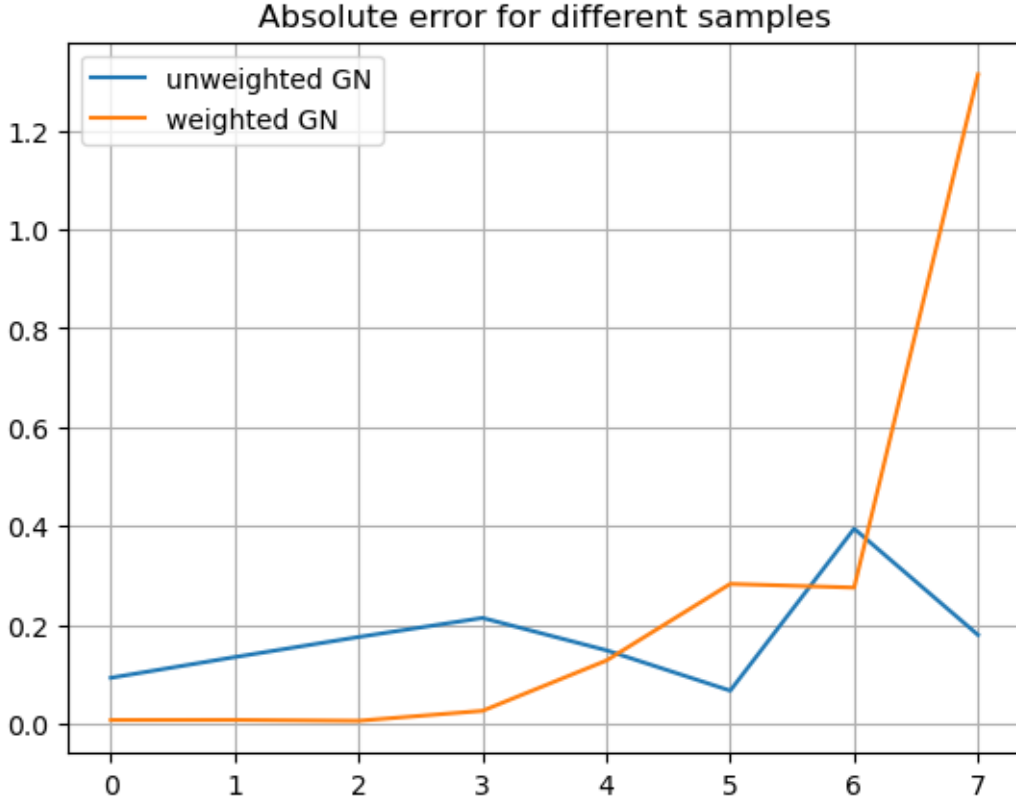
plt.scatter(range(len(data)), data, label='data')
plt.plot(range(len(data)), pred, label='unweighted prediction', color='green')
plt.plot(range(len(data)), pred_weighted, label='weighted prediction',
         color='red', linestyle='--')
plt.yscale('log')
plt.legend()
plt.grid()
plt.show()
```

Weighted parameters found: x1 = 0.197, x2 = 1.862

Residual: 1.378



```
[ ]: # Plot residuals
plt.title('Absolute error for different samples')
plt.plot(t, list_res, label='unweighted GN')
plt.plot(t, list_res_weighted, label='weighted GN')
plt.legend()
plt.grid()
plt.show()
```



As we can tell in the graph above, we find that weighted Gauss-Newton causes absolute errors to be lower for the lower values in the dataset. This is due to the weighting factor of 0.05 used. The overall norm 2 error for the values in the dataset with weighted GN are 1.378 whereas the unweighted norm 2 error is much lower at 0.564. We can see how this relates to the large residual of the 7th data point.

## 6 Exercise 2 (3 points)

A triangle has been measured. The measurements, a vector  $x \in \mathbb{R}^6$ , are as follows:

$$\begin{array}{c|c|c|c|c|c} x_1 = \alpha & x_2 = \beta & x_3 = \gamma & x_4 = a & x_5 = b & x_6 = c \\ \hline 67.5^\circ & 52^\circ & 60^\circ & 172\text{m} & 146\text{m} & 165\text{m} \end{array}.$$

Here  $\alpha, \beta, \gamma$  are the angles opposite the sides with length  $a, b, c$ , respectively. The measurements  $x$  have errors. We would like to correct them so that the new values  $\tilde{x} = x + h$  are consistent quantities of a triangle. They have to satisfy:

$$\begin{array}{ll} \text{Sum of angles:} & \tilde{x}_1 + \tilde{x}_2 + \tilde{x}_3 = 180^\circ \\ \text{Sine theorem:} & \tilde{x}_4 \sin(\tilde{x}_2) - \tilde{x}_5 \sin(\tilde{x}_1) = 0 \\ & \tilde{x}_5 \sin(\tilde{x}_3) - \tilde{x}_6 \sin(\tilde{x}_2) = 0. \end{array} \quad (*)$$

### 6.1 (a)

Solve the constrained least squares problem  $\min_x \|h\|_2^2$  subject to the constraints given by (\*).

Use `scipy.optimize.minimize`.

Hint: Don't forget to work in radians!

Check that for the new values also e.g. the cosine theorem  $c^2 = a^2 + b^2 - 2ab \cos(\gamma)$  holds.

```
[ ]: global constraints
constraints = (
    # 180 degrees in radians
    {'type': 'eq', 'fun': lambda x: x[0] + x[1] + x[2] - np.deg2rad(180)},
    {'type': 'eq', 'fun': lambda x: x[3]*np.sin(x[1]) - x[4]*np.sin(x[0])},
    {'type': 'eq', 'fun': lambda x: x[4]*np.sin(x[2]) - x[5]*np.sin(x[1])},
    {'type': 'eq', 'fun': lambda x: x[3]**2 + x[4]
        ** 2 - 2*x[3]*x[4] * np.cos(x[2]) - x[5]**2}
)

def test_errors(x: np.ndarray, constraint: tuple) -> None:
    i = 1
    for constraint in constraints:
        print(f'C{i}(x) = {constraint["fun"](x)}')
        i += 1

def objective(x: np.ndarray) -> np.ndarray:
    start_x = np.array([67.5, 52, 60, 172, 146, 165])
    start_x[:3] = np.deg2rad(start_x[:3])
    h = start_x - x
    return np.sum(h**2)

x = np.array([67.5, 52, 60, 172, 146, 165])
x[:3] = np.deg2rad(x[:3])
fit = sp.optimize.minimize(objective, x, constraints=constraints)

test_errors(fit.x, constraints)
solution = fit.x
print(solution)
```

C1(x) = 4.440892098500626e-16

C2(x) = 5.684341886080802e-14

C3(x) = -5.684341886080802e-14

C4(x) = 0.0

```
[ 1.16430482  0.90850511  1.06878272 171.25165253 147.02725691
 163.43995528]
```

## 6.2 (b)

You will notice that the corrections will be made mainly to the angles and much less to the lengths of the sides of the triangle. This is because the measurements have not the same absolute errors. While the error in last digit of the sides is about 1, the errors in radians of the angles are about 0.01. Repeat your computation by taking in account with appropriate weighting the difference in measurement errors. Minimize not simply  $\|h\|_2^2$  but

$$\left\| \begin{bmatrix} 100h_1 \\ 100h_2 \\ 100h_3 \\ h_4 \\ h_5 \\ h_6 \end{bmatrix} \right\|_2^2.$$

```
[ ]: def objective(x: np.ndarray) -> np.ndarray:
    start_x = np.array([67.5, 52, 60, 172, 146, 165])
    start_x[:3] = np.deg2rad(start_x[:3])
    h = start_x - x
    h[:3] = h[:3] * 100
    return np.sum(h**2)

fit = sp.optimize.minimize(objective, x, constraints=constraints)
test_errors(fit.x, constraints)
weighted_solution = fit.x

standard_dif = x - solution
standard_dif[:3] = np.rad2deg(standard_dif[:3])

weighted_dif = x - weighted_solution
weighted_dif[:3] = np.rad2deg(weighted_dif[:3])

print("Standard Solution: ", end="")
print(np.array_str(standard_dif, precision=3, suppress_small=True))
print("Weighted Solution: ", end="")
print(np.array_str(weighted_dif, precision=3, suppress_small=True))

C1(x) = 0.0
C2(x) = -3.410605131648481e-13
C3(x) = -4.547473508864641e-13
C4(x) = -2.764863893389702e-10
Standard Solution: [ 0.79 -0.054 -1.237  0.748 -1.027  1.56 ]
Weighted Solution: [ 0.277  0.042 -0.819 -0.321 -1.192  1.824]
```