Chapter 8

Initial Value problems in multiple dimensions

Selected Reading

Numerical Recipes, 2nd edition: Chapter 19

8.1 Introduction

Nearly all of the techniques and concepts for initial value problems in one-dimension carry over into 2 and 3 dimensions. Conceptually, multi-dimensional problems are not any more difficult to solve. However, higher dimensions can introduce significant new physical behaviour and certainly introduce considerable expense in both computation and visualization. The principal difficulty is that the shear number of grid points increases as the power of the dimension, thus while a 1-D problem with 10,000 grid points is trivial on a moderate speed workstation, the comparable 2-D problem has only 100×100 grid points and the 3-D problem $\sim 22 \times 22 \times 22$ (not a lot of liebesraum if you ask me). The bottom line is that in multiple dimensions you either need to get clever, have access to an enormous machine or learn to live with poor resolution. C'est la vie. However, the amount of interesting behaviour gained in higher dimensions is well worth the effort. This section will layout most of the simplest concepts and techniques for solving time-dependent initial value problems in multiple dimensions.

8.2 Practical Matters: Storage, IO and visualization in multiple dimensions

Before we get on to the algorithms etc., however, there are a few practical computational issues that are probably worth going through. Here we will illustrate most of these issues using 2-D arrays but the extensions to 3-D are straightforward.

Array storage The first problem is how to store multi-dimensional arrays. In Fortran77, you actually have several options. First you could declare all your main arrays as 2-D arrays such as

```
parameter(IMAX=201, JMAX=101)
real myarray(IMAX,JMAX)
```

which describes a 2-D grid which is essentially 101 1-D arrays of 201 real numbers in each contiguous array. With this declaration the *physical dimensions* of this array are 201×101. As I tend to use the `i` direction for the x direction, this array would be short and wide. This approach is fine but suppose your problem has only 51×51 points in it i.e. the *logical dimensions* of the problem are 51×51. If you wanted to pass this array to a subroutine you would have to pass both the physical and logical dimensions of the array e.g.

```
call dosomething(myarray,51,51,201,101,...)
```

where the subroutine might look something like

```
subroutine dosomething(array,ni,nj,nimax,njmax,,...)
implicit none
integer ni,nj,nimax,njmax
real array(nimax,njmax)
...
do j=1,nj
   do i=1,ni
       array(i,j)=whatever your heart desires
   enddo
enddo
return
end
```

The problem with this is that it is rather wasteful of space as the last 150 elements of the first 51 rows have to be skipped over. In addition if you wanted to do a problem that had 101×201 grid points (i.e. one that is tall and thin) you would have to redeclare your array because, although, you have enough storage for this many numbers, the array is the *wrong shape*.

In f90, multi-dimensional arrays can be created or reshaped on the fly and many of these issues become much easier to deal with. In f77 however, you must still declare storage for all arrays at compile time. The standard fix for array flexibility in F77 is to do all the storage in **1-D arrays** and to use the important fortran hack that you can pass 1-D arrays to subroutines which can handle them as multiple-dimension arrays. For example, we could do the previous problems as

```
parameter(NMAX=20301)      ! !that's 201*101
real myarray(NMAX)

....
call dosomethingnew(myarray,51,51,...)
....
subroutine dosomethingnew(arr,ni,nj,....)
implicit none
integer ni,nj
```

```
real arr(ni,nj)

etc.
```

With this approach you only use the first 51^2 elements of storage and you can solve any problem with up to `NMAX` points independent of the shape of the array. This is the approach I will take for most of my codes.

IO The next hassle in multi-dimensions is input and output. While ascii is easy to read and transport, it is bulky, inaccurate (unless you save lots of digits) and is surprisingly slow to write. For large multi-dimensional problems your best bet is binary. Unfortunately nobody has agreed upon a common binary format that can be read into other analysis or visualization packages with ease (there are actually lots of competing formats but no real standards[1]). To solve both problems I have written a few basic IO subroutines for reading and writing 2 and 3-D arrays with just enough of a header to give information about grid dimensions, minimum and maximum real dimensions, the time step and the time. For some reason now lost in the mists of time, I've called one of these binary files a *set* and have written a large pile of conversion routines to convert sets to many common formats e.g. `settomat` to convert to matlab `.mat` files.

A set is always stored as 4byte reals (but allows for conversion to and from double precision). The basic format of a set is 3 records

```
ndims, nstep, time
ndimarr(1:ndims), xdimarr(1:2*ndims)
data_array(1:ntot)
```

with variables declared as

```
integer ndims, nstep
integer ndimarr(ndims)
real*4 time
real*4 xdimarr(2*ndims), data_array(ntot)
```

where `ntot` is the total number of points in the array, and is simply the product of the components of `ndimarr`. For example if we wanted to write a 2-D mesh with 50×100 grid-points that spans the physical domain $0 < x < 1$, $1 < y < 3$ to a file called `junkset` we would do something like.

```
ndims=2              ! two-dimensional array
ni=50
nj=100
ndimarr(1)=ni      ! number of i points
ndimarr(2)=nj    !  number of jpoints
xdimarr(1)=0.       ! xmin
xdimarr(2)=1.       ! xmax
xdimarr(3)=1.       ! ymin
xdimarr(4)=3.       ! ymax
call dosomething(myarray,ni,nj)
wrset1('junkset',myarray,wk,ndims,ndimarr,xdimarr,nstep,time)
```

[1] Although much of the climate community is moving towards the NetCDF/Unidata format.

note the work array `wk` is required to guarantee conversion to single precision. The corresponding subroutine to read a set is `rdset1` and both routines can be found in `setio1.f`.

Set conversion and visualization Given at least one consistent way of handling multi-dimensional binary IO, we still need to convert it to a variety of formats so other programs like Matlab, Transform, GMT, xv or AVS (or anything else you like) can do something useful with it. The set conversion routines that I have already written are

settoascii `Usage: settoascii filename > outfile` converts a 2-D set to a 1-D ascii dump of just the surface values plus a 7 line header. writes to standard out.

settoxyz `Usage: settoxyz filename > outfile` does a fully indexed ascii dump of either 2 or 3-D sets plus header. Each line of a 2-D set looks like `x y value` for 3-D sets it is `x y z value`

settomat `Usage: settomat filename` converts a 2-D set to a Matlab .mat file (which will be called filename.mat). These files define 3 matrices X, Y, and Z where X and Y are 1-D vectors that hold the X and Y domain coordinates and Z is a 2-D matrix with the array in it. use `load filename` from within matlab to read it in. It is up to the user to rename Z to whatever.

settopgm `Usage: settopgm filename [amin amax] [-w<width>] >outfile` converts a 2-D set to an 8-bit grey-scale image in PGM format (portable greymap). This file can be read directly into xv or can be converted to an immense number of other image formats using the pbmplus utilities. Options are `[ amin amax ]` will set black to amin and white to amax. Otherwise the image will be scaled to the max and min value of the set. option -w allows to set the width of the image in pixels (i.e. allows to blow up the image using bilinear interpolation). Without the -w option it should default to the natural width of the image but this sometimes makes it explode. Writes to standard out.

If you also have another favourite package that you can't get to from here, let me know and I can tweak these codes.

In addition to the set conversion routines which convert one set to one other kind of file, I also have a set of utility scripts for working on a set of sets from a given run. To use these, they expect that the output files for a given run will be named something like rootname.000 rootname.001, then invoking `c2dtomat rootname` (for example) will convert all the files that start with rootname to matlab files. The current set of utility scripts are

c2dtoascii calls settoascii for all files

c2dtoxyz calls settosxy for all files

c2dtomat calls settomat

c2dtogif `Usage: c2dtogif fileroot[.???] [ min max ]` calls settopgm and for all files and remaps them into 8-bit gif files. Optional min and max values scale the image between min (blue) and max (red). To set the width of any of the image files, first set the environment variable `PGMWIDTH`, i.e. to get images that are 200 pixels wide first use `setenv PGMWIDTH 200` then call `c2dtogif`.

c2dtogifg like c2dtogif but automatically scales to the global min/max of the files

c2dtogifmv `Usage: c2dtogifmv root [min max]`. Calls either of the above routines (depending on if min max is specified) and creates a single animated gif movie. To control movie parameters first set the environment variable `MOVIEENV`. E.g. to make a movie that is 129 pixels wide, has a 1 millisecond delay between frames (and can be speed controlled in Xanim) and loops forever use `setenv MOVIEENV ''129 1 0''`, then call `c2dtogifmv`.

To get the basic usage and any options just type the name of the script. All of these scripts are in either `~mm_ms/classprogs_sun4/` or `~mm_ms/classprogs_sun5/`. Take a look at the scripts and feel free to modify these to your tastes.

8.3 Spatial Differencing in 2-D

Okay, now that all the tedious computation is out of the way, let's get on to actually solving problems in multiple dimensions. Finite difference approximations for time derivatives are exactly the same in all dimensions (e.g. a forward time step is $\partial f/\partial t \sim (f^{n+1} - f^n)/\Delta t$). The only new schemes that are required are approximations for the spatial partial derivatives in several directions. In general the Taylor series approach can always be used to derive finite difference approximations. However, the full Taylor series for a scalar function in n dimensions looks like

$$f(\mathbf{x}+\mathbf{h}) = f(\mathbf{x})+\sum_{i=1}^{n} h_i\frac{\partial f}{\partial x_i}(\mathbf{x})+\frac{1}{2!}\sum_{i,j=1}^{n} h_i h_j\frac{\partial^2 f}{x_i x_j}(\mathbf{x})+\frac{1}{3!}\sum_{i,j,k=1}^{n} h_i h_j h_k\frac{\partial^3 f}{x_i x_j x_k}(\mathbf{x})+O(|\mathbf{h}|^4) \tag{8.3.1}$$

where $\mathbf{h}$ is the displacement vector from point $\mathbf{x}$. Thus in multiple dimensions, second order (and higher) cross derivatives become a possibility and contribute to the truncation error. Nevertheless, for simple schemes on an orthogonal mesh, the cross terms are not directly used. For example, for an orthogonal mesh if we only take a step in the x direction ($\mathbf{h} = (\Delta x, 0, 0)$) then the general Taylor series reduces to the 1-D series

$$f(\mathbf{x}+\Delta x) = f(\mathbf{x}) + \Delta x\frac{\partial f}{\partial x}(\mathbf{x}) + \frac{1}{2}(\Delta x)^2\frac{\partial^2 f}{\partial x^2}(\mathbf{x}) + \frac{1}{3!}(\Delta x)^3\frac{\partial f}{\partial x}(\mathbf{x}) + O(\Delta x^4) \tag{8.3.2}$$

and likewise for the y or z direction. Thus, in the standard way, we could approximate centered first derivatives in each direction as

$$\frac{\partial f}{\partial x} \approx \frac{f_{i+1,j} - f_{i-1,j}}{2\Delta x} + O(\Delta x^2)$$

$$\frac{\partial f}{\partial y} \approx \frac{f_{i,j+1} - f_{i,j-1}}{2\Delta y} + O(\Delta y^2) \tag{8.3.3}$$

etc. (in 3-D we need 3 array indices i, j, k). Which are identical to those developed for 1-D problems. Likewise second derivatives in Cartesian geometry are

$$\begin{aligned}\frac{\partial^2 f}{\partial x^2} &\approx \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{\Delta x^2} + O(\Delta x^2) \\ \frac{\partial^2 f}{\partial y^2} &\approx \frac{f_{i,j+1} - 2fi,j + f_{i,j-1}}{2\Delta y} + O(\Delta y^2)\end{aligned} \tag{8.3.4}$$

It is important to note that while the finite-difference approximations are the same as in 1-D problems, the truncation error is more complicated as it contains all the cross derivative terms inherent in Eq. (8.3.1). For non-Cartesian meshes, these additional terms can become important and the cross derivatives can also introduce interesting new artifacts including anisotropy in the errors.

In addition to the Taylor series approach, the same sort of differencing schemes can be developed somewhat more intuitively using the control volume approach. For example, if we consider each node to represent the average concentration of a small 2-D control volume of area $\Delta x \Delta y$ (See Fig. 8.1). Then the divergence of

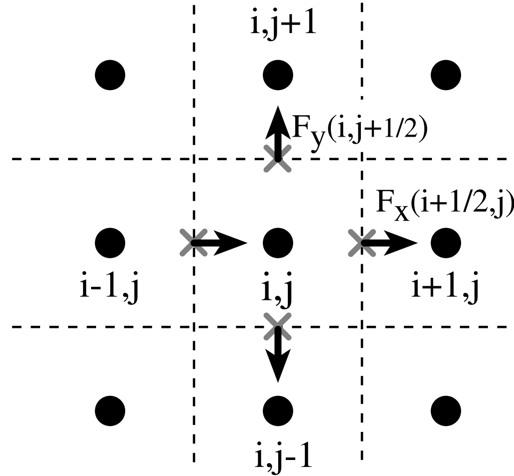

Figure 8.1: A sample of a staggered-mesh used for developing finite difference schemes using the control-volume approach. The principal nodes are denoted by dots, the half-nodes on the staggered mesh are shown by crosses. A cell-centered control volume is the small rectangular region of area $\Delta x \Delta y$ centered on each node.

any flux through this volume can be approximated by

$$\nabla \cdot \mathbf{F} \approx \frac{F^x_{i+1/2,j} - F^x_{i-1/2,j}}{\Delta x} + \frac{F^y_{i,j+1/2} - F^y_{i,j-1/2}}{\Delta y} \tag{8.3.5}$$

Another way to derive (8.3.5) is to consider gauss' theorem

$$\int_V \nabla \cdot \mathbf{F} dV = \int_S \mathbf{F} \cdot d\mathbf{S} \tag{8.3.6}$$

if we set $dV = \Delta x \Delta y$ to be the area (or volume) of the control volume then the average value of the divergence over the control volume becomes

$$\overline{\nabla \cdot \mathbf{F}} \Delta x \Delta y = \left[\bar{F}_x(x + \frac{\Delta x}{2}) - \bar{F}_x(x - \frac{\Delta x}{2}) \right] \Delta y + \left[\bar{F}_y(y + \frac{\Delta y}{2}) - \bar{F}_y(y - \frac{\Delta y}{2}) \right] \Delta x \tag{8.3.7}$$

where $\bar{\mathbf{F}}$ is the average flux through any side of the volume. Comparison of (8.3.5) and (8.3.7) shows that the two formulations are equivalent, however the conservative formulation of (8.3.7) can be more readily generalized to more complex geometries. To complete the differencing scheme, however, requires some interpolation scheme for determining the average fluxes at the edges of the control volume. Different choices of interpolation produce the different standard schemes (and then some).

8.3.1 Boundary conditions

Before we discuss specific schemes for advection and diffusion, we need to discuss the ever-present problem of boundary conditions. In 1-D, the boundaries are single points. In 2- and 3-D, however, the boundaries can have much more significant affects on the behaviour of the solutions (and cause more numerical nightmares). Fortunately, most of the ways of calculating boundary conditions in 1-D go directly to higher dimensions. The only slightly tricky part is dealing with corner points where two (or more) boundaries meet and the fact that one must be careful in selecting boundary conditions that are consistent with the problem to be solved. As usual there are three principal types of boundary conditions that commonly occur: Dirichlet conditions, von Neumann (flux) conditions and periodic (wrap-around) boundaries.

Dirichlet BC's If an edge is Dirichlet then all the points along that edge are specified although they do not have to be constant. There are several ways of implementing Dirichlet conditions. The first is simply to specify the boundary value and only update the interior points. As an example if the left edge is Dirichlet then for $i = 1$ you set `T(1,j)=f(j,t))` and then begin updating the first unknown which is $i = 2$. Alternatively, for implicit schemes, or schemes where stencils are used, you can simply take the known values to the RHS of the equation and assume the boundary is homogeneous `T(1,j)=0`. The simplest (and usually stablest) of all problems is when all the boundaries are Dirichlet. In this case the only unknowns are interior points and the problem is straightforward.

von Neumann BC's Flux boundary conditions are only a bit more difficult to implement. The important point to consider in multiple dimensions is that the only flux that need be specified is the flux *normal* to the boundary. Thus if a diffusive flux is zero on the left hand boundary, the requirement is only that $\partial T/\partial x = 0$ on the left boundary. If you also attempt to specify the vertical temperature gradient on this boundary, as well you are over-specifying the problem. Again, there are two common ways to implement a flux or gradient boundary condition. The first is

to simply include in your 2-D array, another set of *buffer points* just outside your solution domain. In fortran, this can be done easily by dimensioning your array as `T(0:ni+1,0:nj+1)` if there are `ni x nj` points on your computational domain. Since the centered difference approximation to $\partial T/\partial x$ is

$$\frac{\partial T}{\partial x} \approx \frac{T_{i+1,j} - T_{i-1,j}}{2\Delta x} \tag{8.3.8}$$

then the requirement that the horizontal gradient is zero at $i = 1$ becomes simply that $T_{0,j} = T_{2,j}$ and the first interior point can be copied in to the buffer. Alternatively, if we have some form of general stencil equation for our interior points such as

$$\begin{bmatrix} & d_{i,j} & \\ a_{i,j} & b_{i,j} & c_{i,j} \\ & e_{i,j} & \end{bmatrix} T^{n+1} = f_{i,j} \tag{8.3.9}$$

then the boundary condition that $\partial T/\partial x = 0$ (or `T(0,j)=T(2,j)`) becomes the modified stencil equation for the $i = 1$ point

$$\begin{bmatrix} & d_{1,j} & \\ 0 & b_{1,j} & a_{1,j} + c_{1,j} \\ & e_{1,j} & \end{bmatrix} T^{n+1} = f_{1,j} \tag{8.3.10}$$

likewise if we are in the lower left corner of a 2-D problem at the intersection of 2 Neumann boundaries we could write the stencil for the corner point as

$$\begin{bmatrix} & d_{1,j} + e_{1,j} & \\ 0 & b_{1,j} & a_{1,j} + c_{1,j} \\ & 0 & \end{bmatrix} T^{n+1} = f_{1,j} \tag{8.3.11}$$

More general boundary stencil's can be derived using a control volume approach for the edge volumes. In general, Neumann conditions can be considerably less stable than Dirichlet conditions and a lot of problems arise from using an edge condition or approximation that is inconsistent with the interior solution. In multi-dimensional problems we often spend most of our time just getting the boundary conditions to work without exploding.

Periodic BC's Periodic boundary conditions are not really boundary conditions (which makes them infinitely easier to deal with). Very simply if our horizontal i direction is assumed to be periodic, then we are saying that `T(1,j)=T(ni,j)` and therefore when we want the $i - 1$ point at the boundary we simply reach around and assume that `T(0,j)=T(ni-1,j)`. For explicit schemes, wraparound BC's are trivial. Implicit schemes can be more difficult.

Corners and edges In multiple dimensions the possibility exists that different boundaries will intersect. In 2-D, the intersection occurs at corners. In 3-D we have both edges and corners. Some care needs to be exercised at these points to guarantee consistency. In general if a Dirichlet edge intersects a Neumann edge, the

Dirichlet condition takes precedent. If it is Dirichlet and periodic, the periodicity needs to be enforced on that edge (i.e. if it is periodic in the x direction and the bottom edge is Dirichlet, it must also be true that `T(1,1)=T(ni,1)`. In general, the simplest approach is to sketch out the corner points and derive the appropriate corner stencils using a control volume approach. Good luck....

A recipe for implementing commonly occuring Boundary conditions In 2-D, every boundary may need a special condition and if you have to write a special piece of code for each combination of boundary conditions, it can make code maintenance rather messy (but sometimes it is really the only way). In a perfect object-oriented world, we would like some general routines that just `doboundary(myproblem)`. Unfortunately F77 is far from perfect (and definitely not OO). Nevertheless there are a few tricks for allowing you to write relatively general routines that can implement several kinds of boundary conditions with only a few differences in passed parameters. As a test problem, consider some explicit 2-D updating scheme such as

$$T_{i,j}^{n+1} = \begin{bmatrix} & a_5 & \\ a_2 & a_3 & a_4 \\ & a_1 & \end{bmatrix}_{i,j} T_{i,j}^n \tag{8.3.12}$$

which could be a generalized 2-D FTCS diffusion scheme for example (see Section 8.5.1). Now in Fortran we could write this updating scheme for some interior point as

```
      real tp(ni,nj),tn(ni,nj),a(5,ni,nj)
      ...
      im=i-1
      ip=i+1
      jm=j-1
      jp=j+1
      tp(i,j)=a(1,i,j)*tn(i,jm)+a(2,i,j)*tn(im,j)+
     &         a(3,i,j)*tn(i,j)+a(4,i,j)*tn(ip,j)+a(5,i,j)*tn(i,jp)
```

Now the only problem occurs when you get to an edge, for example `i=1` as to how to define `im`. Well this depends somewhat on the Boundary condition. For a Neumann condition, most stencils reduce to reflection conditions and then $T_{0,j} = T_{2,j}$. Therefore, we will get the right answer if we just set `im=2`. Likewise for a periodic boundary, we just wraparound and assume that `im=ni-1`. For a dirichlet condition we can either not update the edges or we can reset the stencil to the identity stencil

$$\mathbf{a}_{1,j} = \begin{bmatrix} & 0 & \\ 0 & 1 & 0 \\ & 0 & \end{bmatrix} \tag{8.3.13}$$

With this approach we can now write a generic 2-D updating algorithm which will do the correct thing when it gets to the boundary. An example for the above problem is

```
      integer ni,nj
      real tp(ni,nj),tn(ni,nj),a(5,ni,nj)
      integer iout(2,2)
```

```
...
do j=1,nj
   jm=j-1
   jp=j+1
   if (j.eq.1) then
      jm=iout(1,2)
   elseif (j.eq.nj)
      jp=iout(2,2)
   endif
   do i=1,ni
      im=i-1
      ip=i+1
      if (i.eq.1) then
         im=iout(1,1)
      elseif (i.eq.ni)
         ip=iout(2,1)
      endif
      tp(i,j)=a(1,i,j)*tn(i,jm)+a(2,i,j)*tn(im,j)+
&           a(3,i,j)*tn(i,j)+a(4,i,j)*tn(ip,j)+a(5,i,j)*tn(i,jp)
   enddo
enddo
return
end
```

All we have to pass it is a small 2-D integer array `iout(2,2)` where `iout` is the index of the point that is outside the boundary. I tend to order my `iout` as `iout(side,dir)` where `dir` is the direction that I would be moving in the grid (above `dir=1` is the `i` direction and `dir=2` is the `j` direction) and `side=1` is the edge in the `dir-1` direction and `side=2` is the edge in the `dir+1` direction. Terribly confusing but maybe a picture will help

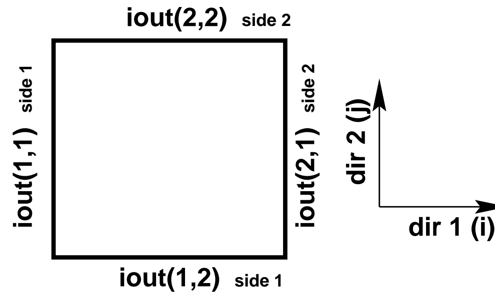

Figure 8.2: A figure trying to explain the ordering of the `iout` array.

8.4 Advection schemes in 2-D

This section will consider some of the commonly used schemes (and their pros and cons) for pure advection problems. In particular we will consider the simplest multi-dimensional advection problem, which is for a constant material derivative

$$\frac{\partial c}{\partial t} + \mathbf{V} \cdot \boldsymbol{\nabla} c = 0 \tag{8.4.1}$$

which (in a perfect world) should just move a set of particles around a flow-field $\mathbf{V}$ without changing the local concentration.

8.4.1 Staggered-Leapfrog - non-conservative form

If we use a staggered leapfrog differencing scheme, which is second-order in time and space, the finite difference approximation to (8.4.1) in 2-D is

$$\frac{c_{i,j}^{n+1} - c_{i,j}^{n-1}}{2\Delta t} = -U_{i,j}\frac{c_{i+1,j}^{n} - c_{i-1,j}^{n}}{2\Delta x} - W_{i,j}\frac{c_{i,j+1}^{n} - c_{i,j-1}^{n}}{2\Delta y} = 0 \tag{8.4.2}$$

or re-arranging, the updating scheme becomes,

$$c_{i,j}^{n+1} = c_{i,j}^{n-1} - \alpha_{i,j}^{x}\left[c_{i+1,j}^{n} - c_{i-1,j}^{n}\right] - \alpha_{i,j}^{y}\left[c_{i,j+1}^{n} - c_{i,j-1}^{n}\right] = 0 \tag{8.4.3}$$

where

$$\begin{aligned} \alpha_{i,j}^{x} &= \frac{U_{i,j}\Delta t}{\Delta x} \\ \alpha_{i,j}^{y} &= \frac{W_{i,j}\Delta t}{\Delta y} \end{aligned} \tag{8.4.4}$$

are the local horizontal and vertical Courant numbers. For 2-D problems, it is often useful to write these sort of equations in "stencil" form which is compact and gives some sense of the spatial differencing. The stencil form of (8.4.3) is

$$c_{i,j}^{n+1} = c_{i,j}^{n-1} - \begin{bmatrix} & \alpha_{i,j}^{y} & \\ -\alpha_{i,j}^{x} & 0 & \alpha_{i,j}^{x} \\ & -\alpha_{i,j}^{y} & \end{bmatrix} c_{i,j}^{n} \tag{8.4.5}$$

Using Neumann stability analysis, it can be shown that the stability requirement for time stepping is the 2-D version of the Courant condition

$$(\alpha_{i,j}^{x})^2 + (\alpha_{i,j}^{y})^2 \leq \frac{1}{2} \tag{8.4.6}$$

(see Numerical Recipes, 2nd ed., p. 846) which for a square grid ($\Delta x = \Delta y$) becomes

$$\Delta t \leq \frac{\Delta}{\sqrt{2}|\mathbf{V}_{max}|} \tag{8.4.7}$$

where $\mathbf{V}_{max}$ is the maximum velocity on the grid. In n dimensions, this result can be generalized to

$$\Delta t \leq \frac{\Delta}{\sqrt{n}|\mathbf{V}_{max}|} \tag{8.4.8}$$

The physical, meaning of the Courant condition is still the same. Roughly speaking, for simple Eulerian schemes, you can't let individual particles move more than a grid-space during a time-step.

8.4.2 Staggered-Leapfrog - conservative form

The differencing scheme in the previous problem will work for many problems, however, it is not a flux-conservative scheme in the sense that the material that

leaves one cell is identical to the material that enters the next. Many times it is better to consider the flux conservative version of the problem.

$$\frac{\partial c}{\partial t} + \boldsymbol{\nabla}\cdot [c\mathbf{V}] = 0 \tag{8.4.9}$$

which for incompressible flows ($\boldsymbol{\nabla}\cdot \mathbf{V} = 0$) should reduce analytically to (8.4.1). For, flux-conservative problems, the control volume approach is a natural way to produce finite-difference approximations. To get centered-time, centered space staggered leapfrog scheme for (8.4.10) we simply use a centered time approximation for the time derivative and use Eq. (8.3.5) (or 8.3.7) with $\mathbf{F}(\mathbf{x}) = c(\mathbf{x})\mathbf{V}(\mathbf{x})$. The only trick is how to calculate the fluxes at the half-grid points. If **V** is known analytically everywhere, then one approach, is to store the velocity values on the *staggered-mesh* which lies on the half-grid points (the X's in figure 8.1) and to use the average (linear interpolation) of the nearest nodes for the concentrations at the half-points. For example, the horizontal flux at $i + 1/2, j$ would be

$$F^x_{i+1/2,j} = U_{i+1/2,j}\frac{c_{i+1,j} + c_{i,j}}{2} \tag{8.4.10}$$

Expanding the rest of the fluxes in a similar manner and collecting terms, the flux-conservative staggered leapfrog scheme in stencil form looks like,

$$c^{n+1}_{i,j} = c^{n-1}_{i,j} - \begin{bmatrix} & \alpha^y_{i,j+1/2} & \\ -\alpha^x_{i-1/2,j} & \sum & \alpha^x_{i+1/2,j} \\ & -\alpha^y_{i,j-1/2} & \end{bmatrix} c^n_{i,j} \tag{8.4.11}$$

where α^x and α^y are still given by (8.4.4) but use the velocities at the half-nodes, and

$$\sum = \alpha^x_{i+1/2,j} - \alpha^x_{i-1/2,j} + \alpha^y_{i,j+1/2} - \alpha^y_{i,j-1/2} \tag{8.4.12}$$

Note for constant α's (or simple incompressible flows) then $\sum = 0$ and the conservative scheme reduces to the standard centered space scheme. The stability requirements for this differencing scheme are again the n-dimensional Courant condition.

8.4.3 2-D Upwind and MPDATA

By simply changing the definitions of the fluxes at the half-nodes, we can generate a whole family of different difference schemes. Using higher order interpolation schemes may give higher order difference schemes. One approach that produces a lower order scheme is the multi-dimensional *upwind-difference* or *donor cell* schemes. These schemes are exactly the same as in 1-D but we now have to calculate the donor cell for both the horizontal and vertical fluxes. If we define the donor flux between two cells at points (i, j) and $(i + 1, j)$ as

$$F(c_{i,j}, c_{i+1,j}, U_{i+1/2,j}) = \max(0., U_{i+1/2,j})c_{i,j} + \min(0., U_{i+1/2,j})c_{i+1,j} \tag{8.4.13}$$

then the general first-order upwind scheme can be written

$$
\begin{aligned}
c_{i,j}^{n+1} = c_{i,j}^{n} - \\
&\frac{\Delta t}{\Delta x}\left[F(c_{i,j}, c_{i+1,j}, U_{i+1/2,j}) - F(c_{i-1,j}, c_{i,j}, U_{i-1/2,j})\right] + \\
&\frac{\Delta t}{\Delta y}\left[F(c_{i,j}, c_{i,j+1}, V_{i,j+1/2}) - F(c_{i,j-1}, c_{i,j}, V_{i,j-1/2})\right]
\end{aligned}
\quad (8.4.14)
$$

For efficiency, the $\Delta t/\Delta x$ terms can be absorbed into the velocity arrays to produce an array of local Courant numbers. In addition, because this scheme is automatically flux-conservative, you only need to calculate the donor-flux at a half point once because what leaves cell i through the face at $i + 1/2$, identically enters cell $i + 1$ (you can verify this symmetry yourself). This can speed up the calculation by at least a factor of 2 (and is how it is implemented in `MPDATA`). As usual, these simplest donor-cell schemes are highly stable but also highly diffusive for time steps smaller than the Courant limit (see below). By approximating the implicit numerical diffusion and correcting for it, however, a 2-D version of `MPDATA` (Smolarkiewicz, 1986), can produce very small implicit diffusion (at the expense of time). The version of the code that appears in the problem set also has a very sophisticated anti-dispersion correction. These additional corrections, however can cause difficulties in implementing non-dirichlet boundary conditions.

8.4.4 Semi-Lagrangian Schemes

In addition to Eulerian schemes, the semi-Lagrangian schemes discussed in Section 5 are also readily extended to multiple dimensions. The overall scheme is the same as in 1-D, i.e. we first seek the trajectory of the characteristic that passes through a grid-point and then use high order interpolation to move the value forward and integrate any source terms along the trajectory. The only slight change from the 1-D codes is that we now need to do interpolation in multiple dimensions. Fortunately, as discussed in Numerical Recipes, for regular grids, this only requires doing a series of sequential 1-D polynomial interpolations in each of the dimensions. As an example Figure 8.3 shows schematically bicubic interpolation in two-dimensions which requires knowing the value of the function at the 16 nearest neighbors. Because of the considerable expense of interpolation, these schemes are quite expensive relative to staggered leapfrog for the same Courant condition. However, the beauty of these schemes is that they have no Courant stability limit and can often pay for themselves handsomely with larger time steps.

8.4.5 Pseudo-spectral schemes

I don't like pseudo-spectral schemes.... But they're not any harder (or faster) in 2-D than one. The biggest issues are sorting out the wavenumber storage schemes for multi-dimensional FFT's. But otherwise, the problem is the same but now you need to take FFT's in each direction to work out the derivatives in each direction. Thus something like $\boldsymbol{\nabla} f$ requires 4 2-D FFT's, i.e. $\partial f/\partial x$ requires one forward and one inverse 2-D FFT and $\partial f/\partial y$ requires two.

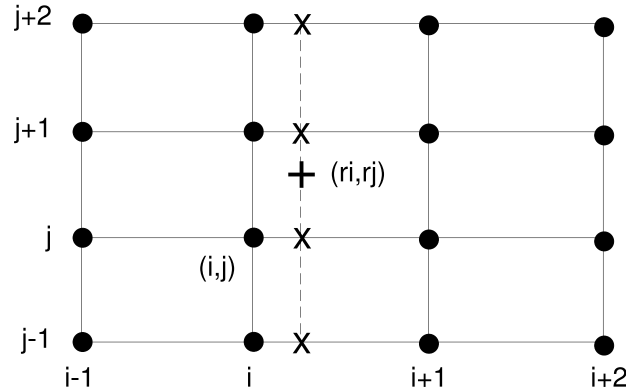

Figure 8.3: Diagram showing bicubic interpolation. Given a regular 2-D grid of values (note Δx not necessarily $= \Delta y$) we wish to find the value of the interpolated function at some intermediate point (ri, rj). As shown here, first do a horizontal interpolation along each of the j lines to find the interpolated values at the X's, then do a vertical interpolation along the line at ri to find the value at the $+$. Because of the symmetry of the interpolation operator, you will get the same answer if you interpolate vertically first then horizontally. Also because of the properties of the interpolating polynomial, the function will always be exact at the grid-points.

8.4.6 Some example tests–2-D

In one dimension, an incompressible flow ($\nabla \cdot \mathbf{V} = 0$) is synonymous with a constant velocity field. In multiple dimensions, however, the constraint of incompressibility is rather weak. It only states that the net flux into a region must equal the outgoing flux but places no constraints on the direction or magnitude of the fluxes. Thus there are an infinite number of incompressible flows and they can have quite different numerical effects. Here we will test the advection schemes against two simple flow fields: rigid body rotation, and a single shear cell (Figure 8.4).

Rigid body rotation in the clockwise direction, around some point (x_0, y_0) is given by

$$\mathbf{V}(x, y) = (y - y_0)\mathbf{i} - (x - x_0)\mathbf{j} \tag{8.4.15}$$

(Figure 8.4a). If you are so inclined, verify that $\nabla \cdot \mathbf{V} = 0$. The streamlines for this field are simply circles centered on (x_0, y_0) and there is no shear between streamlines. In a perfect advection scheme, any initial condition should simply rotate around the center point without changing shape so that after one rotation, the initial condition and final condition would be indistinguishable. Unfortunately, there are no perfect advection schemes, only reasonable compromises. Figure 8.5 shows the behaviour of flux conservative staggered leapfrog, upwind-differencing and several levels of corrections of `MPDATA` for this test using an initial condition of a Gaussian of amplitude 2. Each panel shows contours of the solution at times corresponding to increments of a quarter rotation up to a full rotation. The grid in Figures 8.5a–e is 65×65 points, Figures 8.5f has 129×129 grid points. In general, the two dimensional `mpdata` can be fairly accurate but is 20-40 times more expensive than staggered-leapfrog. For the rigid rotation test, simply halving

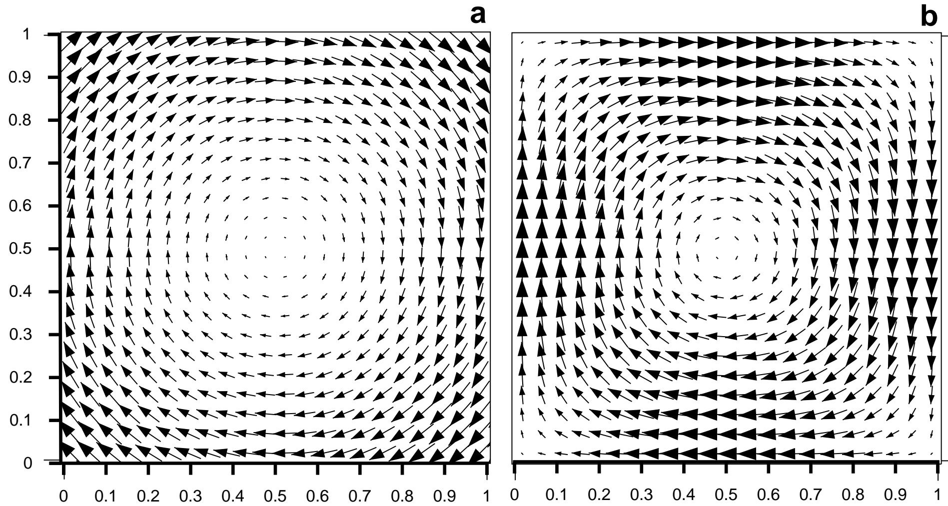

Figure 8.4: Vector plots showing advective velocity fields for (a) rigid-body rotation and (b) a single shear cell. The maximum velocity in 8.4a is 0.71, and is 1. in 8.4b.

the grid-spacing gets rid of most of the dispersion in the lower resolution staggered-leapfrog problem with only the standard factor of 8 increase in time (4 times more points plus 2 times more time steps due to the Courant condition). Figure 8.7a,b shows the results of the rigid body rotation test for the semi-Lagrangian schemes. For this test, these schemes are comparable in time and accuracy to the simplest staggered-leapfrog scheme *if a large Courant number is used.* These schemes preserve the shape of the advected quantity better because the full interpolation used gets rid of the grid anisotropy inherent in the Eulerian schemes. Table 8.1 compares times and accuracy for the various schemes.

The rigid body rotation test is often used for testing numerical advection schemes, however it rarely occurs in nature. In particular it has the peculiar property of having no shear. Much more often, fluid dynamic flows have significant vorticity which can shear and stretch simple initial conditions into a horrific mess. For grid-based schemes, this shearing (without any real diffusion to balance it) can emphasize numerical artifacts. To see this we will consider a single shear cell on the unit square which has the streamfunction

$$\psi(x,y) = -\frac{1}{\pi}\sin(\pi x)\sin(\pi y) \tag{8.4.16}$$

with a corresponding velocity field

$$\mathbf{V} = \boldsymbol{\nabla}\times\psi(x,y)\mathbf{k} = -\sin(\pi x)\cos(\pi y)\mathbf{i} + \cos(\pi x)\sin(\pi y)\mathbf{j} \tag{8.4.17}$$

(see Figure 8.4b). This flow field appears qualitatively similar to the rigid rotation but local vorticity in this flow can be quite large and will cause significant numerical problems. Figure 8.6 demonstrates the behaviour of the same gaussian initial condition as in Fig. 8.4 after half a turn and a whole turn. In a perfect world, the more rapid rotation near the center of the box will cause the initial condition to

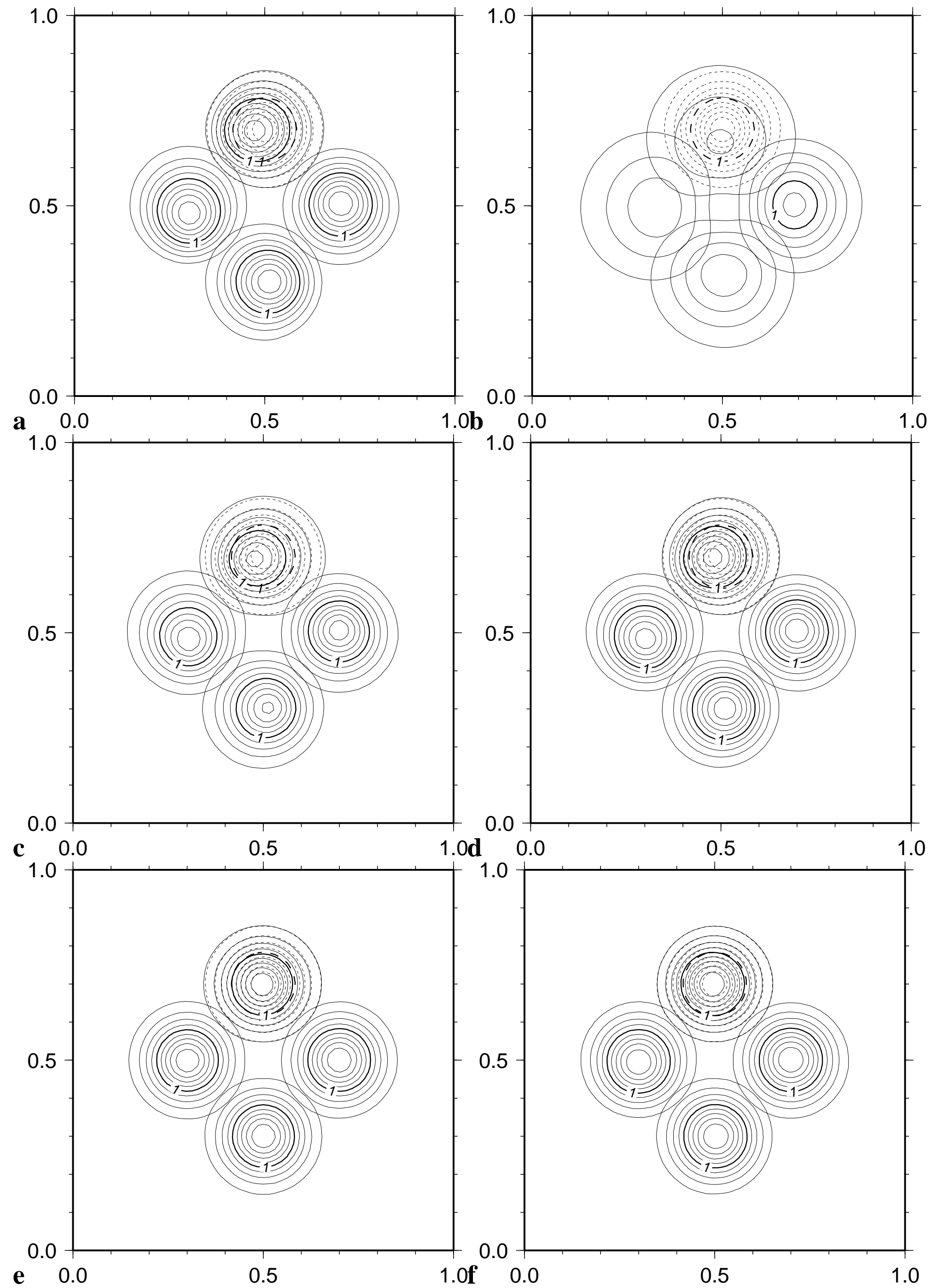

Figure 8.5: Results of the rigid body rotation test for a variety of methods. In each panel, the original gaussian initial condition is shown by dotted contours with the solution at subsequent quarter rotations shown in solid contours up to one full rotation. The direction of rotation is clockwise. The initial condition has a maximum value of 2, the contour interval is 0.2. Figures a–e have 65×65 grid points while f is 129×129. Timing and accuracy results are given in Table 8.1 (a) staggered-leapfrog showing small dispersion (b) upwind donor-cell is really diffusive (c) `mpdata` 1 correction (d) `mpdata` 2 corrections (e) `mpdata` 2 corrections + 3rd order anti-dispersive correction (f) staggered-leapfrog double resolution.

Table 8.1: Comparison of speed and accuracy for the rigid rotation and shear-cell advection tests. Runtime, min and max values are all for a dimensionless time of $t' = 2\pi$. All comparisons were made on a 140 Mhz SunUltra140e with optimization flags -fast -O4.

Rigid Body rotation test						
Method	Grid points	α	nsteps	runtime (s)	c_{min}	c_{max}
Staggered Leapfrog	65×65	1.	804	0.4	$-.0625$	1.925
	129×129	1.	1608	3.0	-5.74e-5	1.992
MPDATA (ncor=0)	65×65	1.	804	1.1	0.	0.6249
ncor=1	65×65	1.	804	4.1	0.	1.675
ncor=2	65×65	1.	804	7.0	0.	1.899
ncor=2, i3rd=1	65×65	1.	804	12.5	0.	1.986
Semi-Lagrangian	65×65	4.	201	0.4	-1.228e-4	1.918
	129×129	8.	201	1.9	-6.56e-3	1.992
Shear Cell test						
Method	Grid points	α	nsteps	runtime (s)	c_{min}	c_{max}
Staggered Leapfrog	65×65	1.	568	0.3	-1.122	1.541
	129×129	1.	1137	2.7	-0.564	1.838
MPDATA (ncor=0)	65×65	1.	568	0.8	0.	0.2828
ncor=1	65×65	1.	568	2.9	0.	0.9009
ncor=2	65×65	1.	568	5.0	0.	1.203
ncor=2, i3rd=1	65×65	1.	568	8.9	0.	1.182
Semi-Lagrangian	65×65	4.	142	0.3	-5.768e-2	1.586
	129×129	8.	142	1.7	-2.384e-2	1.925

get sheared into a paisely pattern, however, the maximum value would remain 2. (i.e. the characteristics solution says that the local concentration must remain constant even during shearing). Unfortunately, the shearing causes excessive numerical dispersion in staggered-leapfrog schemes and excessive diffusion in upwind and `mpdata` schemes. These numerical artifacts are always most severe for components of the solution that vary on a length-scale comparable to the grid spacing. The principal problem here is that the constant shearing always tends to sharpen the frequency content of the solution and makes well resolved regions thin and poorly resolved. If your problem includes a real diffusion term, the difficulties are much less severe because diffusion naturally keeps the solution spread over the grid (e.g. See convection figures in Chapter 2). Without real diffusion, however, the simplest grid-based schemes are really not adequate; however, the semi-Lagrangian schemes are a significant improvement (Fig. 8.7).

8.4.7 Other approaches- particle based fully-Lagrangian schemes

In addition to grid-based methods, it is always possible to use particle based fully *Lagrangian schemes* that are based on the method of characteristics. The big drawback to these methods is that even if you start out with a homogeneous or regular distribution of points, after any significant amount of time they can be spread out in a very irregular distribution which becomes difficult to visualize (e.g. contour) or to use for further calculations at points that do not contain any particles.

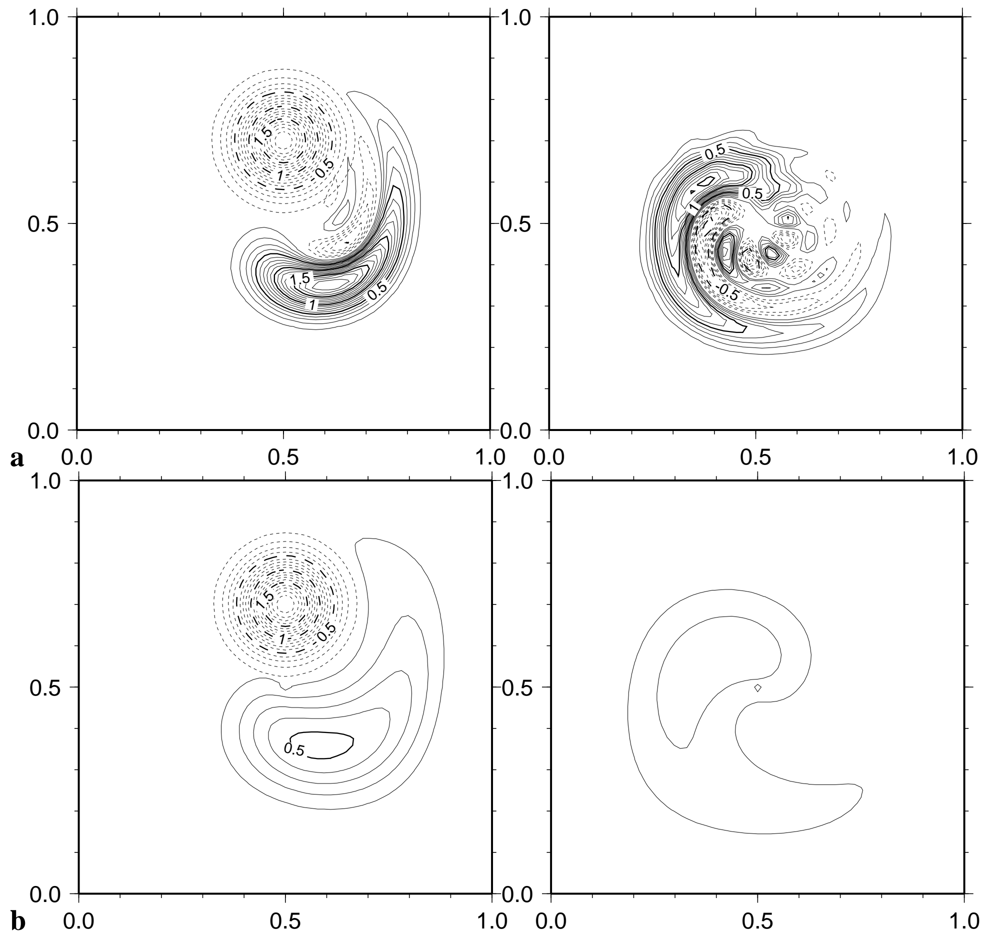

Figure 8.6: Results of the shear cell rotation test for a variety of methods. For each method, the first panel shows the original gaussian initial condition (dotted contours) and the solution at a half rotation (solid contours). The second panel shows the solution after a full rotation. The direction of rotation is clockwise. The initial condition has a maximum value of 2 and the contour interval is 0.1. Figures a–c have 65×65 grid points while d is 129×129. Accuracy and timing information is given in Table 8.1 (a) staggered-leapfrog showing significant dispersion (yuck) (b) upwind donor-cell is hugely diffusive.

More sophisticated particle based methods, however, attempt to remedy these problems without losing the fundamental physics of the underlying characteristics. Most of these methods are a bit beyond the scope of this course but they are worth noting. The first set of methods is known as *contour surgery* which instead of discretizing a continuous field into an arbitrary set of points, it first contours the initial field and then discretizes each contour level into a fixed set of points. Thus a single contour level forms a *marker chain* of points that gets advected around together. The beauty of contour surgery is that by simply plotting each contour at any time provides an instant contour map of the field without any need for interpolation. The only real trick to contour surgery is the surgery part, i.e. selectively removing long wispy tendrils that get sheared out but are no longer resolved. An introduction to contour surgery and a leaping off place for references is the Physics Today (1993)

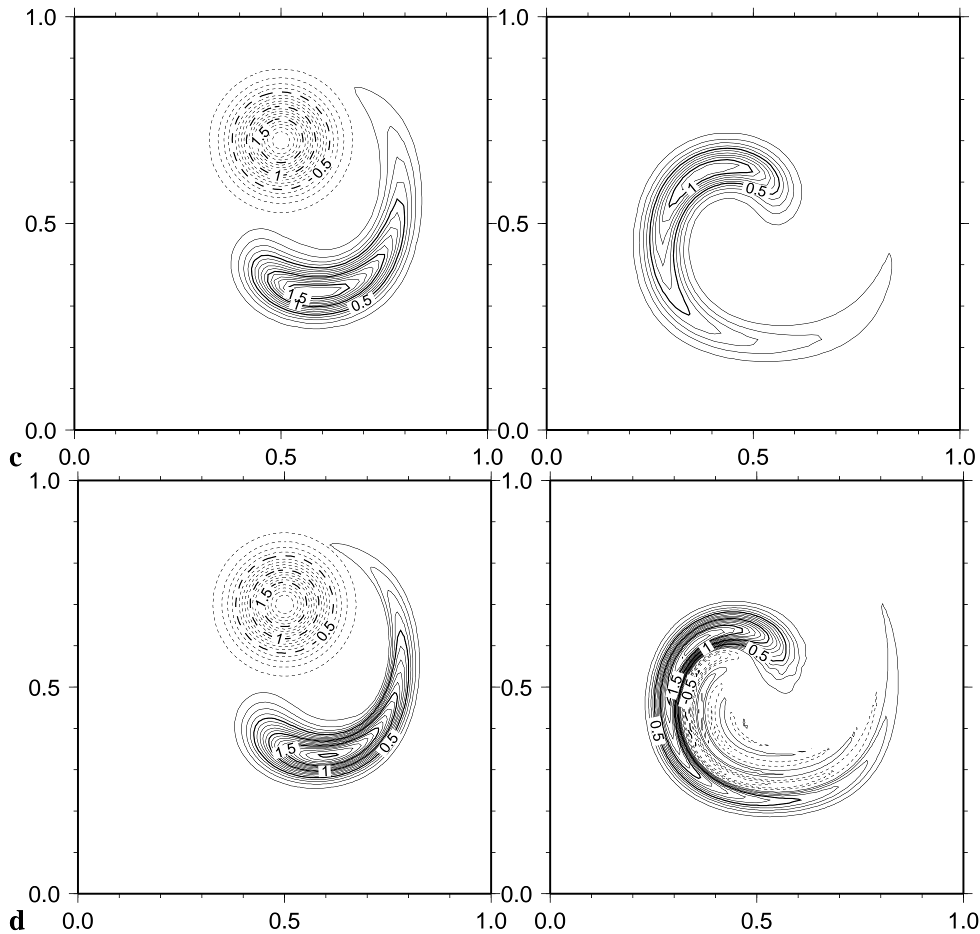

Figure 8.6 contd.: (c) even the best `mpdata` scheme with 2 corrections + 3rd order anti-dispersive correction shows significant diffusion (d) staggered-leapfrog double resolution has noticeable dispersion but might be reasonable for short runs and is cheap and easy.

article by Dritschel and Legras [1] .

Another approach that allows for very smooth interpolation between randomly placed Lagrangian particles is the *Natural-Element-Method* (Braun and Sambridge [2]). This approach leads into the hairy world of unstructured grids but may show considerable promise for certain classes of problems.

8.5 Diffusion schemes in 2-D

The difficulties engendered in multi-dimensional advection problems, stem primarily from the additional degrees of freedom that the advective flow can achieve (it is also this new physical behaviour that is of most interest to solve). In particular, the additional shear produced going from 1-D to n-D can cause great grief in multi-dimensional grid based methods. Diffusion dominated problems tend to be much better behaved in multi-dimensions, however, because while n-dimensional

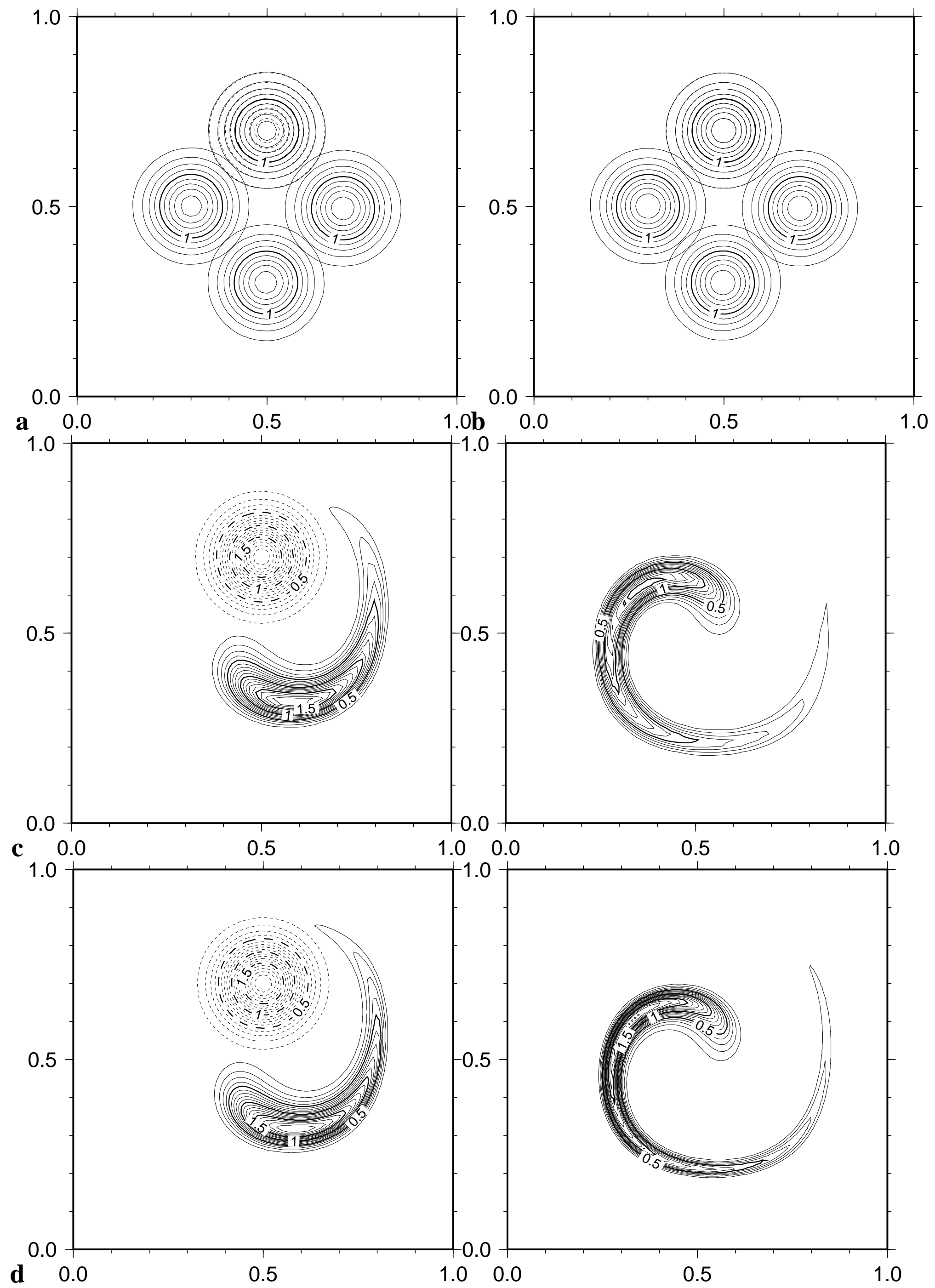

Figure 8.7: Showing off the semi-Lagrangian solvers for rigid body and shear cell tests. (*a*) Rigid body test on a 65×65 grid. (*b*) Rigid body test on 129×129 grid. (*c*) Shear test, 65×65 grid. (*d*) shear test, 129×129. In particular for the high-resolution shear test, the semi-Lagrangian schemes really shine.

diffusion will often be faster because there is more available surface area, the basic physical behaviour does not change significantly. For simplicity here, we will just consider simple differencing schemes for the canonical diffusion equation

$$\frac{\partial T}{\partial t} = \nabla^2 T \tag{8.5.1}$$

as well as some simple extensions to the more general diffusion problem

$$\frac{\partial T}{\partial t} = \boldsymbol{\nabla} \cdot \kappa \boldsymbol{\nabla} T \tag{8.5.2}$$

8.5.1 Explicit FTCS

The basic behaviour of diffusion in multi-dimensions is readily seen (and solved) using an explicit FTCS scheme which is almost identical to its 1-D counterpart. If we use a simple forward-time approximation for the LHS of (8.5.1) together with Eqs. (8.3.4) for the 2nd-order, 5-point Laplacian operator we get.

$$\frac{T_{i,j}^{n+1} - T_{i,j}^n}{\Delta t} = \frac{T_{i+1,j}^n - 2T_{i,j}^n + T_{i-1,j}^n}{\Delta x^2} + \frac{T_{i,j+1}^n - 2T_{i,j}^n + T_{i,j-1}^n}{\Delta y^2} + O(\Delta^2) \tag{8.5.3}$$

or re-arranging into stencil form, the FTCS updating scheme becomes

$$T^{n+1} = \begin{bmatrix} & \beta_y & \\ \beta_x & [1 - 2(\beta_x + \beta_y)] & \beta_x \\ & \beta_y & \end{bmatrix} T^n \tag{8.5.4}$$

where

$$\beta_x = \frac{\Delta t}{(\Delta x)^2}, \qquad \beta_y = \frac{\Delta t}{(\Delta y)^2} \tag{8.5.5}$$

For a uniform square grid spacing $\Delta x = \Delta y$ this becomes

$$T^{n+1} = \begin{bmatrix} & \beta & \\ \beta & [1 - 4\beta] & \beta \\ & \beta & \end{bmatrix} T^n \tag{8.5.6}$$

In the case of a non-constant diffusivity, a control volume approach gives the more general approximation to (8.5.2) for a uniform grid as

$$T^{n+1} = \beta \begin{bmatrix} & \kappa_{i,j+1/2} & \\ \kappa_{i-1/2,j} & [1/\beta - \sum] & \kappa_{i+1/2,j} \\ & \kappa_{i,j-1/2} & \end{bmatrix} T^n \tag{8.5.7}$$

where $\kappa_{i-1/2,j}$ is the diffusivity on the staggered grid point $(i - 1/2, j)$ and

$$\sum = \kappa_{i+1/2,j} + \kappa_{i-1/2,j} + \kappa_{i,j+1/2} + \kappa_{i,j-1/2} \tag{8.5.8}$$

The physical interpretation of the FTCS diffusion operator in Eq. (8.5.6) is that it is a simple smoothing operation that replaces any point $T_{i,j}$ with some fraction of

its original value plus the β times the sum of its nearest neighbors. Since diffusion can only map a positive quantity into a positive quantity, the scheme is only valid if $\beta < 1/4$ or more generally

$$\Delta t < \frac{(\Delta x)^2(\Delta y)^2}{2[(\Delta x)^2 + (\Delta y)^2]} \tag{8.5.9}$$

Thus, roughly speaking, a decrease in grid spacing by a factor of 2 in both directions requires a factor of 16 more time to compute (4 times as many points with 4 times the number of time steps). Unfortunately, because the simplest centered schemes are only second order in space (and first order in time), the order of magnitude more effort only gains you a factor of 4 in reducing the truncation error. This is the usual drawback with explicit schemes (although it is sufficiently easy to code that for quick and dirty problems, the actual development and debugging time outweighs the run time). In one-dimension the answer was to go to a fully implicit or crank nicholson scheme which took advantage of rapid tridiagonal solvers. Unfortunately, in multi-dimensions, the Crank-Nicholson scheme is no longer tridiagonal. E.g. in 2-D it looks like

$$\begin{bmatrix} & -1 & \\ -\alpha^2 & [R + 2(1+\alpha^2)] & -\alpha^2 \\ & -1 & \end{bmatrix} T^{n+1} = \begin{bmatrix} & 1 & \\ \alpha^2 & [R - 2(1+\alpha^2)] & \alpha^2 \\ & 1 & \end{bmatrix} T^n \tag{8.5.10}$$

where $R = 2\Delta y^2/\Delta t$ and $\alpha = \Delta y/\Delta x$ is the aspect ratio of the grid spacing. Equation (8.5.10) is a penta-diagonal system of simultaneous linear equations

$$\mathsf{A}\mathbf{T}^{n+1} = \mathbf{r} \tag{8.5.11}$$

and the necessary inversion of matrix A is significantly more expensive computationally unless you use a rather sophisticated algorithm such as Multi-grid methods or Fast elliptic solvers (if the problem permits). These problems are more related to the issues of Multi-dimensional boundary value problems and we will pick them up in detail in Chapter 9. However, here we will discuss one another approach which combines second order accuracy in space and time with the ease of tridiagonal solvers.

8.5.2 ADI: Alternating-Direction Implicit Schemes

ADI schemes are another example of operator splitting but with a slightly different meaning. Instead of splitting the operator into different processes such as advection and diffusion, ADI schemes, split one process into its different directional components. For example, In the case of the multidimensional diffusion equation, we could rewrite Eq. (8.5.1) as

$$\frac{\partial T}{\partial t} = \mathcal{L}_x T + \mathcal{L}_y T \tag{8.5.12}$$

where $\mathcal{L}_x$ is the operator controlling diffusion in the horizontal direction and $\mathcal{L}_y$ controls diffusion in the vertical. Given this splitting, ADI schemes then solve

(8.5.12) by taking two-passes, first solving an implicit diffusion equation in the horizontal for half a time step and then an implicit diffusion equation in the vertical. More physically, the ADI scheme really pretends that the problem is first a set of 1-D diffusion equations in one direction and then a set of 1-D equations in the other direction. You may ask why this seems worth all the effort, but the answer is simply that 1-D problems are tridiagonal and we can use all the machinery we've already developed.

In more detail, the specific ADI algorithm for Eq. (8.5.1) looks like

$$\frac{T_{i,j}^{n+1/2} - T_{i,j}^{n}}{\Delta t/2} = \frac{1}{\Delta x^2}\begin{bmatrix} 1 & -2 & 1 \end{bmatrix} T^{n+1/2} + \frac{1}{\Delta y^2}\begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} T^{n} \quad (8.5.13)$$

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n+1/2}}{\Delta t/2} = \frac{1}{\Delta x^2}\begin{bmatrix} 1 & -2 & 1 \end{bmatrix} T^{n+1/2} + \frac{1}{\Delta y^2}\begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} T^{n+1} \quad (8.5.14)$$

where a horizontal stencil implies horizontal nearest neighbors and vertical stencils imply vertical neighbors. Equation (8.5.13) is an implicit, tridiagonal equation for the horizontal rows at time $n + 1/2$ which are then used in (8.5.14) to update the vertical columns at time $n + 1$. The tridiagonal nature of these two schemes can be made more apparent if we collect terms of the same time step together. For a uniform grid with $\Delta x = \Delta y = \Delta$ it is also useful to define the parameter $R = 2\Delta^2/\Delta t$. Thus Eqs. (8.5.13)–(8.5.14) can be rewritten as

$$\begin{bmatrix} -1 & R+2 & -1 \end{bmatrix} T^{n+1/2} = \begin{bmatrix} 1 \\ R-2 \\ 1 \end{bmatrix} T^{n} \quad (8.5.15)$$

$$\begin{bmatrix} -1 \\ R+2 \\ -1 \end{bmatrix} T^{n+1} = \begin{bmatrix} 1 & R-2 & 1 \end{bmatrix} T^{n+1/2} \quad (8.5.16)$$

The solution scheme is then to invert the tridiagonal equation (8.5.15) for $T^{n+1/2}$ using a routine such as `TRIDAG` from Numerical Recipes, then use this new solution as the RHS of (8.5.16) and invert again for T^{n+1}. Numerical recipes, first version presents a few more tricks to improve the efficiency of this routine. Unfortunately, to get significant performance improvements on parallel or vector machines requires some rethinking of the simplest algorithm. If anyone is interested, I have a high performance, ADI solver for general 2-D operators.

8.5.3 Some diffusion examples 2-D

A convenient test problem for diffusion solvers, is the evolution of a gaussian initial condition with time. If we have a d-dimensional gaussian initial condition of amplitude A, "width" σ and peak located at some point $\mathbf{x}_0$

$$T(\mathbf{x}) = A \exp\left[\frac{(\mathbf{x} - \mathbf{x}_0)\cdot(\mathbf{x} - \mathbf{x}_0)}{\sigma^2}\right] \quad (8.5.17)$$

then Eq. (8.5.1) has the solution (in an infinite medium) of

$$T(\mathbf{x},t) = \frac{A}{(1+4t/\sigma^2)^{d/2}} \exp\left[\frac{(\mathbf{x}-\mathbf{x}_0)\cdot(\mathbf{x}-\mathbf{x}_0)}{\sigma^2+4t}\right] \tag{8.5.18}$$

Thus the gaussian gets wider and flatter with time. Figure 8.8 shows the numerical solution for this initial condition using a FTCS scheme and Figure 8.9 shows the error from the true solution (normalized to the maximum value). In general, the agreement is quite good with the maximum error less than 3×10^{-4} at early times. At later times the error becomes large at the boundaries where the numerical scheme assumes Neumann, (no-flux) conditions. For stability, this problem requires 400 time steps on a 65×65 grid. While this problem only took about 1 second (on a SparcStation10), a factor of 2 grid refinement would take ~ 16 seconds. Using an ADI scheme, can significantly reduce the number of time steps required (in fact the accuracy of the ADI scheme improves for longer time steps). The ADI solution to the same problem with only 20 time steps produces a plot that is indistinguishable from Figure 8.8. When we look at the details of the error structure (Fig. 8.10) we find that the ADI scheme has about 5 times larger errors (maximum error $\sim 10^{-3}$), however they are more evenly distributed across the solution. It's interesting to note that although the solution contours are apparently circular, the error shows a distinct anisotropy which derives from the underlying grid. If the grid spacings were not equal in each direction, additional grid errors might intrude. Nevertheless, diffusion is a sufficiently stable process, that the end results are nearly always the same independent of the method. The choice of solution method for pure diffusion problems should rest with the amount of effort that is required to get the answer you want. Always remember, *the time you save may be your own*.

8.6 Combined Advection-Diffusion and operator splitting

If your problem contains both advection and diffusion (as is common), then we cannot necessarily be so blasé about the schemes we choose as schemes that are stable for one process need not be stable for another. For example FTCS schemes are stable for diffusion problems but not for advection problems. As long as diffusion is dominant everywhere, it still may be possible to use a FTCS scheme as a quick and dirty solver, however, in regions where advection is dominant, the inherent negative diffusion in these schemes can make the solutions grow or become unstable. Better combined schemes can be solved by adding the advective terms to the matrices in an ADI scheme. I have used this approach several times and it works reasonably well for diffusion dominated schemes. However, in general, the asymmetric nature of advection operators, tend to destabilize diffusion schemes and limit the range of time steps that can be taken. In addition, different differencing schemes for advection and diffusion can have various amounts of numerical dispersion or diffusion, particularly in cross derivatives.

One approach, that appears useful for developing accurate combined advection diffusion schemes is to consider linear combinations of different differencing

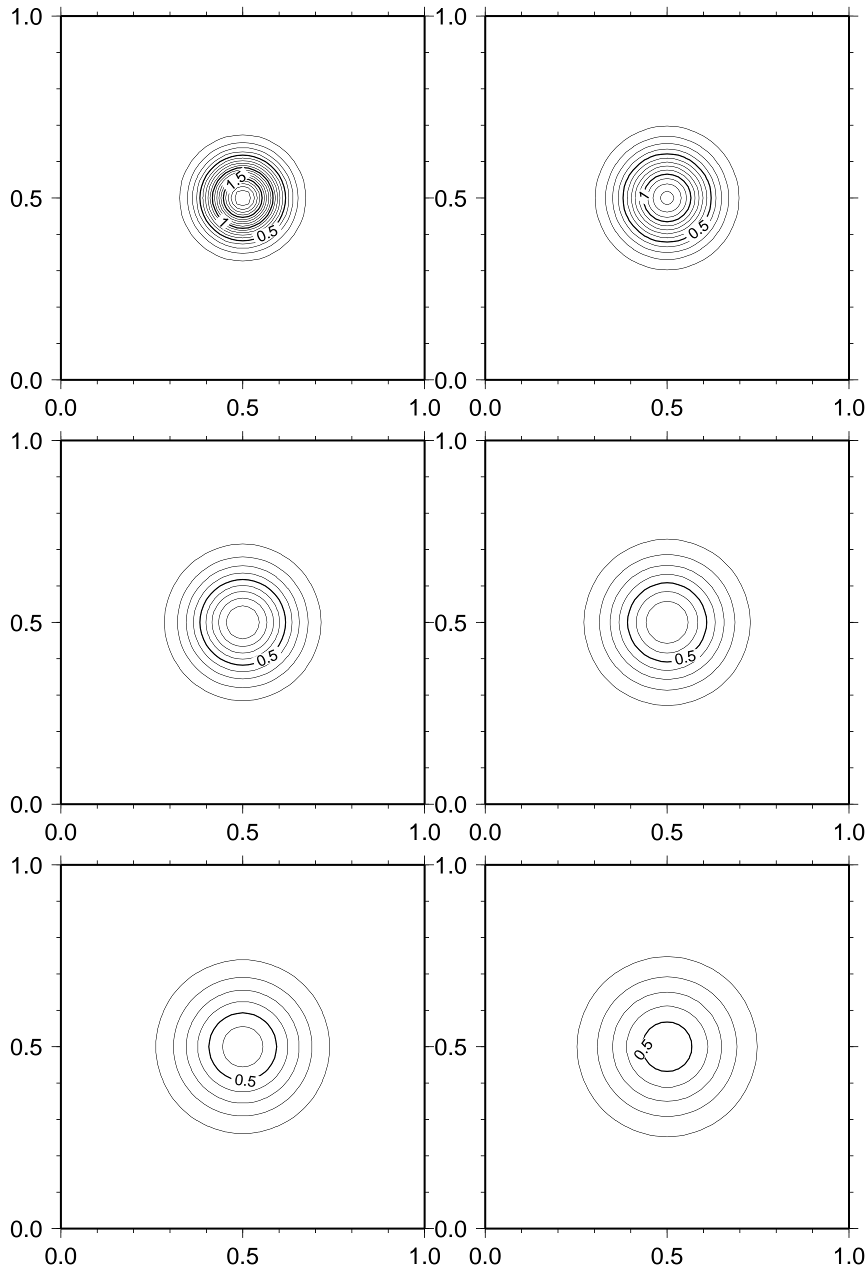

Figure 8.8: Contour plots showing the diffusion of a gaussian initial condition. This plot uses a FTCS explicit scheme on a 65×65 grid with no-flux boundary conditions on all 4 walls. Stability of this scheme requires a small time step and this run uses 400 steps (although the elapsed time is ~ 1 second). A much faster ADI scheme, can achieve results that are indistinguishable to the eye in 20 steps.

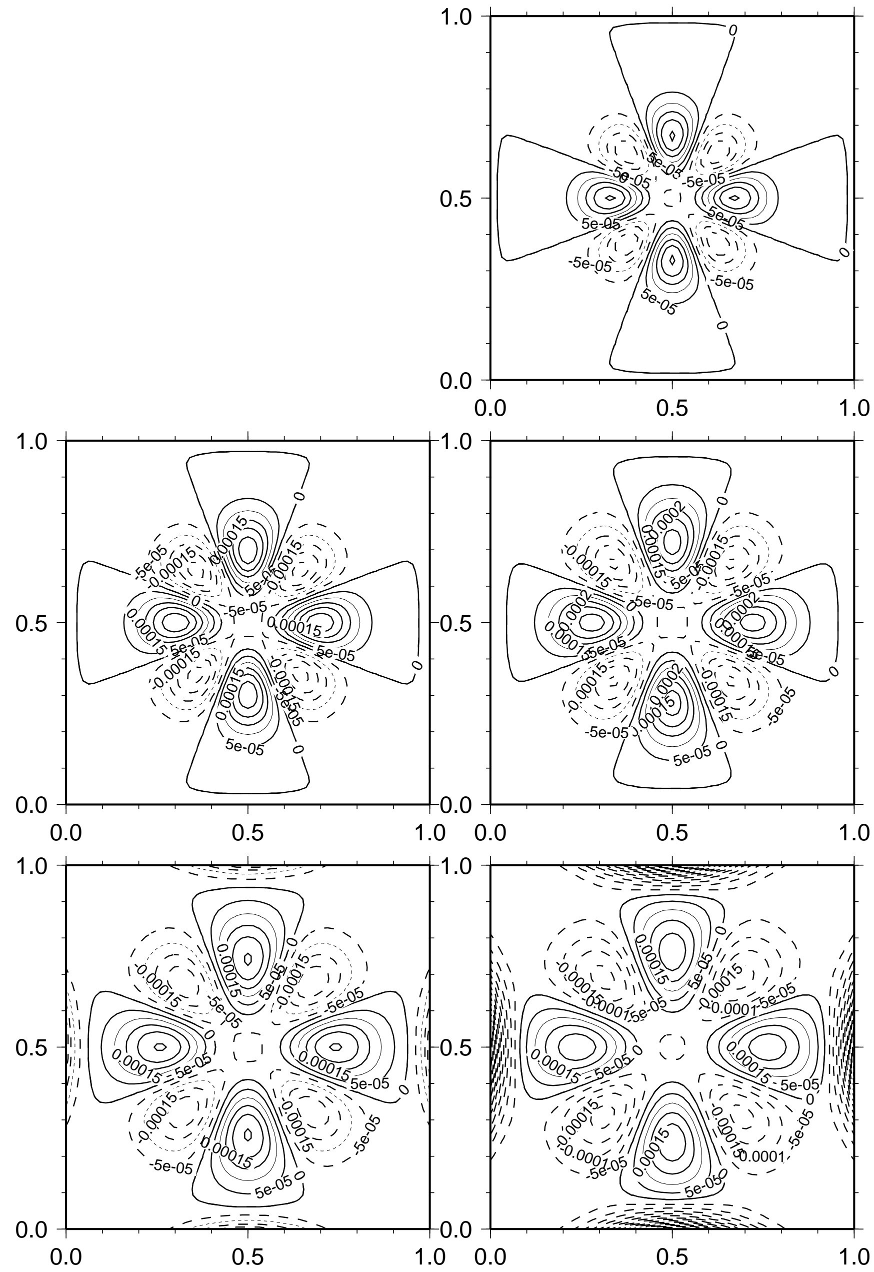

Figure 8.9: Contour plots showing the error from the true solution for figure 8.8 and the FTCS scheme. The maximum error in this scheme (not counting the boundaries) is about 2×10^{-4}. Note the symmetry of the grids that is apparent in the error plots.

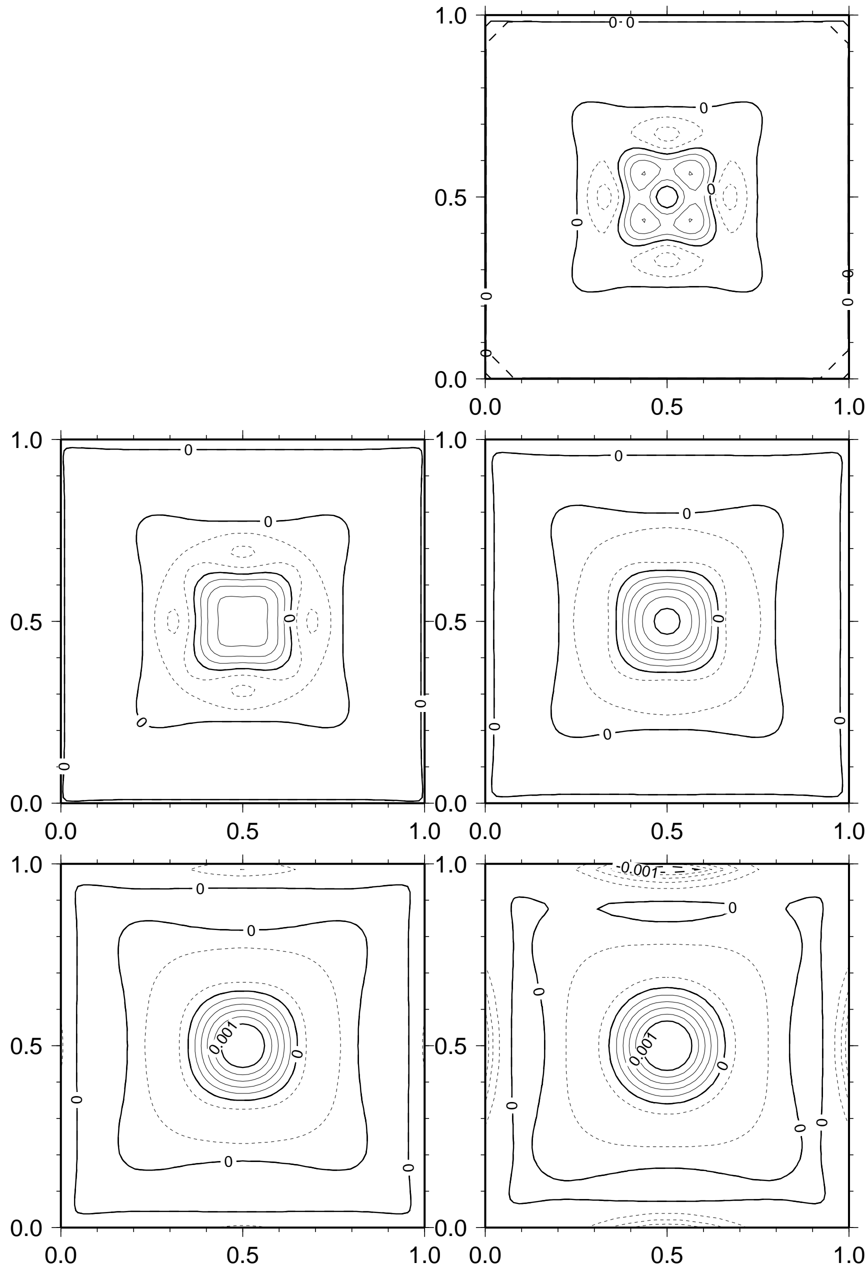

Figure 8.10: Contour plots showing the error from the true solution for figure 8.8 and the ADI scheme. The maximum error in this scheme (not counting the boundaries) is about 1×10^{-3}. The error in this scheme tends to be larger, but more evenly distributed (and more than accurate enough for jazz).

schemes and find the weights that minimize certain aspects of the truncation error. Noye and Tan, (1989) present a whole family of (slightly inscrutable) finite difference schemes for combined advection and diffusion problems and demonstrate their stability and accuracy. It should be noted that their tests, however, are just for a constant advective velocity with no shear and considerable diffusion. It is not clear to me how these schemes will hold up to more general advective fields.

Another solution (or hack) is *operator splitting* where we sequentially apply different updating schemes that are stable for each of the processes individually. For example, we could use an explicit staggered leapfrog scheme for the advection terms then a bit of diffusion using an ADI scheme. Alternatively, for strongly advection dominated flows we could use a semi-Lagrangian scheme for advection followed by ADI. Currently my favorite technique however is the two-dimensional version of the Semi-Lagrangian Crank-Nicholson scheme where you essentially solve Equation (8.5.10) but instead of evaluating the right hand side at point $T^n_{i,j}$ you evaluate it at the interpolated point from which the particle is advecting $T^n_{i',j'}$. In matrix notation what you solve is

$$\mathsf{A}\mathbf{T}^{n+1} = \mathbf{r}' \tag{8.6.1}$$

where $\mathbf{r}'$ is the interpolant of the right hand side taken along the upwind trajectory. The convection calculations in Chapter 2 use this scheme with a multi-grid iterative solver for the implicit matrix inversion. I need to test it a bit more thoroughly but it seems to work quite well. It is not a simple scheme computationally (nor is it particularly cheap) but it is faithful to the underlying physics and is no more expensive than significantly inaccurate schemes. More on this in the next chapter.

Bibliography

[1] D. G. Dritschel and B. Legras. Modeling oceanic and atmospheric vortices, Physics Today 46, 44–51, 1993.

[2] J. Braun and M. Sambridge. A numerical method for solving partial differential equations on highly irregular evolving grids, Nature 376, 655–660, Aug. 24 1995.