

COMPUTATIONAL PHYSICS
PHYS3071
Partial Differential Equations

Paul Cochrane and Professor P. D. Drummond

April 18, 2002

Typeset in L^AT_EX 2_ε by
Paul T. Cochrane,
April 18, 2002.

Contents

1	Partial Differential Equation Lab	1
2	Partial Differential Equations	3
2.1	First order partial derivatives	3
2.1.1	Advection equation	3
2.1.2	Burger's equation	4
2.1.3	Maxwell's equations	4
2.2	Second order partial derivatives	5
2.2.1	Diffusion equation	5
2.2.2	Schrödinger equation	6
2.2.3	Nonlinear Schrödinger equation	7
2.2.4	Burger's equation	7
2.3	Initial and boundary conditions	7
3	Numerical algorithms	11
3.1	Numerical differentiation	11
3.1.1	First derivative	11
3.1.2	Second derivative	13
3.2	FTCS scheme	14
3.3	Matrix form of FTCS scheme	21
3.4	Implicit FTCS scheme	26
3.5	Crank-Nicolson scheme	27
3.6	Spectral method (FFT scheme)	27
3.7	Pseudo-spectral method (Split-step FFT)	31
4	Numerical Errors	35
4.1	Time discretisation errors	35
4.2	Space discretisation errors	35
4.3	Windowing errors	36
4.4	Instability errors	36
4.5	Roundoff (truncation) errors	36

4.6	Range errors	37
5	Projects	39
5.1	PROJECT I	39
5.2	PROJECT II	40
6	Sample m-file	43
6.1	Sample plot	45
A	First Order Numerical Algorithms	47
A.1	Numerical implementation	48
A.2	FTCS methods	48
A.3	BTCS Methods	50
A.4	Numerical errors	51
A.5	Examples:	53
	A.5.1 Advection equation	53
	A.5.2 Diffusion equation	53
B	Higher Order Numerical Methods	55
B.1	CTCS Methods	55
B.2	Crank-Nicolson method	55
B.3	Split-step method	56
B.4	Split-step FFT method	56

Chapter 1

Partial Differential Equation Lab

These notes are an updated and expanded version of the PHYS3070 notes originally written by Professor P. D. Drummond.

This manual covers the partial differential equation lab section of the University of Queensland PHYS3071 course in Computational Physics. This section of the course is assessed by submission of a report analysing and discussing *two* projects. This report must be submitted along with a laboratory workbook. Laboratory assessment **requires** you to keep a laboratory workbook (either on paper or electronically) to keep track of your progress in the lab, as well as the project report.

Partial differential equations (PDEs) occur in many different areas of physics, ranging from quantum mechanics to general relativity, and including fields like hydrodynamics and electrodynamics. These equations have enormous numbers of applications in the real world, from environmental studies to laser physics. They cover practical areas like weather prediction and geophysical studies, as well as the most fundamental studies of cosmology and general relativity. Some current examples of importance to both technology and to novel physics, include pulse propagation in communications fibers, and the theory of Bose-Einstein Condensation (BEC), or atom lasers.

Many of these equations can be cast as quasi-linear parabolic equations, in which there is a first order derivative in one dimension, together with higher-order derivatives in the others. These types of equation, which are called quasi-linear if the derivative terms are all linear in the equation, are extremely common. It may be thought at first that wave-equations cannot be treated this way, since most wave equations have second-order derivatives. However, any wave equations derived from a Lagrangian have equivalent Hamiltonian forms, which are the first-order Hamilton's equations.

The techniques for treating PDEs numerically are extremely diverse. However, the basic criteria of the order of the algorithm, the stability properties, and the types of errors that occur, are common to all types of algorithm. For this reason, we focus

in this lab on two types of algorithm—an explicit type which is simple to program (but rather unstable), and a more stable semi-implicit type, which is slightly more complex, but of greater accuracy.

A large number of useful texts exist. Of particular interest, are:

- S. E. Koonin, D. C. Meredith, *Computational Physics* (Addison-Wesley, Reading, Massachusetts, 1990). An excellent general text with physics examples..
- A. L. Garcia, *Numerical Methods for Physics* (Prentice-Hall, Englewood Cliffs, New Jersey, 1994). Uses MATLAB for the programming examples.
- P. L. DeVries, *A First Course in Computational Physics* (Wiley, New York, 1994). Has an elementary account of split-step FFT methods.
- W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes in C* (Cambridge University Press, 1992). Also see the online version: <http://lib-www.lanl.gov/numerical/>. Has a good survey of many useful numerical methods.

In the rest of this manual, we cover types of PDEs, elementary techniques for solving PDEs numerically, the projects required for the laboratory, and a sample source code.

Chapter 2

Partial Differential Equations

For simplicity, we will restrict ourselves in the numerical examples to equations of the form:

$$\frac{\partial}{\partial t} \mathbf{A} = \mathcal{D} \cdot \mathbf{A} + \mathbf{U}[\mathbf{A}], \quad (2.1)$$

Where \mathbf{A} is the quantity we are analysing, and is a function of time and (often, but not always) position \mathbf{x} , so we may think of this quantity as $\mathbf{A}(\mathbf{x}, t)$, but it is much simpler to just write \mathbf{A} . The variable \mathbf{x} is a vector of spatial coordinates¹ The symbol \mathcal{D} indicates a general linear partial differential operator in the spatial coordinates \mathbf{x} . \mathbf{U} is any functional of \mathbf{A} , usually indicating a nonlinear coupling or potential term. As we will see, there is a very large class of physical problems that may still be treated. At this stage, we consider \mathbf{A} to be a vector of form (A_1, \dots, A_I) , in order to treat the most general cases of interest. In general, \mathbf{A} could be either a real or complex vector; the methods are very similar in either case.

2.1 First order partial derivatives

2.1.1 Advection equation

As an example of this general kind of PDE, consider a simple wave equation for a one-dimensional scalar field ($A(x, t)$), of the form:

$$\frac{\partial}{\partial t} A(x, t) = \mathcal{D} \cdot A(x, t) = -v \frac{\partial}{\partial x} A(x, t) \quad (2.2)$$

Note that for this equation the operator $\mathcal{D} = -v \frac{\partial}{\partial x}$. This is called the advection equation, and has the elementary solution:

$$A(x, t) = f(x - vt) \quad (2.3)$$

¹Combining, for instance, the usual variables x , y , and z in three dimensions.

This corresponds to an arbitrary initial wave-form $f(x)$, traveling at fixed velocity v . This is the simplest possible equation that has wave-like solutions traveling at a fixed velocity.

2.1.2 Burger's equation

A simple nonlinear extension of this, is the inviscid Burger's equation, which is a nonlinear wave-equation used in nonlinear acoustics, shock-wave theory, and modern studies of traffic flow. It has an equation of form:²

$$\frac{\partial}{\partial t}A(x, t) = -v\frac{\partial}{\partial x}[A - A^2/A_0] \quad (2.4)$$

These equations are sometimes used to describe traffic flow, where $A(x, t)$ would represent the density of cars on a road at a certain time and place, traveling at a velocity v that depends on the density of cars, up to a maximum density, A_0 . This equation models the fact that cars travel more slowly when the traffic density increases. Clearly, this is a rather oversimplified picture of the real situation, but it has useful real-world applications in modeling the behaviour of traffic at intersections, or the development of traffic jams after an accident has happened! Note that this is not a quasi-linear equation, and requires slightly modified techniques from the ones used for the other equations studied here. To model cars stopped at a red traffic-light, we can suppose that the initial distribution at $t = 0$ is $A(x, t) = A_0$ for $x < 0$, and $A(x, t) = 0$ for $x > 0$. The solution at times $t > 0$ is then:

$$A(x, t) = \begin{cases} A_0 & \text{for } x < -vt \\ \frac{A_0}{2} \left(1 - \frac{x}{vt}\right) & \text{for } -vt < x < vt \\ 0 & \text{for } x > vt. \end{cases} \quad (2.5)$$

2.1.3 Maxwell's equations

Another example of this general form is Maxwell's equations, which can be summarised as:

$$\begin{aligned} \frac{\partial}{\partial t}\mathbf{d} &= -\mathbf{j} + \nabla \times \mathbf{h}, \\ \frac{\partial}{\partial t}\mathbf{b} &= -\nabla \times \mathbf{e}, \end{aligned} \quad (2.6)$$

²For an good discussion of this equation and its numerical simulation see *Numerical Methods for Physics* by A. L. Garcia.

together with subsidiary conditions of:

$$\begin{aligned}\nabla \cdot \mathbf{d} &= \rho, \\ \nabla \cdot \mathbf{b} &= 0, \\ \mathbf{e} &= \epsilon^{-1} \mathbf{d}, \\ \mathbf{h} &= \mu^{-1} \mathbf{b}.\end{aligned}\tag{2.7}$$

Here the subsidiary conditions can be applied to the initial values of \mathbf{d} and \mathbf{b} , to obtain the overall equations. The physical application in this case is to electromagnetic fields in free space, as in a radio transmitter. One particularly simple type of solution, in the case of free space propagation, is just the plane-wave solution with velocity $c = 1/\sqrt{\epsilon\mu}$, of form:

$$\begin{aligned}\mathbf{d}(t, x) &= \mathbf{d}^{(+)}(x - ct) + \mathbf{d}^{(-)}(x + ct), \\ \mathbf{b}(t, x) &= \mathbf{b}^{(+)}(x - ct) + \mathbf{b}^{(-)}(x + ct).\end{aligned}\tag{2.8}$$

With some modifications, the same equations can be used to describe integrated optics devices used in communications and photonics applications. Note that this example shows how a wave equation can be regarded as being first order, but with a larger number of fields!

2.2 Second order partial derivatives

2.2.1 Diffusion equation

An example in which the spatial derivatives are of second order is the one-dimensional scalar diffusion equation, which describes heat flow and random walks:

$$\frac{\partial}{\partial t} A(x, t) = \kappa \frac{\partial^2}{\partial x^2} A(x, t).\tag{2.9}$$

In this case, there is no potential term of form $\mathbf{U}[\mathbf{A}]$, and the partial differential operator is the one-dimensional scalar diffusion operator:

$$\mathcal{D} = \kappa \frac{\partial^2}{\partial x^2}.\tag{2.10}$$

Initial value problems of this type are not the only problems possible with partial differential equations. We can also have boundary value problems, as in electrostatics; but we restrict ourselves to initial value problems in this lab. This equation can be solved completely analytically using Fourier transform methods, which we discuss

later. A possible solution on the infinite line with vanishing boundary conditions at infinity is the Gaussian solution of form

$$A(x, t) = \frac{1}{\sigma(t)\sqrt{2\pi}} \exp\left(-\frac{x^2}{2\sigma^2(t)}\right), \quad (2.11)$$

where $\sigma(t) = \sqrt{2\kappa t + \sigma_0^2}$, and σ_0 is the initial Gaussian standard deviation.

2.2.2 Schrödinger equation

The Schrödinger equation is one of the most fundamental equations of quantum mechanics, and describes the complex probability amplitude $\Psi(t, x)$ of a particle-wave, as a function of space and time. In one space dimension, it has the form:

$$i\hbar \frac{\partial}{\partial t} \psi(x, t) = -\frac{\hbar^2}{2m} \frac{\partial^2}{\partial x^2} \psi(x, t) + V(x)\psi(x, t). \quad (2.12)$$

In this case, there is a derivative term of the form:

$$\mathcal{D} = \frac{i\hbar}{2m} \frac{\partial^2}{\partial x^2} \psi(x, t), \quad (2.13)$$

together with a potential term of form $\mathbf{U}[\psi](\mathbf{x}, t) = V(x)\psi(x, t)/(i\hbar)$, which is linear in the field $\psi(x, t)$ itself. This equation has a rich family of possible solutions, including the well-known plane-wave solutions for a particle of known momentum $p = \hbar k = mv$, and energy $E = \hbar\omega = V_0 + mv^2/2$, moving in a uniform potential of $V(x) = V_0$:

$$\psi(x, t) = \psi_0 e^{i(kx - \omega t)}. \quad (2.14)$$

More generally, write:

$$\frac{\partial}{\partial t} \psi(x, t) = i\kappa \frac{\partial^2}{\partial x^2} \psi(x, t) + U(x)\psi(x, t), \quad (2.15)$$

then the solution for a Gaussian wave-packet (with $V(x) = 0$) is obtained just by replacing κ with $i\kappa$ in the solution to the diffusion equation, with $\kappa = \hbar/2m$, i.e.,

$$\psi(x, t) = \frac{1}{\sigma(t)\sqrt{2\pi}} \exp\left(ikx - \frac{x^2}{2\sigma^2(t)}\right) \quad (2.16)$$

where $\sigma(t) = \sqrt{2\kappa it + \sigma_0^2}$, and σ_0 is the initial Gaussian standard deviation.

2.2.3 Nonlinear Schrödinger equation

In this case, there is a derivative term as above, together with a potential term of form $\mathbf{U}[\Psi(t, \mathbf{x})] = i\gamma\psi(x, t)|\psi(x, t)|^2$, which is nonlinear in the field $\psi(x, t)$. This is an example of a nonlinear partial differential equation. Such equations are very important, because most real-life PDEs are actually nonlinear to some extent, and the commonly used linear equations are only an approximation to this true situation.

For example, the nonlinear Schrödinger equation describes solitons in optical fiber pulse propagation, which is linear to a first approximation at low intensities, but becomes measurably nonlinear at higher intensities:

$$\frac{\partial}{\partial t}\psi(x, t) = i\kappa\frac{\partial^2}{\partial x^2}\psi(x, t) + i\gamma\psi(x, t)|\psi(x, t)|^2. \quad (2.17)$$

Note that, if you compare this case with the above example, the potential term has the opposite sign; i.e. this equation corresponds to an attractive potential which increases with the modulus of the field. The physical result is that a stable, nonlinear wave can form, which creates its own stabilizing potential that moves with the wave itself. This is called a soliton.

2.2.4 Burger's equation

The full Burger's equation, including viscosity, has an equation of form:

$$\frac{\partial}{\partial t}A(x, t) = -v\frac{\partial}{\partial x}[A - A^2/A_0] + \kappa\frac{\partial^2 A}{\partial x^2}. \quad (2.18)$$

These equations are sometimes used to describe hydrodynamics or traffic flow. This equation models the fact that cars travel more slowly when the traffic density increases, and it includes a viscosity factor which models the finite time-scale of a reaction to changes in conditions—or, in the fluid case, the effects of fluid friction. The extra diffusion term not only makes the equation more realistic, it also makes the equation easier to treat numerically, by removing some of the numerical instabilities present in the inviscid Burgers equation.

2.3 Initial and boundary conditions

It should be clear that, since these equations are first-order in time, it is necessary to specify initial conditions at some initial time (say) $t = 0$. Another important condition which must be stated at this point, is the *boundary condition*, which must typically be defined for the spatial boundaries in many physically interesting situations. The specification of these depends on the transverse partial differential operator involved.

Thus, in the advection equation, we can specify a boundary condition at $x = -X/2$, but not simultaneously one at $x = X/2$. In the diffusion equations, which are second order in space, we typically need to specify two boundary conditions.

The equation is not fully specified unless the boundary conditions are known.

For example, on a one-dimensional transverse space, with boundaries at $x = \pm X/2$, there are three common types of boundary condition:

Dirichlet boundary conditions

The field is fixed at the boundaries, so,

$$\begin{aligned} \mathbf{A}(-X/2) &= \mathbf{A}^{(1)} \\ \mathbf{A}(X/2) &= \mathbf{A}^{(2)}. \end{aligned} \tag{2.19}$$

Neumann boundary conditions

The field gradient is fixed at the boundaries, so,

$$\begin{aligned} \frac{\partial}{\partial x} \mathbf{A}(-X/2) &= \mathbf{B}^{(1)} \\ \frac{\partial}{\partial x} \mathbf{A}(X/2) &= \mathbf{B}^{(2)}. \end{aligned} \tag{2.20}$$

Periodic boundary conditions

The field is periodic at the boundaries, so,

$$\begin{aligned} \mathbf{A}(-X/2) &= \mathbf{A}(X/2) \\ \frac{\partial}{\partial x} \mathbf{A}(-X/2) &= \frac{\partial}{\partial x} \mathbf{A}(X/2). \end{aligned} \tag{2.21}$$

Infinite boundaries

For Maxwell's equations, as with many other examples in physics, it is often the case that we wish to define the boundaries to be at infinity. This option, however, leads to numerical difficulties in defining the transverse lattice. In some cases, we choose to use a finite spatial region or 'window' to model an infinite domain, with the 'window-size' being increased until the results change less than a prescribed accuracy criterion. Another technique is to transform the transverse (i.e., typically spatial) coordinates in some way to map them onto a finite domain. One example is the

mapping $x = w \tan(y)$, which maps an infinite range of x to a finite range of y , from $y = -\pi/2$ to $y = \pi/2$. Note that w can be adjusted to increase or decrease the central region where the mapping is approximately linear; for best accuracy, this should be adjusted so it contains most of the “interesting” region!

To use infinite boundaries in x one has to rewrite the differential equation in terms of the new variable y , which then satisfies an equation defined on a finite domain. This involves using the chain-rule of calculus, so that

$$\frac{\partial}{\partial x} = \frac{\partial y}{\partial x} \frac{\partial}{\partial y}, \quad (2.22)$$

$$\frac{\partial^2}{\partial x^2} = \frac{\partial^2 y}{\partial x^2} \frac{\partial}{\partial y} + \left(\frac{\partial y}{\partial x} \right)^2 \frac{\partial^2}{\partial y^2}. \quad (2.23)$$

Thus, for example, in the y variables, the scalar diffusion equation becomes:

$$\frac{\partial}{\partial t} A(y, t) = \kappa \left(\frac{\partial^2 y}{\partial x^2} \frac{\partial}{\partial y} + \left(\frac{\partial y}{\partial x} \right)^2 \frac{\partial^2}{\partial y^2} \right) A(y, t). \quad (2.24)$$

Chapter 3

Numerical algorithms

To analyse partial differential equations on a computer, we must use some way of calculating derivatives numerically and find stable and accurate techniques of modelling the evolution of the equations. Partial differential equations usually involve functions continuous in both spatial and time dimensions, which are impossible to represent on a digital computer, hence we must use some discrete representation of the functions and their derivatives. Such discrete representations are only approximations of the actual evolution of the continuous functions we are interested in, however we can do a pretty good job, and often using a numerical technique is the only way to model some kinds of physical processes. This is why it is important and interesting to study the numerical ways in which we make these approximations and the numerical techniques that can be used to calculate the evolution of differential equations. In this chapter, we shall look at how derivatives are calculated numerically, and several schemes used to numerically solve differential equations.

3.1 Numerical differentiation

To understand the basics of numerical differentiation, we merely need to recall some high school calculus. To keep stuff nice and simple here, we shall only look at the numerical approximations to first and second derivatives.

3.1.1 First derivative

Let's think about calculating the first derivative to start with. Figure 3.1 may be of some help in understanding the processes I am about to describe. Imagine that we want to find the first derivative (i.e. slope) of some function $f(x)$ at the point x . If we know the analytical form of the function then we could calculate the derivative and use that. In practice in computational physics, we don't know the exact analytical

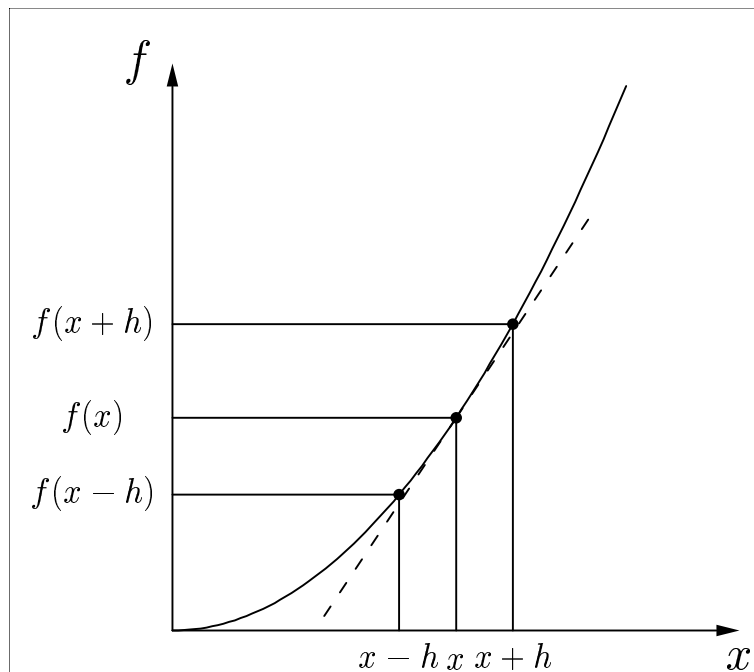


Figure 3.1: Diagram illustrating the ideas of numerical differentiation. To approximate the derivative of the function f at the point x , one must either use a point to the left or right of x and calculate the slope of the line joining the two points, or use both points to the left and right of x and calculate the slope of the line joining the two points.

form of the function we are attempting to find the derivative of, hence we have to do what any good physicist would do: have a guess (some physicists prefer to call this process *approximation*, but we all know the truth). How do we get this guess of the slope? Well, we can be guided by remembering how to calculate the derivative of a curve from first principles. Remember this from high school calculus?

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}. \quad (3.1)$$

This equation says that if we choose a point a distance h away on the x -axis from our point x , we can get a good idea of the slope of the curve at $f(x)$ by taking the slope of the line going through $f(x+h)$ and $f(x)$. The limit tells us that we can get a better and better idea just by letting h get *really* small. On a computer we are limited by just *how* small we can let h become, and so Equation (3.1) is only ever able to be an *approximation* (*guess*) of the actual derivative of f at x . Just to let you know, there are more funky ways of improving the accuracy and/or precision of this guess,

but this is the basic idea used in many cases. It is possible to show¹ using a Taylor expansion about the point $f(x+h)$ that Equation (3.1) has an error (in technical jargon) “of the order of h ”. This statement we will write as $O(h)$ and therefore, to be really explicit about the fact that we are only approximating the derivative, we will write the first derivative as:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h). \quad (3.2)$$

This technique of finding the derivative is quite naïve in the sense that we really aren't using a huge amount of brain power to work out the derivative. So, how can we do a better job? We can use a *centred formula*.

A centred formula uses the idea that the line joining the points $f(x-h)$ and $f(x+h)$ is also an approximation to the slope at the point $f(x)$. The first derivative from first principles then becomes

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x-h)}{2h}. \quad (3.3)$$

Note how this equation is “centred” in x . Even though this equation looks very similar to Equation (3.2), there is a big difference when one considers finite h . Again, using a Taylor series expansion² it can be shown that this centred form has error of order h^2 , hence we write Equation (3.3) as

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2), \quad (3.4)$$

which is really good as we will choose h to be nice and small, and so h^2 will be *really* small. Just as a jargon-type note, if we have an equation that has truncation error / has error / is accurate to order h , then we say that it is *linear* in h . If the equation is accurate to order h^2 , we say that it is *quadratic* in h .

3.1.2 Second derivative

The second derivative $f''(x)$ of the function $f(x)$ is just an extension³ of the concepts in obtaining the first derivative. By using a Taylor series expansion³ we can show that the centred second derivative is

$$f''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} + O(h^2), \quad (3.5)$$

¹see *Numerical Methods in Physics* by A. L. Garcia for details.

²and again, this is discussed in more detail in Garcia.

³guess whose book you should look at for details.

which again is quadratic in h , and therefore nice and accurate.

Just knowing Equations (3.4) and (3.5) is sufficient understanding to be able to cope with the remaining sections, in which we discuss various forms of algorithm for solving PDEs.

3.2 FTCS scheme

The FTCS scheme is possibly the easiest scheme to understand, hence we discuss it first. We will discuss this scheme twice in basically the same way, firstly in a very explicit form, where it is obvious in the computational loop how the differential equation is being processed, and secondly in a matrix form which we will find is more powerful and enables us to easily understand the changes and extensions to the FTCS scheme introduced later.

FTCS stands for **F**orward **T**ime **C**entred **S**pace. What does this mean? Well, it means that we calculate the time derivative in a “forward” manner (using effectively Equation (3.2)) and calculate the spatial derivative in a “centred” manner (using effectively Equation (3.4) and/or Equation (3.5)). It is best to understand this scheme in the context of an example. Let us take for this example, the diffusion equation:

$$\frac{\partial}{\partial t}A(x, t) = \kappa \frac{\partial^2}{\partial x^2}A(x, t). \quad (3.6)$$

In this equation, the scalar field⁴ $A(x, t)$ is a function of both space and time. The variable κ is just a constant describing the rate of diffusion of the field A . Note that we have a *first* derivative in time and a *second* derivative in space, one may think that it would be simpler to discuss as a first example an equation with just a *first* derivative in both time and space, but it turns out that such an equation (the form of which is known as the advection equation) is actually harder to calculate numerically than the diffusion equation. By the end of these notes you may either know the diffusion equation really well, or be completely sick of it, as we will use it in most of the sections and examples that follow.

Okay, to use the FTCS scheme on the diffusion equation, we need to calculate the first derivative in time using the forward first derivative equation (Equation (3.2)) and the second derivative in space using the centred second derivative equation (Equation (3.5)). Substituting these into Equation (3.6) we get:

$$\frac{A(x, t + \Delta t) - A(x, t)}{\Delta t} = \kappa \frac{A(x + h, t) + A(x - h, t) - 2A(x, t)}{h^2} \quad (3.7)$$

⁴In general, one would work with vector fields, such that x would actually be a vector of various (say, spatial) dimensions, it is easier to learn this stuff if we are content just to consider scalar fields.

which is the *discretised form* of the diffusion equation.

Let's simplify things slightly by introducing the notation that spatial indices are represented by a subscript, and time indices are represented by a superscript, such that the value of the function A at the point x_i and at time t_n is represented as A_i^n , or in another form

$$A_i^n = A(x_i, t_n). \quad (3.8)$$

The discretised first derivative in time in this notation becomes

$$\frac{\partial}{\partial t} A(x, t) = \frac{A(x_i, t_{n+1}) - A(x_i, t_n)}{\Delta t} = \frac{A_i^{n+1} - A_i^n}{\Delta t}, \quad (3.9)$$

the discretised second derivative in space becomes

$$\frac{\partial^2}{\partial x^2} A(x, t) = \frac{A(x_{i+1}, t_n) + A(x_{i-1}, t_n) - 2A(x_i, t_n)}{h^2} = \frac{A_{i+1}^n + A_{i-1}^n - 2A_i^n}{h^2}, \quad (3.10)$$

and the diffusion equation is now rewritten as:

$$\frac{A_i^{n+1} - A_i^n}{\Delta t} = \kappa \frac{A_{i+1}^n + A_{i-1}^n - 2A_i^n}{h^2}. \quad (3.11)$$

Since we know the value of the function at the current point in time (t_n which is $A(x_i, t_n) \forall i$), what we are really interested in is the value of the function at the next point in time (t_{n+1} , which is $A(x_i, t_{n+1}) \forall i$). We therefore solve Equation (3.11) for A_i^{n+1} to obtain

$$A_i^{n+1} = A_i^n + \frac{\kappa \Delta t}{h^2} (A_{i+1}^n + A_{i-1}^n - 2A_i^n), \quad (3.12)$$

which is the equation that we are now going to convert into MATLAB code.

I have already written MATLAB code to solve this differential equation (it is adapted from Professor Drummond's code, which was originally adapted from the code in *Garcia*), and I now give the code listing and will give a discussion of the various parts of the program line-by-line. The code listing for `diffuse1.m` is⁵:

```

1 % diffuse1.m - Program to solve the diffusion equation
2 %           using the FTCS method.
3
4 clear;           % clear memory
5 help diffuse1.m % print help information
6
7 %%%%%%%%%% INITIALISATION %%%%%%%%%%%%%%
8
```

⁵It is also given as the sample code at the back, but here I want to discuss it line-by-line, hence I include it here with line numbers.

```

9 % set up input variables
10 tmax = 1; % window size in t
11 nt = 100; % number of grid points in t
12 delta_t = tmax/nt; % step-size in t
13 nplots = 21; % number of plot points in t
14 nplotstep = nt/nplots; % plot step-size in t
15
16 xmax = 2; % window size in x
17 nx = 11; % number of grid points in x
18 delta_x = xmax/(nx + 1); % step-size in x
19
20 kappa = 1; % coupling constant
21 c = kappa*delta_t/delta_x^2; % constant of propagation
22
23 % initial conditions and boundary conditions
24 A = zeros(nx,1); % allocates and zeros a column vector
25 A(floor((nx+1)/2)) = 1/delta_x; % sets up an initial delta 'spike' in A
26 iplot = 1; % initialise the plotting index
27
28 % set up plotting matrices
29 xplot = (1:nx)*delta_x - xmax/2; % row vector of x-coordinates
30 tplot(iplot) = 0; % assign first time-coordinate
31 Aplot(:,iplot) = A(:); % assign first value of A
32
33 %%%%%%%%% MAIN LOOP %%%%%%%%%
34
35 % main loop
36 for i = 1:nt
37
38 % evolve our solution using the FTCS method
39 A(2:(end-1)) = A(2:(end-1)) + c*( A(1:(end-2)) + A(3:end) - 2*A(2:(end-1)) );
40
41 % save current data for plotting
42 if (rem(i,nplotstep) < 1)
43
44 iplot = iplot + 1; % increment plotting index
45 Aplot(:,iplot) = A(:); % record current value of A for plotting
46 tplot(iplot) = i*delta_t; % record current time value
47
48 % print some progress information
49 fprintf('%g out of %g steps completed\n',i,nt);
50
51 end

```

```

52
53 end
54
55 %%%%%%%%% PLOTTING/OUTPUT %%%%%%%%%
56
57 % plot stuff
58 figure(1) % opens a new figure window
59 subplot(121) % divides window into a 1x2 matrix
60 % and assigns next plot to the
61 % left-hand frame (frame 1)
62 mesh(tplot,xplot,Aplot) % wire mesh plot of the solution
63 view([-70 30]) % change the viewing direction
64 xlabel('t') % label the x-axis
65 ylabel('x') % label the y-axis
66 title('Diffusion of a delta spike') % give the thing a title
67
68 subplot(122) % assigns next plot to right-hand
69 % frame (frame 2)
70
71 % find contours of 3D plot and assign to the vector cs
72 cs = contour(xplot,tplot,flipud(rot90(Aplot)),0:0.1:2.5);
73 xlabel('x') % label the x-axis
74 ylabel('t') % label the y-axis
75 clabel(cs,0:5) % label the contours
76 title('Contour of a delta spike') % give the thing a title
77
78 subplot(111) % return plotting mode to one
79 % figure per window

```

Notice that the code is separated into basically three parts: an initialisation section, a main computational loop, and a plotting section. This is a very simple structure that is quite useful in the programs you will be writing in this section of the course.

Going through the code line-by-line we have

lines 1–2: give the name of the program and what it does.

line 4: clear the memory (a good idea to do at the start of each program).

line 5: prints help information. What this does is print any lines of code that appear at the top of the file in comments before any commands are entered. So what it does in this case is merely echo the top two lines of the file out to the terminal window.

lines 9–21: initialise input variables. This code is hopefully fairly self explanatory. A couple

of points to note though: we are not using a variable called h for the step size in x , but Δx ; and we have wrapped up all of the constant terms $\kappa\Delta t/\Delta x^2$ into a variable called `c`. This variable can give us an indication of whether the solution will be stable or not (more on this point later).

- line 24: this line merely allocates some memory for the vector `A`. We allocate its memory by calling the `zeros` function with the size of each dimension of the array as its arguments.
- line 25: we now set up an initial delta spike in `A` (basically a discrete version of a Dirac delta function). The delta spike is dependent upon the step size in x only. We put the delta spike at the mid-point of the array by using the value of `floor((nx+1)/2)`. What does the `floor` function do? Well, it looks at the number in its argument (in this case, the value of $(nx+1)/2$) and if the number isn't an integer, it rounds it *down* to the nearest integer. Why do we want to do this? Well, MATLAB wants integer indices to its arrays, and it's possible that $(nx+1)/2$ could be a non-integer fraction. Just for your information, there is also a function called `ceil` that takes the *ceiling* of the number, i.e. it always rounds a number *up* to the nearest integer.
- line 29: this sets up the vector of x -coordinate positions. Note that we are using the MATLAB colon operator (`:`) to generate the vector. The notation `1:nx` means to create a vector from `1` to `nx` in integer steps. In the case shown here, we are setting up our x -coordinates to be from $-x_{\max}/2$ to $x_{\max}/2$.
- line 31: here we store the initial value of `A` into our plotting storage vector `Aplot`.
- line 39: now we are in the main loop. This is the line that does all of the hard work and implements the evolution of the vector `A` according to the diffusion equation. Compare the code we are using here with Equation (3.12). You may have noticed that we are not assigning to the outside points of `A`. Why is this? We don't assign to these points to maintain the boundary conditions. In this example we are implicitly using the *Dirichlet* boundary conditions that the function value is zero at the boundary. To put this into a physical context, say we are investigating the heating of a metal bar. The diffusion equation describes the way in which the temperature changes with time and position along the bar. The ends of the bar we keep at zero temperature, hence the boundary conditions of our PDE. What is cool⁶ about this line of code is that *all* of the points in space are being acted upon simultaneously, because we are using MATLAB's array *slicing* notation (i.e. in the bit where we say `A(2:end-1)`,

⁶Believe it or not, pun not intended.

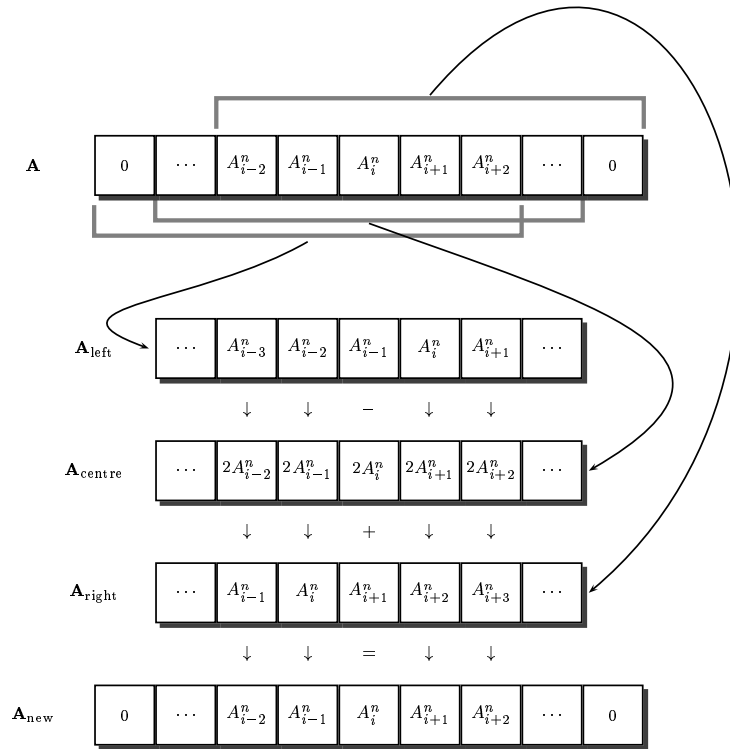


Figure 3.2: The use of element by element addition in calculation of the centred second derivative in space in the FTCS scheme. The vector \mathbf{A} is split into three parts: \mathbf{A}_{left} representing the left-hand elements of \mathbf{A} ; $\mathbf{A}_{\text{centre}}$ representing the centre elements of \mathbf{A} , each multiplied by 2; and $\mathbf{A}_{\text{right}}$ representing the right-hand elements of \mathbf{A} . Each element is added to its corresponding partner shown by the arrows in the figure, to then give the elements of the new value of \mathbf{A} which is denoted here \mathbf{A}_{new} .

we are *slicing* out only the values of \mathbf{A} from the second element to the second to last element, and then using only these elements in the calculation). The first term in the equation on line 39 corresponds directly to the first term in Equation (3.12). As mentioned before, the variable c rolls all of the various constants together. The next term corresponds to the $A_{i+1}^n + A_{i-1}^n - 2A_i^n$ part of Equation (3.12). So, the code `A(1:(end-2))` means to say “all of the *left-hand* x points” i.e. A_{i-1}^n . The code `A(3:end)` means “all of the *right-hand* x points” i.e. A_{i+1}^n . And the code `2*A(2:end-1)` means “all of the *centre* x points” i.e. A_i^n . A pictorial explanation may be helpful at this point. Figure 3.2 shows the process of element by element addition of parts the vector \mathbf{A} in order to calculate the centred second derivative in space in the FTCS scheme. The vector \mathbf{A} is split into three parts: \mathbf{A}_{left} representing the left-hand elements of \mathbf{A} ;

$\mathbf{A}_{\text{centre}}$ representing the centre elements of \mathbf{A} , each multiplied by 2; and $\mathbf{A}_{\text{right}}$ representing the right-hand elements of \mathbf{A} . All elements of these vectors are added to corresponding elements in the other vectors as shown by the arrows in the figure. These element by element processes then produce the elements of \mathbf{A}_{new} which is the new value of the vector \mathbf{A} at the next time step. Notice that the end elements of \mathbf{A} and \mathbf{A}_{new} are set to zero. This is the implementation of the Dirichlet boundary conditions, and is why the intermediate vectors have a length which is two elements shorter than \mathbf{A} .

- line 42: we now do a check to see if we should take a point for plotting⁷. We defined the variable `nplotstep` in the initialisation section so that we can get 21 samples of the function evolution through time. The function we use here is the `rem()` function, and it calculates the remainder of `i/nplotstep`. What we are doing here is using the `if` statement to check to see if this remainder is less than one, if so, to then take a plot point.
- lines 44–46: here we just take the value of `A` and assign it to part of the array `Aplot` and add to the time recording vector (`tplot`) with the current time.
- line 49: this just prints out the number of steps out of the total number of steps we have completed. It's sometimes helpful to have progress information like this to make sure your program is doing something when it is running.
- line 58: this opens a new figure window, and will call it **Figure 1** (hence the "1").
- line 59: this splits the window into two parts, so that we can plot something first on the left half of the figure, and then on the right half. The notation defines a matrix, namely: `subplot(121)` means make a subplot of the current figure window in a 1×2 matrix (the first 1 and 2), and put the next plot into the first element of that matrix (hence the last 1, and the first element of a 1×2 matrix is the left-hand one, and so the plot goes into the left hand half of the figure window).
- line 62: make a *mesh* plot of the array `Aplot` using the vectors `tplot` and `xplot` along the x and y axes respectively.
- line 63: change the angle that we look at the graph from.
- lines 64–66: put x and y labels on the graph, and give it a title.
- line 68: same idea as line 59, just this time we are plotting into the second element (hence the 2 at the end of the `subplot` command).

⁷Have a think about why we may *not* want to take all of the data points.

line 72: generate a variable used for plotting contours of the array `Aplot`. Don't worry about this too much, if you're interested type `help contour` for MATLAB's information on the `contour` function.

line 78: `subplot(111)` resets any subsequent plotting output to take up the whole window: a 1×1 matrix, first element.

The stability of this scheme for the diffusion equation can be shown to be dependent upon the value of the variable c . It is possible to show (and for the analysis on this point see Garcia) that for $c \geq 0.5$ the solution will be unstable. In fact, it is possible to show for the advection equation that the FTCS scheme is unstable for all values of the time step!⁸ One can think of the evolution of the function according to the discretised diffusion equation as being due to perturbations on the currently known step in time. If you look at Equation (3.12), you can see that the new point in time is basically the old point in time plus some perturbation. It isn't desirable in perturbation theory to have large perturbations, so if c is too big, then the perturbation is large and the solution is likely to diverge as we iterate it further. So, when you are running the sample code and you find for certain parameters that the solution "blows up", check the value of c and see what it is; it is quite possible that it is greater than 0.5. To fix this one needs to either *increase* the x step width, and/or *decrease* the time step or coupling constant. Why is this do you reckon? Have a think on it, and look at the diffusion equation again.

Hopefully now you have a reasonable grasp of the FTCS scheme. I'll now introduce a matrix notation that will help us understand subsequent schemes.

3.3 Matrix form of FTCS scheme

To find the matrix form of the FTCS scheme we first recall the discretised form of the FTCS scheme:

$$\frac{A_i^{n+1} - A_i^n}{\Delta t} = \kappa \frac{A_{i+1}^n + A_{i-1}^n - 2A_i^n}{h^2}. \quad (3.13)$$

If we stare at this equation for long enough we might be able to see exactly why we can turn this into a matrix form. Let's pay attention to just the right-hand side of Equation (3.13). We can see that we have three terms of the solution $A(x, t)$ at three points in space, but at the *same* point in time. This suggests that we could write $A(x, t)$ as a vector with each element representing the function value at each position in space and for the current point in time. Also, to "pick out" the relevant points from the vector \mathbf{A} we could multiply it by a square matrix⁹ with relevant numbers

⁸Hence, we use different techniques to treat the advection equation.

⁹Why square? Well, we will get back a vector with the same dimensions as before after applying the matrix to it.

down its diagonals. To make this more clear, let's consider the following matrix:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad (3.14)$$

if we then multiply this matrix by the column vector:

$$\begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \quad (3.15)$$

we get out

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}. \quad (3.16)$$

At which point you go “duh, Paul, that’s because it was the identity”. Well, yes, it was. However, if we wanted to “pick out” the bottom two elements of the vector we would use the following matrix:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \quad (3.17)$$

and we get

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} v_2 \\ v_3 \\ 0 \end{pmatrix}. \quad (3.18)$$

So what? you may ask. Well, look at the vector output. It has what were the bottom two elements now in the top two positions of the new vector, and has therefore picked out the elements that we wanted. It has also shifted the elements of the vector up one (in some sense). As we will hopefully see soon, this is a handy property that we will use. Now, imagine that each element in the vector corresponds to a point in space. Also imagine that we want to calculate some function such that the new “current point in space” is equal to the difference between the points in space on either side of it. Using the vector we have at the moment, we therefore want the middle element (v_2) to become $v_1 - v_3$. We don't know what the other elements should become at this stage since there is no point “outside” them. So we want a vector output that looks something like this:

$$\begin{pmatrix} ? \\ v_1 - v_3 \\ ? \end{pmatrix}. \quad (3.19)$$

We could do this by using the following matrix:

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}. \quad (3.20)$$

Let's see if it works. . .

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} -v_2 \\ v_1 - v_3 \\ v_2 \end{pmatrix}. \quad (3.21)$$

Well, it did basically what we wanted. The central element is now equal to $v_1 - v_3$, but the top element is $-v_2$ and the bottom element is v_2 —why is this? Well, imagine that there are zeroes above and below the vector. So, to calculate what the top element would be, we would have “something like” $0 - v_2$ for the top element and “something like” $v_2 - 0$ for the bottom element. Hence we get the output we see in Equation (3.21).

Note that the process here is much like what we want to do with the diffusion equation, namely: add the left- and right-hand points of a vector and subtract twice the central point. Using the (oversimplified) 3×3 matrices above we would do something like this:

$$\begin{pmatrix} -2 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} v_2 - 2v_1 \\ v_1 - 2v_2 + v_3 \\ v_2 - 2v_3 \end{pmatrix}. \quad (3.22)$$

From the central element of the output vector we can see that this has implemented what we wanted to achieve: $v_2' = v_1 + v_3 - 2v_2$ —look like the diffusion equation? Now we hopefully have a feel for the structure of the \mathbf{D} matrix for the diffusion equation.

Another way of looking at this is if we take the right-hand side of the discretised diffusion equation:

$$\kappa \frac{A_{i+1}^n + A_{i-1}^n - 2A_i^n}{\Delta x^2}, \quad (3.23)$$

then we can rewrite it as

$$\sum_{j=1}^N D_{ij} A_j^n, \quad (3.24)$$

where the D_{ij} are the elements of some matrix \mathbf{D} whose elements can be written as

$$D_{i,j} = \frac{\kappa}{\Delta x^2} (\delta_{i+1,j}^n + \delta_{i-1,j}^n - 2\delta_{i,j}^n), \quad (3.25)$$

where the $\delta_{i,j}$ are Kronecker delta functions and are equal to 1 if and only if $i = j$; zero otherwise. If we now think of the A_i^n in the discretised diffusion equation as

elements in a column vector, we can rewrite the diffusion equation in the following form:

$$\frac{\mathbf{A}^{n+1} - \mathbf{A}^n}{\Delta t} = \frac{\kappa}{\Delta x^2} \mathbf{D} \mathbf{A}^n. \quad (3.26)$$

We wish to find the value of the vector \mathbf{A} at the next point in time, which is at t_{n+1} , hence we solve this equation for \mathbf{A}^{n+1} :

$$\mathbf{A}^{n+1} = \mathbf{A}^n + \frac{\kappa \Delta t}{\Delta x^2} \mathbf{D} \mathbf{A}^n, \quad (3.27)$$

$$= \left(\mathbf{I} + \frac{\kappa \Delta t}{\Delta x^2} \mathbf{D} \right) \mathbf{A}^n, \quad (3.28)$$

$$= \mathbf{M} \mathbf{A}^n, \quad (3.29)$$

where \mathbf{I} is the identity matrix and \mathbf{M} is just a matrix to make the equations look nice and simple. So, our differential equation problem has reduced to finding the \mathbf{D} matrix and then constructing the \mathbf{M} matrix from that and then repeatedly applying \mathbf{M} to evolve the vector \mathbf{A} . This in its essence is all one has to code to use this representation.

Now, given that we have the boundary conditions that the function is constant at the left- and right-hand edges (*Dirichlet* boundary conditions) and are that they equal to zero, we construct the following \mathbf{D} matrix:

$$\mathbf{D} = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & 1 & -2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}. \quad (3.30)$$

For an \mathbf{A} vector (at time instance n) such as

$$\mathbf{A}^n = \begin{pmatrix} A_1^n \\ A_2^n \\ A_3^n \\ \vdots \\ A_N^n \end{pmatrix}, \quad (3.31)$$

verify for yourself that one application of the matrix \mathbf{D} onto \mathbf{A}^n gives

$$\mathbf{D} \mathbf{A}^n = \begin{pmatrix} 0 \\ A_1^n - 2A_2^n + A_3^n \\ A_2^n - 2A_3^n + A_4^n \\ \vdots \\ A_{N-2}^n - 2A_{N-1}^n + A_N^n \\ 0 \end{pmatrix}. \quad (3.32)$$

Notice how the top and bottom rows of zeroes in \mathbf{D} ensure that the “edge” elements of \mathbf{A} stay zero. Thus the boundary conditions can be made implicit in the \mathbf{D} matrix. Handy eh?

What if we wanted periodic boundary conditions? Such a physical example would be if we wished to model the diffusion of a wave through time and space. So, if we had periodic boundary conditions, we could let the wave travel across the “window” in space we have defined (to the left, say) and it would appear from the right traveling to the left. This would allow us to let the wave “see” a longer “distance” without having to have really large windows in space. To do this with a variant of the \mathbf{D} matrix we would have to think carefully about the problem, and then realise how the \mathbf{D} matrix acts upon the \mathbf{A} vector in an element-by-element fashion. You may wish to verify that the \mathbf{D} matrix that would give diffusion equation type behaviour with periodic boundary conditions is

$$\mathbf{D} = \begin{pmatrix} -2 & 1 & 0 & \cdots & 0 & 0 & 1 \\ 1 & -2 & 1 & \cdots & 0 & 0 & 0 \\ 0 & 1 & -2 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -2 & 1 & 0 \\ 0 & 0 & 0 & \cdots & 1 & -2 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 1 & -2 \end{pmatrix}. \quad (3.33)$$

Note the 1’s in the top-right and bottom-left corners. They are like where the \mathbf{D} matrix has “wrapped” around from the other side of the matrix. It is this “wrapped around” kind of structure in the \mathbf{D} matrix that gives the periodic nature of the boundary conditions.

What are the advantages of using this matrix scheme? Well, your code is a lot easier to read, and a lot easier to change. The main bit of mental work on the programmer’s part is in calculation of what the \mathbf{D} matrix should be to apply to the \mathbf{A} vector. Then, inside the main loop, all one has to do is have a line saying something like:

```
A = M*A;
```

to evolve the solution. It is literally *that* simple.

Ok. I hope that you have a bit of a grasp of the matrix representation of differential equations now. Using this form for the equations allows me to be somewhat less verbose in my explanations, and makes some of the mathematics a bit more obvious. We go on to investigate the implicit FTCS scheme in the next section.

3.4 Implicit FTCS scheme

The FTCS scheme is known as an *explicit* numerical technique. It is explicit since it calculates the new value of the evolving function given the function’s known current values. This technique is rather unstable however. In fact, it can be shown for the advection equation that the explicit FTCS scheme is unstable *all for values of the time step*. All is not lost, since we can use some tricks to have schemes with better stability. One such scheme is the *implicit* FTCS scheme. The scheme is still forward in time and centred in space however, in a sense, it calculates the next value of the function in time using the *next* value of the function in time. “Hang on”, you say, “but we don’t know that yet”. Correct. But, what we *can* do is just write the equations out *as if* we knew what the next point in time was and see what happens. What is our motivation or basis for doing this? Well, for sufficiently small time steps, the function changes very little, so \mathbf{A}^n is “pretty close” to \mathbf{A}^{n+1} , and so using such a trick is not such an evil thing after all.

Using this idea, let’s rewrite the matrix form of the diffusion equation, but this time let the \mathbf{D} matrix act on the future value of \mathbf{A} :

$$\mathbf{A}^{n+1} = \mathbf{A}^n + \frac{\kappa\Delta t}{\Delta x^2}\mathbf{D}\mathbf{A}^{n+1}. \quad (3.34)$$

Note that we are only operating the \mathbf{D} matrix on the next point in time—we are still using the current time point for the actual calculation. If we now solve this equation for \mathbf{A}^{n+1} (as we did for the matrix form of the explicit FTCS scheme) we obtain:

$$\mathbf{A}^{n+1} - \frac{\kappa\Delta t}{\Delta x^2}\mathbf{D}\mathbf{A}^{n+1} = \mathbf{A}^n, \quad (3.35)$$

$$\left(\mathbf{I} - \frac{\kappa\Delta t}{\Delta x^2}\mathbf{D}\right)\mathbf{A}^{n+1} = \mathbf{A}^n, \quad (3.36)$$

$$\mathbf{A}^{n+1} = \left(\mathbf{I} - \frac{\kappa\Delta t}{\Delta x^2}\mathbf{D}\right)^{-1}\mathbf{A}^n, \quad (3.37)$$

$$\mathbf{A}^{n+1} = \mathbf{M}\mathbf{A}^n. \quad (3.38)$$

Note that on the second to last line the *matrix inverse* is taken; it is *not* a division operation.

Again, we see that all we have to do is calculate the relevant \mathbf{D} matrix, construct the matrix \mathbf{M} (which is just a conglomeration of other matrices) and you’re away laughing (so-to-speak). Also note that we don’t need to alter the code producing the matrix form of the explicit FTCS scheme that much to implement the implicit scheme. We just need to take the matrix inverse, and a plus sign is changed to a minus sign. Now I hope you see the power of using the matrix formalism for numerically representing differential operators.

It turns out that the implicit FTCS scheme is *unconditionally stable*. This means that it *converges* to a solution for any time step. This may seem amazing, and yes, it is, but there is a down side: the scheme isn't as accurate as the explicit scheme. To get around this problem we can take something like the “average” of the both the implicit and explicit schemes and we can then have the best of both worlds: stability and accuracy. We discuss such a scheme in the next section; it is called the Crank-Nicolson scheme.

3.5 Crank-Nicolson scheme

As mentioned in the previous section, this scheme is basically the average of both the explicit and implicit FTCS schemes. This gives us a diffusion equation in matrix form as

$$\mathbf{A}^{n+1} = \mathbf{A}^n + \frac{\kappa\Delta t}{\Delta x^2} \mathbf{D} \left(\frac{\mathbf{A}^{n+1} + \mathbf{A}^n}{2} \right), \quad (3.39)$$

where we have used the average of the current \mathbf{A} vector (explicit part) and the future \mathbf{A} vector (implicit part). Solving this for \mathbf{A}^{n+1} gives

$$\mathbf{A}^{n+1} = \left(\mathbf{I} - \frac{\kappa\Delta t}{2\Delta x^2} \mathbf{D} \right)^{-1} \left(\mathbf{I} + \frac{\kappa\Delta t}{2\Delta x^2} \mathbf{D} \right) \mathbf{A}^n, \quad (3.40)$$

$$\mathbf{A}^{n+1} = \mathbf{M} \mathbf{A}^n, \quad (3.41)$$

where again we have wrapped up the \mathbf{D} matrix containing terms into the matrix \mathbf{M} .

We can implement this equation straight away in the code that we already have implemented for the matrix form for the explicit FTCS scheme, it now just being a matter of taking one matrix inverse and a couple of matrix multiplications to create the \mathbf{M} matrix and then, again, just run the following code in the main loop:

```
A = M*A;
```

and that's it!

Just to reiterate, this is stable, accurate, easy to understand where all of the parts come from, and very simple to alter the differential equation that we are attempting to model without large changes in code. Other schemes can have handy properties as well, which is why we also study the FFT method, also known as the spectral method in the next section.

3.6 Spectral method (FFT scheme)

With the methods that we have used so far, we have been interested mainly in calculating the new value of the function in the *same space* as the solution exists in. For

example, the diffusion equation has been solved in each example above in *position space*. However, there are times when it can be useful to perform the evolution of a function in some other space. Spectral methods use these ideas by Fourier transforming the problem and performing the calculation in Fourier space, it being easier *in some cases* for the solution to be calculated there than in position space. The discussion here follows that in *A First Course in Computational Physics* by P. DeVries.

The process that spectral methods use is to take an evolving function in some space, transform it into Fourier space, calculate the evolution of the function in Fourier space, and then transform back to the original space by taking the inverse Fourier transform. This process is then iterated for each time step of the algorithm.

Let's describe this process in more detail by considering the diffusion equation¹⁰:

$$\frac{\partial}{\partial t}A(x, t) = \kappa \frac{\partial^2}{\partial x^2}A(x, t). \quad (3.42)$$

The Fourier transform of $A(x, t)$ is

$$\alpha(k, t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} A(x, t) e^{-ikx} dx. \quad (3.43)$$

This has transformed the function A from x -space to k -space. Note that I will use the terms x -space, position space and inverse space interchangeably, as well as the terms k -space, momentum space and Fourier space. We can now rewrite $A(x, t)$ as

$$A(x, t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \alpha(k, t) e^{+ikx} dk. \quad (3.44)$$

These transformations between x and k are occurring for the *same instance in time*. Equation (3.44) allows us to rewrite Equation (3.42) as

$$\frac{\partial}{\partial t} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \alpha(k, t) e^{+ikx} dk = \kappa \frac{\partial^2}{\partial x^2} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \alpha(k, t) e^{+ikx} dk. \quad (3.45)$$

On the left-hand side of this equation, the partial derivative operates only on $\alpha(k, t)$ since it is the only thing dependent upon t . On the right-hand side, the partial derivative acts only on the exponential, since this is the only thing dependent upon x . We take the derivatives through the integral signs and act them on the things

¹⁰I *told* you you'd see a lot of the diffusion equation!

affected.

$$\frac{\partial}{\partial t} \int_{-\infty}^{\infty} \alpha(k, t) e^{+ikx} dk = \kappa \frac{\partial^2}{\partial x^2} \int_{-\infty}^{\infty} \alpha(k, t) e^{+ikx} dk, \quad (3.46)$$

$$\Rightarrow \int_{-\infty}^{\infty} \frac{\partial}{\partial t} \alpha(k, t) e^{+ikx} dk = \kappa \int_{-\infty}^{\infty} \alpha(k, t) \frac{\partial^2}{\partial x^2} e^{+ikx} dk, \quad (3.47)$$

$$\Rightarrow \int_{-\infty}^{\infty} \frac{\partial}{\partial t} \alpha(k, t) e^{+ikx} dk = \kappa \int_{-\infty}^{\infty} \alpha(k, t) [-k^2] e^{+ikx} dk. \quad (3.48)$$

We now multiply both sides of this equation by $e^{-ik'x}$. This is just a mathematical trick, since we can “see” a possibility of constructing a Dirac delta function out of an integral of exponentials. Anyway, to carry on with the explanation, we then integrate both sides over x , giving

$$\int_{-\infty}^{\infty} \frac{\partial}{\partial t} \alpha(k, t) e^{+ikx} dk = \kappa \int_{-\infty}^{\infty} \alpha(k, t) [-k^2] e^{+ikx} dk, \quad (3.49)$$

$$\Rightarrow \int_{-\infty}^{\infty} e^{-ik'x} \int_{-\infty}^{\infty} \frac{\partial}{\partial t} \alpha(k, t) e^{+ikx} dk dx = \kappa \int_{-\infty}^{\infty} e^{-ik'x} \int_{-\infty}^{\infty} \alpha(k, t) [-k^2] e^{+ikx} dk dx. \quad (3.50)$$

We now exchange the order of integration to give

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \frac{\partial}{\partial t} \alpha(k, t) e^{-ik'x} e^{+ikx} dx dk = \kappa \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \alpha(k, t) [-k^2] e^{-ik'x} e^{+ikx} dx dk, \quad (3.51)$$

$$\Rightarrow \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \frac{\partial}{\partial t} \alpha(k, t) e^{+i(k-k')x} dx dk = \kappa \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \alpha(k, t) [-k^2] e^{+i(k-k')x} dx dk. \quad (3.52)$$

We note that

$$\int_{-\infty}^{\infty} e^{ik(k-k')x} dx = 2\pi \delta(k - k'), \quad (3.53)$$

and performing the integral over k , this function will pick out all values of k that are equal to k' and we get the differential equation

$$\frac{\partial}{\partial t} \alpha(k, t) = -k^2 \kappa \alpha(k, t), \quad (3.54)$$

where we have done the trivial relabeling of k' to k . Note that we now only have one first order partial derivative, instead of a first order and a second order partial derivative—this has simplified the calculation rather a lot. Equation (3.54) has the exact analytical solution

$$\alpha(k, t) = e^{-k^2 \kappa t} \alpha(k, 0), \quad (3.55)$$

where $\alpha(k, 0)$ is calculated from the initial distribution of $A = A(x, 0)$ via the Fourier transform

$$\alpha(k, 0) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} A(x, 0) e^{-ikx} dx. \quad (3.56)$$

This has completely solved the problem, and we don't really need to use a computer to do the time evolution for us. However, it gives us a technique useful in more general and more difficult problems. Once we have $\alpha(k, t)$ we just take the inverse Fourier transform to obtain the desired solution:

$$A(x, t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \alpha(k, t) e^{+ikx} dk, \quad (3.57)$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-k^2 \kappa t} \alpha(k, 0) e^{+ikx} dk. \quad (3.58)$$

Overall, we have three steps:

1. Transform the original differential equation and initial conditions into Fourier space.
2. Solve the (hopefully simpler) equation in transform space.
3. Perform the inverse Fourier transform to get the solution in the original variables.

The only problem now is to know what k values to use once the solution is in Fourier space. This is quite a problem with this method as such a choice depends upon the implementation of the Fast Fourier Transform (FFT) in the language or with the system you are using. In the rest of this discussion I shall assume that the implementation is MATLAB's `fft`. Imagine that we have n_x grid points in space, and we assume that they are spaced equally at intervals of Δx . The *range* of x points is therefore $X = (n_x - 1)\Delta x$. Since we only have a finite number of x points¹¹ each one must map to a point in k -space, and so the number of k points is equal to n_x . In MATLAB's implementation of the FFT, the points in k are separated by an amount Δk which is related to the parameters in x by

$$\Delta k = \frac{1}{X} = \frac{1}{(n_x - 1)\Delta x}. \quad (3.59)$$

¹¹Note that we are using a Fast Fourier Transform (FFT) here, which is derived from the Discrete Fourier Transform (DFT). These transforms are, however, *not* Fourier transforms in the strict sense since they are discretised finite data-set approximations of a continuous set of data with infinite extent. Although they are related remember that they *aren't the same thing*. I will throw the terms around here a bit, but just keep this point in mind.

In the problems solved in this course, the x values lie roughly equally about $x = 0$. We therefore set up our k values in a similar manner, where we have a set $k = 0$ (zero momentum) position and successively larger values of k on either side, each separated by Δk until a sufficient number of points is obtained. More explicitly this is

$$k = \dots, -\Delta k, 0, \Delta k, \dots \quad (3.60)$$

where the number of points in k is equal to the number of points in x . So, if n_x were equal to 5, with $\Delta x = 1$, then we would have the x points:

$$x = -2, -1, 0, 1, 2, \quad (3.61)$$

the range of x points is $4(= 2 - (-2))$, so $\Delta k = \frac{1}{4}$ and hence we have the k points:

$$k = -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}. \quad (3.62)$$

This technique seems pretty tricky, but it seems to be fairly specific to certain problems. What if we want to solve a differential equation that is more complicated than the diffusion equation, as is often the case in real life? Well, a possible answer to this is to use what is known as the *pseudo-spectral method* (also known as the split-step FFT scheme), which we discuss in the next section.

3.7 Pseudo-spectral method (Split-step FFT)

There exist some physically interesting problems that have parts easily solvable in position space and parts easily solvable in momentum space. For these kinds of problems we can use a technique known as the pseudo-spectral method. Essentially, all this technique does is break the problem down into parts involving the bits easily solvable in each space and then merely makes the relevant transformations to move between spaces. Breaking the problem into parts usually involves an approximation to the actual evolution desired, but since numerical computation of the evolution PDEs is an approximation, it doesn't hurt to do another if it allows us to solve a problem, and we can be reasonably accurate as it turns out anyway. In this discussion I will again follow the text of DeVries.

Just to be different, we're going to analyse the one-dimensional Schrödinger equation instead of the diffusion equation¹². The reason for this is because the diffusion equation would reduce back to the standard FFT method described in the last section, and therefore wouldn't be very informative or interesting. The 1-D Schrödinger equation is

$$i\hbar \frac{\partial}{\partial t} \psi(x, t) = -\frac{\hbar^2}{2m} \frac{\partial^2}{\partial x^2} \psi(x, t) + V(x)\psi(x, t), \quad (3.63)$$

¹²Yay!

where m is the mass of the quantum mechanical particle, $\psi(x, t)$ is its wavefunction and $V(x)$ is some potential dependent only upon the spatial position. We can write this in terms of a kinetic energy term \mathbf{T} and a potential energy term \mathbf{V} as

$$i\hbar \frac{\partial}{\partial t} \psi(x, t) = (\mathbf{T} + \mathbf{V})\psi(x, t). \quad (3.64)$$

Note that \mathbf{T} is basically the \mathbf{D} matrix (in general, the derivative operator) that we have been using in most of the examples to date. The potential \mathbf{V} merely multiplies ψ . One should mention at this point that the time dependence of Schrödinger's equation in Equation (3.63) can be solved *formally*, but this is really boring, and we want to do this on a computer anyway. Nevertheless, the formal solution can lead us to a practical method of solving Schrödinger's equation computationally. The solution of Schrödinger's equation in the form of Equation (3.64) is

$$\psi(x, t) = e^{-i(\mathbf{T}+\mathbf{V})\delta_t/\hbar} \psi(x, t_0), \quad (3.65)$$

where $\delta_t = t - t_0$ and t_0 is the initial point in time.

Note that this solution involves the exponential of operators. How do we take exponentials of operators? In much the same way we would take the exponential of a scalar;

$$e^{\mathbf{A}} = 1 + \mathbf{A} + \frac{\mathbf{A} \cdot \mathbf{A}}{2} + \frac{\mathbf{A} \cdot \mathbf{A} \cdot \mathbf{A}}{3!} + \dots, \quad (3.66)$$

When we realise that \mathbf{A} could have a representation as a matrix, one can imagine that this could get nasty to calculate fairly quickly.

It would be really cool if we could write the exponential in Equation (3.65) as

$$e^{-i(\mathbf{T}+\mathbf{V})\delta_t/\hbar} = e^{-i\mathbf{T}\delta_t/\hbar} e^{-i\mathbf{V}\delta_t/\hbar}, \quad (3.67)$$

but we can't because \mathbf{T} and \mathbf{A} are operators and in general don't commute. However, even though the above relation isn't true in a strictly mathematical sense, it doesn't mean it isn't useful as an *approximation* for what we want to do. Performing this *decomposition* of the exponential into two factors is known as the *split-operator approach* and is why this technique is also known as the *split-step FFT* method. Although it may seem a bit bizarre, this method is actually widely applicable, but we won't go into that here.

Equation (3.67) is only accurate to $O(\delta_t)$, nevertheless if we take the *symmetric decomposition*

$$e^{-i(\mathbf{T}+\mathbf{V})\delta_t/\hbar} \approx e^{-i\mathbf{V}\delta_t/2\hbar} e^{-i\mathbf{T}\delta_t/\hbar} e^{-i\mathbf{V}\delta_t/2\hbar}, \quad (3.68)$$

then the split-operator approximation is accurate to $O(\delta_t^2)$ (which is much better). What Equation (3.68) is saying is that we will work out the solution for half of the \mathbf{V} step, then all of the \mathbf{T} step then the other half of the \mathbf{V} step. The process we will follow is then:

1. Solve half of the \mathbf{V} step in x -space.
2. Transform into k -space.
3. Solve all of the \mathbf{T} step in k -space.
4. Transform back to x -space.
5. Solve the other half of \mathbf{V} in x -space.
6. Giving the final solution.

Right, let's do it then.

The wavefunction ψ at time t is approximated by

$$\psi(x, t) \approx e^{-i\mathbf{V}\delta_t/2\hbar} e^{-i\mathbf{T}\delta_t/\hbar} e^{-i\mathbf{V}\delta_t/2\hbar} \psi(x, t_0). \quad (3.69)$$

Since \mathbf{V} is evaluated in x -space, we define the function

$$\phi(x) = e^{-i\mathbf{V}\delta_t/2\hbar} \psi(x, t), \quad (3.70)$$

and so we want to evaluate the effect of the kinetic exponential acting on ϕ . In other words

$$e^{-i\mathbf{T}\delta_t/\hbar} \phi(x). \quad (3.71)$$

This is nasty in x -space, but easy in k -space, so we Fourier transform:

$$\Phi(k) = \mathcal{F}[\phi(x)] = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \phi(x) e^{-ikx} dx. \quad (3.72)$$

After a lot of hard work, one can eventually show that

$$e^{-i\mathbf{T}\delta_t/\hbar} \phi(x) = \mathcal{F}^{-1} \left[e^{-iT(k)\delta_t/\hbar} \Phi \right], \quad (3.73)$$

where we have defined the quantity

$$T(k) = \frac{\hbar^2 k^2}{2m}. \quad (3.74)$$

Therefore, the full propagation is described by:

$$\psi(x, t) \approx e^{-iV(x)\delta_t/2\hbar} \mathcal{F}^{-1} \left[e^{-iT(k)\delta_t/\hbar} \mathcal{F} \left[e^{-iV(x)\delta_t/2\hbar} \psi(x, t_0) \right] \right]. \quad (3.75)$$

And that's it, the solution evolves after jumping back and forth between x - and k -space, but nevertheless solves a reasonably difficult problem in an efficient and accurate manner.

Chapter 4

Numerical Errors

Numerical techniques necessarily approximate the systems we are trying to model, as we are taking discrete points from continuous functions. It therefore is obvious that errors are going to occur, and as it turns out, there are a few kinds of errors to keep in mind when running and testing your solutions to various physical problems. The errors can be caused by such things as the imprecision inherent in using digital computers to errors as a result of choice of algorithm. This chapter discusses many of the kinds of errors you should be aware of when programming, testing and writing up the projects for this course.

4.1 Time discretisation errors

We have assumed (to a first approximation) that the time derivative of the field \mathbf{A} doesn't vary in the small time interval from t_n to t_{n+1} . This is obviously a better approximation when Δt is small. However, it is useful to think about this more quantitatively. From the Taylor expansion, the error for making each step in time is $O(\Delta t^2)$. However, one must take N steps of size Δt to reach a given time t_N , where $N = (t_N - t_0)/\Delta t$. Hence, the accumulated error in reaching a given time t_N is approximately proportional to $\Delta t = \Delta t^2/\Delta t$ in the FTCS methods, provided the error simply adds, and does not grow exponentially. This error can therefore be reduced by halving the time-step, until the changes that occur are less than the error tolerance.

4.2 Space discretisation errors

Spatial discretisation introduces further errors. From the Taylor expansion in space, the local error for one step in time due to spatial discretisation is proportional to

$\Delta x^2 \Delta t$, which means that the accumulated errors should be approximately proportional to $\Delta x^2 = \Delta x^2 \Delta t / \Delta t$. Again, this can be reduced by halving the step-size in space, until the changes that occur are less than the error tolerance.

4.3 Windowing errors

Boundary conditions have to be included by careful treatment of the discretised differential operator near the boundaries. This introduces no error if the spatial windows are finite. If the real spatial window is infinite, then the approximation of using a finite spatial window introduces further errors. This can be checked through doubling the window-size, while keeping the step-sizes constant. Note that using mapping techniques also reduces windowing errors. However, mapping does not really eliminate them totally. Instead, it transforms windowing errors into new spatial-discretisation errors—caused by the rapidly increasing spatial variation of the equation terms, as the transformed boundaries in the mapping space are approached. Even so, this method illustrates how mapping techniques can be combined with a numerical method.

4.4 Instability errors

It is possible that the global errors behave rather differently to the above estimates. For example, the local errors may cancel each other (relatively unusual, owing to Murphy's law), or they may themselves become amplified by the algorithm. If this happens, then there is an exponential growth in the errors, which can become enormously large very quickly. Under these conditions, the numerical solution is essentially meaningless.

4.5 Roundoff (truncation) errors

Round-off error is due to the fact that real numbers are represented by binary number approximations in a computer. This effect is complementary to the other error sources, since it accumulates with the total number of floating-point operations, and therefore gets larger when the step-sizes are reduced—or as the spatial window is increased. This effect is often ignored, under the assumption that commonly used double-precision floating-point IEEE operations (64bit, 16-17 decimals) have negligible round-off errors. However, for very accurate calculations with large lattices, this effect can become important. With some computer languages, it is possible to repeat the calculation with quadruple-precision (128bit, 32-34 decimals). If the calculations give results that differ by more than the error tolerance at two different precisions, then round-off is clearly a problem.

4.6 Range errors

Range errors occur when numbers get “too big” or “too small” for the precision we are using. For instance, single precision typically gives numbers with 7 significant digits, with a range of $10^{\pm 37}$. Therefore, if the numbers we are calculating become larger (smaller) than 10^{37} (10^{-37}); errors are going to be introduced into the calculation. Some compilers are quite intelligent and they can introduce code into the executable that do *range checking*, so that at run-time the program can check for *overflow* (numbers too big) or *underflow* (numbers too small) errors. Often, however, if a number gets too small it is represented as zero, and if a number gets too big then unpredictable things can happen, all dependent upon the computer hardware, language (and hence compiler) being used, and other factors.

Chapter 5

Projects

5.1 PROJECT I

Write a computer code to solve the one-dimensional scalar diffusion equation:

$$\frac{\partial}{\partial t} A(x, t) = \kappa \frac{\partial^2}{\partial x^2} A(x, t), \quad (5.1)$$

with zero boundary conditions at $x = \pm\infty$.

Use the explicit FTCS method with Dirichlet boundary conditions, of form $A(-L/2, t) = A(L/2, t) = 0$, and the initial condition of $A(x, 0) = \frac{4}{\sqrt{2\pi}} \exp(-8x^2)$. Note that the ‘window’ of $x = \pm L$ must be chosen to be large enough so that having a finite window doesn’t introduce a large error, but not too large so that the step-size in x produces large discretisation errors.

The code should be written in MATLAB; as an option for experienced students, you may wish to convert this to C, to demonstrate the speed improvement obtained with compiled code. An example code in MATLAB is attached to these notes. It will need some modifications to the initial conditions! You should try to rewrite this code in a matrix equation form.

The C code you write can **either** plot the data with PLPLOT, **or** you can read the output into a MATLAB program for plotting. You could also even try using the C-subroutine options of MATLAB. The purpose of this is to show how MATLAB can be used for program development and prototyping, with C code generally giving faster production runs after the prototyping stage.

The sample code deliberately plots the results for only a subset of all the calculated time-points—do you understand why this is? Note that the analytic solution to this problem (on an infinite line) is something like (check this yourself):

$$A(x, t) = \frac{1}{\sigma(t)\sqrt{(2\pi)}} \exp\left[\frac{-x^2}{2\sigma^2(t)}\right], \quad (5.2)$$

where $\sigma(t) = \sqrt{2\kappa t + 1/16}$.

Your report should include:

- A general description of the problem, and the physics background.
- An FTCS code that you wrote and tested, with a graph of results and errors.
- A table showing how maximum errors vary with step-size and window-size, including what happens when $\Delta x^2/\Delta t$ is varied.
- Results for an infinite domain, using the implicit FTCS method. This is very simple if you use the MATLAB matrix notation,
- The source code should also be attached to an email to cochrane@physics.uq.edu.au, along with the relevant input parameters used to generate the plots.

5.2 PROJECT II

Modify and test—checking errors and convergence as before—either the C or MATLAB code to solve **ONE** of the following problems, using a semi-implicit method:

1. Solve the advection equation for a Gaussian pulse; what instabilities do you observe?

$$\frac{\partial}{\partial t} A(x, t) = -v \frac{\partial}{\partial x} A. \quad (5.3)$$

What results do you get for the nonlinear viscous Burger's equation? Note: this is a numerically challenging problem! Try parameters of $\kappa = 0.1$, $v = 1$, with periodic boundary conditions and an initial 'square profile' of traffic bunched together, with a uniform density of $A_0 = 1$. You will need to iterate the nonlinear differential operators to obtain the central derivatives. Can you explain the phenomena that you see?

2. Solve the Schrödinger equation for a Gaussian pulse scattering off a square potential well, with parameters chosen so that there is a large reflection. It is best to start the pulse near a window boundary, moving inwards to a potential well at the center of the window:

$$i \frac{\partial}{\partial t} \psi(x, t) = -\kappa \frac{\partial^2}{\partial x^2} \psi(x, t) + V(x) \psi(x, t). \quad (5.4)$$

Explain the results in terms of resonant behaviour and interference for the waves created inside the potential well.

3. Solve the nonlinear Schrödinger equation which describes solitons in optical fiber pulse propagation, using an initial condition of form $\text{sech}(x)$, where:

$$i\frac{\partial}{\partial t}\psi(x,t) = -\kappa\frac{\partial^2}{\partial x^2}\psi(x,t) - \gamma\psi(x,t)|\psi(x,t)|^2. \quad (5.5)$$

In this example, there is an exact solution of form $\text{sech}(x)\exp(i\alpha t)$, for testing purposes. It is best to start with the exact analytic form in order to test the code. You will have to choose the appropriate κ and γ value so the given input is a soliton, which has an invariant shape. You should check that colliding solitons are unchanged in a collision; moving solitons require a phase modulation of form $\text{sech}(x)\exp(\pm ivx)$.

It will be necessary to either solve a tri-diagonal matrix equation to obtain the required results, or to use the split-step FFT method, which involves periodic boundary conditions. This is especially useful in higher dimensions. Note that the semi-implicit method is much more useful in practise than the first project, especially for nonlinear equations, where you can try larger initial amplitudes (for example, twice or three times the soliton amplitude), as a more stringent test. This is a nontrivial exercise, as there is a nonlinear term to be treated implicitly. Ask for help from a demonstrator, if you have problems with this.

In each case, your report should include:

- A general description of the problem, and the physics background.
- A code that you wrote and tested, with test output and error-plot using an analytically soluble case.
- Table of maximum errors with various step-sizes.
- Plots of physically relevant outputs, with a conclusion about the physics and the numerical method.
- The source code should also be attached to an email to cochrane@physics.uq.edu.au, along with the relevant input parameters used to generate the plots.

Chapter 6

Sample m-file

```
% diffuse1.m - Program to solve the diffusion equation
%                using the FTCS method.

clear;           % clear memory
help diffuse1.m % print help information

%%%%%%%%% INITIALISATION %%%%%%%%%%

% set up input variables
tmax = 1;           % window size in t
nt = 100;          % number of grid points in t
delta_t = tmax/nt; % step-size in t
nplots = 21;       % number of plot points in t
nplotstep = nt/nplots; % plot step-size in t

xmax = 2;          % window size in x
nx = 11;          % number of grid points in x
delta_x = xmax/(nx + 1); % step-size in x

kappa = 1;         % coupling constant
c = kappa*delta_t/delta_x^2; % constant of propagation

% initial conditions and boundary conditions
A = zeros(nx,1); % allocates and zeros a column vector
A(floor((nx+1)/2)) = 1/delta_x; % sets up an initial delta 'spike' in A
iplot = 1;        % initialise the plotting index
```

```

% set up plotting matrices
xplot = (1:nx)*delta_x - xmax/2; % row vector of x-coordinates
tplot(iplot) = 0; % assign first time-coordinate
Aplot(:,iplot) = A(:); % assign first value of A

%%%%%%%%% MAIN LOOP %%%%%%%%%%%%%%

% main loop
for i = 1:nt

    % evolve our solution using the FTCS method
    A(2:(end-1)) = A(2:(end-1)) + c*( A(1:(end-2)) + A(3:end) - 2*A(2:(end-1)) );

    % save current data for plotting
    if (rem(i,nplotstep) < 1)

        iplot = iplot + 1; % increment plotting index
        Aplot(:,iplot) = A(:); % record current value of A for plotting
        tplot(iplot) = i*delta_t; % record current time value

        % print some progress information
        fprintf('%g out of %g steps completed\n',i,nt);

    end

end

%%%%%%%%% PLOTTING/OUTPUT %%%%%%%%%%%%%%

% plot stuff
figure(1) % opens a new figure window
subplot(121) % divides window into a 1x2 matrix
% and assigns next plot to the
% left-hand frame (frame 1)
mesh(tplot,xplot,Aplot) % wire mesh plot of the solution
view([-70 30]) % change the viewing direction
xlabel('t') % label the x-axis
ylabel('x') % label the y-axis
title('Diffusion of a delta spike') % give the thing a title

```

```

subplot(122)                                % assigns next plot to right-hand
                                             % frame (frame 2)

% find contours of 3D plot and assign to the vector cs
cs = contour(xplot,tplot,flipud(rot90(Aplot)),0:0.1:2.5);
xlabel('x')                                  % label the x-axis
ylabel('t')                                  % label the y-axis
clabel(cs,0:5)                               % label the contours
title('Contour of a delta spike')           % give the thing a title

subplot(111)                                % return plotting mode to one
                                             % figure per window

```

6.1 Sample plot

Figures can be exported as `.eps` files from MATLAB, then imported as `.eps` files into a \LaTeX document. For help on how to do this see the \LaTeX help web page: <http://www.physics.uq.edu.au/people/cochrane/phys3071/latexstuff.html>, or ask a tutor.

The graphics format `.eps` (encapsulated postscript) is a proprietary, and very widely used typesetting format of Adobe Corporation, which allows figures to be resized and repositioned anywhere in a page.

The \LaTeX code used to include the graphics into this document was:

```

\begin{figure}
\centerline{%
\includegraphics[width=140mm]{diffuse1}%
}
\caption{Gaussian diffusion of a delta spike. The variable  $x$  is the
spatial dimension and the variable  $t$  is the time dimension, both are
in arbitrary units. The vertical scale represents the height of the
distribution of the variable  $A$  as a function of both  $x$  and  $t$  and
is in arbitrary units.}
\end{figure}

```

The figure used is `diffuse1.eps` but we don't need to use the complete filename since the \LaTeX `graphicx` package can work out what it should be and use the right extension.

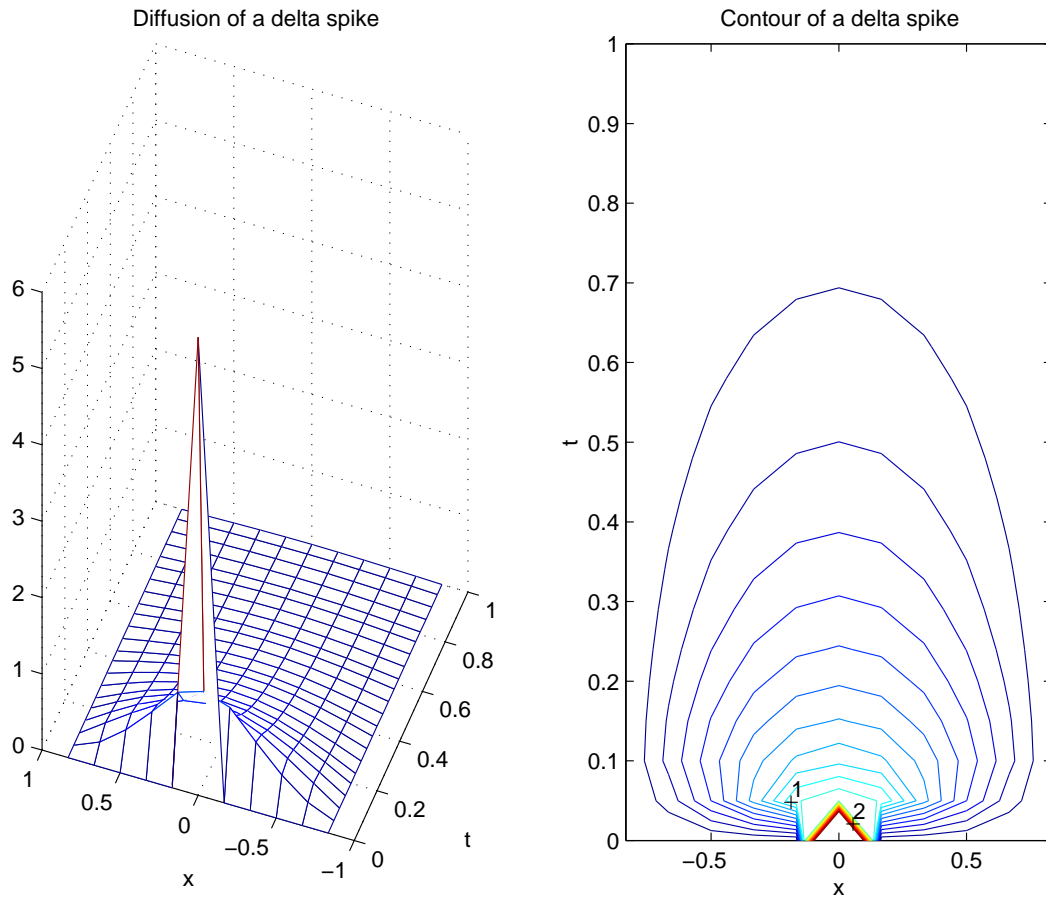


Figure 6.1: Gaussian diffusion of a delta spike. The variable x is the spatial dimension and the variable t is the time dimension, both are in arbitrary units. The vertical scale represents the height of the distribution of the variable A as a function of both x and t and is in arbitrary units.

Appendix A

First Order Numerical Algorithms

These appendices are Professor Drummond's notes on the numerical algorithms discussed in the main part of this text. They are kept here as an alternative discussion to that given above.

How can we treat PDEs numerically?

First of all, it is clear that these PDE problems have many similarities to the coupled first-order differential equations (ODEs) treated earlier in the computational laboratory. Thus, many of the ideas from this field can be used here as well. This leads to the first obvious guess at how to treat this problem numerically, which is the first-order Euler method.

As a general rule, we can develop any method by considering a Taylor series expansion of the spatial and temporal derivatives around a given point, so as to give discrete approximations to the derivatives for numerical integration purposes. Thus, for example, if we divide time into a lattice of t_0, \dots, t_N , with $t_{n+1} = t_n + \Delta t$, then:

$$\mathbf{A}(t_{n+1}, \mathbf{x}) = \mathbf{A}(t_n, \mathbf{x}) + \Delta t \frac{\partial}{\partial t} \mathbf{A}(t_n, \mathbf{x}) + O(\Delta t^2). \quad (\text{A.1})$$

In an explicit method, the propagation is calculated by stepping forward in time; and only using past or present values of the derivatives. Thus, in the Euler method, we simply evaluate the right-hand side of Eq.(A.1) at the initial time $t = t_0$ on a discretised lattice. This gives the derivative $\frac{\partial}{\partial t} \mathbf{A}(t_0, \mathbf{x})$ of the vector field \mathbf{A} . Next, a step is taken in time, by adding $\Delta t \frac{\partial}{\partial t} \mathbf{A}(t_0, \mathbf{x})$ to \mathbf{A} , where we have yet to specify how this derivative is estimated numerically. The process is repeated as many times as necessary.

A.1 Numerical implementation

In practical numerical implementations, another step is required; the differential operator \mathcal{D} must also be approximated on a discrete lattice, since the computer is unable to store a continuously infinite set of initial values of \mathbf{A} at every spatial point.

The overall process can be summarised, by defining a temporal lattice $t_n = (t_0, \dots, t_N)$ together with a spatial lattice $\mathbf{x}_l = (\mathbf{x}_1, \dots, \mathbf{x}_L)$, from $l = 1, \dots, L$, and labeling $\mathbf{A}_l(t) = \mathbf{A}(t, \mathbf{x}_l)$. Generally, it is simplest to use a constant transverse lattice spacing of $\Delta x = x_{l+1} - x_l$. The different components of the vector field \mathbf{A} also need to be labeled explicitly in a numerical approach; we label them with the superscript $i = 1, \dots, I$ if necessary. With this notation, and assuming that U is a purely local function (only depends on the value of \mathbf{A} at the same location in space), we have the EULER method:

$$A_l^i(t_{n+1}) - A_l^i(t_n) = \Delta t \left(\sum_{j,l'} \mathcal{D}_{l,l'}^{i,j} A_{l'}^j(t_n) + U_l^i[\mathbf{A}_l(t_n)] \right), \quad (\text{A.2})$$

Another way to write this equation, which might be clearer, is in a more abstract notation as:

$$\mathbf{A}(t_{n+1}) = [1 + \Delta t \mathbf{D}] \mathbf{A}(t_n) + \Delta t \mathbf{U}[\mathbf{A}(t_n)], \quad (\text{A.3})$$

In this last equation, there is an implied summation over all the vector indices (if they exist) and lattice indices. The process is then repeated N times, with results stored for plotting purposes as often as required.

A.2 FTCS methods

So far, we haven't defined how the spatial derivatives are to be calculated numerically. It is usually most accurate to discretise these in a symmetric way around the current spatial point, essentially through a Taylor expansion around the current point. This leads to a 'Centered Space' discretisation, so that the matrix $\mathcal{D}_{l,m}^{i,j}$ is a spatially symmetric discretised version of the spatial differential operator.

Gradient operator

An example is the discretised spatial differential operator for the scalar gradient, $\partial/\partial x$. First, consider the Taylor expansion:

$$\mathbf{A}(t_n, \mathbf{x}_{l+1}) = \mathbf{A}(t_n, \mathbf{x}_l) + \Delta x \frac{\partial}{\partial x} \mathbf{A}(t_n, \mathbf{x}_l) + \frac{\Delta x^2}{2} \frac{\partial^2}{\partial x^2} \mathbf{A}(t_n, \mathbf{x}_l) + O(\Delta x^3). \quad (\text{A.4})$$

Next, we can write a similar expression for the value of $\mathbf{A}(t_n, \mathbf{x}_{l-1})$:

$$\mathbf{A}(t_n, \mathbf{x}_{l-1}) = \mathbf{A}(t_n, \mathbf{x}_l) - \Delta x \frac{\partial}{\partial x} \mathbf{A}(t_n, \mathbf{x}_l) + \frac{\Delta x^2}{2} \frac{\partial^2}{\partial x^2} \mathbf{A}(t_n, \mathbf{x}_l) + O(\Delta x^3). \quad (\text{A.5})$$

Subtracting these two equations gives the accurate CS discretisation, of form:

$$\frac{\partial}{\partial x} \mathbf{A}(t_n, \mathbf{x}_l) = \frac{1}{2\Delta x} [\mathbf{A}(t_n, \mathbf{x}_{l+1}) - \mathbf{A}(t_n, \mathbf{x}_{l-1})] + O(\Delta x^2). \quad (\text{A.6})$$

If this derivative term was the only spatial derivative, then:

$$\sum_{j,m} \mathcal{D}_{l,m}^{i,j} A_m^j = \frac{1}{2\Delta x} (A_{l+1}^i - A_{l-1}^i) \quad (\text{A.7})$$

Matrix Representation

Suppose we consider a scalar field A , and regard this as a column vector over its spatial indices, defined at each point in time. Then for a 6×6 case, the discretisation matrix \mathbf{D} for the operator $\frac{\partial}{\partial x}$ is:

$$\mathbf{D} = \frac{1}{2\Delta x} \begin{bmatrix} 0 & 1 & & & & \\ -1 & 0 & 1 & & & \\ & -1 & 0 & 1 & & \\ & & -1 & 0 & 1 & \\ & & & -1 & 0 & 1 \\ & & & & -1 & 0 \end{bmatrix} \quad (\text{A.8})$$

Here we assume Dirichlet boundary conditions, with the boundary values set to zero at the boundaries, which are located at $\mathbf{x} = \mathbf{x}_0$ and $\mathbf{x} = \mathbf{x}_{L+1}$. The lattice does not have to include these points, in this case, since the values of the fields at these locations doesn't change.

Diffusion operator

Adding the two equations gives an approximation for the diffusion operator, or second order derivative, of form:

$$\frac{\partial^2}{\partial x^2} \mathbf{A}(t_n, \mathbf{x}_l) = \frac{1}{\Delta x^2} [\mathbf{A}(t_n, \mathbf{x}_{l+1}) - 2\mathbf{A}(t_n, \mathbf{x}_l) + \mathbf{A}(t_n, \mathbf{x}_{l-1})] + O(\Delta x^2). \quad (\text{A.9})$$

Note that the error estimate is proportional to Δx^2 , because it is proportional to fourth order derivative terms. The CS method causes a cancellation to occur in the error terms proportional to third order derivatives, which otherwise would give errors proportional to Δx . If this derivative term was the only spatial derivative, then:

$$\sum_{j,m} \mathcal{D}_{l,m}^{i,j} A_m^j = \frac{1}{\Delta x^2} (A_{l+1}^i - 2A_l^i + A_{l-1}^i) \quad (\text{A.10})$$

The combination of the Euler method in time, with a spatially centered discretisation for the spatial differential operator is called an FTCS algorithm.

Matrix Representation

Suppose we consider a scalar field A , and regard this as a column vector over its spatial indices, defined at each point in time. Then for a 6×6 case, the discretisation matrix \mathbf{D} for the operator $\frac{\partial^2}{\partial x^2}$ is:

$$\mathbf{D} = \frac{1}{\Delta x^2} \begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & 1 & -2 & 1 & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{bmatrix} \quad (\text{A.11})$$

Here we assume Dirichlet boundary conditions, with the boundary values set to zero at the boundaries, which are located at $\mathbf{x} = \mathbf{x}_0$ and $\mathbf{x} = \mathbf{x}_{L+1}$. The lattice does not have to include these points, in this case, since the values of the fields at these locations doesn't change. For periodic boundary conditions, which are often useful for obtaining traveling wave solutions, define:

$$\mathbf{D} = \frac{1}{\Delta x^2} \begin{bmatrix} -2 & 1 & & & & 1 \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & 1 & -2 & 1 & \\ & & & 1 & -2 & 1 \\ 1 & & & & 1 & -2 \end{bmatrix} \quad (\text{A.12})$$

A.3 BTCS Methods

It is also possible to solve ordinary differential equations by calculating the derivatives at a future time point. This inverse Euler method turns out to be often much stabler than conventional, forward Euler methods, especially for so-called 'stiff' differential systems, which have a large range of eigenvalues. Partial differential equations or often 'stiff' equations in this sense, and therefore the inverse Euler method can be useful—although it is clearly more complicated to calculate. In these 'implicit' methods, the propagation is calculated by using future values of the fields in the algorithm. That is, the discretisation includes fields that are presently unknown, and must therefore be solved for numerically as part of the algorithm. While more complex, these techniques are more stable than the explicit schemes.

Thus, in the BTCS or inverse Euler method, we evaluate the right-hand side of Eq.(2.1), at the *final* time t_{n+1} on a discretised lattice. This gives the derivative \mathbf{A}' of the vector field \mathbf{A} . Next, a step is taken in time, by adding $\Delta t \mathbf{A}'$ to \mathbf{A} as usual.

The overall process can be summarized, as:

$$A_l^i(t_{n+1}) - A_l^i(t_n) = \Delta t \left(\sum_{j,m} \mathcal{D}_{l,m}^{i,j} A_m^j(t_{n+1}) + U_l^i[\mathbf{A}(t_{n+1})] \right), \quad (\text{A.13})$$

Here the matrix $\mathcal{D}_{l,m}^{i,j}$ is a discretised version of the spatial differential operator, as before. We can write this schematically for the general case as follows:

$$\mathbf{A}(t_{n+1}) = [1 - \Delta t \mathbf{D}]^{-1} (\mathbf{A}(t_n) + \Delta t \mathbf{U}[\mathbf{A}(t_{n+1})]) \quad (\text{A.14})$$

A practical way to solve this equation is by iteration, since the RHS is not known initially. We can start by estimating $A^{[1]}$, using:

$$\mathbf{A}^{[1]}(t_{n+1}) = [1 - \Delta t \mathbf{D}]^{-1} (\mathbf{A}(t_n) + \Delta t \mathbf{U}[\mathbf{A}(t_n)]) \quad (\text{A.15})$$

Then the estimate can be improved by iterating the process at each time-step, as follows:

$$\mathbf{A}^{[I+1]}(t_{n+1}) = [1 - \Delta t \mathbf{D}]^{-1} (\mathbf{A}(t_n) + \Delta t \mathbf{U}[\mathbf{A}^{[I]}(t_{n+1})]) \quad (\text{A.16})$$

Alternatively, in the case with just differential terms and scalar fields, we must solve the following matrix equation:

$$A_l(t_{n+1}) = [1 - \Delta t \mathcal{D}_{l,m}]^{-1} A_m(t_n) \quad (\text{A.17})$$

This is more stable, but with similar discretisation errors to the previous algorithms.

A.4 Numerical errors

Using any numerical technique to solve differential equations always introduces various numerical errors. It is important to be aware of these errors, and to know how to estimate them, in order to reduce them below the minimum error requirement, or error tolerance, for the numerical problem being solved. Obviously, the error tolerances depend on the physical application.

Errors can usually be reduced by reducing step-sizes, but the rate at which this happens—called the order of the algorithm—depends on the algorithm design. There is generally a trade-off between complexity of code and rate of convergence. Using a rapidly convergent code is not always desirable, if it makes the code so complex that programming errors ('bugs') are introduced! For this reason, we focus on relatively simple methods in this laboratory.

In order to understand error propagation, consider the single-step FTCS description, with the Taylor series (local) errors included:

$$\mathbf{A}(t_{n+1}) = [1 + \Delta t \mathbf{D}] \mathbf{A}(t_n) + \Delta t \mathbf{U}[\mathbf{A}(t_n)] + O(\Delta t^2) + O(\Delta t \Delta x^2). \quad (\text{A.18})$$

Remember that this step must be repeated N times to get the final answer. Clearly N itself depends on Δt , if the total integration time is fixed, and it is the global error (after many steps) that is of most interest to the physicist!

Time-discretisation error

We have assumed (to a first approximation) that the time derivative of the field \mathbf{A} doesn't vary in the small time interval from t_n to t_{n+1} . This is obviously a better approximation when Δt is small. However, it is useful to think about this more quantitatively. From the Taylor expansion, the error for making each step in time is $O(\Delta t^2)$. However, one must take N steps of size Δt to reach a given time t_N , where $N = (t_N - t_0)/\Delta t$. Hence, the accumulated error in reaching a given time t_N is approximately proportional to $\Delta t = \Delta t^2/\Delta t$ in the FTCS methods, provided the error simply adds, and does not grow exponentially. This error can therefore be reduced by halving the time-step, until the changes that occur are less than the error tolerance.

Space-discretisation error

Spatial discretisation introduces further errors. From the Taylor expansion in space, the local error for one step in time due to spatial discretisation is proportional to $\Delta x^2 \Delta t$, which means that the accumulated errors should be approximately proportional to $\Delta x^2 = \Delta x^2 \Delta t / \Delta t$. Again, this can be reduced by halving the step-size in space, until the changes that occur are less than the error tolerance.

Windowing error

Boundary conditions have to be included by careful treatment of the discretised differential operator near the boundaries. This introduces no error if the spatial windows are finite. If the real spatial window is infinite, then the approximation of using a finite spatial window introduces further errors. This can be checked through doubling the window-size, while keeping the step-sizes constant. Note that using mapping techniques also reduces windowing errors. However, mapping does not really eliminate them totally. Instead, it transforms windowing errors into new spatial-discretisation errors—caused by the rapidly increasing spatial variation of the equation terms, as the transformed boundaries in the mapping space are approached. Even so, this method illustrates how mapping techniques can be combined with a numerical method.

Instability errors

It is possible that the global errors behave rather differently to the above estimates. For example, the local errors may cancel each other (relatively unusual, owing to Murphy's law), or they may themselves become amplified by the algorithm. If this happens, then there is an exponential growth in the errors, which can become enormously large very quickly. Under these conditions, the numerical solution is essentially meaningless.

Round-off errors

The final error source is the round-off error, due to the fact that real numbers are represented by binary number approximations in a computer. This effect is complementary to the other error sources, since it accumulates with the total number of floating-point operations, and therefore gets larger when the step-sizes are reduced—or as the spatial window is increased. This effect is often ignored, under the assumption that commonly used double-precision floating-point IEEE operations (64bit, 16-17 decimals) have negligible round-off errors. However, for very accurate calculations with large lattices, this effect can become important. With some computer languages, it is possible to repeat the calculation with quadruple-precision (128bit, 32-34 decimal). If the calculations give results that differ by more than the error tolerance at two different precisions, then round-off is clearly a problem.

A.5 Examples:

A.5.1 Advection equation

The advection equations gives an example of the numerical integration problem that can occur where the algorithm itself can amplify the error. This is called a numerical instability, or divergence. For example, suppose we consider the equation:

$$\frac{\partial}{\partial t}A(x, t) = \mathcal{D}A = \kappa \frac{\partial}{\partial x}A \quad (\text{A.19})$$

In this case, the FTCS method becomes (using Einstein summation convention for summing over the spatial indices m):

$$A_l(t_{n+1}) = [\delta_{l,m} + \Delta t \mathcal{D}_{l,m}]A_m(t_n) = A_l(t_n) + \frac{\kappa \Delta t}{2\Delta x}(A_{l+1}(t_n) - A_{l-1}(t_n)) \quad (\text{A.20})$$

It turns out that the FTCS algorithm is unstable for ALL step-sizes! It is possible to modify the algorithm to stabilize the result, but usually only by including various effects which reduce the accuracy (for example, by introducing a spurious diffusion).

A.5.2 Diffusion equation

Another example is the case of diffusion of a scalar field in one space dimension. Let:

$$\frac{\partial}{\partial t}A(x, t) = \mathcal{D} \cdot A = \kappa \frac{\partial^2}{\partial x^2}A \quad (\text{A.21})$$

The simplest ‘Centered Space’ discretised operator in this case is:

$$[\mathcal{D} \cdot A_l(t_n)] = \sum_m \mathcal{D}_{l,m} A_m(t_n) = \frac{\kappa}{\Delta x^2} (A_{l+1}(t_n) - 2A_l(t_n) + A_{l-1}(t_n)) \quad (\text{A.22})$$

In the abstract notation given above, since there are only derivative terms, the algorithm can be written in an extended matrix form as:

$$\mathbf{A}(t_{n+1}) = [1 + \Delta t \mathbf{D}] \mathbf{A}(t_n), \quad (\text{A.23})$$

Alternatively, to give the full, spatially-indexed ‘FTCS’ algorithm:

$$A_l(t_{n+1}) = \sum_m [\delta_{l,m} + \Delta t \mathcal{D}_{l,m}] A_m(t_n) = A_l(t_n) + \frac{\kappa \Delta t}{\Delta x^2} (A_{l+1}(t_n) - 2A_l(t_n) + A_{l-1}(t_n)) \quad (\text{A.24})$$

The equation above must be modified in the boundary regions, to handle the boundary conditions. In the case of Dirichlet boundary conditions, with the boundary value set to zero, the procedure is simple. Terms in the above sums that have indices with $l = 0$ or $l = L+1$ are set to zero. This means that the matrix \mathbf{D} will only operate on the fields inside the boundary, which do not vanish. We note from the discussion above, that the Euler method has relatively slow convergence, with local errors at least of order Δt^2 , and global errors—after propagating for many steps—of order Δt . However, this is not the worst of it. In addition, there is a well-known instability if we use this method for the diffusion operator. This arises if $\Delta t = t_{n+1} - t_n > \Delta x^2 / 2\kappa$.

If the time-step is larger than this critical value, the FTCS method diverges extremely rapidly.

However, in this case, the BTCS method is stable. This is very useful in treating situations where the step-size or coupling constants are varying over the region of interest, and it might not be very practical to satisfy the critical step-size requirement everywhere!

Appendix B

Higher Order Numerical Methods

B.1 CTCS Methods

To obtain higher accuracy and good stability, the semi-implicit or CTCS method is very useful, which combines both explicit and implicit schemes. We start by writing this method in a generic, abstract form:

$$\mathbf{A}(t_{n+1}) = \mathbf{A}(t_n) + \Delta t[\mathbf{D}\mathbf{A}(t_{n+1/2}) + \mathbf{U}[\mathbf{A}(t_{n+1/2})]] + O(\Delta t^3) + O(\Delta t\Delta x^2). \quad (\text{B.1})$$

From the Taylor expansion, the error due to the time-discretisation, for making each step in time is $O(\Delta t^3)$. Hence, the accumulated error in reaching a given time t_N is approximately proportional to $\Delta t^2 = \Delta t^3/\Delta t$ in the CTCS methods. Thus, a much lower error can be achieved for the same time-step, while at the same time there is usually an increase in stability.

B.2 Crank-Nicolson method

The above algorithm leaves an ambiguity in the practical meaning of how to calculate the derivative evaluated in the central time-location of $t_{n+1/2} = (t_n + t_{n+1})/2$. To clarify this, we could regard this technique as a combination of a FTCS method, then a BTCS method, each with only half the total step-size in time. When the equations only have differential terms, this is straightforward, giving the obvious result that:

$$\mathbf{A}(t_{n+1}) = [1 - \Delta t\mathbf{D}/2]^{-1}[1 + \Delta t\mathbf{D}/2]\mathbf{A}(t_n) \quad (\text{B.2})$$

In one space dimension, this is called the Crank-Nicolson method, where it is easy to implement, since the resulting matrices are tri-diagonal.

B.3 Split-step method

The pure BCTS method is somewhat inefficient to implement when there are nonlinear terms present, as the solution of the nonlinear implicit equations couples all the different spatial points together. For this reason, it is more efficient to implement this in two stages, with an approach called a split-step method. This method still has a local discretisation error proportional to $O(\Delta t^3)$, and can be written in greater detail as:

$$\mathbf{A}(t_{n+1}) = \mathbf{A}(t_n) + \Delta t[\mathbf{D}(\mathbf{A}(t_n) + \mathbf{A}(t_{n+1}))/2 + \mathbf{U}[\mathbf{A}(t_{n+1/2})]] + O(\Delta t^3), \quad (\text{B.3})$$

In this last equation, there is an implied summation over all the vector indices (if they exist) and lattice indices: Thus, the total method can be written in two steps as:

$$\mathbf{A}(t_{n+1/2}) = [1 + \Delta t \mathbf{D}/2] \mathbf{A}(t_n) + (\Delta t/2) \mathbf{U}[\mathbf{A}(t_{n+1/2})] \quad (\text{B.4})$$

$$\mathbf{A}(t_{n+1}) = [1 - \Delta t \mathbf{D}/2]^{-1} (\mathbf{A}(t_{n+1/2}) + (\Delta t/2) \mathbf{U}[\mathbf{A}(t_{n+1/2})]), \quad (\text{B.5})$$

It should be noted that the term $\mathbf{U}[\mathbf{A}(t_{n+1/2})]$ must be calculated implicitly—this can be achieved iteratively, as in the BTCS method treated previously.

B.4 Split-step FFT method

Another method of achieving the goal of stable propagation with high accuracy, is to use the so-called split-step FFT method, in which the differential operator is calculated in Fourier space. This technique involves the use of periodic boundary conditions, but has the advantage that the inverse differential operator is completely trivial in Fourier space!

The Fourier transform on a computer is a DISCRETE Fourier transform, as obviously one cannot Fourier transform over a continuous domain. These have many applications in computational physics, experimental data processing and image processing.

It is an especially useful method in higher dimensions, as inverting a Fourier transform is much faster than inverting a large multi-dimensional matrix. The FFT operator is usually defined for these discrete Fourier transforms, as

$$[\mathcal{F}A]_j = \sum_{l=0}^{L-1} e^{-2\pi i l j / L} A_l \quad (\text{B.6})$$

In the MATLAB implementation (`fft`), where indices start at $l = 1$, the definition is actually:

$$\tilde{A}_j = [\mathcal{F}A]_j = \sum_{l=1}^L e^{-2\pi i (l-1)(j-1)/L} A_l \quad (\text{B.7})$$

Note that the inverse FFT operation may require normalization by the number of lattice sites L , since the inverse FFT operator is defined for these discrete Fourier transforms, as

$$A_j = [\mathcal{F}^{-1}\tilde{A}]_j = \frac{1}{L} \sum_{l=0}^{L-1} e^{2\pi ilj/L} \tilde{A}_l \quad (\text{B.8})$$

The additional normalization is not required in the MATLAB implementation of `ifft` since the normalization is carried out by the inverse Fourier transform subroutine, but is sometimes required in other implementations. The main point is that $[\mathcal{F}^{-1}\mathcal{F}] = 1$. Note, however, that the normalization of the FFT for a discrete Fourier transformation is not the same as the usual convention for a continuous Fourier transform, given earlier. This does not matter, as long as we only plot the results in x -space, not in k -space!

A more subtle point is the indexing convention with FFTs, which typically identifies $k = 0$ (zero momentum) with the first lattice site in reciprocal space, say at $j = 1$ in MATLAB. Positive k values then correspond to $k = \Delta k(j - 1)$, where the definition of Δk is given by examination of the relationship between the discrete approximation to the Fourier transform, and the usual integral form; this leads to the results that: $\Delta k = 2\pi/X$. The first negative value of $k = -\Delta k$ is then obtained by using the $l = L$ site, and reducing the lattice number from its highest value, to give increasingly negative k values! The value of k at $l = L/2$ is ambiguous, and can be given either a positive or negative value.

Thus, the total method for propagating a distance $\Delta t/2$ for the diffusion problem, can be written in just one step, as:

$$\mathbf{A}' = \mathcal{P}[\mathbf{A}(t_n)] \quad (\text{B.9})$$

$$= F^{-1} \left[\mathbf{e}^{-\Delta t \kappa k^2/2} \mathbf{F}[\mathbf{A}(t_n)] \right] \quad (\text{B.10})$$

To use this method more generally with nonlinear terms, requires one Fourier step like this, then a nonlinear step in the x -domain, then a further Fourier step. The total method can then be written in two steps, as:

$$\begin{aligned} \mathbf{A}(t_{n+1/2}) &= \mathcal{P}[\mathbf{A}(t_n)] + \frac{\Delta t}{2} \mathbf{U}[\mathbf{A}(t_{n+1/2})] \\ \mathbf{A}(t_{n+1}) &= \mathcal{P} \left[\mathbf{A}(t_{n+1/2}) + \frac{\Delta t}{2} \mathbf{U}[\mathbf{A}(t_{n+1/2})] \right], \end{aligned} \quad (\text{B.11})$$

Here, in general, the midpoint value of $\mathbf{A}(t_{n+1})$ is to be solved for implicitly, since $\mathbf{A}(t_{n+1})$ is present on both sides of the equation. This can be achieved simply in two or three iterations. In the Schrödinger equation—either linear or nonlinear—iteration isn't needed, since we can write:

$$\mathbf{U}[\mathbf{A}] = i\mathbf{A}\Phi[|\mathbf{A}|]. \quad (\text{B.12})$$

In this case, the evolution under the local term \mathbf{U} —without the dispersion (Laplacian) term—is exact, because \mathbf{U} only affects the phase of \mathbf{A} , and not its amplitude. Thus, one can treat the problem as three exactly soluble stages—i.e., Fourier propagation followed by local time-evolution, then Fourier propagation. A local discretisation error of size Δt^3 per step occurs as previously, because the three stages don't commute with each other:

$$\begin{aligned} \mathbf{A}' &= \mathcal{P}[\mathbf{A}(t_n)] \\ \mathbf{A}(t_{n+1}) &= \mathcal{P} \left[e^{i\Delta t\Phi[|\mathbf{A}'|]} \cdot \mathbf{A}' \right]. \end{aligned} \tag{B.13}$$

There are alternative schemes to this one, based on Runge-Kutta methods, with better convergence properties—although these require more intermediate storage, resulting in caching problems with large lattices. Note that the most efficient way to use the split-step FFT is to only transform to the x-space variables at the mid-points, since this is the only time that x-space values are needed. A potential problem with this method is ‘aliasing’ error, caused by the fact that periodic boundaries result in feed-through from one boundary to the other. This is usually not physically appropriate, though it can be eliminated by including a large absorption term (‘apodisation’) at the boundaries.