Solving Nonlinear Systems via Jacobian Iterative Methods

Ali Bijari

June 10, 2022

1 Introduction

This document demonstrates the solution of a nonlinear system of equations using symbolic computation and the Jacobian-based iterative method (Newton-like), implemented in Python with `sympy` and `numpy`.

1.1 Mathematical Modeling

Consider the following system of equations:

$$\begin{aligned} f_1(x, y, z) &= 3x + x^2 - 2yz - 0.1 \\ f_2(x, y, z) &= 2y - y^2 + 3xz - 0.2 \\ f_3(x, y, z) &= -z + z^2 + 2xy - 0.3 \end{aligned}$$

We seek the roots (x, y, z) such that $f_1 = f_2 = f_3 = 0$.

1.2 Mathematical Modeling of Nonlinear Systems

Many problems in physics, engineering, and applied mathematics can be reduced to solving a system of nonlinear equations:

$$\begin{cases} f_1(x_1, x_2, \ldots, x_n) = 0 \\ f_2(x_1, x_2, \ldots, x_n) = 0 \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) = 0 \end{cases} \tag{1}$$

where f_i are nonlinear functions and x_i are the variables to be determined.

Nonlinear systems arise frequently, for example, in modeling chemical reaction equilibria, solving quantum mechanical systems, optimization, and in describing coupled physical phenomena.

1.3 The Newton-Raphson and Generalized Iterative Methods

A common approach for finding the roots of such systems is the **Newton-Raphson method** generalized to multiple variables. The method relies on a first-order Taylor expansion of the functions:

$$\mathbf{F}(\mathbf{x} + \Delta\mathbf{x}) \approx \mathbf{F}(\mathbf{x}) + J(\mathbf{x})\Delta\mathbf{x} \tag{2}$$

where $\mathbf{F} = [f_1, \ldots, f_n]^T$ and $J(\mathbf{x})$ is the Jacobian matrix with elements

$$J_{ij} = \frac{\partial f_i}{\partial x_j} \tag{3}$$

The Newton step solves:

$$J(\mathbf{x}_k)\Delta\mathbf{x}_k = -\mathbf{F}(\mathbf{x}_k) \tag{4}$$

and the new guess is updated by:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k \tag{5}$$

1.4 Steepest Descent/Relaxation Variants

For better stability, especially if the Jacobian is ill-conditioned or the initial guess is far from the root, a *relaxation parameter* or a *steepest-descent-inspired step* is used:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mu_k J(\mathbf{x}_k)\mathbf{F}(\mathbf{x}_k) \tag{6}$$

where μ_k is a dynamically chosen step size:

$$\mu_k = \frac{\mathbf{F}^T J J^T \mathbf{F}}{(J J^T \mathbf{F})^T (J J^T \mathbf{F})} \tag{7}$$

This approach blends Newton's method with gradient descent, improving robustness for strongly nonlinear systems.

1.5 Python Implementation Structure

The following code implements this iterative approach using symbolic computation (`sympy`) and numerical evaluation:

1. **Input:** User provides initial guess vector.
2. **Symbolic Model:** The system functions and Jacobian matrix are defined symbolically.
3. **Iteration:**
   (a) Evaluate functions and Jacobian at current guess.
   (b) Compute the update step using the weighted Jacobian.
   (c) Update the guess vector.
   (d) Check for convergence (maximum change $< \epsilon$).
4. **Output:** The algorithm prints detailed information at each step and the final solution.

1.6 Annotated Code Snippet

Listing 1: Python code for iterative solution of nonlinear systems

```python
import numpy as np
import sympy

def calculate_jacobian(functions, variables):
J = sympy.zeros(len(functions), len(variables))
for i, f in enumerate(functions):
for j, v in enumerate(variables):
```

```
J[i, j] = sympy.diff(f, v)
return J

# Define the nonlinear system
x, y, z = sympy.symbols('x y z')
f1 = 3*x + x**2 - 2*y*z - 0.1
f2 = 2*y - y**2 + 3*x*z - 0.2
f3 = -z + z**2 + 2*x*y - 0.3
functions = [f1, f2, f3]
variables = [x, y, z]

# Initialize guess vector, iteration parameters
vector = sympy.Matrix([1.0, 1.0, 1.0])
epsilon = 1e-5
max_iterations = 100

for _ in range(max_iterations):
J = calculate_jacobian(functions, variables)
x_val, y_val, z_val = [float(v) for v in vector]
f_vals = [f.subs([(x, x_val), (y, y_val), (z, z_val)]) for f in functions]
f_vec = sympy.Matrix(f_vals)
J_eval = J.subs([(x, x_val), (y, y_val), (z, z_val)])
Jt_eval = J_eval.transpose()
# Steepest descent-like update
A0 = J_eval * Jt_eval * f_vec
mu = (f_vec.transpose() * A0)[0, 0] / ((A0.transpose() * A0)[0, 0])
vector_new = vector - mu * J_eval * f_vec
if np.max(np.abs(vector_new - vector)) < epsilon:
break
vector = vector_new
print("Solution:", vector)
```

1.7 Discussion and Applications

- This method is well-suited for systems where the Jacobian can be computed analytically or symbolically.
- The hybrid step-size adaption improves global convergence compared to the pure Newton step.
- This technique is widely used in computational physics (nonlinear eigenproblems, reaction networks), chemistry, engineering optimization, and many areas where coupled nonlinear equations arise.

1.8 Summary

By leveraging symbolic computation and iterative updates with Jacobian information, this method provides a powerful framework for numerically solving nonlinear systems that are otherwise intractable analytically.