Migrating populations

4.1 Learning outcomes

Mathematics and Physics

- Partial differential equations
- Boundary Conditions
- Relaxation methods

Computing and Python

- Using sparse matrices

4.2 Introduction

So far we have assumed that all the populations are spatially homogeneous, but in reality, animals move and populate different areas. This is what we will try to model next using a simple model for rabbits, inspired by our previous models, which considers rabbits living on the slope of a mountain. In the model, the food supplies on the slope decrease with altitude and rabbits are killed by lazy hunters who do not like to walk up the mountain, so the hunting rate will decrease with altitude as well.

Mathematically we will describe how to compute static solutions of first order partial differential equations using various relaxation methods. We will start with the Jacobi relaxation method and then proceed with the Gauss-Seidel relaxation method. These techniques are very general and will be used again in chapter 6.

4.3 Model

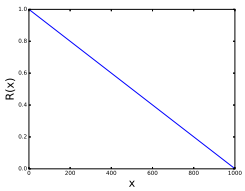
The position on the mountain slope will be given by the variable x which corresponds to the distance from the river at the bottom of the valley. The ground elevation is also proportional to x as we assume a constant slope. Our first modelling attempt is to use

$$\frac{\mathrm{d}N(t,x)}{\mathrm{d}t} = KN(t,x)\exp\left(-\frac{N(t,x)}{R(x)}\right) - Y(x)N(t,x)\,. \tag{4.1}$$

Here we now assume that $N(t,x)$, the rabbit population density, is a function of both time, t, and position x. The first term on the right hand side of (4.1) describes the rabbits' breeding rate. It is proportional to N but decreases as N becomes larger than a reference population $R(x)$. We expect $R(x)$ to decrease with altitude as there is less food for rabbits. We will take

$$R(x) = R_{\text{max}} - \frac{x}{L}\left(R_{\text{max}} - R_{\text{min}}\right)\,, \tag{4.2}$$

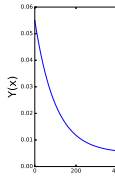
where R_{max} is the reference population at the bottom of the hill, $x = 0$, and R_{min} the reference population at the top of the hill, $x = L$. This is of course a modelling assumption.



The reference population as a function of the position on the hill; more food at the bottom means a larger reference population.

The second term of (4.1) describes the dying rate of rabbits, through both hunting and old age. We expect it to be constant for the natural death rate, but the hunting death rate must be decreasing as one moves away from the the bottom of the hill. We can use

$$Y(x) = Y_H\,\exp\left(-\frac{x}{\lambda}\right) + Y_n\,, \tag{4.3}$$



The death rate
the position o
at the bottom
hunters.

where Y_n is the natural death rate and Y_H the hunting death rate at the bottom of the hill.

Equation (4.1) does capture some space dependency, but what it fails to describe is the fact that rabbits will move. We thus need to add extra terms to our equation. We can reasonably assume that rabbits move randomly in all directions and to model this, we can divide our x interval in small boxes b_i of width $\mathrm{d}x$ and centred at position x_i, each containing $N_i = N(t, x_i)\mathrm{d}x$ rabbits. We then assume that the number of rabbits from box b_i moving up the slope during a unit time interval (say a day) is proportional to the number of rabbits in box i: $\Delta N_i^+ = k_i^+ N_i$, where k_i^+ is the rate at which rabbits move up the hill. Similarly the number of rabbits moving towards the valley will be $\Delta N_i^- = k_i^- N_i$ where k_i^- is the rate at which rabbits move downhill. The total number of rabbits moving out off the box b_i is thus $\Delta N_i = (k_i^+ + k_i^-)N_i$ while the number of rabbits moving into box b_i (coming from boxes b_{i+1} and b_{i-1}) is given by $k_{i-1}^+ N_{i-1} + k_{i+1}^- N_{i+1}$.

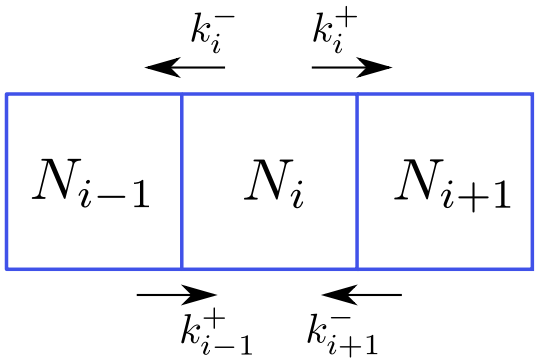


Figure 4.1: Rabbit movement ('diffusion') between boxes.

So the number of rabbits in box i will change by $\delta N_i = k_{i-1}^+ N_{i-1} + k_{i+1}^- N_{i+1} - (k_i^+ + k_i^-)N_i$. If we assume that k_i^+ and k_i^- are constant, we can drop the index i and write

$$\delta N_i = k^+(N_{i-1} - N_i) + k^-(N_{i+1} - N_i) \tag{4.4}$$

or more explicitly

$$\delta N(t, x_i) = k^+\Big(N(t, x_{i-1}) - N(t, x_i)\Big) + k^-\Big(N(t, x_{i+1}) - N(t, x_i)\Big)\,. \tag{4.5}$$

In order to rewrite this as an equation which involves the population only at one single position x_i (i.e. in order to get rid of the $N(t, x_{i\pm1})$), we can now use the Taylor series

$$\begin{aligned} N(t, x_{i\pm1}) &= N(t, x_i \pm \mathrm{d}x) \\ &= N(t, x_i) \pm \mathrm{d}x\frac{\mathrm{d}N}{\mathrm{d}x}(t, x_i) + \frac{\mathrm{d}x^2}{2}\frac{\mathrm{d}^2N}{\mathrm{d}x^2}(t, x_i) + \mathcal{O}(\mathrm{d}x^3)\,. \end{aligned} \tag{4.6}$$

Substituting this Taylor series into (4.5) allows us to convert the latter equation into one that involves only continuous variables x,

$$\delta N(t, x_i) = \mathrm{d}x(k^+ - k^-)\frac{\mathrm{d}N}{\mathrm{d}x}(t, x_i) + \frac{\mathrm{d}x^2}{2}(k^+ + k^-)\frac{\mathrm{d}^2N}{\mathrm{d}x^2}(t, x_i)\,. \tag{4.7}$$

To make this look a bit more compact we introduce the symbols

$$D = \frac{\mathrm{d}x^2}{2}(k^+ + k^-), \qquad V = \mathrm{d}x\,(k^+ - k^-)\,, \tag{4.8}$$

which turns (4.7) into

$$\delta N(t, x_i) = V\frac{\mathrm{d}N}{\mathrm{d}x}(t, x_i) + D\frac{\mathrm{d}^2N}{\mathrm{d}x^2}(t, x_i)\,. \tag{4.9}$$

This equation describes the variation of the rabbit population density induced by random displacement. We can thus add $\delta N(t, x_i)$ to (4.1) to finally obtain

$$\frac{\partial N}{\partial t} = KN \exp\left(-\frac{N}{R}\right) - YN + V\frac{\partial N}{\partial x} + D\frac{\partial^2 N}{\partial x^2}\,. \tag{4.10}$$

For clarity we have removed the explicit x and t dependence of N, R and Y. Notice that as we have more than one variable in the equation we use the symbol for partial derivative $\partial/\partial x$ instead of $\mathrm{d}/\mathrm{d}x$. The partial derivatives are evaluated by considering that all the other variables remain constant, which is equivalent so saying that the variables are independent.

4.4 An aside: the diffusion equation

While the model above describes rabbits it can actually be used to describe a much broader range of physical situation. For example, tiny dust particles suspended in air are constantly bombarded by the surrounding air molecule and, as a result, move randomly. In that case the particles do not die and do not breed, so $Y = K = 0$ and the equation reduces to

$$\frac{\partial N}{\partial t} = V\frac{\partial N}{\partial x} + D\frac{\partial^2 N}{\partial x^2}\,. \tag{4.11}$$

This is a well-known equation called the *diffusion equation*. It describes the movement of dust particles, but also the displacement of molecule inside our bodies, ink particles in a liquid and so on.

Homework 4.1:

a) Show by direct substitution that the function

$$N = \frac{A}{\sqrt{4\pi D t}} \exp\left(\frac{-(x+Vt)^2}{4Dt}\right) \tag{4.12}$$

is a solution of (4.11) for any constant A.

b) What does the equation describe? Take $A = D = 0$ and plot the function at $t = 1$. What does the function look like in the limit $t \to 0$ and $t \to \infty$. What is the interpretation of the parameters D and V?

Homework 4.2:

Define

$$Q = \int_{-\infty}^{\infty} N(x)\mathrm{d}x\,. \tag{4.13}$$

a) Show that for solutions of (4.11) which satisfy $\lim_{x\to\pm\infty} N(x) = 0$ as well as $\lim_{x\to\pm\infty} \partial N(x)/\partial x = 0$ the quantity Q is constant, that is

$$\frac{\partial Q}{\partial t} = 0\,, \tag{4.14}$$

for all values of t. Hint: move the derivative inside the integral and use the equation to replace $\partial N/\partial t$.

b) Evaluate Q for (4.12) using the property above to make the problem simpler.

c) What is the interpretation of (4.14)?

4.5 Boundary conditions

Equation (4.10) is not sufficient to fully describe the problem. We must indeed decide what happens at the boundary of the rabbit domain, *i.e.* near the river $x = 0$ and at the top of the hill $x = L$ (which we can picture as the top of a vertical cliff). There are two main conditions that we can impose: decide that all rabbits that reach the river or the cliff edge die instantly (drowning, being shot by hunters or falling of the cliff edge). We can also consider that the rabbits reaching the edges of their domain simply stop and move back. The first condition is mathematically described by imposing the condition $N(t, 0) = N(t, L) = 0$ and is also known as the *Dirichlet* boundary condition. The second condition is obtained by imposing $\frac{\partial N}{\partial x}(t, 0) = \frac{\partial N}{\partial x}(t, L) = 0$ and is also known as the *Neumann* boundary condition. We should emphasise here that *which* boundary condition you choose depends entirely on which problem you are modelling.

We do not need to take the same boundary condition at the two ends. We can decide that the rabbits drown in the river, $N(t, 0) = 0$, but walk back when facing the cliff, $\frac{\partial N}{\partial x}(t, L) = 0$. In general, one can also impose that N at the boundary is a fixed constant different from 0 (a farmer could feed or kill the rabbits to ensure a constant population near the river), or one could impose that $\frac{\partial N}{\partial x}(t, L)$ is a non-zero constant.

When the domain is infinite, one usually has to impose that both N and its derivative converge to 0 to ensure a finite population; we will see examples of such boundary conditions in 6 (for a different problem).

Homework 4.3:

If K, R, Y, V and D are constants (independent of both t and x), then eq (4.10) can have non-trivial, constant solutions.

a) Compute the constant solution N as a function of the model parameters.

b) Which constraints on the model parameters must be satisfied for the constant solution to exist?

c) Which type of boundary condition does this solution satisfy?

4.6 Discretisation for numerical solutions

While the diffusion equation (4.11) can be solved analytically, solving (4.10) is much harder, especially if R and Y depend on x. Numerically on the other hand, it is just as easy. The method is the same as the one we used earlier to solve ODEs numerically. We use the Taylor series of $N(t, x)$ to replace derivatives by finite differences. One can easily show (see homework below) that

$$\begin{aligned} \frac{\partial N(x)}{\partial x} &= \frac{N(x + \mathrm{d}x) - N(x - \mathrm{d}x)}{2\,\mathrm{d}x} + \mathcal{O}(\mathrm{d}x^2)\,, \\ \frac{\partial^2 N(x)}{\partial x^2} &= \frac{N(x + \mathrm{d}x) + N(x - \mathrm{d}x) - 2N(x)}{\mathrm{d}x^2} + \mathcal{O}(\mathrm{d}x^2)\,, \end{aligned} \tag{4.15}$$

using the second order Taylor expansion of $N(x)$ around x. A similar expansion holds for the t-derivative. Substituting (4.15) into (4.10) we get

$$N_{n+1,i} = N_{n,i} + \mathrm{d}t \Big[K N_{n,i} \exp\Big(- \frac{N_{n,i}}{R} \Big) - Y N_{n,i} + V\, \frac{N_{n,i+1} - N_{n,i-1}}{2\,\mathrm{d}x} + D\, \frac{N_{n,i+1} + N_{n,i-1} - 2N_{n,i}}{\mathrm{d}x^2} \Big], \tag{4.16}$$

where the index n and i label respectively the time and spatial discretisation, so that $x_i = i\mathrm{d}x$ and

$$N_{n,i} = N(t_0 + n\,\mathrm{d}t, i\,\mathrm{d}x)\,. \tag{4.17}$$

If we split the range $[0, L]$ in M_L intervals we have $\mathrm{d}x = L/M_L$. In case $Y = K = 0$ the system above reduces to the discretisation of the diffusion equation,

$$N_{n+1,i} = N_{n,i} + \frac{D\mathrm{d}t}{\mathrm{d}x^2}\Big[N_{n,i+1} + N_{n,i-1} - 2N_{n,i}\Big]\,. \tag{4.18}$$

Notice that we have once again used the Euler method to perform the time integration, but we can easily replace it by the 4th order Runge-Kutta method if needed.

Homework 4.4:

Prove the expressions (4.15).

4.7 Computing solutions with Jacobi relaxation

Solving (4.16) for $N_{n,i}$ subject to whatever boundary conditions we decide to impose is still a complicated task. However, if we are given an initial population distribution $N_{0,i}$ we can use (4.16) to find the distribution after a time step $\mathrm{d}t$, and then iterate this repeatedly to find the population at any time. As we will see, for late time, we will typically find that the solutions become static, i.e. change less and less as time evolves. Absence of time dependence does not mean that our rabbits do not move up or down the hill. Rather, it means that the change of the number of rabbits in any given box is such that migration, death and birth rate precisely cancel out, to produce no change.

This general idea, that we can find static solutions by simply evolving the system until it becomes time-independent, goes under the name of *Jacobi relaxation* and *Gauss-Seidel relaxation*. Effectively, it allows us to solve for a solution to the simpler system

$$0 = KN_i \exp\Big(-\frac{N_i}{R}\Big) - YN_i + V\,\frac{N_{i+1} - N_{i-1}}{2\,\mathrm{d}x} + D\,\frac{N_{i+1} + N_{i-1} - 2N_i}{\mathrm{d}x^2}\,, \tag{4.19}$$

where we have dropped the n index because of time independence. To see how this works, let us first look at a simple example with only a few degrees of freedom.

For this simpler example we will set $K = Y = V = 0$, that is, consider the system arising from the diffusion equation. Its discrete form was given in (4.18). We will limit ourselves to $i = 0, \ldots, 4$ and take as the boundary conditions $N_0 = 1$ and $N_4 = 0$. The system of equations (4.18) then reduces to

$$\begin{aligned} N_{n+1,1} &= N_{n,1} + \nu_J\big(N_{n,2} + 1 - 2N_{n,1}\big)\,,\\ N_{n+1,2} &= N_{n,2} + \nu_J\big(N_{n,3} + N_{n,1} - 2N_{n,2}\big)\,,\\ N_{n+1,3} &= N_{n,3} + \nu_J\big(N_{n,2} - 2N_{n,3}\big)\,, \end{aligned} \tag{4.20}$$

where $\nu_J = D\mathrm{d}t/\mathrm{d}x^2$ (and the subscript J refers to Jacobi iteration). This is better written in matrix form, as

$$\vec{N}_{n+1} = \vec{N}_n + \nu_J\Big(A\cdot\vec{N}_n - \vec{B}\Big)\,, \tag{4.21}$$

where

$$A = \begin{pmatrix} -2 & 1 & 0\\ 1 & -2 & 1\\ 0 & 1 & -2 \end{pmatrix}\,, \qquad \vec{B} = \begin{pmatrix} -1\\ 0\\ 0 \end{pmatrix}\,. \tag{4.22}$$

Homework 4.5:

a) Write a small Python program to iterate equation (4.21) starting from $N_0 = (0, 0, 0)$. Run it for $\nu_J = 0.1$ and $\nu_J = 0.5$ and in each case display the result for the first 20 iterations.

b) The static solution satisfies $A \cdot \vec{N}_n = \vec{B}$. What is the exact solution to this equation? Did your Python program reproduce it?

c) What happens when $\nu_J > 0.5$?

In the homework above you have seen that the maximal value of ν_J for which the iteration behaves well is $\nu_J = 1/2$. It is easy to see why this value is special. If we write A as the sum of a diagonal part D and the rest M, (4.21) for $\nu_J = 1/2$ becomes

$$\vec{N}_{n+1} = \frac{1}{2}\left(\vec{B} - M \cdot \vec{N}_n\right), \tag{4.23}$$

that is to say, the contribution from the diagonal part of D disappears. The system converges quickest to the static solution for this maximal value of ν_J. For systems other than the diffusion equation, this maximal ν_J is typically different.

Let us now return to our equation (4.19). This equation is clearly much more complicated than the example we have discussed above: it contains many more terms, and in particular these terms can be non-linear in the population numbers $N_{n,i}$, so that we cannot write it as a matrix equation anymore. Nevertheless, the same kind of logic applies: we will want to iterate

$$N_{n+1,i} = N_{n,i} + \nu_J \, F\big(N, i\big) \, . \tag{4.24}$$

Let us therefore now turn to writing a class which can solve systems like (4.24) independent of the precise details of the expression for $F(N, i)$.

The structure of the `RelaxJacobi1D` class listed below is very similar to the classes we have used to solve ODEs or systems of ODEs before.

The class function `F` is the function that describes the equation by computing what we called $F(N, i)$ above. In the test code below, we have implemented a class which defines this for the simple diffusion equation. The variable `v` contains the array of functions we integrate and `i` is the index where it must be evaluated, so that `|v[i]—` is the same as what we call N_i in (4.10). When we solve a different equation, `F` is the class function we must implement (often one must modify `reset` as well).

The main relaxation algorithm, which is independent of the particular details of $F(N, i)$, is implemented in the class function `relax_1_step`. This applies (4.24) once for every N_i, except for the first and last element for which it imposes the boundary conditions. It does the latter by calling `boundary`, which is another function that needs to be implemented for every concrete problem we need to solve. The function `relax_1_step` keeps track of the largest update, so we can see how close we are to a static solution. The class function `relax` calls this basic building block multiple times.

file: `relax_jacobi_1d.py`

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  class RelaxJacobi1D:
5    """ A class to solve F(v)=0, where F is a differential
6        operator acting on the vector 'v', using Jacobi relaxation."""
7
8    def __init__(self, v0, xmin, xmax, Np):
9      """
10     :param v0         : initial condition
11     :param xmin, xmax : domain boundaries
12     :param Np         : number of points
13     """
14     self.reset(v0, xmin, xmax, Np)
15
16
17   def reset(self, v0, xmin, xmax, Np):
18     """ Set the initial parameters.
```



```
19
20      :param v0         : initial condition
21      :param xmin, xmax : domain boundaries
22      :param Np         : number of points
23      """
24      self.Np = Np
25      self.xmin = xmin
26      self.xmax = xmax
27      self.dx = (xmax-xmin)/(Np-1)
28      self.n = 0 # time iteration number
29      self.v = np.array(v0, dtype='float64') # ensure we use floats!
30      self.l_x = []  # list of t values for figures
31      self.l_f = []  # list of f values (arrays) for figures
32      self.boundary() # make sure the boundary condition is set
33
34    def F(self, v, i):
35      """ Returns update for v at position i. To be implemented in subclass.
36
37      :param v : current function value (a vector).
38      :param i : spatial index.
39      :return  : update for v at position i.
40      """
41      pass
42
43    def boundary(self):
44      """ Enforce the boundary conditions. To be implemented in subclass."""
45      pass
46
47    def relax_1_step(self, nu=0.5):
48      """ Perform a single Jacobi relaxation iteration.
49
50      :param nu : relaxation parameter.
51      :return   : the largest update.
52      """
53      w = np.array(self.v)
54      dvmax = 0
55      for i in range(1, self.Np-1):
56        dv = nu*self.dx**2*self.F(self.v, i)
57        if(np.fabs(dv) > dvmax) : dvmax = np.fabs(dv)
58        w[i] = self.v[i] + dv
59
60      self.v = w
61      return(dvmax)
62
63    def relax(self, err, nu=0.5):
64      """ Relax until largest update is smaller than err.
65
66      :param err : target "error".
67      :param nu  : relaxation parameter.
68      :return    : number of iterations required.
69      """
70      e = 1e64  # absurdly large
71      n = 1
72      while(e > err):
73        e = self.relax_1_step(nu)
74        self.boundary()
75        n += 1
76        if(n % 1000==0): print("n=",n)
77
78      # set plot data
79      self.l_x = np.linspace(self.xmin,self.xmax,self.Np)
80      self.l_f = np.array(self.v)
81      return(n)
82
83    def plot(self, style="k-"):
84      """ Plot the vector 'v'.
85
86      :param style: format for the plot function.
87      """
88      plt.plot(self.l_x, self.l_f, style);
89
```

```
90
91 # Tests follow; only run when not importing the module.
92
93 if __name__ == "__main__":
94
95   class RelaxDiffusion(RelaxJacobi1D):
96     """ A class to relax to static solutions of the diffusion equation. """
97     def F(self, v, i):
98       return((v[i+1]+v[i-1]-2*v[i])/self.dx**2)
99
100    def boundary(self):
101      self.v[0] = 1
102      self.v[self.Np-1] = 0;
103
104
105  Np=200
106  relax = RelaxDiffusion(np.zeros([Np]), xmin=0, xmax=10, Np=Np)
107  n = relax.relax(1e-5)
108  print("No of iterations: ",n)
109  relax.plot()
110  plt.show()
```

Question 4.1:

Run the program `relax_jacobi_1d.py` and take note of the number of iteration returned by the program when run with `Np=100`, `Np=200` and `Np=300` (set line 106). How do the solutions differ between these choices of `Np`?

The equation we are solving is $\mathrm{d}^2N/\mathrm{d}x^2 = 0$ with boundary condition $N(0) = 1$ and $N(10) = 0$. Is the solution what you expect?

4.8 Gauss-Seidel relaxation for moving rabbits

As you may have noted, the Jacobi relaxation method requires a quite large number of iterations in order to converge to the static solution. Our rabbit problem is more complicated than the simple diffusion problem, so it would be good if we could speed up the relaxation.

There exists a small and simple modification which one can make to the relaxation method in order to make the convergence go much faster. This improved method goes under the name of *Gauss-Seidel relaxation*. The idea is to notice that when using the Jacobi method in (4.24), we always first compute the $N_{n+1,i}$ for all i (see the code above: in `relax_1_step` we first store all new values in `self.w`, and only copy that back into `self.v` at the end of the iteration step). The Gauss-Seidel method consists of using updated values as soon as they become available. It turns out that this method can be significantly faster.

Coding task 4.1:

The program `relax_GS_1d.py` defines a new class, called `RelaxGaussSeidel1D`, which is a subclass of `RelaxJacobi1D`. The only difference between the two is the function `relax_1_step`, which now implements the Gauss-Seidel method instead of the Jacobi method.

Complete the definition of the class function `relax_1_step`. The code will be very similar to the code used in `RelaxJacobi1D`; only two lines need to be modified.

Homework 4.6:

Run the programs `relax_jacobi_1d.py` and `relax_GS_1d.py`. Make sure `Np=200` and use `1e-5` as the precision argument of the `relax` function. Use `nu=0.9` as the second argument of the `relax` function for the Gauss-Seidel method.

a) How many iteration does each program need before the accuracy is reached?

b) Are the two solutions identical and is it what you expect?

We are now ready to return to our rabbit problem. In order to implement a subclass of `RelaxGaussSeidel1D` which solves it, we need to do two things: write the function `F(self, v, i)` so that it implements (4.16), and implement the function `boundary` so that it implements the correct boundary conditions for our rabbit model. We will assume that rabbits walk back when facing the river or the cliff, so that

$$\frac{\partial N}{\partial x}(t,0) = \frac{\partial N}{\partial x}(t,L) = 0\,. \tag{4.25}$$

In the discrete variables this implies that

$$N_{n,1} = N_{n,0}\,, \qquad N_{n,M_L} = N_{n,M_L-1} \qquad \forall n\,. \tag{4.26}$$

Coding task 4.2:

Complete the program `relax_rabbits.py` so that its solves the equation (4.16).

a) Complete the class functions `Y(self,x)` and `R(self,x)`. Use $R_{\max} = 1$, $R_{\min} = 0.001$, $Y_H = 0.05$ and $Y_n = 0.005$.

b) Assume a hill $L = 1000$ meters long and take $K = 0.1$, $V = 0$ and $D = 1$. Use `Np=100`.

c) In the function `F` complete the line setting dv. Notice that the function returns `dv/D`. This is because the Gauss-Seidel method was programmed assuming that $D = 1$, so we divide the equation by the parameter to fulfil that condition.

Homework 4.7:

a) Run the programs `relax_rabbits.py` and produce the profile of the rabbit density on the hill.

b) Explain the shape of the profile. Why does it have minima near each boundary?

4.9 Matrix method

The iterative Jacobi and Gauss-Seidel methods have the advantage that they work not only for linear systems of equations, but also when non-linear terms are present. Our rabbit model (4.16) has such non-linearities in the form of the exponential breeding term multiplying the K coefficient. However, for sufficiently small $N_{n,i}$, we can approximate the exponential by 1, and the system becomes linear in $N_{n,i}$. After setting $V = 0$ for simplicity (4.19) now reads

$$0 = KN_i - YN_i + D\,\frac{N_{i+1} + N_{i-1} - 2N_i}{\mathrm{d}x^2}\,. \tag{4.27}$$

This set of equations can subsequently be written in the form of a matrix equation,

$$\tilde{Q}\vec{f} = 0\,, \qquad \vec{f} = (N_0, \ldots, N_{M_L})^T\,, \tag{4.28}$$

where

$$\tilde{Q} = \frac{D}{\mathrm{d}x^2}\left(\begin{array}{c|cccccccc|c} 1 & q_1 & 1 & 0 & 0 & . & . & . & . & . & . \\ 0 & 1 & q_2 & 1 & 0 & . & . & . & . & . & . \\ 0 & 0 & 1 & q_3 & 1 & . & . & . & . & . & . \\ . & \multicolumn{8}{c|}{\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots} & . & . \\ . & . & . & . & . & . & 1 & q_{M_L-3} & 1 & 0 & 0 \\ . & . & . & . & . & . & 0 & 1 & q_{M_L-2} & 1 & 0 \\ . & . & . & . & . & . & 0 & 0 & 1 & q_{M_L-1} & 1 \end{array}\right), \tag{4.29}$$

and the diagonal elements are given by

$$q_i = -2 + \frac{\mathrm{d}x^2}{D}\big(K - Y(x_i)\big)\,. \tag{4.30}$$

Notice that we only have $M_L - 1$ equations because N_0 and N_{M_L} are both determined by the boundary condition. If we decide to impose the Dirichlet boundary condition $N_0 = V_0$ at the origin and the Neumann boundary condition $N_{M_L-1} = N_{M_L}$ at $x = L$ we can rewrite (4.28) as

$$Q\vec{f} = \vec{F}\,, \qquad \vec{f} = (N_1, \ldots, N_{M_L-1})^T\,, \tag{4.31}$$

where the matrix Q is

$$Q = \frac{D}{\mathrm{d}x^2}\begin{pmatrix} q_1 & 1 & 0 & 0 & . & . & . & . & . \\ 1 & q_2 & 1 & 0 & . & . & . & . & . \\ 0 & 1 & q_3 & 1 & . & . & . & . & . \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & . \\ . & . & . & . & . & 1 & q_{M_L-3} & 1 & 0 \\ . & . & . & . & . & 0 & 1 & q_{M_L-2} & 1 \\ . & . & . & . & . & 0 & 0 & 1 & 1+q_{M_L-1} \end{pmatrix}, \tag{4.32}$$

$$\vec{F} = (-V_0, 0, \ldots, 0)^T. \tag{4.33}$$

Note that $\vec{f}$ now has only $M_L - 2$ components. In order to arrive at this form we have removed the first and the last column of $\tilde{Q}$ as follows: as $N_0 = V_0$, the first equation in (4.28) is $V_0 + (-2 + q_1)N_1 + N_2 = 0$ and we can move the constant term V_0 to the right-hand side of the equation, defining $\vec{F}$. Then the last equation in (4.28) is $N_{M_L-2} + (-2 + q_{M_L-1})N_{M_L-1} + N_{M_L} = 0$ and as $N_{M_L} = N_{M_L-1}$ we have $N_{M_L-2} + (-1 + q_{M_L-1})N_{M_L-1} = 0$. This is why the bottom-right element of Q now contains an extra term 1. Equation (4.31) is a system of M_{L-1} equations with M_{L-1} unknowns and it can be solved by inverting the matrix Q.

Usually one will take $M_L = 100$ or much larger values, so the matrix one has to invert is huge. Fortunately there are very effective methods to solve the system of equations (4.31) and Python has modules doing precisely that. For this, one should note that most of the element of Q vanish. Indeed out of the $(M_L - 1)^2$ elements, at most $3(M_L - 1)$ are non-zero. Q is said to be *sparse*. Such sparse matrices occur frequently in modelling problems, and mathematicians have developed algorithms to store sparse matrices and to solve linear systems of equations like (4.31) described by sparse matrices. In Python this is done by the module `scipy.sparse`. One can solve the system of equations (4.31), taking $q_i = -2$ (i.e. $K = Y = 0$) for simplicity, as follows

file: `simple_sparse.py`

```
import numpy as np
import scipy.sparse
import scipy.sparse.linalg

Ne=100                                                  # number of equations
Q=scipy.sparse.lil_matrix((Ne,Ne),dtype=np.float64) # create sparse matrix
for i in range(Ne):                                     # populate Q
    Q[i,i] = -2
    if (i > 0) :
        Q[i,i-1] = 1.
        Q[i-1,i] = 1.
    Q[Ne-1,Ne-1] += 1.

F = np.zeros(Ne)                                    # create F
F[0] = -1                                           # V0=1 nonzero element
W = scipy.sparse.linalg.spsolve(Q.tocsc(), F)       # solve

print(W)
```

The variable `W` contains the values of all the N_i with $i > 0$ and $i < M_L$. For the full solution one has to add the two endpoints: $N_0 = V_0$ and $N_{M_L} = W[M_L - 1]$. For this particular example the solution is rather boring because of the boundary conditions (do you understand this?).

Homework 4.8:

Write down the matrices Q and F for a system with $M_L = 7$ and $q_i = i$ with the Neumann boundary condition $N_1 = N_0$ and the Dirichlet boundary condition $N_{M_L-1} = 5$.

Coding task 4.3:

The program `diffusion_matrix.py` solves the diffusion equation (which is equivalent to $q_i = -2, \forall i$ in (4.29)) using the boundary conditions $N_0 = 1$, $N_{M_L} = 0$. Notice that in our notation M_L is the number of intervals, so that there is a total of $M_L + 1$ lattice points and thus the number of equations, labelled `Ne` in the code, is $M_L - 1$.

a) Run the code `diffusion_matrix.py`. Is the solution the expected one?

b) Modify `diffusion_matrix.py` and set the number of points to 4. Uncomment the print commands at the end of the `Qmat`, `Fmat`, `solve_matrix` and `plot` function to print all the arrays on the screen. Run the modified code `diffusion_matrix.py` and check that the solution solves the equation.

c) Copy the file `diffusion_matrix.py` into a file called `diffusion_matrix_2.py` and modify the boundary condition so that it correspond to $N_0 = 1$, $N_{M_L} = N_{M_L-1}$ (Equivalent to the Neumann boundary condition at $x = L$). You need to add 1 line in `Qmat`, remove one line from `Fmat` and modify 1 line in `plot`.

d) Run the code `diffusion_matrix_2.py` with `Ne=4` and check that the matrices are the expected ones.

e) Set `Ne=100`, comment out the print statements and run the code in the program `diffusion_matrix_2.py` again. Is the solution the expected one? Look at the vertical axis label to understand what is being plotted.

Homework 4.9:

a) What are the matrix `Q` and `F` output by the program `diffusion_matrix_2.py` in coding task 4.3.c?

b) Explain the solution obtained by the boundary conditions of coding task 4.3.c.

Coding task 4.4:

a) Complete the program `rabbits_matrix.py` so that it solves equation (4.27) for the simplified rabbit model with the boundary conditions $N(0) = V_0$ and $\frac{\mathrm{d}N(L)}{\mathrm{d}x} = 0$. Use the following parameter values $V_0 = 0.001$, $N_e = 100$, $x_{\text{min}} = 0$, $x_{\text{max}} = 1000$, $Y_H = 0.0005$, $Y_n = 0.001$, $\lambda = 100$, $K = 0.001$ and $D = 1$. Proceed as follows

- Complete the class function `Y` (same as in `relax_rabbits.py`).
- Complete the class function `Qmat`, `F` and `plot` (all similar to `diffusion_matrix_2.py`)

b) Copy the program `relax_rabbits.py` into relax `relax_rabbits_simple.py` so that it solves the equation for the simple rabbit model with the parameter above and the same boundary condition.

- Remove all references to the function `R`, and the parameters `Rmin` and `Rmax`
- Simplify the function `F` so that it solves the simplified rabbit model.
- Modify the function `boundary` to implement a fixed boundary condition at $x = 0$.
- Modify the parameters values at the bottom of the program. Take `Np=102`.
- Call the relax function with the following arguments: `n=rel.relax(err=1e-6,nu=1)`.
- Add the line `print("V[xmax]=",rel.v[rel.Np-1])` just before the `rel.plot()` to print the value of the population density at `xmax`.

Question 4.2:

Run the code `rabbits_matrix.py` and `relax_rabbits_simple.py`. Are the solution similar? To improve the relaxation method, you have to decrease the `err` argument of the `relax` function. Take the following values for the parameter `err` : 10^{-6}, 10^{-7}, 10^{-8}, 10^{-9} and 10^{-10}. Compare the values of $N(L)$ for these relaxation and the one obtained from the matrix method.

Homework 4.10:

a) Output the figure generated by the program `relax_rabbits_simple.py` of coding task 4.4.b .

b) What is the value of `V[xmax]` returned by the program?

