Progress Report: Broyden method for nonlinear PDE

Eric Montalbán

June 27, 2025

Contents

1 Introduction

In this report, we focus on the viscous Burgers' equation and compare Newton's with Broyden's method from this nonlinear PDE.

2 Viscous Burgers' equation

To compare Broyden's method and Newton's method, we choose the viscous Burgers' equation as our candidate:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \quad x \in [0,1],\ t > 0 \tag{1}$$

with the initial condition:

$$u(x,0) = \sin(\pi x) \tag{2}$$

and boundary conditions:

$$u(0,t) = u(1,t) = 0 \tag{3}$$

3 PDE to Systems

We begin by discretizing the nonlinear PDE, resulting in a nonlinear system. In other words, the vector $F(u^{n+1}) = 0$ cannot be written as a linear system $Au_{n+1} = b$. To solve it, we will use iterative solvers like Broyden's, a quasi-newton method.

The first step is to discretize the time domain in N_t, $\{u^1, u^2, ...u^{N_t}\}$. Once it is done, for each time step, we will create a non-linear system of equations by discretizing the spatial domain into N_x grid points and approximating spatial derivatives using central and upwind differences:

$$\left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_{i+1} - u_{i-1}}{2\,\Delta x}, \quad \left(\frac{\partial^2 u}{\partial x^2}\right)_i \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}$$

To estimate the derivatives of $\frac{\partial u^{n+1}}{\partial x}$ using finite differences, there are two different kind of techniques:

- Backward Euler (Implicit): It uses the same time step iterations to calculate derivates. It is naturally more accurate in detriment of the computation cost.

$$\frac{\partial u^{n+1}}{\partial x} = \frac{u_{i+1}^{n+1} - u_{i-1}^{n+1}}{2\,\Delta x}$$

- Forward Euler (Explicit): This one, however, is based on the known values already calculated in previous time steps iterations.We look in the past to predict the future. Therefore, it is cheaper computationally speaking but also condtiionally unstable for bigger time steps.

$$\frac{\partial u^{n+1}}{\partial x} = \frac{u_{i+1}^{n} - u_{i-1}^{n}}{2\,\Delta x}$$

We shall proceed by using the explicit method in nonlinear terms looking for efficiency as they are more complex than usual. For the rest of terms, we will use the traditional backward Euler method.This approach results in a semi-implicit Euler scheme. It is not Crank-Nicolson, which would involve averaging time levels; instead, we use an implicit scheme for the linear terms and an explicit scheme for the nonlinear convection. The result after discretizing the PDE is as follows:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + u^n \cdot \frac{u_{i+1}^n - u_{i-1}^n}{2\,\Delta x} = \nu \cdot \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2}$$

Here we used Backward Euler (implicit) for both the time derivate and the second space derivate whereas we opted for approximating the nonlinear space derivative explicitly. This nonlinear system is then solved iteratively using either Newton's method and Broyden's method.

4 Python: Numerical Analysis

The nonlinear system we solve at each time step is defined by:

$$F_i(u^{n+1}) = \frac{u_i^{n+1} - u_i^n}{\Delta t} + u_i^n \frac{u_{i+1}^n - u_{i-1}^n}{2\,\Delta x} - \nu \frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2}$$

The Jacobian J is a tridiagonal matrix, which using later `np.linalg.inv()`, we will be able to compute its inverse. Its entries are given by:

$$J_{i,j} = \frac{\partial F_i}{\partial u_j^{n+1}} = \begin{cases} \dfrac{1}{\Delta t} + \dfrac{2\nu}{\Delta x^2}, & j = i, \\ -\dfrac{\nu}{\Delta x^2}, & j = i+1, \\ -\dfrac{\nu}{\Delta x^2}, & j = i-1, \\ 0, & \text{otherwise.} \end{cases}$$

Note that the Jacobian is of size $(N_x - 2)$
$times(N_x - 2)$, as we exclude the boundary nodes due to Dirichlet conditions.

4.1 Broyden's Method Implementation

Broyden's method updates an approximation of the inverse Jacobian instead of recalculating it at every iteration: We leave the algorithm as well:

Step 1 Set $A_0 = J(\mathbf{x})$ where $J(\mathbf{x})_{i,j} = \frac{\partial f_i}{\partial x_j}(\mathbf{x})$ for $1 \leq i, j \leq n$;
$\mathbf{v} = \mathbf{F}(\mathbf{x})$. (*Note:* $\mathbf{v} = \mathbf{F}(\mathbf{x}^{(0)})$.)

Step 2 Set $A = A_0^{-1}$. (*Use Gaussian elimination.*)

Step 3 Set $\mathbf{s} = -A\mathbf{v}$; (*Note:* $\mathbf{s} = \mathbf{s}_1$.)
$\mathbf{x} = \mathbf{x} + \mathbf{s}$; (*Note:* $\mathbf{x} = \mathbf{x}^{(1)}$.)
$k = 2$.

Step 4 While $(k \leq N)$ do Steps 5–13.

Step 5 Set $\mathbf{w} = \mathbf{v}$; (*Save* $\mathbf{v}$.)
$\mathbf{v} = \mathbf{F}(\mathbf{x})$; (*Note:* $\mathbf{v} = \mathbf{F}(\mathbf{x}^{(k)})$.)
$\mathbf{y} = \mathbf{v} - \mathbf{w}$. (*Note:* $\mathbf{y} = \mathbf{y}_k$.)

Step 6 Set $\mathbf{z} = -A\mathbf{y}$. (*Note:* $\mathbf{z} = -A_{k-1}^{-1}\mathbf{y}_k$.)

Step 7 Set $p = -\mathbf{s}^t\mathbf{z}$. (*Note:* $p = \mathbf{s}_k^t A_{k-1}^{-1}\mathbf{y}_k$.)

Step 8 Set $\mathbf{u}^t = \mathbf{s}^t A$.

Step 9 Set $A = A + \frac{1}{p}(\mathbf{s} + \mathbf{z})\mathbf{u}^t$. (*Note:* $A = A_k^{-1}$.)

Step 10 Set $\mathbf{s} = -A\mathbf{v}$. (*Note:* $\mathbf{s} = -A_k^{-1}\mathbf{F}(\mathbf{x}^{(k)})$.)

Step 11 Set $\mathbf{x} = \mathbf{x} + \mathbf{s}$. (*Note:* $\mathbf{x} = \mathbf{x}^{(k+1)}$.)

Step 12 If $||\mathbf{s}|| < TOL$ then OUTPUT $(\mathbf{x})$;
(*The procedure was successful.*)
STOP.

Figure 1: Broyden's algorithm

```
    # Parameters
L, Nx = 1, 50
T, dt = 1, 0.01
nu = 0.01
x = np.linspace(0, L, Nx)
dx = L / (Nx - 1)
Nt = int(T / dt) + 1
TOL = 1e-15
MAX_ITER = 30

# Initial condition
u0 = np.sin(np.pi * x)
u0[0] = u0[-1] = 0

def F(u, dx, dt, nu, u_old):
    f = np.zeros_like(u)
    for i in range(1, len(u) - 1):
        derivx = (u_old[i + 1] - u_old[i - 1]) / (2 * dx)
```

```
        derivxx = (u[i + 1] - 2 * u[i] + u[i - 1]) / dx**2
        f[i] = (u[i] - u_old[i]) / dt + u_old[i] * derivx - nu * derivxx
    return f

def J(u, dx, dt, nu):
    N = len(u)
    J = np.zeros((N - 2, N - 2))
    diag = 1 / dt + 2 * nu / dx**2
    sup = -nu / dx**2
    inf= sup
    for i in range(N - 2):
        if i > 0:
            J[i, i - 1] = sup
        J[i, i] = diag
        if i < N - 3:
            J[i, i + 1] = inf
    return J
def broyden(u, dx, dt, nu, TOL, N, u_old):
    Fx = F(u, dx, dt, nu, u_old)
    Jx = J(u, dx, dt, nu)
    A = np.linalg.inv(Jx)
    s = -A @ Fx[1:-1]
    u[1:-1] += s
    errors, residuals = [np.linalg.norm(s)], [np.linalg.norm(Fx)]

    for _ in range(1, N):
        Fx_new = F(u, dx, dt, nu, u_old)
        y = Fx_new[1:-1] - Fx[1:-1]
        z = -A @ y
        p = -s @ z
        uv = s @ A
        A += (1 / p) * np.outer(s + z, uv)
        s = -A @ Fx_new[1:-1]
        u[1:-1] += s
        errors.append(np.linalg.norm(s))
        residuals.append(np.linalg.norm(Fx_new))
        Fx = Fx_new
        if errors[-1] < TOL:
            break

    return u, errors, residuals
```

We use the `@` operator for matrix-vector multiplication. For example, `A @ b` performs a matrix product where b is a column vector and returns a vector `c`. The function `np.outer(a, b)` computes the product of vectors `a` and `b`, resulting in a matrix that allocates in the ele-

ment A(i,j) the product `a[i] * b[j]`. This is used in the rank-one update step of Broyden's method. We use `x[1:-1]` to exclude boundary conditions from updating, as they are fixed by the Dirichlet boundary condition.

The matrix update used in Broyden's method is based on the Sherman-Morrison formula, a rank-one update for inverse matrices:

$$A_{k+1}^{-1} = A_k^{-1} + \frac{(s_k + z_k)(u_k)^T}{p_k}$$

4.2 Comparison with Newton

Newton's method computes the Jacobian at each iteration, which costs $\mathcal{O}(n^3)$ and requires n^2 functional operations. Broyden's method, in contrast, only needs n function evaluation and updates an approximation of the inverse Jacobian, making it much cheaper. As a result, Newton offers quadratic convergence versus the superlineal convergence of Broyden but with a more friendly computational cost as we can see.

Method	**Function Evals**	**Jacobian Update**	**Total Cost**
Newton	$n^2 + n$ (F(u) + J(u))	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$
Broyden	n	$\mathcal{O}(n^2)$ (matrix*vector)	$\mathcal{O}(n^2)$

4.3 Main Loop

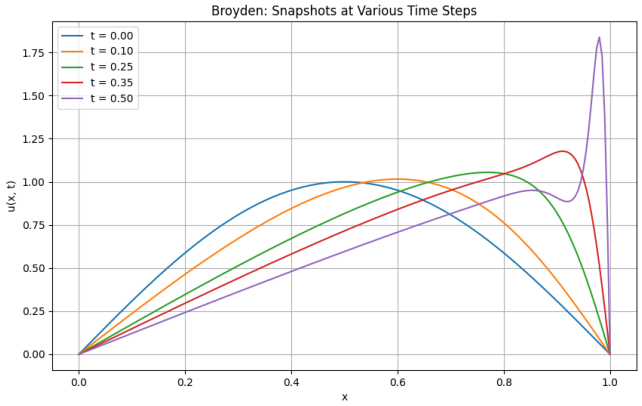
Broyden is called at each time step, solving N_t nonlinear systems altogether.

```
for n in range(1, Nt):
    u_old = u.copy()
    u = broyden(u.copy(), dx, dt, nu, TOL, N, u_old)
    U[n, :] = u
```

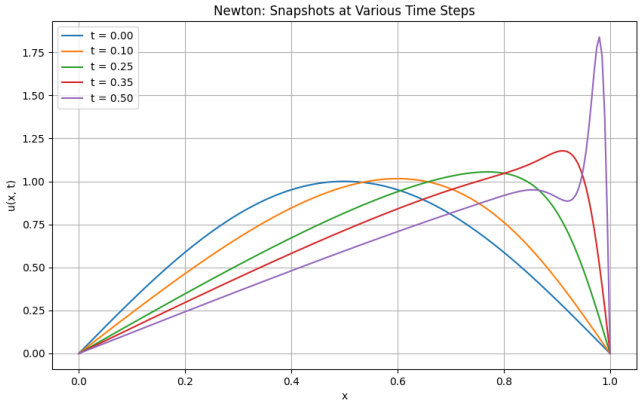
4.4 Solutions

We got a maximum difference of 4.44e-16 when executing both methods with this parameter values:

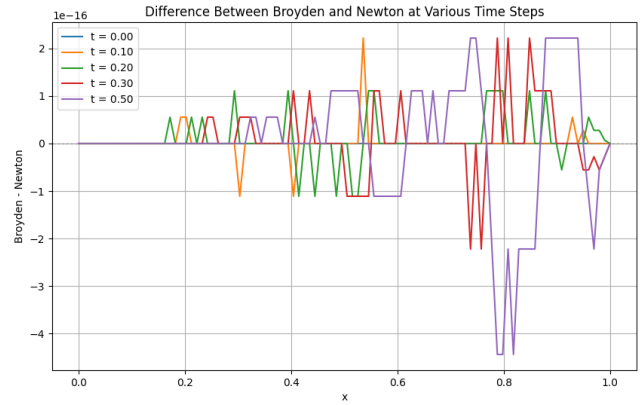
```
# Parameters
L, Nx = 1, 100
T, dt = 0.5, 0.1
nu = 0.01
x = np.linspace(0, L, Nx)
dx = L / (Nx - 1)
Nt = int(T / dt) + 1
TOL = 1e-15
MAX_ITER = 30
```



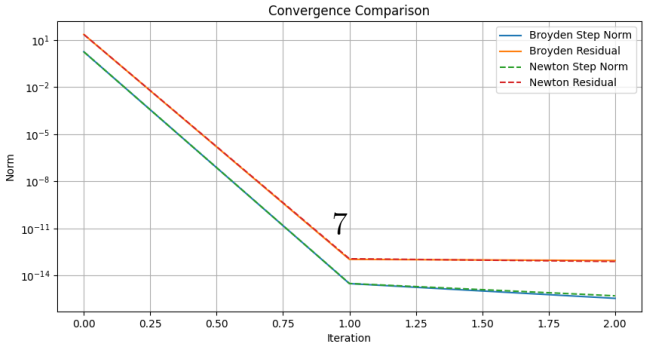
(a) Broyden Snapshot



(b) Newton Snapshot



(c) Difference Between Methods



References

[1] Burden, R. L., & Faires, J. D. (2010). *Numerical Analysis*. Brooks/Cole.

[2] Cordero, A., Soleymani, F., & Torregrosa, J. R. (2014). *Dynamical analysis of iterative methods for nonlinear systems or how to deal with the dimension?* Applied Mathematics and Computation.

[3] Kansal, M., Cordero, A., Bhalla, S., & Torregrosa, J. R. (2020). *New fourth- and sixth-order classes of iterative methods for solving systems of nonlinear equations and their stability analysis*. Numerical Algorithms.

[4] Cordero, A., Garrido, N., Torregrosa, J. R., & Triguero-Navarro, P. (2022). *Symmetry in the multidimensional dynamical analysis of iterative methods with memory*. Symmetry, 14(3), 442.

[5] Chicharro, F. I., Cordero, A., Garrido, N., & Torregrosa, J. R. (2020). *On the effect of the multidimensional weight functions on the stability of iterative processes*. Journal of Computational and Applied Mathematics, 113052.

[6] Behl, R., Cordero, A., & Torregrosa, J. R. (2020). *High order family of multivariate iterative methods: Convergence and stability*. Journal of Computational and Applied Mathematics, 113053.

[7] Cordero, A., Giménez-Palacios, I., & Torregrosa, J. R. (2019). *Avoiding strange attractors in efficient parametric families of iterative methods for solving nonlinear problems*. Applied Numerical Mathematics, 141, 157–172.