Solving Nonlinear Systems

Wanzhu Zheng

June 2023

1 Problem Description

This report demonstrates how to solve nonlinear systems using iterative methods.

Suppose we have a system of nonlinear equations:

$$x_1^2 - 10x_1 + x_2^2 + 8 = 0$$
$$x_1 x_2^2 + x_1 - 10x_2 + 8 = 0$$

We can solve this using two methods.

Fixed Point Method:
The Fixed Point Method takes an initial guess, say $x^{(0)} = 0$. We set the $f(x) = 0$ to solve for $x = g(x)$ and then use the initial guess to complete the sequence $x_i^k = g(x_1^{(k-1)}, x_2^{(k-1)}, ...)$.

Newton's Method:
Newton's Method also takes an initial guess $x^{(0)}$. We define the Jacobian matrix $J(x)$. We perform this method in a two-step process:

1. Find a vector y that satisfies $J(x^{(k-1)})y = -F(x^{(k-1)})$
2. Solve for the new approximation $x^{(k)}$ by adding y to $x^{(k-1)}$

2 Results

Here, we demonstrate the result of finding the roots for the nonlinear system described above at $TOL < 10^{-6}$. We see below that Newton's method returns a more accurate result compared to fixed point.

Fixed Point Method: $(0.99999957, 0.99999957)$
Newton's Method: $(1, 1)$

We see in the table below the wall clock time and number of iterations for each method. Although Newton's method requires less iterations, it has the same wall clock time. This can be explained by the need to compute $J(x)$ at each step. Moreover, we need to solve for y at each iteration, meaning we would have to call our LU Decomposition function at each step.

	Time	Iterations
Fixed Point Method	6.2088×10^{-4}	16
Newton's Method	5.8388×10^{-4}	5

The graph at 1 depicts the residual at each iteration for the two different methods. We see that Newton's Method converges faster which is to be expected since it requires less iterations.

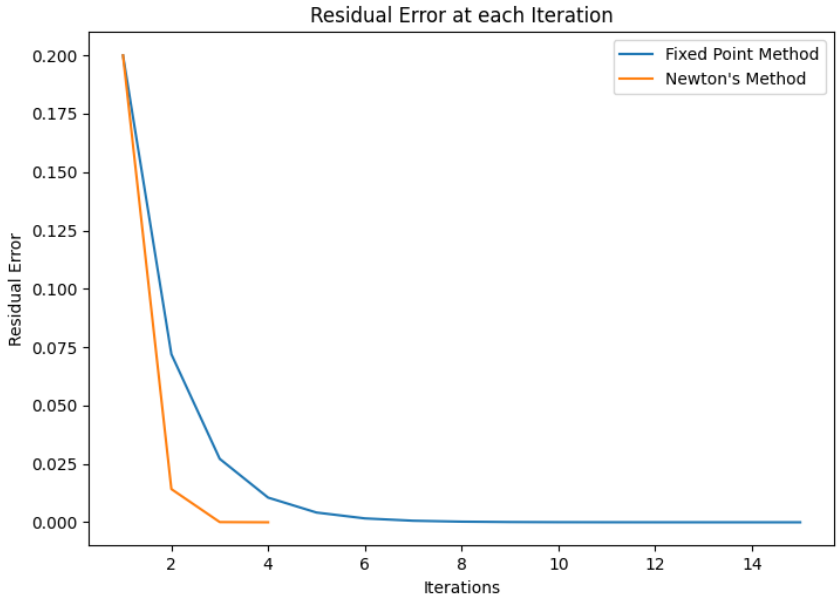

Figure 1: Iteration vs. Residual Error

3 Appendix

```
import numpy as np
from scipy.optimize import fsolve
import matplotlib.pyplot as plt
import time


def f(x):
    f1 = x[0] ** 2 - 10 * x[0] + x[1] ** 2 + 8
    f2 = x[0] * x[1] ** 2 + x[0] - 10 * x[1] + 8
    return np.array([f1, f2])


root = fsolve(f, [1, 1])


def g(x):
    g1 = (x[0] ** 2 + x[1] ** 2 + 8) / 10.0
    g2 = (x[0] * x[1] ** 2 + x[1] + 8) / 10.0
    return np.array([g1, g2])


def fixed_point(x_old, tol=1e-6, max_iter=1000):
    step = 1
    norm = np.array([])
    for i in range(max_iter):
        x_new = g(x_old)
        x_old = x_new
```

```
        step += 1
        norm = np.append(norm, np.linalg.norm(x_new - root, ord=np.inf))
        if np.linalg.norm(x_new - root, ord=np.inf) < tol:
            break
    return x_new, step, norm


print(fixed_point(np.array([0, 0])))


# LU decomposition
def lu(A):
    L = np.zeros_like(A)
    U = np.zeros_like(A)
    N = np.size(A, 0)

    for k in range(N):
        L[k, k] = 1
        U[k, k] = (A[k, k] - np.dot(L[k, :k], U[:k, k])) / L[k, k]
        for j in range(k + 1, N):
            U[k, j] = (A[k, j] - np.dot(L[k, :k], U[:k, j])) / L[k, k]
        for i in range(k + 1, N):
            L[i, k] = (A[i, k] - np.dot(L[i, :k], U[:k, k])) / U[k, k]
    return L, U


# backwards substitution
def back_sub(A, b):
    n = len(b)
    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        x[i] = (b[i] - np.dot(A[i, i+1:], x[i+1:])) / A[i, i]
    return x


# forward substitution
def forward_sub(A, b):
    n = len(b)
    x = np.zeros(n)
    for i in range(n):
        x[i] = b[i] - np.dot(A[i, :i], x[:i]) / A[i][i]
    return x


# LUx = b
def lu_solver(A, b):
    L = lu(A)[0]
    U = lu(A)[1]
    y = forward_sub(L, b)
    x = back_sub(U, y)
    return x


def jacobian(x):
```

```
    J = np.array([[2 * x[0] - 10, 2 * x[1]],
                  [x[1] ** 2 + 1, 2 * x[0] * x[1] - 10]])
    return J


def newton(x, tol=1e-6, max_iter=1000):
    step = 1
    norm = np.array([])
    for k in range(max_iter):
        y = lu_solver(jacobian(x), -f(x))
        x = x + y
        step += 1
        norm = np.append(norm, np.linalg.norm(x - root, ord=np.inf))
        if np.linalg.norm(x - root, ord=np.inf) < tol:
            break
    return x, step, norm


print(newton(np.array([0, 0])))


def clock(type=0):
    if type == 0:
        t0 = time.time()
        fixed_point(np.array([0, 0]))
        t1 = time.time()
        total_time = t1 - t0
        return total_time
    if type == 1:
        t0 = time.time()
        newton(np.array([0, 0]))
        t1 = time.time()
        total_time = t1 - t0
        return total_time


print(clock(type=0))
print(clock(type=1))

iter1 = np.linspace(1, 15, num=15)
iter2 = np.linspace(1, 4, num=4)
n_error = newton(np.array([0, 0]))[2]
newton_error = n_error.tolist()
f_error = fixed_point(np.array([0, 0]))[2]
fixed_error = f_error.tolist()

plt.plot(iter1, fixed_error)
plt.plot(iter2, newton_error)
plt.xlabel("Iterations")
plt.ylabel("Residual Error")
plt.title("Residual Error at each Iteration")
plt.legend(["Fixed Point Method", "Newton's Method"], loc="upper right")
plt.show()
```