Chapter 1 : Newtom Method

Newton Method

Consider a vector valued function $\mathbf{F} : \mathbb{R} \to \mathbb{R}$ such that

$$\mathbf{F}(x) = 0 \tag{1}$$

where $x \in \mathbb{R}$. Therefor,

$$\mathbf{F}(x_1, x_2, x_3, \dots) = \begin{matrix}[f_1(x_1, \dots, x_n) \\ \dots f_n(x_1, \dots, x_n)]\end{matrix} \tag{2}$$

We needed to solve the **Eq-1** which represents a system of equations, for sake of generality we consider that system may be nonlinear.

Newton's method for solving systems of nonlinear equations is an extension of the Newton-Raphson method for scalar equations, applied to vector-valued functions. It's used to find the roots (or zeros) of a system of nonlinear equations, where the system can be expressed as:

$$F(\mathbf{x}) = \mathbf{0}$$

where $F(\mathbf{x})$ is a vector of nonlinear functions $F_1(x_1, x_2, \dots, x_n), F_2(x_1, x_2, \dots, x_n), \dots, F_n(x_1, x_2, \dots, x_n)$ that map from $\mathbb{R}^n$ to $\mathbb{R}^n$.

Steps for Newton's Method

The steps for applying Newton's method to solve a system of nonlinear equations are as follows:

1. **Initial Guess**: Start with an initial guess $\mathbf{x}^{(0)}$.

2. **Update Formula**: Update the guess iteratively using the formula:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left[\mathbf{J}(\mathbf{x}^{(k)})\right]^{-1} F(\mathbf{x}^{(k)})$$

   where:

   - $\mathbf{x}^{(k)}$ is the current approximation of the root.
   - $\mathbf{J}(\mathbf{x}^{(k)})$ is the Jacobian matrix of $F(\mathbf{x})$ evaluated at $\mathbf{x}^{(k)}$.
   - $F(\mathbf{x}^{(k)})$ is the vector of function values at $\mathbf{x}^{(k)}$.
   - $\left[\mathbf{J}(\mathbf{x}^{(k)})\right]^{-1}$ is the inverse of the Jacobian matrix.

3. **Convergence**: Iterate until the change between consecutive guesses is sufficiently small (i.e., $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|$ is less than a tolerance).

Jacobian Matrix

For a system of n equations in n unknowns, the Jacobian matrix $\mathbf{J}(\mathbf{x})$ is the $n \times n$ matrix of partial derivatives:

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \frac{\partial F_n}{\partial x_2} & \cdots & \frac{\partial F_n}{\partial x_n} \end{pmatrix}$$

Example

Consider the system of equations:

$$\begin{cases} x_1^2 + x_2^2 = 1 \\ x_1^2 - x_2 = 0 \end{cases}$$

The system can be written as $F(\mathbf{x}) = \mathbf{0}$, where $F_1(x_1, x_2) = x_1^2 + x_2^2 - 1$ and $F_2(x_1, x_2) = x_1^2 - x_2$.

Jacobian Matrix

The Jacobian matrix is:

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} 2x_1 & 2x_2 \\ 2x_1 & -1 \end{pmatrix}$$

Newton's Update

The update formula for Newton's method is:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left[\mathbf{J}(\mathbf{x}^{(k)})\right]^{-1} F(\mathbf{x}^{(k)})$$

You would compute the Jacobian at each iteration and solve for $\mathbf{x}^{(k+1)}$.

Considerations

- **Convergence**: Newton's method may not always converge, especially if the initial guess is far from the true solution or if the Jacobian is singular or nearly singular.
- **Choosing Initial Guess**: The choice of initial guess $\mathbf{x}^{(0)}$ is critical. A good starting point can improve the likelihood of convergence.

Newton's Method for Solving Nonlinear Systems

Initialization: Let $\mathbf{x}^{(0)} = \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \\ \vdots \\ x_n^{(0)} \end{bmatrix}$ be a given initial vector.

Jacobian Matrix and Function Vector: Compute the Jacobian matrix $J(\mathbf{x}^{(0)})$ and the function vector $\mathbf{F}(\mathbf{x}^{(0)})$.

Linear System: Solve for $\mathbf{y}^{(0)}$ from:

$$J(\mathbf{x}^{(0)})\mathbf{y}^{(0)} = -\mathbf{F}(\mathbf{x}^{(0)}).$$

Update: Update the solution:

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \mathbf{y}^{(0)}.$$

for $k = 1, 2, \ldots$ **do**
  $\mathbf{y}^{(k-1)} = -J(\mathbf{x}^{(k-1)})^{-1}\mathbf{F}(\mathbf{x}^{(k-1)})$
  $\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + \mathbf{y}^{(k-1)}$
  if $\|\mathbf{F}(\mathbf{x}^{(k)})\|$ (or the norm of the update $\|\mathbf{y}^{(k-1)}\|$) is below a predefined tolerance ϵ **then**
    break
  end if
end for

Algorithm 1: Newton's Method for Solving Nonlinear Systems
1. Input Functions: $F(x)$ represents the vector-valued function $\mathbf{F}(\mathbf{x})$.
2. Initialization: Start with an initial guess $\mathbf{x}^{(0)}$.
3. Linear Solver: Solve the system $J(\mathbf{x})\mathbf{y} = -\mathbf{F}(\mathbf{x})$ using 'np.linalg.solve'.
4. Update: Compute the next iteration $\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + \mathbf{y}^{(k-1)}$.
5. Convergence: Check if the norm of $\mathbf{y}$ is less than the tolerance ϵ.

Implementations of above algorithms

Listing 1: Python code for solving a nonlinear system using Newton's method

```python
import numpy as np

def NewtonMethodforSys(F, J, x0, tol=1e-6, max_iter=100):
    """
    Newton's Method for solving nonlinear systems.

    Parameters:
        F: callable
            Function vector F(x), where x is an n-dimensional array.
        J: callable
            Jacobian matrix J(x), where x is an n-dimensional array.
        x0: ndarray
            Initial guess for the solution.
        tol: float
            Tolerance for convergence.
        max_iter: int
            Maximum number of iterations.

    Returns:
        x: ndarray
            Approximation to the root of F(x) = 0.
        num_iter: int
            Number of iterations performed.
    """
    x = np.array(x0, dtype=float)
    for k in range(max_iter):
        Fx = F(x)  # Evaluate F(x)
        Jx = J(x)  # Evaluate J(x)

        # Solve J(x) * y = -F(x) for y using numpy's linear solver
        try:
            y = np.linalg.solve(Jx, -Fx)
        except np.linalg.LinAlgError:
            raise ValueError("Jacobian is singular at iteration {}.".format(k))

        # Update x
        x = x + y

        # Check for convergence
        if np.linalg.norm(y, ord=2) < tol:
            return x, k + 1  # Return the solution and iterations

    raise ValueError("Newton's method did not converge within the maximum iterations.")
```

Listing 2: Python code for solving a nonlinear system using Newton's method

```python
import matplotlib.pyplot as plt

def plot_solution(F, solution, original_guess):
    """
    Plot the original guess vs the approximate solution.

    Parameters:
        F: callable
            Function vector F(x), where x is an n-dimensional array
                .
        solution: ndarray
            Approximate solution to the system of equations.
        original_guess: ndarray
            Initial guess for the solution.
    """
    x_labels = [f"x{i+1}" for i in range(len(solution))]  # Labels
        for variables
    width = 0.35  # Bar width for plotting

    # Prepare the data
    original_values = original_guess
    approx_values = solution

    # Plot original vs approximate values
    x = np.arange(len(solution))
    fig, ax = plt.subplots(figsize=(8, 5))
    ax.bar(x - width / 2, original_values, width, label="Initial
        Guess", color="skyblue")
    ax.bar(x + width / 2, approx_values, width, label="Approximate
        Solution", color="orange")

    # Adding labels and title
    ax.set_xlabel("Variables")
    ax.set_ylabel("Values")
    ax.set_title("Initial Guess vs Approximate Solution")
    ax.set_xticks(x)
    ax.set_xticklabels(x_labels)
    ax.legend()

    # Display the plot
    plt.grid(axis="y", linestyle="--", alpha=0.7)
    plt.tight_layout()
    plt.show()
```

Exercise 1. *Solve systems of polynomial equations* $F(x) = [x_1^2 + x_2^2 - 1, x_1^2 - x_2]$.

The Procedure to Solve the Polynomial Systems of Equations which are also Nonlinear Systems of Equations

1. From the second equation: $x_2 = x_1^2$.
2. Substitute into the first equation: $x_1^2 + (x_1^2)^2 - 1 = 0$, which simplifies to: $x_1^4 + x_1^2 - 1 = 0$..
3. Let $y = x_1^2$, so $y^2 + y - 1 = 0$. Solve for y using the quadratic formula: $y = \frac{-1 \pm \sqrt{5}}{2}$. Only the positive root is valid, so $y = \frac{-1+\sqrt{5}}{2}$.
4. Thus, $x_1 = \pm\sqrt{\frac{-1+\sqrt{5}}{2}}$ and $x_2 = x_1^2$.

The solutions are: $x_1 = \pm\sqrt{\frac{-1+\sqrt{5}}{2}}, \quad x_2 = \frac{-1+\sqrt{5}}{2}$.

Solving above exercise using our Newtom Algorithm

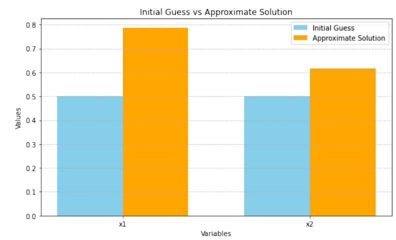

Figure 1: A `marginfigure` shows up in the margin.

Listing 3: Python code for solving a nonlinear system using Newton's method

```python
def F(x):
    # Example: System of nonlinear equations
    # F(x) = [x1^2 + x2^2 - 1, x1^2 - x2]
    return np.array([
        x[0]**2 + x[1]**2 - 1,
        x[0]**2 - x[1]
    ])

def J(x):
    # Jacobian of F(x)
    # J(x) = [[2*x1, 2*x2],
    #         [2*x1, -1]]
    return np.array([
        [2 * x[0], 2 * x[1]],
        [2 * x[0], -1]
    ])

# Initial guess
x0 = [0.5, 0.5]

# Solve using Newton's Method
try:
    solution, iterations = NewtonMethodforSys(F, J, x0)
    print("Solution:", solution)
    print("Iterations:", iterations)
except ValueError as e:
    print(e)


x1 = np.sqrt((-1 + np.sqrt(5))/2)
x2 = (-1 + np.sqrt(5))/2
original_guess = np.array([x1, x2])  # Initial guess
original_guess = [0.5, 0.5]  # Initial guess
solution, _ = NewtonMethodforSys(F, J, original_guess)  # Solve the system
```

```
# Plot the results
plot_solution(F, solution, original_guess)
```

Exercise 2 (2). *Solve the system of nonlinear equations*

Listing 4: Python code for solving a nonlinear system using Newton's method

```
# Example
def F(x):
    return np.array([
        3*x[0] - np.cos(x[1]*x[2]) - 1/2,
        x[0]**2 - 81*(x[1] + 0.1)**2 + np.sin(x[2]) + 1.06,
        np.exp(-x[0]*x[1]) + 20*x[2] + (10*np.pi - 3)/3
        ])

def J(x):
    return np.array([
        [3, x[2]*np.sin(x[1]*x[2]), x[1]*np.sin(x[1]*x[2])],
        [2*x[0], -162*(x[1] + 0.1), np.cos(x[2])],
        [-x[1]*np.exp(-x[0]*x[1]), -x[0]*np.exp(-x[0]*x[1]), 20]
        ])

x0 = [0.1, 0.1, -0.1]
solution, iterations = NewtonMethodforSys(F, J, x0)
print("Solution:", solution)
print("Iterations:", iterations)
```

% Set custom margins for one page with adjusted text height

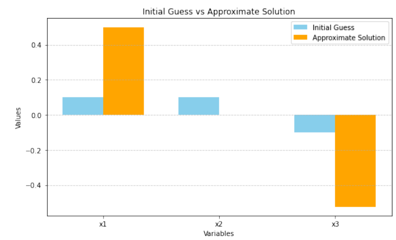

Figure 2: A `marginfigure` shows up in the margin.

Conclusion

Newton-Raphson algorithm is working well for the solving systems of nonlinear equations, where we have to compute the jacobian for the systems of the equations using the iterative procedure i.e the gradient of the system of function ok. there may be possibility that function has local minima where gradient is equal to zero. so we have no solution there. Thus computing Jacobian for the discontinuous or the non-differentiable function it is become difficult to get the solutions.

Chapter 2 : Broyden method

We know that the Newtom method uses the Jacobian computation exactly, while it is difficult to compute the derivatives for the discontinuous for non differetialble function therefore we uses to approximate this jacobian matrix suing the Quasi-Newtom Approximation method where we update the matrix at each iterations.

Let $F(x) = 0$ be the nonlinear system of equation that we wanted to solve thus we approximate the jacobian $J(x)$ by A at each iteration get updated. $x^{i+1} = x^i - A_i^{-1}F(x^{(1)}$

Listing 1: Python code for solving a nonlinear system using Newton's method

```
import numpy as np

def broyden_method(F, x0, tol=1e-5, max_iter=100):
    """
    Broyden's method for solving F(x) = 0.

    Parameters:
    F : function
        The function for which we are seeking a root.
    x0 : numpy array
        Initial guess for the root.
    tol : float
        Tolerance for convergence.
    max_iter : int
        Maximum number of iterations.

    Returns:
    x : numpy array
        The estimated root.
    """
    n = len(x0)
    x = x0
    B = np.eye(n)  # Initial approximation to the Jacobian is the identity matrix
    for i in range(max_iter):
        Fx = F(x)
        if np.linalg.norm(Fx, ord=2) < tol:
            print(f'Converged in {i} iterations')
            return x
        dx = -np.linalg.solve(B, Fx)
        x_new = x + dx
        Fx_new = F(x_new)
        y = Fx_new - Fx
        B += np.outer((y - B @ dx), dx) / np.dot(dx, dx)
        x = x_new
```

```
    raise ValueError('Broyden method did not converge')

# Example usage
def F(x):
    return np.array([x[0]**2 + x[1]**2 - 1, x[0] - x[1]])
x0 = np.array([1.0, 1.0])
#x0 = np.array([0.5, 0.5])
root = broyden_method(F, x0)
print('Root:', root)
```

Exercise 1. *Solve the system of nonlinear equation using Broyden method*

Listing 2: Python code for solving a nonlinear system using Newton's method

```

# Example usage
def F(x):
    return np.array([x[0]**2 + x[1]**2 - 4, x[0] - x[1]])

def J(x):
    return np.array([[2*x[0], 2*x[1]], [1, -1]])

x0 = np.array([0.5, 0.5])
solution = broyden_method(F, J, x0)
print('Solution:', solution)
```