Numerical solution of PDEs
Assignment 2

Department of Physics, University of Surrey module:
Energy, Entropy and Numerical Physics (PHY2063)

1 Numerical Physics part of Energy, Entropy and Numerical Physics

This numerical physics course is part of the second-year Energy, Entropy and Numerical Physics (PHY2063) module, and is online at the EENP module on SurreyLearn. See there for assignments, deadlines etc. The course is about numerically solving ODEs (ordinary differential equations) and PDEs (partial differential equations), and introducing the Monte Carlo technique and Bayesian statistics.

This assignment is on PDEs. PDEs are very common in physics. Maxwell's equations that govern electromagnetism are PDEs, as is Schrödinger's equation in quantum mechanics. Heat and particle diffusion obey PDEs, as do waves.

2 Introduction

This assignment aims to provide insight into how second-order partial differential equations (PDEs) are solved on a computer. We will only consider second-order PDEs as these are by far the most common in physics.

In the first half of the assignment we will consider an example of a time-independent PDE, in two dimensions, while in the second half we will consider an example of a time-dependent PDE, in one dimension. The time-independent PDE is the linearised Poisson-Boltzmann equation, which is a model for the distribution of charged ions (e.g., salt ions) in a solution (e.g., in salt solutions such as sea water or the fluids inside our bodies). We will solve for the potential distribution round a segment of our DNA. Then we will consider the diffusion PDE as an example of a time-dependent PDE. Diffusion is a very common phenomenon in physics. Electrons diffuse in metals and semiconductors, molecules diffuse in liquids, etc., and thermal energy also moves via diffusion in solids.

3 Linearised Poisson-Boltzmann PDE in two dimensions

This assignment starts off with a PDE that combines electromagnetism (as taught in the two EM modules in second year) with statistical physics (as taught in Energy, Entropy and Numerical Physics module). When we consider charged ions at thermal equilibrium we need to consider both electromagnetism and statistcal physics.

We will solve, in a simple geometry, the linearised Poisson-Boltzmann equation. To see where this PDE comes from, we start from one of Maxwell's equations. This is called the Poisson equation:

$$\nabla^2\phi(\mathbf{r}) = -\frac{\rho(\mathbf{r})}{\epsilon} \qquad \text{Poisson equation}$$

with $\phi(\mathbf{r})$ the electrostatic potential at point $\mathbf{r}$, $\rho(\mathbf{r})$ the charge density at the same point, and ϵ the permittivity. This PDE cannot be solved unless we can relate the charge density ρ to the potential ϕ, as Poisson's equation is one equation with two unknown functions (ϕ and ρ), which is one too many. The relationship between ρ and ϕ will depend on the system.

Let us consider salty water (e.g., brine, sea water, the solutions inside our arteries, veins and cells. Plasmas at thermal equilibrium are similar). Salty water is just ions in water. Water has an effective permittivity of $\epsilon \simeq 80\epsilon_0$, i.e., 80 times the permittivity of vacuum. In the water are positive and negative ions, e.g., in sea water there are mainly sodium, Na^+ and chloride, Cl^-, ions. It is these ions that contribute to the charge density ρ (water molecules are neutral).

From statistical physics we know that at thermal equilibrium probabilities are proportional to Boltzmann weights, i.e., $\exp[-\epsilon/kT]$ for a state of energy ϵ. Here the electrostatic energy of a sodium ion at point $\mathbf{r}$ is $e\phi(\mathbf{r})$, because the ion has a charge of +e. Thus the charge density at a point due to sodium ions is approximately $ce\exp[-e\phi(\mathbf{r})/kT]$, for c the concentration of sodium chloride in the water. The charge density due to chloride ions is $-ce\exp[+e\phi(\mathbf{r})/kT]$. Note that the average concentrations of sodium and chloride ions have to be identical in order that the volume has zero net charge. Here their average concentrations are c. In salt water c is about 1 pair of sodium and chloride ions per 10 nm^3.

If we put these Boltzmann weight expressions into Poisson's equation we get the Poisson-Boltzmann equation

$$\nabla^2\phi(\mathbf{r}) = -\frac{ce}{\epsilon}\left(\exp\left[\frac{-e\phi(\mathbf{r})}{kT}\right] - \exp\left[\frac{e\phi(\mathbf{r})}{kT}\right]\right) \qquad \text{Poisson-Boltzmann equation}$$

This is often soluble but here we will make life simpler by linearising it. We expand out the exponentials and truncate after the linear terms

$$\nabla^2\phi(\mathbf{r}) = -\frac{ce}{\epsilon}\left(1 - \frac{e\phi(\mathbf{r})}{kT} - 1 - \frac{e\phi(\mathbf{r})}{kT}\right)$$

or

$$\nabla^2\phi(\mathbf{r}) = \frac{2ce^2}{\epsilon kT}\phi(\mathbf{r})$$

The prefactor in front of ϕ on the right-hand side has dimensions of one over a length squared. This length is usually written as κ^{-1} and is called the Debye length, after Peter Debye. Debye was a Dutch physicist from the last century who worked on this problem. Then we can rewrite the linearised Poisson-Boltzmann equation as

$$\nabla^2\phi(\mathbf{r}) = \kappa^2\phi(\mathbf{r}) \qquad \kappa^2 = \frac{2ce^2}{\epsilon kT}$$

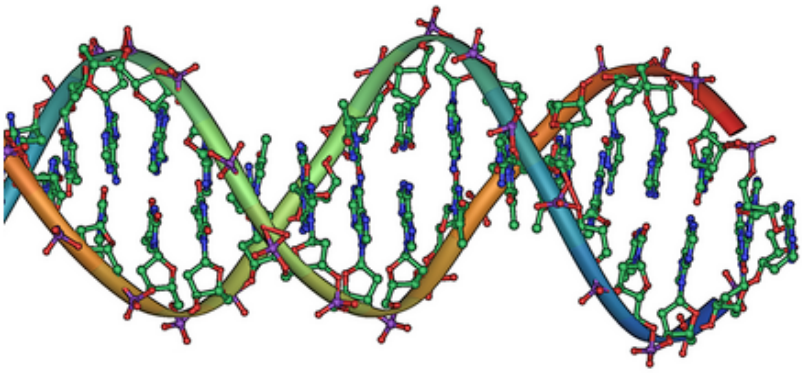


Figure 1: A schematic of the DNA double helix (from Wikimedia). The outer spirals are what is called the backbone of DNA (it is this part of the molecule that holds DNA together). They contain phosphate groups which are negatively charged. DNA therefore has a very large negative charge per unit length, along its double helix. Indeed it is this charge that makes DNA water soluble. Although as you can see from the schematic there are spiral groves in the double helix, to a first approximation DNA is often treated as a smooth cylinder 2 nm in diameter. On lengthscales of nanometres and tens of nanometres, DNA is quite rigid (it can bend on larger lengthscales), so on these short lengthscales not only can DNA be modelled as a cylinder, it can be modelled as a rigid cylinder. Inside the spiralling backbone are the base pairs (AT and CG pairs) that encode our genes. There are approximately three base pairs per 1 nm of length along the double helix. So the DNA shown is about 5 nm in length. Our chromosomes are about 1 cm long when stretched out to their full length, i.e., much much longer.

Approximate formulas for the derivatives in 2D

We want to write the derivatives at a point (x, y) in terms of values of the function at adjacent points. We will solve for values of the function $\phi(x, y)$ on a grid. This is illustrated in Fig. 2 where we have drawn the grid and indicated (with black circles) the points where the values of $\phi(x, y)$ are stored. In two dimensions, we subdivide the x and y axes into squares h by h in size and then $\phi(x, y)$ will be represented in the computer by a two-dimensional array of real numbers: the values of $\phi(x, y)$ at the points of the grid.

To obtain the derivatives, we start from the Taylor expansion at the point (x, y) in the x-direction

$$\phi(x + h, y) = \phi(x, y) + \left(\frac{\partial \phi(x, y)}{\partial x}\right) h + \frac{1}{2}\left(\frac{\partial^2 \phi(x, y)}{\partial x^2}\right) h^2 + \cdots$$

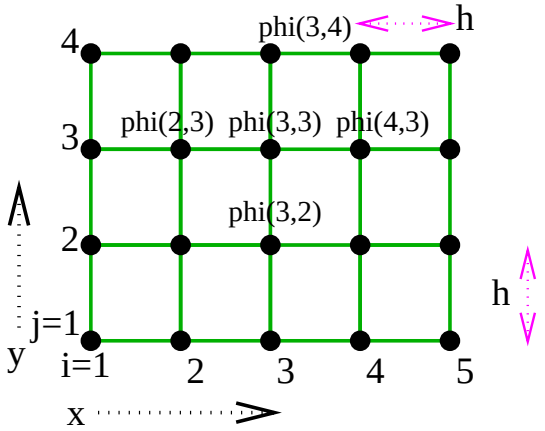
where both derivatives are evaluated at the point (x, y). We denote the first derivative of ϕ with respect to x at the point (x, y) by $\phi_x(x, y)$, and the second derivative with respect to x by $\phi_{xx}(x, y)$. Then

$$\phi(x + h, y) = \phi(x, y) + \phi_x(x, y)h + \frac{1}{2}\phi_{xx}(x, y)h^2 + \cdots$$

The corresponding equation for the function ϕ at $\phi(x - h, y)$ is

$$\phi(x - h, y) = \phi(x, y) - \phi_x(x, y)h + \frac{1}{2}\phi_{xx}(x, y)h^2 + \cdots$$

Figure 2: A schematic of a grid of points in the xy-plane, where the spacing between the points is h in both the x and y directions. At the grid points the values of the function $\phi(x, y)$ are known and are the elements of a Fortran two-dimensional array `phi(i,j)`. Here x is distance along the x-axis and i is an integer that denotes the array element along the x-axis. As the spacing is h along the x-axis we have that at $x = 0$, $i = 0$, at $x = h$, $i = 1$, at $x = 2h$, $i = 2$, etc. Similarly at $y = 0$, $j = 0$, at $y = h$, $j = 1$, etc. Shown is `phi` at the grid point $x = 3h$, $y = 3h$, i.e., at $i = 3$, $j = 3$, together with its four neighbours: the grid points north, south, east and west of it.



Now, we want the second derivative. For this we add the equations for $\phi(x-h, y)$ and $\phi(x+h, y)$

$$\phi(x + h, y) + \phi(x - h, y) = 2\phi(x, y) + \phi_{xx}(x, y)h^2 + \cdots$$

We can easily rearrange this to give us an equation for the second derivative

$$\phi_{xx}(x, y) = \frac{\phi(x + h, y) - 2\phi(x, y) + \phi(x - h, y)}{h^2}$$

which is the expression we need. It gives us the second derivative of $\phi(x, y)$ in terms of the values of $\phi(x, y)$ at the grid points. In terms of the numbers of the array elements, i and j, this becomes

$$\texttt{phi\_xx(i,j)} = (\texttt{phi(i+1,j)} - 2.0 * \texttt{phi(i,j)} + \texttt{phi(i-1,j)})/\texttt{h**2}$$

The expression for the second derivative with respect to y is completely analogous. It is

$$\texttt{phi\_yy(i,j)} = (\texttt{phi(i,j+1)} - 2.0 * \texttt{phi(i,j)} + \texttt{phi(i,j-1)})/\texttt{h**2}$$

Now that we have the two second derivatives we can use these expressions to numerically evaluate the Laplacian in two dimensions.

4 Potential around the DNA double helix

In this part of this assignment, we are going to solve the linearised Poisson-Boltzmann PDE in a simple, two-dimensional geometry. The geometry is that of the cross-section of a long cylinder, which is a simple circle in two dimensions. The cylinder is a simple model of a segment of DNA, see Fig. 1. The long (in comparison to its radius) cylinder we want to study has radius r_{DNA}, is charged, and has a defined constant potential, $\phi(r_{DNA}) = \phi_S$ at its surface. This potential is with

respect to the potential at infinity, i.e., we set the potential at $r \to \infty$ to be zero: $\phi(r \to \infty) = 0$. Thus the BCs are the values of ϕ on the outside surface of a cylinder, which in two dimensions is a circle of radius r_{DNA}, and at infinity, and we want to solve the PB equation between these boundaries. In practice, we need a finite system for a numerical study, so will set the box size to be large but not infinite. Also, as the cylinder has rotational symmetry the potential $\phi(r)$ will also only depend on r, the distance from the centre of the cylinder, but here we will use the Cartesian coordinates x and y.

The cylinder is a simple model of DNA, the molecule in our cells that our genes are made of. DNA forms the double helix structure discovered by Watson and Crick, see Fig. 1. Each of our chromosomes is a very long such double helix, which has a radius of about $r_{DNA} = 1$ nm. On the nanometre lengthscales we will study this double helix is quite rigid so we will treat it as a perfectly rigid cylinder. Of course as a helix it also has a spiral structure but for simplicity we will neglect that. We will work in units of nanometres, as these are convenient here as the distances are all of this order.

4.1 Numerical solution is in the form of an array

As with ODEs we will not determine the function $\phi(x, y)$ at all points x and y but will discretise space into points h apart, see Fig. 2 for a schematic. In two dimensions this means we are solving for a two-dimensional array `phi(i,j)` which is the function on a square grid of grid spacing h. For small h this will be a good approximation to the function.

For example, if we want the potential ϕ at $h = 0.1$ nm intervals over a rectangular area of length 20 nm in the x and y directions, and centred on the origin, we need a grid of 201 x-values and 201 y-values,

```
integer,parameter ::  n=100
real ::  phi(-n:n,-n:n)
```

4.2 Boundary conditions

We are going to be using what is called a relaxation method of solving a time-independent PDE. This starts from an initial guess, which we then iterate towards the solution. The specific relaxation method we will be using is called Gauss-Seidel iteration.

Gauss-Seidel iteration starts with a guess at the solution; which should include the imposed BCs. This can be a simple guess, e.g., that each array element is zero, except for those at the surface of the cylinder which are at ϕ_S.

To do this we start by setting the whole array to zero

```
phi=0.0
```

As the outer BC is $\phi = 0$, with this starting guess we then don't need to impose the outer BC as we have already set ϕ to zero there. We do however need to set ϕ at the surface of the cylinder. In fact it is easiest to set all elements of the array `phi`, that are inside a cylinder of radius `r_dna` to the BC value, `phi_s`. To do this we just run over the complete array checking r and setting all elements inside the cylinder to `phi_s`

```
do i=-n,n
  do j=-n,n
    x=real(i)*h
    y=real(j)*h
    if(x**2+y**2<r_dna**2+tolerance)phi(i,j)=phi_s
  enddo
enddo
```

where `tolerance` is a small number, 10^{-5} will do, that we add to accommodate rounding errors in the numerical arithmatic. For example, if say we are at the point which should be $x = 1$ and $y = 0$, but due to rounding errors $x = 0.999999$, then due to this tiny error $x^2 + y^2 < 1$ when $x^2 + y^2$ should equal 1. Now we have an initial guess at the solution. This is not the true solution. Although it does satisfy the BCs, it is not a solution to the PDE. We now use Gauss-Seidel iteration to converge this guess to the true solution.

4.3 Gauss-Seidel iteration

To iterate we need an expression that allows us to update each array element `phi(i, j)`. To do this we will obtain an equation that relates element `phi(i, j)` to the surrounding elements. In Cartesian coordinates in two dimensions (x and y), the Laplacian is just the sum of the second derivative with respect to x, ϕ_{xx}, and the second derivative with respect to y, ϕ_{yy}. So we can write the linearised Poisson-Boltzmann equation as

$$\nabla^2\phi(\mathbf{r}) = \phi_{xx}(\mathbf{r}) + \phi_{yy}(\mathbf{r}) = \kappa^2\phi(\mathbf{r})$$

Using the expressions for the derivatives that we get from truncated Taylor expansions (see box), we have

$$\frac{\phi(x+h,y) - 2\phi(x,y) + \phi(x-h,y)}{h^2} + \frac{\phi(x,y+h) - 2\phi(x,y) + \phi(x,y-h)}{h^2} = \kappa^2\phi(x,y)$$

Multiplying by h^2, and rearranging to get an expression for $\phi(x, y)$, we get

$$\phi(x,y) = \frac{\phi(x+h,y) + \phi(x-h,y) + \phi(x,y+h) + \phi(x,y-h)}{4 + h^2\kappa^2}$$

or in terms of arrays

$$\texttt{phi(i, j)} = (\texttt{phi(i + 1, j)} + \texttt{phi(i - 1, j)} + \texttt{phi(i, j + 1)} + \texttt{phi(i, j - 1)})/(\texttt{4.0} + \texttt{h} * * \texttt{2} * \texttt{kappa} * * \texttt{2})$$

I will not go into the theory behind this but note that *if* the array `phi` has elements that satisfy this equation then that array is a solution to the PDE (because the equation is just the PDE, rearranged). The initial guess will not satisfy this set of equations, which means that when we calculate the right-hand side and put it in the left-hand side, the left-hand side will change. With a bit of luck[1], this change will take the array nearer the true solution, and applying it repeatedly takes the array nearer the solution.

4.3.1 Iteration

Starting from the initial guess, we now iteratively improve on our initial guess, using the equation we derived for `phi(i, j)` in terms of `phi` on the four neighbouring grid points. In each iteration, we need to loop over all interior points of the grid, calculating at each point the next estimate of the solution at point (i, j) from the current values at the four adjacent grid points, i.e. schematically

[1] Mathematicians have obtained expressions for under what conditions this iteration works, but we will not go into those details here

```
flag=0
do i=-n+1,n-1
  do j=-n+1,n-1
    oldval=phi(i,j)
    x=real(i)*h
    y=real(j)*h
    if(x**2+y**2<r_dna**2+tolerance)then
!  we are inside cylinder so do not update elements
    else
      phi(i,j)= (phi(i+1,j)+phi(i-1,j)+phi(i,j+1)+phi(i,j-1))/(4.0+h**2*kappa**2)
      if(abs(oldval-phi(i,j)) > accuracy) flag=1
    endif
  enddo
enddo
```

Note that when we loop over the array we need to avoid changing the values of the `phi` elements at the boundaries, because these need to remain at the values already set by the BCs. Thus the loops are from $-n+1$ to $n-1$, which avoids the outer edge elements, that then remain at their set BC values. Also, the if statement ensures that the set values of `phi` on the DNA cylinder remain at their BC values.

This looping over the complete `phi` array must be repeated many times, and at each iteration the array `phi` should get closer to the true solution of the PDE. So the above two do loops need to be nested within a third loop to iterate.

We want the solution to some high accuracy. To do this we note that as the array `phi` converges to the true solution the changes in the elements at each iteration get smaller and smaller. When `phi` is very close to the true solution, another iteration only changes the array very little. Thus we want to exit the outer iteration do loop when each iteration hardly changes `phi`, which we define as no element changing by more than some small number, called `accuracy`. We can set `accuracy` to a small value, say $\texttt{accuracy} = \texttt{1.0e} - \texttt{3}$.

We can then iterate until we get an accurate solution as follows. As we loop over the array elements, we store each pre-iteration value $\texttt{phi}(\texttt{i},\texttt{j})$ in `oldval` and then after each $\texttt{phi}(\texttt{i},\texttt{j})$ has been updated, we check to see if it has changed by more than `accuracy`. If the element has changed by more than the accuracy has we set $\texttt{flag} = \texttt{1}$, which is our way of saying that we need another iteration. If at the end of the do loops over i and j, $\texttt{flag} = \texttt{0}$ we know that we have updated all points in the area of interest, and that none of elements of `phi` has changed more than the value of `accuracy`. So, then we can stop iterating.

It is a good idea for your program to count the number of iterations needed to reach the required accuracy of solution, and print it out to the screen.

Task 1

The task is to determine the electrostatic potential in the xy plane, $\phi(x, y)$, in the presence of a DNA double helix aligned with its axis along the z axis. The double helix is modelled as a very long rigid charged cylinder, of radius $r_{DNA} = 1$ nm, parallel to the z axis and at the origin in the xy plane. See Fig. 1 for a schematic of the DNA double helix. As the cylinder is assumed to be much larger than a few nanometres long (this is reasonable, DNA molecules are almost always much longer than this) the problem reduces to a two-dimensional problem, as the potential does not change with distance parallel to the double helix (along the z axis). Also, as the cylinder has rotational symmetry it will also only depend on r, the distance from the centre of the cylinder, but here we will use the Cartesian coordinates x and y. We will work in units of nanometres, as these are convenient here as the distances are all of this order. The potential should be obtained as an array of real numbers, with a spacing of $h = 0.05$ nm.

Our bodies are at $T = 37°\text{C} = 310$ K, for water $\epsilon \simeq 80\epsilon_0$, and the salt concentration is about $3 \times 10^{26}/\text{m}^3$. From these numbers determine the value of κ, and hence the value of the Debye length κ^{-1}.

Assume that the surface of the DNA cylinder is at a fixed potential $\phi = 100$ mV higher than far away from the DNA, i.e., take the BCs to be $\phi(r = 1\text{ nm}) = 100$ mV and $\phi(r \to \infty) = 0$ mV. You should continue calculations until the potential at every point changes by less than 10^{-3} mV in one iteration. The array `phi` should have `n = 80`, i.e., go from $-80h = -4$ nm, to $+4$ nm. Your program should count the number of iterations needed to reach the required accuracy of solution, and write this number to the screen. The solution should be written to a file (as always file name needs to be written to screen).

At the edges of the profile the array elements `phi(i, j)` will be very close to zero, and so changing by amounts so small that you may see the warning: "Warning: Floating underflow occurred". You can ignore this warning, it is telling you some elements are changing by amounts so small that they are zero to the precision you are working at. As the potential is very close to zero near the edges of the region this is to be expected.

Checking the solution

Solutions should always be checked. One good way to do this is to plot the solution with gnuplot. Gnuplot will do 3D plots, using the command 'splot'. If you write three columns, x, y and $\phi(x, y)$ to a file, then in gnuplot the command

```
gnuplot> splot 'name_of_datafile'
```

will produce a type of 3D plot of the data. There are several other types of 3D plot that you can do. You may want to search for help with 3D plotting (i.e., Google it) as 3D plotting is a bit more involved than 2D plotting. This 3D plot will enable you to see if the solution looks sensible.

A more quantitative way to check the answer uses a 2D plot of $\phi(r)$. This is easy, just fix $y = 0$ and plot $\phi(x, y = 0)$ as this is equivalent to $\phi(r)$. Just write two columns to a file, x and ϕ, and plot that. The linearised Poisson-Boltzmann PDE we have solved is from a class of PDEs called Helmholtz PDEs, and the solutions in 2D are a type of function called Bessel functions (named after the German physicist Friedrich Bessel). The solution here is close to a function called $K_0(s)$, a modified Bessel function of the second kind. This in turns is roughly $K_0(s) \sim \exp(-\xi s)/s^{1/2}$. Here the decay coefficient $\xi = \kappa$, so you should find that outside the cylinder, the solution is close to $\exp(-\kappa r)/r^{1/2}$. Gnuplot will fit functions to data, you can try to fit a function of that form to your profile.

Task 1A

In the ELECTROMAGNETISM, SCALAR AND VECTOR FIELDS (PHY2064) module, we are also doing PDEs. In that module, we looked at Laplace's equation

$$\nabla^2 \phi = 0$$

which is the zero salt, and so $\kappa = 0$, limit of the linearised Poisson Boltzmann equation we solved in Task 1. In two dimensions we saw that the general solution to Laplace's equation is

$$\phi(r) = A + B \ln(r)$$

for A and B two constants, i.e., the solution varies as $\ln(r)$.

Take the program for Task 1 and set $\kappa = 0$. As the solution now varys very slowly at large values of r (because $\ln(r)$ varies slowly with r at large r), initially increase the area studied by increasing `n` from 80 to 200. You should then see that the decay of $\phi(r)$ with r scales as $B \ln(r)$. To show this

you can use gnuplot to fit a function of the form $A + B\ln(r)$ to the calculated $\phi(r)$. The numerically calculated $\phi(r)$ will only be approximate due to this function's very slow decay. To obtain a very accurate numerical solution very large grids and many iterations are needed.

5 Mathematics of diffusion in one dimension

If in a solid object the temperature is not uniform, i.e., it is hot in one place and colder in another, then diffusion is the process whereby thermal energy moves from hot regions to cold regions, and so ultimately makes the temperature uniform. As you have learnt or will learn in the EENP lectures, at equilibrium the temperature is uniform. The diffusion of thermal energy is described by the diffusion PDE. The diffusion PDE for the temperature T is

$$\frac{\partial T(\mathbf{r},t)}{\partial t} = D\nabla^2 T(\mathbf{r},t)$$

Here D is the diffusion constant for the temperature. D has dimensions of length squared over time. For diffusion in one dimension, along the x axis, this simplifies to

$$\frac{\partial T(x,t)}{\partial t} = D\frac{\partial^2 T(x,t)}{\partial x^2}$$

5.1 BCs for the diffusion PDE

To obtain a particular solution we need boundary conditions (BCs). An example set of BCs is that at $t = 0$ the temperature is 20°C within 1 cm of the origin and 0°C everywhere else along the x axis. We assume that the system is effectively infinite along the x axis, i.e., that heat can diffuse out to $x \to \pm\infty$. Then the BCs are

$$T(x, t=0) = \begin{cases} 0°\text{C} & x < -1 \text{ cm} \\ 20°\text{C} & -1 \le x \le 1 \text{ cm} \\ 0°\text{C} & x > 1 \text{ cm} \end{cases}$$

These BCs plus the diffusion PDE allow us to calculate $T(x,t)$ at all times.

Note that these BCs are initial conditions, i.e., the function at $t = 0$. They are what we have if we know what the temperature profile is at one instant, and want to know what it will look like in the future. These are very common BCs.

6 Numerical solution of the diffusion PDE in one dimension

Now that we know the mathematics we need to solve, we will look at how we will numerically solve this PDE.

6.1 Numerical solution is in the form of an array

We will not determine the function $T(x)$ at all values of x, but will again discretise space into points h apart. In one dimension this means that we are solving for a one-dimensional array `temp(i)`. For small h this will be a good approximation to the function. This array, i.e., its values, will depend on time t.

For example, if we want the temperature T at $h = 0.1$ cm intervals over a rectangular area of total length 80 cm in the x direction, and centred at the origin, we need a grid of 801 x-values,

```
integer,parameter ::  n=400
real ::  temp(-n:n)
```

for `temp` the array of values of the temperature. Note that here we want to model a system that is infinite along the x axis, but of course we cannot have an infinite array. So we need to define the array to be big enough so that the during the time interval we are interested in, the temperature diffusion is confined to the region of the x axis where we do have points. As diffusion tends to spread out the profile, an array of this size will be big enough only for diffusion over not too long a time.

6.2 Boundary conditions on the array

Here the BCs are the initial temperature distribution, i.e., the initial values of the array elements. We will apply the above BCs. To do this, we first set all array elements to zero, with

```
temp=0.0
```

and then set the elements between -1 cm and $+1$ cm to $T = 20°$C, using

```
do i=-10,10
   temp(i)=20.0
enddo
```

which will work for $h = 0.1$ cm, as then there are 10 elements in the range 0 to 1 cm.

Approximate formulas for the derivatives in one dimension

When we solve a PDE in one dimension (x), the function, $T(x)$, we are solving for is a one-dimensional array. From the numbers in this array, we want to determine the second-derivative $\partial^2 T/\partial x^2$. We start with Taylor expansions.

The Taylor expansion expression for the value of the function T at the point $x + h$, using a Taylor expansion about the point x, is

$$T(x+h) = T(x) + \left(\frac{\partial T(x)}{\partial x}\right)_x h + \frac{1}{2}\left(\frac{\partial^2 T(x)}{\partial x^2}\right)_{xx} h^2 + \cdots$$

where both derivatives are evaluated at the point x. We denote the first derivative of T by $T_x(x)$, and the second derivative by $T_{xx}(x)$. Then

$$T(x+h) = T(x) + T_x(x)h + \frac{1}{2}T_{xx}(x)h^2 + \cdots$$

The corresponding equation for the function T at the point $x - h$ is

$$T(x-h) = T(x) - T_x(x)h + \frac{1}{2}T_{xx}(x)h^2 + \cdots$$

If we add the equations for $T(x - h)$ and $T(x + h)$, we get

$$T(x+h) + T(x-h) = 2T(x) + T_{xx}(x)h^2 + \cdots$$

We can easily rearrange this to give us an equation for the second derivative, which is what we need. It is

$$T_{xx}(x) = \frac{T(x+h) - 2T(x) + T(x-h)}{h^2}$$

In terms of the array elements, labelled i, this becomes

$$\texttt{temp\_xx(i)} = (\texttt{temp(i+1)} - 2.0 * \texttt{temp(i)} + \texttt{temp(i-1)})/\texttt{h} * *2$$

6.3 Numerical form of the Laplacian in one dimension

In one dimension the Laplacian of $T(x, t)$ is just

$$\frac{\partial^2 T}{\partial x^2} = T_{xx}(x,t) = \frac{T(x+h,t) - 2T(x,t) + T(x-h,t)}{h^2}$$

using our formula for derivatives in the boxed section. In terms of the numbers of the array elements, i this becomes

$$\texttt{temp\_xx(i)} = (\texttt{temp(i+1)} - 2.0 * \texttt{temp(i)} + \texttt{temp(i-1)})/\texttt{h} * *2$$

where we call the temperature variable `temp` and the second derivative of the temperature with respect to x, `temp_xx`.

6.4 Numerical form of the diffusion equation in one dimension

In one dimension the diffusion equation is

$$\frac{\partial T(x,t)}{\partial t} = DT_{xx}(x,t)$$

Here D is the diffusion constant for temperature (dimensions of length squared over time). Converting this to code is easy, it is just

$$\texttt{dtempdt(i)} = \texttt{diffc} * \texttt{temp\_xx(i)}$$

or

$$\texttt{dtempdt(i)} = \texttt{diffc} * (\texttt{temp(i+1)} - 2.0 * \texttt{temp(i)} + \texttt{temp(i-1)})/\texttt{h} * *2$$

where `dtempdt` is the time derivative of the temperature, and `diffc` is the diffusion constant, D. This allows us to calculate the values of the time derivative of the temperature at every point i. Thus `dtempdt` will need to be defined as an array. One way to do this, is by

```
real ::  dtempdt(-n:n)=0.0
```

Here, in one line we have both defined the array, and set all the elements to zero. You can do this like in the line above, or define the array in one line, and then later set it to 0. Iit is good practice to zero arrays early in the program, as then you are sure that you know what the values are, i.e., zero. If you do not do this and attempt to use a variable or array element that is not been set by you, it can contain any value, which will cause problems.

Once we have the time derivative in `dtempdt`, we can integrate the PDE forward in time using the Euler method (as we used for ODEs) at each point

$$\texttt{temp(i)} = \texttt{temp(i)} + \texttt{dt} * \texttt{dtempdt(i)}$$

for `dt` a small time step. Note that for PDEs we need a small step size along the x axis, `h`, *and* a small step size along the time axis, `dt`.

6.5 Structure of the program

The structure of the program is simple. We need an outer do loop over time that integrates forward in time from the start at $t = 0$, up until the end time t_{end}, in n_t steps of small size. Nested inside this loop we have two do loops over x, one following after the other. The first one calculates the values of $\partial T/\partial t$ at each position (i.e., the array `dtempdt`), and then the second do loop uses these values in the Euler method to calculate the temperature at all x values, at the next time step.

6.6 Step sizes and errors

As with ODEs, our solution is approximate. Here the size of the error depends on both h and δt. At a given time, as with ODEs, the smaller is h (i.e., the finer is the grid) the smaller the error. But here as we are integrating with time there is also an error due to this time integration. To keep this error small we can only make small changes in the `temp(i)` elements at each step. These changes are

proportional to δt, and to D but also scale as $1/h^2$ (from the h dependence in the Laplacian). These numbers combine to form a dimensionless group $(D\delta t)/h^2$. This group must be much less than one, to keep the errors in the time integration low. If this number is too large, the time integration can fail, which can cause the program to crash. It is a good idea to write this number to the screen so you can check its value.

RP Sear 2015–2017

Assignment 2

Write a Fortran program, using an array of real numbers, to calculate the time evolution, by diffusion, of a temperature distribution along the x-axis. The diffusion constant $D = 1.1$ cm^2/s, the value for copper at room temperature. The initial BCs are

$$T(x, t = 0) = \begin{cases} 0°\text{C} & x < -1 \text{ cm} \\ 20°\text{C} & -1 \leq x \leq 1 \text{ cm} \\ 0°\text{C} & x > 1 \text{ cm} \end{cases}$$

Use a grid spacing of $h = 0.1$ cm, and cover the region -30 cm $\leq x \leq 30$ cm. Use a time step $\delta t = 10^{-4}$s. Study the evolution of temperature from $t = 0$ until a final time $t_{end} = 10$ s. Write out the final (at $t = t_{end}$) temperature profile to a file, and write the name of that file to the screen.

At the edges of the profile the array elements `temp(i)` will be very close to zero, and so changing by amounts so small that you may see the warning: "Warning: Floating underflow occurred". You can ignore this warning, it is telling you some elements are changing by amounts so small that they are zero to the precision you are working at. As the temperature is very close to zero near the edges of the distribution this is to be expected. You will not lose marks for this warning.

Checking the solution

For Assignment 2, all your program needs to do is calculate the final temperature profile, and write it to a file, and the name of the file to a screen. But is always good to check a solution, and see what it says about the physics of the problem. You can do this as follows.

Diffusion causes the initially localised temperature peak at the origin to spread out. It is useful to quantify the rate at which this happens. To do this we need a measure of the width of the temperature profile. There is more than one way to do this. A simple way is via the Full-Width-at-Half-Maximum (FWHM), i.e., the width of the peak $T(x, t)$ at a point where T is half its value at the peak.

The function is symmetric about the origin and the peak position does not move in this case. Thus the peak is at $x = 0$ and so array element `i = 0`, at all times. So the peak height is just `T(0)` and so the half-maximum value is `T(0)/2.0`. On the right-hand side of the peak ($x > 0$) the point at half-maximum can be defined as the array element with the largest value of `i` for which `T(i)` > `T(0)/2.0`. Note that as we only have T at a discrete set of points we will not have a point with a T value of exactly half the peak value, and so need to pick a point where T is as close as we can get to half the peak value. We call that value `i_pos`. We do the same on the other side, i.e., find `i_neg`, the largest value of i for which T is greater than half the peak value. The FWHM is then `real(i_pos - i_neg) * h`. Note we need to convert from the integer array numbers to a real x value.

Plotting this FWHM as a function of time will show how diffusion causes the temperature profile to broaden. It should be approximately a power law, with exponent 1/2, i.e., FWHM $\approx a + bt^{1/2}$, with a and b constants. If you vary the value of the diffusion constant D in your program, you should find that to a good approximation the fitting constant $b \propto D^{1/2}$. In other words that the width of the profile after a time t is proportional to the square root of the diffusion constant. So for example, if the diffusion constant is doubled, the distance heat diffuses over only increases by a factor of $2^{1/2} \simeq 1.4$.

It is also a good idea to verify that the solution is such that truncating the array at $|x| = 30$ cm introduces only a negligible error. You can do this by checking that the array elements near $x = 30$ cm are close to zero, and/or by shrinking the array a bit and checking that this does not affect the solution.

A final check on your program is that the temperature profile after $t \gtrsim 2$ s should be well fit by a Gaussian function.