Diffusion

6.1 Learning outcomes

Mathematics and Physics

- Partial differential equations
- Boundary Conditions
- Relaxation methods

Computing and Python

- Using matrices for code optimisation

6.2 Introduction

So far we have modelled populations assuming that the population density does not vary with position and this allowed us to use ordinary differential equations. If we want to model the population spatial displacements one must use differential equations with more than one variable: usually, t, x and y or in other words partial differential equations.

Instead of doing this, we will describe a model which describe a range of systems: diffusion. Diffusion is the process by which a large number of identical objects or particles move randomly independently of each others. This applies to atoms or molecules but also to bacteria and even heat in solids.

We will start by deriving the equations describing diffusion. We will then describe 2 methods to compute the asymptotic solution: the solution to which the system evolves after a large time.

6.3 Diffusion

Many chemical reactions are taking place constantly in our bodies. These reactions occur mostly between organic molecules which are dissolved in the water contained in our cell or blood vessels. Because we are warm, all the molecules in our body are agitated, constantly bumping into each others and this forces them to move randomly. This is the same phenomena as when one puts a drop of ink in a water glass: the ink slowly spreads into the water until the liquid assumes a uniform colour.

We will model diffusion by considering a thin pipe filled with water in which we add some ink. As the pipe is thin, meaning that its diameter is much smaller than its length, we can split it in n small identical segments which we label with an index i. We then denote by $N_i(t)$ the number of ink molecules in segment i of the pipe at time t.

As they are constantly kicked by water molecules, the ink molecules move randomly in all directions and as a result, some of them will move between segments. The number of ink molecules moving from segment i to segment $i+1$ per time interval Δt will be proportional to N_i as well as Δt and so can be written as $k_i^+ N_i \Delta t$. Here k_i^+ is a proportionality constant which will depend on how the ink and water molecules interact with each other, and will typically also depend on temperature. Similarly the number of ink molecules moving from segment i to $i-1$ will be given by $k_i^- N_i \Delta t$. As a result, the total change of N_i will be

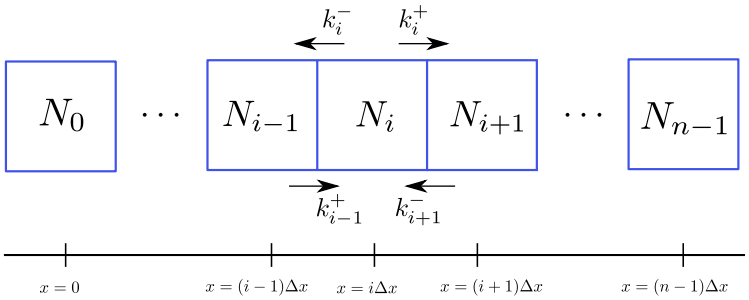


Figure 6.1: Diffusion of ink in a thin pipe. There are n cells with labels 0 to $n-1$. Their midpoints are located at positions $x = 0, \Delta x, \ldots, (n-1)\Delta x$.

given by the difference between what comes in from the two adjacent cells, $(k_{i-1}^+ N_{i-1}(t) + k_{i+1}^- N_{i+1}(t))\Delta t$, and what leaves the cell on both sides $(k_i^+ + k_i^-)N_i(t)\,\Delta t$:

$$\delta N_i(t) = (k_{i-1}^+ N_{i-1}(t) + k_{i+1}^- N_{i+1}(t) - (k_i^+ + k_i^-)N_i(t))\Delta t, \qquad i = 0 \ldots n-1. \tag{6.1}$$

As $\delta N_i(t)$ describes by how much N_i has changed between the time $t + \Delta t$ and t, we can also write

$$\delta N_i(t) = N_i(t + \Delta t) - N_i(t). \tag{6.2}$$

Combining (6.1) and (6.2) we then have

$$N_i(t + \Delta t) = N_i(t) + \Delta t(k_{i-1}^+ N_{i-1}(t) + k_{i+1}^- N_{i+1}(t) - (k_i^+ + k_i^-)N_i(t)). \tag{6.3}$$

This is the diffusion equation in its discrete form. We now compute the Taylor series of $N_i(t + \Delta t)$ to first order in Δt

$$N_i(t + \Delta t) = N_i(t) + \Delta t \frac{\mathrm{d}N_i(t)}{\mathrm{d}t} + \mathcal{O}(\Delta t^2). \tag{6.4}$$

and substitute the result in (6.3) to obtain

$$\frac{\mathrm{d}N_i(t)}{\mathrm{d}t} = k_{i-1}^+ N_{i-1}(t) + k_{i+1}^- N_{i+1}(t) - \left(k_i^+ + k_i^-\right)N_i(t)\,. \tag{6.5}$$

Notice that(6.5) is a a system of ordinary differential equations. This is thus very similar to the equation we have solved before except that the number of equations is much larger and given, in this case, by the number of boxes that we consider.

So far we have assumed that the domain is subdivided into boxes of equal sizes, but in reality such domain do not exist and are only an approximation. To describe the real world, we need to take the limit where the box size, Δx goes to zero. This is called the continuum limit. As the midpoint of the ith box along the x-axis is located at $x_i = i\,\Delta x$, we consider a function $N(t, x)$ which is such that $N_i(t) = N(t, x_i)$. We must then use the Taylor series to evaluate terms such as $N_{i\pm1}(t) = N(t, x_i \pm \Delta x)$ around $N(t, x_i)$ to obtain and equation which only involves $N(t, x_i)$ and its derivatives.

Problem 6.1:

Compute the series of $N_{i\pm1}(t) = N(t, x_i \pm \Delta x)$ to second order in Δx around x_i. The variable t is a constant here, so you only have to compute the Taylor series for the variable x. Assume that the $k_i^\pm$ are all identical to show that, in the limit $\Delta t \to 0$ and $\Delta x \to 0$, (6.5) becomes

$$\frac{\partial N(t,x)}{\partial t} = D\frac{\partial^2 N}{\partial x^2}(t,x)\,. \tag{6.6}$$

where the parameter $D = k\,(\Delta x)^2$ is called the diffusion constant. As we have more than one variable, we have replaced the usual derivative symbol $\mathrm{d}f/\mathrm{d}y$ by partial derivatives: $\partial f/\partial y$. For all practical purposes, this is exactly the same, the meaning being that t does not depend on x and x does not depend on t either.

Copyright © 2015-2023 Bernard Piette and Kasper Peeters; Durham University.

6.4 Boundary Conditions

Before we start solving the differential equation, we must specify what happens at the edges of the finite domain, i.e. for $i = 0$ and $i = n-1$. If we look at the right hand side of eq (6.5) when $i = 0$, we see that it needs N_{-1} which is not defined, and similarly for $i = n-1$ which needs N_n. This means that eq. (6.5) is not properly defined for $i = 0$ and $i = n-1$ and that we must specify separately what N_0 and N_{n-1} are. This is called the boundary condition, and is part of the problem description.

There are two main classes of conditions we can consider: for the boundary on the left end of the tube, we can assume that N_0 is a given constant, or that $N_0 = N_1$ at all times (we have a similar choice for the other end). The first condition means that we have a way to fix the concentration at the end of the domain, as illustrated in the example below. The second condition corresponds to the ink bouncing on the edge of the domain and this is what we would take if the domain was closed (no flow coming from the edges). Notice that as the boundaries of the domain are independent of each other, one can impose different types of boundary conditions on the two sides, if needed.

In our example, chosen to have non trivial solutions, we will assume that our pipe is connected to a large reservoir of pure ink at $i = 0$ and that at $i = n-1$ it is connected to a huge reservoir of pure water. In practice this means that $N_0 = Q$, where Q corresponds to the number of ink molecules per segment volume in the ink reservoir. We also have $N_{n-1} = 0$ which corresponds to the number of ink molecules per segment volume in the pure water reservoir. The idea here is that the reservoirs are so large that the concentrations of ink in them does not change significantly over the time that we will consider.

With these boundary conditions, equations (6.8) are defined for $i = 1 \dots n-2$ while $N_0(t)$ and $N_{n-1}(t)$ are fixed in time, thus leading to a well-posed problem.

6.5 Solution methods

Before we try to solve (6.5) we will try to compute its simplest solution, which is the solution independent of time. The simplest method consists of substituting

$$k_i^{\pm} = \frac{D_i^{\pm}}{(\Delta x)^2} \tag{6.7}$$

into (6.3) to obtain

$$N_i(t+\Delta t) = N_i(t) + \frac{\Delta t}{(\Delta x)^2}\Big(D_{i-1}^{+}N_{i-1}(t) + D_{i+1}^{-}N_{i+1}(t) - (D_i^{+} + D_i^{-})N_i(t)\Big). \tag{6.8}$$

This is nothing but solving (6.5) using the Euler method. It can be shown that applying (6.8) repeatedly will converge towards a constant solution if Δt, which is a free parameter, is not too large ($\Delta t \max_{i,\pm}(D_i^{\pm})/(\Delta x)^2 < 1/2$). This method is called the *Jacobi* method.

In general (6.8) can be written in matrix form as follows

$$\vec{N}(t+\Delta t) = \vec{N}(t) + \nu_J\Big(A \cdot \vec{N}(t) - \vec{B}\Big), \tag{6.9}$$

where, to follow conventional notations, we define $\nu_J = \Delta t/(\Delta x)^2$, called the relaxation parameter. If we consider the very simple case with $D_i = 1$ and $n = 5$, we have

$$\begin{aligned}
N_1(t+\Delta t) &= N_1(t) + \nu_J\big(N_0(t) + N_2(t) - 2N_1(t) - B_1\big), \\
N_2(t+\Delta t) &= N_2(t) + \nu_J\big(N_1(t) + N_3(t) - 2N_2(t) - B_2\big), \\
N_3(t+\Delta t) &= N_3(t) + \nu_J\big(N_2(t) + N_4(t) - 2N_3(t) - B_3\big).
\end{aligned} \tag{6.10}$$

Because the boundary conditions are $N_0(t) = 1$ and $N_4(t) = 0$ for all t, these three equations can be written in the form (6.9) with

$$A = \begin{pmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -2 \end{pmatrix}, \qquad \vec{B} = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}, \qquad \vec{N} = \begin{pmatrix} N_1 \\ N_2 \\ N_3 \end{pmatrix}. \tag{6.11}$$

Copyright © 2015-2023 Bernard Piette and Kasper Peeters; Durham University.

Problem 6.2:

a) Write a Python program called `simple_relaxation.py` to iterate equation (6.9), using (6.11), starting from $N(t=0)=(0,0,0)$. Run it for $\nu_J = 0.1$, $\nu_J = 0.5$ and $\nu_J = 0.6$. In each case display the result for the first 30 iterations. You cab take $\Delta t = 1$ so that t becomes an integer index for labelling the iterations.

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

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

Check Point 6.1

In the problem 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, i.e. $A = -2 \times \mathbb{1} + M$, then equation (6.9) for $\nu_J = 1/2$ becomes

$$\vec{N}(t+\Delta t) = \frac{1}{2}\Big(M \cdot \vec{N}(t) - \vec{B}\Big)\,, \tag{6.12}$$

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.

The program `relax_jacobi_1d.py` implements the Jacobi relaxation method to solve a system of equations of the type

$$\frac{\mathrm{d}N_i(t)}{\mathrm{d}t} = f_i\big(t, N_1(t), \ldots, N_{n-1}(t)\big)\,, \quad i = 1 \ldots n-1\,. \tag{6.13}$$

The module contains at the end a simple class example corresponding to the diffusion equation with constant diffusion constant $D_i = 1$. In that example, the boundary conditions are chosen to be $N_0(t) = 1$ and $N_{n-1}(t) = 0$. This program also uses the parameter `nu` instead of $\Delta t/(\Delta x)x^2$. The structure of the program is very similar to the program we have used so far to solve systems of ordinary differential equations:

- The class variable `v` keeps the value of the function (N_i in (6.13)).
- The class function `F(v,i)` computes the right hand side of the equation for the index `i`, or in other words $f_i(t, N_1, \ldots, N_{n-1})$ in (6.13). `v` is an array containing the values of N and `i` the index of the equation that must be evaluated.
- The class function `boundary()` implements the boundary conditions. (see the example below the class definition).
- The class function `relax_1_step(nu)` performs a simple relaxation step using the relaxation parameter `nu`. It updates the class variable `self.v` and return the largest value by which `v` has changed.
- The class function `relax(self, err, nu)` calls the function `relax_1_step(nu)` until the value it returns is smaller than `err`. The function returns the number of iterations that it has performed.
- As before, the plot function displays the final value of `v` as a function of the index value i.

Problem 6.3:

Run the program `relax_jacobi_1d.py` and use the numerical solution to determine the analytic solution to the equation. Try the following values of `nu`: 0.25, 0.5 and 0.6 What difference does it make?

In the Jacobi method we first compute, for all i, the right-hand side of (6.13), and then update all the N_i. Instead, we could compute the right hand side for $i = 1$, use (6.13) to update N_1, and then immediately use that updated value to compute the right-hand side for $i = 2$. This small modification is called the *Gauss-Seidel* method.

Copyright © 2015-2023 Bernard Piette and Kasper Peeters; Durham University.

Problem 6.4:

Modify the program `relax_GS_1d.py` which defines the class `RelaxGS1D` as a subclass of `RelaxJacobi1D`, but where the class function `relax_1_step` still implements the Jacobi method. Modify it so that it implements the Gauss-Seidel relaxation instead. Only 3 lines in `relax_1_step` need to be changed.

How many iterations are now needed before the program stops (take $\nu = 0.5$)?

Check Point 6.2

Problem 6.5:

Write a program called `diffusion_tube.py` so that it computes the static solutions of

$$\delta N_i(t) = (D_{i-1}N_{i-1}(t) + D_{i+1}N_{i+1}(t) - 2D_iN_i(t))\Delta t, \qquad i = 0 \ldots n-1, \qquad (6.14)$$

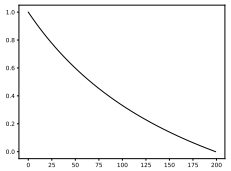
using the Gauss-Seidel method for a diffusion constant D that varies linearly from $D = 1$ for $i = 0$ to $D = 2$ for $i = n - 1$ (in other words $D_i = 1 + i/(n-1)$). This could correspond to a tube that widens as we move from the ink tank to the water tanks.

You should proceed as follows:

- Create a new file called `diffusion_tube.py`.
- Import the modules `numpy` as `np`, `matplotlib.pyplot` as `plt` and `relax_GS_1d` as `gs1d`.
- Create a new class called `diffusion_tube` as a subclass of `RelaxGS1D`.
- Define the class function `__init__(self,v0,Dmin,Dmax)` where `v0` is the initial value, `Dmin` the diffusion constant at $i = 0$ and `Dmax` the diffusion constant at $i = n - 1$. The function must first call the parent class initialiser. It must also set the class variable `self.D` to be an array with the same number of elements as `v0` and with values ranging linearly from `Dmin` to `Dmax`. (Use the numpy function `linspace`)
- Define the class function `F(self, v,i)` to implement the right hand side of (**??**), using `self.D` as the diffusion constant.
- Define the class function `boundary` so that it sets the first value of `self.v` to 1 and the last one to 0.
- Add the following lines below the class definition:

```
if __name__ == "__main__":
  Np=200
  relax = diffusion_tube(np.zeros([Np]),1.,2.)
  n = relax.relax(1e-9,0.5)
  print("No of iterations: ",n)
  relax.plot()
  plt.savefig("diff-tube.pdf")
  plt.show()
```

This creates an instance of `diffusion_tube`, using an empty array of 200 element as the initial condition and taking `Dmin=1` and `Dmax=2`. It the solves the equation and generate a figure for the solution.



Profile of N_i in a tube of varying area.

Problem 6.6:

How does the solution of the tube with a varying cross section differ from the solution for a tube with a constant cross section $(D = 1)$? Can you interpret the result?

Check Point 6.3

Copyright © 2015-2023 Bernard Piette and Kasper Peeters; Durham University.

Problem 6.7:

The program `run_diffusion_tube.py` uses the class `diffusion_tube` from the program `diffusion_tube.py` to solve the equation for a non constant diffusion parameter. It then compute the average value of the temperature and the variance as well as generate a figure of the temperature relative to its average value. The program is very slow and hard to read, but it can be greatly improved. (The time needed to compute the diffusion equation can't be improved, but computing the average and variance can be much faster. Hint: use `numpy` to get rid of all the loops).

Modify the program `run_diffusion_tube.py` to make it faster and easier to read. The lines

```
Np=500
difftube = dt.diffusion_tube(np.zeros([Np]),1.,2.)
# and relax the solution
n = difftube.relax(1e-9,0.5)
```

must be left unchanged, but you might want to replace the value `1e-9` by `1e-5` when you test your code to make the first part of the program faster (the solution is not very good, but for testing it does not matter).

Problem 6.8:

The program `relax_jacobi_1d_np.py` is nearly identical to `relax_jacobi_1d.py`. The only difference is that it defines the class `RelaxJacobi1D_np` where `relax_1_step` does not loop over the different elements of the array one by one, but instead use the arithmetic features of `numpy`.

The program `relax_jacobi_1d_np_diff.py` imports the module `relax_jacobi_1d_np` and creates a subclass of `RelaxJacobi1D_nd` called `RelaxDiffusion`.

Complete the definition of the function `F(self,v)` in the class `RelaxDiffusion` so that it returns `v[i+1]+v[i-1]-2*v[i]` as an array of `Np-2` elements which excludes the end points of the grid (index `0` and `Np-1`). Before solving this problem you can go back to the numpy problem sheet or read the second section of `debugging` both of which explain how to perform arithmetic computations with arrays efficiently.

Compare the speed of execution of `relax_jacobi_1d_np_diff.py` with `relax_jacobi_1d.py` and `relax_GS_1d.py`. Do you see the advantage of using `numpy`?

Could you modify `relax_GS_1d.py` in the same way?

Check Point 6.4

Copyright © 2015-2023 Bernard Piette and Kasper Peeters; Durham University.