Newton Raphson Method

Numerical Methods (MCSC 202)

Name:	**Manish Pokharel**
Roll:	42
Group:	BSc. Computer Science
Batch:	2021
Date of Sub:	2023/09/17

1 INTRODUCTION

Welcome to this Python project centered around the Newton-Raphson method, a versatile numerical technique essential in computational mathematics and engineering. The Newton-Raphson algorithm, named after its brilliant inventors Sir Isaac Newton and Joseph Raphson, serves as a dynamic tool for approximating solutions to complex equations.

It excels in rapid convergence and finds applications in diverse fields, including optimization, scientific modeling, and root-finding..

In this project, I will dive into the theory behind the Newton-Raphson method, offering a clear understanding of its inner workings. I'll implement it in Python, providing practical coding examples to solve real-world mathematical problems.

2 The Newton-Raphson Iteration

Let $f(x)$ be a well-behaved function, and let r be a root of the equation $f(x) = 0$. Let x_0 be a good estimate of r and let $r = x_0 + h$. Since the true root is r, and $h = r - x_0$, the number h measures how far the estimate x_0 is from the truth.

Since h is 'small', we can use the linear (tangent line) approximation to conclude that

$$0 = f(r) = f(x_0 + h) \simeq f(x_0) + hf'(x_0)$$

and therefore, unless $f'(x_0)$ is close to 0,

$$h \simeq -\frac{f(x_0)}{f'(x_0)} \, .$$

It follows that

$$r = x_0 + h \simeq x_0 - \frac{f(x_0)}{f'(x_0)}$$

Our new improved (?) estimate x_1 of r is therefore given by

$$x_2 = x_1 - \frac{f(x_0)}{f'(x_0)}$$

The next estimate x_2 is obtained from x_1 in exactly the same way as x_1 was obtained from x_0):

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

Continuing in this way, if x_n is the current estimate, the next estimate x_{n+1} is given by

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

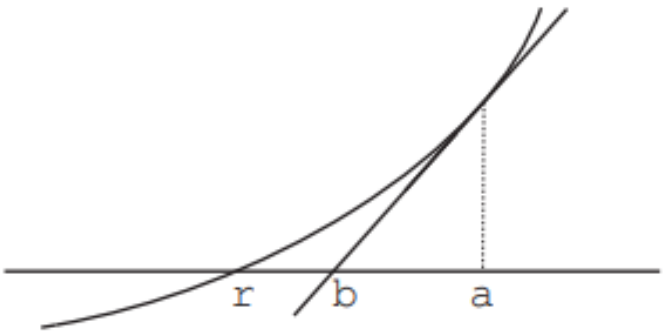
3 Geometric Interpretation of the Newton Raphson

In this picture below, the curve $y = f(x)$ meets the x-axis at r. Let a be the current estimate of r. The tangent line to $y = f(x)$ at the point $(a, f(a))$ has equation

$$y = f(a) + (x - a)f'(a).$$

Let b be the x-intercept of the tangent line. Then,

$$b = a - \frac{f(a)}{f'(a)}$$



Comparing with Equation 1: b is just the 'next' Newton-Raphson estimate of r. The new estimate b is obtained by drawing the tangent line at $x = a$, and then sliding to the x-axis along this tangent line. Now the tangent line is drawn at$(b, f(b))$ and the new tangent line is rode to the x-axis to get a new estimate x. This process is repeated.

We can use the geometric interpretation to design functions and starting points for which the Newton Method runs into trouble. For example, by putting a little bump on the curve at $x = a$ we can make b fly far away from r. When a Newton Method calculation is going badly, a picture can help us diagnose the problem and fix it.

4 CODE

I've used **Python** to write the codes for determining the root of a function using the Newton Raphson.

It consists of 5 primary parts:

4.1 IMPORTING DIFFERENT LIBRARIES

4 libraries have been used in this program:

pandas : Used to tabulate data obtained from NR iterations.

sympy : Used to work with expressions and their derivatives.

matplotlib : Used to graph the expressions, and convergence

numpy : Used to generate grid and points during graphing

The libraries have been imported as:

```
import sympy as sp
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

4.2 SYMBOL SPECIFICATION

Using **sympy**'s sympy.Symbol() function, 'x' was used as a symbol.

```
x = sp.Symbol('x')
```

4.3 ITERATOR FUNCTION DEFINITION

This is where the Newton Raphson Iterator function is defined as:

```
def Newton_Raphson(f, ig, err_tol):
    exp = sp.sympify(f, evaluate = False)
    delta = sp.diff(exp, x) #derivative of exp

    data = []
    error_diff = err_tol
    init_guess = ig
    f_x = d_x = next_guess = 0
    while (err_tol <= error_diff):
        f_x = (exp.subs(x, init_guess)).evalf()
        d_x = (delta.subs(x, init_guess)).evalf()
        next_guess = (init_guess - (f_x/d_x)).evalf()
        error_diff = abs((f_x/d_x).evalf())

        row = [init_guess, f_x, d_x, next_guess, error_diff]
        data.append(row)

        init_guess = next_guess

    df = pd.DataFrame(data, columns = ["x(n)", "f(x)", "f'(x)",
    "x(n+1)", "e(n) - e(n-1)"])
    return [data, df]
```

This function takes in the function, error difference and initial guess as its parameters and returns an array of arrays, in which the first array contains data in the form: $[x_n,\ f(x_n),\ f'(x_n),\ x_{n+1},\ e(n) - e(n-1)]$, and the second object is a data-frame of the data.

4.4 INITIALIZATION, FUNCTION CALLING AND RESULT

This part of the program consists of 3 parts:

- Initializing the initial values and python dictionary:

```
 f = "x**3 - x - 1 "          # function as a string
func = sp.sympify(f, evaluate = False)
error_diff = 0.000001         # error(x) - error(x-1)
init_guess = 1                #initial guess
```

- Calling the **Newton Raphson** function:

```
solution = Newton_Raphson(f, init_guess, error_diff)
```

- Display of the python list using pandas

  The data is tabulated using the data-frame returned by the function.

```
solution[1]
```

4.5 VISUALIZATION

This part visualizes the data[ ] array obtained from the function.

- The points for the x-values from the Newton Raphson iterations are generated.

```
x_values = [x_val[0] for x_val in solution[0]]
y_values = [y_val[1] for y_val in solution[0]]
root = float((x_values[-1]).evalf())
```

- The points for the input expression is generated

```
f_numeric = sp.lambdify(x, func, 'numpy')
x_vals = np.linspace(root+1, root-1, 100)
y_vals = f_numeric(x_vals)
```

- Everything is plotted

```
fig, ax = plt.subplots()
ax.plot(x_vals, y_vals, label='f(x)', linestyle='-', color='blue')
ax.axhline(y=0, color='red', linestyle='--', label='y = 0')
ax.scatter(x_values, y_values, color='green', label='x-values')
plt.legend(loc='upper left')
plt.show()
```

5 ADVANTAGES AND DISADVANTAGES OF NEWTON RAPHSON METHOD

5.1 ADVANTAGES

- **Rapid Convergence:** The method typically converges quickly to a solution, especially when an initial guess is reasonably close to the actual root. This can lead to significant time savings compared to other iterative techniques.
- **High Precision:** Newton-Raphson often provides highly accurate approximations of the root, making it suitable for problems where precision is critical, such as scientific simulations and engineering design.
- **Versatility:** It can be applied to both linear and nonlinear equations, as well as systems of equations. This versatility makes it a go-to method for a wide range of mathematical problems.

5.2 DISADVANTAGES

- **Sensitivity to Initial Guess:** The method can be highly sensitive to the choice of the initial guess. If the initial guess is far from the actual root or lies in a region with poor convergence properties, the method may fail to converge or converge to the wrong root.
- **Requires Derivative Information:** Calculating or approximating derivatives is necessary for the method to work effectively. In some cases, computing derivatives can be challenging or computationally expensive.
- **Limited Domain of Convergence:** Newton-Raphson has a limited domain of convergence. It may not work well for functions with multiple roots in close proximity, as it can easily get trapped in the vicinity of one root and fail to converge to the others.

6 CONCLUSION

In conclusion, the Newton-Raphson method stands as a formidable tool in the realm of numerical analysis and problem-solving. Its rapid convergence, high precision, and versatility make it indispensable for approximating solutions to both linear and nonlinear equations. However, its sensitivity to initial guesses and limited convergence domain necessitate cautious application. Understanding the underlying mathematical principles, derivative calculations, and domain-specific considerations are essential for harnessing its power effectively. In the realm of computational mathematics and engineering, the Newton-Raphson method continues to play a vital role, enabling us to tackle complex problems with efficiency and accuracy, while its limitations remind us of the importance of thoughtful algorithm selection.

7 APPENDIX

7.1 SCREENSHOTS

```python
import sympy as sp
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

```python
#declaring x as symbol
x = sp.Symbol('x')
```

```python
def Newton_Raphson(f, ig, err_tol):
    exp = sp.sympify(f, evaluate = False) # changing string to sympy function
    delta = sp.diff(exp, x) #derivative of exp

    data = []
    error_diff = err_tol
    init_guess = ig
    f_x = d_x = next_guess = 0
    while (err_tol <= error_diff):
        f_x = (exp.subs(x, init_guess)).evalf()
        d_x = (delta.subs(x, init_guess)).evalf()
        next_guess = (init_guess - (f_x/d_x)).evalf()
        error_diff = abs((f_x/d_x).evalf())

        row = [init_guess, f_x, d_x, next_guess, error_diff]
        data.append(row)

        init_guess = next_guess

    df = pd.DataFrame(data, columns = ["x(n)", "f(x)", "f'(x)", "x(n+1)", "e(n) - e(n-1)"])
    return [data, df]
```

```python
f = "x**3 - x - 1 "
func = sp.sympify(f, evaluate = False)
error_diff = 0.000001
init_guess = 1
solution = Newton_Raphson(f, init_guess, error_diff)
```

```python
x_values = [x_val[0] for x_val in solution[0]]
y_values = [y_val[1] for y_val in solution[0]]
root = float((x_values[-1]).evalf())
```

```python
f_numeric = sp.lambdify(x, func, 'numpy')
x_vals = np.linspace(root+1, root-1, 100)
y_vals = f_numeric(x_vals)
fig, ax = plt.subplots()
ax.plot(x_vals, y_vals, label='f(x)', linestyle='-', color='blue')
ax.axhline(y=0, color='red', linestyle='--', label='y = 0')
ax.plot(x_vals, y_vals, color='blue', label='Points')
ax.scatter(x_values, y_values, color='green', label='Points')
plt.show()
```

Figure 1: Code for Newton Raphson Method in Python

In [13]:
```
f = "x**3 - x - 1 "
func = sp.sympify(f, evaluate = False)
error_diff = 0.000001
init_guess = 1
solution = Newton_Raphson(f, init_guess, error_diff)
solution[1]
```

Out[13]:

	x(n)	f(x)	f'(x)	x(n+1)	e(n) - e(n-1)
0	1	-1.00000000000000	2.00000000000000	1.50000000000000	0.500000000000000
1	1.50000000000000	0.875000000000000	5.75000000000000	1.34782608695652	0.152173913043478
2	1.34782608695652	0.100682173091148	4.44990548204159	1.32520039895091	0.0226256880056149
3	1.32520039895091	0.00205836191666342	4.26846829213893	1.32471817399905	0.00048222495185315 7
4	1.32471817399905	9.24377759670136e-7	4.26463472157016	1.32471795724479	2.16754263851652e-7

Figure 2: Solution of $x^3 - x - 1$

```
f_numeric = sp.lambdify(x, func, 'numpy')
x_vals = np.linspace(root+1, root-1, 100)
y_vals = f_numeric(x_vals)
fig, ax = plt.subplots()
ax.plot(x_vals, y_vals, label='f(x)', linestyle='-', color='blue')
ax.axhline(y=0, color='red', linestyle='--', label='y = 0')
ax.scatter(x_values, y_values, color='green', label='x-values')
plt.legend(loc='upper left')
plt.show()
```

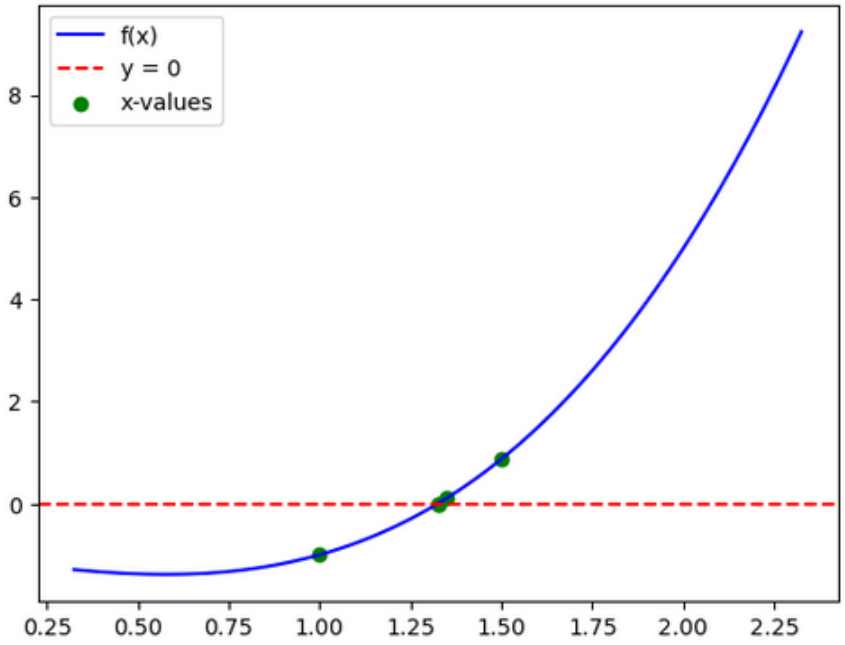


Figure 3: Newton Raphson Visualization