PHAS0030: Further Practical Mathematics & Computing
Session 5: More Differential Equations

David Bowler

February 8, 2019

In this session, we will continue our examination of differential equations, introducing matrix methods for their solution, and beginning to look at partial differential equations (PDEs). We will recall how PDEs are characterised, and investigate how two of these types are solved.

Contents

1 Objectives

The objectives of this session are to:

- Understand matrix approaches to boundary value problems for ordinary differential equations
- Characterise partial differential equations into hyperbolic, elliptic and parabolic
- Demonstrate how parabolic equations can be solved, first with explicit finite differences and then with implicit methods
- Consider elliptic equations, and explore the use of matrix approaches in their solution, investigating inversion and alternatives to inversion

2 Review of Session 4

In the fourth session, we looked at ordinary differential equations (ODEs), seeing how Euler's method can be used as a simple but often unreliable approach. We discussed how second order ODEs can be broken down into coupled first order ODEs, and investigated more accurate approaches to their solution. We introduced boundary value problems and the shooting method to solve them, and considered the SciPy functions that can be used in place of hand-coded functions.

2.1 Implicit methods

In Session 4, we saw that the Euler method is inherently unstable for large step sizes. For the simple differential equation:

$$\frac{dy}{dt} = -ky \tag{1}$$

we showed that this instability[1] comes directly from the use of the forward difference approximation for dy/dt. What would happen if we used the *backward* difference instead?

$$\begin{aligned}
\frac{dy}{dt} \simeq \frac{y_n - y_{n-1}}{\Delta t} &= f(y_n, t_n) = -ky_n & (2)\\
y_n &= y_{n-1} + \Delta t(-ky_n) & (3)\\
y_n(1 + k\Delta t) &= y_{n-1} & (4)\\
y_n &= y_{n-1}(1 + k\Delta t)^{-1} & (5)\\
y_n &= y_0(1 + k\Delta t)^{-n} & (6)
\end{aligned}$$

This formula is stable, and does not suffer from the problems of the simple Euler method. However, in most cases, the form of the function $f(y,t)$ is not as simple as this, and we have the difficulty that the update for the step y_n depends on the value y_n. This type of

[1] We showed that at the n^{th} timestep, t_n, we could write $y_n = y_0(1 - k\Delta t)^n$.

problem is an *implicit* problem, and requires some form of iteration, often using a root-finder to solve each step. In the next section, we will see an alternative way of solving implicit problems.

3 *Matrix approach to boundary-value problems*

We can model the one-dimensional flow of heat in a bar with cross-sectional area $A(x)$ with the following simple equation:

$$Q = -\kappa A(x)\frac{d\theta}{dx} \tag{7}$$

In this case, Q is a constant (if we are in the steady state), θ is the temperature, κ is the thermal conductivity and we fix the temperature at the ends of the bar. This gives a boundary value problem[2].

There is an alternative approach to the shooting method, using matrices, that is a standard approach to diffusion problems (which are *partial* differential equations) that we can use for this simple ODE. We assume that Q is a constant; the shooting method would then seek the value of Q that matches the specified boundary values

Instead, we discretise the problem[3] (so that we will evaluate the temperature at a set of points along the bar, $\{x_i\}$) and note that, if Q is constant, then we must have:

$$A(x_i)\left.\frac{d\theta}{dx}\right|_i = A(x_{i+1})\left.\frac{d\theta}{dx}\right|_{i+1} \tag{8}$$

Now, substituting for the finite differences, and assuming that A is constant for simplicity, we have:

$$\begin{aligned} \frac{A}{\Delta x}(\theta_i - \theta_{i-1}) &= \frac{A}{\Delta x}(\theta_{i+1} - \theta_i) && (9)\\ \frac{A}{\Delta x}(\theta_{i+1} - 2\theta_i + \theta_{i-1}) &= 0 && (10) \end{aligned}$$

This general relationship links points that are adjacent along the bar; if you think about how we could write each of these equations for each point along the bar, you should see that it would form a matrix. We will have a series of equations like:

$$\begin{aligned} c_{1,1}\theta_1 + c_{1,2}\theta_2 + c_{1,3}\theta_3 &= 0 && (11)\\ c_{2,2}\theta_2 + c_{2,3}\theta_3 + c_{2,4}\theta_4 &= 0 && (12)\\ \vdots \quad &\quad \vdots && (13)\\ c_{N-2,N-2}\theta_{N-2} + c_{N-2,N-1}\theta_{N-1} + c_{N-2,N}\theta_N &= 0 && (14)\\ & && (15) \end{aligned}$$

For a boundary-value problem, we will *specify* θ_1 and θ_N, so we can

[2] You will solve this using the shooting method from Session 4 in Problem Sheet 2.

[3] Of course, we discretise the problem with the shooting method, but in that case we solve for each point sequentially, whereas here we solve for all points at once.

re-arrange these equations into a single matrix equation:

$$\begin{pmatrix} c_{1,2} & c_{1,3} & 0 & 0 & \dots & 0 & 0 \\ c_{2,2} & c_{2,3} & c_{2,4} & 0 & \dots & 0 & 0 \\ 0 & c_{3,3} & c_{3,4} & c_{3,5} & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & c_{N-2,N-2} & c_{N-2,N-1} \end{pmatrix} \begin{pmatrix} \theta_2 \\ \theta_3 \\ \theta_4 \\ \vdots \\ \theta_{N-1} \end{pmatrix} = \begin{pmatrix} -c_{1,1}\theta_1 \\ 0 \\ 0 \\ \vdots \\ -c_{N-2,N}\theta_N \end{pmatrix} \tag{16}$$

To see how we got this, note that we have $N-2$ equations because the end points are fixed. But in the original equations there are entries that would make N columns in a matrix. We need to eliminate two columns to make a $(N-2)\times(N-2)$ matrix. We can remove the first and last columns by imposing the boundary conditions.

We now have to solve a problem that can be written[4] as $\underline{\underline{\mathbf{M}}}\,\underline{\theta} = \mathbf{b}$, which is a classic problem in linear algebra. There are a number of Numpy and Scipy approaches: `np.linalg.solve(M,b)` will solve for and return $\underline{\theta}$; `np.linalg.inv(M)` will calculate the inverse matrix to `M` and return it (and it can then be applied to any boundary condition vectors). Confusingly, Scipy also offers a module `linalg` and solvers `linalg.inv` and `linalg.solve` though they are not identical. We will talk further about the form of this type of matrix later.

3.1 Exercises

In class

1. Construct the matrix M for the bar with uniform cross-section, following these steps:

   (a) Define a problem size, N, and create a $N \times N$ matrix of zeros

   (b) Create a stencil using `np.array` with elements $(1, -2, 1)$

   (c) Loop over the rows of the matrix M excluding the first and the last

   (d) For each row, copy the stencil into the appropriate place: $M[i, i-1 : i+2]$

   (e) Set the appropriate parts of the first and last rows to `stencil[1:3` and `stencil[0:2]` respectively

   Your matrix should look something like this (example for $N = 7$):

```
[[-2.  1.  0.  0.  0.  0.  0.]
 [ 1. -2.  1.  0.  0.  0.  0.]
 [ 0.  1. -2.  1.  0.  0.  0.]
 [ 0.  0.  1. -2.  1.  0.  0.]
 [ 0.  0.  0.  1. -2.  1.  0.]
 [ 0.  0.  0.  0.  1. -2.  1.]
 [ 0.  0.  0.  0.  0.  1. -2.]]
```

[4] A typographical note: LaTeXseems unable to render bold Greek letters here, so I am using $\underline{\theta}$ to indicate the vector of temperatures at points along the bar

2. Use `np.linalg.inv` to invert the matrix and solve for the temperature, with boundary conditions $\theta(x)$ for $\theta(x = 0) = 500K$ and $\theta(x = 1) = 300K$, using `np.dot(Minv,b)`. Plot the result (including the end-points). Note that we assume that the bar is 1m long.

Further work

1. Now consider a bar which has cross-sectional area $\pi(2-x)^2/10,000$. Write a function that returns the area for an input of x.
2. Construct a matrix for the varying cross-section, following these steps, which are an adaptation of the method we used above:
   (a) Start as you did before with a size, N, and a matrix of zeros
   (b) Define $\Delta x = 1/(N+1)$ for a bar of length 1m.
   (c) For the stencil, you will now need to use $(A(x_i), -A(x_i) - A(x_{i+1}), A(x_{i+1})$ where $x_i = i\Delta x$
   (d) Loop over the rows of the matrix M excluding the first and the last, and define `x_i = i*delta_x`
   (e) For each row, calculate and copy the stencil into the appropriate place $M[i, i-1 : i+2]$
   (f) Set the appropriate parts of the first and last rows to `stencil[1:3` and `stencil[0:2]`, and with `i=0` and `i=N-1`. Print your matrix to check that it is symmetrical (you may find `np.set_printoptions(precision=5)` helpful)
   (g) Now solve for the temperature, setting the boundary conditions as $-500A(0)$ and $-300A(1.0 - \Delta x)$. Plot your result, with the boundary conditions, and ensure that there are no discontinuities.

4 *Partial Differential Equations*

Partial differential equations (PDEs) involve solving for a function of more than one independent variable, which are most commonly position and time in physics. While it is perfectly possible to write a first-order PDE, the most common forms are second order. If we consider a function of x and y for simplicity[5] then a general linear, second-order PDE can be written as:

$$A(x,y,\phi)\frac{\partial^2\phi}{\partial x^2} + B(x,y,\phi)\frac{\partial^2\phi}{\partial x\partial y} + C(x,y,\phi)\frac{\partial^2\phi}{\partial y^2} = f\left(x,y,\phi,\frac{\partial\phi}{\partial x},\frac{\partial\phi}{\partial y}\right) \tag{17}$$

If the right-hand side of the equation is zero, or is linear in $\phi(x,y)$ and its first-order derivatives, then the equation is called *homogeneous*: any constant multiple of a solution $\phi(x,y)$ is also a solution. There are many examples of linear, second-order PDEs in physics,

[5] We could just as easily have chosen x and t, and these arguments extend to three dimensions as well.

including:

$$\begin{aligned}
\frac{\partial^2\phi}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2\phi}{\partial t^2} &= 0 & (18)\\
\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2} &= 0 & (19)\\
\nabla^2\phi - \frac{1}{\kappa}\frac{\partial\phi}{\partial t} &= 0 & (20)\\
-\frac{\hbar^2}{2m}\nabla^2\phi + V\phi &= i\hbar\frac{\partial\phi}{\partial t} & (21)
\end{aligned}$$

which are the one-dimensional wave equation (easily extended to three dimensions), Laplace's equation, the diffusion or heat equation and the time-dependent Schrödinger equation respectively. These are all homogeneous; by adding a charge density to Laplace's equation (to get Poisson's equation) or a heat source $Q(x, t)$ to the heat equation we would obtain *inhomogeneous* equations, which are also extremely important.

These equations all behave differently, and require different boundary conditions and approaches to solving. They can be characterised by considering the discriminant function:

$$\Delta = B^2 - 4AC \qquad (22)$$

where there are three possibilities, named after conic sections (which are defined by the same equation):

1. $\Delta > 0$ Hyperbolic: propagating oscillations (waves)
2. $\Delta = 0$ Parabolic: transport processes, such as heat flow and diffusion
3. $\Delta < 0$ Elliptic: stationary systems, such as electrostatic fields and steady-state temperature distributions

We will consider the solution of each of these types separately (with hyperbolic considered in Session 6). Of course, as with any broad classification scheme, care must be taken: notice that the heat equation is both parabolic (when the temperature varies with time) and elliptic (in a steady-state situation).

4.1 Boundary Conditions

For ordinary differential equations, the specification of boundary conditions was rather simple, and required only a number of pieces of information equivalent to the order of the equation. It is considerably more complex for second-order PDEs, with the information required and the domain over which it is specified varying with the nature of the problem.

Each equation type requires boundary conditions named after famous mathematicians who worked in the area. The boundary conditions are specified in terms of the value of the function, ϕ, or its derivative along the normal to the boundary, $\nabla_n\phi$.

1. Hyperbolic: Cauchy conditions (ϕ and $\nabla_n\phi$ on an open boundary)
2. Parabolic: Dirichlet (ϕ) or von Neumann ($\nabla_n\phi$) along an open boundary
3. Elliptic: Dirichlet or von Neumann on a closed boundary

Note that both hyperbolic and parabolic are time-dependent, while elliptic is not. A helpful example of Dirichlet and von Neumann boundary conditions comes from electrostatics: we would specify either the potential ϕ (Dirichlet) or $\mathbf{E}\cdot\mathbf{n}$ (von Neumann) along a boundary (where the normal vector is $\mathbf{n}$).

5 *Parabolic equations*

We will return to our example of one-dimensional heat flow in a bar, but now consider time dependence: the whole bar will start at 300K, but we will raise the end at $x = 0m$ to 500K at $t = 0s$. The equation that we need to solve is:

$$\frac{\partial\theta}{\partial t} = \frac{\kappa}{C\rho}\frac{\partial^2\theta}{\partial x^2} \tag{23}$$

where C is the specific heat capacity of the material, ρ is the (mass) density and κ is the thermal conductivity. For a typical metal, these would have values of $\kappa = 200\text{Wm}^{-1}\text{K}^{-1}$, $C = 1,000\text{Jkg}^{-1}\text{K}^{-1}$ and $\rho = 2,500\text{kgm}^3$.

How do we go about solving this? The simplest approach would be to take finite differences; for the second derivative in x we can use a centred formula. The time component is more complex; we will start with a simple forward difference, and investigate its behaviour[6]. We will then introduce a better approach. We will discretise space and time, so that a point $(x_i, t_n) = (i\Delta x, n\Delta t)$. Keep track of these indices: we will write the temperature as a function of space and time as: $\theta(x_i, t_n) = \theta_{i,n}$.

The spatial derivative is given by:

$$\left(\frac{\partial^2\theta}{\partial x^2}\right)_{i,n} \simeq \frac{\theta_{i+1,n} - 2\theta_{i,n} + \theta_{i-1,n}}{\Delta x^2} \tag{24}$$

while we will, for now, write the time derivative as:

$$\left(\frac{\partial\theta}{\partial t}\right)_{i,n} \simeq \frac{\theta_{i,n+1} - \theta_{i,n}}{\Delta t} \tag{25}$$

How do we proceed? We should have a starting temperature distribution (say $\theta_{i,0}$ for all values of i), and we want to know how it evolves over time. So we want to find $\theta_{i,n}$ for all i in our bar, for various times indexed by n. We can substitute Eq. (24) and Eq. (25) into Eq. (23) and re-arrange:

[6] We may well suspect that it will not be stable beyond a certain time step, on the basis of our experience with Euler's method.

$$\begin{aligned}
\frac{\theta_{i,n+1}-\theta_{i,n}}{\Delta t} &= \frac{\kappa}{C\rho}\frac{\theta_{i+1,n}-2\theta_{i,n}+\theta_{i-1,n}}{\Delta x^2} && (26)\\
\theta_{i,n+1} &= \theta_{i,n}+\frac{\kappa\Delta t}{C\rho\Delta x^2}\left(\theta_{i+1,n}-2\theta_{i,n}+\theta_{i-1,n}\right) && (27)\\
\theta_{i,n+1} &= \zeta\left(\theta_{i+1,n}+\theta_{i-1,n}\right)+(1-2\zeta)\theta_{i,n} && (28)
\end{aligned}$$

where $\zeta = \kappa\Delta t/(C\rho\Delta x^2)$. Notice that the value of the temperature is given by its value and its neighbours' values at the previous time step: a simple, explicit scheme. You will construct a simple implementation of this in the exercises.

You should find during the implementation that there is a critical value of the parameter ζ; notice that this combines both physical parameters of the system, and the timestep and grid spacing; there is a key ratio for stability when the timestep becomes too large (just as we saw for the Euler method). You may notice that the update for each timestep is rather similar to what we saw in Section 4. Indeed, we might imagine that we could write the overall process in a similar matrix equation to Eq. (16)—and indeed we can. It turns out that we can show that the stability is related to the eigenvalues of the update matrix, which need to be smaller than one[7]

5.1 *Implicit methods*

We know from our work on finite differences that centred formulae are more accurate and stable; can we use this here? We can indeed. If we think of the formula we've been using not as a forward difference, but as a *centred* difference at the timestep $n+1/2$ then, so long as we can update the right-hand side of the equation for the spatial derivative, we should be more stable.

We will approximate the spatial derivative at time $n+1/2$ by averaging:

$$\left(\frac{\partial^2\theta}{\partial x^2}\right)_{i,n+1/2} = \frac{1}{2}\left[\left(\frac{\partial^2\theta}{\partial x^2}\right)_{i,n}+\left(\frac{\partial^2\theta}{\partial x^2}\right)_{i,n+1}\right] \qquad (29)$$

If we substitute these into Eq. (23), we find:

$$\begin{aligned}
\frac{\theta_{i,n+1}-\theta_{i,n}}{\Delta t} = \frac{\kappa}{2C\rho}\left(\frac{\theta_{i-1,n}-2\theta_{i,n}+\theta_{i+1,n}}{\Delta x^2}\right. &+ \left.\frac{\theta_{i-1,n+1}-2\theta_{i,n+1}+\theta_{i+1,n+1}}{\Delta x^2}\right)\\
-\zeta\theta_{i-1,n+1}+2(1+\zeta)\theta_{i,n+1}-\zeta\theta_{i+1,n+1} &= \zeta\theta_{i-1,n}+2(1-\zeta)\theta_{i,n}+\zeta\theta_{i+1,n}
\end{aligned} \qquad (30)$$

This is an *implicit* method: the values of θ at a point at each timestep depend on the values of its neighbours in the previous timestep *and* at the present timestep. We cannot solve this by a direct iteration, as before, but we can use a matrix approach, exactly paralleling what we did in Section 3. As we have to incorporate values of θ from neighbours at two timesteps, and include boundary conditions, the overall result is a little more complex than before:

$$\underline{\underline{\mathbf{M}}}\,\underline{\theta}_{n+1} = \underline{\underline{\mathbf{N}}}\,\underline{\theta}_n + \underline{\mathbf{b}} \qquad (31)$$

[7] We will not pursue this further, but it's important to be aware of this kind of analysis.

where the matrices $\underline{\underline{\mathbf{M}}}$ and $\underline{\underline{\mathbf{N}}}$ are defined from Eq. (30) as:

$$\underline{\underline{\mathbf{M}}} = \begin{pmatrix} 2(1+\zeta) & -\zeta & 0 & 0 & \dots \\ -\zeta & 2(1+\zeta) & -\zeta & 0 & \dots \\ 0 & -\zeta & 2(1+\zeta) & -\zeta & \dots \\ \vdots & \vdots & \vdots & \ddots & \vdots \end{pmatrix} \quad (32)$$

$$\underline{\underline{\mathbf{N}}} = \begin{pmatrix} 2(1-\zeta) & \zeta & 0 & 0 & \dots \\ \zeta & 2(1-\zeta) & \zeta & 0 & \dots \\ 0 & \zeta & 2(1-\zeta) & \zeta & \dots \\ \vdots & \vdots & \vdots & \ddots & \vdots \end{pmatrix} \quad (33)$$

The boundary conditions for our bar would be a vector of zeros except at the first and last entries, which would be $2\zeta\theta_1$ and $2\zeta\theta_N$. If we solve for $\underline{\underline{\mathbf{M}}}^{-1}$ once at the start and also calculate $\underline{\underline{\mathbf{M}}}^{-1}\,\underline{\mathbf{b}}$ and $\underline{\underline{\mathbf{M}}}^{-1}\underline{\underline{\mathbf{N}}}$ then the propagation becomes rather simple. This implict method is known as the Crank-Nicolson method, which can be shown to be stable for all values of ζ (though of course large values of ζ will not give accurate results).

5.2 *Exercises*

In class

1. Write a function to perform the update in Eq. (28). You should pass as parameters an array of temperatures at timestep n only (i.e. a 1D array) and the constant `zeta`. Return an array of temperatures at timestep $n+1$. When you iterate along the bar, remember to *exclude* the end-points. [You may find `np.size` useful to determine the iteration.]

2. Create a 2D array to store the temperature; don't make the spatial domain too large (I chose 7 points) and use 10 time points. Set the boundary conditions ($\theta = 300K$ at $t = 0$ for all points, except $\theta_{0,0} = 500K$). Define a parameter `zeta` (start with 0.1) and loop over time, calling the update routine to evolve the differential equation forward in time. Ensure that you set the boundary conditions at each step if necessary. [Remember that, if you have an array `temperature[I,N]` with `I` time points and `N` spatial points then `temperature[i]` will give a 1D array with all points along the bar at timestep `i`. You can also store a 1D array in your 2D array using something like `temperature[i] = array`.]

3. Plot the resulting evolution of the temperature distribution (you could loop over time steps and put them all on the same graph using `plt.plot` or you create an array of plots using the `figure` approach).

4. Now repeat the calculation for `zeta=0.7`. What happens? If you have time, identify a critical value of `zeta`.

Further work

1. Write two functions to create the matrices $\underline{\underline{\mathbf{M}}}$ and $\underline{\underline{\mathbf{N}}}$. You should pass as parameters the dimensions of the problem (remember that for N points your matrices should have size $(N-2 \times N-2)$) and ζ and return the matrix. You can use the same stencil approach as in Section 3.1.
2. Now solve the same problem as we did in Question 2 of the in-class work, and experiment with the value of ζ. You should ask yourself two questions: 1. Is the approach stable for all values of ζ? 2. How does the long-time solution vary with ζ? You may need to change the number of steps you use with smaller values of ζ.

6 *Elliptic Equations*

We now turn to elliptic equations, which have two spatial dimensions, instead of one spatial and one temporal dimension as we have seen just now. The simplest equation is Laplace's equation in two dimensions, which is written:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0 \tag{34}$$

We can apply a centred finite-difference scheme, where we now have points x_i and y_j and will notate $\phi(x_i, y_j)$ as $\phi_{i,j}$. We can then write:

$$\frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{\Delta x^2} + \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{\Delta y^2} = 0 \tag{35}$$

If we set $\Delta x = \Delta y$, then we can make a considerable simplification:

$$\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j} = 0 \tag{36}$$

How do we go about solving this problem? If we map the points on the $x - y$ grid into a vector (so that the index in the vector, $n = (j-1)N + i$ for an $N \times N$ grid) then we can write the differential equation as a matrix acting on a vector; to calculate the matrix elements we will use Eq. (36) *and* the boundary conditions which will be imposed. The final form is:

$$\underline{\underline{\mathbf{M}}}\,\underline{\phi} = \underline{\mathbf{b}} \tag{37}$$

The mapping from physical problem to computer code is a little complicated: we go from a 2D grid of points to a 1D vector for the potential. To establish the matrix and the boundary conditions we use Eq. (36) for *each* row, choosing i and j corresponding to the entry in the vector.

To make this concrete, we will consider a small, simple problem: a (3×3) mesh with the potential fixed at $3V$ on the x boundaries and $4V$ on the y boundaries. Note that you will need to use both N_x (or N_y) and $N = N_x \times N_y$. In this case, $N_x = N_y = 3$ and $N = 9$. There are 9 sites corresponding to 3 x coordinates and 3 y coordinates. The matrix is (9×9). The rest of the derivation is in the exercises.

6.1 Alternatives to inversion

Matrix inversion becomes expensive as matrices get large (it scales with the cube of the size: $\mathcal{O}(N^3)$ scaling). As the matrix that we are using scales as the square of the number of grid points in x or y, this can rapidly become very expensive. As our matrices are rather structured (not quite tridiagonal, but certainly dominated by the diagonal) then other methods are worth exploring.

We saw some of these methods in Session 2 when considering the optimisation of functions: we are again trying to solve the standard linear algebra problem $\underline{\underline{\mathbf{A}}}\mathbf{x} = \mathbf{b}$. The Jacobi method writes $\underline{\underline{\mathbf{A}}} = \underline{\underline{\mathbf{D}}} + \underline{\underline{\mathbf{R}}}$ where $\underline{\underline{\mathbf{D}}}$ is just the diagonal of the matrix, and defines the iteration:

$$\mathbf{x}^{(k+1} = \underline{\underline{\mathbf{D}}}^{-1}\left(\mathbf{b} - \underline{\underline{\mathbf{R}}}\mathbf{x}^{(k)}\right) \tag{38}$$

and the solution is iterated until it converges. This is often not a fast converging method, and the Gauss-Seidel method is used. Here we write the original matrix in terms of a lower-diagonal matrix $\underline{\underline{\mathbf{L}}}$ and an upper-diagonal matrix $\underline{\underline{\mathbf{U}}}$ (though note that $\underline{\underline{\mathbf{U}}}$ does not include the diagonal entries) and iterate:

$$\underline{\underline{\mathbf{L}}}\mathbf{x}^{(k+1)} = \mathbf{b} - \underline{\underline{\mathbf{U}}}\mathbf{x}^{(k)} \tag{39}$$

This looks, at first sight, as if it will require inversion of the matrix $\underline{\underline{\mathbf{L}}}$, but it can be shown that the following element-by-element update solves without inversion:

$$x_i^{(k+1)} = \frac{1}{A_{ii}}\left(b_i - \sum_{j=1}^{i-1} A_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{N} x_j^{(k)}\right) \tag{40}$$

This is easy to implement, and is iterated until convergence is reached.

You should be aware that this approach may converge slowly, and a further refinement is sometime used, called the successive over-relaxation (SOR) method. In this case, at each step in the iteration, we mix a proportion of the previous solution with a proportion of the Gauss-Seidel solution:

$$\mathbf{x}^{(k+1)} = (1-\omega)\mathbf{x}^{(k)} + \omega\mathbf{x}^{(k+1),GS} \tag{41}$$

where $0 < \omega < 2$. To do this, we have to decompose the matrix as $\underline{\underline{\mathbf{A}}} = \underline{\underline{\mathbf{D}}} + \underline{\underline{\mathbf{L}}} + \underline{\underline{\mathbf{U}}}$. The SOR method can be effective, but requires some study of the system to find the correct value of ω. If ω is chosen poorly, then the convergence can be very slow.

6.2 *Exercises*

In class

1. Write functions to convert an (i, j) pair to an index and back again; as parameters, you should use (i, j) and N_x or the index and N_x, returning either the index or the (i, j) pair. [You may find the `math.floor` function useful - import it appropriately.]

2. Now write a function to calculate the Laplacian matrix and the boundary condition vector, using the following steps:

   (a) Pass as parameters *either* N_x *or* N and the boundary conditions for both x and y (in this case $3V$ for x and $4V$ for y). You will need to calculate whichever of the two sizes you did not pass.

   (b) Create an output matrix which is $(N \times N)$ and an output boundary condition vector which is a 1D vector with N entries

   (c) Iterate over the rows in the matrix, N, and calculate the corresponding (i, j) pair

   (d) Set the on-site element (use the same column as the row)

   (e) Now iterate over the four neighbours (you could use something like `for delta in (-1,1):` if you liked and apply to i and j separately, or another method)

   (f) Check on whether the neighbour is within the grid: $0 \leq i < N_x$ or $0 \leq j < N_y$. [Remember that python allows constructs like `if 0 <= a <= b`]

   (g) If it is, set the appropriate element using Eq. (36) (calculate the column index for the neighbour from i and j)

   (h) If it is not, accumulate the boundary condition: $-3V$ or $-4V$ in this case.

3. Now solve for the potential, using `np.linalg.inv` and `np.dot` to solve Eq. (37). You will need to reshape the resulting vector into a grid (use `np.reshape(potential,(Nx,Ny))`) and plot using `plt.imshow`. You may find the optional parameter `interpolation='bicubic'` makes the very blocky result clearer.

Further work

1. Create a Jacobi or Gauss-Seidel solver for the electrostatic grid problem above, and check that the results match the exact inversion. You will need to define a tolerance and stop when the change in the solution (defined in some way - maybe the RMS change between iterations) is smaller than this. You might like to increase the size of the grid, and see how whether the numerical cost becomes noticeable.

7 Elliptic equations: an alternative

The creation of the matrix involved in solving the elliptic equation can seem rather complex, and certainly involves non-trivial indexing. The methods described in Sec. 6.1 as an alternative to matrix inversion (generally known as *relaxation* methods) can be rewritten to give an iterative procedure for the update of each point in the domain where we are solving the equation, *without* the need for matrices or converting from a 2D (or 3D) grid to a 1D vector.

At the simplest level, we need an update for the potential ϕ at each step, k; if we re-write Eq. (36), we find:

$$\begin{aligned} \phi_{i,j} &= \frac{1}{4}\left(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}\right) && (42) \\ \Rightarrow \phi^{(k+1)}_{i,j} &= \frac{1}{4}\left(\phi^{(k)}_{i+1,j} + \phi^{(k)}_{i-1,j} + \phi^{(k)}_{i,j+1} + \phi^{(k)}_{i,j-1}\right) && (43) \end{aligned}$$

where in the second equation we have indicated how the update works. In this case, after creating an initial set of values for ϕ, we simply set the boundary values of ϕ at each step and then update. This form is equivalent to the Jacobi method; a sample implementation of the update is shown in Fig. 1.

```python
def update_phi(phi, N):
    """Update NxN grid of phi using Jacobi method"""
    # Copy rather than equate to avoid update issues
    phiout = np.copy(phi)
    # Avoid boundaries in update
    for i in range(1,N-1):
        for j in range(1,N-1):
            phiout[i,j] = 0.25*(phi[i-1,j] + phi[i+1,j]
                                + phi[i,j-1] + phi[i,j+1])
    return phiout
```

Figure 1: Update algorithm for the Jacobi relaxation method

We can easily show that the Gauss-Seidel method (described above in Sec. 6.1) can be implemented in almost the same as the Jacobi method above, but allowing the updated values of ϕ to be used instead of the current values[8]. This is a more efficient, and importantly a more stable, method to use, particularly when introducing a source term. In that case, the equation we solve would be something like $-\nabla^2\phi = \rho/\epsilon_0$, and the update equation would need an extra term $-h^2\rho_{i,j}/4\epsilon_0$ where the grid spacing is h.

The successive over-relaxation (SOR) approach can also be used here, and has a significant effect on the efficiency. Essentially it mixes the present and update values of $\phi_{i,j}$ via a parameter ω. We use Eq. (41) but rewrite it to find:

$$\phi^{(k+1)}_{i,j} = \frac{1+\omega}{4}\left(\phi^{(k)}_{i+1,j} + \phi^{(k)}_{i-1,j} + \phi^{(k)}_{i,j+1} + \phi^{(k)}_{i,j-1}\right) - \omega\phi^{(k)}_{i,j} \qquad (44)$$

There is no general way to find the optimum value of ω, and some

[8] In the example code, we would not need the variable `phiout`, but would instead just update `phi`.

experimentation on small, simple problems is often needed. You should note that $0 < \omega < 1$ guarantees stability.

7.1 Exercises

1. Adapt the basic Jacobi solver in Fig. 1 to implement first the Gauss-Seidel solver, and then the SOR solver with Gauss-Seidel. For the SOR method you should pass ω as a parameter.

2. Now set up an $(N \times N)$ grid for ϕ (where N is a variable that you can adjust) which will *include* the boundaries. Set the initial values of ϕ to include the boundary conditions (3V for x and 4V for y) with $\phi = 0$ elsewhere. Using a `while` loop, iterate using the Gauss-Seidel method to find a solution for ϕ (you should calculate the maximum change in any element of ϕ from step to step using `np.max` and `np.abs`). Keep a record of the number of iterations, and output it at the end.

3. Plot the resulting potential using `plt.imshow` or `plt.contourf`. With `imshow` you might experiment with interpolation, and compare to a solution using larger numbers of points. With `contourf`, remember that you can pass an array of contour values to use.

4. Now use the SOR method for values of ω between 0.1 and 0.9, and find the most efficient value

8 Progress Review

Once you have finished *all the material* associated with this session (both in-class and extra material), you should be able to:

- Use matrix approaches to solve boundary value problems for ODEs (where appropriate)

- Understand how to classify PDEs

- Solve simple examples of both elliptic and parabolic PDEs given appropriate boundary conditions