Multidimensional Newton's Method

MATH 310 Project 1

Anny

February 2024

1 Methodology and Implementation

Consider a system of nonlinear equations given by the function

$$F : \mathbb{R}^n \to \mathbb{R}^n$$

We need to find a solution of $x \in \mathbb{R}^n$ such that $F(x) = \mathbf{0}$.

For the two-dimensional case with 2 functions, the function F and its system is given below:

$$F(x, y) = \begin{pmatrix} f(x, y) \\ g(x, y) \end{pmatrix}$$

The Jacobian matrix J of F is a square matrix of all first-order derivatives of components of F, shown below:

$$J(x, y) = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{pmatrix}$$

The Newton iteration in multidimensional with the Jacobian matrix is given by

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} x_n \\ y_n \end{pmatrix} - J^{-1}(x_n, y_n)F(x_n, y_n)$$

where x and y are approximations.

1.1 Example 1

$$F(x, y) = \begin{pmatrix} xsin(y) \\ cos(x) + sin(y^2) \end{pmatrix}$$

$$J(x, y) = \begin{pmatrix} sin(y) & xcos(y) \\ -sin(x) & 2ycos(y^2) \end{pmatrix}$$

In Python, we need to firstly define the functions of F and their corresponding Jacobian matrix.

```python
# Define the function F(x, y) = [f(x, y), g(x, y)]
def F(x, y):
    return np.array([x * np.sin(y), np.cos(x) + np.sin(y**2)])

# Define the Jacobian matrix of F
def J(x, y):
    return np.array([[np.sin(y), x * np.cos(y)],
                     [-np.sin(x), 2 * y * np.cos(y**2)]])
```

For the implementation of the Multidimentional Newton's method, we need to perform iterations for solving for the updated step $\mathbf{b}$. Instead of calculating the inverse of the Jacobian, we can solve the system:

$$J(x_n, y_n)\mathbf{b} = -F(x_n, y_n)$$

```python
# Multidimensional Newton's method for 2 functions
def newtons_method(F, J, x0, y0, tol):
    x, y = x0, y0

    # initialize the step size vector b with infinity to enter the while loop
    b = np.array([np.inf, np.inf])

    # iterating until b is smaller than the tolerance
    while np.linalg.norm(b)>=tol:

        # solve the linear system J(x, y) * b = -F(x, y)
        b = np.linalg.solve(J(x,y), -F(x,y))

        # update the guesses for x and y
        x, y = x + b[0], y + b[1]

    return x, y
```

The above function can be applied to solve for the example system with the initial guess of $x_0, y_0) = (\frac{\pi}{2}, \pi)$:

```python
# Initial guess
x0, y0 = np.pi/2, np.pi

# Apply Newton's method
solution = newtons_method(F, J, x0, y0, 1e-6)

# Print the solution
print("Solution: x = {:}, y = {:}".format(solution[0], solution[1]))
```

The returned solution is Solution: Solution: x = 1.1259698864749177, y = 3.141592653589793.

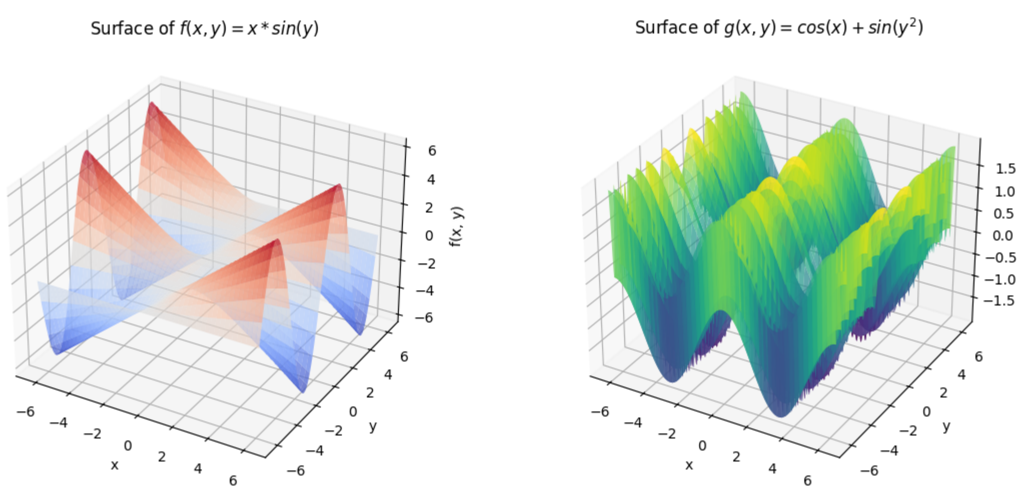

Figure 1: C.13 Example

Actually, the given example functions have multiple roots as shown in figure 1. and the returned solution is highly depended on the initial guess. If we try to solve with the initial guess of $(x_1, y_1) = (1.0, 1.0)$:

```
# Initial guess
x1, y1 = 1.0, 1.0

# Apply Newton's method
solution1 = newtons_method(F, J, x1, y1, 1e-6)

# Print the solution
print("Solution: x = {:}, y = {:}".format(solution1[0], solution1[1]))
```

The solution is changed to be Solution: x = 1.5707963267948966, y = -6.64410208813345e-25.

The Multidimensional Newton's Method can converge to different solutions based on our initial guess. We also need to ensure that the given functions are differentiable and the Jacobian matrix is non-singular at the beginning and is invertible at each iteration, otherwise, the method might fail to converge.

1.2 Example 2

The new functions are given below:

$$F(x,y) = \begin{pmatrix} 1 + x^2 - y^2 + e^x cos(y) \\ 2xy + e^x sin(y) \end{pmatrix}$$

$$J(x,y) = \begin{pmatrix} 2x + e^x cos(y) & -2y - e^x sin(y) \\ 2y + e^x sin(y) & 2x + e^x cos(y) \end{pmatrix}$$

```
# Define the function F(x, y) = [f(x, y), g(x, y)]
def F(x, y):
    return np.array([1 + x**2 - y**2 + np.exp(x)*np.cos(y), 2*x*y + np.exp
   (x)*np.sin(y)])

# Define the Jacobian matrix of F
def J(x, y):
    return np.array([[2*x + np.exp(x)*np.cos(y), -2*y - np.exp(x)*np.sin(y
   )],
                     [2*y + np.exp(x)*np.sin(y), 2*x + np.exp(x)*np.cos(y)
   ]])

# Initial guess
x0, y0 = 1, 1

# Apply Newton's method
solution = newtons_method(F, J, x0, y0, 1e-6)

# Print the solution
print("Solution: x = {:}, y = {:}".format(solution[0], solution[1]))
```

Then solution with initial guess of $(x, y) = (1, 1)$ is Solution: x = -0.2931626870672417, y = 1.1726598176735787.

Think about a case of more functions:

$$F(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_k(x) \end{pmatrix}$$

Then, the Jacobian matrix will be:

$$J(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x) & \frac{\partial f_1}{\partial x_2}(x) & \cdots & \frac{\partial f_1}{\partial x_n}(x) \\ \frac{\partial f_2}{\partial x_1}(x) & \frac{\partial f_2}{\partial x_2}(x) & \cdots & \frac{\partial f_2}{\partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_k}{\partial x_1}(x) & \frac{\partial f_k}{\partial x_2}(x) & \cdots & \frac{\partial f_k}{\partial x_n}(x) \end{pmatrix}$$

Thus, the Multidimentional Newton's Method solves for

$$x_{n+1} = x_n - J(x_n)^{-1}F(x_n)$$

where x represents the approximations in a vector

In Python, the general function is shown below:

```python
# General Multidimensional Newton's Method
# Can implement with any number of given functions

def multidimensional_newton(funcs, jacobian, initial_guess, tol=1e-10):
# Initialize the guess
    x_k = np.array(initial_guess)

    while True:
        # Evaluate the functions
        F_k = np.array([func(*x_k) for func in funcs])

        # Evaluate the Jacobian matrix
        J_k = jacobian(*x_k)

        # Check if the Jacobian is near singular
        if np.linalg.cond(J_k) > 1 / np.finfo(float).eps:
            print("Jacobian is near singular at the initial guess. Newton's method may not converge.")
            return None

        # Solve the system J_k * b = -F to find delta
        b, residuals, rank, s = np.linalg.lstsq(J_k, -F_k, rcond=None)

        # If the system is underdetermined or overdetermined, no solution was found
        if residuals.size > 0 and np.any(residuals > tol):
            print("The system does not have a solution.")
            return None

```

```
        if rank < J_k.shape[0]:
            print("Jacobian matrix is rank deficient at the initial guess,
    and the Newton's method may not converge.")

        # Update the guess
        x_k += b

        # Check for convergence
        if np.linalg.norm(b) < tol:
            break

    return x_k
```

Apply the functions in C.15 with an initial guess of [1.0, 1.0]:

```
# C. 15 example, where the number of functions is 2
def f1(x, y):
    return 1 + x**2 - y**2 + np.exp(x)*np.cos(y)

def g1(x, y):
    return 2*x*y + np.exp(x)*np.sin(y)

F = [f1, g1]
initial = [1.0, 1.0]
sol = multidimensional_newton(F, J, initial)
print("Solution:", sol)
```

The solution is given as: Solution: [-0.29316269 1.17265982].

1.3 Example 3

The given functions are:

$$F(x) = \begin{pmatrix} x^2 + y^2 + z^2 - 100 \\ xyz - 1 \\ x - y - sin(z) \end{pmatrix}$$

Then, the Jacobian matrix is shown below:

$$J(x) = \begin{pmatrix} 2x & 2y & 2z \\ yz & xz & xy \\ 1 & -1 & -cos(z) \end{pmatrix}$$

With the initial guess of $(x, y, z) = (1.0, 1.0, \pi)$, we can implement the multidimentional newton's method:

```
def f(x, y, z):
    return x**2 + y**2 + z**2 -100

def g(x, y, z):
    return x*y*z - 1

def h(x, y, z):
    return x - y - np.sin(z)

```

```
def Jacobian(x, y, z):
    return np.array([[2*x, 2*y, 2*z],
                     [y*z, x*z, x*y],
                     [1, -1, -np.cos(z)]])

F_new = [f, g, h]
initial = [1.0, 1.0, np.pi]

# Apply the method which allows any number of functions
sol = multidimensional_newton(F_new, Jacobian, initial)
print(f"Solution: {sol}")
```

The solution is given as [-7.06104719 -7.08104601 0.02000016].

1.4 Example 4

The system of differential equations is:

$$x' = \alpha x - \beta xy$$
$$y' = \delta y + \gamma xy$$

where $\alpha = 1$, $\beta = 0.05$, $\gamma = 0.01$ and $\delta = 1$

The Jacobian matrix is defined:

$$J(x, y) = \begin{pmatrix} \alpha - \beta y & -\beta x \\ \gamma y & \delta + \gamma x \end{pmatrix}$$

To find the equilibrium points, we set x' and y' to zero and solve for x and y. This gives the following system of algebraic equations:

$$\begin{cases} 0 = x - 0.05xy \\ 0 = y + 0.01xy \end{cases}$$

From this system of equations, the first equation indicates that either $x = 0$ or $y = \frac{\alpha}{\beta} = 20$. The second equation indicates that either $y = 0$ or $x = -\frac{\delta}{\gamma} = -100$.

Therefore, our potential initial guesses for equilibrium points can be:

$$\begin{cases} (x, y) = (0, 0) \\ (x, y) = (0, 20) \\ (x, y) = (-100, 0) \\ (x, y) = (nonzero, nonzero) \end{cases}$$

To solve this system, we can use the multidimensional Newton's method with an initial guesses of [0.0, 0.0], [0.0, 20.0], [-100.0, 0], $[-\frac{\delta}{\gamma + (\frac{\gamma\alpha}{\beta})}, \frac{\alpha}{\beta}]$.

```
# Parameters for the system
alpha, beta, gamma, delta = 1, 0.05, 0.01, 1

```

```python
# Defining the functions for the system
def f1(x, y):
    return alpha * x - beta * x * y

def f2(x, y):
    return delta * y + gamma * x * y

# Defining the Jacobian matrix for the system
def jacobian(x, y):
    return np.array([[alpha - beta * y, -beta * x],
                     [gamma * y, delta + gamma * x]])

F = [f1, f2]
initial0 = [0.0, 0.0]
initial1 = [0.0, alpha/beta]
initial2 = [-delta/gamma, 0.0]

# substitute y = alpha / beta in the second equation
initial3 = [-delta / (gamma + (gamma * alpha / beta)), alpha/beta]
sol0 = multidimensional_newton(F, jacobian, initial0)
sol1 = multidimensional_newton(F, jacobian, initial1)
sol2 = multidimensional_newton(F, jacobian, initial2)
sol3 = multidimensional_newton(F, jacobian, initial3)

print("Solution:", sol0)
print("Solution:", sol1)
print("Solution:", sol2)
print("Solution:", sol3)
```

The output is:

```
Jacobian is singular at the initial guess.
Newton's method may not converge with the initial guess of [0.0, 20.0].

Jacobian is singular at the initial guess.
Newton's method may not converge with the initial guess of [-100.0, 0.0].

Solution: [0. 0.] with initial guess of [0.0, 0.0]
Solution: None with initial guess of [0.0, 20.0]
Solution: None with initial guess of [-100.0, 0.0]
Solution: [-100.   20.] with initial guess of [-4.761904761904762, 20.0]
```

Therefore, we have 2 equilibrium points as solutions: [0, 0] and [-100, 20].

1.5 Example 5

The system of differential equations is defined as:

$$x' = -0.1xy - x$$
$$y' = -x + 0.9y$$
$$z' = cos(y) - xz$$

The Jacobian matrix is defined as:

$$J(x, y, z) = \begin{pmatrix} -0.1y - 1 & -0.1x & 0 \\ -1 & 0.9 & 0 \\ -z & -sin(y) & -x \end{pmatrix}$$

Set x', y', z' to zero and solve for x, y, z. The following shows the algebraic equations:

$$\begin{cases} 0 = -0.1xy - x \\ 0 = -x + 0.9y \\ 0 = cos(y) - xz \end{cases}$$

From the firs equation, either $x = 0$ or $y = -10$ since $x(-0.1y - 1) = 0$.
From the second equation, either both x, y are zero or nonzero, yielding $y = \frac{x}{0.9}$.
For the third equation, if x is zero, z can be any value and $cos(y)$ needs to be zero. If $x \neq 0$, we need to define $z = \frac{cos(y)}{x}$

After applying multiple initial guesses, most of them give a singular Jacobian matrix, which cannot be solved throught the multidimensional newton's method, unless applying the initial guess of $(x', y', z') = (1.0, -10.0, 1.0)$.

```python
def f20(x, y, z):
    return -0.1*x*y - x

def g20(x, y, z):
    return -x + 0.9 * y

def h20(x, y, z):
    return np.cos(y) - x*z

def Jaco(x, y ,z):
    return np.array([[-0.1*y - 1, -0.1*x, 0],
                     [-1, 0.9, 0],
                     [-z, -np.sin(y), -x]])


F = [f20, g20, h20]
initial0 = [1.0, -10.0, 1.0]
sol0 = multidimensional_newton(F, Jaco, initial0)
print(sol0)
```

The solution is approximated as [-9. -10. 0.09323017].

2 General Function in Python

```python
import numpy as np

# General Multidimensional Newton's Method
def multidimensional_newton(funcs, jacobian, initial_guess, tol=1e-10):
    """
    funcs: list of functions, the system of nonlinear equations
    jacobian: function that computes the Jacobian matrix
    initial_guess: initial guess for the variables
    tolerance: tolerance for the convergence criterion
    """
    # Initialize the guess
    x_k = np.array(initial_guess)

    while True:
        # Evaluate the functions
        F_k = np.array([func(*x_k) for func in funcs])

        # Evaluate the Jacobian matrix
        J_k = jacobian(*x_k)

        # Check if the Jacobian is near singular
        if np.linalg.cond(J_k) > 1 / np.finfo(float).eps:
            # If the Jacobian is near singular, Newton's method may not converge
            print(f"Jacobian is singular at the initial guess. Newton's method may not converge with the initial guess of {initial_guess}.")
            return None

        # Solve the system J_k * b = -F to find delta
        b = np.linalg.solve(J_k, -F_k)

        # Update the guess
        x_k += b

        # Check for convergence
        if np.linalg.norm(b) < tol:
            break

    return x_k
```

3 Example solution in Summary

```python
# C. 13 example
def f(x, y):
    return x * np.sin(y)
def g(x, y):
    return np.cos(x) + np.sin(y**2)
def jacobian(x, y):
    return np.array([[np.sin(y), x * np.cos(y)],
```

```
                     [-np.sin(x), 2 * y * np.cos(y**2)]])
funcs = [f, g]
initial = [np.pi/2, np.pi]
sol = multidimensional_newton(funcs, jacobian, initial)
print(f"C.13 example: {sol} with initial guess of {initial}")
######################################################################

# C. 15 example
def f1(x, y):
    return 1 + x**2 - y**2 + np.exp(x)*np.cos(y)
def g1(x, y):
    return 2*x*y + np.exp(x)*np.sin(y)
# Define the Jacobian matrix of F
def jacobian1(x, y):
    return np.array([[2*x + np.exp(x)*np.cos(y), -2*y - np.exp(x)*np.sin(y)],
                     [2*y + np.exp(x)*np.sin(y), 2*x + np.exp(x)*np.cos(y)]])
funcs1 = [f1, g1]
initial1 = [1.0, 1.0]
sol1 = multidimensional_newton(funcs1, jacobian1, initial1)
print(f"C.15 example: {sol1} with initial guess of {initial1}")
####################################################################

# C.17 example
def f(x, y, z):
    return x**2 + y**2 + z**2 -100
def g(x, y, z):
    return x*y*z - 1
def h(x, y, z):
    return x - y - np.sin(z)
def Jacobian(x, y, z):
    return np.array([[2*x, 2*y, 2*z],
                     [y*z, x*z, x*y],
                     [1, -1, -np.cos(z)]])
F_new = [f, g, h]
initial = [1.0, 1.0, np.pi]
sol = multidimensional_newton(F_new, Jacobian, initial)
print(f"C.17 example: {sol} with initial guess of {initial}")
######################################################################

#C.19 example
alpha, beta, gamma, delta = 1, 0.05, 0.01, 1
def f1(x, y):
    return alpha * x - beta * x * y
def f2(x, y):
    return delta * y + gamma * x * y
def jacobian(x, y):
    return np.array([[alpha - beta * y, -beta * x],
                     [gamma * y, delta + gamma * x]])

F = [f1, f2]
initial0 = [0.0, 0.0]
initial1 = [0.0, alpha/beta]
```

```
initial2 = [-delta/gamma, 0.0]

# substitute y = alpha / beta in the second equation
initial3 = [-delta / (gamma + (gamma * alpha / beta)), alpha/beta]

print("\nC.19 example:")
sol0 = multidimensional_newton(F, jacobian, initial0)
sol1 = multidimensional_newton(F, jacobian, initial1)
sol2 = multidimensional_newton(F, jacobian, initial2)
sol3 = multidimensional_newton(F, jacobian, initial3)

print("Solution:", sol0, "with initial guess of", initial0)
print("Solution:", sol1, "with initial guess of", initial1)
print("Solution:", sol2, "with initial guess of", initial2)
print("Solution:", sol3, "with initial guess of", initial3)
######################################################################

# C.20 example
def f20(x, y, z):
    return -0.1*x*y - x
def g20(x, y, z):
    return -x + 0.9 * y
def h20(x, y, z):
    return np.cos(y) - x*z

def Jaco(x, y ,z):
    return np.array([[-0.1*y - 1, -0.1*x, 0],
                     [-1, 0.9, 0],
                     [-z, -np.sin(y), -x]])
F = [f20, g20, h20]
initial0 = [1.0, -10.0, 1.0]
sol0 = multidimensional_newton(F, Jaco, initial0)
print(f"\nC.20 example: {sol0} with initial guess of {initial0}")

#####################################################################
The outputs are:
C.13 example: [1.12596989 3.14159265] with initial guess of [1.5707963267948966, 3.141592653589793]
C.15 example: [-0.29316269  1.17265982] with initial guess of [1.0, 1.0]
C.17 example: [-7.06104719 -7.08104601  0.02000016] with initial guess of [1.0, 1.0, 3.141592653589793]

C.19 example:
Jacobian is singular at the initial guess. Newton's method may not converge with the initial guess of [0.0, 20.0].
Jacobian is singular at the initial guess. Newton's method may not converge with the initial guess of [-100.0, 0.0].
Solution: [0. 0.] with initial guess of [0.0, 0.0]
Solution: None with initial guess of [0.0, 20.0]
Solution: None with initial guess of [-100.0, 0.0]
Solution: [-100.   20.] with initial guess of [-4.761904761904762, 20.0]

C.20 example: [ -9.          -10.            0.09323017] with initial guess of [1.0, -10.0, 1.0]
```