Chapter 5

Transport: Non-diffusive, flux conservative initial value problems and how to solve them

Selected Reading

Numerical Recipes, 2nd edition: Chapter 19

A. Staniforth and J. Cote. Semi-Lagrangian integration schemes for atmospheric models - a review, Monthly Weather Review 119, 2206–2223, Sep 1991.

5.1 Introduction

This chapter will consider the physics and solution of the simplest partial differential equations for flux conservative transport such as the continuity equation

$$\frac{\partial \rho}{\partial t} + \boldsymbol{\nabla} \cdot \rho \mathbf{V} = 0 \tag{5.1.1}$$

We will begin by demonstrating the physical implications of these sorts of equations, show that they imply non-diffusive transport and discuss the meaning of the material derivative. We will then demonstrate the relationship between these PDE's and the ODE's of the previous sections and demonstrate *particle methods* of solution. We'll discuss the pros and cons of particle methods and then show how to solve these equations using simple finite difference methods. We will also show that these equations are perhaps the most difficult to solve accurately on a fixed grid. By the time we are done, I hope you will instinctively think transport whenever you see an equation or terms of this form.

5.2 Non-diffusive initial value problems and the material derivative

As a representative problem we will consider conservation of mass for a non-diffusive, stable tracer in one dimension.

$$\frac{\partial \rho c}{\partial t} + \frac{\partial \rho c V}{\partial x} = 0 \tag{5.2.1}$$

Using either equation (5.1.1) or in the special case that ρ and V are constant, (5.2.1) can also be written as

$$\frac{\partial c}{\partial t} + V\frac{\partial c}{\partial x} = 0 \tag{5.2.2}$$

This combination of partial derivatives is known as **the material derivative** and is often written as

$$\frac{D_V}{Dt} \equiv \frac{\partial}{\partial t} + V\frac{\partial}{\partial x} \tag{5.2.3}$$

The material derivative (or Lagrangian derivative) has the physical meaning that it is the time rate of change that would be experienced by a particle traveling along at velocity V. The next two examples will try to show this.

Example 1: Solutions for constant velocity If V is constant in (5.2.2) then it can be shown that the concentration has the general solution that $c(t,x) = f(x - Vt)$ where f is any arbitrary function. To show this let us first define a new variable $\zeta = x - Vt$ and set $c = f(\zeta)$. Therefore by simple substitution and the chain-rule

$$\frac{\partial c}{\partial t} = \frac{df}{d\zeta}\frac{\partial \zeta}{\partial t} = -V\frac{df}{d\zeta} \tag{5.2.4}$$

$$\frac{\partial c}{\partial x} = \frac{df}{d\zeta}\frac{\partial \zeta}{\partial x} = \frac{df}{d\zeta} \tag{5.2.5}$$

Substitution these equations into (5.2.2) shows that it is satisfied identically. But what does it mean? It means that any arbitrary initial condition $f(x_0)$ just propagates to the right at constant speed V. To show this just note that for any constant value of ζ, f remains constant. However a constant value of $\zeta = x_0$ implies that $x = x_0 + Vt$ i.e. the position x simply propagates to the right at speed V.

Example 2: Non-constant Velocity and the method of characteristics It turns out that Eq. (5.2.2) can be solved directly even if V isn't constant because the material derivative applies to a particle in any flow field, not just constant ones. To show this, let us assume that we can write the concentration as

$$c(t,x) = c(t(\tau), x(\tau)) = c(\tau) \tag{5.2.6}$$

where τ is the *local elapsed time* experienced by a particle. Thus the parametric curve $l(\tau) = (t(\tau), x(\tau))$ is the trajectory in space and time that is tracked out by

the particle. If we now ask, what is the change in concentration with τ along the path we get

$$\frac{dc}{d\tau} = \frac{\partial c}{\partial t}\frac{dt}{d\tau} + \frac{\partial c}{\partial x}\frac{dx}{d\tau} \tag{5.2.7}$$

Comparison of (5.2.7) to (5.2.2) shows that (5.2.2) can actually be written as a coupled set of ODE's.

$$\frac{dc}{d\tau} = 0 \tag{5.2.8}$$

$$\frac{dt}{d\tau} = 1 \tag{5.2.9}$$

$$\frac{dx}{d\tau} = V \tag{5.2.10}$$

Which state that the concentration of the particle remains constant along the path, the local time and the global time are equivalent and the change in position of the particle with time is given by the Velocity. The important point is that if $V(x,t)$ is known, Eqs. (5.2.8)–(5.2.10) can be solved with all the sophisticated techniques for solving ODE's. Many times they can be solved analytically. This method is called *the method of characteristics* where the characteristics are the trajectories traced out in time and space.

Particle based methods The previous sections suggest that one manner of solving non-diffusive transport problems is to simply approximate your initial condition as a set of discrete particles and track the position of the particles through time. As long as the interaction between particles does not depend on spatial gradients (e.g. diffusion) this technique is actually very powerful. Figure 5.1 shows the analytic solution for c given a gaussian initial condition and $V = 0.2x$. This method is quite general, can be used for non-linear equations (shock waves) and for problems with source and sink terms such as radioactive decay (as long as the source term is independent of other characteristics). If you just want to track things accurately through a flow field it may actually be the preferred method of solution. However, if there are lots of particles or you have several species who interact with each other, then you will need to interpolate between particles and the method becomes very messy. For these problems you may want to try a *Eulerian* grid-based method. These methods, of course, have their own problems. When we're done discussing all the pitfalls of grid-based advection schemes we will come back and discuss a potentially very powerful hybrid method called *semi-lagrangian schemes* which combine the physics of particle based schemes with the convenience of uniform grids.

5.3 Grid based methods and simple finite differences

The basic idea behind Finite-difference, grid-based methods is to slap a static grid over the solution space and to approximate the partial differentials at each point in the grid. The standard approach for approximating the differentials comes from

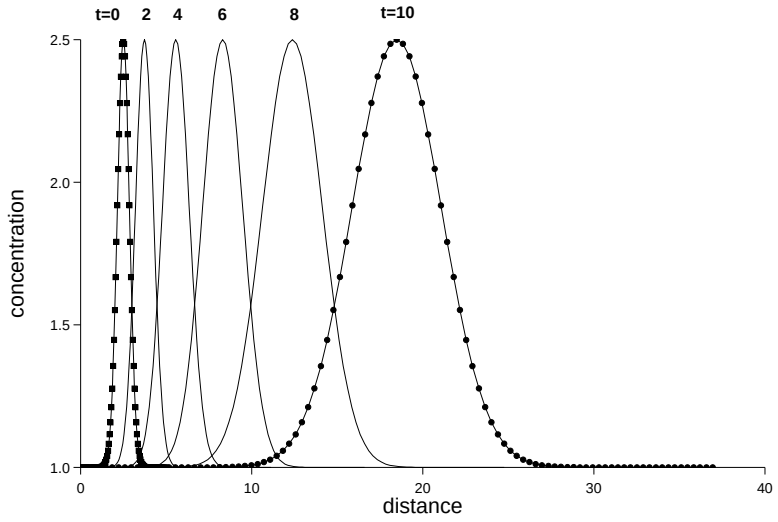

Figure 5.1: Evolution of gaussian initial condition in a velocity field $V = 0.2x$ This velocity field leads to stretching. Analytic solution is by method of characteristics

truncated Taylors series. Consider a function $f(x,t)$ at a fixed time t. If f is continuous in space we can expand it around any point $f(x+\Delta x)$ as

$$f(x+\Delta x) = f(x) + \Delta x \frac{\partial f}{\partial x}(x_0) + \frac{(\Delta x)^2}{2}\frac{\partial^2 f}{\partial x^2}(x_0) + O(\Delta x^3 f_{xxx}) \quad (5.3.1)$$

where the subscripted x imply partial differentiation with respect to x. If we ignore terms in this series of order Δx^2 and higher we could approximate the first derivative at any point x_0 as

$$\frac{\partial f}{\partial x}(x_0) \approx \frac{f(x_0+\Delta x) - f(x_0)}{\Delta x} + O(\Delta x f_{xx}) \quad (5.3.2)$$

If we consider our function is now stored in a discrete array of points f_j and $x = \Delta x j$ where Δx is the grid spacing, then at time step n we can write the *forward space* or FS derivative as

$$\frac{\partial f}{\partial x}(x_0) \approx \frac{f^n_{j+1} - f^n_j}{\Delta x} + O(\Delta x f_{xx}) \quad (5.3.3)$$

An identical procedure but expanding in time gives the forward time derivative (FT) as

$$\frac{\partial f}{\partial t}(t_0) \approx \frac{f^{n+1}_j - f^n_j}{\Delta t} + O(\Delta t f_{tt}) \quad (5.3.4)$$

Both of these approximations however are only *first order accurate* as the leading term in the truncation error is of order Δx or Δt. More importantly, this approximation will only be exact for piecewise linear functions where f_{xx} or $f_{tt} = 0$.

Other combinations and higher order schemes In Eq. (5.3.1) we considered the value of our function at one grid point forward in Δx. We could just have

easily taken a step backwards to get

$$f(x-\Delta x) = f(x) - \Delta x \frac{\partial f}{\partial x}(x_0) + \frac{(\Delta x)^2}{2}\frac{\partial^2 f}{\partial x^2}(x_0) - O(\Delta x^3 f_{xxx}) \quad (5.3.5)$$

If we truncate at order Δx^2 and above we still get a first order approximation for the *Backward space step* (BS)

$$\frac{\partial f}{\partial x}(x_0) \approx \frac{f_j^n - f_{j-1}^n}{\Delta x} - O(\Delta x f_{xx}) \quad (5.3.6)$$

which isn't really any better than the forward step as it has the same order error (but of opposite sign). We can do a fair bit better however if we combine Eqs. (5.3.1) and (5.3.5) to remove the equal but opposite 2nd order terms. If we subtract (5.3.5) from (5.3.1) and rearrange, we can get the *centered space* (CS) approximation

$$\frac{\partial f}{\partial x}(x_0) \approx \frac{f_{j+1}^n - f_{j-1}^n}{2\Delta x} - O(\Delta x^2 f_{xxx}) \quad (5.3.7)$$

Note we have still only used two grid points to approximate the derivative but have gained an order in the truncation error. By including more and more neighboring points, even higher order schemes can be dreamt up (much like the 4th order Runge Kutta ODE scheme), however, the problem of dealing with large patches of points can become bothersome, particularly at boundaries. By the way, we don't have to stop at the first derivative but we can also come up with approximations for the second derivative (which we will need shortly). This time, by adding (5.3.1) and (5.3.5) and rearranging we get

$$\frac{\partial^2 f}{\partial x^2}(x_0) \approx \frac{f_{j+1}^n - 2f_j^n + f_{j-1}^n}{(\Delta x)^2} + O(\Delta x^2 f_{xxxx}) \quad (5.3.8)$$

This form only needs a point and its two nearest neighbours. Note that while the truncation error is of order Δx^2 it is actually a 3rd order scheme because a cubic polynomial would satisfy it exactly (i.e. $f_{xxxx} = 0$). B.P. Leonard [1] has a field day with this one.

5.3.1 Another approach to differencing: polynomial interpolation

In the previous discussion we derived several difference schemes by combining various truncated Taylor series to form an approximation to differentials of different orders. The trick is to combine things in just the right way to knock out various error terms. Unfortunately, this approach is not particularly intuitive and can be hard to sort out for more arbitrary grid schemes or higher order methods. Nevertheless, the end product is simply a weighted combination of the values of our function at neighbouring points. This section will develop a related technique that is essentially identical but it is general and easy to modify.

The important point of these discretizations is that they are all effectively assuming that the our function can be described by a truncated Taylor's series. However, we also know that polynomials of fixed order can also be described exactly

by a truncated Taylor's series. For example a second order quadratic polynomial $f(x) = ax^2 + bx + c$ can be described exactly with a Taylor series that contains only up to second derivatives (all third derivatives and higher are zero). What does this buy us? Fortunately, we also know (thanks to M. Lagrange) that given any N points $y_1 = f(x_1), y_2 = f(x_2), \ldots, y_N = f(x_N)$, there is a unique polynomial of order $N-1$ that passes exactly through those points, i.e.

$$\begin{aligned} P(x) = & \frac{(x-x_2)(x-x_3)\ldots(x-x_N)}{(x_1-x_2)(x_1-x_3)\ldots(x_1-x_N)}y_1 + \\ & \frac{(x-x_1)(x-x_3)\ldots(x-x_N)}{(x_2-x_1)(x_2-x_3)\ldots(x_2-x_N)}y_2 + \ldots \\ & + \frac{(x-x_1)(x-x_2)\ldots(x_-x_{N-1})}{(x_N-x_1)(x_N-x_2)\ldots(x_N-x_{N-1})}y_N \end{aligned} \tag{5.3.9}$$

which for any value of x gives the polynomial interpolation that is simply a weighting of the value of the functions at the N *nodes* $y_{1\ldots N}$. Inspection of Eq. (5.3.9) also shows that $P(x)$ is exactly y_i at $x = x_i$. $P(x)$ is the interpolating polynomial, however, given P(x), all of its derivatives are also easily derived for any x between x_1 and x_N[1] These derivatives however will also be exact weightings of the known values at the nodes. Thus up to $N-1$-th order differences at any point in the interval can be immediately determined.

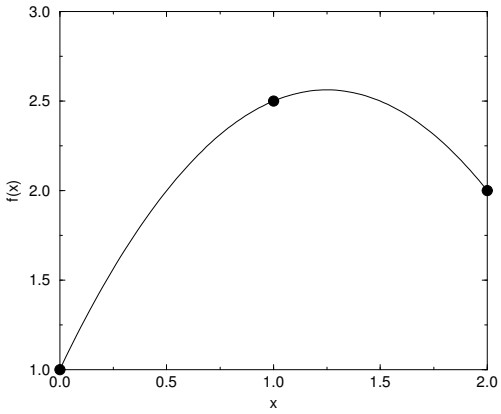

Figure 5.2: The second order interpolating polynomial that passes through the three points $(0\Delta x, 1)$, $(1\Delta x, 2.5)$, $(2\Delta x, 2)$

As an example, Fig. 5.2 shows the 2nd order interpolating polynomial that goes through the three *equally* spaced points $(0\Delta x, 1)$, $(1\Delta x, 2.5)$, $(2\Delta x, 2)$ where

[1]The derivatives and the polynomial are actually defined for all x however, while interpolation is stable, extrapolation beyond the bounds of the known points usually highly inaccurate and is to be discouraged.

$x = i\Delta x$ (note: i need not be an integer). Using Eq. 5.3.9) then yields

$$P(x) = \frac{(i-1)(i-2)}{2} f_0 + \frac{(i-0)(i-2)}{-1} f_1 + \frac{(i-0)(i-1)}{2} f_2 \quad (5.3.10)$$

$$P'(x) = \frac{1}{\Delta x}\left[\frac{(i-1)+(i-2)}{2} f_0 + \frac{i+(i-2)}{-1} f_1 + \frac{i+(i-1)}{2} f_2\right] \quad (5.3.11)$$

$$P''(x) = \frac{1}{\Delta x^2}\left[f_0 - 2f_1 + f_2\right] \quad (5.3.12)$$

where P' and P'' are the first and second derivatives. Thus using Eq. (5.3.11),the first derivative of the polynomial at the center point is

$$P'(x = \Delta x) = \frac{1}{\Delta x}\left[-\frac{1}{2} f_0 + \frac{1}{2} f_2\right] \quad (5.3.13)$$

which is identical to the centered difference given by Eq. (5.3.7). As a shorthand we will often write this sort of weighting scheme as a *stencil* like

$$\frac{\partial f}{\partial x} \approx \frac{1}{\Delta x}\Big[\ -1/2 \quad 0 \quad 1/2\ \Big] f \quad (5.3.14)$$

where a stencil is an operation at a point that involves some number of nearest neighbors. Just for completeness, here are the 2nd order stencils for the first derivative at several points

$$f_x = \frac{1}{\Delta x}\Big[\ -3/2 \quad 2 \quad -1/2\ \Big] \qquad \text{at } x = 0 \quad (5.3.15)$$

$$f_x = \frac{1}{\Delta x}\Big[\ -1 \quad 1 \quad 0\ \Big] \qquad \text{at } x = 1/2\Delta x \quad (5.3.16)$$

$$f_x = \frac{1}{\Delta x}\Big[\ 0 \quad -1 \quad 1\ \Big] \qquad \text{at } x = 3/2\Delta x \quad (5.3.17)$$

$$f_x = \frac{1}{\Delta x}\Big[\ 1/2 \quad -2 \quad 3/2\ \Big] \qquad \text{at } x = 2\Delta x \quad (5.3.18)$$

Note as a check, the weightings of the stencil for any derivative should sum to zero because the derivatives of a constant are zero. The second derivative stencil is always

$$f_{xx} = \frac{1}{\Delta x^2}\Big[\ 1 \quad -2 \quad 1\ \Big] \quad (5.3.19)$$

for all points in the interval because the 2nd derivative of a parabola is constant. Generalizing to higher order schemes just requires using more points to define the stencil. Likewise it is straightforward to work out weighting schemes for unevenly spaced grid points.

5.3.2 Putting it all together

Given all the different approximations for the derivatives, the art of finite-differencing is putting them together in stable and accurate combinations. Actually it's not really an art but common sense given a good idea of what the actual truncation error

is going to do to you. As an example, I will show you a simple, easily coded and totally unstable technique known as *forward-time centered space* or simply the FTCS method. If we consider the canonical 1-D transport equation with constant velocities (5.2.2) and replace the time derivative with a FT approximation and the space derivative as a CS approximation we can write the finite difference approximation as

$$\frac{c_j^{n+1} - c_j^n}{\Delta t} = -V \frac{c_{j+1}^n - c_{j-1}^n}{2\Delta x} \tag{5.3.20}$$

or rearranging for c_j^{n+1} we get the simple updating scheme

$$c_j^{n+1} = c_j^n - \frac{\alpha}{2}\left(c_{j+1}^n - c_{j-1}^n\right) \tag{5.3.21}$$

where

$$\alpha = \frac{V\Delta t}{\Delta x} \tag{5.3.22}$$

is the *Courant number* which is simply the number of grid points traveled in a single time step. Eq. (5.3.21) is a particularly simple scheme and could be coded up in a few f77 lines such as

```
con=-alpha/2.
do i=2,ni-1
   ar1(i)=ar2(i)+con*(ar2(i+1)-ar2(i-1))  ! take ftcs step
enddo
```

(we are ignoring the ends for the moment). The same algorithm using Matlab or f90 array notation could also be written

```
con=-alpha/2.
ar1(2:ni-1)=ar2(2:ni-1)+con*(ar2(3:ni)-ar2(1:ni-2))
```

Unfortunately, this algorithm will almost immediately explode in your face as is shown in Figure 5.3. To understand why, however we need to start doing some *stability analysis.*

5.4 Understanding differencing schemes: stability analysis

This section will present two approaches to understanding the stability and behaviour of simple differencing schemes. The first approach is known as *Hirt's Stability analysis*, the second is *Von Neumann Stability analysis*. Hirt's method is somewhat more physical than Von Neumann's as it concentrates on the effects of the implicit truncation error. However, Von Neumann's method is somewhat more reliable. Neither of these methods are fool proof but they will give us enough insight into the possible forms of error that we can usually come up with some useful rules of thumb for more complex equations. We will begin by demonstrating Hirt's method on the FTCS equations (5.3.21).

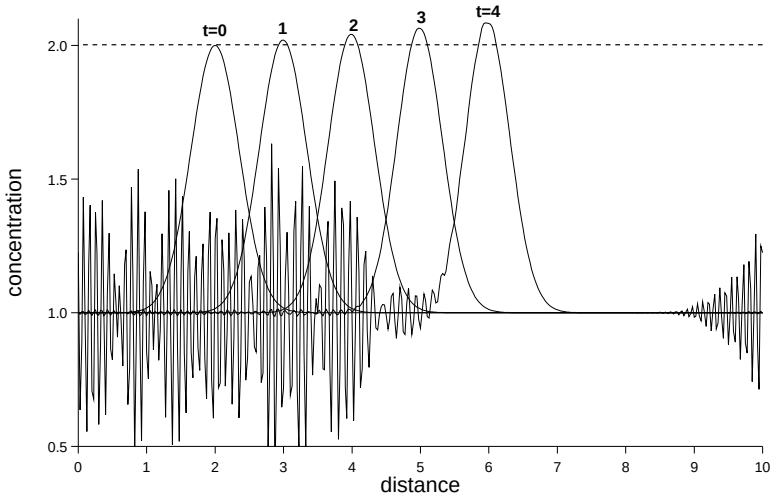

Figure 5.3: Evolution of a gaussian initial condition in a constant velocity field $V = 1$ using a FTCS scheme. A perfect scheme would have the initial gaussian just propagate at constant speed without changing its shape. The FTCS scheme however causes it to grow and eventually for high frequency noise to swamp the solution. By understanding the nature of the truncation error we can show that it is because the FT step has inherent *negative diffusion* which is always unstable.

5.4.1 Hirt's method

Hirt's method can be thought of as reverse Taylor series differencing where we start with finite difference approximation and come up with the actual continuous partial differential equation that is being solved. Given the FTCS scheme in (5.3.21) we first expand each of the terms in a Taylor series about each grid point e.g. we can express the point c_j^{n+1} as

$$c_j^{n+1} = c_j^n + \Delta t \frac{\partial c}{\partial t} + \frac{(\Delta t)^2}{2} \frac{\partial^2 c}{\partial t^2} + O(\Delta t^3 c_{ttt}) \tag{5.4.1}$$

likewise for the spatial points $c_{j\pm1}^n$

$$c_{j+1}^n = c_j^n + \Delta x \frac{\partial c}{\partial x} + (\Delta x)^2 \frac{\partial^2 c}{\partial x^2} + O(\Delta x^3 c_{xxx}) \tag{5.4.2}$$

$$c_{j-1}^n = c_j^n - \Delta x \frac{\partial c}{\partial x} + (\Delta x)^2 \frac{\partial^2 c}{\partial x^2} - O(\Delta x^3 c_{xxx}) \tag{5.4.3}$$

Substituting (5.4.1) and (5.4.2) into (5.3.21) and keeping all the terms up to second derivatives we find that we are actually solving the equation

$$\frac{\partial c}{\partial t} + V \frac{\partial c}{\partial x} = -\frac{\Delta t}{2} \frac{\partial^2 c}{\partial t^2} + O(\Delta x^2 f_{xxx} - \Delta t^2 f_{ttt}) \tag{5.4.4}$$

To make this equation a bit more familiar looking, it is useful to transform the time derivatives into space derivatives. If we take another time derivative of the original

equation (with constant V) we get

$$\frac{\partial^2 c}{\partial t^2} = -V\frac{\partial}{\partial x}\left(\frac{\partial c}{\partial t}\right) \tag{5.4.5}$$

and substituting in the original equation for $\partial c/\partial t$ we get

$$\frac{\partial^2 c}{\partial t^2} = V^2\frac{\partial^2 c}{\partial x^2} \tag{5.4.6}$$

Thus Eq. (5.4.4) becomes

$$\frac{\partial c}{\partial t} + V\frac{\partial c}{\partial x} = -\frac{\Delta t V^2}{2}\frac{\partial^2 c}{\partial x^2} + O(f_{xxx}) \tag{5.4.7}$$

But this is just an advection-diffusion equation with effective diffusivity $\kappa_{eff} = -\Delta t V^2/2$. Unfortunately, for any positive time step $\Delta t > 0$ the diffusivity is negative which is a physical no-no as it means that small perturbations will gather lint with time until they explode (see figure 5.3). This negative diffusion also accounts for why the initial gaussian actually narrows and grows with time. Thus the FTCS scheme is unconditionally unstable.

5.4.2 Von Neumann's method

Von Neumann's method also demonstrates that the FTCS scheme is effectively useless, however, instead of considering the behaviour of the truncated terms we will now consider the behaviour of small sinusoidal errors. Von Neumann stability analysis is closely related to Fourier analysis (and linear stability analysis) and we will begin by positing that the solution at any time t and point x can be written as

$$c(x,t) = e^{\sigma t + ikx} \tag{5.4.8}$$

which is a sinusoidal perturbation of wavenumber k and growth rate σ. If the real part of σ is positive, the perturbation will grow exponentially, if it is negative it will shrink and if it is purely imaginary, the wave will propagate (although it can be dispersive). Now because we are dealing in discrete time and space, we can write $t = n\Delta t$ and $x = j\Delta x$ and rearrange Eq. (5.4.8) for any timestep n and gridpoint j as

$$c_j^n = \zeta^n e^{ik\Delta x j} \tag{5.4.9}$$

where $\zeta = e^{\sigma\Delta t}$ is the amplitude of the perturbation (and can be complex). If $\zeta = x + iy$ is complex, then we can also write ζ in polar coordinates on the complex plane as $\zeta = re^{i\theta}$ where $r = \sqrt{x^2+y^2}$ and $\tan\theta = y/x$. Given $e^{\sigma\Delta t} = re^{i\theta}$ we can take the natural log of both sides to show that

$$\sigma\Delta t = \log r + i\theta \tag{5.4.10}$$

and thus if (5.4.9) is going to remain bounded with time, it is clear that the magnitude of the amplitude $r = \|\zeta\|$ must be less than or equal to 1. Substituting (5.4.9) into (5.3.21) gives

$$\zeta^{n+1}e^{ik\Delta x j} = \zeta^n e^{ik\Delta x j} - \frac{\alpha}{2}\zeta^n\left(e^{ik\Delta x(j+1)} - e^{ik\Delta x(j-1)}\right) \tag{5.4.11}$$

or dividing by $\zeta^n e^{ik\Delta xj}$ and using the identity that $e^{ikx} = \cos(kx) + i\sin(kx)$, (5.4.11) becomes

$$\zeta = 1 - i\alpha \sin k\Delta x \tag{5.4.12}$$

Thus

$$\|\zeta\|^2 = 1 + (\alpha \sin k\Delta x)^2 \tag{5.4.13}$$

Which is greater than 1 for all values of α and $k > 0$ (note $k = 0$ means c is constant which is always a solution but rather boring). Thus, von Neumann's method also shows that the FTCS method is no good for non-diffusive transport equations (it turns out that a bit of diffusion will stabilize things if it is greater than the intrinsic negative diffusion). So how do we come up with a better scheme?

5.5 Usable Eulerian schemes for non-diffusive IVP's

This section will demonstrate the behaviour and stability of several useful schemes that are stable and accurate for non-diffusive initial value problems. While all of these schemes are an enormous improvement over FTCS (i.e. things don't explode), they each will have attendant artifacts and drawbacks (there are no black boxes). However, by choosing a scheme that has the minimum artifacts for the problem of interest you can usually get an effective understanding of your problem. Here are a few standard schemes....

5.5.1 Staggered Leapfrog

The staggered leap frog scheme uses a 2nd ordered centered difference for both the time and space step. i.e. our simplest advection equation (5.2.2) is approximated as

$$\frac{c_j^{n+1} - c_j^{n-1}}{2\Delta t} = -V\frac{c_{j+1}^n - c_{j-1}^n}{2\Delta x} \tag{5.5.1}$$

or as an updating scheme

$$c_j^{n+1} = c_j^{n-1} - \alpha(c_{j+1}^n - c_{j-1}^n) \tag{5.5.2}$$

which superficially resembles the FTCS scheme but is now a two-level scheme where we calculate spatial differences at time n but update from time $n-1$. Thus we need to store even and odd time steps separately. Numerical Recipes gives a good graphical description of the updating scheme and shows how the even and odd grid points (and grids) are decoupled in a "checkerboard" pattern (which can lead to numerical difficulties). This pattern is where the "staggered-leapfrog" gets its name.

Using von Neumann's method we will now investigate the stability of this scheme. Substituting (5.4.9) into (5.5.2) and dividing by $\zeta^n e^{ik\Delta xj}$ we get

$$\zeta = \frac{1}{\zeta} - i2\alpha \sin k\Delta x \tag{5.5.3}$$

or multiplying through by ζ we get the quadratic equation

$$\zeta^2 + i(2\alpha \sin k\Delta x)\zeta - 1 = 0 \tag{5.5.4}$$

which has the solution that

$$\zeta = -i\alpha \sin k\Delta x \pm \sqrt{1 - (\alpha \sin k\Delta x)^2} \tag{5.5.5}$$

The norm of ζ now depends on the size of the Courant number α. Since the maximum value of $\sin k\Delta x = 1$ (which happens for the highest frequency sine wave that can be sampled on the grid) the largest value that α can have before the square root term becomes imaginary is $\alpha = 1$. Thus for $\alpha \leq 1$

$$\|\zeta\| = 1 \tag{5.5.6}$$

which says that for any value of $\alpha \leq 1$. this scheme has no numerical diffusion (which is one of the big advantages of staggered leapfrog). For $\alpha > 1$, however, the larger root of ζ is

$$\|\zeta\| \sim \left[\alpha + \sqrt{\alpha^2 - 1}\right] \tag{5.5.7}$$

which is greater than 1. Therefore the stability requirement is that $\alpha \leq 1$ or

$$\Delta t \|V_{max}\| \leq \Delta x \tag{5.5.8}$$

This result is known as the *Courant condition* and has the physical common-sense interpretation that if you want to maintain accuracy, any particle shouldn't move more than one grid point per time step. Figure 5.4 shows the behaviour of a gaussian initial condition for $\alpha = .9$ and $\alpha = 1.01$

While the staggered-leapfrog scheme is non-diffusive (like our original equation) it can be dispersive at high frequencies and small values of α. If we do a Hirt's stability analysis on this scheme we get an effective differential equation

$$\begin{aligned} \frac{\partial c}{\partial t} + V\frac{\partial c}{\partial x} &= \frac{V}{6}\left(\Delta t^2 V^2 - \Delta x^2\right)\frac{\partial^3 c}{\partial x^3} \\ &= \frac{V\Delta x^2}{6}\left(\alpha^2 - 1\right)\frac{\partial^3 c}{\partial x^3} \end{aligned} \tag{5.5.9}$$

which is dispersive except when $\alpha = 1$. Unfortunately since α is defined for the maximum velocity in the grid, most regions usually (and should) have $\alpha < 1$. For well resolved features and reasonable Courant numbers, the dispersion is small. However, high frequency features and excessively small time steps can lead to more noticeable *wiggles*. Figure 5.5 shows a few examples of dispersion for $\alpha = .5$ and $\alpha = .5$ but for a narrower gaussian. In both runs the gaussian travels around the grid 10 times ($t_{max} = 100$) (for $\alpha = 1$ you can't tell that anything has changed).

For many problems a little bit of dispersion will not be important although the wiggles can be annoying looking on a contour plot. If however the small negative values produced by the wiggles will feed back in dangerous ways into your solution you will need a non-dispersive scheme. The most commonly occuring schemes are known as *upwind schemes*. Before we develop the standard upwind differencing and a iterative improvement on it, however it is useful to develop a slightly different approach to differencing.

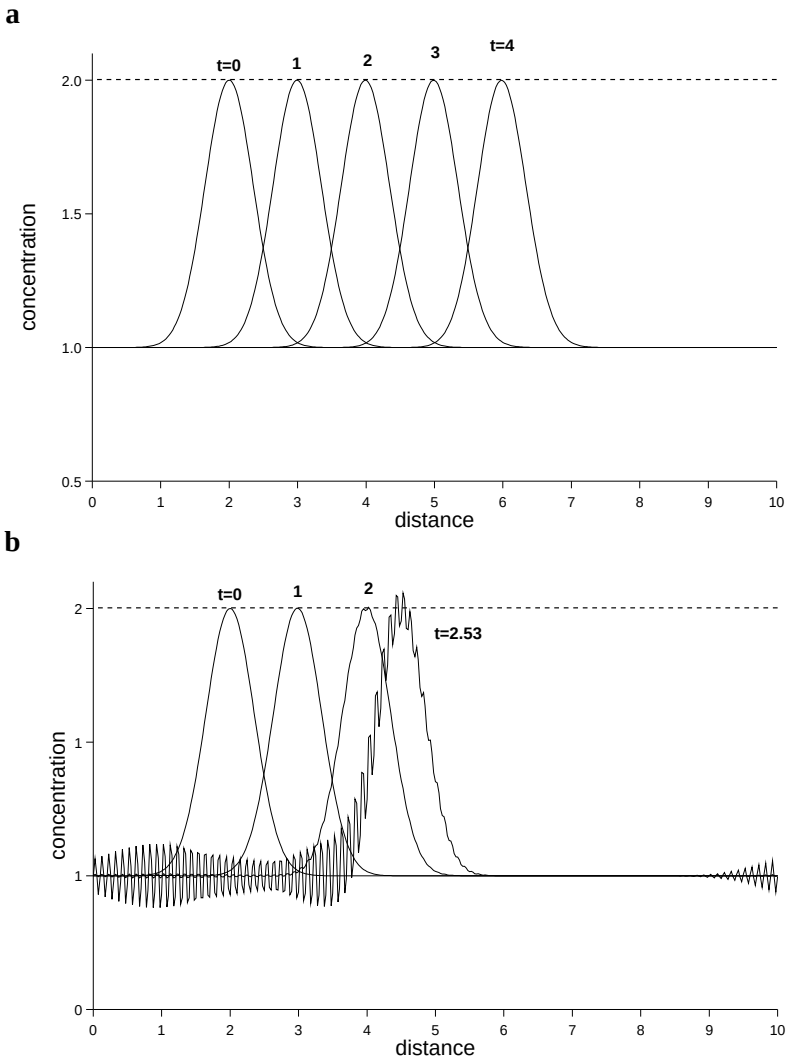

Figure 5.4: (a) Evolution of gaussian initial condition in using a staggered leapfrog scheme with $\alpha = 0.9$. (b) $\alpha = 1.01$ is unstable.

5.5.2 A digression: differencing by the finite volume approach

Previously we developed our differencing schemes by considering Taylor series expansions about a point. In this section, we will develop an alternative approach for deriving difference equations that is similar to the way we developed the original conservation equations. This approach will become useful for deriving the upwind and mpdata schemes described below.

The *control volume* approach divides up space into a number of control volumes of width Δx surrounding each node i.e. and then considers the integral form of the conservation equations

$$\frac{d}{dt}\int_V c dV = -\int_s c\mathbf{V}\cdot d\mathbf{S} \tag{5.5.10}$$

If we now consider that the value of c at the center node of volume j is representa-

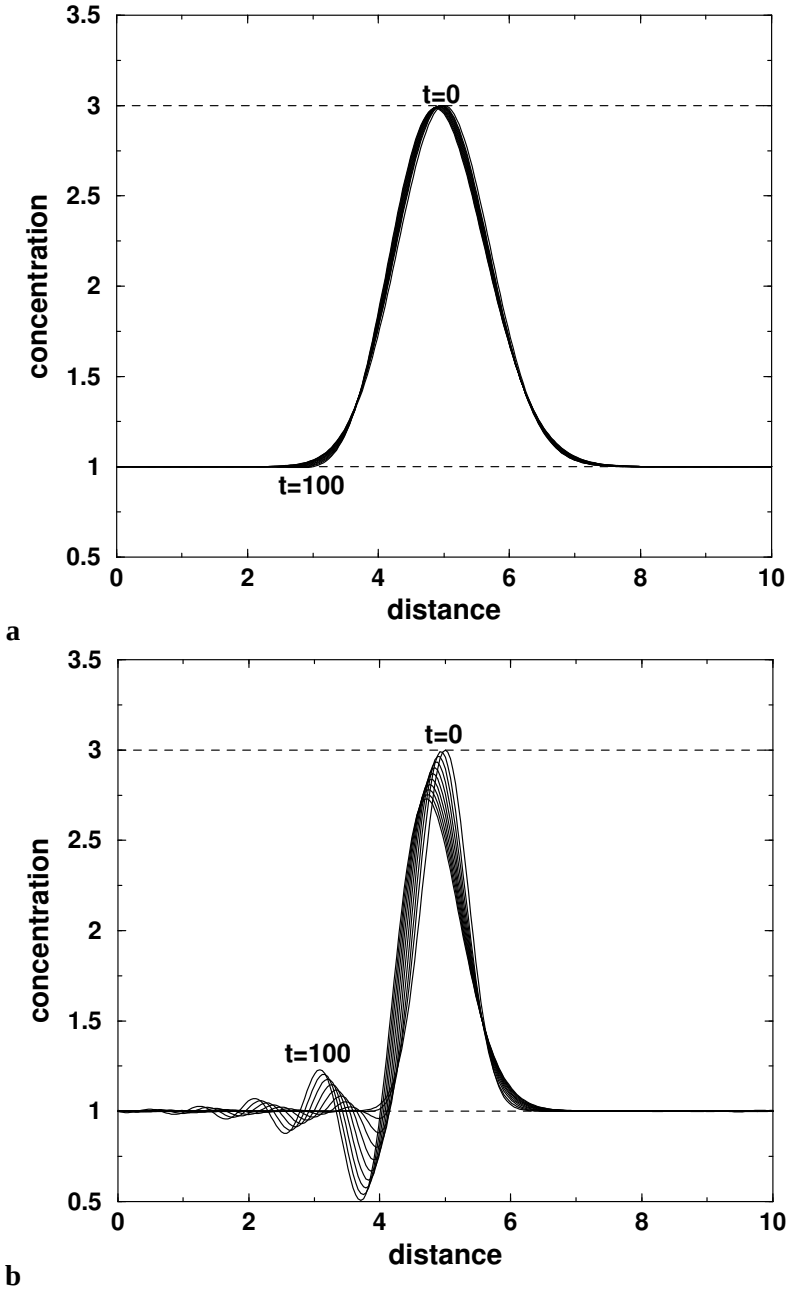

Figure 5.5: (a) Evolution of gaussian initial condition (amplitude 3, width 1., 257 grid points) using a staggered leapfrog scheme with $\alpha = 0.5$. (b) Gaussian is narrower (width=.5) but $\alpha = .5$ For well resolved problems with α close to one however, this is an easy and accurate scheme.

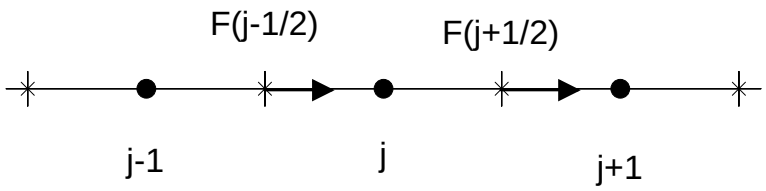

Figure 5.6: A simple staggered grid used to define the control volume approach. Dots denote nodes where average values of the control volume are stored. X's mark control volume boundaries at half grid points.

tive of the average value of the control volume, then we can replace the first integral by $c_j \Delta x$. The second integral is the surface integral of the flux and is exactly

$$\int_s c\mathbf{V} \cdot d\mathbf{S} = c_{j+1/2} V_{j+1/2} - c_{j-1/2} V_{j-1/2} \tag{5.5.11}$$

which is just the difference between the flux at the boundaries $F_{j+1/2}$ and $F_{j-1/2}$. Eq. (5.5.11) is exact up to the approximations made for the values of c and V at the boundaries. If we assume that we can interpolate linearly between nodes then $c_{j+1/2} = (c_{j+1} + c_j)/2$. If we use a centered time step for the time derivative then the flux conservative centered approximation to

$$\frac{\partial c}{\partial t} + \frac{\partial cV}{\partial z} = 0 \tag{5.5.12}$$

is

$$c_j^{n+1} - c_j^{n-1} = -\frac{\Delta t}{\Delta x}\left[V_{j+1/2}(c_{j+1} + c_j) - V_{j-1/2}(c_j + c_{j-1})\right] \tag{5.5.13}$$

or if V is constant Eq. (5.5.13) reduces identically to the staggered leapfrog scheme. By using higher order interpolations for the fluxes at the boundaries additional differencing schemes are readily derived. The principal utility of this sort of differencing scheme is that it is automatically flux conservative as by symmetry what leaves one box must enter the next. The following section will develop a slightly different approach to choosing the fluxes by the direction of transport.

5.5.3 Upwind Differencing (Donor Cell)

The fundamental behaviour of transport equations such as (5.5.13) is that every particle will travel at its own velocity independent of neighboring particles (remember the characteristics), thus physically it might seem more correct to say that if the flux is moving from cell $j-1$ to cell j the incoming flux should only depend on the concentration *upstream*. i.e. for the fluxes shown in Fig. 5.6 the *upwind differencing* for the flux at point $j - 1/2$ should be

$$F_{j-1/2} = \begin{cases} c_{j-1} V_{j-1/2} & V_{j-1/2} > 0 \\ c_j V_{j-1/2} & V_{j-1/2} < 0 \end{cases} \tag{5.5.14}$$

with a similar equation for $F_{j+1/2}$. Thus the concentration of the incoming flux is determined by the concentration of the *donor cell* and thus the name. As a note, the donor cell selection can be coded up without an `if` statement by using the following trick

$$F_{j-1/2}(c_{j-1}, c_j, V_{j-1/2}) = \left[(V_{j-1/2} + \|V_{j-1/2}\|)c_{j-1} + (V_{j-1/2} - \|V_{j-1/2}\|)c_j\right]/2 \tag{5.5.15}$$

or in fortran using `max` and `min` as

```
donor(y1,y2,a)=amax1(0.,a)*y1 + amin1(0.,a)*y2
f(i)=donor(x(i-1),x(i),v(i))
```

Simple upwind donor-cell schemes are stable as long as the Courant condition is met. Unfortunately they are only first order schemes in Δt and Δx and thus the truncation error is second order producing large numerical diffusion (it is this diffusion which stabilizes the scheme). If we do a Hirt's style stability analysis for constant velocities, we find that the actual equations being solved to second order are

$$\frac{\partial c}{\partial t} + V\frac{\partial c}{\partial x} = \frac{\|V\|\Delta x - \Delta t V^2}{2}\frac{\partial^2 c}{\partial x^2} \tag{5.5.16}$$

or in terms of the Courant number

$$\frac{\partial c}{\partial t} + V\frac{\partial c}{\partial x} = (1-\alpha)\frac{\|V\|\Delta x}{2}\frac{\partial^2 c}{\partial x^2} \tag{5.5.17}$$

Thus as long as $\alpha < 1$ this scheme will have positive diffusion and be stable. Unfortunately, any initial feature won't last very long with this much diffusion. Figure (5.7) shows the effects of this scheme on a long run with a gaussian initial condition. The boundary conditions for this problem are periodic (wraparound) and thus every new curve marks another pass around the grid (i.e. after $t = 10$ the peak should return to its original position). A staggered leapfrog solution of this problem would be almost indistinguishable from a single gaussian.

5.5.4 Improved Upwind schemes: mpdata

Donor cell methods in their simplest form are just too diffusive to be used with any confidence for long runs. However a useful modification of this scheme by Smolarkiewicz [2] provides some useful corrections that remove much of the numerical diffusion. The basic idea is that an upwind scheme (with non-constant velocities) is actual solving an advection-diffusion equation of the form

$$\frac{\partial c}{\partial t} + \frac{\partial Vc}{\partial x} = \frac{\partial}{\partial x}\left(\kappa_{impl}\frac{\partial c}{\partial x}\right) \tag{5.5.18}$$

where

$$\kappa_{impl} = .5(\|V\|\Delta x - \Delta t V^2) \tag{5.5.19}$$

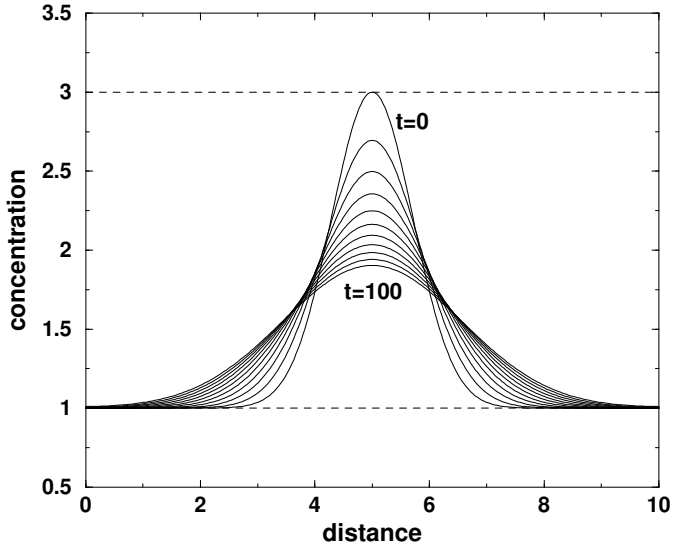

Figure 5.7: Evolution of gaussian initial condition (amplitude 3, width 1., 257 grid points) using an upwind differencing donor-cell algorithm $\alpha = 0.5$. After 10 passes around the grid ($t = 100$) the numerical diffusion has reduced the initial condition to less than half of its amplitude and broadened the peak significantly.

is the implicit numerical diffusivity. One obvious approach (to quote Smolarkiewicz) "is to make the advection step using a [donor cell method] and then reverse the effect of the diffusion equation

$$\frac{\partial c}{\partial t} = \frac{\partial}{\partial x}\left(\kappa_{impl}\frac{\partial c}{\partial x}\right) \tag{5.5.20}$$

in the next corrective step.

The problem is that the diffusion process and the equation that describes it are irreversible. But it is not true that the solution of the diffusion equation cannot be reversed in time. Just as a film showing the diffusion process may be reversed in time, the equivalent numerical trick may be found to produce the same effect. It is enough to notice that (5.5.20) may be written in the form

$$\frac{\partial c}{\partial t} = -\frac{\partial V_d c}{\partial x} \tag{5.5.21}$$

where

$$V_d = \begin{cases} -\frac{\kappa_{impl}}{c}\frac{\partial c}{\partial x} & c > 0 \\ 0 & c = 0 \end{cases} \tag{5.5.22}$$

[this scheme assumes that the advected quantity is always positive]. Here V_d will be referred to as the "diffusion velocity." Now, defining an "anti-diffusion velocity"

$$\tilde{V} = \begin{cases} -V_d & c > 0 \\ 0 & c = 0 \end{cases} \tag{5.5.23}$$

the reversal in time of the diffusion equation may be simulated by the advection equation (5.5.21) with an anti-diffusion velocity $\tilde{V}$. Based on these concepts, the following advection scheme is suggested…".

If we define the donor cell algorithm as

$$c_j^{n+1} = c_j^n - \Big[F_{j+1/2}(c_j, c_{j+1}, V_{j+1/2}) - F_{j-1/2}(c_{j-1}, c_j, V_{j-1/2})\Big] \quad (5.5.24)$$

where F is given by (5.5.15) then the mpdata algorithm is to first take a trial donor-cell step

$$c_j^* = c_j^n - \Big[F_{j+1/2}(c_j^n, c_{j+1}^n, V_{j+1/2}) - F_{j-1/2}(c_{j-1}^n, c_j^n, V_{j-1/2})\Big] \quad (5.5.25)$$

then take a corrective step using the new values and the anti-diffusion velocity $\tilde{V}$, i.e.

$$c_j^{n+1} = c_j^* - \Big[F_{j+1/2}(c_j^*, c_{j+1}^*, \tilde{V}_{j+1/2}) - F_{j-1/2}(c_{j-1}^*, c_j^*, \tilde{V}_{j-1/2})\Big] \quad (5.5.26)$$

This scheme is actually iterative and could be repeated ad nauseum although in practice, any more than 2 additional corrections (3 iterations) is a waste of time. Figure 5.8a,b shows the results of the mpdata algorithm for 1 and 2 corrective steps. Comparison to the standard upwind scheme (which is just one pass of mpdata) in Fig. 5.7 shows the remarkable improvement this approach can deliver. In addition to the anti-diffusive correction, the more recent versions of mpdata also continue the analysis to third order truncation terms and offer an option for an anti-dispersive correction as well. The routine provided in the problem set (`upmpdata1p.f`) implements both the 2nd and 3rd order corrections. This scheme is comparable and computationally cheaper than the most sophisticated *flux corrected transport* schemes, however, it is still much more expensive than the simple staggered-leapfrog scheme. Moreover, given it's computational expense and the fact that it still has a stability requirement given by the courant condition, it is difficult to whole-heartedly recommend this scheme. The principal difficulty is that it seems to be taking most of it's time correcting a really quite poor advection scheme and it would make more sense to find a scheme that better mimics the underlying physics. Fortunately, there are the *semi-lagrangian schemes*.

5.6 Semi-Lagrangian schemes

As the previous sections show, there are really two principal difficulties with Eulerian grid-based schemes. First, they introduce unwanted length scales into the problem because information can propagate from grid-point to grid-point even though the underlying transport equations has no information transfer between characteristics. Usually, this only affects wavelengths that are comparable to the grid-spacing (which are not resolved anyway), however over time these effects can lead to numerical diffusion or dispersion depending on the structure of the truncation error. The second problem with Eulerian schemes is that the Courant condition couples the time step to the spatial resolution, thus to increase the number of grid

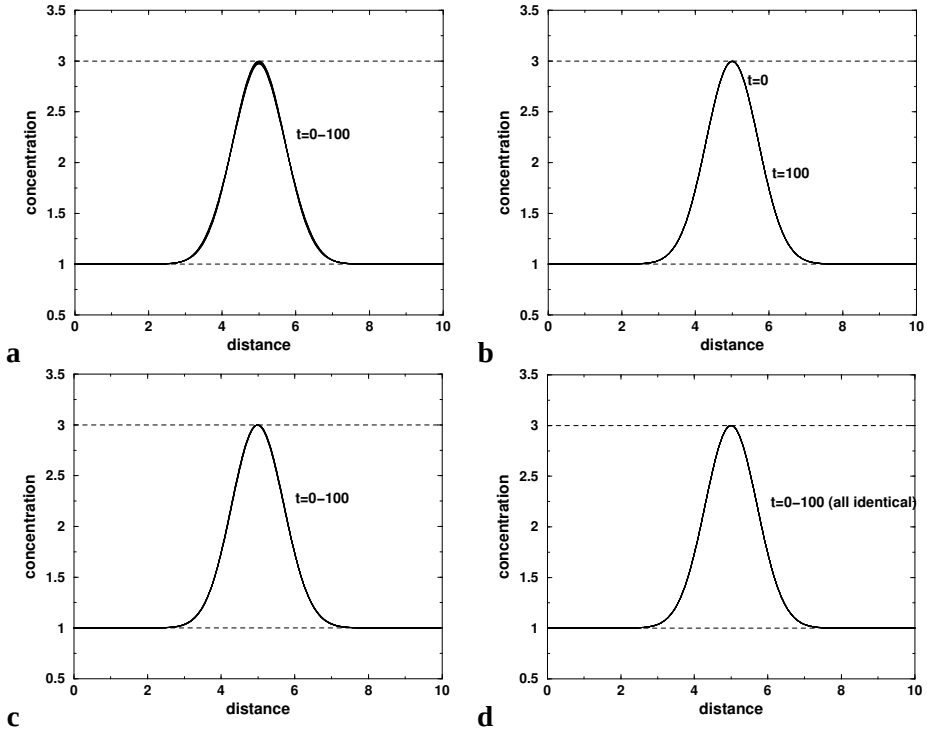

Figure 5.8: Some schemes that actually work. Evolution of gaussian initial condition (amplitude 3, width 1., 257 total grid points) with a variety of updating schemes. All times are for a SunUltra 140e compiled `f77 -fast` *(a)* mpdata with a single upwind correction. $\alpha = 0.5$. (0.49 cpu sec)*(b)* Same problem but with two corrections and a third order anti-dispersion correction (1.26 s) Compare to Fig. 5.7 which is the same scheme but no corrections. Impressive eh? *(c)* two-level pseudo-spectral solution ($\alpha = 0.5$, 256 point iterative scheme with `tol=1.e6` and 3 iterations per time step). Also not bad but deadly slow (9.92 s). (half the grid points (3 s) also works well but has a slight phase shift) But save the best for last *(d)* A semi-lagrangian version of the same problem which is an exact solution, has a Courant number of $\alpha = 2$ and takes only 0.05 cpu sec! (scary eh?)

points by two, increases the total run time by four because we have to take twice as many steps to satisfy the Courant condition. For 1-D problems, this is not really a problem, however in 3-D, doubling the grid leads to a factor of 16 increase in time.

Clearly, in a perfect world we would have an advection scheme that is true to the underlying particle-like physics, has no obvious Courant condition yet still gives us regularly gridded output. Normally I would say we were crazy but the *Semi-Lagrangian schemes* promise just that. They are somewhat more difficult to code and they are not inherently conservative, however, for certain classes of problems they more than make up for it in speed and accuracy. Staniforth and Cote [3] provide a useful overview of these techniques and Figure 5.9 illustrates the basic algorithm.

Given some point c_j^{n+1} we know that there is some particle with concentration $\tilde{c}$ at step n that has a trajectory (or characteristic) that will pass through our grid

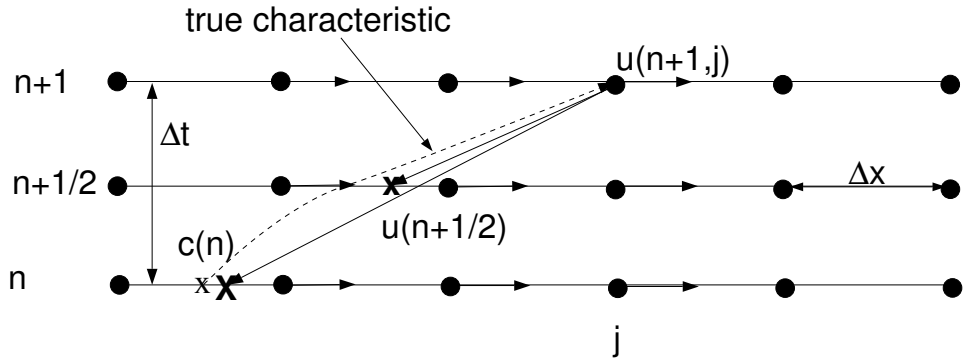

Figure 5.9: Schematic description of the semi-lagrangian advection scheme. x marks the point at time n that lies on the true characteristic that will pass through our grid-point j in time Δt. X is the best approximation for this point found by taking a backwards mid-point step from the future time; To get the concentration at this point we must use some interpolation scheme. If there are no source terms, however, concentration remains constant along a characteristic and $c_j^{n+1} = \tilde{c}^n$. x is the position at time $\Delta t/2$ where we calculate the mid-point velocity.

point c_j^{n+1} in time Δt. The problem is to find this point because the solution to Eq. (5.2.2) is that the concentration remains constant along characteristics i.e. $c_j^{n+1} = \tilde{c}$. The important feature of transport equations, however, is that they can be run backwards in time stably, thus rather than guessing at an old point and moving it forward, we will *start* at our new point $(j, n+1)$ and use our ODE techniques to shoot backwards in time. Fig. 5.9 (and 5.8d) uses a simple mid-point scheme. The basic algorithm can be summarized as follows.

For each point j

1. given the velocity at our future point u_j^{n+1}, find the fractional grid-point x with coordinate $\tilde{j} = j - (\Delta t u_j^{n+1})/(2\Delta x)$ (i.e. take an Euler step back for half a time step).

2. Using some interpolation scheme find the velocity at the half-time step $u_{\tilde{j}}^{n+1/2}$.

3. repeat the step `nvit` times using the new velocity to iterate and find a better approximation to $u_{\tilde{j}}^{n+1/2}$.

4. When you're happy use the centered velocity to find the fractional grid-point X with coordinate $\tilde{j}' = j - (\Delta t u_{\tilde{j}}^{n+1/2})/\Delta x)$

5. Using another interpolation scheme, find the concentration at point $\tilde{j}'$ and copy into c_j^{n+1}

This formulation should be treated as a recipe that is easily varied. For example, here we have used an iterative mid-point scheme, to find the point at time $t - \Delta t$, however, with only bit more work, a 4th order fixed step Runge-Kutta scheme could also be easily employed.

An example of this algorithm in Matlab using a mid-point scheme with first order interpolation for velocity and cubic interpolation for the values can be written

```
it=1;   %  iteration counter (toggles between 1 and 2)
nit=2; % next iteration counter (toggles between 2 and 1)
for n=1:nstep
  t=dt*n;  % set the time
  vm=.5*(v(:,it)+v(:,nit));  % find mean velocity at  mid time
  vi=v(:,nit);  % set initial velocity to the future velocity on the grid points
  for k=1:kit
    xp=x-.5*dt*vi; % get new points
    xp=makeperiodic(xp,xmin,xmax); % patch up boundary conditions
    vi=interp1(x,vm,xp,'linear');  % find centered velocities at mid-time
  end
  xp=x-dt*vi; % get points at minus a whole step;
  xp=makeperiodic(xp,xmin,xmax);  % patch up boundary conditions
  c(:,nit)=interp1(x,c(:,it),xp,'cubic'); %  use cubic interpolation to get the new point
end
```

The function `makeperiodic` implements the periodic boundary conditions by simply mapping points that extend beyond the domain $x_{min} \leq x \leq x_{max}$ back into the domain. Other boundary conditions are easily implemented in 1-D.

These matlab routines demonstrate the algorithm rather cleanly, and make considerable use of the object oriented nature of Matlab. Unfortunately, Matlab is nowhere near as efficient in performance as a compiled language such as fortran. Moreover, if there are many fields that require interpolation, it is often more sensible to calculate the weights separately and update the function point-by-point. The following routines implement the same problem but in f77.

The following subroutines implement this scheme for a problem where the velocities are constant in time. This algorithm uses linear interpolation for the velocities at the half times and cubic polynomial interpolation for the concentration at time step n. This version does cheat a little in that it only calculates the interpolating coefficients once during the run in subroutine `calcintrp`. But this is a smart thing to do if the velocities do not change with time. The results are shown in Fig. 5.8d for constant velocity. For any integer value of the courant condition, this scheme is a perfect scroller. Fig. 5.8d has a courant number of 2, thus every time step shifts the solution around the grid by 2 grid points.

```
c**********************************************************************
c     upsemlag1: 1-D semi-lagrangian updating scheme for updating an
c     array without a source term.  Uses cubic interpolation for the initial
c     value at time -\Delta t   Assumes that all the interpolation
c     weightings have been precomputed using calcintrp1d01 (which
c     calls getweights1d01 in semlagsubs1d.f)
c     arguments are
c           arn(ni): real array for new values
c           aro(ni): real array of old values
c           ni:  full array dimensions
c           wta : array of nterpolation weights for bicubic interpolation at
c                 the n-1   step, precalculated in calcintrp1d01
c           ina : array of coordinates for interpolating 4 points at the n-1
c                 step  precalculated in calcintrp1d01
c           is,ie: bounds of domain for updating
c**********************************************************************

      subroutine upsemlag1(arn,aro,wta,ina,ni,is,ie)
```

```
      implicit none
      integer ni,i,is,ie
      real arn(ni),aro(ni),wta(4,ni)
      integer ina(4,ni)

      do i=is,ie                !    cubic interpolation
         arn(i)=(wta(1,i)*aro(ina(1,i))+wta(2,i)*aro(ina(2,i))+
     &            wta(3,i)*aro(ina(3,i))+wta(4,i)*aro(ina(4,i)))
      enddo
      return
      end

c**********************************************************************
c     calcintrp1d01 routine for calculating interpolation points and
c     coefficients for use in semi-lagrangian schemes.
c     does linear interpolation of velocities and cubic
c     interpolation for end points  for use with upsemlag1
c
c     Version  01:  just calls getweights1d01 where all the heavy
c      lifting is done
c     arguments are
c           ni: full array dimensions
c           u(ni): gridded velocity
c           wta :array of interpolation weights for cubic interpolation at n-1 time point
c           ina: array of coordinates for interpolating 4 points at the n-1 step
c           is,ie  bounds of domain for updating
c           dtdz: dt/dz time step divided by space step (u*dt/dz) is
c           grid points per time step
c           it:  number of iterations to try to find best mid-point velocity
c**********************************************************************

      subroutine calcintrp1d01(wta,ina,u,ni,is,ie,dtdz,it)
      implicit none
      real hlf,sxt
      parameter(hlf=.5, sxt=0.166666666666666667d0)
      integer ni,i,is,ie
      real u(ni)                ! velocity array
      real wta(4,ni)
      real dtdz,hdt
      integer ina(4,ni)
      integer it

      hdt =.5*dtdz

      do i=is,ie
         call getweights1d01(u,wta(1,i),ina(1,i),ni,i,hdt,it)
      enddo
      return
      end

      include 'semilagsubs1d.f'
```

and all the real work is done in the subroutine `getweights1d01`

```
c**********************************************************************
c     Semilagsubs1d:  Basic set of subroutines for doing semilangrangian
c     updating schemes in 1-D.  At the the moment there are only 2
c     routines:
c           getweigths1d01:  finds interpolating weigths and array
c                 indices given a velocity field and a starting index i
c                 version 01 assumes periodic wrap-around boundaries
c           cinterp1d:  uses the weights and indices to return the
c                 interpolated value of any array
c
c     Thus a simple algorithm for semilagrangian updating  might look like
c            do i=is,ie
c               call getweights1d01(wk,wt,in,ni,i,hdtdx,it)
c               arp(i)=cinterp1d(arn,wt,in,ni)
c            enddo
c**********************************************************************

c**********************************************************************
```

```
c     getweights1d01 routine for calculating interpolation points and
c     coefficients for use in semi-lagrangian schemes.
c     does linear interpolation of velocities and cubic
c     interpolation for end points  for use with upsemlag2
c
c     Version  01 calculates full interpolating index and assumes
c     periodic wraparound boundaries
c
c     arguments are
c           ni: full array dimensions
c           u(ni): gridded velocity
c           wt : interpolation weights for bicubic interpolation at the n-1 step
c           in: indices  for interpolating 4 points at the n-1 step
c           is,ie  bounds of domain for updating
c           dtdz: dt/dz time step divided by space step (u*dt/dz) is
c           grid points per time step
c           it:  number of iterations to try to find best mid-point velocity
c**********************************************************************

      subroutine getweights1d01(u,wt,in,ni,i,hdt,it)
      parameter(hlf=.5, sxt=0.16666666666666667d0)
      implicit none
      integer ni,i
      real wt(4),ri,di,u(ni),ui,s
      real hdt
      real t(4)  ! offsets
      integer in(4),k,it,i0,ip
      ui(s,i,ip)=(1.-s)*u(i)+s*u(ip) ! linear interpolation pseudo-func

      di=hdt*u(i)               ! calculate length of trajectory to half-time step
      do k=1,it                 !iterate to get gridpoint at half-time step
         ri=i-di
         if (ri.lt.1) then      ! fix up periodic boundaries
            ri=ri+ni-1
         elseif (ri.gt.ni) then
            ri=ri-ni+1
         endif
         i0=int(ri)             ! i0 is lower interpolating index
         s=ri-i0                !s is difference between ri and i0
         di=hdt*ui(s,i0,i0+1)   !recalculate half trajectory length
      enddo
      ri=i-2.*di-1.             !calculate real position at full time step
      if (ri.lt.1) then         ! fix up periodic boundaries again
         ri=ri+ni-1
      elseif (ri.gt.ni) then
         ri=ri-ni+1
      endif
      in(1)=int(ri)             !set interpolation indices
      do k=2,4
         in(k)=in(k-1)+1
      enddo
      if (in(1).gt.ni-3) then
         do k=2,4               ! should only have to clean up k=2,4
            if (in(k).gt.ni) in(k)=in(k)-ni+1
         enddo
      endif
      t(1)=ri+1.-float(in(1))   !calculate weighted distance from each interpolating point
      do k=2,4
         t(k)=t(k-1)-1.
      enddo
      wt(1)= -sxt*t(2)*t(3)*t(4) ! calculate interpolating coefficients for cubic interpolation
      wt(2)= hlf*t(1)*t(3)*t(4)
      wt(3)=-hlf*t(1)*t(2)*t(4)
      wt(4)= sxt*t(1)*t(2)*t(3)
      return
      end


c**********************************************************************
c     cinterp1d:  do cubic interpolation on an array given a set of
c     weights and indices
c**********************************************************************
```

```
      real function cinterp1d(ar,wt,in,ni)
      implicit none
      integer ni
      real ar(ni)
      real wt(4)
      integer in(4)

      cinterp1d=(wt(1)*ar(in(1))+wt(2)*ar(in(2))+    ! cubic interpolation
     &               wt(3)*ar(in(3))+wt(4)*ar(in(4)))
      return
      end
```

For more interesting problems where the velocities change with time, you will need to recalculate the weights every time step. This can be costly but the overall accuracy and lack of courant condition still makes this my favourite scheme. Here's another version of the code which uses function calls to find the weights and do the interpolation[2].

```
c**********************************************************************
c     upsemlag2: 1-D semi-lagrangian updating scheme for updating an
c     array without a source term.  Uses cubic interpolation for the
c     initial value at time -\Delta t.
c
c     Version 2 assumes velocities are known but changing with time and
c     uses function calls for finding weights and interpolating.  A good
c     compiler should inline these calls in the loop
c
c     Variables:
c           arn(ni): real array of old values (n)
c           arp(ni): real array for new values (n+1)
c           vn(ni): velocities at time n
c           vp(ni): velocities at time n+1
c           wk(ni): array to hold the average velocity .5*(vn+vp)
c           ni:  full array dimensions
c           dx: horizontal grid spacing
c           dt: time step
c           is,ie: bounds of domain for updating
c           it:  number of iterations to find interpolating point
c**********************************************************************

      subroutine upsemlag2(arn,arp,vn,vp,wk,ni,is,ie,dx,dt,it)
      implicit none
      integer ni,i,is,ie
      real arp(ni),arn(ni)
      real vp(ni),vn(ni)
      real wk(ni)
      real dx,dt
      integer it

      real hdt,hdtdx
      real wt(4),in(4) ! weights and indices for interpolation
      real cinterp1d
      external cinterp1d

      hdt=.5*dt
      hdtdx=hdt/dx

      call arrav3(wk,vn,vp,ni) ! get velocity at mid level

      do i=is,ie
         call getweights1d01(wk,wt,in,ni,i,hdtdx,it)
         arp(i)=cinterp1d(arn,wt,in,ni)
      enddo
```

[2]note: a good compiler will inline the subroutine calls for you if you include the subroutines in the file. On Suns running Solaris an appropriate option would be `FFLAGS=-fast -O4`, look at the man pages for more options

```
return
end

include 'semilagsubs1d.f'
```

Using this code, the run shown in Figure 5.8d is about 10 times slower (0.5 seconds) but the courant condition can easily be increased with no loss of accuracy. Moreover, it is still about 50 times faster than the matlab script (as fast as computers are getting, a factor of 50 in speed is still nothing to sneeze at).

5.6.1 Adding source terms

The previous problem of advection in a constant velocity field is a cute demonstration of the semi-lagrangian schemes but is a bit artificial because we already know the answer (which is to exactly scroll the solution around the grid). More interesting problems arise when there is a source term, i.e. we need to solve equations of the form

$$\frac{\partial c}{\partial t} + V\frac{\partial c}{\partial x} = G(x,t) \tag{5.6.1}$$

However, if we remember that the particle approach to solving this is to solve

$$\frac{Dc}{Dt} = G(x,t) \tag{5.6.2}$$

i.e. we simply have to integrate G along the characteristic. Again we can use some of the simple ODE integrator techniques. For a three-level, second order scheme we could just use a mid point step and let

$$c_j^{n+1} = c_{\tilde{j}'}^{n-1} + \Delta t G(\tilde{j}, n) \tag{5.6.3}$$

alternatively it is quite common for G to have a form like $cg(x)$ and it is somewhat more stable to use a trapezoidal integration rule such that, for a two-level scheme

$$c_j^{n+1} = c_{\tilde{j}'}^{n} + \frac{\Delta t}{2}\left[(cg)_{\tilde{j}'}^{n} + (cg)_j^{n+1})\right] \tag{5.6.4}$$

or re-arranging

$$c_j^{n+1} = c_{\tilde{j}'}^{n}\left[\frac{1+\beta g_{\tilde{j}'}^{n-1}}{1-\beta g_j^{n+1}}\right] \tag{5.6.5}$$

where $\beta = \Delta t/2$. Figure 5.6.1 shows a solution of Eq. (5.6.5) for a problem with non-constant velocity i.e.

$$\frac{\partial c}{\partial t} + V\frac{\partial c}{\partial x} = -c\frac{\partial V}{\partial x} \tag{5.6.6}$$

and compares the solutions for staggered-leapfrog, mpdata and a pseudo-spectral technique (next section). Using a Courant number of 10 (!) this scheme is two-orders of magnitude faster and has fewer artifacts than the best mpdata scheme.

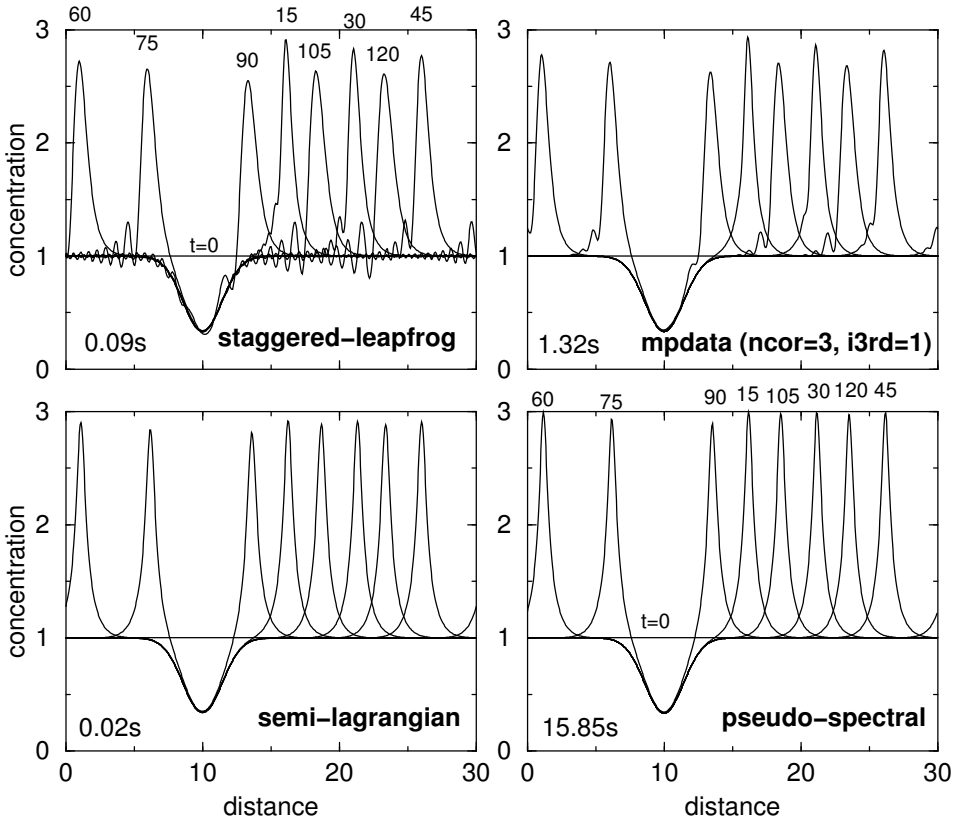

Figure 5.10: Comparison of solutions for advection of a tracer in a non-constant velocity field all runs have 513 (or 512) grid points. Times in figures are for user time (cpu time) on a SunUltra140e compiled with `f77 -fast` (*a*) Staggered-Leapfrog showing significant numerical dispersion with courant number $\alpha = 0.5$. 0.09 seconds. (*b*) 3rd-order 2 corrections mpdata scheme. Lower dispersion but some diffusion (and 15 times slower) $\alpha = 0.5$. (*c*) Semi-Lagrangian scheme, 0.02 seconds $\alpha = 10$. (*d*) iterative Pseudo-spectral scheme. (512 points, $\alpha = .5$) Excellent results but uber-expensive. Once the velocity is not constant in 1-D, advection schemes become much more susceptible to numerical artifacts. Overall, the semi-lagrangian scheme is far superior in speed and accuracy.

5.7 Pseudo-spectral schemes

Finally we will talk about Pseudo-spectral schemes which are more of related to low-order finite difference techniques like staggered-leapfrog than to characteristics approach of semi-lagrangian schemes. For some problems like seismic wave propagation or turbulence the pseudo-spectral techniques can often be quite useful but they are not computationally cheap.

The main feature of pseudo-spectral techniques is that they are *spectrally accurate* in space. Unlike a second order scheme like centered space differencing which uses only two-nearest neighgbors, the pseudo-spectral scheme is effectively infinite order because it uses all the grid points to calculate the derivatives. Normally this

would be extremely expensive because you would have to do N operations with a stencil that is N points wide. However, PS schemes use a trick owing to Fast Fourier Transforms that can do the problem in order $N \log_2 N$ operations (that's why they call them fast).

The basic trick is to note that the discrete Fourier transform of an evenly spaced array of N numbers h_j ($j = 0 \ldots N-1$) is

$$H_n = \Delta \sum_{j=0}^{N-1} h_j e^{ik_n x_j} \tag{5.7.1}$$

where Δ is the grid spacing and

$$k_n = 2\pi \frac{n}{N\Delta} \tag{5.7.2}$$

is the wave number for frequency n. $x_j = j\Delta$ is just the position of point h_j. The useful part of the discrete Fourier Transform is that it is invertable by a similar algorithm, i.e.

$$h_j = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-ik_n x_j} \tag{5.7.3}$$

Moreover, with the magic of the *Fast Fourier Transform* or FFT, all these sums can be done in $N \log_2 N$ operations rather than the more obvious N^2 (see numerical recipes). Unfortunately to get that speed with simple FFT's (like those found in Numerical Recipes [4] requires us to have grids that are only powers of 2 points long (i.e. $2, 4, 8, 16, 32, 64 \ldots$). More general FFT's are available but ugly to code; however, a particularly efficient package of FFT's that can tune themselves to your platform and problem can be found in the FFTW[3] package.

Anyway, you may ask how this is going to help us get high order approximations for $\partial h/\partial x$? Well if we simply take the derivative with respect to x of Eq. (5.7.3) we get

$$\frac{\partial h_j}{\partial x} = \frac{1}{N} \sum_{n=0}^{N-1} -ik_n H_n e^{-ik_n x_j} \tag{5.7.4}$$

but that's just the inverser Fourier Transform of $-ik_n H_n$ which we can write as $F^{-1}[-ik_n H_n]$ where $H_n = F[h_j]$. Therefore the basic pseudo-spectral approach to spatial differencing is to use

$$\frac{\partial c_j}{\partial x} = F^{-1}[-ik_n F[c_j]] \tag{5.7.5}$$

i.e. transform your array into the frequency domain, multiply by $-ik_n$ and then transform back and you will have a full array of derivatives that use *global* information about your array.

In Matlab, such a routine looks like

```
function [dydx] = psdx(y,dx)
% psdx - given an array y, with grid-spacing dx, calculate dydx using fast-fourier transforms
```

[3] Fastest Fourier Transform in the West

```
ni=length(y);
kn=2.*pi/ni/dx;  % 1/ni times the Nyquist frequency
ik=i*kn*[0:ni/2 -(ni/2)+1:-1]';  % calculate ik
dydx=real(ifft(ik.*fft(y)));  % pseudo-spectral first derivative
return;
```

In f77, a subroutine might look something like

```
c**************************************************************************
c     psdx01:  subroutine to calculate df/dx = F^{-1} [ ik F[f]] using
c     pseudo-spectral techniques.  here F is a radix 2 FFT and k is the
c     wave number.
c         routines should work in place, ie on entrance ar=f and on
c         exit  ar=df/dx.   ar is complex(ish)
c**************************************************************************

      subroutine psdx01(ar,ni,dx)
      implicit none
      integer ni
      real ar(2,ni),dx

      integer n
      double precision k,pi,kn,tmp
      real ini
      data pi /3.14159265358979323846/
      ini=1./ni
      kn=2.*pi*ini/dx

      call four1(ar,ni,1)       ! do forward fourier transform
      do n=1,ni/2+1               !for positive frequncies multiply by ik
         k=kn*(n-1)
         tmp=-k*ar(1,n)
         ar(1,n)=k*ar(2,n)
         ar(2,n)=tmp
      enddo
      do n=ni/2+2,ni            !for negative frequncies
         k=kn*(n-ni-1)
         tmp=-k*ar(1,n)
         ar(1,n)=k*ar(2,n)
         ar(2,n)=tmp
      enddo
      call four1(ar,ni,-1)
      call arrmult(ar,2*ni,ini)
      return
      end
```

Note that the array is assumed to be complex and of a length that is a power of 2 so that it can use the simplest FFT from Numerical recipes `four1`.

5.7.1 Time Stepping

Okay, that takes care of the spatial derivatives but we still have to march through time. The Pseudo part of the Pseudo-Spectral methods is just to use standard finite differences to do the time marching and there's the rub. If we remember from our stability analysis, the feature that made the FTCS scheme explode was not the high-order spatial differencing but the low order time-differencing. So we might expect that a FTPS scheme like

$$c_j^{n+1} = c_j^n - \Delta t \mathrm{PS}_x \left[cV\right]^n \tag{5.7.6}$$

is unstable (and it is). Note $\mathrm{PS}_x \left[cV\right]^n$ is the Pseudo-spectral approximation to $\partial cV/\partial x$ at time step n. One solution is to just use a staggered-leapfrog stepper

(which works but has a stability criterion of $\alpha < 1/\pi$ and still is dispersive) or I've been playing about recently with some two-level *predictor-corrector* style update schemes that use

$$c_j^{n+1} = c_j^n - \frac{\Delta t}{2}\left(\mathrm{PS}_x\left[cV\right]^n + \mathrm{PS}_x\left[cV\right]^{n+1}\right) \tag{5.7.7}$$

which is related to a 2nd order Runge-Kutta scheme and uses the average of the current time and the future time. The only problem with this scheme is that you don't know $(cV)^{n+1}$ ahead of time. The basic trick is to start with a guess that $(cV)^{n+1} = (cV)^n$ which makes Eq. (5.7.7) a FTPS scheme, but then use the new version of c^{n+1} to correct for the scheme. Iterate until golden brown.

A snippet of code that implements this is

```
       do n=1,nsteps
         t=dt*n                 ! calculate time
         nn=mod(n-1,2)+1        !set pointer for step n
         nnp=mod(n,2) +1        !set pointer for step n+1
         gnn=gp(nn)                 ! starting index of grid n
         gnp=gp(nnp)               ! starting index of grid n+1
         tit=1
         resav=1.
         do while (resav.gt.tol)
            if (tit.eq.1) call arrcopy(ar(gnp),ar(gnn),npnts)
            call  uppseudos01(ar(gnn),ar(gnp),wp(gnn),wp(gnp),dar,npnts
     &           ,dt,dx,resav)  !update using  pseudo-spec technique
            tit=tit+1
         enddo
       enddo   ! end the loop
```

and the pseudo-spectral scheme that does a single time step is

```
c**************************************************************************
c     uppseudos01:  subroutine to do one centered time update using
c     spatial derivatives calculated by pseudo-spectral operators
c     updating scheme is
c              ddx=d/dx( .5*(arn*wn+arp*wp)
c              arp=arn+dt*ddx
c     returns L2 norm of residual for time step
c**************************************************************************
      subroutine uppseudos01(arn,arp,wn,wp,ddx,npnts,dt,dx,resav)
      implicit none
      integer npnts             !number of 1d grid points
      real arn(npnts),arp(npnts) ! array at time n and n+1
      real wn(npnts),wp(npnts)  ! velocity at time n and n+1
      real ddx(2,npnts) ! complex array for holding derivatives
      real dt,dx
      real res,resav,ressum

      integer n
      real ap

      do n=1,npnts              ! load real component of ddx (and zero im)
         ddx(1,n)=.5*(arn(n)*wn(n)+arp(n)*wp(n))
         ddx(2,n)=0.
      enddo

      call psdx01(ddx,npnts,dx) ! use pseudo spectral techniques to
                                          !   calculate derivative
      ressum=0.
      do n=1,npnts              ! update arp at time n+1 and calculate residual
         ap=arn(n)-dt*ddx(1,n)
         res=ap-arp(n)
         ressum=ressum+res*res
```

```
   arp(n)=ap
enddo
resav=sqrt(ressum)/npnts
return
end
```

The various figures using pseudo-spectral schemes in this chapter use this algorithm. For problems with rapidly changing velocities it seems quite good but it is extremely expensive and it seems somewhat backwards to spend all the energy on getting extremely good spatial derivatives when all the error is introduced in the time derivatives. Most unfortunately, these schemes are still tied to a courant condition and it is usually difficult to implement boundary conditions. Still some people love them. C'est La Guerre.

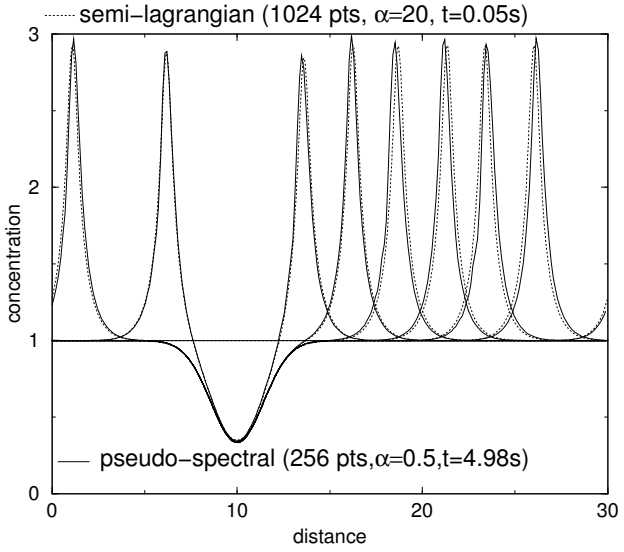

Figure 5.11: Comparison of non-constant velocity advection schemes for Semi-Lagrangian scheme with 1025 points and a Pseudo-Spectral scheme at 256 points. The two schemes are about comparable in accuracy but the semi-lagrangian scheme is nearly 100 times faster.

5.8 Summary

Pure advective problems are perhaps the most difficult problems to solve numerically (at least with Eulerian grid based schemes). The fundamental physics is that any initial condition just behaves as a discrete set of particles that trace out their own trajectories in space and time. If the problems can be solved analytically by characteristics then do it. If the particles can be tracked as a system of ODE's that's probably the second most accurate approach. For grid based methods the semi-lagrangian hybrid schemes seem to have the most promise for real advection problems. While they are somewhat complicated to code and are not conservative,

they are more faithful to the underlying characteristics than Eulerian schemes, and the lack of a courant condition makes them highly attractive for high-resolution problems. If you just need to get going though, a second order staggered leapfrog is the simplest first approach to try. If the numerical dispersion becomes annoying a more expensive mpdata scheme might be used but they are really superseded by the SL schemes. In all cases beware of simple upwind differences if numerical diffusion is not desired.

Bibliography

[1] B. P. Leonard. A survey of finite differences of opinion on numerical muddling of the incomprehensible defective confusion equation, in: T. J. R. Hughes, ed., Finite element methods for convection dominated flows, vol. 34, pp. 1–17, Amer. Soc. Mech. Engrs. (ASME), 1979.

[2] P. K. Smolarkiewicz. A simple positive definite advection scheme with small implicit diffusion, Mon Weather Rev 111, 479–486, 1983.

[3] A. Staniforth and J. Cote. Semi-Lagrangian integration schemes for atmospheric models—A review, Mont. Weather Rev. 119, 2206–2223, Sep 1991.

[4] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling. Numerical Recipes, Cambridge Univ. Press, New York, 2nd edn., 1992.