Introduction to Numerical Data Science, RUB

Report on the Gauss Newton method

by Peter Moeller

January 31st, 2025

Introduction

In this report, we will take a closer look at the Gauss-Newton method for solving nonlinear least-squares problems. We will use the Python library TensorFlow for many of the operations involved, such as the computing of the Jacobian and solving systems of equations. We will show the importance of regularization in nonlinear least-squares problems, which in some cases would be insoluble without this technique, which we will illustrate in what follows through various plots.

We will also take reader through the gist of deriving the Gauss-Newton method, which can be an extremely useful tool, provided that the initial guess for the solution is within range (unless the function is quite simple) and a regularization term is added to the approximation for the Hessian, as we will see.

Outline of the task

The nonlinear least-squares problem can be stated as follows:

$$\min_{x\in\mathbb{R}^3} F(x) = \min_{x\in\mathbb{R}^3} \frac{1}{2}\sum_{i=1}^{n}(z_i - z(y_i;x))^2, \tag{1}$$

with $(y_i, z_i)_{i=1,\ldots,m}$ being the data points provided beforehand. The $\frac{1}{2}$ in front of the expression, will, as we will see, cancel out, once we derive the Jacobian (the derivative), and will allow us to avoid having to "carry a 2 around with us" after that. The function relationship given for our task is:

$$\hat{z}(y;x) = x_1 y + x_2 \cos(x_3 y). \tag{2}$$

The data points are:

$y =$ `(0.0000000, 0.6283185, 1.2566371, 1.8849556,`
`2.513274, 3.1415927, 3.7699112, 4.3982297,`
`5.0265482, 5.6548668, 6.2831853)`

$z =$ `(0.9299887, 0.53383386, -0.15017393, 0.11093735,`
`1.5128875, 2.4723399, 2.2487612, 1.3162203,`
`1.6767914, 3.3423154, 4.0957375)`

The points were generated by sampling the function on equidistant points within the domain for the true solution $x = (0.5, 1, 2)^T$ but overlaid with random noise of small magnitude.

We seek to implement the solution, i.e. to find the correct parameters for x, via the Gauss-Newton method in Python making use of the library TensorFlow to automate the computation of the Jacobian (the derivative). Before moving on to the implementation in Python, we will lay from theoretical groundwork for the Gauss-Newton method.

Theoretical foundation

The standard version of Newton's Method uses the first-order Taylor expansion around a point x_k:

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k),$$

and rearranges it to calculate the next step in the iteration:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)},$$

setting $f'(x) \neq 0$ (or, in the Gauss-Newton method, that the determinant of $J(x)^T J(x)$, the approximation for the Hessian, must be non-zero). We then repeat the steps with x_{k+1} and so on until reaching the desired precision. One caveat of importance is that the initial guess x_0 is crucial, if the function is even slightly more complex. Take for instance the function $f(x) = 0.1x + cos(3x) + 1$:

This function is quite similar to function (2) given above (with the values for $x = (0.5, 1, 2)^T$ inserted):

The problem potentially arising, when working trigonometric functions among others, is, that Newton's method can often become instable. If we, for instance, seem to get closer to an (almost) root (i.e. $f(x) = 0$) around $x = 1$ in Figure 1, we might reach a point, where the computed tangent is close to horizontal (i.e. $f'(x) \approx 0$), and the next x_k-step seem to shoot off in a completely different direction. If we, however, make an initial guess of $x = -0.5$, we will quite quickly and stably arrive at a root around $x = 0.904$. So, an inital guess for x_0 is essential. To make up for this, we will introduce a regularization term

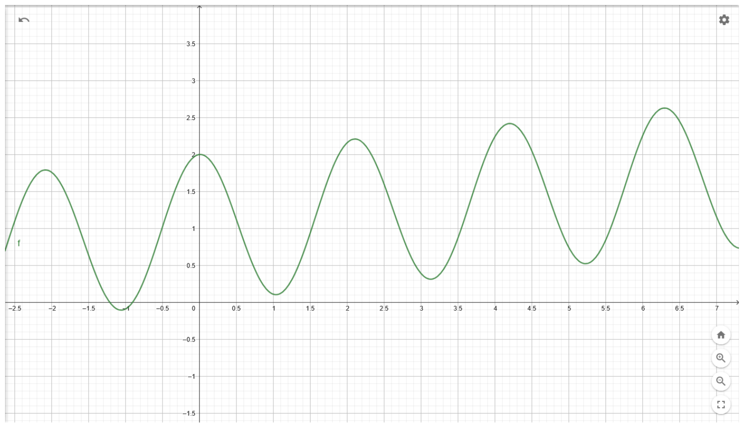


Figure 1: Plot of the function $f(x) = 0.1x + cos(3x) + 1$

into the algorithm. But before addressing this concept, we wish to expand this simple, but potentially very powerful idea, to the Gauss-Newton method.

We seek to minimize a given function (which is equivalent to applying Newton's method to its derivative, and confirming that it is a minimum, for example by checking that its second derivative is positve at that point). In our case, we seek to minimize $F(x)$, and, setting the **residual** $r_i(x)$ equal to the expression $z_i - z(y_i; x)$, we therefore get:

$$\min_{x\in\mathbb{R}^3} F(x) = \min_{x\in\mathbb{R}^3} \frac{1}{2}\sum_{i=1}^{n} r_i(x)^2 = \min_{x\in\mathbb{R}^3} \frac{1}{2}\|r(x)\|^2,$$

By again applying the Taylor expansion, we get:

$$r_i(x) \approx r_i(x_k) + J_i(x_k)(x - x_k).$$

Insert this into the objective function:

$$F(x) \approx \frac{1}{2}\|r(x_k) + J(x_k)(x - x_k)\|^2$$

This is what we will then have to take the derivative of with respect to $x \in \mathbb{R}^3$. In the Python code, we will use the automatic differentiation feature in TensorFlow, specifically GradientTape, but will here give an overview, such that the reason for the expressions used in the implemented function `gauss_newton` is clear.

$$\begin{aligned} F(x) &= \tfrac{1}{2}\left((r(x_k) + J(x_k)(x - x_k))^T (r(x_k) + J(x_k)(x - x_k))\right) \\ &= \tfrac{1}{2}\left[\|r(x_k)\|^2 + 2(r(x_k))^T J(x_k)(x - x_k) + (x - x_k)^T J(x_k)^T J(x_k)(x - x_k)\right] \end{aligned}$$

We notice:

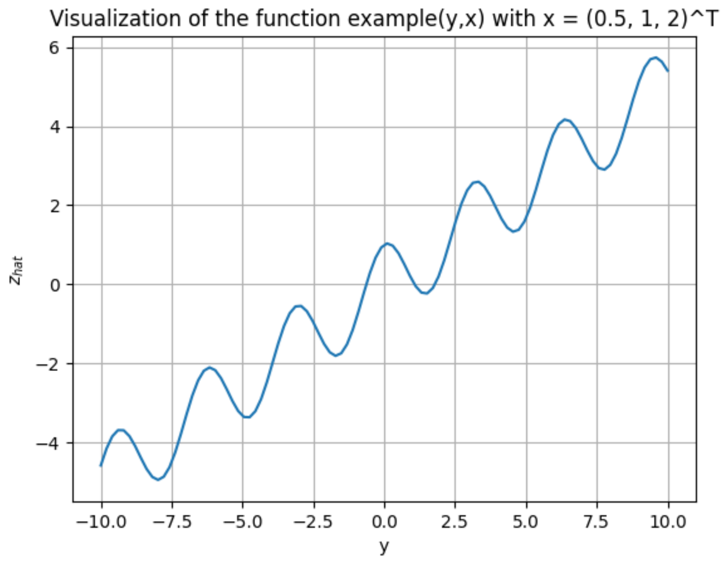


Figure 2: Plot of function (2) with $x = (0.5, 1, 2)^T$

- The first term $\|r(x_k)\|^2$ does not depend on x, so its derivative is 0.
- In the second term, the derivative is $2J(x_k)^T r(x_k)$.
- In the third term, the derivative is $2J(x_k)^T J(x_k)(x - x_k)$.

Thus, the 2 and $\frac{1}{2}$ cancels out, and the gradient $\nabla_x F(x)$ becomes:

$$\nabla_x F(x) = J(x_k)^T r(x_k) + J(x_k)^T J(x_k)(x - x_k).$$

Since we wish to minimize $F(x)$, we set the gradient $\nabla_x F(x)$ equal to zero:

$$J(x_k)^T r(x_k) + J(x_k)^T J(x_k)(x - x_k) = 0,$$

and finally subtract the first term on both sides to reach the expression for the Gauss-Newton equations, quite similar to the well-known normal equations for least squares:

$$J(x_k)^T J(x_k)(x - x_k) = -J(x_k)^T r(x_k).$$

This equation can then be solved (i.e. $(x - x_k) =: \Delta x$ computed), for instance with QR decomposition or SVD (here, however, we will use the build-in feature in TensorFlow for its solution). Just like with the standard Newton's method, we can then update our guess, and repeat the steps anew:

$$x_{k+1} = x_k + \Delta x$$

With the function relationship provided for $z(y;x)$, we get:

$$J_i = \begin{bmatrix} y_i & \cos(x_3 y_i) & -x_2 y_i \sin(x_3 y_i) \end{bmatrix}$$

for the Jacobian with $J(x) \in \mathbb{R}^{11\times 3}$.

In the **Results** section, The reader will notice that the third element of x, i.e. x_3, seems to maintain its value, regardless of the initial guess, which was surprising to us. We suspect the reason to be that the other two values x_1 and x_2 are linearly related in the function, and the algorithm will therefore simply adjust these two, almost immediately, to find a suitable approximation for the solution. We have therefore introduced a second function relationship, which has an increased nonlinear element, because of x_3 being both outside and inside the cosine function, namely:

$$\hat{z}(y;x) = yx_1 + x_2 \cos(x_3 y) + 0.5x_3.$$

As we will see, this introduced the problem of stability in a dramatic way, and we were forced to introduce a regularization term into the Gauss-Newton algorithm in order to correct for this. This done with the following procedure:

$$H_{approx,stabil} = H_{approx} + \lambda \cdot I,$$

with I being the identity matrix of the same dimensionality as H_{approx}. We suspect the potentially problematic nature of trigonometric functions (which, in the second function relationship cannot simply be compensated by the other linear factors) are to blame. In testing the condition of the two functions for the the approximation of the Hessian matrix via $J(x_k)^T J(x_k)$, we discovered a resulting large condition number for the initial x_0, given by:

$$\kappa(A) = \|A^{-1}\| \|A\|,$$

In our case:

$$\kappa(J(x_0)^T J(x_0)) = \|(J(x_0)^T J(x_0))^{-1}\| \|J(x_0)^T J(x_0)\|,$$

which we were able to minimize by increasing the value λ term, thereby stabilizing the Gauss-Newton algorithm in case for more nonlinear functions. In our case, we choose $\lambda = (20, 5, 1, 0.1, 10^{-6})$.

Results

As the reader will see in the code section, we have chosen to create the following functions:

- The function provided in the task: $z(y;x) = x_1 y + x_2 \cos(x_3 y)$
- The same function with an added term: $z(y;x) = x_1 y + x_2 \cos(x_3 y) + 0.5x_3$

- A function `calculate_condition_number` to compute the condition number
- A function `calculate_H_approx` to determine the Jacobi matrix
- The function `gauss_newton` for the Gauss-Newton algorithm

The first two functions simply returns the value for $\hat{z}$ given the (y, x) inputs. The function `calculate_condition_number` computes the condition number by using a built-in function from the numpy library.

The next function, `calculate_H_approx`, which computes the approximation for Hessian matrix, uses the TensorFlow built-in feature for automatically differentiating a function. First, make sure to convert x into a variable within the TensorFlow framework, otherwise it would through an error. We next define the variable, i.e. with respect to which variable we wish to differentiate via the GradientTape function. This is what we do, when we ask it to "watch" x with via `tape.watch(x)`. Right below, the function we wish to differentiate. With this, we can generate the Jacobian and its transpose, and approximate the Hessian matrix with $J^T J$. We then adjust the Hessian approximation by adding a regularization to it (which can be done with any desired λ), and return the approximated Hessian in numpy terms.

Next, the function for the Gauss-Newton algorithm. For the first part, we essentially repeat the steps from the function `calculate_H_approx` with addition of computing $J^T r(x)$ reshaped to be able to be multiplied with J^T. We then calculate Δx by solving the equation described above by using the built-in feature from TensorFlow. We then add Δx to x_k to get our updated x_{k+1}, add it to the history list, and increase iteration count once. Finally, it checks whether the Frobenius norm of Δx step is smaller than the given tolerance of 10^{-6}, and ends the for loop and returns the value, the recorded historiy of the steps, the iteration counts and the value of the norm, if this is the case.

This is essentially the setup of the code. We now proceed to utilize these for various inputs and computations.

We begin by calculating the condition numbers for the various regularization values $\lambda = (20, 5, 1, 0.1, 10^{-6})$ for the two functions with an initial guess of $x_0 = (0.3, 1.2, 1.9)^T$, which is relatively close to the known solution $x = (0.5, 1, 2)^T$. The results are the following:

These results point us toward an important aspect of regularization: increasing λ significantly reduces the condition number, thereby improving numerical stability. Interestingly, while the condition number for the function `example(y,x)` grows quite drastically as λ is decreased, the function `example_2(y,x)`,

Regularization Factor	**Condition Number**
20	8.5996
5	31.3985
1	152.9924
0.1	1520.9242
10^{-6}	151992416.0

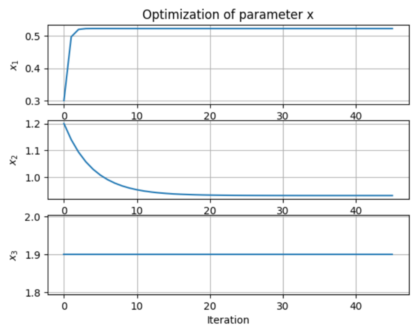
Table 1: Condition number for $J^T J$ of function `example(y,x)` based on initial guess x_0.

Regularization Factor	**Condition Number**
20	8.3793
5	27.5912
1	87.9748
0.1	178.8375
10^{-6}	202.1911

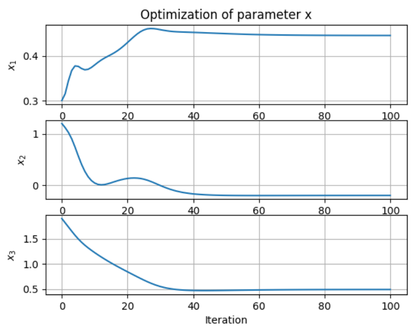
Table 2: Condition number for $J^T J$ of function `example_2(y,x)` based on initial guess x_0.

in spite of its greater nonlinearity, shows a more gradual increase. For `example(y,x)`, this seems to indicate that small λ-values implies that $J^T J$ is nearly singular. Yet, as we will see, the convergence via the Gauss-Newton method still happens quite rapidly - in fact, ever faster the smaller the λ-value. One explanation for this seeming irony, could be that the first function `example(y,x)` is closer to being a linear problem, since we are able to adjust the parameters x_1 and x_2, which are linear, leaving x_3 untouched, and still find a solution to the problem posed. This is not the case for the second function, `example_2(y,x)`, where the parameter x_3 is located both inside and outside the cosine function, given it greater nonlinearity.

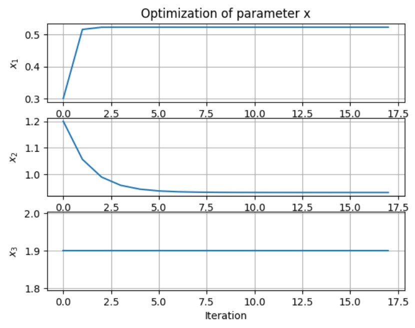
To explore this further, let us take a look at how this question is reflected in the graphs plotting the history of each step in the algorithms (Figures 3-5).



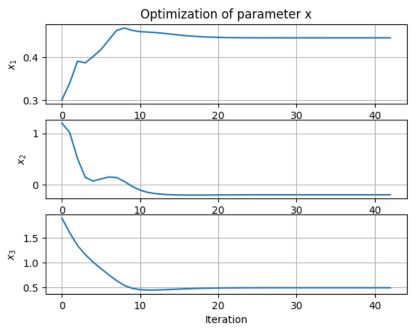
(a) $\lambda = 20$, example(y,x)



(b) $\lambda = 20$, example_2(y,x)

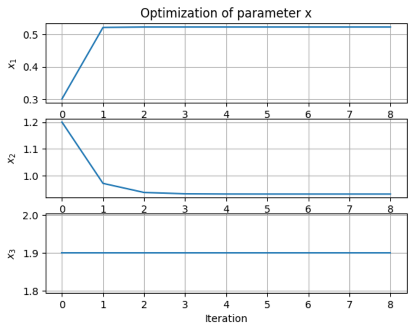


(c) $\lambda = 5$, example(y,x)

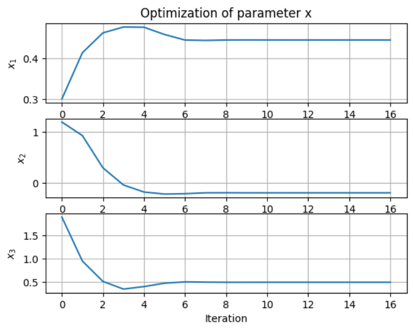


(d) $\lambda = 5$, example_2(y,x)

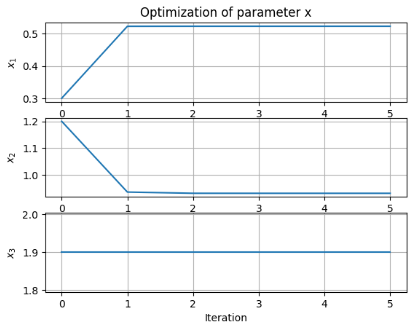
Figure 3: Images 1-4



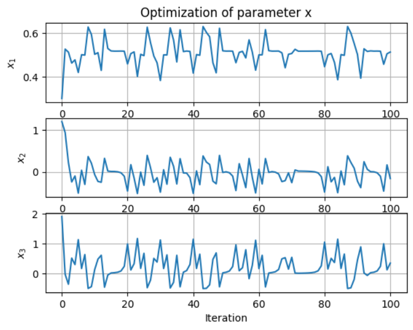
(a) $\lambda = 1$, example(y,x)



(b) $\lambda = 1$, example_2(y,x)

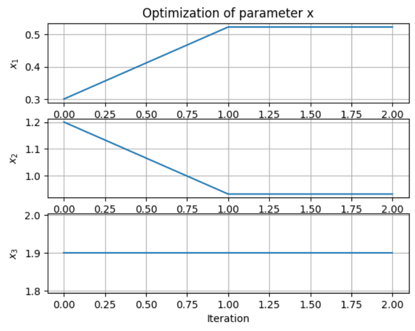


(c) $\lambda = 0.1$, example(y,x)

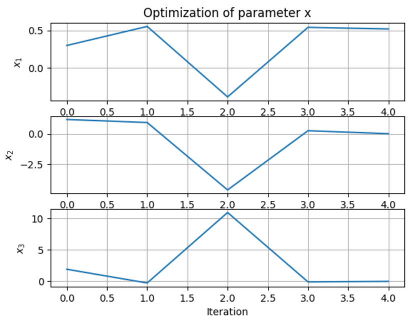


(d) $\lambda = 0.1$, example(y,x)

Figure 4: Images 5-8



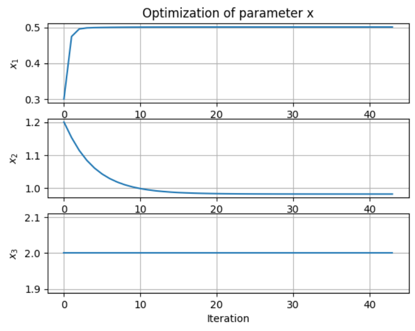
(a) $\lambda = 10^{-6}$, example(y,x)



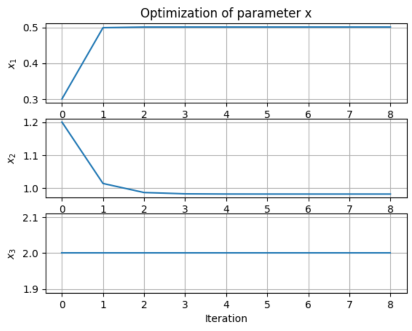
(b) $\lambda = 10^{-6}$, example_2(y,x)

Figure 5: Images 9-10

One of the first things we see is that in the case of `example(y,x)` the value x_3 doesn't seem to be altered at all, which we would explain by the fact that the two other parameters x_1 and x_2 are linear and therefore easier for the algorithm to adjust. When trying this for other initial x_0 guesses, the same result occured. For $x_3 = 2$, its value in the solution provided, it stay the same, while the other two values adjusts to $x_1 = 0.5011$ and $x_2 = 0.9824$ in all cases from $\lambda = (20, 5, 1, 0.1, 10^{-6})$ (Figure 6 shows the cases for $\lambda = 20, 1$).



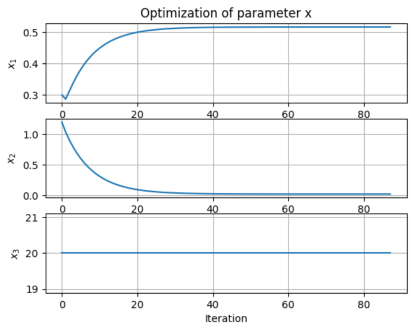
(a) $\lambda = 20$, example(y,x)



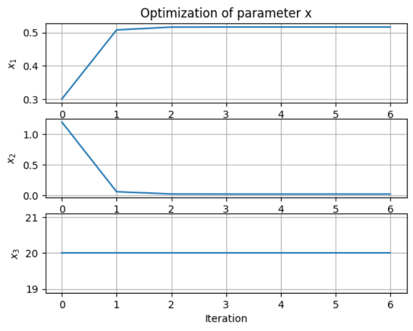
(b) $\lambda = 1$, example(y,x)

Figure 6: Images 11-12

If we set $x_3 = 20$, we still see that it keeps its value, while the other two values settle at $x_1 = 0.5164$ and $x_2 = 0.0222$, for all our chosen λ-values (see values Figure 7). Regarding the mentioned irony, the linearity of the solution might be one hypothesis that could explain why the convergence towards the solution isn't affected negatively by a dwindling regularization term, since this, as we will see, is not the case in the more linear function `example_2(y,x)`.



(a) $\lambda = 20$, example(y,x)



(b) $\lambda = 0.1$, example(y,x)

Figure 7: Images 13-14

Staying with `example(y,x)` for now, we notice that the regularization factor $\lambda \cdot I$ does not change the result. It does however seem to effect the time it takes to reach a result close enough to the desired solution. If we again pick our intital $x_0 = (0.3, 1.2, 1.9)$, we can observe for `example(y,x)` in the Figures 3-5, an increase in the rate at which a satisfactory solution is reached (error less than 10^{-6}), as the regularization factor goes down. With λ equals $20, 5, 1, 0.1, 10^{-6}$ as regularization factor, we need $45, 17, 8, 5, 2$ iterations respectively to reach below the tolerance.

This is not the case for `example_2(y,x)`, which is more nonlinear than the first. Here (`example_2(y,x)` in Figures 3-5) we observe a different pattern. For $\lambda = 20, 5, 1$, we obtain the same result and solution is reached (or would be reached in the case of $\lambda = 20$ after 138 steps, if we would have let it run beyond 100 itertations). For $\lambda = 20$, after 100 iterations, the algorithm stops with an error in the Frobenius norm of $3.42 \cdot 10^{-5}$. For $\lambda = 5$ it reaches a solution after 42 iterations, and for $\lambda = 1$ after 16 iterations. But then we observe something different: For $\lambda = 0.1$ the steps become instable, and the algorithm never able to reach a convergence towards any value, and therefore stops after 100 iterations with an error of 0.396. For $\lambda = 10^{-6}$ it simply returns ”nan” (not a number) after 100 iterations.

Conclusion

We have thus seen how regularization is essential - especially as the function relationships becomes increasingly nonlinear (as more often than not is the case for real-world events).

We have also laid the foundation for understanding the reason why the Gauss-Newton method works, and illustrated (via the standard Newton’s method) why trigonometric functions, among others, can be problematic.

The computation of the condition number has been applied to a specific case of the approximated Hessian matrix, showing its role in assessing the need for a regularization term - especially when it comes to nonlinear function relationships.

All the results mentioned in the text are provided in the code directly (with the exception that one might have to adjust the initial guess x_0 by changing a single number, specifically in x_3). We realize that the code is perhaps not the cleanest ever to been written down by humankind (admittedly some more planning could have gone into it), but we hope the reader is still able to follow the steps.

We used TensorFlow for a lot of the computations, and one is justified in asking how one knows that the Jacobian and other operations are carried out correctly. Certainly, if the TensorFlow computations, for example for the Jacobian, were wrong, people working in a very wide range of fields from machine learning, natural language processing, medical image analysis to self-driving car research would suddenly find themselves having to do a lot of overtime. Nevertheless, this doesn't diminish the validity of the question itself - on the contrary! One approach, which we unfortunately could not find the time to carry out, could be to implement an algorithm for the finite differences method and make a numerical comparison.

We hope both the theoretical foundations, as well as the numerical examples and plots, helped to give the reader new insights into the Gauss-Newton method in particular, and into the vast field of optimization methods in general.

The Python code

```python

import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# given values and an initial guess
y = np.array([0., 0.6283185, 1.2566371, 1.8849556, 2.513274,
     3.1415927, 3.7699112, 4.3982297, 5.0265482, 5.6548668,
    6.2831853])
z = np.array([0.9299887, 0.53383386, -0.15017393,
    0.11093735, 1.5128875, 2.4723399, 2.2487612, 1.3162203,
    1.6767914, 3.3423154, 4.0957375])
x_solution = np.array([0.5,1,2])
x_initial = np.array([0.3,1.2,1.9])
reg_values = [20,5,1,1e-1,1e-6]
```

```

# function
def example(y,x):
    z_hat = y*x[0] + x[1]*np.cos(x[2]*y)
    return z_hat

def example_2(y,x):
    z_hat = y*x[0] + x[1]*np.cos(x[2]*y) + 0.5*x[2]
    return z_hat

# function for calculating the condintioning number
def calculate_condition_number(H_approx):
    cond_number = np.linalg.cond(H_approx)
    return cond_number

# calculating the Jacobian via tf
def calculate_H_approx(f, x0, y, z, reg_term):
    x = tf.Variable(x0, dtype=tf.float32)
    with tf.GradientTape() as tape:
        tape.watch(x)  # w.r.t. x
        res = z - f(y,x) # residual

    J = tape.jacobian(res,x)  # compute Jacobian using
        automatic differentiation
    J_T = tf.transpose(J)  # transpose Jacobian
    H_approx = J_T @ J  # the Hessian approximation
    H_approx = H_approx + reg_term*tf.eye(tf.shape(J)[1])
    return H_approx.numpy()  # return H_approx as numpy
        array

# implementation of gauss-newton verfahren
def gauss_newton(f,x0,y,z, reg_term, max_iters=100, tol=1e
    -6):
    x = tf.Variable(x0, dtype=tf.float32) # convert x0 to a
        variable form for tf
    history = [x.numpy()] # create list to save each
        iteration in (numpy form)
    iter_count = 0
    # for loop iterating until 100 (max) or until tol
        reached (checked after each iteration):
    for _ in range(max_iters):
        with tf.GradientTape() as tape:
            tape.watch(x)  # the w.r.t. x part
            res = z - f(y,x) # residual squared; the term to
                be
            obj_fkt = 0.5*tf.reduce_sum(res**2) # the
                objective function

        J = tape.jacobian(res,x) # find Jacobian using
            automatic differentiation feature
```

```python
        J_T = tf.transpose(J)
        res_to_column = tf.reshape(res,(-1,1)) # reshape to (n,1) so the multiplication following works
        JT_res = J_T @ res_to_column
        H_approx = J_T @ J # Hessian approximation
        H_approx = H_approx + reg_term*tf.eye(tf.shape(x)[0]) # add small regularization term to ensure that an inverse of H_approx exists
        delta_x = tf.linalg.lstsq(H_approx, -JT_res, fast=True)[:,0] # least squares feature in tf

        x.assign_add(delta_x) # update x_k
        history.append(x.numpy())  # save iteration step
        iter_count += 1 # count number of iterations

        # check if we have reached the tolerance, break if it's the case
        if tf.norm(delta_x) < tol:
            break

    return x.numpy(), history, iter_count, tf.norm(delta_x) # return final x and iteration history

print(f"Condition number for J^T J of function example(y,x) based on the initial guess x_0:")
for i in reg_values:
    hessian_approx = calculate_H_approx(example,x_initial,y,z,i)
    cond_number = calculate_condition_number(hessian_approx)
    print(f"Regularization factor {i}: {cond_number}")

print()

print(f"Condition number for J^T J of function example_2(y,x) based on the initial guess x_0:")
for i in reg_values:
    hessian_approx = calculate_H_approx(example_2, x_initial, y, z, i)
    cond_number = calculate_condition_number(hessian_approx)
    print(f"Regularization factor {i}: {cond_number}")

print()

# visualize function example(y,x)
y_values = np.linspace(-10, 10, 100)
z_hat = example(y_values,x_solution)

plt.figure()
plt.plot(y_values,z_hat)
plt.xlabel('y')
plt.ylabel(r'$\hat{z}(y;x)$')
plt.title('Visualization of the function example(y,x) with x = (0.5, 1, 2)^T')
plt.grid(True)
plt.show()

for i in reg_values:
    x_final, hist_list, iter_number, delta_x = gauss_newton(example,x_initial,y,z,i)
    x_final_2, hist_list_2, iter_number_2, delta_x_2 = gauss_newton(example_2,x_initial,y,z,i)

    history_array = np.array(hist_list)  # convert to numpy array
    history_array_2 = np.array(hist_list_2)  # convert to numpy array

    # print the final x value
    print(f"\nUsing the Gauss-Newton method for example(y,x) with {i} as regularization factor and a tolerance of 1e-6, we get:\nx = ({x_final[0]:.4f}, {x_final[1]:.4f}, {x_final[2]:.4f}) after {iter_number} iterations.")
    print(f"\nUsing the Gauss-Newton method for example_2(y,x) with {i} as regularization factor and a tolerance of 1e-6, we get:\nx = ({x_final_2[0]:.4f}, {x_final_2[1]:.4f}, {x_final_2[2]:.4f}) after {iter_number_2} iterations with a Frobenius norm of {delta_x_2:.2e}.")

    # plot the history of the steps for each x-value
    plt.figure
    plt.subplot(3,1,1)
    plt.plot(history_array[:,0], label=r'$x_1$')
    plt.title(f'Optimization of parameter x')
    plt.ylabel(r'$x_1$')
    plt.grid(True)

    plt.subplot(3,1,2)
    plt.plot(history_array[:,1], label=r'$x_2$')
    plt.ylabel(r'$x_2$')
    plt.grid(True)

    plt.subplot(3,1,3)
    plt.plot(history_array[:,2], label=r'$x_3$')
    plt.ylabel(r'$x_3$')
    plt.xlabel('Iteration')
    plt.grid(True)

    plt.show()

    # plot the history of the steps for each x-value
    plt.figure
    plt.subplot(3,1,1)
    plt.plot(history_array_2[:,0], label=r'$x_1$')
    plt.title(f'Optimization of parameter x')
    plt.ylabel(r'$x_1$')
    plt.grid(True)

    plt.subplot(3,1,2)
    plt.plot(history_array_2[:,1], label=r'$x_2$')
    plt.ylabel(r'$x_2$')
    plt.grid(True)

    plt.subplot(3,1,3)
    plt.plot(history_array_2[:,2], label=r'$x_3$')
    plt.ylabel(r'$x_3$')
    plt.xlabel('Iteration')
    plt.grid(True)

    plt.show()
```