Assignment

Math3101/5305 Term 3 2022

Due: 9AM Monday 21 November

Instructions. The assignment has two parts. Part A is for both Math3101 and Math5305, whereas Part B is for Math5305 only. Create a folder to hold your assignment files and, when you are ready to submit the assignment, create a zip archive from the folder and upload the archive via the assignment activity on Moodle. To save space, do not include any files generated by your code. (You can include plots in the pdf with your written answers). I will run your scripts as necessary to look at any outputs.

Be sure to read Section 6 carefully before attempting the programming questions in the assignment.

Marking scheme. Part A is worth 50 marks, as follows.

Question	1	2	3	4	5	6	7	8	9	10
Marks	4	2	4	4	6	8	4	6	4	8

Part B is worth a further 20 marks, as follows.

Question	11	12	13	14
Marks	8	8	2	2

In addition, there are 10 marks for the overall quality of the code and of the written solutions (correct spelling and grammar, typesetting of the equations, etc.). Thus, the final mark is out of 60 for Math3101, and out of 80 for Math5305.

Part A: both Math3101 and Math5305

1 Reaction–diffusion equation

A PDE of the form

$$u_t - \kappa \nabla^2 u = f(u) \tag{1}$$

is called a *reaction–diffusion equation*. Notice that the inhomogeneous term is a given function of the dependent variable u, in contrast to the heat equation in which f is a known function of the spatial variables and time. The *reaction term* $f(u)$ is typically nonlinear.

Suppose that we want to solve (1) in a 2D spatial domain $\Omega = (0, L_x) \times (0, L_y)$ for $0 \le t \le T$, so that $u = u(x, y, t)$. We assume homogeneous Neumann boundary conditions,

$$\frac{\partial u}{\partial n}(x, y, t) = 0 \quad \text{for } (x, y) \in \partial\Omega \text{ and } 0 \le t \le T,$$

and an initial condition

$$u(x, y, 0) = u_0(x, y) \quad \text{for } (x, y) \in \Omega.$$

To compute a numerical solution, we set up a spatial finite difference grid

$$(x_p, y_q) = (p\,\Delta x, q\,\Delta y) \quad \text{for } 0 \le p \le P \text{ and } 0 \le q \le Q, \text{ where } \Delta x = \frac{L_x}{P} \text{ and } \Delta y = \frac{L_y}{Q},$$

and time levels

$$t_n = n\,\Delta t \quad \text{for } 0 \le n \le N, \text{ where } \Delta t = \frac{T}{N}.$$

Consider the implicit Euler approximation $U_{pq}^n \approx U(x_p, y_q, t_n)$, defined by

$$\frac{U_{pq}^n - U_{pq}^{n-1}}{\Delta t} - \kappa\left(\frac{U_{p+1,q}^n - U_{pq}^n + U_{p-1,q}^N}{\Delta x^2} + \frac{U_{p,q+1}^n - U_{pq}^n + U_{p,q-1}^n}{\Delta x^2}\right) = f(U_{pq}^n) \tag{2}$$

for $0 \le p \le P$ and $0 \le q \le Q$, subject to the discrete boundary conditions

$$\frac{U_{-1,q}^n - U_{+1,q}^n}{\Delta x} = \frac{U_{P+1,q}^n - U_{P-1,q}^n}{\Delta x} = 0 \quad \text{for } 0 \le q \le Q,$$
$$\frac{U_{p,-1}^n - U_{p,+1}^n}{\Delta y} = \frac{U_{p,Q+1}^n - U_{p,Q-1}^n}{\Delta y} = 0 \quad \text{for } 0 \le p \le P,$$

and the discrete initial condition

$$U_{pq}^0 = u_0(x_p, y_q) \quad \text{for } 0 \le p \le P \text{ and } 0 \le q \le Q.$$

We have used "ghost" grid points $x_{-1} = -\Delta x$, $x_{P+1} = L_x + \Delta x$, $y_{-1} = -\Delta y$ and $y_{Q+1} = L_y + \Delta y$, but the boundary conditions allow us to eliminate the corresponding unknowns:

$$U_{-1,q}^n = U_{+1,q}^n, \quad U_{P+1,q}^n = U_{P-1,q}^n, \quad U_{p,-1}^n = U_{p,+1}^n, \quad U_{p,Q+1}^n = U_{p,Q-1}^n.$$

In the usual way, for each n we define a vector $\boldsymbol{U}^n$ of dimension $M = (P+1)(Q+1)$ by stacking the U_{pq}^n in column-major order, so that

$$U_k^n = U_{pq}^n \quad \text{for } k = p + 1 + q(P+1) \text{ where } 0 \le p \le P \text{ and } 0 \le q \le Q. \tag{3}$$

The finite difference equations (2) are then equivalent to an $M \times M$ (nonlinear) system of equations

$$\frac{\boldsymbol{U}^n - \boldsymbol{U}^{n-1}}{\Delta t} + \boldsymbol{A}\boldsymbol{U}^n = \boldsymbol{f}(\boldsymbol{U}^n), \tag{4}$$

for a sparse matrix $\boldsymbol{A}$ and a vector function $\boldsymbol{f}$ with components $f_k(\boldsymbol{U}^n) = f(U_k^n)$. The matrix has the form

$$\boldsymbol{A} = \boldsymbol{I}_y \otimes \boldsymbol{A}_x + \boldsymbol{A}_y \otimes \boldsymbol{I}_x \tag{5}$$

where $\boldsymbol{I}_x$ and $\boldsymbol{I}_y$ are the $(P+1)\times(P+1)$ and $(Q+1)\times(Q+1)$ identity matrices, respectively, and where

$$\boldsymbol{A}_x = \frac{\kappa}{\Delta x^2}\begin{bmatrix} 2 & -2 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -2 & 2 \end{bmatrix} \in \mathbb{R}^{(P+1)\times(P+1)} \tag{6}$$

and

$$\boldsymbol{A}_y = \frac{\kappa}{\Delta y^2}\begin{bmatrix} 2 & -2 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -2 & 2 \end{bmatrix} \in \mathbb{R}^{(Q+1)\times(Q+1)}.$$

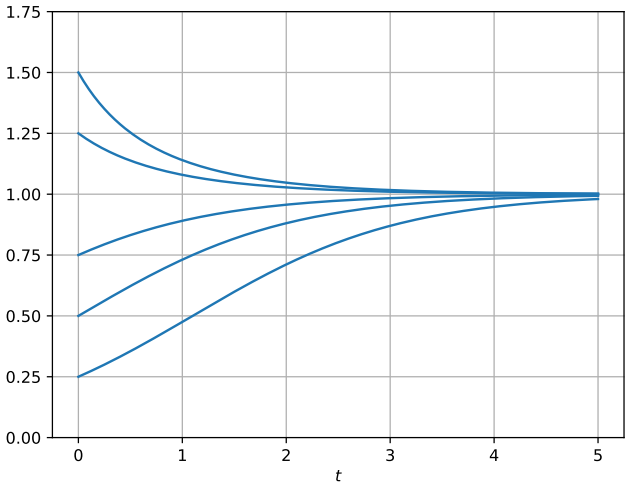

Figure 1: Solutions of the logistic equation (11) for various choices of the initial value u_0.

Rearranging (4), we see that at the nth time step we have to solve the nonlinear system

$$(\boldsymbol{I} + \Delta t\, \boldsymbol{A})\boldsymbol{V} = \boldsymbol{U}^{n-1} + \Delta t\, \boldsymbol{f}(\boldsymbol{V}) \tag{7}$$

to find $\boldsymbol{V} = \boldsymbol{U}^n$. A simpler alternative is to replace the right-hand side of (4) by $\boldsymbol{f}(\boldsymbol{U}^{n-1})$, so that

$$\frac{\boldsymbol{U}^n - \boldsymbol{U}^{n-1}}{\Delta t} + \boldsymbol{A}\boldsymbol{U}^n = \boldsymbol{f}(\boldsymbol{U}^{n-1}). \tag{8}$$

This *semi-implicit method* — also known as an implicit-explicit or IMEX method — requires that we solve only a *linear* system of equations,

$$(\boldsymbol{I} + \Delta t\, \boldsymbol{A})\boldsymbol{V} = \boldsymbol{U}^{n-1} + \Delta t\, \boldsymbol{f}(\boldsymbol{U}^{n-1}), \tag{9}$$

for $\boldsymbol{V} = \boldsymbol{U}^n$. As long as $\boldsymbol{f}(\boldsymbol{U}^n)$ differs from $\boldsymbol{f}(\boldsymbol{U}^{n-1})$ by $O(\Delta t)$, the error should still be $O(\Delta t + \Delta x^2)$. However, we can expect the accuracy to suffer in areas where $f(u)$ is changing rapidly in time.

We therefore consider using a Newton iteration to solve (7). From an initial guess $\boldsymbol{V}^{\langle 0\rangle}$ for $\boldsymbol{U}^n$, we compute iterates

$$\boldsymbol{V}^{\langle 1\rangle} = \boldsymbol{V}^{\langle 0\rangle} + \boldsymbol{\Delta V}^{\langle 0\rangle}, \quad \boldsymbol{V}^{\langle 2\rangle} = \boldsymbol{V}^{\langle 1\rangle} + \boldsymbol{\Delta V}^{\langle 1\rangle}, \quad \ldots$$

by solving the sequence of linear systems

$$\big[\boldsymbol{I} + \Delta t\, \boldsymbol{A} - \Delta t\, \mathbf{D}\boldsymbol{f}(\boldsymbol{V}^{\langle j\rangle})\big]\boldsymbol{\Delta V}^{\langle j\rangle} = \boldsymbol{U}^{n-1} + \Delta t\, \boldsymbol{f}(\boldsymbol{V}^{\langle j\rangle}) - \big(\boldsymbol{I} + \Delta t\, \boldsymbol{A}\big)\boldsymbol{V}^{\langle j\rangle} \tag{10}$$

for $j = 0, 1, \ldots$, until $\|\boldsymbol{\Delta V}^{\langle j\rangle}\|$ is smaller than some chosen accuracy tolerance. Here, $\mathbf{D}\boldsymbol{f}(\boldsymbol{V})$ denotes the Frećhet derivative of $\boldsymbol{f}$, or in other words the $M \times M$ Jacobian matrix with jk-entry $\partial f_j/\partial V_k$. One natural choice for the initial guess is to let $\boldsymbol{V}^{\langle 0\rangle}$ equal to the solution of (9), that is, the IMEX approximation to $\boldsymbol{U}^n$.

2 Fisher's equation

If, for some constant $\alpha > 0$,

$$f(u) = \alpha u(1-u)$$

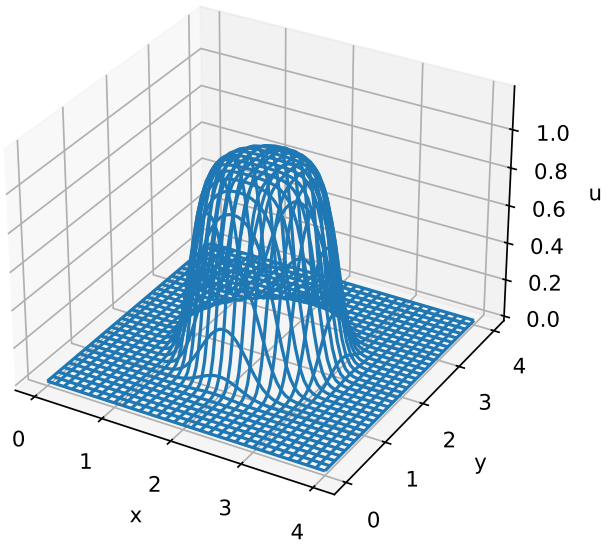

Figure 2: The reference solution at $t = 1$, used in Q10.

then (1) is known as *Fisher's equation*, and provides a simple model for the growth and spread of a population. In the special case $\kappa = 0$, when the spatial variables (x, y) become mere parameters in the problem, Fisher's equation reduces to an ordinary differential equation, namely, the *logistic equation*,

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \alpha u(1 - u). \tag{11}$$

Fig. 1 illustrates the behaviour of $u(t)$ for different choices of the initial value u_0. We see that $u(t) = 1$ is a stable equilibrium solution, and $u(t) = 0$ is an unstable equilibrium: from any initial level $u_0 > 0$ the population will tend towards 1 as $t \to \infty$. Our aim is to investigate what happens in the general case $\kappa > 0$ when the population diffuses in space (due to the term $-\kappa\nabla^2 u$) at the same time as it grows (due to the reaction term $f(u)$).

3 Questions for both Math3101 and Math5305

1. Derive the formula (10) by letting $\boldsymbol{V} = \boldsymbol{V}^{\langle j+1\rangle}$ in (7) and using the linear approximation

$$\boldsymbol{f}(\boldsymbol{V}^{\langle j\rangle} + \boldsymbol{\Delta V}^{\langle j\rangle}) \approx \boldsymbol{f}(\boldsymbol{V}^{\langle j\rangle}) + \mathbf{D}\boldsymbol{f}(\boldsymbol{V}^{\langle j\rangle})\,\boldsymbol{\Delta V}^{\langle j\rangle}. \tag{12}$$

2. Explain why $\mathbf{D}\boldsymbol{f}(\boldsymbol{V})$ is a diagonal matrix. What are the diagonal entries?

3. Solve the (separable) ODE (11) for a general initial condition $u(0) = u_0$, and verify that $u(t) \to 1$ as $t \to \infty$ for any $u_0 > 0$.

4. Write a function

```
gr = grid_points(Lx, Ly, T, P, Q, N)
```

that returns a grid data structure `gr` storing the vectors of grid points

$$\boldsymbol{x} = [x_0, x_1, \ldots, x_P], \qquad \boldsymbol{y} = [y_0, y_1, \ldots, y_Q], \qquad \boldsymbol{t} = [t_0, t_1, \ldots, t_N],$$

and the step sizes Δx, Δy and Δt.

5. Write a function

```
Ax = d2matrix(P, Dx, kappa)
```

that computes the tridiagonal matrix (6), and hence write a function

```
A = Poisson_matrix(gr, kappa)
```

that computes (5). The argument `gr` is a grid data structure of the type created in Q4, and you must ensure that `A` has a suitable *sparse* data type. Write a third function

```
f, df = Fisher(V, alpha)
```

that computes the vector $\boldsymbol{f}(\boldsymbol{V})$ and the (sparse) diagonal matrix $\mathbf{D}\boldsymbol{f}(\boldsymbol{V})$.

6. Write a function

```
U, iterations = implicit_Euler(gr, kappa, alpha, reaction, u0mat, tol, maxits)
```

that computes U_{pq}^n for $0 \le n \le N$, $0 \le p \le P$ and $0 \le q \le Q$ using the IMEX scheme (8). The argument `t` is the vector of time levels $[t_0, t_1, \ldots, t_N]$, and the argument `reaction` is a function that evaluates $\boldsymbol{f}(\boldsymbol{V})$ and $\mathbf{D}\boldsymbol{f}(\boldsymbol{V})$ for a given vector $\boldsymbol{V}$, that is,

```
V, dV = reaction(V).
```

The argument `u0mat` is the $(P+1)\times(Q+1)$ matrix of initial values $U_{pq}^0 = u_0(x_p, y_q)$. For now, ignore the arguments `tol` and `maxits`, and the second return variable `iterations`.

7. Write a script `run_implicit_Euler` that solves the Fisher equation and displays the solution as a wireframe animation, for an initial condition of the form

$$u_0(x,y) = H\exp\bigl(-[(x-C_x)^2 + (y-C_y)^2]/R^2\bigr).$$

To assist you, some files are provided on Moodle. The file `q7_params` defines all of the simulation parameters, including `height` $= H$, `centre` $= [C_x, C_y]$ and `radius` $= R$. There are also functions `snapshot` and `animate_soln`[1]. The first displays a static surface plot of the solution for a chosen time level, and may prove useful for debugging your code. The second is for generating the output of your script, namely, an animation of the solution. As well as displaying the animation as your script executes, and the function saves a copy in a video file that you should call `IMEX`.

8. Modify your `implicit_Euler` function so that it implements the Newton iteration scheme (10). Use an IMEX step to obtain the initial guess at each time step, and stop once $\|\boldsymbol{\Delta V}\|_\infty$ is less than `tol` or when the number of iterations exceeds `maxits`, whichever occurs first. The second output argument, `iterations`, is a vector of length N whose nth entry records the number of Newton iterations used for the nth time step. Write your code in such a way that setting `maxits` to zero produces the IMEX solution. The `iterations` argument is useful for checking that the Newton iteration is converging as expected.

[1]The python versions are in the file `visualisation.py`, and the Julia versions in the (incomplete) file `Assgn.jl`.

9. Modify your `run_implicit_Euler` script so that it prompts the user for the value of `maxits`. If `maxits` is zero, then save the animation to a file `IMEX` as before, but if `maxits` is positive, then save to a file `implicit_Euler`. Check the `iterations` vector. You should find that no more than 4 Newton iterations were needed for each time step.

For this question only, marks will be awarded for providing a help message (docstring) for the function, and for any appropriate code comments.

10. From lectures, we expect that

$$U_{pq}^n = u(x_p, y_q, t_n) + O(\Delta t + \Delta x^2 + \Delta y^2).$$

Write a script `convergence` that prints a table of the form

N	Error	Rate
32		
64		
128		
256		

Since no analytical solution is available, treat the implicit Euler solution with $N = 1,024$ as "exact"; see Fig. 2. Use $T = 1$, $P = 64$ and $Q = 64$ in all cases, and in the "Error" column show the maximum error at the final time $t = T$. Since P and Q are fixed, this means we are not testing the $O(\Delta x^2 + \Delta y^2)$ convergence in space, but only the $O(\Delta t)$ convergence in time.

Part B. Math5305 only

4 An implicit Runge–Kutta scheme (Math5305 only)

The implicit Euler scheme is only first-order accurate in time. We can achieve third-order accuracy using the 2-stage *Radau IIA scheme*, which is an implicit Runge–Kutta (IRK) method. At the nth time step, we must solve the nonlinear system

$$\begin{aligned}
\boldsymbol{\xi}_1 &= \boldsymbol{U}^{n-1} + \Delta t\Big(\tfrac{5}{12}[\boldsymbol{f}(\boldsymbol{\xi}_1) - \boldsymbol{A}\boldsymbol{\xi}_1] - \tfrac{1}{12}[\boldsymbol{f}(\boldsymbol{\xi}_2) - \boldsymbol{A}\boldsymbol{\xi}_2]\Big),\\
\boldsymbol{\xi}_2 &= \boldsymbol{U}^{n-1} + \Delta t\Big(\tfrac{3}{4}[\boldsymbol{f}(\boldsymbol{\xi}_1) - \boldsymbol{A}\boldsymbol{\xi}_1] + \tfrac{1}{4}[\boldsymbol{f}(\boldsymbol{\xi}_2) - \boldsymbol{A}\boldsymbol{\xi}_2]\Big),
\end{aligned}$$

to find the *stage vectors* $\boldsymbol{\xi}_1, \boldsymbol{\xi}_2 \in \mathbb{R}^M$, and then put $\boldsymbol{U}^n = \boldsymbol{\xi}_2$. Rearranging the stage equations, we have a block 2×2 system

$$\begin{aligned}
(\boldsymbol{I} + \tfrac{5}{12}\Delta t\, \boldsymbol{A})\boldsymbol{\xi}_1 - \tfrac{1}{12}\Delta t\, \boldsymbol{A}\boldsymbol{\xi}_2 &= \boldsymbol{U}^{n-1} + \tfrac{5}{12}\Delta t\, \boldsymbol{f}(\boldsymbol{\xi}_1) - \tfrac{1}{12}\Delta t\, \boldsymbol{f}(\boldsymbol{\xi}_2),\\
\tfrac{3}{4}\Delta t\, \boldsymbol{A}\boldsymbol{\xi}_1 + (\boldsymbol{I} + \tfrac{1}{4}\Delta t\, \boldsymbol{A})\boldsymbol{\xi}_2 &= \boldsymbol{U}^{n-1} + \tfrac{3}{4}\Delta t\, \boldsymbol{f}(\boldsymbol{\xi}_1) + \tfrac{1}{4}\Delta t\, \boldsymbol{f}(\boldsymbol{\xi}_2),
\end{aligned} \tag{13}$$

5 Questions for Math5305 only

11. Formulate Newton's method for the nonlinear equations (13), by writing

$$\boldsymbol{\xi}_\ell^{\langle j+1\rangle} = \boldsymbol{\xi}_\ell^{\langle j\rangle} + \boldsymbol{\Delta\xi}_\ell^{\langle j\rangle} \quad \text{for } \ell = 1,\, 2,$$

and deriving a 2×2, block linear system of the form

$$\begin{bmatrix} \boldsymbol{B}_{11}^{\langle j\rangle} & \boldsymbol{B}_{12}^{\langle j\rangle} \\ \boldsymbol{B}_{21}^{\langle j\rangle} & \boldsymbol{B}_{22}^{\langle j\rangle} \end{bmatrix} \begin{bmatrix} \boldsymbol{\Delta\xi}_1^{\langle j\rangle} \\ \boldsymbol{\Delta\xi}_2^{\langle j\rangle} \end{bmatrix} = \begin{bmatrix} \boldsymbol{c}_1^{\langle j\rangle} \\ \boldsymbol{c}_2^{\langle j\rangle} \end{bmatrix}. \tag{14}$$

Hint: $\boldsymbol{B}_{11}^{\langle j\rangle} = \boldsymbol{I} + \frac{5}{12}\Delta t\, \big[\boldsymbol{A} - \mathbf{D}\boldsymbol{f}(\boldsymbol{\xi}_1^{\langle j\rangle})\big]$.

12. Write a function

```
U, iterations = IRK(gr, t, kappa, reaction, u0mat, tol, maxits)
```

that uses the IRK scheme to compute the U_{pq}^n. The arguments have the same meaning as for the `implicit_Euler` function. To obtain the initial guess for the Newton iteration, compute $\widetilde{\boldsymbol{U}}^n$ by performing an IMEX step, and put

$$\boldsymbol{\xi}_1^{\langle 0\rangle} = \tfrac{2}{3}\boldsymbol{U}^{n-1} + \tfrac{1}{3}\widetilde{\boldsymbol{U}}^n \quad \text{and} \quad \boldsymbol{\xi}_2^{\langle 0\rangle} = \widetilde{\boldsymbol{U}}^n.$$

13. Write a script `run_IRK` that uses `IRK` to solve Fisher's equation and produces an animation of the solution. Again use the parameter values provided in `q7_params`, except for `N` = 50 and `maxits` = 10.

14. Modify your `convergence` script so that it prompts the user to choose the PDE solver by entering '1' for `implicit_Euler` or '2' for `IRK`. You should observe $O(\Delta t^3)$ convergence in the latter case.

6 Programming notes

Apart from the `implicit_Euler` function (see Q8), you are not required to include comments in your code. You are free to write additional functions if you find that helpful. For example, you could write a function to produce the matrix of initial data `u0mat`.

You should aim to ensure that your code is reasonably efficient, but will not lose marks if the performance is only a little sub-optimal. As a very rough guide, on my desktop PC[2], computing the implicit Euler reference solution in Q10 took about 15 seconds using Matlab or Julia, and about 20 seconds using Python. The IRK reference solution in Q14 took about 45 seconds using Matlab or Julia, and about 60 seconds using Python.

6.1 Matlab.

By starting a script with the line

```
q7_params
```

you can intialise all of the problem parameters, and then redefine any that need changing. In Q4, use the `struct` function to create the grid data structure, and the `kron` function to construct the sparse matrix `A`. These techniques are used in Lab 8; the only change is that we are using Neumann instead of Dirichlet boundary conditions, so you have to modify the definitions of $\boldsymbol{A}_x$ and $\boldsymbol{A}_y$. You should store U_{pq}^n as `U(p,q,n+1)`.

You will find Lab 7 useful for answering Q6. The main difference is that, since we have 2D rather than a 1D problem, you need to use the `reshape` function. For example, the function call `reshape(U(:,:,n), M, 1)` returns the vector $\boldsymbol{U}^{n-1}$, and after solving the linear system (9) you can do `U(:,:,n+1) = reshape(V, P+1, Q+1)`. In Q9, you can use the `input` function to obtain the value of `maxits` from the user.

I found that the code ran about five times faster if I disabled multithreading by calling `maxNumCompThreads(1)` in my `convergence` script.

[2]It has a Ryzen 7 3700X processor.

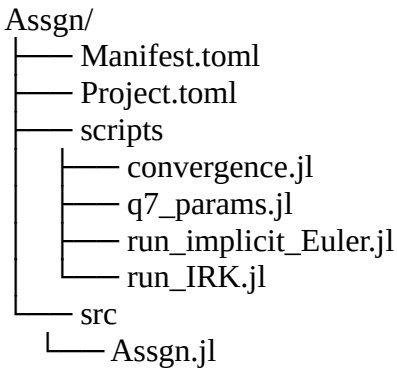

Figure 3: File layout for a Julia assignment.

6.2 Python.

Put the functions you write into a module file `assgn.py` and import them into your scripts. You can do

```
from q7_params import *
```

to initialise all of the problem parameters, and then redefine any that need changing. If you wish, you can create a Jupyter notebook for the assignment, and write the contents of each script in a code cell. (The cells will need to import functions from your module.) In this case, you may also write your answers to the theory questions as text cells in the notebook.

In Q4, the easiest way to create the grid data structure is with a named tuple, and in Q5 the easiest way to construct the sparse matrix `A` is to use the `scipy.sparse.kron` function. These techniques are used in Lab 8; the only change is that we are using Neumann instead of Dirichlet boundary conditions, so you have to modify the definitions of $\boldsymbol{A}_x$ and $\boldsymbol{A}_y$. Also, since Python uses *row*-major ordering, the formula (5) has to be replaced with

$$\boldsymbol{A} = \boldsymbol{A}_x \otimes \boldsymbol{I}_y + \boldsymbol{I}_x \otimes \boldsymbol{A}_y,$$

and U^n_{pq} should be stored as `U[n,p,q]`.

You will find Lab 7 useful for answering Q6. The main difference is that, since we have a 2D rather than just a 1D problem, you need to use the `reshape` method. For example, `U[n-1,:,:].reshape(M)` gives the vector $\boldsymbol{U}^{n-1}$, and after solving the linear system (9) you can do `U[n,:,:]=V.reshape(P+1, Q+1)`. In Q9, use the `input` function to obtain the value of `maxits` from the user.

In Q12, you should use the function `scipy.sparse.bmat` to construct the 2×2 block matrix in (14), and use `numpy.hstack` to construct the right-hand side.

6.3 Julia.

You should create a Julia project `Assgn` with a module in the `src` sub-folder to hold your functions, and a `scripts` sub-folder to hold your scripts, as shown in Fig. 3. The downloaded file `Assgn.jl` includes the functions `snapshot` and `animate_soln` as well as the type definitions for `PDE_params` and `Bump_params` needed in the file `q7_params.jl`, and a partial definition of the `Grid` data type from Q4. To complete the assignment you will need to add your functions to `Assgn.jl`.

In a script, the statement

```
include("q7_params.jl")
```

gives access the `q7_p` and `bump_p` structures, from which any parameters you need can be extracted using the `@unpack` macro, e.g.,

```
@unpack Lx, Ly = q7_p.
```

Once you complete the missing body of the constructor, you can simply do `gr = Grid(pde_p)` where `pde_p` is a copy of `q7_p` with any necessary changes. For more details, see the documentation for the `Parameters` package.

In Q5, use the `kron` function to construct the sparse matrix `A`, as in Lab 8 except with suitable modifications to $\boldsymbol{A}_x$ and $\boldsymbol{A}_y$ to handle Neumann rather than Dirichlet boundary conditions. You will find Lab 7 helpful for answering Q6. The main difference is that we have a 2D instead of just a 1D problem, so you will need to switch between treating the solution at a given time level as a $(P+1) \times (Q+1)$ matrix and as a vector of length M.

I suggest defining

```
U = OffsetVector{OffsetMatrix{Float64}}(undef, 0:N)
for n = 0:N
    U[n] = OffsetMatrix{Float64}(undef, 0:P, 0:Q)
end
```

so that U_{pq}^n is referenced as `U[n][p,q]`. In this way, doing `V=vec(U[n])` makes the vector `V` reference the same storage as the matrix `U[n]`. Hence, if `FactB` holds the Cholesky factorisation of $\boldsymbol{B} = \boldsymbol{I} + \Delta t\, \boldsymbol{A}$ and if `rhs` holds the right-hand side of the linear system (9), then doing

```
V .= FactB \ rhs
```

updates `U[n]` to hold the IMEX solution at time t_n.

In Q9, you can use the `readline` function to obtain the value of `maxits` from the user.