Assignment

Exercise sheet 1

Name TN 1: Tim Peinkofer

tim.peinkofer@student.uni-tuebingen.de

Name TN 2: Fabian Kostow

fabian.kostow@student.uni-tuebingen.de

Tutor: Jose Carlos Olvera Meneses

Date: 29. Oktober 2024

Inhaltsverzeichnis

1 Task 1

In the first task we are considering different methods to find the roots of an equation, we want especially discuss following equation.

$$f(x) = e^{\sqrt{5}x} - 13,5\cos(0.1x) + 25x^4$$

If we look at the Function we can guess that the roots should be in the intervals [0,1] and [-1,0], which gives us our initial guesses.

1. First we want to find the root by using linear interpolation.

```
import numpy as np

def f(x):
    return np.exp(np.sqrt(5) * x) - 13.5 * np.cos(0.1 * x
                                        ) + 25 * x**4

def linear_interpolation(f, x0, x1, epsilon, max_iter):
    for i in range(max_iter):
        f0, f1 = f(x0), f(x1)

        x2 = x1 - f1 * (x1 - x0) / (f1 - f0)

        if abs(x2 - x1) < epsilon:
            print(f(x2))
            return x2
        x0, x1 = x1, x2

root1 = linear_interpolation(f, 0, 1,1e-5,100)
root2 = linear_interpolation(f, -1, 0,1e-5,100)
print(f"Solution: {root1}")
print(f"Solution: {root2}")
```

Which gives us the following roots:

$$x_{root1} = 0.7537011684452429$$
$$x_{root2} = -0.8540824152882668$$

2. In the next step we want to use Newton's method:

```
import numpy as np

def f(x):
    return np.exp(np.sqrt(5) * x) - 13.5 * np.cos(0.1 * x
                                    ) + 25 * x**4

def df(x):
    return np.sqrt(5) * np.exp(np.sqrt(5) * x) + 1.35 *
                                    np.sin(0.1 * x) + 100
                                    * x**3

def newtons_method(f, df, x0, epsilon, max_iter):
    x = x0
    for i in range(max_iter):
        fx = f(x)
        dfx = df(x)

        if abs(fx) < epsilon:
            return x

        x = x - fx / dfx
root1 = newtons_method(f, df, 1,1e-5,100)
root2 = newtons_method(f, df, -1,1e-5,100)
print(f"Root1: {root1}")
print(f"Root2: {root2}")
```

As a result we get:

$$x_{root1} = 0.7537011684452429$$
$$x_{root2} = -0.8540824152882668$$

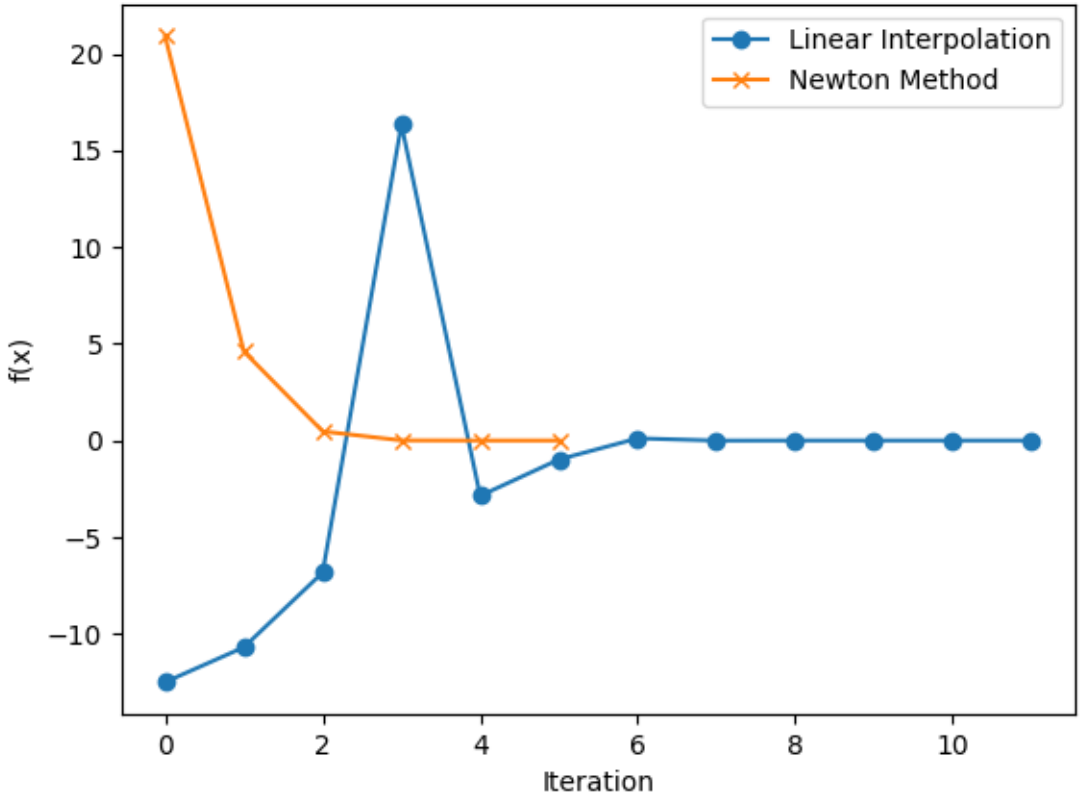
3. Examine the convergence of the two methods.

   It's not surprising to find the same roots with the different methods. It's more interesting to discuss the convergence.

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

def f(x):
    return np.exp(np.sqrt(5) * x) - 13.5 * np.cos(0.1 * x
                                    ) + 25 * x**4

def df(x):
    return np.sqrt(5) * np.exp(np.sqrt(5) * x) + 1.35 *
                                    np.sin(0.1 * x) + 100
                                    * x**3
# Linear interpolation
def linear_interpolation(f, x0, x1, epsilon, max_iter):
    ArrayLin = [x0]
    for i in range(max_iter):
        f0, f1 = f(x0), f(x1)
        x2 = x1 - f1 * (x1 - x0) / (f1 - f0)
        ArrayLin.append(x2)
        if abs(x2 - x1) < epsilon:
            return x2, ArrayLin
        x0, x1 = x1, x2
# Newton
def newtons_method(f, df, x0, epsilon, max_iter):
    ArrayNewton = [x0]
    x = x0
    for i in range(max_iter):
        fx, dfx = f(x), df(x)
        if abs(fx) < epsilon:
            return x, ArrayNewton
        x = x - fx / dfx
        ArrayNewton.append(x)

root_linear, Array_linear = linear_interpolation(f, 0, 1,
                                1e-12,100)
root_newton, Array_newton = newtons_method(f, df, 1,1e-12
                                ,100)

plt.plot(Array_linear, label='Linear Interpolation',
                                marker='o')
plt.plot(Array_newton, label="Newton Method", marker='x')
plt.plot(root_linear)
plt.xlabel('Iteration')
plt.ylabel('x')
plt.legend()
```

```
plt.show()
```

this gives us the following plot



Newton (x)	$f(x)$ (Newton)	Lin (x)	$f(x)$ (Lin)
1	20.92391278	0	-12.5
0.8271557747225875	4.6060684192	0.3739837427256477	-10.693808994
0.7622078445289572	0.474885075273	0.5857162112832605	-6.8294693391
0.7538291284951731	0.00703668901733	0.9599122010794229	16.342302032
0.753701197820581	1.6167330265e-06	0.696003853868324	-2.859410590
0.7537011684140197	8.1712414612e-14	0.73530359435016	-0.97857382052

We see that Newton's method has faster convergence. While the Newton method converges in a quadratic way. This can be seen particularly clearly in the last column (10^{-6} to 10^{-14}). The linear interpolation converges slower than the newton method. As can be seen from the values, the convergence is in the range of a power of $\approx 1.5 - 1.6$.

4. Now we want to test the methods if we set the tolerance ϵ below machine precision $(2,22 \cdot 10^{-16})$.

```
import numpy as np
#Newton
def f(x):
    return np.exp(np.sqrt(5) * x) - 13.5 * np.cos(0.1 * x
                                    ) + 25 * x**4

def df(x):
    return np.sqrt(5) * np.exp(np.sqrt(5) * x) + 1.35 *
                                    np.sin(0.1 * x) + 100
                                    * x**3

def newtons_method(f, df, x0, epsilon, max_iter):
    x = x0
    for i in range(max_iter):
        fx = f(x)
        dfx = df(x)

        if abs(fx) < epsilon:
            return x,i
        x = x - fx / dfx
    return x,max_iter
#linear Interpolation
def linear_interpolation(f, x0, x1, epsilon, max_iter):
    for i in range(max_iter):
        f0, f1 = f(x0), f(x1)

        x2 = x1 - f1 * (x1 - x0) / (f1 - f0)

        if abs(x2 - x1) < epsilon:
            print(f(x2))
            return x2,i
        x0, x1 = x1, x2
    return x2,max_iter
rootNewton1,Iteration_Newton1 = newtons_method(f, df, 100
                                ,2.22e-18,1000)
rootNewton2,Iteration_Newton2 = newtons_method(f, df, -
                                100,2.22e-18,1000)
rootLin1,Iteration_Lin1 = linear_interpolation(f, 50, 100
                                ,2.22e-18,1000)
rootLin2,Iteration_Lin2 = linear_interpolation(f, -50, -
                                100,2.22e-18,1000)
print(f"IterationNewton1: {Iteration_Newton1}")
print(f"IterationNewton2: {Iteration_Newton2}")
print(f"IterationLin1: {Iteration_Lin1}")
print(f"IterationLin2: {Iteration_Lin2}")
```

If we run the code we can see that Linear Interpolation can still manage to find solutions, but newtons method does not work (it diverges).

The last thing we want to check is how well the two methods handle different initial conditions. So we use the code above, set the tolerance above the machine accuracy and check if the methods find solutions.

Guess	IterationsNewton	IterationsLinearInterpolation
[50,100] and [-100,-50]	225 , 21	1 , 27
[1,2] and [-2,-1]	7 , 7	6 , 6
[0,1] and [-1,0]	4 , 4	8 , 12

We can see that if the guess is close to the root, Newton works faster. but if the root is not in the interval of the guess, Newton may need many more many more iterations.

2 Task 2

Next we want to find the eigenvalues of the differential equation:

$$y'' + \lambda^2 y = 0$$

with boundary conditions $y(0) = 0$ and $y'(1) = y(1)$.

We also know the general Solution of this type of equation:

$$y(x) = A\sin(\lambda x) + B\cos(\lambda x)$$

With the boundary condition $y(0) = 0$ we get:

$$A = 0$$

With the other boundary condition:

$$cos(\lambda x) = -\lambda \sin(\lambda x)$$

Which is equivalent to:

$$\cos(\lambda) + \lambda \sin(\lambda) = 0$$

In this form we need to find the root of this non-linear equation, which we do in python using Newton's method.

The following code finds the first solutions of the equation:

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

def f(x):
    return x * np.cos(x) - np.sin(x)

def df(x):
    return -x * np.sin(x)

def newtons_method(f, df, guess, epsilon=1e-5, max_iter=100):
    Lambda = guess
    for i in range(max_iter):
        f_val = f(Lambda)
        df_val = df(Lambda)
        if abs(f_val) < epsilon:
            return Lambda

        Lambda = Lambda - f_val / df_val
    return Lambda

initial_guesses = [3,6,9,12]

eigenvalues_newton = [newtons_method(f, df, guess) for guess
                                in initial_guesses]

print("Eigenvalues:")
for eigenvalues in eigenvalues_newton:
    print(eigenvalues)
```

As a result the first four possible eigenvalues are:

$$\begin{aligned}\lambda &= 2.7983860463920855\\ &= 6.1212520149114145\\ &= 9.31786646210445\\ &= 12.486454400298788\end{aligned}$$

But there are infinite many. We can get them approximately using the following equation:

$$\lambda_n \approx n\pi - \frac{\pi}{7}$$

3 Task 3

3.1 A1

In the following task we have to calculate the roots of the following system of equations:

$$f_1(x, y) = xy - 0.1$$
$$f_2(x, y) = x^2 + 3y^2 - 2$$

We will now use the newton method to get the 4 different roots:

```
import numpy as np

# Define the system of equations
def f1(x, y):
    return x*y - 0.1

def f2(x, y):
    return x**2 + 3*y**2 - 2

# Define the Jacobian matrix
def jacobian(x, y):
    return np.array([[y, x],
                     [2*x, 6*y]])

# Define the Newton-Raphson method
def newton_method(x0, y0, tol=1e-6, max_iter=100):
    x, y = x0, y0
    for _ in range(max_iter):
        # Evaluate the function values
        F = np.array([f1(x, y), f2(x, y)])

        # Check if we are close enough to the solution (Norm
                                        in R^2)
        if np.linalg.norm(F, ord=2) < tol:
            return x, y

        # Evaluate the Jacobian and solve to get the delta x
        J = jacobian(x, y)
        delta = np.linalg.solve(J, -F)

        # Update the values of x and y
        x, y = x + delta[0], y + delta[1] #update the values

    return x, y  # Return the solution after max iterations
```

```
# Find the solutions (using different initial guesses)
initial_guesses = [(0, -1), (-1, 0), (0, 1), (1, 0)]

solutions = []

for guess in initial_guesses:
    sol = newton_method(guess[0], guess[1])
    solutions.append(sol)

# Display the solutions
for i, solution in enumerate(solutions):
    print(f"Solution {i+1}: x = {solution[0]}, y = {solution[
                                        1]}")
```

As a result we get:

$$
\begin{aligned}
(x_1, y_1) &= (-0.12293961, -0.813405722) \\
(x_2, y_2) &= (-1.4088597447, -0.07097938626) \\
(x_3, y_3) &= (0.1229396103, 0.81340572230) \\
(x_4, y_4) &= (1.4088597447, 0.070979386269)
\end{aligned}
$$

The above code assumes that our Jacobian Matrix is invertible. But this is not the normal case. We can also get a Jacobian Matrix, that is not invertible. In this case our code would create an error, because of the *linalg.solve* function we use in our code.

Another problem also could be a wrong initial guess. If we for example have a function:

$$f(x, y) = \frac{y}{x}$$

We can't use $x = 0$ as an initial guess, because the function is not defined for this value. The result will be a NaN. We also have to adjust our initial guesses, so that they are not to far away from our root. This asjusting could solve diverging problems. In addition, we must note that different initial conditions can converge to different roots!

We can fix this by adjusting the code as follows:

```
import numpy as np

# Define the system of equations
def f1(x, y):
    return x*y - 0.1

def f2(x, y):
    return x**2 + 3*y**2 - 2

# Define the Jacobian matrix
def jacobian(x, y):
    return np.array([[y, x],
                     [2*x, 6*y]])

# Define the Newton-Raphson method
def newton_method(x0, y0, tol=1e-6, max_iter=100):
    x, y = x0, y0
    for _ in range(max_iter): #ends the calculating if the
                              function values are
                              diverging
        # Evaluate the function values
        try:
            F = np.array([f1(x, y), f2(x, y)])
        except ZeroDivisionError:
            return None, None  # Exit if division by zero

        # Check for NaN in function evaluations
        if np.isnan(F).any():
            return None, None

        # Check if we are close enough to the solution
        if np.linalg.norm(F, ord=2) < tol:
            return x, y

        # Evaluate the Jacobian and solve for the delta
        J = jacobian(x, y)
        try: # check if the matrix is invertible
            delta = np.linalg.solve(J, -F)
        except np.linalg.LinAlgError:
            return None

        # Update the values of x and y
        x, y = x + delta[0], y + delta[1]

    return x, y  # Return the solution after max iterations
```

```
# Find the solutions (using different initial guesses)
initial_guesses = [(0, -1), (-1, 0), (0, 1), (1, 0)]
solutions = []

for guess in initial_guesses:
    sol = newton_method(guess[0], guess[1])
    if sol[0] is not None and sol[1] is not None:
        solutions.append(sol)

# Display the solutions
for i, solution in enumerate(solutions):
    if solution is None or solution[0] is None or solution[1]
                                        is None:
        print(f"Solution {i+1}: No valid solution found.")
    else:
        print(f"Solution {i+1}: x = {solution[0]}, y = {
                                           solution[1]}")
```

This will fix most of the errors.

3.2 A2

In the last step we have to refomulate our equations and add a function for the improved $x = g(x)$ method. Our equations will be:

$$x_1 = \frac{0.1}{y}$$

$$y_1 = \pm\sqrt{\frac{1}{3} \cdot (2 - x^2)}$$

$$x_2 = \pm\sqrt{2 - 3y^2}$$

$$y_2 = \frac{0.1}{x}$$

Now we can calculate the roots of our systems of equations:

```
import numpy as np

# Definition of the functions
def func(x, y):
    return x * y - 0.1

def g_y_1(x):
    return np.sqrt(1/3*(2 - x**2))

def g_x_2(y):
    return np.sqrt(2 - 3*y**2)

def g_x_1(y):
    return 0.1/y

def g_y_2(x):
    return 0.1 / x

def negative_g_x_2(y):
    return -g_x_2(y)

def negative_g_y_1(x):
    return -g_y_1(x)
```

```
def solve_fixed_point(f1, f2, x_init, y_init, tol=1e-6,
                                    max_iter=100):
    # Initial guesses for x and y
    x, y = x_init, y_init
    for i in range(max_iter):
        # Update x and y using the fixed-point iterations
        x_new = f1(y)
        y_new = f2(x)

        # Check for convergence
        if abs(x_new - x) < tol and abs(y_new - y) < tol:
            print(f"Converged in {i+1} iterations.")
            return x_new, y_new

        x, y = x_new, y_new

    print("Did not converge.")
    return None, None

# Run the function with different initial guesses
initial_guesses = [(1.5, 0.1), (0.1, 1), (2,1)]
solutions = []
for x_init, y_init in initial_guesses:
    x_sol, y_sol = solve_fixed_point(g_x_1, g_y_1,x_init,
                                     y_init)
    if x_sol is not None and y_sol is not None:
        solutions.append((x_sol, y_sol,func(x_sol,y_sol),
                                        x_init,y_init))

for x_init, y_init in initial_guesses:
    x_sol, y_sol = solve_fixed_point(g_x_2, g_y_2,x_init,
                                     y_init)
    if x_sol is not None and y_sol is not None:
        solutions.append((x_sol, y_sol, func(x_sol,y_sol),
                                        x_init,y_init))


for x_init, y_init in initial_guesses:
    x_sol, y_sol = solve_fixed_point(g_x_1,negative_g_y_1,
                                     x_init, y_init)
    if x_sol is not None and y_sol is not None:
        solutions.append((x_sol, y_sol,func(x_sol,y_sol),
                                        x_init,y_init))
```

```
for x_init, y_init in initial_guesses:
    x_sol, y_sol = solve_fixed_point(negative_g_x_2, g_y_2,
                                     x_init, y_init)
    if x_sol is not None and y_sol is not None:
        solutions.append((x_sol, y_sol, func(x_sol,y_sol),
                                        x_init,y_init))

# Display solutions
for i, (x_sol, y_sol, per,x0,y0) in enumerate(solutions, 1):
    print(f"Solution {i}: x     {x_sol}, y     {y_sol},
                                  Performance = {per}, x0 =
                                  {x0}, y0 = {y0}")
```

The results are:

Solution	**x**	**y**	**Performance**	$\mathbf{x_0}$	$\mathbf{y_0}$
1	0.122 940	0.813 406	-7.39×10^{-9}	0.1	1.0
2	1.408 860	0.070 979	2.45×10^{-8}	1.5	0.1
3	−0.122 940	−0.813 406	-7.39×10^{-9}	0.1	1.0
4	−1.408 860	−0.070 979	2.45×10^{-8}	1.5	0.1

Tabelle 1: Solutions with initial guesses and performance values.

As we see the relative performance of the root is very good.

We also tested (2,1) as an initial value. The result was diverging function values, since the initial condition is not very close to the roots.

But what are the requirements for convergence?

- Good initial guesses (near to root and no Value Errors like Zero Division Errors)
- Choosing a good way to reformulate the equation into $x = g(x)$
- Trying different initial values to check if there are multiply roots, because a initial value which is near both roots will generate converging problems.