Lab Manual for Numerical Analysis

Lab No. 4

SOLUTION OF NON-LINEAR EQUATIONS BY NEWTON-RAPHSON METHOD AND FIXED-POINT ITERATIVE METHOD

BAHRIA UNIVERSITY KARACHI CAMPUS
Department of Software Engineering

NUMERICAL ANALYSIS

LAB EXPERIMENT # 4

Solution of Non-linear Equations by Newton-Raphson Method & Fixed-Point Iterative Method

OBJECTIVE:

This lab aims to teach students the Newton-Raphson and Fixed-point iterative methods for solving nonlinear equations, emphasizing practical applications in scientific and engineering contexts.

Introduction

As we studied in previous chapter that **nonlinear equations** are mathematical expressions in which the relationship between variables is not linear, often involving complex interactions and non-constant rates of change. These equations appear frequently in various scientific, engineering, and real-world scenarios.

The solutions to nonlinear equations can be approached through two broad categories of methods: **direct** and **indirect**. Direct methods involve solving the equation directly without iterative processes and are effective for relatively simple equations. In contrast, indirect methods require iterative procedures to approximate the solution, making them suitable for complex, nonlinear systems.

Indirect numerical methods for solving equations can be categorized into two primary groups: **bracketing methods** and **open methods**. Bracketing methods focus on restricting the solution within a predetermined interval, offering a structured approach. On the other hand, open methods are characterized by their iterative nature, allowing them to fine-tune the solution without the constraint of bracketing, thus offering versatility in tackling a broader spectrum of mathematical problems and functions.

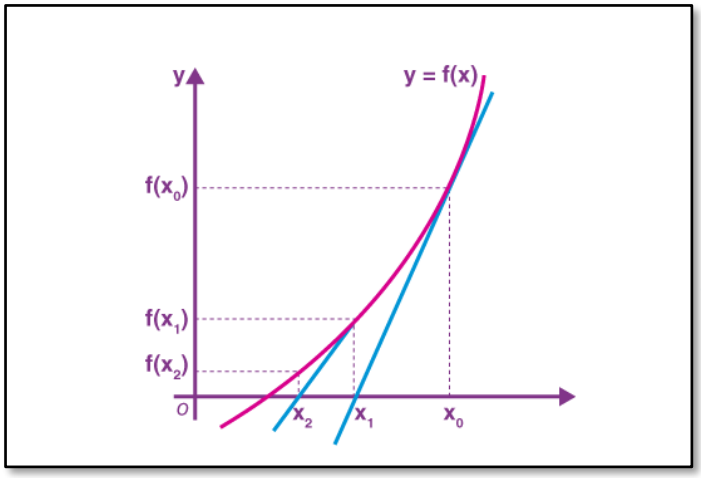
In our previous discussions, we have covered the Bisection Method and the False Position Method for solving nonlinear equations. In this chapter, we will now delve into the **Newton-Raphson method** and the **Fixed-Point Iterative method,** two additional approaches renowned for their effectiveness in tackling nonlinear equations. These methods offer advanced techniques for achieving faster convergence and handling a wider range of mathematical functions, providing valuable tools for numerical problem-solving.

Implementation of Newton-Raphson Method & Fixed-point Iterative Method

In this section, we will provide Python implementations for both the Newton-Raphson Method and the Fixed-Point Iterative Method. These methods are renowned for their effectiveness in numerically solving nonlinear equations. They offer distinct approaches for iteratively refining solutions and are widely used to achieve accurate results across a wide spectrum of mathematical equations.

a. Newton Raphson Method

The **Newton-Raphson** method is a widely recognized and potent numerical approach for solving equations. Its fundamental principle is based on **linear approximation**, and it often demonstrates significantly faster convergence when compared to methods that exhibit linear convergence rates. It is categorized as an **open method** since it relies on a single initial guess for the root to initiate the iterative process. In contrast, some other open methods employ two initial guesses for the root but do not necessitate that these guesses bracket the root.



The Newton method operates **iteratively**, continuously refining its estimates of the desired root. In contrast to solely **computing f(x),** it also computes the derivative, **f'(x).** The method assumes that the next estimate of the root lies where the tangent line intersects the x-axis. This iterative process involves calculating both **f(x)** and **f'(x)** values, and the method repeats until it converges to the desired solution.

Starting from initial guess x_1, the Newton Raphson method uses below **formula** to find next value of x, i.e., x_{n+1} from previous value x.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Where;

- x_n is the value of x at iteration n,
- $f(x_n)$ is the value of the equation at iteration n, and
- $f'(x_n)$ is the value of the first order derivative of the equation or function at iteration n.

Implementation in Python

```
# Define the function f(x)
def func(x):
    return x ** 3 - 5

# Define the derivative of the function, f'(x)
def derivFunc(x):
    return 3*x**2

# Function to find the root using the Newton-Raphson method
def newtonRaphson(x):
    # Initialize the previous result to a large value
    prev_x = float('inf')

    # Iterate until previous result is not equal to new result
    while x != prev_x:
        prev_x = x

        #Calculate approximation using x(i+1) = x(i)-f(x)/f'(x)
        x = x - func(x) / derivFunc(x)

    # Print the approximate root value
    print("The approximate root value is:", "%.4f" % x)

initial_guess = 2  # Initial guess for the root
newtonRaphson(initial_guess)
```

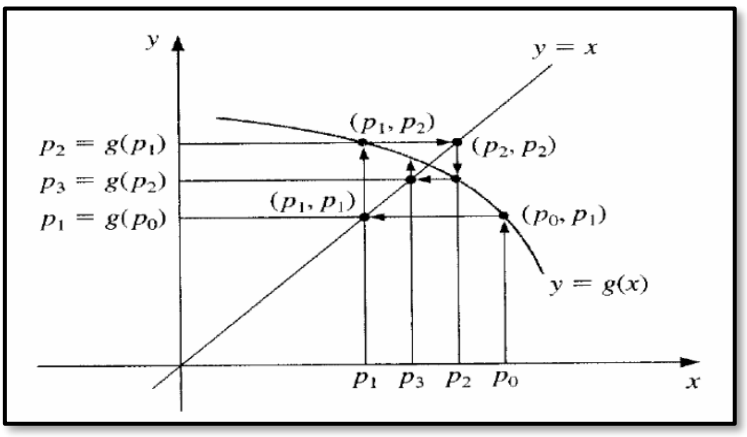
b. Fixed Point Iterative Method

The **fixed-point iteration** method, employed in numerical analysis, serves to approximate solutions for both algebraic and transcendental equations. Solving equations with complexities like **cubic**, **bi-quadratic**, and **transcendental** can often be arduous. In such scenarios, numerical techniques come to our aid, and one of these techniques is the fixed-point iteration method.

This method operates by repeatedly using the concept of a **fixed point** to compute the equation's solution. A fixed point signifies a **point** within a function's domain where the function itself equals that point, denoted as **g(x) = x**. In the fixed-point iteration method, the initial equation is transformed algebraically into the form **g(x) = x**, facilitating the iterative process for finding the solution.

Let consider an equation represented as **f(x) = 0**, which requires finding its solution. To facilitate this, we can express the equation in the form **x = g(x).** The choice of **g(x)** should satisfy the condition **|g'(x)| < 1** at the initial guess **x = x_0**, where xo serves as the starting point for the fixed-point iterative scheme. The iterative process then proceeds by generating

successive approximations, denoted as $x_n = g(x_n - 1)$. In simpler terms, we calculate x_1 using $g(x_0)$, x_2 using $g(x_1)$, and so forth, as we iteratively approach the desired solution.



Implementation in Python

```
import math

# Define a different function f(x) which is to be solved
def f(x):
    return x**3 - 2*x**2 + 1  # Change the function as needed

# Define a different function g(x) based on x = g(x)
def g(x):
    return (2*x**2 - 1)**(1/3)  # Change the function as needed

def fixed_point_iteration(x0, max_iterations):
    step = 1
    x1 = g(x0)

    while step <= max_iterations:
        x0 = x1
        x1 = g(x0)
        step += 1
    return x1

# Set initial guess and maximum iterations
x0 = 1.0  # Initial guess
max_iterations = 50  # Maximum number of iterations

# Call the function & print final approximate root
final_answer = fixed_point_iteration(x0, max_iterations)
print(f"Root is {final_answer:.6f}")
```

Working with User-Input Mathematical Equations in Python

In addition to using predefined mathematical equations and their derivatives, Python offers the convenience of working with user-input mathematical equations through its built-in libraries. By leveraging Python and the powerful **"sympy"** library, which provides extensive symbolic mathematics capabilities, we can undertake tasks such as equation conversion, derivative calculation, and value evaluation.

Following is the description of using this library for working with mathematical equations:

Getting the Equation from user and converting it into a Mathematical Expression

- **Getting the User's Equation:** Start by using the input() function to collect a mathematical equation from the user. This equation is a vital part of the numerical analysis process as it represents the problem to be solved.

```
from sympy import symbols, sympify

# Step 1: Get the equation from the user
equation_str = input("Enter the mathematical equation: ")
```

- **Defining Symbolic Variables:** To manipulate the equation symbolically, define symbolic variables. In numerical analysis, these variables often represent parameters or variables within the mathematical model.

```
# Step 2: Define symbolic variables
x = symbols('x')
```

- **Converting the Equation String:** Utilize the sympify function to convert the user-provided equation string into a mathematical expression. This transformation is essential for numerical analysis.

```
# Step 3: Convert the equation string into a expression
equation = sympify(equation_str)
```

Calculating the Equation's Derivative

- **Importing Necessary Functions:** Before proceeding with numerical analysis, import essential functions from the "sympy" library to facilitate symbolic calculations.

```
from sympy import diff
```

- **Calculating the Derivative:** Calculate the derivative of the equation using the diff function with respect to the symbolic variable. In numerical analysis, derivatives often come into play when approximating solutions or analyzing the behavior of functions.

```
# Calculate the derivative
derivative = diff(equation, x)
```

Evaluating the Expression with Specific Values

- **Evaluating the Expression:** For evaluating the expressions the subs function is used to substitute specific values into the equation.

```
# Evaluate the expression with specific values (e.g., x = 2)
result = derivative.subs(x, 2)
```

Lab Tasks

1. Write a python program for solving the following non-linear equations using Newton-Raphson method correct up to 5 decimal places:
    a. $\cos x = xe^x$ (having initial guess $x_0 = 1$)
    b. $x - 2\sin x - 3 = 0$ (having initial guess $x_0 = 4$)

2. Write a python program for solving the following non-linear equations using fixed point iterative method correct up to 3 decimal places:
    a. $\cos x = 3x - 1$
    b. $2x^3 - 7x^2 - 6x + 1 = 0$