Chapter 3

Advection algorithms I. The basics

Numerical solutions to (partial) differential equations always require *discretization* of the problem. This means that instead of a continuous space dimension x or time dimension t we now have:

$$x \rightarrow x_i \in \{x_1, \cdots, x_{N_x}\} \tag{3.1}$$
$$t \rightarrow t_n \in \{t_1, \cdots, x_{N_t}\} \tag{3.2}$$

In other words: we have replaced spacetime with a discrete set of points. This is called a *grid* or a *mesh*. The numerical solution is solved on these discrete grid points. So we must replace functions such as $q(x)$ or $q(x,t)$ by their discrete counterparts $q(x_i)$ or $q(x_i,t_n)$. From now on we will write this as:

$$q(x_i,t_n) =: q_i^n \tag{3.3}$$

NOTE: The upper index of q^n is not a powerlaw index, but just the time index. We must now replace the partial differential equation also with a discretized form, with q_i^n as the quantities we wish to solve. In general we with to find q_i^{n+1} for given q_i^n, so the equations must be formulated in the way to yield q_i^{n+1} for given q_i^n. There are infinite ways to formulate the discretized form of the PDEs of hydrodynamics, and some formulations are better than others. In fact, as we shall find out, the simplest formulations tend to be even numerically unstable. And many stable algorithms turn out to be very *diffusive*. The art of numerical hydrodynamics, or of numerical solutions to PDEs in general, is to formulate a discretized form of the PDEs such that the solutions for q_i^n for given initial condition q_i^1 are numerically stable and are as close as possible to the true $q(x_i,t_n)$. Over the past 40 years this has turned out to be a very challenging problem, and even to this day research is on-going to design even better methods than before. One of the main problem is that there is no ideal and universally good method. Some problems require entirely different methods than others, even though they solve exactly the same equations. For example: hydrodynamics of very subsonic flows usually requires different methods than hydrodynamics of superonic flows with shock waves. In the first case we require higher-order precision algorithms while in the second case we need so-called shock-capturing methods. We will discuss such methods in later chapters.

In this chapter we will focus our attention to the fundamental and easy to formulate problem of *numerical advection on a grid*. As we have seen in Chapter 2, the equations of hydrodynamics can be reduced to signals propagating with three different speeds: the two sound waves and the gas motion. These 'signals' are nothing else than the eigenvectors of the Jacobian matrix, and are therefore combinations of ρ, ρu and $\rho e_{\rm tot}$, or in other words the eigenvector decomposition

coefficients $(\tilde{q}_1, \tilde{q}_2, \tilde{q}_3)$. So it should be possible to formulate numerical hydrodynamics as a numerical advection of these signals over a grid.

To simplify things we will not focus on the full set of signals. Instead we focus entirely on how a *scalar* function $q(x,t)$ can be numerically advected over a grid. The equation is simply:

$$\partial_t q(x,t) + \partial_x[q(x,t)u(x,t)] = 0 \tag{3.4}$$

which is the conserved advection equation. This problem sounds nearly trivial, but it is far from trivial in practice. In fact, finding a proper algorithm for numerical advection of scalar functions over a grid has been one of the main challenges for numerical hydrodynamics in the early years of hydrodynamics. We will in fact reduce the complexity of the problem even further and study simply:

$$\partial_t q(x,t) + u\partial_x[q(x,t)] = 0 \tag{3.5}$$

with u is a constant. In this case the equation is automatically a conservation equation in spite of the fact that u is outside of the differential operator.

Since not all readers may be familiar with numerical methods, we will start this chapter with a recapitulation of some basic methods for the integration of functions.

3.1 *Prelude:* Numerical integration of an ordinary differential equation

In spite of its simplicity, the advection equation is a 2-D problem (in x,t) which therefore already naturally has some level of complexity. To introduce some of the basic concepts that we can use later, we first turn our attention to a simple problem: the solution to an ordinary differential equation (ODE) such as

$$\frac{dq(t)}{dt} = F(t, q(t)) \tag{3.6}$$

where $F(t,q)$ is a function of both t and q. We assume that at some time $t = t_0$ we know the value of q and we wish to integrate this equation in time t using a numerical method. To do this we must discretize time t in discrete steps t_0, t_1, t_2 etc, and the values of $q(t)$ belonging to these time nodes can be denoted as q_0, q_1, q_2 etc. So, given q_0, how can we compute q_i with $i > 0$? The most straightforward method is the *forward Euler method*:

$$\frac{q_{n+1} - q_n}{\Delta t} = F(t_n, q_n) \tag{3.7}$$

which can be written as an expression for q_{n+1}:

$$q_{n+1} = q_n + F(t_n, q_n)\Delta t \tag{3.8}$$

This method is also called *explicit integration*, because the new value of q is explicitly given in terms of the old values. This is the easiest method, but it has several drawbacks. One of these drawbacks is that it becomes numerically unstable if the time step Δt is taken too large.

→ **Exercise:** Consider the equation

$$\frac{dq(t)}{dt} = -q(t)^2 \tag{3.9}$$

Write down the analytic solution (to later compare with). Assume $q(t = 0) = 1$ and numerically integrate this equation using the forward Euler method to time $t = 10$. Plot

the numerical resulting function $q(t)$ over the analytic solution. Experiment with different time steps Δt, and find out what happens when Δt is taken too large. Derive a reasonable condition for the maximum Δt that can be allowed. Find out which Δt is needed to get reasonable accuracy.

A way to stabilize the integration even for very large time steps is to use the *backward Euler method*:

$$q_{n+1} = q_n + F(t_{n+1}, q_{n+1})\Delta t \tag{3.10}$$

which is also often called *implicit integration*. This equation may seem like magic, because to calculate q_{n+1} we need q_{n+1} itself! The trick here is that one can often manipulate the equation such that in the end q_{n+1} is written in explicit form again. For instance:

$$\frac{dq(t)}{dt} = -q \tag{3.11}$$

discretizes implicitly to

$$q_{n+1} = q_n - q_{n+1}\Delta t \tag{3.12}$$

While this is an implicit equation, one can rewrite it to:

$$q_{n+1} = \frac{q_n}{1 + \Delta t} \tag{3.13}$$

which is stable for all $\Delta t > 0$. However, in many cases $F(t, q)$ is non-linear, and this simple manipulation is not possible. In that case one is either forced to linearize the equations about the current value of q_n and perform the manipulations with $\delta q_{n+1/2} \equiv q_{n+1} - q_n$, or one uses iteration, in which a first guess is made for q_{n+1} and this is re-computed iteratively until convergence. The latter method is, however, rather time-consuming.

Whether implicit methods produce accurate results for large Δt is another issue. In fact, such implicit integration, while being numerically extraordinarily stable, is about as inaccurate as explicit integration.

An alternative method, that combines the ideas of both forward and backward Euler integration is the *midpoint rule*:

$$q_{n+1} = q_n + F(t_{n+1/2}, q_{n+1/2})\Delta t \tag{3.14}$$

where $n + 1/2$ stands for the position between t_n and t_{n+1}. Here the problem is that we do not know $q_{n+1/2}$, neither currently, nor once we know q_{n+1}. For problems of integrating Hamiltonian systems this method can nevertheless work and turns out to be very useful. This is because the 'coordinates' q_i are located at t_n and the conjugate momenta p_i are located at $t_{n+1/2}$, and their time derivatives only depend on their conjugate (q on p and p on q). This forms the basis of *symplectic integrators* such as the leapfrog method.

An integration method very akin to the midpoint rule, but more readily applicable is the *trapezoid rule*:

$$q_{n+1} = q_n + \frac{1}{2}[F(t_n, q_n) + F(t_{n+1}, q_{n+1})]\Delta t \tag{3.15}$$

It is half-implicit. For the above simple example (Eq. 3.11) we can thus write:

$$q_{n+1} = q_n - \frac{1}{2}q_n\Delta t - \frac{1}{2}q_{n+1}\Delta t \tag{3.16}$$

which results in:

$$q_{n+1} = \frac{1-\Delta t/2}{1+\Delta t/2} q_n \tag{3.17}$$

This method, when generalized to multiple dimensional partial differential equations, is called the *Crank-Nicholson* method. It has the advantage that it is usually numerically stable (though not as stable as fully implicit methods) and since the midpoint is used, it is also naturally more accurate. A variant of this method is the *predictor-corrector* method (or better: the most well-known of the predictor-corrector methods) in which a temporate prediction of q_{n+1} is made with the explicit forward Euler method, which is then used as the q_{n+1} in the trapezoid rule.

So far we have always calculated q_{n+1} on the basis of the known q_n (and, in case of implicit schemes, on q_{n+1} as well). Such methods are either first order accurate (such as the forward and backward Euler methods) or second order accurate (such as the trapezoid rule), but they can never be of higher order. However, there exist higher-order methods for integration. They either make predictor-steps in between t_n and t_{n+1} (these are the Runge-Kutta type methods) or they fit a Lagrange polynomial through q_n, q_{n-1} (or even q_{n-2} and further) to compute the integral of the ODE in the interval $[t_n, t_{n+1}]$ for finding q_{n+1} (these are Adams-Bashforth methods). The first order Adams method is equal to the forward Euler method. The second and third order ones are:

$$q_{n+1} = q_n + \frac{\Delta t}{2}\left[3F(t_n,q_n) - F(t_{n-1},q_{n-1})\right] \tag{3.18}$$

$$q_{n+1} = q_n + \frac{\Delta t}{12}\left[23F(t_n,q_n) - 16F(t_{n-1},q_{n-1}) + 5F(t_{n-2},q_{n-2})\right] \tag{3.19}$$

However, when generalized to multi-dimensional systems (such as hydrodynamics equations) such higher order multi-point schemes in time are not very often used. In astrophysics the `PENCIL` hydrodynamics code is a code that uses higher order Runge-Kutta type integration in time, but many astrophysical codes are first or (more often) second order in time.

3.2 Numerical spatial derivatives

3.2.1 Second order expressions for numerical spatial derivatives

To convert the PDEs of Eqs. (3.4,3.5) into a discrete form we need to formulate the derivatives in discrete form. A derivative is defined as:

$$\frac{\partial q}{\partial x} = \lim_{\Delta x \to 0} \frac{q(x+\Delta x) - q(x)}{\Delta x} \tag{3.20}$$

For Δx we could take $x_{i+1} - x_i$, but in a numerical calculation we cannot do the $\lim_{\Delta x \to 0}$ because it would require infinite number of grid points. So the best we can do is write:

$$\left.\frac{\partial q}{\partial x}\right|_{i+1/2} = \frac{q_{i+1}-q_i}{x_{i+1}-x_i} + \mathcal{O}(\Delta x^2) \simeq \frac{q_{i+1}-q_i}{x_{i+1}-x_i} \tag{3.21}$$

where $\Delta x \equiv (x_{i+1} - x_i)$, and we assume for the moment that the grid is constantly spaced. Note that this is an approximate expression of the derivative defined *in between* the grid points x_{i+1} and x_i. For this reason we denote this as the derivative at $i + 1/2$, which is just a way to index locations in-between grid points. Often one needs the derivative not in between two grid points $(i + 1/2)$, but precisely at a grid point (i). This can be written as:

$$\left.\frac{\partial q}{\partial x}\right|_{i} = \frac{q_{i+1}-q_{i-1}}{x_{i+1}-x_{i-1}} + \mathcal{O}(\Delta x^2) \simeq \frac{q_{i+1}-q_{i-1}}{x_{i+1}-x_{i-1}} \tag{3.22}$$

So depending where we need the derivative, we have different expressions for them.

The $\mathcal{O}(\Delta x^2)$ in Eqs. (3.21, 3.22) is a way of writing the deviation between the 'true' derivative and the approximation. Of course, the true derivative is only defined as long as $q(x)$ is smoothly defined. In our numerical algorithm we do not have $q(x)$: we only have q_i, and hence the 'true' derivative is not defined. But it does show that if we re-express the whole problem on a finer grid, i.e. with smaller Δx, then the approximate answer approaches the true one as Δx^2.

To see this, we assume we know the smooth function $q(x)$. We say that $q_i = q(x_i)$ and $q_{i+1} = q(x_{i+1}) = q(x_i + \Delta x)$ and we express $q(x_i + \Delta x)$ as a Taylor series:

$$q(x_i + \Delta x) = q(x_i) + q'(x_i)\Delta x + \frac{1}{2}q''(x_i)\Delta x^2 + \frac{1}{6}q'''(x_i)\Delta x^4 + \mathcal{O}(\Delta x^4) \tag{3.23}$$

where the $'$ (prime) denotes the derivative to x. Then we insert this into the numerical derivative at i:

$$\begin{aligned}\frac{q_{i+1} - q_{i-1}}{2\Delta x} &= \frac{q_i + q_i'\Delta x + \frac{1}{2}q_i''\Delta x^2 + \frac{1}{6}q_i'''\Delta x^3 + \cdots - q_i + q_i'\Delta x - \frac{1}{2}q_i''\Delta x^2 + \frac{1}{6}q_i'''\Delta x^3 + \cdots}{2\Delta x} \\ &= q_i' + \frac{1}{6}q_i'''\Delta x^2 + \cdots\end{aligned} \tag{3.24}$$

This shows that the deviations are of order $\mathcal{O}(\Delta x^2)$.

3.2.2 Higher-order expressions for numerical derivatives

The $\mathcal{O}(\Delta x^2)$ also shows that there must be various other expressions possible for the derivative that are equally valid. Eq. 3.21 is an example of a *two-point* derivative at position $i + 1/2$. Eq. 3.22 is also a two-point derivative, this time at position i, but it is defined on a *three-point stencil*. A *stencil* around point i is a mini-grid of points that contribute to the things we wish to evaluate at i. We can also define a derivative at i on a *five-point stencil*:

$$\left.\frac{\partial q}{\partial x}\right|_i = \frac{-q_{i+2} + 8q_{i+1} - 8q_{i-1} + q_{i-2}}{12\Delta x} + \mathcal{O}(\Delta x^4) \tag{3.25}$$

Note: this expression is only valid for a constantly-spaced grid!

→ **Exercise:** Show that this expression indeed reproduces the derivative to order $\mathcal{O}(\Delta x^4)$.

Some hydrodynamics codes are based on such higher-order numerical derivatives. Colloquially it is usually said that higher-order schemes are more accurate than lower-order schemes. However, this is only true if the function $q(x)$ is reasonably smooth over length scales of order Δx. In other words: the $\mathcal{O}(\Delta x^4)$ is only significantly smaller than $\mathcal{O}(\Delta x^2)$ if $\partial_x^5 q(x)\Delta x^4 \ll \partial_x^3 q(x)\Delta x^2 \ll \partial_x q(x)$. Higher-order schemes are therefore useful for flows that have no strong discontinuities in them. This is often true for *subsonic* flows, i.e. flows for which the sound speed is much larger than the typical flow speeds. For problems involving shock waves and other types of discontinuities such higher-order schemes turn out to be worse than lower order ones, as we will show below.

3.3 Some first advection algorithms

In this section we will try out a few algorithms and find out what properties they have. We focus again on the advection equation

$$\partial_t q + u\partial_x q = 0 \tag{3.26}$$

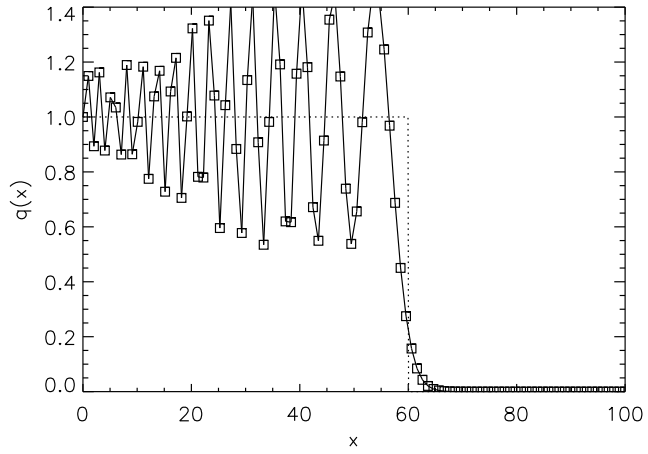

Figure 3.1. Result of center-difference algorithm for advection of a step-function from left to right. Solid line and symbols: numerical result. Dotten line: true answer (produced analytically).

with constant $u > 0$. The domain of interest is $[x_0, x_1]$. We assume that the initial state $q(x, t = t_0)$ is given on this domain, and we wish to solve for $q(x, t > t_0)$. A boundary condition has to be specified at $x = x_0$.

3.3.1 Centered-differencing scheme

The simplest discretization of the equation is:

$$\frac{q_i^{n+1} - q_i^n}{t_{n+1} - t_n} + u\frac{q_{i+1}^n - q_{i-1}^n}{x_{i+1} - x_{i-1}} = 0 \tag{3.27}$$

in which we use the $n + 1/2$ derivative in the time direction and the i derivative in space. Let us assume that the space grid is equally-spaced so that we can always write $x_{i+1} - x_{i-1} = 2\Delta x$. We can then rewrite the above equation as:

$$q_i^{n+1} = q_i^n - \frac{\Delta t}{2\Delta x}u(q_{i+1}^n - q_{i-1}^n) \tag{3.28}$$

This is one of the simplest advection algorithms possible.

Let us test it by applying it to a simple example problem. We take $x_0 = 0, x_1 = 100$ and:

$$q(x, t = t_0) = \begin{cases} 1 & \text{for} \quad x \leq 30 \\ 0 & \text{for} \quad x > 30 \end{cases} \tag{3.29}$$

As boundary condition at $x = 0$ we set $q(x = 0, t) = 1$. Let us use an x-grid with spacing $\Delta x = 1$, i.e. we have 100 grid points located at $i + 1/2$ for $i \in [0, 99]$. Let us choose $\Delta t \equiv t_{n+1} - t_n$ to be $\Delta t = 0.1\Delta x/u$ for now. If we do 300 time steps, then we expect the jump in q to be located at $x = 60$. In Fig. 3.1 we see the result that the numerical algorithm produces.

One can see that this algorithm is numerically unstable. It produces strong oscillations in the downsteam region. For larger Δt these oscillations become even stronger. For smaller Δt they may become weaker, but they are always present. Clearly this algorithm is of no use. We should find a better method

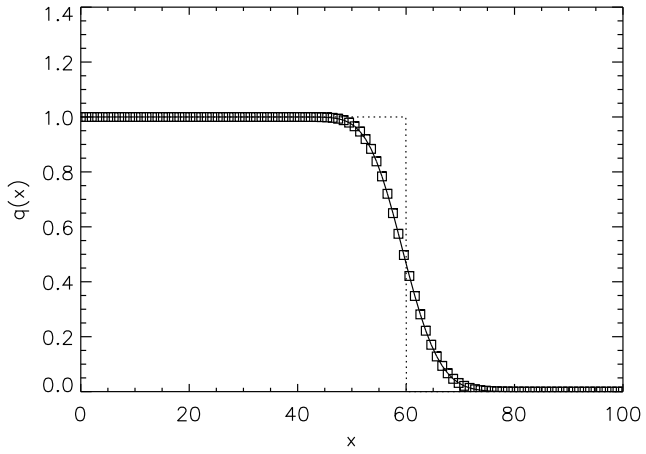

Figure 3.2. Result of center-difference algorithm for advection of a step-function from left to right. Solid line and symbols: numerical result. Dotten line: true answer (produced analytically).

3.3.2 Upstream(Upwind) differencing

One reason for the failure of the above centered-difference method is the fact that the information needed to update q_i^n in time is derived from values of q in both *upstream* and *downstream* directions[1]. The upstream direction at some point x_i is the direction $x < x_i$ (for $u > 0$) since that is the direction from which the flow comes. The downsteam direction is $x > x_i$, i.e. the direction where the stream goes. Anything that happens to the flow downstream from x_i should never affect the value of $q(x_i)$, because that information *should* flow further away from x_i. This is because, by definition, information flows downstream. Unfortunately, for the centered differencing scheme, information can flow upstream: the value of q_i^{n+1} depends as much on q_{i+1}^n (downstream direction) as it does on q_{i-1}^n (upstream direction). Clearly this is unphysical, and this is one of the reasons that the algorithm fails.

A better method is, however, easily generated:

$$\frac{q_i^{n+1} - q_i^n}{t_{n+1} - t_n} + u\frac{q_i^n - q_{i-1}^n}{x_i - x_{i-1}} = 0 \tag{3.30}$$

which is, by the way, only valid for $u > 0$. In this equation the information is clearly only from upstream to downstream. This is called *upstream differencing*, often also called *upwind differencing*. The update for the state is then:

$$q_i^{n+1} = q_i^n - \frac{\Delta t}{\Delta x} u(q_i^n - q_{i-1}^n) \tag{3.31}$$

If we now do the same experiment as before we obtain the results shown in Figure 3.2. This looks already much better. It is not unstable, and it does not produce values larger than the maximum value of the initial state, nor values smaller than the minimal value of the initial state. It is also *monotonicity preserving*, meaning that it does not produce new local minima or maxima. However, the step function is smeared out considerably, which is an undesirable property.

[1] These are also often called *upwind* and *downwind* directions.

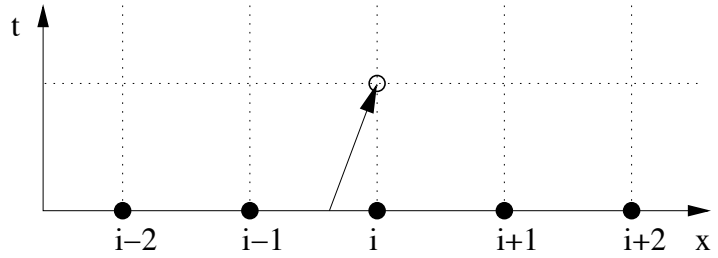

Figure 3.3. Graphical representation of the back-tracing method of the upwind scheme.

In Fig. 3.3 shows the 'physical' interpretation of the upstream algorithm. It shows that if we want to know what the value of q_i is at time $n+1$, then we can trace back the flow with a speed $-u$ from time $n+1$ to time n. We land somewhere in the middle between grid point i and grid point $i-1$ (for positive u). We say that $q_i^{n+1} = q^n(x = x_i - u\Delta t)$, and we find $q^n(x = x_i - u\Delta t)$ by linear interpolation between $i-1$ and i. If we do so, we arrive precisely at Eq. (3.31).

3.4 Numerical diffusion

The smearing-out of the solution in Section 3.3.2 is a result of *numerical diffusion*. To understand numerical diffusion, we first have to understand how true diffusion is modeled in numerical methods. This is what we will do in Subsection **??**. Then we will analyze numerical diffusion in more detail.

3.4.1 *Intermezzo:* The diffusion equation

Let us, for now, forget about the advection equation and concentrate on another type of equation:

$$\partial_t q - D\partial_x^2 q = 0 \tag{3.32}$$

This is the diffusion equation for constant diffusion coefficient D. A delta function $q(x,0) = \delta(x)$ will smear out as a Gaussian:

$$q(x,t) = \frac{1}{\sqrt{\pi}h} \exp\left(-x^2/h^2\right) \tag{3.33}$$

with

$$h(t) = \sqrt{4Dt} \tag{3.34}$$

In discretized form we obtain:

$$\frac{q_i^{n+1} - q_i}{t_{n+1} - t_n} - D\frac{2}{x_{i+1} - x_{i-1}}\left(\frac{q_{i+1}^n - q_i^n}{x_{i+1} - x_i} - \frac{q_i^n - q_{i-1}^n}{x_i - x_{i-1}}\right) = 0 \tag{3.35}$$

For constant grid spacing we obtain:

$$\frac{q_i^{n+1} - q_i}{\Delta t} - D\frac{q_{i+1}^n - 2q_i^n + q_{i-1}^n}{\Delta x^2} = 0 \tag{3.36}$$

This shows that the combination $q_{i+1}^n - 2q_i^n + q_{i-1}^n$ is a discrete way of writing a diffusion term.

3.4.2 Numerical diffusion of advection algorithms

So now let us go back to the pure advection equation. Even though this equation does not have any diffusion in it, the numerical algorithm to solve this advection equation intrinsically has some diffusion. In fact, there exists no numerical method without numerical diffusion. Some algorithms have more of it, some have less, and there exist method to constrain the smearing-out of discontinuities (see Section 4.4.3 on *flux limiters*). But in principle numerical diffusion is unavoidable.

One way of seeing this is by doing the following exercise. Consider the upwind scheme. It uses the derivative $i-1/2$ for the update of the state at i. In principle this is a bit cheating, since one 'should' use the i derivative for the update at i. So let us write the derivative at $i-1/2$ as:

$$\frac{q_i - q_{i-1}}{\Delta x} = \frac{q_{i+1} - q_{i-1}}{2\Delta x} - \Delta x \frac{q_{i+1} - 2q_i + q_{i-1}}{2\Delta x^2} \tag{3.37}$$

The left-hand-side is the upstream difference, the first term on the right-hand-side is the centered difference and the second term on the right-hand-side can be recognized as a diffusion term with diffusion constant

$$D = \frac{\Delta x u}{2} \tag{3.38}$$

This shows that the upstream difference scheme can be regarded to be the same as the centered difference scheme supplemented with a diffusion term. The pure centered difference scheme is unstable, but once a bit of diffusion is added, the algorithm stabilizes. The drawback is, however, that the diffusion smears any features out. If one would *define* the centered difference formulation of the x-derivative as the 'true' derivative (which is of course merely a definition), then the numerical diffusion of the upstream differencing scheme is quantified by D as given in Eq. (3.38).

In practice it is not possible to perfectly define the diffusivity of an algorithm since the centered difference formulation of the derivative is also merely an approximation of the true derivative. But it is nevertheless a useful way of looking at the concept of numerical diffusion. In principle one could say that it is as if we are solving

$$\partial_t q + u\partial_x q - \frac{\Delta x u}{2}\partial_x^2 q = 0 \tag{3.39}$$

Clearly, for $\Delta x \to 0$ the diffusion vanishes. This is obviously a necessary condition, otherwise we would be modeling the true diffusion equation, which is not what we want. The diffusion that we see here is merely a by-product of the numerical algorithm we used.

Note that sometimes (as we shall see below) it is useful to add some viscosity on purpose to an algorithm. This is called *artificial viscosity*. One could therefore say that the upstream differencing is equal to centered differencing plus artificial viscosity.

3.5 Courant-Friedichs-Lewy condition

No matter how stable an explicit numerical algorithm is, it cannot work for arbitrarily large time step Δt. If, in the above example (with $\Delta x = 1$ and $u = 1$), we were to use the upstream differencing method but we would take $\Delta t = 2$, then the algorithm would produce completely unstable results. The reason is the following: The function q is advected over a distance of $u\Delta t$ in a time step Δt. If $u\Delta t > \Delta x$, then within a single time step the function is advected over a larger distance than the grid spacing. However, with the above upstream differencing method the

new q_i^{n+1} depends *only* on the old q_{i-1}^n and q_i^n values. The algorithm does not include information about the value of q_{i-2}^n, but with such a large Δt it should have included it. The algorithm does not know (at least within a single time step) about q_{i-2}^n and therefore it produces something that is clearly not a solution.

To keep a numerical algorithm stable the time step has to obey the *Courant-Friedrichs-Lewy condition* (CFL condition) which states that the domain of dependence of q_i^{n+1} of the algorithm at time $t = t_n$ should include the true domain of dependence at time $t = t_n$. Or in other words: nothing is allowed to flow more than 1 grid spacing within one time step. This means quantitatively

$$\Delta t \leq \frac{\Delta x}{u} \tag{3.40}$$

So the higher the velocity u, the smaller the maximum allowed time step.

For the case that $u \rightarrow u(x)$ (i.e. space-dependent velocity) this gives different time step constraints at different locations. The allowed global time step is then the smallest of these.

Not always one wants to take this maximum allowed time step. Typically one takes:

$$\Delta t = C \min(\Delta x/u) \tag{3.41}$$

where C is the *Courant number*. If one takes this 1, then one takes the maximum allowed time step. If it is 0.5 then one takes half of it.

The CFL condition is a nessecary (but not sufficient) condition for the stability of any *explicit differencing method*. All the methods we have discussed here, and most of the methods we will discuss lateron, are explicit differencing methods. The work 'explicit' points to the fact that the updated state q_i^{n+1} is explicitly formulated in terms of $q_{i\pm k}^n$. There exist also so called 'implicit differencing' methods, but they are often too complex and therefore less often used.

3.6 Local truncation error and order of the algorithm

Now that we have seen some of the basics of numerical advection algorithms, let us analyze how accurate such algorithms are. Let us define $q_e(x,t)$ to be an exact solution to the advection equation and q_i^n a discrete solution to the numerical advection equation. The numerical algorithm will be represented by a *transport operator* T:

$$q_i^{n+1} = T[q_i^n] \tag{3.42}$$

which is another way of writing the discretized PDE. In case of the upstream differencing method we have $T[q_i^n] = q_i^n - \frac{\Delta t}{\Delta x} u(q_i^n - q_{i-1}^n)$ (cf. Eq. 3.31). For this method the T operator is a *linear operator*. Note, incidently, that if the PDE is a linear PDE, that does not guarantee that the transport operator T is also necessarily linear. Lateron in this chapter we will get to know non-linear operators that represent linear PDEs.

We can also define the discrete values of the exact solution:

$$q_{e,i}^n \equiv q_e(x_i, t_n) \tag{3.43}$$

The values $q_{e,i}^n$ do not in general strictly obey Eq. (3.42). But we can apply the operator T to $q_{e,i}^n$ and compare to $q_{e,i}^{n+1}$. In other words: we can see which error the discrete operator T introduces in one single time step compared to the true solution $q_{e,i}^{n+1}$. So let us apply T to $q_{e,i}^n$ and define the *one step error (OSE)* as follows:

$$OSE = T[q_{e,i}^n] - q_{e,i}^{n+1} \tag{3.44}$$

By using a Taylor expansion we can write $q_{e,i}^{n+1}$ as:

$$q_{e,i}^{n+1} = q_{e,i}^{n} + \left(\frac{\partial q(x_i,t)}{\partial t}\right)_{t=t_n} \Delta t + \frac{1}{2}\left(\frac{\partial^2 q(x_i,t)}{\partial t^2}\right)_{t=t_n} \Delta t^2 + \mathcal{O}(\Delta t^3) \tag{3.45}$$

If we use the upstream difference scheme, we can write:

$$T[q_{e,i}^{n}] = q_{e,i}^{n} - \frac{\Delta t}{\Delta x} u(q_{e,i}^{n} - q_{e,i-1}^{n}) \tag{3.46}$$

where we can write:

$$q_{e,i-1}^{n} = q_{e,i}^{n} - \left(\frac{\partial q(x,t_n)}{\partial x}\right)_{x=x_i} \Delta x + \frac{1}{2}\left(\frac{\partial^2 q(x,t_n)}{\partial x^2}\right)_{x=x_i} \Delta x^2 + \mathcal{O}(\Delta x^3) \tag{3.47}$$

So the OSE of this scheme becomes:

$$\begin{aligned} OSE = &- u\left(\frac{\partial q(x,t_n)}{\partial x}\right)_{x=x_i} \Delta t + \frac{1}{2} u\left(\frac{\partial^2 q(x,t_n)}{\partial x^2}\right)_{x=x_i} \Delta t \Delta x + u \Delta t \mathcal{O}(\Delta x^2) - \\ &\left(\frac{\partial q(x_i,t)}{\partial t}\right)_{t=t_n} \Delta t - \frac{1}{2}\left(\frac{\partial^2 q(x_i,t)}{\partial t^2}\right)_{t=t_n} \Delta t^2 + \mathcal{O}(\Delta t^3) \end{aligned} \tag{3.48}$$

If we ignore all terms of higher order we obtain:

$$OSE = -\Delta t \left[\left(\frac{\partial q(x_i,t)}{\partial t}\right)_{t=t_n} + u\left(\frac{\partial q(x,t_n)}{\partial x}\right)_{x=x_i} - \mathcal{O}(\Delta t) - \mathcal{O}(\Delta x)\right] \tag{3.49}$$

From the PDE we know that the first two terms between brackets cancel identically, so we obtain:

$$OSE = \Delta t[\mathcal{O}(\Delta t) + \mathcal{O}(\Delta x)] \tag{3.50}$$

So what happens when we make Δt smaller: the OSE gets smaller by a factor of Δt^2. However, one must keep in mind that one now has to do more time steps to arrive at the same simulation end-time. Therefore the final error goes down only as Δt, i.e. linear instead of quadratic. That is why it is convenient if we define the so-called *local truncation error (LTE)* as:

$$LTE \equiv \frac{1}{\Delta t}(T[q_{e,i}^{n}] - q_{e,i}^{n+1}) \tag{3.51}$$

which for this particular scheme is:

$$LTE = \mathcal{O}(\Delta t) + \mathcal{O}(\Delta x) \tag{3.52}$$

In general an algorithm is called *consistent* with the partial differential equation when the LTE behaves as:

$$LTE = \sum_{k=0,l} \mathcal{O}(\Delta t^k \Delta x^{l-k}) \tag{3.53}$$

with $l \geq 1$. The LTE is of order l, and the algorithm is said to be l-th order.

The upstream differencing algorithm is clearly a first order scheme.

3.7 Lax-Richtmyer stability analysis of numerical advection schemes

The mere fact that the LTE goes with $\mathcal{O}(\Delta t)$ and $\mathcal{O}(\Delta x)$ is not a sufficient condition for stability. It says that the operator T truly describes the discrete version of the PDE under consideration. But upon multiple successive actions of the operator T, representing the time sequence of the function q, tiny errors could conceivably grow exponentially and eventually dominate the solution. We want to find out under which conditions this happens.

To analyze stability we need to first define what we mean by stability. To do this we must define a *norm* $||.||$ by which we can measure the magnitude of the error. In general we define the p-norm of a function $E(x)$ as:

$$||E||_p = \left(\int_{-\infty}^{\infty} |E(x)|^p dx \right)^{1/p} \tag{3.54}$$

For the discretized function E_i this becomes:

$$||E||_p = \left(\Delta x \sum_{i=-\infty}^{\infty} |E_i|^p \right)^{1/p} \tag{3.55}$$

The most commonly used are the 1-norm and the 2-norm. For conservation laws the 1-norm is attractive because this norm can be directly used to represent the conservation of this quantity. However, the 2-norm is useful for linear problems because it is compatible with a Fourier analysis of the problem (see Section 3.8).

Now suppose we start at $t = 0$ with a function $q_e(x, t = 0)$ and we set the discrete numerical solution at that time to these values: $q_i^0 = q_{e,i}^0 \equiv q_e(x_i, 0)$. The evolution in time is now given by a successive application of the operator T, such that at time $t = t_n$ the discrete solution is:

$$q_i^n = T^n[q_i^0] \tag{3.56}$$

In each of these time steps the discrete solution acquires an error. We can write the total accumulated global error at time $t = t_n$ as E_i^n defined as:

$$E_i^n = q_i^n - q_{e,i}^n \tag{3.57}$$

i.e. the difference between the discrete solution and the true solution. So when we apply the operator T to q_i^n we can write:

$$q_i^{n+1} \equiv T[q_i^n] = T[q_{e,i}^n + E_i^n] \tag{3.58}$$

So we can now write the global error at time t_{n+1}, E_i^{n+1} as

$$\begin{aligned} E_i^{n+1} &= q_i^{n+1} - q_{e,i}^{n+1} \\ &= T[q_{e,i}^n + E_i^n] - q_{e,i}^{n+1} \\ &= T[q_{e,i}^n + E_i^n] - T[q_{e,i}^n] + T[q_{e,i}^n] - q_{e,i}^{n+1} \\ &= T[q_{e,i}^n + E_i^n] - T[q_{e,i}^n] + \Delta t LTE[q_{e,i}^n] \end{aligned} \tag{3.59}$$

Now in the next few paragraphs we will show that the numerical method is stable in some norm $||.||$ if the operator $T[]$ is a *contractive operator*, defined as an operator for which

$$||T[P] - T[Q]|| \leq ||P - Q|| \tag{3.60}$$

for any functions P and Q. To show this we write

$$\begin{aligned} ||E^{n+1}|| &\leq ||T[q_e^n + E^n] - T[q_e^n]|| + \Delta t||LTE[q_e^n]|| \\ &\leq ||E^n|| + \Delta t||LTE[q_e^n]|| \end{aligned} \tag{3.61}$$

If we apply this recursively we get

$$||E^N|| \leq \Delta t \sum_{n=1}^{N} ||LTE[q_e^n]|| \tag{3.62}$$

where we assume that the error at $t = t_0 = 0$ is zero. Now, the LTE is defined always on the true solution $q_{e,i}^n$. So since the true solution is for sure well-behaved (numerical instabilities are only expected to arise in the numerical solution q_i^n), we expect that the $||LTE[q_e^n]||$ is a number that is bounded. If we define M_{LTE} to be

$$M_{LTE} = \max_{1 \leq n \leq N} ||LTE[q_e^n]|| \tag{3.63}$$

which is therefore also a bound number (not suject to exponential growth), then we can write:

$$||E^N|| \leq N \Delta t M_{LTE} \tag{3.64}$$

or with $t = N\Delta t$:

$$||E^N|| \leq t M_{LTE} \tag{3.65}$$

This shows that the final global error is bound, i.e. not subject to run-away growth, if the operator T is contractive. Note that this analysis, so far, holds both for linear operators $T[.]$ as well as for non-linear operators $T[.]$. We will cover non-linear operators in the next chapter.

If $T[.]$ is linear one can write $T[q_e^n + E^n] - T[q_e^n] = T[E^n]$. In this case the stability requirement is:

$$||T[E^n]|| \leq ||E^n|| \tag{3.66}$$

which must be true for any function E_i^n. In Section 3.8 we will verify this for some simple algorithms.

Sometimes the above stability requirement is loosened a bit. The idea behind this is that we are not concerned if modes grow very slowly, as long as these modes do not start to dominate the solution. If the operator $T[.]$ obeys

$$||T[P] - T[Q]|| \leq (1 + \alpha \Delta t)||P - Q|| \tag{3.67}$$

where α is some constant (which we will constrain later), then Eq.(3.61) becomes:

$$||E^{n+1}|| \leq (1 + \alpha \Delta t)\ ||E^n|| + \Delta t||LTE[q_e^n]|| \tag{3.68}$$

and thereby Eq.(3.65) becomes:

$$||E^N|| \leq t M_{LTE} e^{\alpha t} \tag{3.69}$$

One sees that the error growth exponentially in time, which one would in principle consider an instability. But $e^{\alpha t}$ is a constant that does not depend on Δt. So no matter how large $e^{\alpha t}$ is, as long as the LTE is linear or higher in Δt (a requirement for *consistency*) one can always find a Δt small enough such that $||E^N|| \ll ||q_e^N||$ even though this might require a very large number of time steps for the integration. This means that for an operator $T[.]$ obeying Eq.(3.67) the

algorithm is formally stable. In practice, of course, the α cannot be too large, or else one would require too many time steps (i.e. too small Δt) to be of any practical use.

This leads us to a fundamental theorem of numerical integration methods: *Lax Equivalence Theorem* which says that:

$$\textit{Consistency + Stability} \;\rightarrow\; \textit{Convergence}$$

In other words: if an algorithm is *consistent* (see Eq.3.53) and *stable* (see Eq.3.69), then one can be assured that one can find a small enough Δt such that at time t the right answer is reached down to an accuracy of choice.

3.8 Von Neumann stability analysis of numerical advection schemes

The theoretical stability analysis outlined in the previous section has only reformulated the condition for stability as a condition on the operator $T[.]$. We now analyze whether a certain algorithm in fact satisfies this condition. For linear operators the Von Neumann analysis does this in Fourier space using the 2-norm. Also we will require strong stability, in the sense that we want to show that $||T[E^n]|| \leq ||E^n||$ (i.e. without the $(1 + \alpha\Delta t)$ factor).

Any function can be expressed as a sum of complex wave functions. For an infinite space one can therefore write the initial condition function $q(x)$ as

$$q(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \tilde{q}(k) e^{ikx} dk \tag{3.70}$$

Since the advection equation

$$\partial_t q + u\partial_x q = 0 \tag{3.71}$$

merely moves this function with velocity u, the solution $q(x,t) = q(x-ut)$ translates in a $\tilde{q}(k,t)$ given by

$$\tilde{q}(k,t) = \tilde{q}(k) e^{-iukt} \tag{3.72}$$

which is just a phase rotation. In Fourier space, the true operator $T_e[.]$ is therefore merely a complex number: $T_e = e^{-iuk\Delta t}$. As we shall see below, the numerical operator in Fourier space is also a complex number, though in general a slightly different one. So we need to compare the numerical operator $T[.]$ with the true operator $T_e[.]$ to find out what the local truncation error (LTE) is.

Formally, when we follow the logic of Section 3.7, we need to let the operator $T[.]$ act on the Fourier transforms of $q^n_{e,i} + E^n_i$ and $q^n_{e,i}$ and subtract them. Or since the operator is linear, we must apply it to the Fourier transform of E^n_i. However, since we have assumed that the operator is linear, we can also do the analysis directly on $\tilde{q}^n_e(k)$ (the Fourier transform of $q^n_{e,i}$) and check if the resulting amplitude is ≤ 1. The advantage is that we can then directly derive the LTE in terms of an amplitude error and a phase error. If the amplitude of the operator T is ≤ 1 for all values of k, Δx and $\Delta t \leq C\Delta x/|u|$ (where u is the advection speed and C is the Courant number) then we know that the function $\tilde{q}(k,t)$ is not growing exponentially, and therefore also the error is not growing exponentially. Moreover, we know that if this amplitude is much smaller than 1, then the algorithm is very diffusive. We can also analyze the phase error to see if the algorithm transports each mode with the correct speed.

3.8.1 Analyzing the centered differencing scheme

Let us descretize this function as:

$$q_i^0 := q(x = x_i, 0) = e^{ikx_i} \tag{3.73}$$

Now insert this into the centered difference scheme:

$$q_i^{n+1} = q_i^n - \frac{\Delta t}{2\Delta x} u(q_{i+1}^n - q_{i-1}^n) \tag{3.74}$$

We obtain

$$q_i^1 = e^{ikx_i} - \frac{\Delta t}{2\Delta x} u(e^{ik(x_i+\Delta x)} - e^{ik(x_i-\Delta x)}) = e^{ikx_i}\left[1 - \frac{\Delta t}{2\Delta x} u(e^{ik\Delta x} - e^{-ik\Delta x})\right] \tag{3.75}$$

Let us define:

$$\epsilon \equiv u\frac{\Delta t}{\Delta x} \tag{3.76}$$

and

$$\beta \equiv k\Delta x \tag{3.77}$$

then we can write

$$q_i^{n+1} = q_i^n T^C \tag{3.78}$$

with T^T the transfer function, which for centered differencing is apparently

$$T^C = 1 - \frac{\epsilon}{2}(e^{i\beta} - e^{-i\beta}) \tag{3.79}$$

The C in the transfer function stands for 'centered differencing'. We can write T^C as

$$T^C = 1 - i\epsilon \sin\beta \tag{3.80}$$

This transfer function is most easily analyzed by computing the squared magnitude R

$$R = T^*T = (\mathrm{Re}T)^2 + (\mathrm{Im}T)^2 \tag{3.81}$$

and the phase Φ

$$\tan\Phi = \frac{\mathrm{Im}T}{\mathrm{Re}T} \tag{3.82}$$

which for this algorithm are:

$$R^T = 1 + \epsilon^2 \sin^2\beta \qquad , \qquad \tan\Phi^T = -\epsilon\sin\beta \tag{3.83}$$

We can now compare this to our analytic solution (the solution that should have been produced): $q(x,t) = e^{ik(x-u\Delta t)}$. Clearly this analytic solution has $R = 1$ and a phase of $\Phi = uk\Delta t$ (if phase is measured negatively). Compared to this solution we see that:

1. The centered differencing scheme diverges: the amplitude of the solution always gets bigger. For very small time steps this happens with a factor $1 + (uk\Delta t)^2$. So clearly it gets better for smaller time steps (even if we have to take more of them), but it still remains *unconditionally unstable*.

2. The phase also has an error: $u\Delta t[\sin(k\Delta x) - k\Delta x]/\Delta x$. For very small time steps the one-step phase error becomes: $-k^3 u\Delta t\Delta x^2/6$, which means that the phase error grows linearly in time, and for a given final time t it is thus independent of Δt.

These results confirm our numerical experience that the centered differencing method is unconditionally unstable.

3.8.2 Now adding artificial viscosity

We have seen in the numerical experiments of Section 3.3.2 that adding *artificial viscosity* (see Section 3.4.2) can stabilize an algorithm. So let us now consider the following advection scheme:

$$q_i^{n+1} = q_i^n - \frac{\Delta t}{2\Delta x} u(q_{i+1}^n - q_{i-1}^n) + D\frac{\Delta t}{\Delta x^2}(q_{i+1}^n - 2q_i^n + q_{i-1}^n) \tag{3.84}$$

Let us define

$$\nu = D\frac{\Delta t}{\Delta x^2} \tag{3.85}$$

so that we get

$$q_i^{n+1} = q_i^n - \frac{\epsilon}{2}(q_{i+1}^n - q_{i-1}^n) + \nu(q_{i+1}^n - 2q_i^n + q_{i-1}^n) \tag{3.86}$$

Let us again insert $q_i^0 := q(x = x_i, 0) = e^{ikx_i}$ so that we obtain

$$\begin{aligned} q_i^1 &= e^{ikx_i} - \frac{\epsilon}{2}(e^{ik(x_i+\Delta x)} - e^{ik(x_i-\Delta x)}) + \nu(e^{ik(x_i+\Delta x)} - 2e^{ikx_i} + e^{ik(x_i-\Delta x)}) \\ &= e^{ikx_i}\left[1 - \frac{\epsilon}{2}(e^{ik\Delta x} - e^{-ik\Delta x}) + \nu(e^{ik\Delta x} - 2 + e^{-ik\Delta x})\right] \end{aligned} \tag{3.87}$$

so the transfer function becomes:

$$T^{CD} = 1 - \frac{\epsilon}{2}(e^{ik\Delta x} - e^{-ik\Delta x}) + \nu(e^{ik\Delta x} - 2 + e^{-ik\Delta x}) \tag{3.88}$$

or in other terms:

$$T^{CD} = 1 - i\epsilon\sin\beta + 2\nu(\cos\beta - 1) \tag{3.89}$$

The R and Φ are:

$$R^{CD} = \epsilon^2\sin^2\beta + (1 + 2\nu(\cos\beta - 1))^2 \tag{3.90}$$

and

$$\tan\Phi^{CD} = -\frac{\epsilon\sin\beta}{1 + 2\nu(\cos\beta - 1)} \tag{3.91}$$

Figure 3.4 shows the transfer function in the complex plane. Whenever the transfer function exceeds the unit circle, the mode grows and the algorithm is unstable. Each of the ellipses shows the complete set of modes (wavelengths) for a given ϵ and ν. None of the points along the ellipse is allowed to be beyond the unit circle, because if any point exceeds the unit circle, then there exists an unstable mode that will grow exponentially and, sooner or later, will dominate the solution.

From these figures one can guess that whether an ellipse exceeds the unit circle or not is already decided for very small β (i.e. very close to $T = 1$). So if we expand Eq. (3.90) in β we obtain

$$R^{CD} \simeq 1 + \beta^2(\epsilon^2 - 2\nu) + \mathcal{O}(\beta^3) \tag{3.92}$$

We see that for the centered-differencing-and-diffusion algorithm to be stable one must have

$$\nu \geq \epsilon^2/2 \tag{3.93}$$

→ **Exercise:** Argue what is the largest allowed ϵ for which a ν can be found for which all modes are stable. Hint: Use the graphs in Fig.3.4 (and their versions for other values of ν) for your argument.

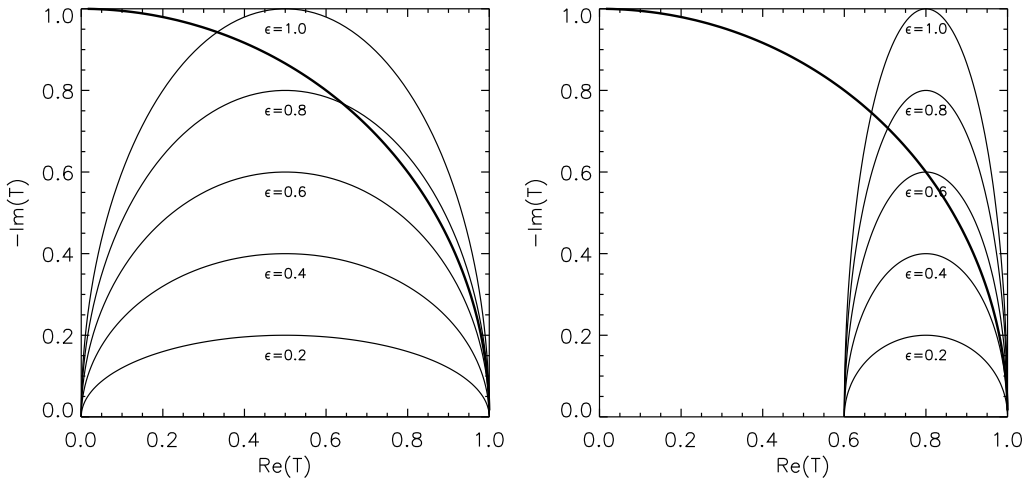

Figure 3.4. The complex transfer function for the centered differencing advection scheme with added artificial viscosity (imaginary axis is flipped). The ellipses are curves of constant advection parameter $\epsilon = u\Delta t/\Delta x$ (marked on the curves) and varying β (i.e. varying wavelength). The thick line is the unit circle. Left: $\nu = 0.25$, right: $\nu = 0.1$. Whenever the transfer function exceeds the unit circle, the algorithm is unstable since the mode grows.

→ **Exercise:** Analyze the upstream differencing scheme of Section 3.3.2 with the above method, show that it is stable, and derive whether (and if so, how much) this algorithm is more diffusive than strictly necessary for stability.

Clearly for $\nu \geq \epsilon^2/2$ the scheme is stable, but does it produce reasonable results? Let us first study the phase Φ^{CD} and compare to the expected value. The expected value is:

$$\Phi = -uk\Delta t = -\epsilon\beta \tag{3.94}$$

Comparing this to Eq. (3.91) we see that to first order the phase is OK, at least for small β. The error appears when β gets non-small, i.e. for short wavelength modes. This is not a surprise since short wavelength modes are clearly badly sampled by a numerical scheme.

What about damping? From the phase diagram one can see that if we choose ν larger than strictly required, then the ellipse moves more to the left, and thereby generally toward smaller R^{CD}, i.e. strong damping. For small ϵ (assuming we choose $\nu = \epsilon^2/2$) we see that the ellipses flatten. This means that short-wavelength (i.e. large β) modes are strongly damped, and even longer wavelength modes (the ones that we should be able to model properly) are damped. Since this damping is an exponential process (happening after each time step again, and thereby creating a cumulative effect), even a damping of 10% per time step will result in a damping of a factor of 10^{-3} after only 65 time steps. Clearly such a damping, even if it seems not too large for an individual time step, leads to totally smeared out results in a short time. This is not what we would like to have. The solution should, ideally, move precisely along the unit circle, but realistically this is never attainable. The thing we should aim for is an algorithm that comes as close as possible to the unit circle, and has an as small as possible error in the phase.

3.8.3 Lax-Wendroff scheme

If we choose, in the above algorithm, precisely enough artificial viscosity to keep the algorithm stable, i.e.

$$\nu = \frac{1}{2}\epsilon^2 \tag{3.95}$$

then the algorithm is called the *Lax-Wendroff* algorithm. The update for the Lax-Wendroff scheme is evidently:

$$q_i^{n+1} = q_i^n - \frac{\epsilon}{2}(q_{i+1}^n - q_{i-1}^n) + \frac{\epsilon^2}{2}(q_{i+1}^n - 2q_i^n + q_{i-1}^n) \tag{3.96}$$

Interestingly, the Lax-Wendroff scheme also has another interpretation: that of a *predictor-corrector method*. In this method we first calculate the $q_{i-1/2}^{n+1/2}$ and $q_{i+1/2}^{n+1/2}$, i.e. the mid-point fluxes at half-time:

$$q_{i-1/2}^{n+1/2} = \tfrac{1}{2}(q_i^n + q_{i-1}^n) + \tfrac{1}{2}\epsilon(q_{i-1}^n - q_i^n) \tag{3.97}$$

$$q_{i+1/2}^{n+1/2} = \tfrac{1}{2}(q_i^n + q_{i+1}^n) + \tfrac{1}{2}\epsilon(q_i^n - q_{i+1}^n) \tag{3.98}$$

Then we write for the desired q_i^{n+1}:

$$q_i^{n+1} = q_i^n + \epsilon(q_{i-1/2}^{n+1/2} - q_{i+1/2}^{n+1/2}) \tag{3.99}$$

Working this out will directly yield Eq. (3.96).

3.9 Phase errors and Godunov's Theorem

The Lax-Wendroff scheme we derived in the previous section is the prototype of second order advection algorithms. There are many more types of second order algorithms, and in the next chapter we will encounter them in a natural way when we discuss piecewise linear advection schemes. But most of the qualitative mathematical properties of second order linear schemes in general can be demonstrated with the Lax-Wendroff scheme.

One very important feature of second order schemes is the nature of their phase errors. Using the following Taylor expansions

$$\text{atan}x = x - \tfrac{1}{3}x^3 + O(x^4) \qquad \cos x = 1 - \tfrac{1}{2}x^2 + O(x^4) \qquad \sin x = x - \tfrac{1}{6}x^3 + O(x^4) \tag{3.100}$$

we can write the difference $\Phi^{CD} - \Phi_e$:

$$\begin{aligned} \delta\Phi^{CD} \equiv \Phi^{CD} - \Phi_e &= -\text{atan}\left[\frac{\epsilon \sin\beta}{1 + 2\nu(\cos\beta - 1)}\right] + \epsilon\beta \\ &\simeq \epsilon\beta^3\left[\tfrac{1}{6} - \nu + \tfrac{1}{3}\epsilon^2\right] \end{aligned} \tag{3.101}$$

A phase error can be associated with a *lag* in space: the wave apparently moves too fast or too slow. The lag corresponding to the particular phase error is: $\delta x = \delta\Phi/k \propto \delta\Phi/\beta$. The lag per unit time is then $d(\delta x)/dt = \delta x/\Delta t \propto \delta\Phi/(\beta\epsilon)$. So this becomes

$$\frac{d(\delta x)}{dt} \propto \beta^2\left[\tfrac{1}{6} - \nu + \tfrac{1}{3}\epsilon^2\right] \tag{3.102}$$

One sees that the spatial lag is clearly dramatically rising when $\beta \to 1$, i.e. for wavelength that approach the grid size. In other words: the shortest wavelength modes have the largest error in their propagation velocity.

Now suppose we wish to advect a block function:

$$q(x) = \begin{cases} 1 & \text{for } x < x_0 \\ 0 & \text{for } x > x_0 \end{cases} \tag{3.103}$$

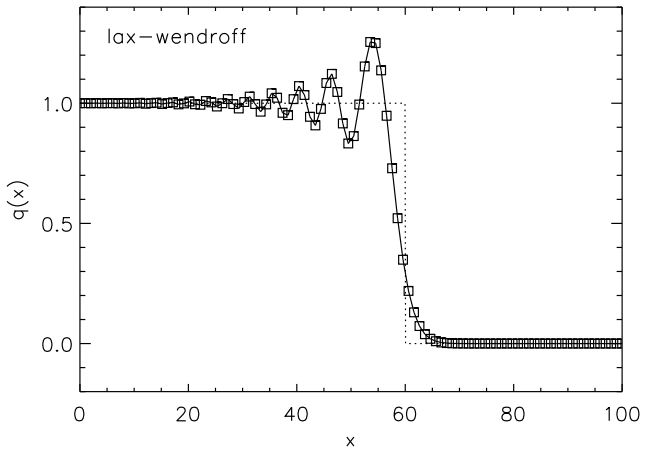

Figure 3.5. The spurious oscillations which spontaneously arise if the Lax-Wendroff method is applied to a jump solution.

Such a block function can be regarded as a Fourier sum of waves of varying wavelength. In such a sharp jump all wavelengths are represented. If we now advect this function over the grid, then we see that the shortest wavelength components will lag most behind while the larger wavelength modes do relatively fine.

For the upwind scheme ($\nu = 0$) these phase errors simply smear out any jumps like jelly. For the Lax-Wendroff scheme these phase errors produce spurious oscillations (see Fig. 3.5). Note that in contrast to the unstable centered-difference scheme these oscillations remain bound and cause no major damage. Yet, they are clearly unwanted.

Can one find a second order scheme that does not have these spurious oscillations? There are methods that at least produce less strong oscillations, such as the Fromm method (see Chapter 4). But there is a theorem, due to Sergei K. Godunov, that states that *any linear algorithm for solving partial differential equations, with the property of not producing new extrema, can be at most first order.* This is known as *Godunov Theorem*. There is therefore no hope of finding a linear second order accurate scheme that does not produce these unwanted wiggles. In Chapter 4 we will discuss *non-linear* schemes that combine the higher order accuracy and the prevention of producing unwanted oscillations.

3.10 Computer implementation (adv-1): grids and arrays

So far everything in this chapter was theoretical. Now let us see how we can put things in practice. Everything here will be explained in terms of the computer programming language IDL *Interactive Data Language* (or its public domain clone "GDL", *Gnu Data Language*). This language starts counting array elements always from 0 (like C, but unlike Fortran, although in Fortran one can set the range by hand to start from 0).

Let us write a program for testing the upwind algorithm and let us call it 'advection.pro'. Both the x-grid and the quantity q will now be arrays in the computer:

```
nx   = 100
x    = dblarr(nx)
q    = dblarr(nx)
qnew = dblarr(nx)
```

We can produce a regular grid in x in the following way

```
dx = 1.d0          ; Set grid spacing
for i=0,nx-1 do x[i] = i*dx
```

In IDL this can be done even easier with the command `dindgen`, but let us ignore this for the moment. Now let us put some function on the grid:

```
for i=0,nx-1 do if x[i] lt 30. then q[i]=1.d0 else q[i]=0.d0
```

Now let us define a velocity, a time step and a final time:

```
u    = 1.d0
dt   = 2d-1
tend = 30.d0
```

As a left boundary condition (since $u > 0$) we can take

```
qleft = q[0]
```

Now the simple upwind algorithm can be done for grid point `i=1,nx-1`:

```
time = 0.d0
while time lt tend do begin
    ;;
    ;; Check if end time will not be exceeded
    ;;
    if time + dt lt tend then begin
        dtused = dt
    endif else begin
        dtused = tend-time
    endelse
    ;;
    ;; Do the advection
    ;;
    for i=1,nx-1 do begin
        qnew[i] = q[i] - u * ( q[i] - q[i-1] ) * dtused / dx
    endfor
    ;;
    ;; Copy qnew back to q
    ;;
    for i=1,nx-1 do begin
        q[i] = qnew[i]
    endfor
    ;;
    ;; Set the boundary condition at left side (because u>0)
    ;; (Note: this is not explicitly necessary since we
    ;; didn't touch q[0])
    ;;
    q[0] = qleft
```

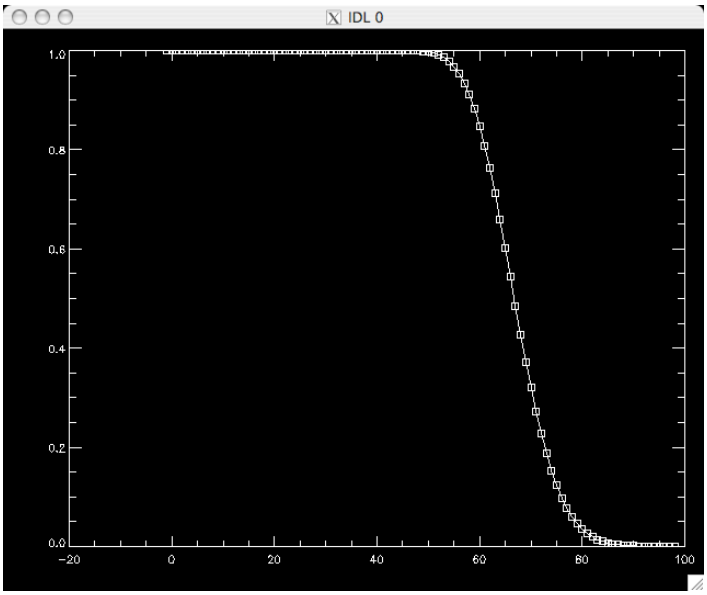

Figure 3.6. The plot resulting from the advect.pro program of Section 3.10.

```
    ;;
    ;; Update time
    ;;
    time = time + dtused
endwhile
```

Now we can plot the result

```
plot,x,q,psym=-6
```

At the end of the program we must put an

```
end
```

Now we can go into IDL and type `.r advection.pro` and we should get a plot on the screen (Fig. 3.6).