# L_Bijman_S_Gijsbers_HW5

December 5, 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.

Loes Bijman 15211312

Sacha Gijsbers 12798525

## 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

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.

[ ]:
```python
# define Gauss-Newton here

def fit_function(t, x0):
    '''function used to fit to the data'''
    return x0[0] * x0[1]**t

def Jacobian_unweighted(t_data, x0, y_data):
    '''returns the Jacobian matrix of the unweighted residual function'''
    return np.column_stack((-x0[1]**t_data, -t_data * x0[0] *␣
 ↪x0[1]**(t_data-1)))

def residual(t_data,y_data,function,x0):
    return y_data - function(t_data, x0)

def Gauss_Newton(t_data, y_data, function, x0, max_iterations,␣
 ↪Jacobian_function = Jacobian_unweighted, residual_function = residual):
    x = x0
    for _ in range(max_iterations):
        s = np.linalg.
 ↪lstsq(Jacobian_function(t_data,x,y_data),-residual_function(t_data,y_data,function,x),␣
 ↪rcond=None)[0]
        x += s
    return x
```

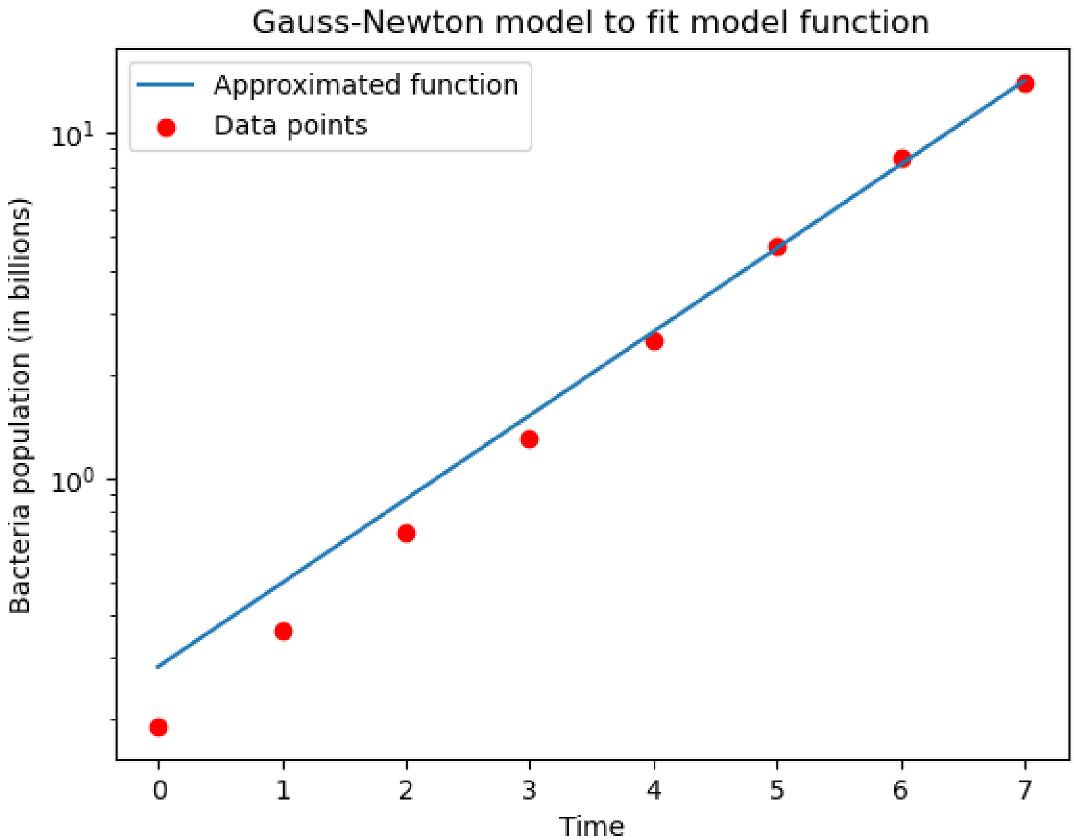
[ ]:
```python
import matplotlib.pyplot as plt

# make definitions for bacterial model and run Gauss-Newton here
t_data = np.array([0.0, 1, 2, 3, 4, 5, 6, 7])

x0 = [1,1]
max_iterations = 10

x_gauss = Gauss_Newton(t_data, data, fit_function, x0, max_iterations)

plt.plot(fit_function(t_data, x_gauss), label = 'Approximated function')
plt.scatter(t_data, data, color = 'red', label = 'Data points')
plt.yscale('log')
plt.title('Gauss-Newton model to fit model function')
plt.ylabel('Bacteria population (in billions)')
plt.xlabel('Time')
plt.legend()
plt.show()
```




## 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

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

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 $F_i(x) = \frac{1}{\sigma_i} f(x)$ , $Y_i = \frac{1}{\sigma_i} y_i$. 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.

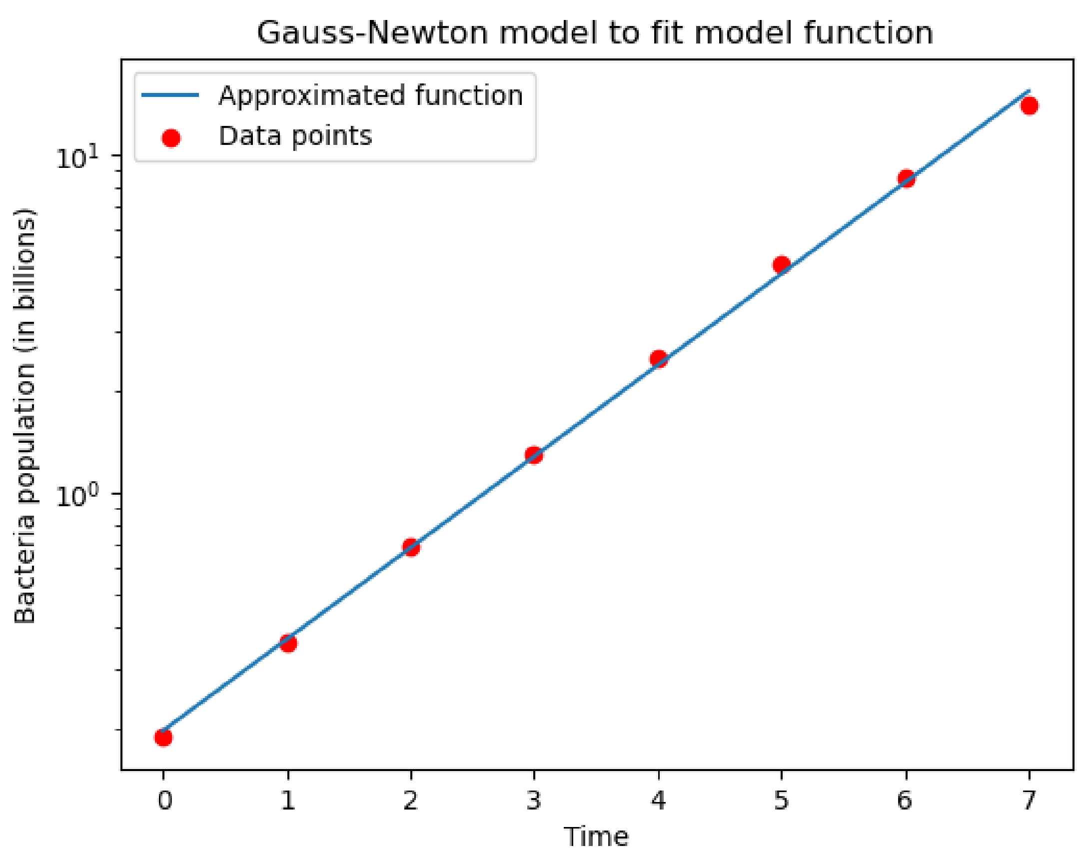
Compare the residuals, i.e. the values of $y_i - f_i(x))$ obtained in (a) and (b), and discuss the differences between the results of the weighted and the unweighted optimization.

[ ]:
```
def Jacobian_weighted(t_data, x0, y_data):
    '''returns the Jacobian matrix of the unweighted residual function'''
    return np.column_stack((-1/(0.05*y_data) * x0[1] ** t_data, -1/(0.
↪05*y_data) * x0[0] * t_data * x0[1] ** (t_data-1)))

def residual_weighted(t_data,y_data,function,x0):
    return 1/0.05 - 1/(0.05*y_data) * x0[0]*x0[1]**t_data
```

[ ]:
```
x_weight = Gauss_Newton(t_data, data, fit_function, x0, max_iterations,␣
↪Jacobian_function = Jacobian_weighted, residual_function = residual_weighted)

plt.plot(fit_function(t_data, x_weight), label = 'Approximated function')
plt.scatter(t_data, data, color = 'red', label = 'Data points')
plt.yscale('log')
plt.title('Gauss-Newton model to fit model function')
plt.ylabel('Bacteria population (in billions)')
plt.xlabel('Time')
plt.legend()
plt.show()
```



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_unweighted = Gauss_Newton(t_data, data, fit_function, x0, max_iterations)
x_weight = Gauss_Newton(t_data, data, fit_function, x0, max_iterations,␣
 ↪Jacobian_function = Jacobian_weighted, residual_function = residual_weighted)

res_gauss = np.sum(abs(residual(t_data,data,fit_function,x_gauss)))
res_weight = np.sum(abs(residual(t_data,data,fit_function,x_weight)))

print(f"Sum of absolute residuals unweighted: {res_gauss}")
print(f"Sum of absolute residuals weighted: {res_weight}")

norm_gauss = np.linalg.norm(residual(t_data,data,fit_function,x_gauss))**2
norm_weight = np.linalg.norm(residual(t_data,data,fit_function,x_weight))**2

print(f"Norm of residual vector unweighted: {norm_gauss}")
print(f"Norm of residual vector weighted: {norm_weight}")

relative_res_gauss = np.sqrt(norm_gauss)/(np.linalg.
 ↪norm(Jacobian_unweighted(t_data,x0,data))*np.linalg.
 ↪norm(-residual(t_data,data,fit_function,x0)))
relative_res_weight = np.sqrt(norm_weight)/(np.linalg.
 ↪norm(Jacobian_weighted(t_data,x0,data))*np.linalg.
 ↪norm(-residual_weighted(t_data,data,fit_function,x0)))

print(f"Relative residual unweighted: {relative_res_gauss}")
print(f"Relative residual weighted: {relative_res_weight}")
```

```
Sum of absolute residuals unweighted: 1.4061964887309202
Sum of absolute residuals weighted: 2.0465074792308724
Norm of residual vector unweighted: 0.31774312792834714
Norm of residual vector weighted: 1.8996643171878163
Relative residual unweighted: 0.002975798192748197
Relative residual weighted: 8.738817219644113e-05
```

We observe that the absolute value of the residuals is larger when LLS is performed with the weighted residual function. We explain this by the fact that the unweighted residual is fitted more to the higher values of the function, whereas the weighted residual line is closer to the lower values, but further from one high value.

However, we can also look at the relative residual, given by $\frac{||r(x)||_2}{||J(x_0)||_2 \cdot ||r(x_0)||_2}$ (where $x_0$ is the initial guess and $x$ the final solution). We observe that the weigted residual function has a much lower relative residual. The relative error in the solution is thus smaller for the weighted residual function than for the unweighted one.

## 6 Exercise 2 (3 points)

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

| $x_1 = \alpha$ | $x_2 = \beta$ | $x_3 = \gamma$ | $x_4 = a$ | $x_5 = b$ | $x_6 = c$ |
|---|---|---|---|---|---|
| 67.5° | 52° | 60° | 172m | 146m | 165m |

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. The have to satisfy:

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

### 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.

[ ]:
```
import math
import scipy

x = [math.radians(67.5), math.radians(52), math.radians(60), 172, 146, 165]

fun = lambda h: np.linalg.norm(h)**2

constraints = ({'type': 'eq', 'fun': lambda h: x[0]+ h[0] + x[1] + h[1] + x[2]␣
↪+ h[2] - math.radians(180)},
               {'type': 'eq', 'fun': lambda h: (x[3] + h[3]) *np.sin(x[1] +␣
↪h[1]) - (x[4] + h[4])*np.sin(x[0] + h[0])},
               {'type': 'eq', 'fun': lambda h: (x[4] + h[4]) * np.sin(x[2] +␣
↪h[2]) - (x[5] + h[5]) * np.sin(x[1] + h[1])})

initial_guess = np.zeros(6)
h = scipy.optimize.minimize(fun, initial_guess,constraints=constraints).x

x_new = x + h

cos_theorem_old = -x[5]**2 + x[3]**2 + x[4]**2 - 2*x[3]*x[4]*np.cos(x[2])
cos_theorem = -x_new[5]**2 + x_new[3]**2 + x_new[4]**2 - 2*x_new[3]*x_new[4]*np.
↪cos(x_new[2])
print(f'The cosine theorem for the original x values gives {cos_theorem_old}.')
print(f'The cosine theorem for the adjusted x values gives {cos_theorem}, which␣
↪is close to zero.')
```

```
x_dif = x_new - x
print(f'The difference between the old and new x values {x_dif},\nindicating␣
↪that the angles are adjusted more than the lengths of the sides.')

x_new[0] = math.degrees(x_new[0])
x_new[1] = math.degrees(x_new[1])
x_new[2] = math.degrees(x_new[2])
print(f'The new x value modified by h is {x_new}.')
```

```
The cosine theorem for the original x values gives -1437.0000000000073.
The cosine theorem for the adjusted x values gives -8.487404556944966e-09, which
is close to zero.
The difference between the old and new x values [-0.01169255 -0.01231171
0.0327309   0.0002013   0.00012197 -0.00032557],
indicating that the angles are adjusted more than the lengths of the sides.
The new x value modified by h is [ 66.83006608  51.29459126  61.87534266
172.0002013  146.00012197
 164.99967443].
```

### 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

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

[ ]:
```
import math

x = [math.radians(67.5), math.radians(52), math.radians(60), 172, 146, 165]

fun = lambda h: np.linalg.norm([100 * h[0], 100 * h[1], 100 * h[2], h[3], h[4],␣
↪h[5]])**2

constraints = ({'type': 'eq', 'fun': lambda h: x[0]+ h[0] + x[1] + h[1] + x[2]␣
↪+ h[2] - math.radians(180)},
               {'type': 'eq', 'fun': lambda h: (x[3] + h[3]) * np.sin(x[1] +␣
↪h[1]) - (x[4] + h[4]) * np.sin(x[0] + h[0])},
               {'type': 'eq', 'fun': lambda h: (x[4] + h[4]) * np.sin(x[2] +␣
↪h[2]) - (x[5] + h[5]) * np.sin(x[1] + h[1])})
```

```python
initial_guess = np.zeros(6)
h = scipy.optimize.minimize(fun, initial_guess, constraints = constraints).x

x_new = x + h

cos_theorem_old = -x[5]**2 + x[3]**2 + x[4]**2 - 2 * x[3] * x[4] * np.cos(x[2])
cos_theorem = -x_new[5]**2 + x_new[3]**2 + x_new[4]**2 - 2 * x_new[3] *␣
  ↪x_new[4] * np.cos(x_new[2])
print(f'The cosine theorem for the original x values gives {cos_theorem_old}')
print(f'The cosine theorem for the adjusted x values gives {cos_theorem}, which␣
  ↪is close to zero')

x_dif = x_new - x
print(f'The difference between the old and new x values {x_dif},\nindicating␣
  ↪that the angles are adjusted more the the lengths of the sides')

x_new[0] = math.degrees(x_new[0])
x_new[1] = math.degrees(x_new[1])
x_new[2] = math.degrees(x_new[2])
print(f'The new x value modified by h is {x_new}')
```

```
The cosine theorem for the original x values gives -1437.0000000000073
The cosine theorem for the adjusted x values gives -1.9273284124210477e-07,
which is close to zero
The difference between the old and new x values [-0.00194679 -0.00574568
0.01641912  0.71863495  0.77295329 -1.45208222],
indicating that the angles are adjusted more the the lengths of the sides
The new x value modified by h is [ 67.38845729  51.67079658  60.94074613
172.71863495 146.77295329
 163.54791778]
```