1 Preliminaries

The objective of this practical is to introduce numerical methods to solve systems of reaction-diffusion equations in 1D and 2D spatial domains,

$$\frac{\partial u}{\partial t} = f(t, u) + D\Delta u. \tag{1}$$

The language we use is `Matlab` because implementation of numerical schemes is straightforward.

On the Moodle page for the course, wou will find a short tutorial on `Matlab` to get you started.

Solving partial differential equations numerically requires discretizing the problem. Here we will convert the PDE into a large system of ODE by discretizing the Laplacian term in the equation. The Laplacian term represents the diffusion. For a function $u : I \times \Omega \to \mathbb{R}^n$, where I is a time interval, and the spatial domain $\Omega \subset \mathbb{R}^d$ is an open connected region with a smooth boundary, the **Laplacian** is

Laplacian

$$\Delta u = \sum_{i=1}^{d} \frac{\partial^2 u}{\partial x_i^2}, x \in \Omega.$$

Example 1 In 1D, the domain Ω can be an open interval (a, b). The Laplacian is not evaluated at the boundary points a and b; this is the matter of boundary conditions. In theory, the interval can be infinite: $(a, +\infty)$, $(-\infty, b)$, or $(-\infty, +\infty)$.

Example 2 Alternatively, the domain Ω can be a one-dimensional torus. For an interval $[a, b)$, we can identify the points a and b

Example 3 In 2D, the domain can be more varied. The simplest domain is a rectangle

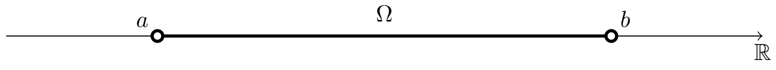


Figure 1: 1D spatial domain: the interval.

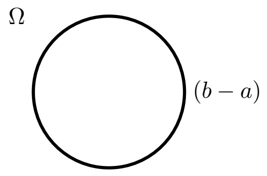


Figure 2: 1D spatial domain: the torus.

$\Omega_1 = [x_0, x_1] \times [y_0, y_1]$. Other finite domains can be included in a rectangle

For vector u, the derivatives are taken component by component. For example, for a domain $\Omega \subset \mathbb{R}^2$, and a function u with three components $u(t, x, y) \in \mathbb{R}^3$, the Laplacian is

$$\begin{aligned}\Delta u &= \Delta \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} \\ &= \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \\ &= \begin{pmatrix} \frac{\partial^2 u_1}{\partial x^2} \\ \frac{\partial^2 u_2}{\partial x^2} \\ \frac{\partial^2 u_3}{\partial x^2} \end{pmatrix} + \begin{pmatrix} \frac{\partial^2 u_1}{\partial y^2} \\ \frac{\partial^2 u_2}{\partial y^2} \\ \frac{\partial^2 u_3}{\partial y^2} \end{pmatrix}.\end{aligned}$$

The finite difference method consists in approximating the Laplacian by a finite-dimension matrix L. When substituted for the Laplacian, equation 1 becomes a system of ODE

$$\frac{\partial u}{\partial t} = f(t, u) + D(Lu).$$

By doing that, u is not a function of space anymore, but a (large) vector (or set of vectors) that approximate the solution at different points in space.

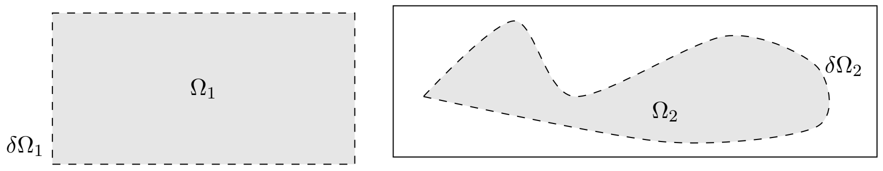


Figure 3: 2D spatial domains.

The practical will proceed as follows.

- Explicit finite difference in 1D spatial domains
- Implicit finite difference (Crank-Nicolson) in 1D spatial domains
- Explicit finite difference in 2D spatial domains
- Implicit finite difference (Crank-Nicolson) in 2D spatial domains
- Implicit finite difference (Crank-Nicolson) with alternating direction (ADI) in 2D spatial domains

For each case, a basic `Matlab` code is provided. All codes are functional, and the exercises require to run and modify those codes. You can refer to the material presented at the end of this document for more theoretical perspectives. The goal of this practical is for you to be able to adapt codes to solve new reaction-diffusion systems, in one or two dimensional domains, using explicit or implicit methods.

1.1 List of codes

- `fitzhughnagumo_diffusion.m` link.
- `FKPP_2D_fd_explicite.m` link.
- `FKPP_2D_fd_implicite.m` link.
- `FKPP_2D_fd_implicite_ADI.m` link.
- `FKPP_2D_fd_implicite_ADI.m` link.
- `turing_patterns_2D.m` link.

2 Exercises

First create a code folder where you will store the codes for this practical. Launch `Matlab`. You should have a graphical interface with many windows. Then, `cd` to your code folder, either by navigating the `Current Folder`, or by typing `cd path/to/your/codefolder/` in the `Command Window`.

Exercice 1 Explicit method in 1D – Travelling waves in the FitzHugh-Nagumo equations with diffusion

In this exercise, we will explore the **explicit finite difference scheme in 1D**. As an test case, we will use the FitzHugh-Nagumo equations with diffusion, with parameters set to have a travelling wave solution.

explicit finite difference scheme in 1D

The FitzHugh-Nagumo equations with diffusion are

$$\begin{aligned}\frac{dv}{dt} &= v - v^3/3 - w + I + D\Delta v;\\ \frac{dw}{dt} &= \varepsilon(v + b - cw);\end{aligned}$$

This system of reaction-diffusion equations has been used to model the propagation of an action potential along an axon, or to model the propagation of action potential in the heart muscle. The model describes the evolution of the voltage created by concentration difference in ions as they cross a cellular membrane. The variable v is the difference of potential (the voltage) across the cellular membrane. The variable w is a recovery variable that lumps together several mechanisms for ion balance At rest, the voltage is negative (more negative inside the cell). When a current I is applied to the cell, ion channels open up an let positive ions enter the cell, and the voltage rapidly becomes positive, at which point the channels close down and the cell starts pumping ions out to go back to the resting potential. The action potential is the event of voltage spiking. The action potential is followed by a refractory period, where no new action potential can occur. The parameter ε controls the length of the refractory period.

The spatial domain is the interval (x_0, x_1), $x_0 < x_1$.

(a) Download the script file `fitzhughnagumo_diffusion.m` at
`https://gist.github.com/samubernard/0b2c18001bbbd2d712ea533359c9b496`.
Save the file in your code folder. This code implement the FitzHugh-Nagumo equations on a 1D interval with either Neumann boundary conditions, periodic boundary conditions, or Dirichlet boundary conditions.

Run the code `fitzhughnagumo_diffusion` by typing

```
>> fitzhughnagumo_diffusion
```

in the `Command Window` followed by `Enter` [1]. The command will execute the script contained in the file. All defined variables will appear in the `Workspace`. You can list defined variables with the command `whos`. A figure window will also appear in a new `Figures` window. You should see the evolution of spatial profiles $v(t, x)$ and $w(t, x)$ for advancing values of time t.

[1]The >> is the `Matlab` prompt, it is not part of the command. `Matlab` can auto-complete by pressing the `Tab` key after entering the first few characters of the name of a script file in you current folder. This is useful to check that `fitzhughnagumo_diffusion.m` is actually in your folder. The current folder is indicated in the address bar on top of the `Matlab` window, or with the command `cd`. The `Command Window` works much like a unix shell, and a few commands are the same: `cd`, `ls`, `mkdir`. You can issue a shell command by preceding it with `!`. For example, to get the date, type `>> ! date`.

(b) Check the structure of the discretized Laplacian L with the command `spy`.

(c) Increase the time step and rerun the script until solutions start doing strange stuff. At what values does the numerical scheme lose stability? Does it satisfy the condition $k < h^2/(2D)$?

Exercice 2 Explicit method in 1D – Turing Patterns

Save the file `fitzhughnagumo_diffusion.m` under the name `gierermeinhardt.m`. Change the equations on (v, w) by the Gierer-Meinhardt model

$$\frac{\partial a}{\partial t} = \rho \frac{a^2}{h} - \mu_a a + D_a \frac{\partial^2 a}{\partial x^2},$$
$$\frac{\partial h}{\partial t} = \rho a^2 - \mu_h h + D_h \frac{\partial^2 h}{\partial x^2}.$$

Take $D_a \ll D_h$, $\mu_h > \mu_a$. You might need to change the notation to avoid clashing with existing variables. Use periodic boundary conditions, and random initial conditions generated with `rand(J,1)` for instance. A parameter set that shows Turing patterns is $D_a = 0.01, D_h = 0.2, \rho = 0.5, mu_h = 0.5, mu_a = 0.45$.

Exercice 3 Implicit Crank-Nicolson method in 1D – FPKK equation

In this exercise, we numerically solve the Fisher-KPP equation

$$\frac{\partial u}{\partial t} = ru(1-u) + D\Delta u$$

on an interval $\Omega = (a, b)$ with Neumann no flux boundary conditions.

(a) Download the code at
`https://gist.github.com/samubernard/c080bca02ec7ba55594b9eb3e77187b1`.
Save it in your code folder. Run the script `FKPP_1D_fd_implicite`. You should see a travelling wave moving right.

(b) We have seen in class that the wave speed $c \geq c_0 = 2\sqrt{rD}$. Compute c_0 and compare with the speed you see in the numerical solution.

Exercice 4 Explicit method in 2D – FKPP equation

(a) Download the code at
`https://gist.github.com/samubernard/f989ebc8fde012f53897e6dfe998aace`.
Save it in your code folder. Run the script `FKPP_2D_fd_explicite`. You should see... not much. The code is slow. Stop the code by pressing `Ctrl-C` when the `Command Window` is in focus.

(b) Decrease spatial resolution by increasing the value of h by a factor 10. Now you should see a 2D travelling wave front progressing in the South-East direction.

(c) Reset $h = 0.025$ and increase set the timestep to $k = 0.01$. Run the code again. What do you see?

Exercice 5 Implicit Crank-Nicolson method in 2D – FKPP equation

(a) Download the code at
`https://gist.github.com/samubernard/7a9ccd1fc76e268b44ed4d3ba737ac15`.
Save it in your code folder. Run the script `FKPP_2D_fd_implicite`. You should see... a travelling wave.

(b) How much can you increase the time step ?

Exercice 6 Implicit Crank-Nicolson ADI method in 2D – FKPP equation

(a) Download the code at
`https://gist.github.com/samubernard/2eadf8d4d494cb334656a5340207a746`.
Save it in your code folder. Run the script `FKPP_2D_fd_implicite_ADI`.

(b) Compare computational times with the non-ADI code.

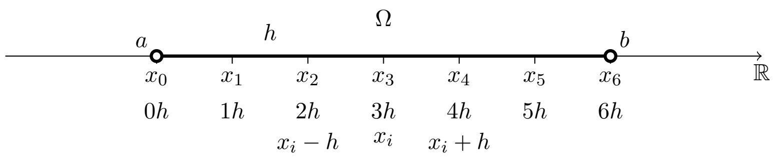
Exercice 7 Turing patterns in 2D

(a) Download the code at
`https://gist.github.com/samubernard/02451fc8a4789ca08819cbd541626065`.
Save it in your code folder. Run the script `turing_patterns_2D`. Use the command `view(2)` and `view(3)` to switch between 2D and 3D view of the solution.

(b) Try to find nice pattern by changing model parameters, in particular D_v and D_w. Once you are satisfied, you can try to increase the size of the domain (see Figure 4).

---

3 Finite-difference in 1D



Ω
a
h
b
$\mathbb{R}$
x_0 x_1 x_2 x_3 x_4 x_5 x_6
$0h$ $1h$ $2h$ $3h$ $4h$ $5h$ $6h$
$x_i - h$ x_i $x_i + h$

We discretize the domain $\Omega = (a, b)$ into J distinct points x_i, $i = 0, ..., J-1$, equally spaced with step $h = \frac{b-a}{J-1}$. These points include the boundary $x_0 = a$ and $b = x_{J-1}$. We will look for a discretized solution u_i^t at point x_i: $u_i^t \approx u(t, x_i)$. The finite-

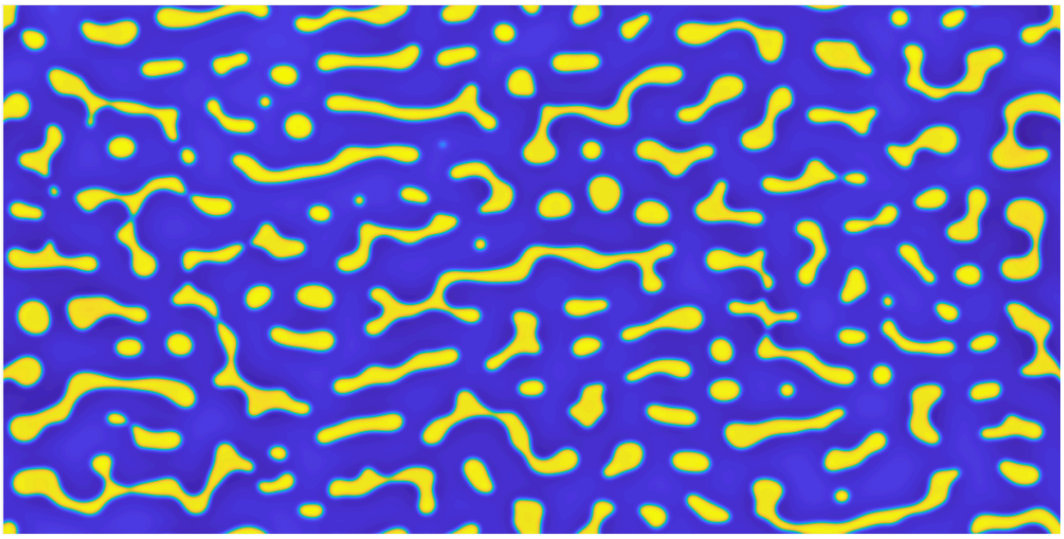


Figure 4: My best shot for Turing patterns in 2D.

difference scheme consists in approximating the derivative of a function by

$$\begin{aligned}\frac{\partial u}{\partial x} &\approx \frac{u(x+h)-u(x)}{h},\\ \frac{\partial^2 u}{\partial x^2} &\approx \frac{\partial}{\partial x}\frac{u(x+h)-u(x)}{h},\\ &\approx \frac{(u(x+h)-u(x))-(u(x)-u(x-h))}{h^2},\\ &\approx \frac{u(x+h)-2u(x)+u(x-h)}{h^2}.\end{aligned}$$

If the finite-difference scheme is applied at the discretized values of the domain Ω, we obtain

$$\begin{aligned}\Delta u(t,x_i) &\approx \frac{u(t,x_i+h)-2u(t,x_i)+u(t,x_i-h)}{h^2},\\ &\approx \frac{u(t,x_{i+1})-2u(t,x_i)+u(t,x_{i-1})}{h^2},\\ &\approx \frac{u_{i+1}^t-2u_i^t+u_{i-1}^t}{h^2}.\end{aligned}$$

Subsituting the discretized solution and Laplacian in the original PDE 1, we obtain a large system of ordinary differential equations

$$\frac{du_i}{dt} = f(t,u_i) + D\frac{u_{i+1}-2u_i+u_{i-1}}{h^2}. \tag{2}$$

You will notice that the finite-difference scheme only works for $i = 1, ..., J-2$, i.e. for points inside the domain Ω, but the scheme depends on points on the boundary (when $i = 1$ or $J-2$). This means that we need to provide separate rules for u_0^t

and u^t_{J-1}. These rules are called **boundary conditions**. The need for boundary conditions comes from the PDE problem itself, it is not an artifact of the discretization. The discretization merely makes the boundary condition problem transparent. This means that the boundary condition can be expressed in term of the original variable u. We will consider two types of boundary conditions: Neumann and Dirchlet. The Dirichlet boundary condition sets the values of u on the boundary

boundary conditions

$$u(t,x) = c(t,x),\ x \in \delta\Omega,$$

for a known function b. For the heat equation, the **Dirichlet boundary condition** means setting a fixed temperature at the boundary. The **Neumann boundary condition** sets the values of the outward normal derivatives of u on the boundary. The normal derivative can be interpreted as a flux.

Dirichlet boundary condition
Neumann boundary condition

$$\frac{\partial u}{\partial \vec{n}} = \phi(t,x),\ x \in \delta\Omega,$$

For the heat equations, this means either heating the material if the derivative is positive, or that the material is dissipating heat if the derivative is negative. A important particular case of the Neumann boundary condition is setting the derivatives to zero. This case is called **no flux**. For the heat equations, this mean that the domain is fully insulated: there is no heat loss or gain. In 1D, the normal vector at the boundary $\vec{n}$ is 1 on the right bound, and -1 on left bound. Therefore, the normal derivative is

no flux

$$\frac{\partial u}{\partial \vec{n}} = \begin{cases} -\frac{\partial u}{\partial x} & x \text{ is on left side,} \\ +\frac{\partial u}{\partial x} & x \text{ is on right side.} \end{cases}$$

In the no flux conditions, the derivative $\partial u/\partial x = 0$ on at both ends of the domain.

How do these conditions look on the discretized PDE? For the Dirichlet conditions one needs to specify $u_0 = c(t,a)$ and $u_{J-1} = c(t,b)$. With these specifications the discretized system is well defined, and can be solved numerically using time-stepping solvers. For the Neumann condition, the derivative needs to be discretized, using finite-difference

$$\frac{\partial u}{\partial \vec{n}} \approx \begin{cases} -\frac{u_1 - u_0}{h} & \text{left side,} \\ \frac{u_{J-1} - u_{J-2}}{h} & \text{right side.} \end{cases}$$

On the left end, the condition becomes $u_0 - u_1 = h\phi(t,a)$, or $u_0 = u_1 + h\phi(t,x_0)$. To check consistency, if the flux iϕ is negative, then $u_0 < u_1$, solution loses at the border.

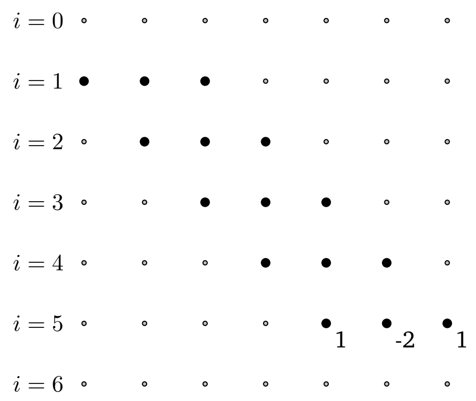


Figure 5: Structure of the discretized Laplacian L, for interval 1D domains.

On the right end, the condition becomes $u_{J-1} - u_{J-2} = h\phi(t,b)$, or $u_{J-1} = u_{J-2} + h\phi(t,b)$. Consistency check: if $\phi < 0$, $u_{J-1} < u_{J-2}$, and again the solution loses at the border. For the no flux condition, we have $u_0 = u_1$ and $u_{J-1} = u_{J-2}$.

So far we have considered 1D spatial domains as intervals, but it is also possible to consider a one-dimensional torus instead. The torus can be expressed as an interval with periodic boundary, where $a = b$. Therefore, we do not have to compute the solution at point b, it is the same as in a, this gives us a point for free. This affect the discretization, as we do not want to include b. As before, $x_i = a + ih$, but now the last point is $x_{J-1} = b - h$. This leads to $x_{J-1} = a + (J-1)h = b - h$, or $h = \frac{b-a}{J}$.

3.1 Explicite finite-difference scheme in 1D

The discretization scheme can be expressed in matrix-vector form. Take $\boldsymbol{u} = (u_0, u_1, ..., u_{J-1})^t \in \mathbb{R}^J$, then the discretized system can be expresssed as

$$\frac{d}{dt}\boldsymbol{u} = f(t, \boldsymbol{u}) + \frac{D}{h^2}L\boldsymbol{u}. \qquad (3)$$

The Laplacian has been replaced by a $J \times J$ matrix L. The structure of L is simple: for each row $i = 1, 2, ..., J-2$ corresponding to a point x_i inside the domain, there are coefficients $(1, -2, 1)$ at columns $i-1, i, i+1$. This forms a tri-diagonal matrix: -2 on the main diagonal, and 1 on the off-diagonals.

For periodic boundaries, when the domain is a torus, the matrix L has a regular structure System (3) is solved using a time-stepping scheme. The simplest one is the Forward Euler (FE) scheme, a first-order explicit scheme of the Runge-Kutta family. The solution $\boldsymbol{u}$ is discretized at times $t_0, t_0 + k, ...$, so that the solution $\boldsymbol{u}^n \approx \boldsymbol{u}(t_n)$,

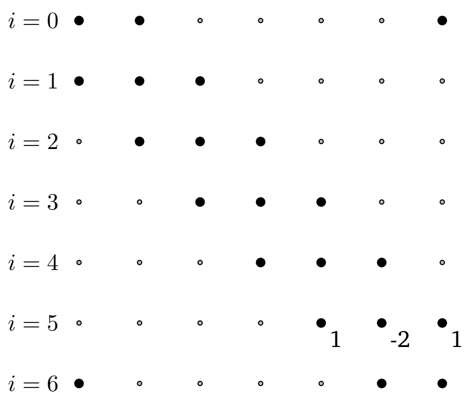


Figure 6: Structure of the discretized Laplacian L, for periodic 1D domains.

with $t_n = t_0 + nk$. One time-step of the Forward Euler scheme is

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + kf(\boldsymbol{u}^n) + k\frac{D}{h^2}DL\boldsymbol{u}^n.$$

We are ready to implement the scheme

```
% Pseudo-code, Matlab style
% du/dt = f(u) + D d^2u/dx^2
% define the discretized domain (a,b)
set the number of points J;
if periodic_boundary
  h = (b-a)/J;
else
  h = (b-a)/(J-1);
end
for i = 1 to J
  x(i) = h*(i-1);     % Matlab index starts a 1, not 0!
end
L = sparse(J,J);      % initialize at sparse JxJ matrix
L(diag) = -2;         % set diagonal to -2
L(off_diagonals) = 1; % set off-diagonals to +1
if periodic_boundary
  L(1,J) = 1;
  L(J,1) = 1;
else
  L(1,:) = 0;           % set boundary rows to 0
  L(J,:) = 0;           % set boundary rows to 0
end
```

```
u = initial_condition(x);  % initialize solution vector at t = 0
t = 0;                     % initialize time
set tfinal > 0;
set time step k;

while t < tfinal
  u = u + k*( f(u) + D/h^2*L*u ); % Forward-Euler update
  u(1) = u(2);                    % No flux boundary condition
  u(J) = u(J-1);                  % No flux boundary condition
  t = t + k;
end
```

One last detail. The Forward-Euler scheme is not stable for any value of k. The time step must be small: $k < h^2/(2D)$. Given that h is presumably already small, this scheme can be very slow. Moreover, the reaction term can limit the timestep to lower values. Implicit schemes are much better.

3.2 Implicite finite-difference scheme in 1D – Crank-Nicolson

The Crank-Nicolson scheme in 1D is a simple change of when the discretized Laplacian is evaluated. In the explicit scheme, the diffusion term is Lu^t. In the Crank-Nicolson scheme, it is $\big(Lu^t + Lu^{t+k}\big)/2$. This lead to an updating scheme

$$u^{t+k} = u^t + kf(u^t) + \frac{D}{2h^2}(Lu^t + Lu^{t+k}).$$

Given that the unknown is the solution at the next step u^{t+k}, this can be solved as

$$(I - k\frac{D}{2h^2}L)u^{t+k} = u^t + kf(u^t) + k\frac{D}{2h^2}Lu^t.$$

This is just a plain linear system. Call $A = (I - k\frac{D}{2h^2}L)$, the solution for u^{t+K} writes

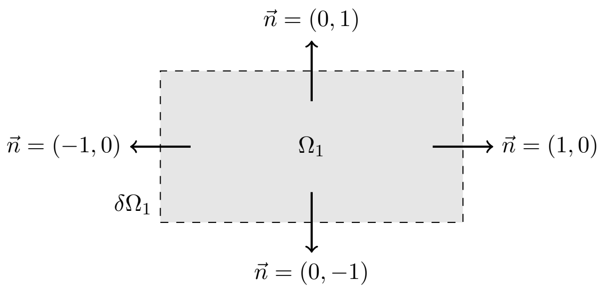
$$u^{t+k} = A^{-1}\big(u^t + kf(u^t) + k\frac{D}{2h^2}Lu^t\big).$$

With such an implicit scheme, there is not condition on k anymore, it becomes independent from the space step h. Of course, the linear system needs to be solved at each time step, which is slow. Fortunately the matrix A is tri-diagonal and there are very efficient algorthims to solve these tri-diagonal linear systems. This makes the implicit Crank-Nicolson scheme quite efficient.

4 Finite-difference in 2D

For a rectangular domain, the normal derivative is easy to compute

$$\frac{\partial u}{\partial \vec{n}} = \begin{cases} -\frac{\partial u}{\partial x} & x \text{ is on left side,} \\ +\frac{\partial u}{\partial x} & x \text{ is on right side,} \\ -\frac{\partial u}{\partial y} & x \text{ is on bottom side,} \\ +\frac{\partial u}{\partial y} & x \text{ is on top side.} \end{cases}$$



5 Solutions to the exercises

Solution to exercise 1

Solution to exercise 2 (c) Numerical artefact–the code is unstable

Solution to exercise 3 (b) Approximately $k < 2.0$

Solution to exercise 4 (b) I get 4.5s for ADI, and 26s for the non-ADI code.