MTH 654-2024 Homework B. **Noah Prentice**
Date: December 3, 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]

[a] URL: https://github.com/NoahPrentice/Nonlinear-Coupled-PDE-MTH654-F24/tree/main/B

Exercise 1.

(a) Consider the data $k = 10^{-12}[m^2]$, $\mu = 10^{-3}[Pa \cdot s]$ for incompressible Darcy flow due to pressure boundary conditions and some $\Delta p = p_{right} - p_{left}$. Answer: What Δp must be applied for the flow to have velocity (i) $1[mm/day]$, (ii) $1[mm/s]$, (iii) $1[m/s]$?

To see which (if any) is realistic, compare Δp to the difference of pressures ΔP_{grav} which balances the gravity $\rho G \Delta D$ for $\Delta D = 1[m]$, with $\rho \approx 10^3[kg/m^3]$, and $G \approx 10[m/s^2]$.

(b) Consider the flow field $q(x, y)$ given below, and project q to the field Q_h on a $M_x \times M_y$ grid over $(-1, 1)^2$. Answer: Is q divergence free? Is Q_h also discrete-divergence free ? (Note $Q_h \in V_h$, the discrete space of velocities from Raviart-Thomas space on quadrangles obtained when solving elliptic PDE with FV/CCFD).

$$q(x, y) = (1, 2); \quad q(x, y) = (y, -x); \quad q(x, y) = (x, y); \tag{1a}$$
$$q(x, y) = (x^2 y, -xy^2); \quad q(x, y) = (x^2 y + e^y, -xy^2 + e^x) \tag{1b}$$

If possible, construct your own $q(x, y)$ for which $\nabla \cdot q = 0$, but $\nabla_h \cdot Q_h \neq 0$.

(c) Following the class discussion, consider the transport model $(cu)_t + (qu)_x = 0$. Let c be found from some coupled model, so that in practice you work with $\tilde{c}$ rather than with c. Estimate the modeling error depending on $c - \tilde{c}$. What if also q is replaced by $\tilde{q}$.

Solution.

(a) Since the data is given in SI units, we will work in SI units. We supress the units of quantities when they are in SI units.

(i) For $q = 1[\text{mm/day}] = 1.16 \cdot 10^{-8}$, the incompressible Darcy flow model yields, for $L = 1$,

$$\begin{aligned} q &= -\frac{k\Delta p}{\mu L} \\ \implies \quad \Delta p &= -\frac{\mu L q}{k} \\ &= -\frac{(10^{-3})(1)(1.16 \cdot 10^{-8})}{10^{-12}} \\ &= \boxed{-11.57[\text{Pa}].} \end{aligned}$$

(ii) For $q = 1[\text{mm/s}] = 10^{-3}$ and $L = 1$,

$$\Delta p = -\frac{\mu L q}{k} = -\frac{(10^{-3})(1)(10^{-3})}{10^{-12}} = \boxed{-10^6[\text{Pa}].}$$

(iii) For $q = 1[\text{m/s}] = 1$ and $L = 1$,

$$\Delta p = -\frac{\mu L q}{k} = -\frac{(10^{-3})(1)(1)}{10^{-12}} = \boxed{-10^9[\text{Pa}].}$$

Given the quantities above, we compute

$$\Delta P_{grav} = \rho G \Delta D = (10^3)(10)(1) = \boxed{10^4}.$$

Since this is at least 2 orders of magnitude away from any of the pressures in (i)-(iii), none of the results seem particularly realistic.

(c) Suppose we are given $\tilde{c} \approx c$ and we find $\tilde{u} \approx u$, where

$$\begin{aligned}(cu)_t + (qu)_x &= 0\\ (\tilde{c}\tilde{u})_t + (q\tilde{u})_x &= 0\end{aligned}$$

Taking the difference of these equations and letting $E = u - \tilde{u}$ denote the modeling error yields

$$\begin{aligned} & \big[(cu)_t - (\tilde{c}\tilde{u})_t\big] + \big[q(u-\tilde{u})\big]_x = 0\\ \implies \quad & \big[(cu)_t - (\tilde{c}u)_t + (\tilde{c}u)_t - (\tilde{c}\tilde{u})_t\big] + \big[q(u-\tilde{u})\big]_x = 0\\ \implies \quad & (\tilde{c}E)_t + (qE)_x = -\big[(c-\tilde{c})u\big]_t,\end{aligned}$$

so that the error source term is $-\big[(c-\tilde{c})u\big]_t$.

If, in addition, q is approximated by $\tilde{q}$, then similar calculations yield

$$(\tilde{c}E)_t + (\tilde{q}E)_x = -\big[(c-\tilde{c})u\big]_t - \big[(q-\tilde{q})u\big]_x,$$

so that the error source term is $-\big[(c-\tilde{c})u\big]_t - \big[(q-\tilde{q})u\big]_x$. □

Exercise 2. Implement the transport model to simulate

$$\partial_t(cu) + \nabla \cdot (qu) = 0,\ x \in \Omega, 0 < t \le T_{end}; u(x,0) = u_{init}(x);\ u|_{\Gamma_{in}} = 0. \tag{2}$$

You can start by extending the code given in `TRANSPORT1d.m` to the grid $M_x \times M_y$. **Example to turn in:** Produce the numerical solutions for the field Q_h, a projection of i) $q(x,y) = (-1,-1)$, (ii) $q(x,y) = (y,-x)$. Let $c \equiv 1$. Let U_{init} be a projection of $u_{init}(x) = \chi_{\Omega_{blob}}(x)$, where Ω_{blob} is located by the following code

```
midpointx = floor([nx/3*2,nx/6*5]);midpointy=floor([ny/3*2,ny/6*5]);
uinit=zeros(nx,ny,1);uinit(midpointx(1):midpointx(2),midpointy(1):midpointy(2))=1;
```

Please **plot your solutions** for (i), (ii) at $t = 1, t = 2, t = 3$ for grid 50x50, 100x50, and 100x100. **Answer** the following questions based on your implementation:

1. Case (i). At what time t does your approximation to $u(x,t)$ essentially disappear for grid 50x50? What about for grid 100x50? 100x100?
2. Case (ii). At what time t does your approximation to the total amount $\int_\Omega u(x,t)dx$ start to differ significantly (more than 10^{-4}) from $\int_\Omega u_{init}(x)dx$ for grid 50x50? What about for grid 100x100?

Based on these answers, do you think the Godunov scheme in 2D has the properties you would desire? (What would you expect from the true solution?)

Solution. The main components of the algorithm operate as follows:

Within `Ex2_constructors.py`

```python
def build_left_right_fluxes(
    x_boundaries: np.ndarray, y_boundaries: np.ndarray, previous_solution: np.ndarray
) -> np.ndarray:
    left_right_fluxes = build_left_right_velocities(x_boundaries, y_boundaries)
    for i in range(left_right_fluxes.shape[0]):
        for j in range(left_right_fluxes.shape[1]):
            if left_right_fluxes[i, j] >= 0 and i - 1 >= 0:
                left_right_fluxes[i, j] *= previous_solution[i - 1, j]
            elif left_right_fluxes[i, j] <= 0 and i + 1 < left_right_fluxes.shape[0]:
                left_right_fluxes[i, j] *= previous_solution[i, j]
            else:  # Inflow boundary
                left_right_fluxes[i, j] = 0
    return left_right_fluxes


def build_up_down_fluxes(
    x_boundaries: np.ndarray, y_boundaries: np.ndarray, previous_solution: np.ndarray
) -> np.ndarray:
    up_down_fluxes = build_up_down_velocities(x_boundaries, y_boundaries)
    for i in range(up_down_fluxes.shape[0]):
        for j in range(up_down_fluxes.shape[1]):
            if up_down_fluxes[i, j] >= 0 and j - 1 >= 0:
                up_down_fluxes[i, j] *= previous_solution[i, j - 1]
            elif up_down_fluxes[i, j] <= 0 and j + 1 < up_down_fluxes.shape[1]:
                up_down_fluxes[i, j] *= previous_solution[i, j]
            else:  # Inflow boundary
                up_down_fluxes[i, j] = 0
    return up_down_fluxes
```

Within `Ex2_main_algorithms.py`

```python
def find_tau_from_CFL(
    initial_tau: float,
    x_boundaries: np.ndarray,
    y_boundaries: np.ndarray,
```

```
    x_widths: np.ndarray,
    y_widths: np.ndarray,
) -> float:
    left_right_velocities = build_left_right_velocities(x_boundaries, y_boundaries)
    up_down_velocities = build_up_down_velocities(x_boundaries, y_boundaries)

    nx = up_down_velocities.shape[0]
    ny = left_right_velocities.shape[1]
    left_right_velocities_at_centers = np.zeros((nx, ny))
    up_down_velocities_at_centers = np.zeros((nx, ny))

    for i in range(nx):
        left_right_velocities_at_centers[i, :] = 0.5 * (
            left_right_velocities[i, :] + left_right_velocities[i + 1, :]
        )
    for j in range(ny):
        up_down_velocities_at_centers[:, j] = 0.5 * (
            up_down_velocities[:, j] + up_down_velocities[:, j + 1]
        )
    speeds_at_centers = np.sqrt(
        np.square(left_right_velocities_at_centers)
        + np.square(up_down_velocities_at_centers)
    )

    vccof = np.ones((nx, ny))
    for i in range(nx):
        for j in range(ny):
            vccof[i, j] *= x_widths[i] * y_widths[j]
    datcfl = np.divide(vccof, speeds_at_centers)
    dtcfl = np.min(datcfl)

    if initial_tau > 0:
        print("Time step is chosen by CFL as " + str(min(0.5 * dtcfl, initial_tau)))
        return min(0.5 * dtcfl, initial_tau)
    print("Time step is chosen by CFL as " + str(0.5 * dtcfl))
    return 0.5 * dtcfl


def find_solution_at_next_timestep(
    previous_solution: np.ndarray,
    x_boundaries: np.ndarray,
    y_boundaries: np.ndarray,
    x_widths: np.ndarray,
    y_widths: np.ndarray,
    tau: float,
) -> np.ndarray:
    left_right_fluxes = build_left_right_fluxes(
        x_boundaries, y_boundaries, previous_solution
    )
    up_down_fluxes = build_up_down_fluxes(x_boundaries, y_boundaries, previous_solution)

    for i in range(previous_solution.shape[0]):
        for j in range(previous_solution.shape[1]):
            previous_solution[i, j] -= (tau / x_widths[i]) * (
                left_right_fluxes[i + 1, j] - left_right_fluxes[i, j]
            )
            previous_solution[i, j] -= (tau / y_widths[j]) * (
                up_down_fluxes[i, j + 1] - up_down_fluxes[i, j]
            )
    return previous_solution
```

The initial condition (for a 100×100 grid) looks as in figure 1.

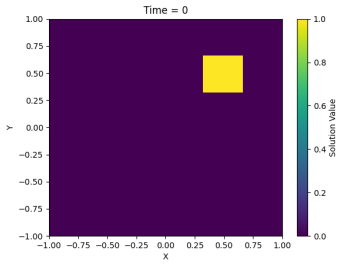
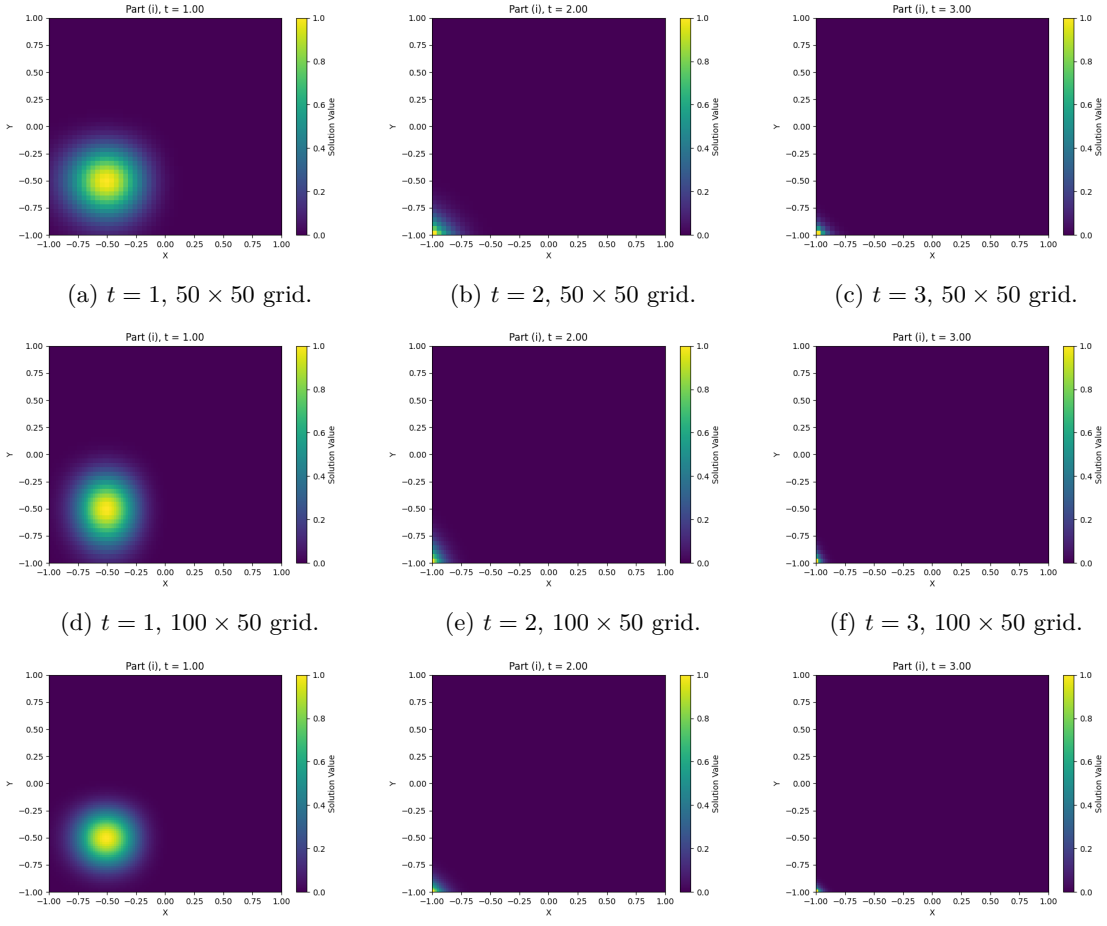


Figure 1: $u_{init}(x, y)$

(i) Running the code for $q(x, y) = (-1, -1)$, $t = 1, 2, 3$, and grid sizes $50 \times 50, 100 \times 50, 100 \times 100$ yielded the plots in figure 2.



(a) $t = 1$, 50×50 grid. (b) $t = 2$, 50×50 grid. (c) $t = 3$, 50×50 grid.

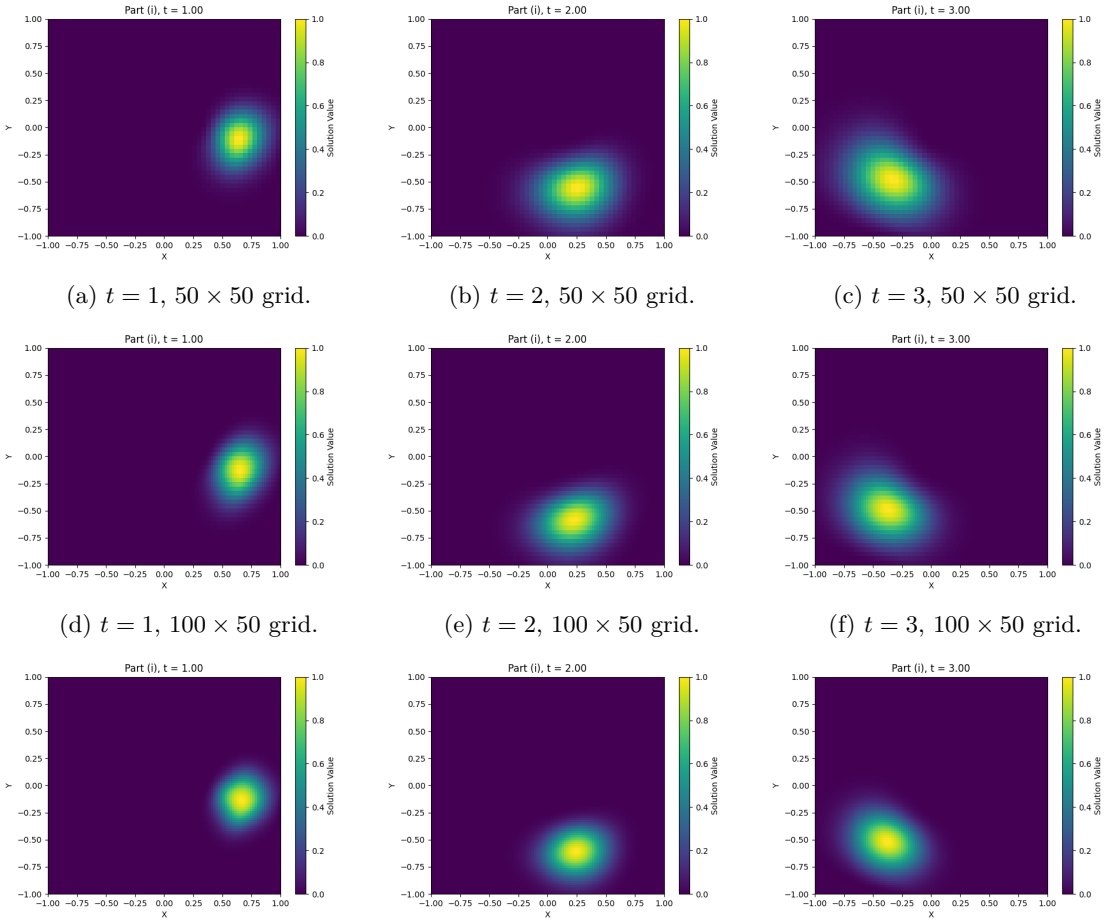
(d) $t = 1$, 100×50 grid. (e) $t = 2$, 100×50 grid. (f) $t = 3$, 100×50 grid.

(g) $t = 1$, 100×100 grid. (h) $t = 2$, 100×100 grid. (i) $t = 3$, 100×100 grid.

Figure 2: Plots for $q(x, y) = (-1, -1)$.

Upon further investigation of plots at intermittent time values, the quantity being advected effectively disappears from the $(-1, 1)^2$ range at around $t = 1.8$ for all grid sizes.

(ii) Running the code for the same t values and grid sizes, but with $q(x, y) = (y, -x)$, yielded the plots in figure 3. By estimating the quantity $\int_\Omega u(t)\, dx$ at each time t, we observe that the quantity differs from



(a) $t = 1$, 50×50 grid. (b) $t = 2$, 50×50 grid. (c) $t = 3$, 50×50 grid.

(d) $t = 1$, 100×50 grid. (e) $t = 2$, 100×50 grid. (f) $t = 3$, 100×50 grid.

(g) $t = 1$, 100×100 grid. (h) $t = 2$, 100×100 grid. (i) $t = 3$, 100×100 grid.

Figure 3: Plots for $q(x, y) = (y, -x)$.

that at $t = 0$ by more than 10^{-4} at time 0.379 (iteration #670) for a 50×50 grid and time 0.474 (iteration #3350) for a 100×100 grid.

These results are underwhelming. As indicated by the numerical diffusion seen in the plot, the Godunov scheme in 2D is very dispersive, while we would expect no diffusion of the true solution. This likely impacts the scheme's performance in the tests done above. For instance, in part (i), the true solution leaves the region $(-1, 1)^2$ at time $t = 5/3 \approx 1.66$[1] while the numerical solution takes slightly more time to effectively disappear. Then, in part (ii), the numerical diffusion clearly impacts the quantity $\int_\Omega u(t)\, dx$ as t increases. This quantity is not conserved by the scheme, which is undesirable as the quantity is conserved for the true solution. □

[1]This was computed from the fact that the initial condition is a characteristic function and the speed of flow is 1 in both the x and y directions.