MTH 654-2024 Homework A2. **Noah Prentice**
Date: November 7, 2024
Prof. M. Peszynska

> **Note.**
>
> Code for this assignment was done in Python. Only important code snippets and outputs are shown here, but complete files can be found at this GitHub repository.[a] All code for Exercise N is in `ExN.py`.
>
> [a] URL: https://github.com/NoahPrentice/Nonlinear-Coupled-PDE-MTH654-F24/tree/main/A2

Exercise 1. Consider the PDE

$$u_t - \varepsilon u_{xx} = g(x,t,u),\ x \in (0,1), t \in (0,T] \tag{1a}$$

$$u(x,0) = u_{init}(x) \tag{1b}$$

$$u_x|_{x=0,t} = u_x|_{x=1,t} = 0, \tag{1c}$$

and let $g(x,u) = f(u) + F(x,t)$.

Describe, implement and test a FV scheme based on `ELLIPTIC1d.m` for (1) with time-stepping following at least one of the choices (i) `IM`plicit (diffusion)/`EX`plicit (reaction), (ii) fully implicit with fixed-point iteration. Test with at least one of the choices (a) $f(u) = a(u - u^3)$ (phase transitions), and/or (b) $f(u) = bu(1-u)$ (neuroscience).

Set $F(x,t) \equiv 0$ and $u_{init}(x) = \sin(4\pi(x + \sin(x))) + \pi x^4$. Show the solution at $u(x, .1)$ when $\varepsilon = 1, a = 1$ or $b = 1$.

Solution. Recall that `ELLIPTIC1d.m` solves the PDE $-(ku_x)_x = f(x)$ (with boundary conditions) through a finite volume (FV) approach. The problem for this assignment is different:

$$u_t - (\varepsilon u_x)_x = f(u), \quad \text{plus boundary/initial conditions.}$$

There are two notable differences: the presence of a time derivative and the dependence of f on u instead of x. Luckily this does not affect the spatial discretization of our domain. So, we discretize as follows:

- Space. We discretize in space according to FV: we split $\Omega = (0,1)$ into cells $\omega_1, \omega_2, \ldots, \omega_M$ with centers $x_1, x_2, \ldots, x_M$ and lengths $h_1, h_2, \ldots, h_M$, respectively.
- Time. We follow (i), an IMEX temporal discretization, in which we treat the diffusion term $-(\varepsilon u_x)_x$ implicitly and the reaction term $f(u)$ explicitly.

This gives us the following fully discrete equations for the interior cells $j = 2, \ldots, M-1$:

$$\frac{1}{\tau}\big(U_j^n - U_j^{n-1}\big) + \frac{1}{h_j}\big(-\mathcal{T}_{j-1/2}U_{j-1}^n + (\mathcal{T}_{j-1/2} + \mathcal{T}_{j+1/2})U_j^n - \mathcal{T}_{j+1/2}U_{j+1}^n\big) = f(U_j^{n-1})$$

where, as in the ODE with constant coefficients, we have

$$\mathcal{T}_{j+1/2} = \frac{2}{\frac{h_j}{K_j} + \frac{h_{j+1}}{K_{j+1}}}, \quad K_j = k(x_j) = \varepsilon.$$

Of course, for the initial conditions we set $U_j^0 = u_{init}(x_j)$, and the homogeneous Neumann boundary conditions provide the additional equations

$$\frac{1}{\tau}\big(U_1^n - U_1^{n-1}\big) + \frac{1}{h_1}\big(-\mathcal{T}_{1+1/2}(U_2^n - U_1^n)\big) = f(U_1^{n-1}),$$

$$\frac{1}{\tau}\big(U_M^n - U_M^{n-1}\big) + \frac{1}{h_M}\big(\mathcal{T}_{M-1/2}(U_M^n - U_{M-1}^n)\big) = f(U_M^{n-1}).$$

For implementation, I construct these equations in matrix-vector form at each time-step $t_n > 0$ and solve using a sparse solver (as the resulting matrix is tri-diagonal):

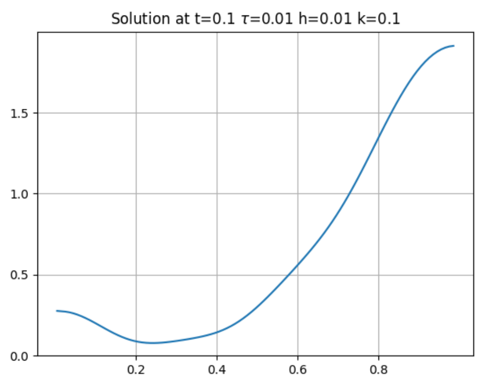
```python
def build_LHS_matrix(h_values: np.ndarray):
    """Builds the sparse matrix A that results from putting the fully-discrete equations
    into matrix-vector form AU = F.
    """

    # Note that, in ELLIPTIC1d.m, what I call "transmissibility_vector" is called "tx,"
    # and what I call "number_of_cells" is called "nxdx."
    transmissibility_vector = build_transmissibility_vector(h_values)
    number_of_cells = h_values.size
    LHS_matrix = sparse.lil_array((number_of_cells, number_of_cells))

    # --- Interior cells ---
    for j in range(1, number_of_cells - 1):
        LHS_matrix[j, j - 1] = -tau * transmissibility_vector[j - 1][0]

        LHS_matrix[j, j] = h_values[j] + tau * (
            transmissibility_vector[j - 1][0] + transmissibility_vector[j][0]
        )

        LHS_matrix[j, j + 1] = -tau * transmissibility_vector[j][0]

    # --- Boundary cells ---
    # First cell, index 0
    LHS_matrix[0, 0] = h_values[0] + tau * transmissibility_vector[1]
    LHS_matrix[0, 1] = -tau * transmissibility_vector[1]

    # Last cell, index number_of_cells - 1 = M - 1.
    last_cell_index = number_of_cells - 1
    LHS_matrix[last_cell_index, last_cell_index - 1] = (
        -tau * transmissibility_vector[last_cell_index - 1]
    )
    LHS_matrix[last_cell_index, last_cell_index] = (
        h_values[last_cell_index] + tau * transmissibility_vector[last_cell_index - 1]
    )
    return LHS_matrix
```

```python
def build_RHS_vector(
    h_values: np.ndarray, tau: float, previous_solution: np.ndarray
) -> np.ndarray:
    """Builds the column vector F that results from putting the fully-discrete equations
    in matrix-vector form AU = F.
    """
    RHS_vector = reaction_function_for_vectors(previous_solution)  # f(u) in Pbm. 1
    RHS_vector *= tau
    RHS_vector += previous_solution
    RHS_vector *= h_values
    return RHS_vector
```

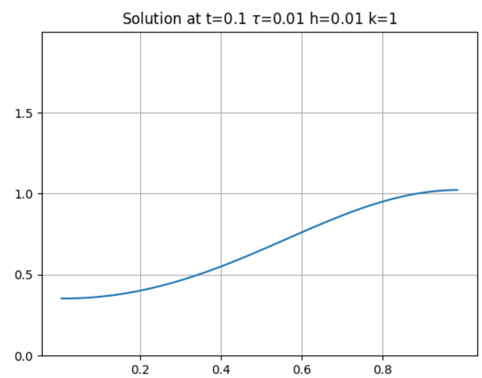
```python
def find_solution_at_next_timestep(
    previous_solution: np.ndarray, h_values: np.ndarray
) -> np.ndarray:
    LHS_matrix = sparse.csr_matrix(build_LHS_matrix(h_values))
    RHS_vector = build_RHS_vector(h_values, tau, previous_solution)
    return spsolve(LHS_matrix, RHS_vector)[None].T
```

```python
# --- Time Stepping ---
while True:
    update_tau()
    if current_time + tau > end_time:
        break

    current_time += tau
    previous_solution = find_solution_at_next_timestep(previous_solution, h_values)
plot_solution(cell_centers, previous_solution)
```

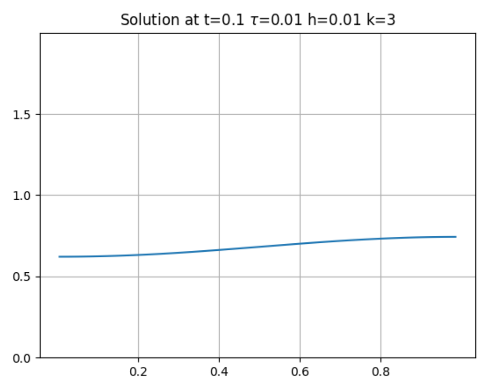
Doing this with reaction term (b) $f(u) = u(1-u)$, a uniform grid $h_j = 0.01$, $\tau = 0.01$, and various values of ε yields the following numerical solutions at time $t = 0.1$:



(a) Sol'n for $\varepsilon = 0.1$



(b) Sol'n for $\varepsilon = 1$



(c) Sol'n for $\varepsilon = 3$

Note that the figure with $\varepsilon = 1$ very closely resembles the one provided by the instructor. $\square$

Exercise 2. Consider the ODE system

$$
\begin{aligned}
u' + \alpha u^3 + c(u - v) &= g(t) && \text{(2a)}\\
v' + c(v - u) &= 0 && \text{(2b)}
\end{aligned}
$$

(i) Define a fully implicit scheme with Newton solver for (2), discuss its properties (solvability, properties of the Jacobian, ...). Implement and test when $g(t) = -\sin(4t), u(0) = 1, v(0) = 0.1$. Report on the performance of the solver depending on the data.

(ii) Suggest your own scheme which is not fully implicit and not fully explicit. Study its properties, implement, and test. Motivate. Is it better than implicit?

Solution. **(i)** We approximate $u(t_n)$ and $v(t_n)$ as U_n and V_n, respectively. Then we uniformly discretize the time domain into intervals of size τ, so that a fully implicit finite difference scheme for (2) becomes

$$
\begin{aligned}
\frac{U_n - U_{n-1}}{\tau} + \alpha U_n^3 + c(U_n - V_n) &= g(t_n) && \text{(3a)}\\
\frac{V_n - V_{n-1}}{\tau} + c(V_n - U_n) &= 0. && \text{(3b)}
\end{aligned}
$$

Solving (3b) for V_n, plugging this into (3a), and rearranging yields the following system of equations:

$$
\begin{aligned}
U_n - U_{n-1} + \alpha\tau U_n^3 + c\tau\left(U_n - \frac{V_{n-1} + c\tau U_n}{1 + c\tau}\right) - \tau g(t_n) &= 0 && \text{(4a)}\\
\frac{V_{n-1} + c\tau U_n}{1 + c\tau} &= V_n. && \text{(4b)}
\end{aligned}
$$

We solve (4a) for U_n using Newton iteration, as $U_{n-1}, V_{n-1}, c, \alpha, \tau$, and g are all known at time t_n. Then we use these to find V_n from (4b).

Newton iteration here involves finding the zeros of the function

$$
F(x) := x - U_{n-1} + \alpha\tau x^3 + c\tau\left(x - \frac{V_{n-1} + c\tau x}{1 + c\tau}\right) - \tau g(t_n).
$$

By a Theorem given in Lecture notes, this function has guaranteed *local* convergence to a root u_* if there exist $\beta, \gamma \geq 0$ such that (a) $F'(x) \neq 0$ for any x and $\left|\frac{1}{F'}\right| \leq \beta$, and (b) F' is Lipschitz with Lipschitz constant γ. We check each of these conditions separately:

(a) Computing F' yields

$$
F'(x) = 1 + 3\tau\alpha x^2 + c\tau - \frac{(c\tau)^2}{1 + c\tau} = 3\tau\alpha x^2 + \frac{1 + 2c\tau}{1 + c\tau},
$$

which is ≥ 1 if $c, \alpha \geq 0$. Thus $F'(x) \neq 0$ for any x and $\left|\frac{1}{F'}\right| \leq 1$ so long as $c, \alpha \geq 0$. Taking $\beta = 1$, (a) therefore holds.

(b) Note that $F''(x) = 6\tau\alpha x$, which is bounded on any bounded subset of $\mathbb{R}$. The Mean Value Theorem therefore implies that, in any neighborhood N of u_*, F' is Lipschitz with Lipschitz constant $6\tau\alpha \sup_{x \in N} |x|$. Taking this to be γ, (b) therefore holds, and local convergence is therefore guaranteed.

We implement the scheme in the obvious way:

```python
def F(x: float) -> float:
    return (
        x
        - u_prev
        + a * tau * math.pow(x, 3)
        + c * tau * (x - (v_prev + c * tau * x) / (1 + c * tau))
        - tau * g(current_time)
    )


def F_prime(x: float) -> float:
    return (
        1
        + 3 * a * tau * math.pow(x, 2)
        + c * tau
        - (math.pow(c * tau, 2) / (1 + c * tau))
    )


def find_v_from_u(u: float) -> float:
    return (v_prev + c * tau * u) / (1 + c * tau)


def one_newton_iteration(last_iterate: float) -> float:
    correction = -F(last_iterate) / F_prime(last_iterate)
    return last_iterate + correction
```

```python
# --- Time Stepping ---
while current_time < end_time:
    if current_time + tau > end_time:
        break
    current_time += tau

    last_iterate = u_prev
    for i in range(iteration_depth):
        last_iterate = one_newton_iteration(last_iterate)
    u_prev = last_iterate
    v_prev = find_v_from_u(u_prev)
```

(ii) Motivated by convexity-splitting, we develop a scheme which treats the linear terms in (2) explicitly and the cubic term implicitly:

$$\frac{U_n - U_{n-1}}{\tau} + \alpha U_n^3 + c(U_{n-1} - V_{n-1}) \quad = \quad g(t_{n-1}) \tag{5a}$$

$$\frac{V_n - V_{n-1}}{\tau} + c(V_{n-1} - U_{n-1}) \quad = \quad 0. \tag{5b}$$

We can easily solve (5b) for V_n, but (5a) will require Newton iteration to find the roots of the function

$$F(x) := x - U_{n-1} + \tau\alpha x^3 + c\tau(U_{n-1} - V_{n-1} - g(t_{n-1}).$$

As in (i), we prove local convergence of Newton iteration through two conditions:

(a) Here $F'(x) = 1 + 3\tau\alpha x^2$, which is ≥ 1 if $\alpha \geq 0$. We may therefore take $\beta = 1$ so that condition (a) holds.

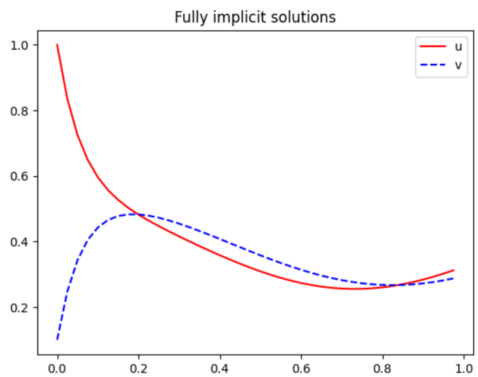
(b) Again $F''(x) = 6\tau\alpha x$, which is bounded on any bounded subset of $\mathbb{R}$. So, we can take $\gamma = 6\tau\alpha \sup_{x\in N} |x|$ for any neighborhood N of u_*, so that (b) holds just as in (i). Local convergence is therefore guaranteed.

Again, implementation is obvious:

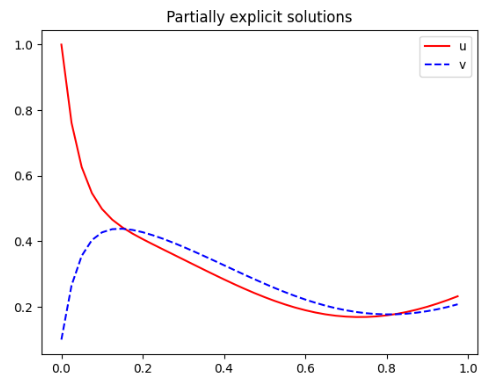
```python
def F(x: float) -> float:
    return (
        x
        - u_prev
        + a * tau * math.pow(x, 3)
        + c * tau * (u_prev - v_prev)
        - tau * g(current_time)
    )


def F_prime(x: float) -> float:
    return 1 + 3 * a * tau * math.pow(x, 2)


def get_v_from_previous_values() -> float:
    return v_prev - c * tau * (v_prev - u_prev)


def one_newton_iteration(last_iterate: float) -> float:
    correction = -F(last_iterate) / F_prime(last_iterate)
    return last_iterate + correction
```

```python
while current_time < end_time:
    if current_time + tau > end_time:
        break
    current_time += tau

    last_iterate = u_prev
    for i in range(iteration_depth):
        last_iterate = one_newton_iteration(last_iterate)
    u_prev = last_iterate
    v_prev = get_v_from_previous_values()
```

Implementing the two schemes and testing with $g(t) = -\sin(4t), u(0) = 1, v(0) = 0.1, \tau = 0.025$ yields very similar results, indicating that neither scheme produces results that differ significantly enough to see with the naked eye:[1]



(a) The fully implicit solution from (i)



(b) The partially explicit solution from (ii)

The fully implicit scheme from (i) is somewhat slower, though, taking approximately 221 microseconds longer on the MacBook used. Seeing as the partially explicit scheme from (ii) is also slightly easier to implement, this makes it preferable (in the absence of accuracy comparisons) to the fully implicit scheme. □

[1] An interested reader should manufacture solutions u and v (reverse-engineering the necessary function g) and test accuracy empirically using a p-norm.