Project 4, Part 2 Scientific Computing 2021

Kimi Cardoso Kreilgaard (twn176)

November 14, 2021

1 Problem Description

The project concerns simulating a reaction diffusion process, describebd by the following nonlinear partial differential equations:

$$\frac{dp}{dt} = D_p \nabla p + p^2 q + C - (K+1)p \tag{1}$$

$$\frac{dq}{dt} = D_q \nabla q - p^2 q + Kp \tag{2}$$

The coupled partial differential equations include the functions $p(x, y, t)$ and $q(x, y, t)$, each describing the concentration of a molecule p or q in space and time. We are considering solutions on the square formed by x and y being between 0 and 40. The square is $[0, 40] \cdot [0, 40]$ meaning that the square has dimensions 41×41. The Laplacian is given by ∇ and the positive diffusion coefficients are given by d_p and d_q. The two other parameters C, K are both larger than zero. I suspect K is the kill rate, i.e. a constant reaction changing p to q. C must be the feed rate, i.e. a constant adding of p molecules in time.

The initial values are given by:

- $p(x, y, t = 0) = C + 0.1$ for all (x, y) belonging to the square $10 < x < 30$ and $10 < y < 30$.
- $q(x, y, t = 0) = (K/C) + 0.2$ for all (x, y) belonging to the square $10 < x < 30$ and $10 < y < 30$.

There are no-flux Neumann boundary conditions on the boundary of the square. Neumann boundary conditions specify values of the directional derivative on the boundary where n denotes the outward unit normal vector. The flux out of the square in the direction of the unit normal vector is zero for all times on the square boundary.

We will use the values $D_p = 1$, $D_q = 8$, $C = 4.5$ and use six different values of K, ($K = 7, 8, 9, 10, 11$ and 12).

2 Solution

2.1 No flux Neunmann Boundary Conditions and Ghosts

Before we can start simulating we need to set up the problem just right. We are told that the box is 41×41 which we can represent rather simply with two matrices, one containing concentrations of p and

the other containing concentrations of q. Thinking ahead however we know that we need to implement the no-flux Neunmann boundary conditions, which tells us that the outward flux is zero, i.e. nothing is to leave the box. One way of doing this with a discrete Laplacian, is to add ghost elements to our box. Ghosts elements are placed around the borders, thus expanding the full matrices (one for p and one for q) to size 43×43 as can be seen in figure 1 below. If we let the value of each ghost element in the full matrix be equal to the element on the inside of the border (directly across following the outflow vector n), the flow through the border (outflow) will be none. The ghost elements are added when setting up the problem with a 43×43 matrix. The function `update_ghosts` ensures that all ghost elements takes on the value of the element inside the border, and the function should be used in each iteration of the simulation.

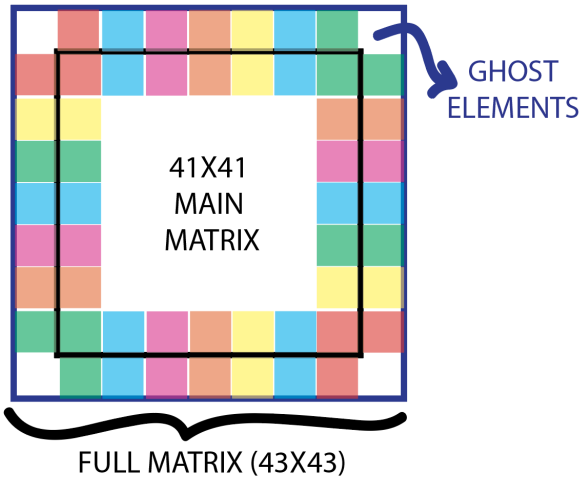

Figure 1: Illustration describing the addition of ghost elements to the original 41×41 matrix, making the full matrix 43×43. The value of each ghost element should be equal to the value of the element inside the border, here illustrated which corresponding colors.

Now we just need a function that calculates the Laplacian to ensure that the boundary conditions are upheld. This is because only the term containing the Laplacian in the coupled differential equations (given in eq. 1 and 2) describe diffusion. Thus it is only this term that could account for outflow. The function `Laplacian` calculates the Laplacian given a matrix and a step size $step_size = dx = dy$ (we let the step size along each axis be of equivalent sizes), with a finite difference approximation described on p. 454 [1]. Implementing the Laplacian for this problem, we get the following formula (this example calculates ∇p but exchaning all p's for q's will give you ∇q):

$$\frac{\partial^2 p}{\partial x^2}|_{x=L} = \frac{p(L-dx) - 2p(L) + p(L+dx)}{(dx)^2} \tag{3}$$

$$\frac{\partial^2 p}{\partial y^2}|_{x=L} = \frac{p(L-dy) - 2p(L) + p(L+dy)}{(dy)^2} \tag{4}$$

$$\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} \tag{5}$$

This formula calculates each gridpoint in space individually and will lead to a slow implementation. Introducing submatrices we can instead calculate the Laplacian with array programming thus only using a for loop for each time iteration. When calculating the finite difference Laplacian for one point, we

consider the neighbouring points (one to the left, one to the right, one above and one below). We therefor introduce the submatrices `Left`, `Right`, `Up` and `Up`, which all have the same size 41×41 and are shown schematically in figure 2 below. Now the Laplacian for all points in one of the matrices can be calculated

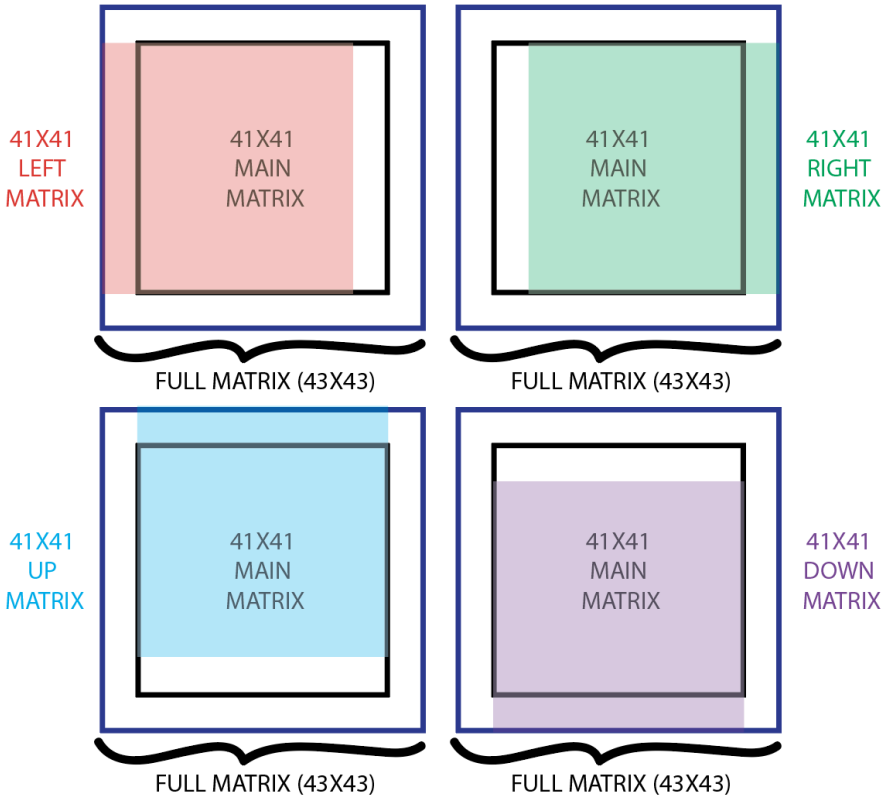

Figure 2: matrices

at the same time, using:

$$\nabla \mathbf{p} = \frac{1}{\texttt{step\_size}^2} \cdot (\texttt{Left} + \texttt{Right} + \texttt{Up} + \texttt{Down} - 4\texttt{Main}) \tag{6}$$

2.2 Update of State for Each Time Step Using the Coupled Diff. Equations

We create the function `update_state` which updates the p and q full matrices one time step dt. This function takes the parameter values of D_p, D_q, C and K along with the full matrices for p and q, calculates the laplacian with `laplacian` and finds $\frac{dp}{dt}$ and $\frac{dq}{dt}$ from the coupled differential equations given in eq. 1 and 2. Using a forward Euler method (described on p. 391 [1]), the function calculates the next matrices by:

$$p(t + dt) = p(t) + dt \cdot \frac{dp}{dt} \tag{7}$$

$$q(t + dt) = q(t) + dt \cdot \frac{dq}{dt} \tag{8}$$

2.3 Initial Values and Setting up the Simulation

When setting up the problem from the initial values, we first need to choose which resolution we will use, i.e. choose $dx = dy$. The matrices we will work with will thus have dimensions $\frac{43}{dx} \times \frac{43}{dx}$. And the initial

conditions correspondingly needs to be scaled so we work with matrix indices rather than values of x and y. The initial conditions become:

- $p(x, y, t = 0) = C + 0.1$ for all matrix indices belonging to the square $\frac{11}{dx} < i < \frac{31}{dx}$ and $\frac{11}{dx} < j < \frac{31}{dx}$. Notice we use 11 and 31 instead of 10, since we are working with matrix indices and index 0 belongs to ghost elements not included in the problem formulation.
- $q(x, y, t = 0) = (K/C) + 0.2$ for all matrix indices belonging to the square $\frac{11}{dx} < i < \frac{31}{dx}$ and $\frac{11}{dx} < j < \frac{31}{dx}$.

We can now simulate the reaction diffusion proces by putting all the steps below together:

- Choose dx and make a p and q matrix with proper dimenstions (including ghosts).
- Define the parameter values of D_p, D_q, C and K.
- Fill out the matrices with the initial values, by using the initial conditions for matrix indices and Python slice notation.
- Choose a suitable time step size dt to ensure a stable solution. This can be found from the stability region for Euler's method. For a almost similar problem, the book suggest you choose $dt \leq \frac{(dx)^2}{2C}$ (p. 458 [1]). Since our problem is a little different and K is usually the highest value, we instead choose $dt \leq \frac{(dx)^2}{2K}$.
- Choose a value of t you want to simulate until, meaning you will run $\frac{t}{dt}$ iterations of the simulation. In each iteration, we need to:
  - Update ghost elements using `update_ghosts`.
  - Update the matrices (in time) using `update_state`.

2.4 Results

Using the suggested parameter values of $D_p = 1$, $D_q = 8$, $C = 4.5$ and $K = 9$. I choose $dx = 0.5$ and $dt = 0.001$. We can now simulate the reaction diffusion process up until t=2000, resulting in $t/dt = 2000/0.001 = 2000000$ time iterations. This ends up taking around 10 mins, but seems necessary since $dx = 1$ results in results for all K's being almost similar. I don't think it can be done much faster, since there is only a for loop in the time iteration, while all space iterations are made with array programming. I have choosen a 12 of the 2 million iterations of the reaction, to freeze and show a frame of. These can be seen in figure 3 below. Notice that the process seems to converge before $t = 2000$ since we obtained the same image for $t = 1000$. Since the initial values were symmetrical I would assume the end results to be so to, and they are a long part of the way up until t=300 at least, before it becomes unsymmetrical. I believe this is a product of the forward Euler method used for the time iterations, which allows errors to acumulate. Thus a small error could result in this unsymmetrical pattern. A way to solve this would be to increase the resolution and the iterations, but this didn't seem necessary for the

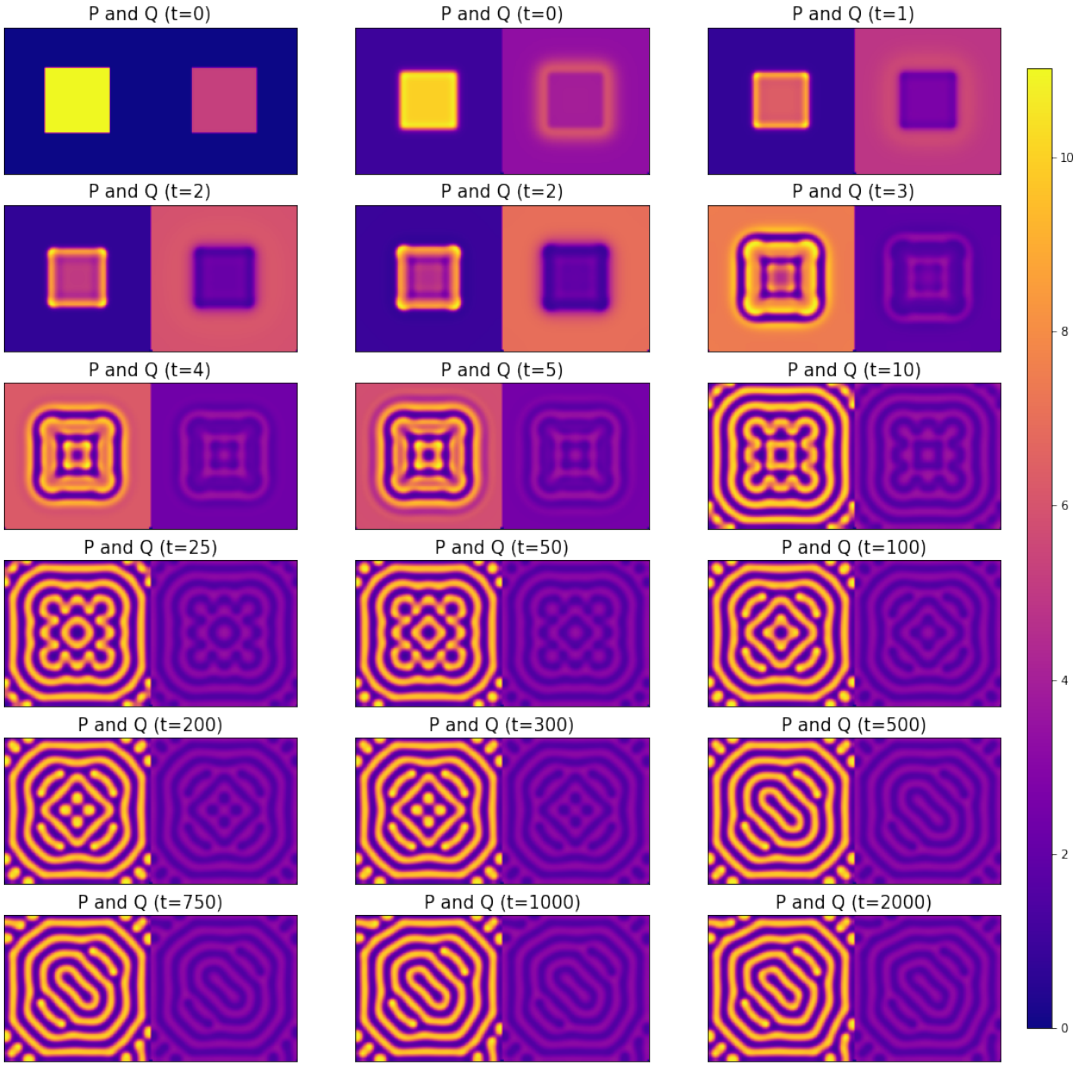

Figure 3: Frames from the simulation with K=9

assignment. Lastly I need to present contour plots of the solutions $p(x, y)$ and $q(x, y)$ at $t = 2000$ for six different values of K ($K = 7, 8, 9, 10, 11$ and 12). I repeat the configurations used above, except for changing K. I save the last frame of each simulation for $t = 2000$ and these are plottet together in figure 4 below. The solutions are "almost" symmetrical but it does seem there is a slight problem here as well, but with this figure taking over an hour to produce I did not wish to investigate further. Overall we see that local concentrations of p are higher for higher K's, and that there in general is more p than q, which is after all what we expect. One way we could improve the solution but still using "only an hour" is if we assume from figure 3 above that the solutions converge after $t = 750$, we could decrease the time step size

but allow for more iterations with the same run time. This might fix the problem of symmetries.

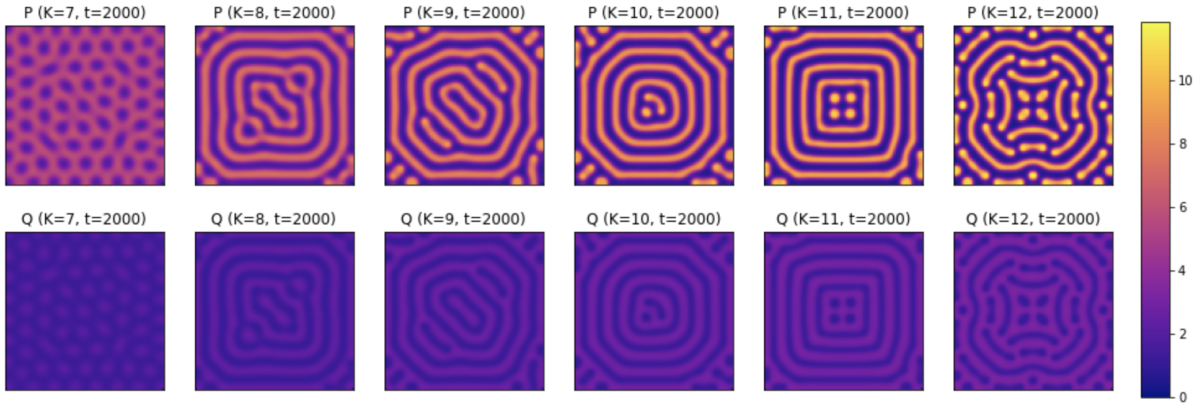

Figure 4: Simulation results for 6 different K's.

References

[1] Michael T. Heath, *Scientific Computing: An Introductory Survey*, 2nd Ed.