Homework set 3

Please **submit this Jupyter notebook through Canvas** no later than **Monday December 2**. **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. On canvas there are hints about creating a nice pdf version.**

Before you hand in, please make sure the notebook runs, by running "Restart kernel and run all cells..." from the Kernel 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.

Exercise 0

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

Noa Roebersen, 12247014

Paul Jungnickel, 15716554

Run the following cell to import NumPy, Matplotlib. If anything else is needed you can import this yourself.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        np.set_printoptions(precision=9, floatmode='fixed')
```

Exercise 1: Nonlinear least squares

This exercise is about the Gauss-Newton method, and the Levenberg-Marquardt method, which are discussed in section 6.6 of Heath. Please read this section before making this homework set. **In this exercise set the Levenberg-Marquardt method is a little different from the one in Heath. The first equation in subsection 6.6.2 is replaced by**

$$\left(\boldsymbol{J}^T(\boldsymbol{x}_k)\boldsymbol{J}(\boldsymbol{x}_k) + \mu_k \,\text{Diagonal}\left(\boldsymbol{J}^T(\boldsymbol{x}_k)\boldsymbol{J}(\boldsymbol{x}_k)\right)\right)\boldsymbol{s}_k = -\boldsymbol{J}^T(\boldsymbol{x}_k)\,\boldsymbol{r}(\boldsymbol{x}_k)$$

Here $\text{Diagonal}(\boldsymbol{B})$ denotes the diagonal part of $\boldsymbol{B}$. So $\text{Diagonal}(\boldsymbol{B})$ has the same shape as $\boldsymbol{B}$ and identical entries on the diagonal and it has zero off-diagonal entries.

The algorithm for Levenberg-Marquardt, with μ_k constant (denoted μ here), is then

```
$\boldsymbol{x}_0 = \text{initial guess}$    $\mu = \text{constant}$    $\backslash\text{bf for } k = 0, 1, 2, \ldots$    $\boldsymbol{A} = \boldsymbol{J}_f(\boldsymbol{x}_k)$
    solve $\boldsymbol{s}_k$ from $(\boldsymbol{A}^T\boldsymbol{A} + \mu\,\text{Diagonal}(\boldsymbol{A}^T\boldsymbol{A}))\boldsymbol{s}_k = -\boldsymbol{A}^T\boldsymbol{r}(\boldsymbol{x}_k)$    $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{s}_k$
$\backslash\text{bf end}$
```

This reduces to the Gauss-Newton method if $\mu = 0$.

(a)

Implement the Levenberg-Marquardt method with constant μ using a suitable stopping criterion. Make it such that the user can specify the value of the tolerance in the stopping criterion via a parameter `tol` and the maximum number of iterations via a parameter `maxIter`. In the implementation you can use library functions for linear algebra operations.

In [2]:
```
def nonlinearLeastSquares(t, y, f, J,  x0, mu, tol=1e-6, max_iter = 100):
    """"
    Solves the nonlinear least squares problem with the Levenberg-Marquardt method

    Arguments:
        - t, y : data to be fitted as time series
        - f, J : Model and its jacobian for all t
        - x0: starting guess
        - mu: sensitivity of Levenberg-Marquardt - smaller mu mean more agressive but less ac
        - tol: tolerance of the approximation
        - max_iter: maximum number of iterations

    Returns:
        - x: model parameters that give the best fit
    """

    k, xk, sk = 0, x0.copy(), 1e100*np.ones_like(x0),
    rk = y - f(xk, t)

    print("iteration,\t\t x, \t\t\t\t |r|")

    while(np.linalg.norm(sk) > tol and k<max_iter):

        print(k+1,'\t\t', xk,'\t\t',  np.linalg.norm(rk))

        A = J(xk, t)

        Arr = A.T @ A
        Arr = Arr + mu*np.diag(np.diag(Arr))

        rk = y - f(xk, t)
        JTrk = -A.T @  rk

        if not (np.isnan(Arr).any() or np.isnan(JTrk).any()):
            sk, _, _, _ = np.linalg.lstsq(Arr, JTrk)
        else:
            print("Method diverged")
            return xk


        xk += sk

        k+=1


    return xk
```

(b)

The time course of drug concentration y in the bloodstream is well described by

$$y = c_1 t e^{c_2 t}, \tag{1}$$

where t denotes time after the drug was administered. The characteristics of the model are a quick rise as the drug enters the bloodstream, followed by slow exponential decay. The half-life of the drug is the time from the peak concentration to the time it drops to half that level. The measured level of the drug norfluoxetine in a patient's bloodstream at whole hours after it was administered is given in the following data:

In [3]:
```python
# time in hours
hour = np.array( [ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 ] )
# concentration in ng/ml
concentration = np.array( [ 8.0, 12.3, 15.5, 16.8, 17.1, 15.8, 15.2, 14.0 ] )
```

Use the Gauss-Newton method to fit this data to the blood concentration model (1).

Also use the Levenberg-Marquardt method with $\mu = 0.1$ to address the same problem.

Which method produces the least number of iterations? N.B. clearly state the starting point.

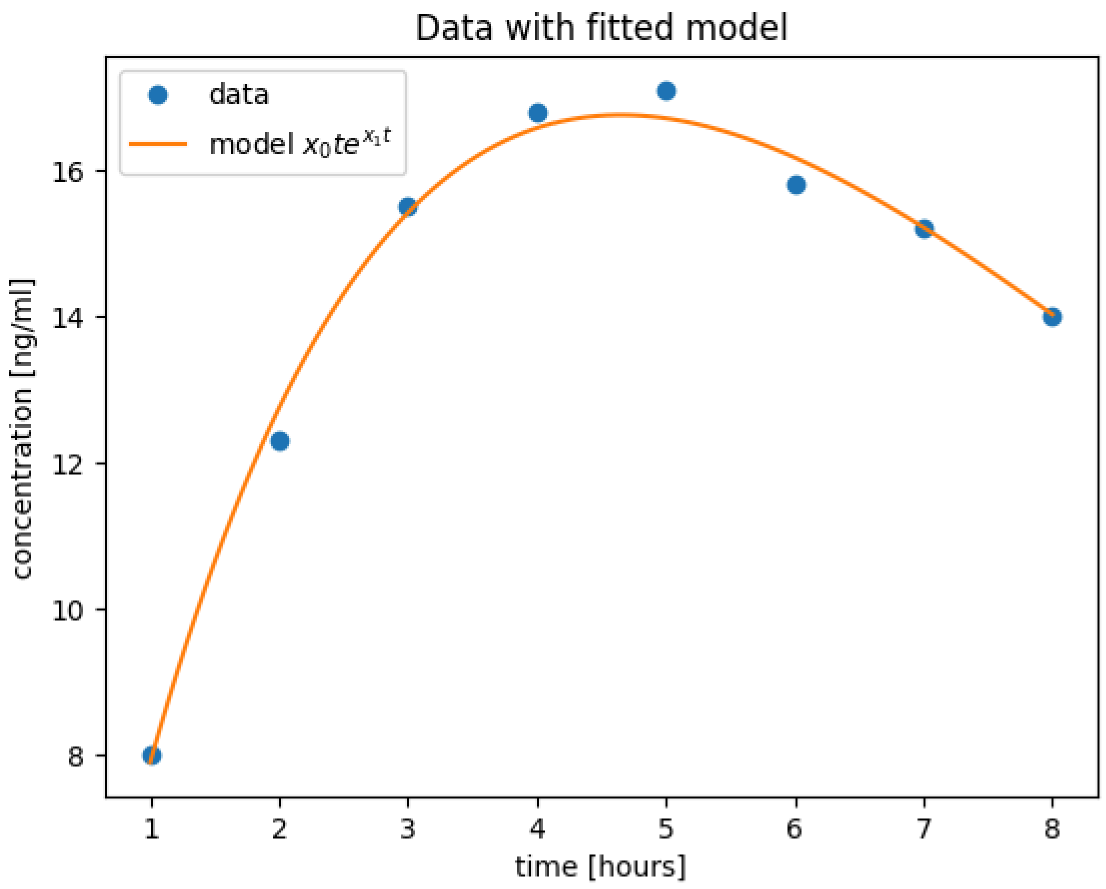
You are asked to use your own version of Gauss-Newton and Levenberg-Marquardt.

We select a starting point of $c_1 = 10$ and $c_2 = -0.1$, since c_1 has to be negative to get a decay and c_1 should roughly correspond to the initial concentration value.

The Gauss-Newton (GN) method converges much more quickly with only 6 iterations compared to 29 vor Levenberg-Marquardt (LM):

In [4]:
```python
# YOUR CODE HERE
#Defining the model function and its Jacobian:
def f(x,t):
    return x[0] * t * np.exp(x[1]*t)

def Jf(x, tArr):
    return -1.* np.array([[t*np.exp(x[1]*t), x[0]*(t**2)*np.exp(x[1]*t)] for t in tArr])



#starting guess
x0 = np.array([10.,-.1])

#solve the system with Gauss-Newton:
mu = 0.0
x = nonlinearLeastSquares(hour, concentration, f, Jf, x0, mu)

#plot the fit cuve along with the data
plt.plot(hour, concentration, marker='o', linestyle='', label='data')

tArr = np.linspace(hour[0], hour[-1], 100)
plt.plot(tArr, f(x, tArr), label=r'model $x_0 t e^{x_1 t}$')
plt.title("Data with fitted model")
plt.legend()
plt.xlabel('time [hours]')
plt.ylabel('concentration [ng/ml]')
plt.show()
```

```
iteration,                x,                                |r|
1                 [10.000000000 -0.100000000]          38.675356654423254
2                 [ 8.792056496 -0.162944795]          38.675356654423254
3                 [ 9.572886302 -0.208086990]          8.436967040747804
4                 [ 9.794300113 -0.215053960]          1.0519331050096865
5                 [ 9.796937581 -0.215087368]          0.7445402119698231
6                 [ 9.796928097 -0.215087165]          0.744522497943086
```



Solving the same LLS problem with the Levenberg-Marquardt method where $\mu = 0.1$, we get the same result, but this method requires 29 iterations to converge.

In [5]:
```
print("\nLM converges in 29 iterations:")
mu = 0.1
x = nonlinearLeastSquares(hour, concentration, f, Jf, x0, mu)
```

```
LM converges in 29 iterations:
iteration,               x,                             |r|
1                 [10.000000000 -0.100000000]           38.675356654423254
2                 [ 8.036639361 -0.146386970]           38.675356654423254
3                 [ 8.130055938 -0.176902693]           9.241958119231146
4                 [ 8.680267888 -0.192967155]           3.2080274448463144
5                 [ 9.117483609 -0.201951934]           1.9897660366801144
6                 [ 9.394373355 -0.207323967]           1.331557851806397
7                 [ 9.560967352 -0.210535298]           0.9885043711008963
8                 [ 9.659411863 -0.212433170]           0.835971588997995
9                 [ 9.717051801 -0.213545101]           0.7767114791721915
10                [ 9.750622475 -0.214193026]           0.7555213520850246
11                [ 9.770114507 -0.214569345]           0.7482342308982924
12                [ 9.781411781 -0.214787495]           0.7457686186620067
13                [ 9.787952669 -0.214913813]           0.7449399142820452
14                [ 9.791737416 -0.214986910]           0.7446621759147076
15                [ 9.793926612 -0.215029192]           0.7445692136900429
16                [ 9.795192643 -0.215053645]           0.7445381179314808
17                [ 9.795924713 -0.215067785]           0.7445277198454713
18                [ 9.796347998 -0.215075960]           0.7445242434507496
19                [ 9.796592732 -0.215080687]           0.7445230813001498
20                [ 9.796734229 -0.215083420]           0.7445226928172529
21                [ 9.796816036 -0.215085001]           0.7445225629595168
22                [ 9.796863334 -0.215085914]           0.7445225195528963
23                [ 9.796890679 -0.215086442]           0.7445225050438214
24                [ 9.796906488 -0.215086748]           0.7445225001940552
25                [ 9.796915628 -0.215086924]           0.7445224985729935
26                [ 9.796920913 -0.215087026]           0.7445224980311406
27                [ 9.796923968 -0.215087085]           0.7445224978500234
28                [ 9.796925734 -0.215087119]           0.7445224977894856
29                [ 9.796926755 -0.215087139]           0.7445224977692513
```

(c)

Try to find a starting point such that Gauss-Newton does not converge, while Levenberg-Marquardt does.

Just changing c_2 from -0.1 to -0.5 is enough:

In [6]:
```python
#worse starting guess
x0 = np.array([10.,-.5])

print("Gauss-Newton does not converge:")
mu = 0.0
x = nonlinearLeastSquares(hour, concentration, f, Jf, x0, mu)



print("\nLM does converge in 33 iterations:")
mu = 0.1
x = nonlinearLeastSquares(hour, concentration, f, Jf, x0, mu)


#
```

```
Gauss-Newton does not converge:
iteration,              x,                              |r|
1               [10.000000000 -0.500000000]             29.946569860739206
2               [-0.328039244  0.375286700]             29.946569860739206
3               [1.245255960 0.824976078]               98.76036952612799
4               [0.067537434 0.818398955]               7901.186906285651
5               [0.070671012 0.691523926]               388.0032991363594
6               [0.167793112 0.404673688]               139.83195023918435
7               [ 1.048295947 -0.337841020]             33.5981236929678
8               [8.077503899 1.661827978]               38.82545198787655
9               [1.814825201e-04 1.661825214e+00]               38923722.6821352
10              [1.814863936e-04 1.538788812e+00]               858.5353716221063
11              [4.359061466e-04 1.243769969e+00]               313.15484111365345
12              [0.003557474 0.245878882]               67.90141099496918
13              [  2.521335477 -81.777220918]           41.10544171522665
14              [3.562683106e+35 8.982719307e+35]               41.31428324441803
Method diverged

LM does converge in 33 iterations:
iteration,              x,                              |r|
1               [10.000000000 -0.500000000]             29.946569860739206
2               [8.202356864 0.092473713]               29.946569860739206
3               [4.958349251 0.033567916]               180.72605768110216
4               [ 4.127631647 -0.026472880]             53.65706293362881
5               [ 4.618341597 -0.083418973]             18.02193553702995
6               [ 5.778160028 -0.128835517]             10.30613906992383
7               [ 7.032710597 -0.160167568]             7.358632656394215
8               [ 8.029315114 -0.180812732]             4.847623662204784
9               [ 8.711383068 -0.194157876]             3.0456896238315783
10              [ 9.145961662 -0.202543979]             1.9275550997261057
11              [ 9.412231125 -0.207670577]             1.2920345125319754
12              [ 9.571607936 -0.210740505]             0.9696058488711363
13              [ 9.665662371 -0.212553725]             0.8282907221750151
14              [ 9.720698642 -0.213615475]             0.7739016203405268
15              [ 9.752742077 -0.214233944]             0.7545456549172762
16              [ 9.771343718 -0.214593080]             0.7479028034021646
17              [ 9.782123712 -0.214801243]             0.745657042289212
18              [ 9.788364694 -0.214921771]             0.7449024927724547
19              [ 9.791975769 -0.214991513]             0.7446496462409808
20              [ 9.794064463 -0.215031855]             0.7445650218109073
21              [ 9.795272357 -0.215055185]             0.744536716085656
22              [ 9.795970805 -0.215068675]             0.7445272511428606
23              [ 9.796374647 -0.215076475]             0.7445240867602492
24              [ 9.796608140 -0.215080985]             0.7445230289209391
25              [ 9.796743137 -0.215083593]             0.744522675308359
26              [ 9.796821187 -0.215085100]             0.7445225571069135
27              [ 9.796866311 -0.215085972]             0.7445225175966032
28              [ 9.796892400 -0.215086476]             0.7445225043899151
29              [ 9.796907483 -0.215086767]             0.7445224999754844
30              [ 9.796916204 -0.215086935]             0.7445224984999304
31              [ 9.796921245 -0.215087033]             0.7445224980067223
32              [ 9.796924160 -0.215087089]             0.7445224978418618
33              [ 9.796925845 -0.215087122]             0.7445224977867586
```

```
/tmp/ipykernel_31148/2521306670.py:7: RuntimeWarning: overflow encountered in exp
  return -1.* np.array([[t*np.exp(x[1]*t), x[0]*(t**2)*np.exp(x[1]*t)] for t in tArr])
/tmp/ipykernel_31148/1647872752.py:29: RuntimeWarning: invalid value encountered in multiply
  Arr = Arr + mu*np.diag(np.diag(Arr))
/tmp/ipykernel_31148/2521306670.py:4: RuntimeWarning: overflow encountered in exp
  return x[0] * t * np.exp(x[1]*t)
```

(d)

So far for simplicity, we considered constant μ. However Levenberg-Marquardt is often applied adaptively with a varying μ. A common strategy is to continue to decrease μ by a factor of 10 on each iteration step as long as the residual sum of squared errors is decreased by the step, and if the sum increases, to reject the step and increase μ by a factor of 10.

Implement an adaptive variant of Levenberg-Marquardt using such a strategy for choosing μ.

Compare the performance (iteration number) of the adaptive variant with Gauss-Newton and the previous, non-adaptive variant of Levenberg-Marquardt. Consider a starting point for which Gauss-Newton converged rapidly and a starting point for which Gauss-Newton did not converge, but non-adaptive Levenberg-Marquardt did. Give your answer in a table for clarity, also indicating the starting point and if relevant other parameters.

In [7]:
```python
def nonlinearLeastSquares(t, y, f, J,  x0, mu=0.1, tol=1e-6, max_iter = 100):
    """"
    Solves the nonlinear least squares problem with the adaptive Levenberg-Marquardt method

    Arguments:
        - t, y : data to be fitted as time series
        - f, J : Model and its jacobian for all t
        - x0: starting guess
        - mu: starting sensitivity of Levenberg-Marquardt - smaller mu mean more agressive bu
              gets automatically adapted during the iterations
        - tol: tolerance of the approximation
        - max_iter: maximum number of iterations

    Returns:
        - x: model parameters that give the best fit
    """
    k, xk, sk, = 0, x0.copy(), 1e100*np.ones_like(x0)
    rk = y - f(xk, t)

    print("iteration,\t\t x, \t\t\t\t |r|")

    while(np.linalg.norm(sk) > tol and k<max_iter):

        print(k+1,'\t\t', xk,'\t\t',  np.linalg.norm(rk))

        A = J(xk, t)

        Arr = A.T @ A
        Arr = Arr + mu*np.diag(np.diag(Arr))

        JTrk = -A.T @  rk

        if not (np.isnan(Arr).any() or np.isnan(JTrk).any()):
            sk, _, _, _ = np.linalg.lstsq(Arr, JTrk)
        else:
            print("Method diverged")
            return xk


        #only accept the iteration if residual norm decreases and adapt mu
        rk_new = y - f(xk + sk, t)

        errorDiff = np.linalg.norm(rk_new) - np.linalg.norm(rk)
        if errorDiff > 0:
            mu *= 10

        else:
            mu /= 10
            xk += sk
            rk = rk_new


        k+=1



    return xk
```

For testing the method we use the same points [10,-0.1] and [10,-0.5] as in b) and c)

Starting point where both methods converged: LM now takes 6 compared to 29 iterations, same as GN.

In [8]:
```
#worse starting guess
x0 = np.array([10.,-.1])


print("\nLM converges in 6 iterations:")
mu = 0.1
x = nonlinearLeastSquares(hour, concentration, f, Jf, x0, mu)
```

```
LM converges in 6 iterations:
iteration,                x,                              |r|
1                [10.000000000 -0.100000000]             38.675356654423254
2                [ 8.036639361 -0.146386970]             9.241958119231146
3                [ 9.108543164 -0.199396824]             1.4967026318546588
4                [ 9.768127861 -0.214916450]             0.7497140086532463
5                [ 9.796947768 -0.215088056]             0.7445225076836771
6                [ 9.796927904 -0.215087161]             0.7445224977592114
```

Starting point where GN diverged: LM now converges in 8 compared to 33 iterations.

In [9]:
```
#worse starting guess
x0 = np.array([10.,-.5])


print("\nLM converges in 8 iterations:")
mu = 0.1
x = nonlinearLeastSquares(hour, concentration, f, Jf, x0, mu)
```

```
LM converges in 8 iterations:
iteration,                x,                              |r|
1                [10.000000000 -0.500000000]             29.946569860739206
2                [10.000000000 -0.500000000]             29.946569860739206
3                [14.085637392 -0.251234309]             9.315290348481764
4                [11.527824389 -0.246102451]             2.727389240628528
5                [ 9.894995889 -0.218874514]             0.8789256294459421
6                [ 9.796460987 -0.215072430]             0.7445241581336867
7                [ 9.796931516 -0.215087237]             0.7445224977817514
8                [ 9.796928135 -0.215087166]             0.7445224977590913
```