Myths & Methods in Modeling

Marc Spiegelman
LDEO, Columbia University

March 18, 2000

Contents

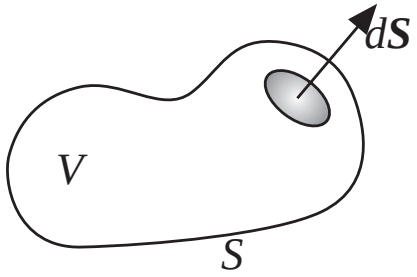
Chapter 1

Conservation Equations

Nearly all of the modeling of physical phenomena is based on the simple statement "You can't get something for nothing" i.e. there are certain important physical properties that must be conserved. This section presents a generic recipe for deriving conservations equation of all kinds and will demonstrate the physical basis of most of the frequently occurring terms in partial (and ordinary) differential equations. When we are finished, we should, with a bit of thought, be able to formulate any quantitative problem in continuum mechanics.

1.1 Conservation of anything

Integral Form Consider an arbitrary inertial frame in space of volume V enclosed by a surface S i.e.



Here, $d\mathbf{S}$ is the vector normal to a small patch on the surface S. This vector points outwards by convention.

If we now consider how any quantity Φ (in units of stuff per unit volume) can change within this volume, the only way to change the amount of Φ with time is to flux it through the boundary or create it within the volume. If we let $\mathbf{F}$ be the flux of Φ in the absence of fluid transport (e.g. heat conduction), $\Phi\mathbf{V}$ be the transport flux (stuff per unit area per unit time) and H be a source or sink of Φ then the statement of conservation of Φ for the volume V becomes

$$\frac{d}{dt}\int_V \Phi dV = -\int_S \mathbf{F}\cdot d\mathbf{S} - \int_S \Phi\mathbf{V}\cdot d\mathbf{S} + \int_V H dV \qquad (1.1.1)$$

The negative signs in from of the surface integrals are present because a positive outward flux corresponds to a negative rate of change of the integral on the left

side of Eq. (1.1.1). This equation is always true, independent of the size of the blob and even if the fields are not continuous; however, because of the integrals, any information on the spatial structure of the fields on a scale smaller than the "blob" size is lost.

Equations for averaged properties: ODE's and box models This loss of spatial information is not always a bad thing and sometimes we are only interested in the changes in the *average* properties in the blob with time. An example would be the mean concentration of a tracer in an ocean basin or "geochemical reservoir". Equations for average properties, however, are readily derived from Eq. (1.1.1).

First we define the volume average of a function f as

$$\bar{f} = \frac{1}{V}\int_V f dV \tag{1.1.2}$$

Next we will simply lump all the flux terms into two terms i.e. the stuff coming in minus the stuff coming out

$$\int_S [\mathbf{F} + \Phi\mathbf{V}] \cdot d\mathbf{S} = \dot{M}_{out} - \dot{M}_{in} \tag{1.1.3}$$

where $\dot{M}$ has units of stuff per unit time. Using these definitions and dividing Eq. (1.1.1) by the volume V yields

$$\frac{d\bar{\Phi}}{dt} = \dot{\Phi}_{in} - \dot{\Phi}_{out} + \bar{H} \tag{1.1.4}$$

where $\dot{\Phi} = \dot{M}/V$.

Equation (1.1.4) is an *Ordinary differential equation* for changes in the average volume density of Φ with time. All information about the spatial variation of Φ within the volume, the fluxes or the sources has been removed but this approach is often good enough for government work or *Box Models* if only averaged properties are necessary. If the functional form of the sources and fluxes are known, Eq. (1.1.4) can often be solved analytically, however, quite often the fluxes depend on the concentrations, can be strongly non-linear or there are many substances to solve for simultaneously. Chapter 4 will deal with numerical solutions of systems of ODE's.

Equations for locally continuous fields: PDE's If we're interested in spatial variations as well as time, we need to subdivide into many smaller blobs. How small is small? In continuum mechanics, we have the concept of the *Representative Volume Element* or RVE (or useful blob size). Given some spatial field Φ, the scale at which the RVE is defined is determined by several properties

1. Φ is relatively constant on a scale comparable to the RVE, i.e. the average value of $\bar{\Phi}$ defined for the RVE is a good approximation to Φ anywhere in the RVE.

2. the average of Φ for each contiguous RVE varies smoothly i.e. $\bar{\Phi}$ is differentiable (e.g. $\boldsymbol{\nabla}\Phi$ makes sense at the scale of the RVE).

Caveats: Once we presume that there is a scale where the RVE is well defined we are also assuming that

1. when we discuss the variation of Φ in space we really are talking about the average of Φ i.e. $\bar{\Phi}$ defined for the RVE

2. We are not interested in anything smaller than this scale

3. Any variation smaller than this scale does not change the gross behaviour of the problem.[1]

Given the existence of a suitable continuum length scale, we can now rewrite Eq. (1.1.1) as a local partial differential equation. Because the property of interest is differentiable, we can replace the surface integrals in Eq. (1.1.1) using Gauss' theorem

$$-\int_S \mathbf{F}\cdot d\mathbf{S} - \int_S \Phi\mathbf{V}\cdot d\mathbf{S} = -\int_V \boldsymbol{\nabla}\cdot(\mathbf{F}+\Phi\mathbf{V})dV \tag{1.1.5}$$

Moreover, because the surface and volume are fixed in an inertial frame then the time derivative of the summed properties is equal to the sum of the local time derivatives or

$$\frac{d}{dt}\int_V \Phi dV = \int_V \frac{\partial\Phi}{\partial t}dV \tag{1.1.6}$$

Substituting Eqs. (1.1.5)–(1.1.6) into (1.1.1) yields

$$\int_V \left[\frac{\partial\Phi}{\partial t} + \boldsymbol{\nabla}\cdot(\mathbf{F}+\Phi\mathbf{V}) - H\right] dV = 0 \tag{1.1.7}$$

Because V is of arbitrary shape and size, Eq. (1.1.7) can only be satisfied if the term in square brackets is zero everywhere (or at least for every RVE), therefore

$$\frac{\partial\Phi}{\partial t} + \boldsymbol{\nabla}\cdot(\mathbf{F}+\Phi\mathbf{V}) - H = 0 \tag{1.1.8}$$

This is the general form which all conservation laws take in continuum mechanics.

1.2 General conservation of mass, energy and momentum

Given Eq. (1.1.8) for the conservation of anything, it is now straightforward to consider conservation of the 3 most important quantities, mass, energy and momentum (force balance).

[1] Quite often, problems in Earth Sciences violate caveats 2 and 3 yet we persevere by introducing "sub-grid" parameterizations of small scale processes or produce combined "micro-macro" models that try to patch together important small scale processes and the large scale dynamics.

Conservation of Mass To derive conservation of mass we just substitute $\Phi = \rho$ (density is the amount of mass per unit volume), $\mathbf{F} = 0$ (mass flux can only change due to transport) and $H = 0$ (mass cannot be created or destroyed) into Eq. (1.1.8) to get

$$\frac{\partial \rho}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \mathbf{V}) = 0 \tag{1.2.1}$$

This equation is often referred to as *the continuity equation*.

Conservation of Energy (heat) For a single phase material, the amount of heat per unit volume is $\Phi = \rho c_P T$ where c_P is the specific heat (energy per unit mass per degree Kelvin) at constant pressure and T is the temperature. The heat flux has two components due to conduction and transport. In the absence of transport the heat flux is $\mathbf{F} = -k\boldsymbol{\nabla} T$ where k is the *thermal conductivity*. Note that heat flows opposite to $\boldsymbol{\nabla} T$, i.e. heat flows from hot to cold. The transport flux is $\rho c_P T \mathbf{V}$. Finally, unlike mass, heat can be created in a region due to terms like radioactive decay or viscous dissipation and shear heating. We will just lump all the source terms into H. Thus the simplest conservation of heat equation is

$$\frac{\partial \rho c_P T}{\partial t} + \boldsymbol{\nabla} \cdot (\rho c_P T \mathbf{V}) = \boldsymbol{\nabla} \cdot k \boldsymbol{\nabla} T + H \tag{1.2.2}$$

For constant c_P and k, this equation can also be rewritten using Equation (1.2.1) as

$$\frac{\partial T}{\partial t} + \mathbf{V} \cdot \boldsymbol{\nabla} T = \kappa \nabla^2 T + H \tag{1.2.3}$$

Where $\kappa = k/\rho c_P$ is the *thermal diffusivity* with units $\mathrm{m}^2\mathrm{s}^{-1}$. Note: terms that look like

$$\frac{D_{\mathbf{V}} T}{Dt} = \frac{\partial T}{\partial t} + \mathbf{V} \cdot \boldsymbol{\nabla} T \tag{1.2.4}$$

are known as the *material derivative* and can be shown to be the change in time of some property (here temperature) as observed in a frame moving with at velocity $\mathbf{V}$ (we will show this explicitly in Section 5.2).

Conservation of Momentum Conservation of momentum or force balance can be derived in exactly the same way, however momentum is a vector field (not a scalar field like temperature). In general momentum is $m\mathbf{V}$, therefore the amount of momentum per unit volume is $\Phi = \rho \mathbf{V}$. Other than advecting momentum, the only other way to change the momentum in our RVE is to exert forces on it. These forces come in two flavors. First, there is the stress that acts on the surface of the volume with local force $\mathbf{f} = \boldsymbol{\sigma} \cdot d\mathbf{S}$. As stress is simply force per unit area, the stress can also be thought of as a flux of force or $\mathbf{F} = -\boldsymbol{\sigma}$ (the negative sign insures that if the net force on the volume points in, the momentum increases). The second force acting on the volume are any body forces such as gravity. The body force acts a source of momentum, thus $H = \rho \mathbf{g}$ where $\mathbf{g}$ is the net acceleration. Substituting into Eq. (1.1.8) yields conservation of momentum

$$\frac{\partial \rho \mathbf{V}}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \mathbf{V}\mathbf{V}) = \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \rho \mathbf{g} \tag{1.2.5}$$

This equation can also be derived (perhaps more simply) by considering the continuous form of Newton's Law $\mathbf{f} = m\mathbf{a}$. The easiest way to understand Eq. (1.2.5) is to think in terms of each of the three components of the momentum which must be conserved individually. Using index notation, Eq. (1.2.5) can be written for the ith component of the momentum as

$$\frac{\partial \rho V_i}{\partial t} + \frac{\partial}{\partial x_j}\left[\rho V_i V_j\right] = \frac{\partial \sigma_{ij}}{\partial x_j} + \rho g_i \tag{1.2.6}$$

where $i = 1, 2, 3$ and summation is assumed over $j = 1, 2, 3$. Using conservation of mass, Eq. (1.2.5) can also be written

$$\frac{\partial \mathbf{V}}{\partial t} + (\mathbf{V}\cdot\boldsymbol{\nabla})\mathbf{V} = \frac{1}{\rho}\boldsymbol{\nabla}\cdot\boldsymbol{\sigma} + \mathbf{g} \tag{1.2.7}$$

Note that the advection of momentum $(\mathbf{V}\cdot\boldsymbol{\nabla})\mathbf{V}$ is non-linear and this is the term that leads to much of the interesting behaviour in fluid mechanics.

1.3 Constitutive relations and approximations

Equations (1.2.1), (1.2.2) and (1.2.6) are applicable to any continuum. To complete the equations, however, requires some additional constraints that relate stress to velocity (strainrate) or displacement, as well as any thermodynamic equations of state for material properties such as heat capacity, density or conductivity (although these are often assumed to be constant).

Viscous fluids The simplest rheology for a fluid is that of an isotropic incompressible fluid where the stress is

$$\sigma_{ij} = -P\delta_{ij} + \eta\left(\frac{\partial V_i}{\partial x_j} + \frac{\partial V_j}{\partial x_i}\right) \tag{1.3.1}$$

where P is the fluid pressure, η is the shear viscosity, and the final bracket on the right hand side is the strain-rate tensor $\dot{\epsilon}_{ij}$. Substituting into (1.2.7) for a constant viscosity fluid gives the Navier-Stokes equation

$$\frac{\partial \mathbf{V}}{\partial t} + (\mathbf{V}\cdot\boldsymbol{\nabla})\mathbf{V} = -\frac{1}{\rho}\boldsymbol{\nabla}P + \nu\nabla^2\mathbf{V} + \mathbf{g} \tag{1.3.2}$$

where $\nu = \eta/\rho$ is the dynamic viscosity.

Elastic bodies Conservation of momentum also applies to elastic rheologies. Here however the stress is proportional to displacement not velocity. Hooke's law for an isotropic elastic medium is

$$\sigma_{ij} = 2\mu\epsilon_{ij} + \lambda\epsilon_{kk}\delta_{ij} \tag{1.3.3}$$

where

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{1.3.4}$$

is the strain tensor, μ is the shear modulus and λ is the bulk modulus. $\mathbf{u}$ are the elastic displacements from equilibrium. In an elastic medium, the position of a particle is $\mathbf{x} = \mathbf{x}_0 + \mathbf{u}$ and the displacements are assumed small. The velocity of a particle is therefore

$$\mathbf{V} = \frac{\partial \mathbf{x}}{\partial t} = \frac{\partial \mathbf{u}}{\partial t} \tag{1.3.5}$$

Substituting into Eq. (1.2.7) and assuming that the advective terms are small (they are order $\mathbf{u} \cdot \mathbf{u}$ which is very small) yields

$$\frac{\partial^2 \mathbf{u}}{\partial t^2} = \frac{\mu}{\rho}\nabla^2 \mathbf{u} + \frac{\lambda + \mu}{\rho}\boldsymbol{\nabla}(\boldsymbol{\nabla} \cdot \mathbf{u}) + \mathbf{g} \tag{1.3.6}$$

which can be shown to be a wave equation for seismic waves (and can be decomposed into shear waves and compressional waves!). When all the body forces balance the displacements so that there is no time dependence, the equation is said to be the equilibrium condition that

$$\frac{\partial \sigma_{ij}}{\partial x_j} + f_i = 0 \tag{1.3.7}$$

Rotating frames and fictitious forces All of the above equations are derived in an *inertial frame* that is undergoing no accelerations. Although our laboratory frame is actually on a rotating planet (and therefore in an accelerating frame), these equations are usually adequate for most solid-earth problems. The principal exceptions are for flows of low viscosity fluids over large regions of the surface (e.g. ocean, atmosphere and core dynamics). For the most part, however, these fluids are moving only slowly relative to the rate of rotation of the earth and therefore it is convenient to transform the inertial equations into equations relative to a frame rotating with the earth. As it turns out, only the momentum equation is actually effected by this transformation (because spatial derivatives are instantaneous in time and material derivatives are invariant to rotation, e.g. see [1] for a derivation). It can be shown that in a frame rotating at a constant angular velocity $\boldsymbol{\Omega}$ (e.g. 1 revolution per day) the Navier-Stokes Equation becomes

$$\frac{\partial \mathbf{V}}{\partial t} + (\mathbf{V} \cdot \boldsymbol{\nabla})\mathbf{V} + 2\boldsymbol{\Omega} \times \mathbf{V} = -\frac{1}{\rho}\boldsymbol{\nabla} P + \nu \nabla^2 \mathbf{V} + \mathbf{g}' \tag{1.3.8}$$

where $2\boldsymbol{\Omega} \times \mathbf{V}$ is the *Coriolis Force* and $\mathbf{g}'$ is gravity minus the *centripetal acceleration* (which is about 1/300th of gravity and is negligible) [1]. Both of these forces are "fictitious forces" that arise from the accelerating frame. Nevertheless at least coriolis force has far from fictitious consequences.

1.4 Scaling and "dimensional analysis"

Every term in each of the above equations reflects a physical process and each of these processes have inherent length scales and time scales. If we had to solve for everything, for every problem we would quickly end up in an intractable mess with

no hope of salvation. Fortunately, not every process is important in every problem and through judicious use of approximations and *scaling*, we can usually tailor the equations to our problem. The most important tool for determining the relative magnitude of various terms is through *non-dimensional scaling* (often called by the overly fancy name *dimensional analysis*). There are actually three principal purposes to making the equations dimensionless.

1. reduce the number of true parameters.
2. Understand the relative magnitudes of the various processes.
3. Make the equations more tractable for numerical solution (all variables are of order 1).

The mechanics of scaling are straightforward but are best demonstrated with specific examples. Here we will do two basic examples to get the flavor of the exercise. More examples can be found in Chapter 2.

Example 1: Heat flow and the Peclet number The first problem will demonstrate the basic techniques of scaling on a simple two-process problem. In the absence of any heat sources, and assuming constant material properties, the simplest 1-D equation for heat flow is

$$\frac{\partial T}{\partial t} + W\frac{\partial T}{\partial z} = \kappa\frac{\partial^2 T}{\partial z^2} \tag{1.4.1}$$

This equation includes two processes, advection of heat at velocity W, and diffusion of heat with thermal diffusivity κ. As an example problem, this equation could be used to solve for the temperature distribution directly beneath an upwelling mantle plume or ridge (see Figure 1.1a). While it may appear that there are at least two free parameters (W and κ as well as some temperatures), there is in fact only one parameter and it is independent of temperature.

To show this we begin by replacing the dimensional variables with dimensionless ones. The choice of scaling values is a bit of an art. Examination of Figure 1.1a shows that for the case of an upwelling through a thermal layer of depth d, which has constant temperatures T_0 at $z = 0$ and T_1 at $z = d$ and has a characteristic velocity W_0, the sensible scaling is

$$\begin{aligned} z &= dz' \\ t &= \frac{d}{W_0}t' \\ \frac{\partial}{\partial z} &= \frac{1}{d}\frac{\partial}{\partial z'} \\ W &= W_0 W' \\ T &= T_0 + (T_1 - T_0)T' \end{aligned} \tag{1.4.2}$$

where the primes denote dimensionless variables. Brute force substitution of (1.4.2) into (1.4.1) gives

$$\frac{\Delta T W_0}{d}\left[\frac{\partial T'}{\partial t'} + W'\frac{\partial T'}{\partial z'}\right] = \frac{\kappa \Delta T}{d^2}\frac{\partial^2 T'}{\partial z'^2} \tag{1.4.3}$$

where $\Delta T = T_1 - T_0$. Multiplying both sides by $d/(\Delta T W_0)$ and dropping the primes yields (in 1-D where $\mathbf{V} = W_0\mathbf{k}$)

$$\frac{\partial T}{\partial t} + \frac{\partial T}{\partial z} = \frac{1}{\mathrm{Pe}}\frac{\partial^2 T}{\partial z^2} \tag{1.4.4}$$

where

$$\mathrm{Pe} = \frac{W_0 d}{\kappa} \tag{1.4.5}$$

is the *Peclet number* which controls the relative strength of advection to diffusion. If Pe is large, advection dominates and the last term is negligible[2]. If Pe is small, diffusion dominates. However there is only one parameter that controls all solutions. Figure 1.1 shows the analytic steady state solution to (1.4.4) with dimensionless boundary conditions $T(0) = 1$, $T(1) = 0$ and a range of Pe.

Another, more physical way to derive the Peclet number is to consider the time it takes each process to affect the entire layer. The time it takes to advect across the layer at speed W_0 is $t_{adv} = d/W_0$ while the time it takes for heat to diffuse a distance d is $t_{diff} = d^2/\kappa$. Thus the Peclet number is simply the ratio of the diffusion time to the advection time, i.e. $\mathrm{Pe} = t_{diff}/t_{adv}$ (compare also to Eq. (1.4.3)). This is the characteristic property of all the zillions of dimensionless numbers that crop up in continuum mechanics. They are always simply ratios of times or forces generated by any two terms. Thus any two processes will generate a dimensionless number (although if the processes are coupled the overall number of numbers may be less). Based on the sizes of these dimensionless numbers, further approximations are made to simplify the equations.

Example 2: Scaling the quick and dirty way… instant dimensionless numbers This example will demonstrate the maxim "two processes = 1 number" by demonstrating some quick and dirty scaling arguments for understanding the momentum equation. Consider the dimensional Navier stokes equation for a rotating frame

$$\frac{\partial \mathbf{V}}{\partial t} + (\mathbf{V}\cdot\boldsymbol{\nabla})\mathbf{V} + 2\boldsymbol{\Omega}\times\mathbf{V} = \nu\nabla^2\mathbf{V} - \frac{1}{\rho}\boldsymbol{\nabla}P + \mathbf{g}' \tag{1.4.6}$$

which has at least 4 independent processes, advection of momentum, Coriolis force, viscous diffusion and pressure terms. The first trick to Q&D scaling is to simply replace all the derivatives with fractions that have the same units and scales. For example the non-linear advection term $(\mathbf{V}\cdot\boldsymbol{\nabla})\mathbf{V}$ is of the same units and scale as the fraction U_0^2/L where U_0 is a characteristic velocity and L is a characteristic length scale. If we again scale time to the advection time $t_{adv} = L/U_0$ then we can approximate (1.4.6) as

$$\frac{U_0^2}{L} + \frac{U_0^2}{L} + 2\Omega U_0 = \nu\frac{U_0}{L^2} - O(g) \tag{1.4.7}$$

where all the pressure and body force terms have been lumped into one term of order the body forces. Since all of these terms have units of acceleration (length/time^2)

[2] except in narrow boundary layers

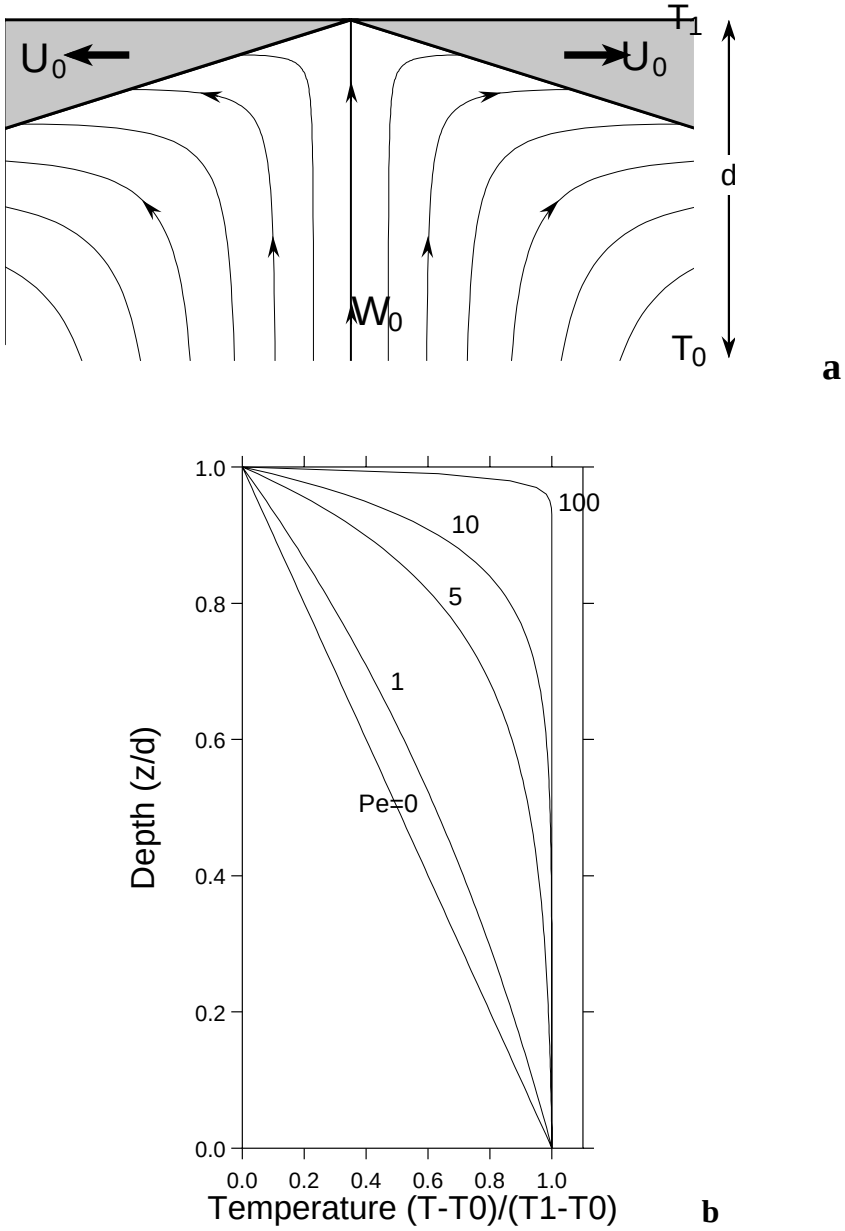


Figure 1.1: (a) Cornerflow solution for solid flow beneath a ridge. Layer depth is d, Upwelling rate on axis is W_0. (b) Analytic solution to the simplest steady state-advection diffusion problem. The solution is $T(z) = \left(e^{\mathrm{Pe}z} - e^{\mathrm{Pe}}\right) / \left(1 - e^{\mathrm{Pe}}\right)$. This problem is a good estimate for the thermal structure directly on axis. Note that the Peclet number controls the width of the thermal boundary layers which are of order $l = 1/\mathrm{Pe}$.

the ratio of any two of the processes is a dimensionless number with a fancy name. It is clear that the first two terms the local acceleration and the advected acceleration are of the same magnitude and we will call the first two terms the *inertial terms* and treat them together. To generate any of the important dimensionless numbers we just consider the relative magnitude of any two terms. For this problem there are three important numbers.

The Reynolds number is the ratio of inertial accelerations to viscous forces i.e.

$$\mathrm{Re} = \frac{U_0 L}{\nu} \tag{1.4.8}$$

Comparison to Eqs. (1.4.4)–(1.4.5) shows that the Reynolds number and the Peclet number fill the same role. Thus the Reynolds number controls the size of viscous boundary layers. Large Re means turbulent flow with thin viscous boundary layers. Small Re flow are strongly viscous.

The Rossby number is the ratio of inertial accelerations to coriolis accelerations i.e.

$$\mathrm{Ro} = \frac{U_0}{2\Omega L} \tag{1.4.9}$$

If $\mathrm{Ro} \to 0$, the system is effectively in solid body rotation. For $\mathrm{Ro} > 1$ rotation can be neglected relative to *inertial forces* (e.g. laboratory flume experiments). Many interesting problems happen for Rossby numbers of order 1.

The Eckman number is the square root of the ratio of viscous forces to coriolis forces

$$\mathrm{Ek} = \left[\frac{\nu}{2\Omega L^2}\right]^{1/2} \tag{1.4.10}$$

The Eckman number controls the size of boundary layers in rotational problems. For large Ek viscous forces dominate and coriolis terms can be neglected.

Table 1.1 gives some representative values of scales and dimensionless numbers for the earth's mantle, the Gulf Stream, and your bathtub. By inspecting the relative magnitudes of the different processes, this quick scaling suggests that for mantle convection we can neglect inertial and rotational forces. For the oceans we can neglect (with some care) viscous forces, and for our bathtub we can neglect coriolis forces (thus destroying the direction of bathtub draining myth).

Caveats: Small parameters This approach is good for a quick back of the envelope estimate of the relative magnitude of different terms. Usually when one term is much smaller than another it can just be thrown out (at least as a first guess). Some caution should be exerted however when the small value multiplies the terms with the highest derivatives (e.g. $(1/\mathrm{Pe})\nabla^2 T$ in example 1). As demonstrated in figure 1.1b, a large Peclet number does not necessarily imply that diffusion can be neglected, rather it means that it only becomes important in narrow boundary layers. Without these boundary layers, however, some problems are poorly posed. A more drastic example from magma migration will be shown in class.

Table 1.1: Some scales and scaling for three fluid problems.

	The mantle	The Gulf Stream	Your Bathtub
U_0	3 cm/yr (10^{-9} ms^{-1})	1 ms^{-1}	.01 ms^{-1}
L	3000 km	100 km	1 m
ν	10^{18} m^{2}s^{-1}	10^{-6} m^{2}s^{-1}	10^{-6} m^{2}s^{-1} (10^{-2} m^{2}s^{-1} corn syrup)
$\mathbf{\Omega}$	2π/day (7.3×10^{-5} s^{-1})	7.3×10^{-5} s^{-1}	7.3×10^{-5} s^{-1}
Re	3×10^{-21}	10^{11}	10^4 (1 corn syrup)
Ro	2×10^{-12}	.07	70
Ek^2	8×10^8	7×10^{-13}	7×10^{-3} (70 for cs)

1.5 Summary

Beginning with conservation of anything for a fixed volume, we have come up with a large number of equations that govern the physics of just about every continuum problem we can think of. Using scaling, we have also shown how to simplify (?) these equations and how to get the first vague understanding of the relative importance of different processes for different problems. Now it's time to start solving these equations. Fortunately there are really only three basic types of equations that come out of this analysis, Ordinary Differential Equations which depend only on time, Time dependent Partial differential equations (space and time) and boundary value problems (just space). The following sections will show how to deal with each of these basic types in turn.

Bibliography

[1] M. Ghil and S. Childress. Topics in geophysical fluid dynamics : atmospheric dynamics, dynamo theory, and climate dynamics, vol. 60 of Applied mathematical sciences, Springer-Verlag, New York, 1987.

Chapter 2

Some Real(?) Problems in Earth Science

2.1 Introduction

Chapter 1 provided a basic recipe for concocting systems of conservation equations for just about any continuum problem you can imagine. This chapter will use this technique to set up a suite of fundamental physical problems that are important to many aspects of Earth science. The purpose of this chapter is not only derive the basic problems but to start to develop some basic physical intuition into how they behave. This intuition will be exceptionally important when we start to choose numerical methods that are appropriate for each problem. Nevertheless, when we are done, this chapter should demonstrated that despite the very different kinds of behaviour, as far as we are concerned numerically there are really only two kinds of problems; initial value problems (IVP's) that need to be marched carefully through time and boundary value problems (BVP's) that need to be satisfied simultaneously everywhere in space. The most interesting physics comes from combinations of these kinds of equations.

When you are done this course, should be able to solve all of the problems in this chapter as well as concoct your own custom problem. As for the basic problems in this chapter, these should be treated as the simplest examples of a class of problems and not taken for gospel. In general you should make up your own stories and build upon the framework of these problems (that's what theory is about). However, to be successful it is important to know to know the basic stories as well.

2.2 Thermal Convection

The first problem we will consider is that of thermal convection in fluids. Convection is simply the statement that hot fluids rise and cold fluids sink (more correctly low density fluids rise and high density fluids sink). Convection is a crucial process throughout the earth. Heating of the equator and cooling at the poles provides the basic engine for weather and climate change. The formation of cold

North-atlantic water drives the global *thermo-haline* circulation (which is technically *double-diffusive convection* because both heat and salt affect the densities.) Thermal convection in the mantle is the principal engine for plate tectonics and magneto-hydrodynamic convection in the core drives the Earth's dynamo and controls the Earth's magnetic field. Not too bad for a process where hot things rise and cold things sink. To gain some insight into convection, we will consider the simplest problem of thermal convection in a layer heated from below. This is the classic *Rayleigh-Benard* convection problem and is a favorite of physicists and earth-scientists everywhere.

2.2.1 Derivation

Starting with the general conservation equations for mass, momentum and heat for a viscous fluid, and assuming that the fluid is incompressible and has a constant viscosity then the *dimensional* governing equations for thermal convection can be written

$$\boldsymbol{\nabla}\cdot\mathbf{V} = 0 \tag{2.2.1}$$

$$\frac{\partial T}{\partial t} + \mathbf{V}\cdot\boldsymbol{\nabla}T = \kappa\nabla^2 T \tag{2.2.2}$$

$$\frac{\partial \mathbf{V}}{\partial t} + \mathbf{V}\cdot\boldsymbol{\nabla}\mathbf{V} = \nu\nabla^2\mathbf{V} - \frac{1}{\rho_0}\boldsymbol{\nabla}P + \frac{\rho}{\rho_0}\mathbf{g} \tag{2.2.3}$$

Equation (2.2.1) is conservation of mass, (2.2.2) is conservation of heat and (2.2.3) is conservation of momentum. These equations have been written assuming a "Boussinesq Approximation" which says that the fluid densities are effectively constant ρ_0 except in the body force terms where they drive most of the flow. For thermal convection the density of the fluid is temperature dependent

$$\rho = \rho_0(1 - \alpha(T - T_0)) \tag{2.2.4}$$

where α is the *coefficient of thermal expansion*. In 2-D we can write out the x and z components of Eq. (2.2.3) as

$$\frac{\partial U}{\partial t} + \mathbf{V}\cdot\boldsymbol{\nabla}U = \nu\nabla^2 U - \frac{1}{\rho_0}\frac{\partial P}{\partial x} \tag{2.2.5}$$

$$\frac{\partial W}{\partial t} + \mathbf{V}\cdot\boldsymbol{\nabla}W = \nu\nabla^2 W - \frac{1}{\rho_0}\frac{\partial P}{\partial z} + (1 - \alpha(T - T_0))g \tag{2.2.6}$$

where we have substituted in (2.2.4). To make our lives easier (and this section much more confusing) it is useful to take the Curl of Eq. (2.2.3) to remove the gradient terms (remember $\boldsymbol{\nabla}\times\boldsymbol{\nabla}f = 0$). If we also use $\boldsymbol{\nabla}\cdot\mathbf{V} = 0$ it can be shown that Eq. (2.2.3) becomes in 2-D

$$\frac{\partial \omega}{\partial t} + \mathbf{V}\cdot\boldsymbol{\nabla}\omega = \nu\nabla^2\omega - g\alpha\frac{\partial T}{\partial x} \tag{2.2.7}$$

where

$$\omega = (\boldsymbol{\nabla}\times\mathbf{V})\cdot\mathbf{k} = \left(\frac{\partial W}{\partial x} - \frac{\partial U}{\partial z}\right) \tag{2.2.8}$$

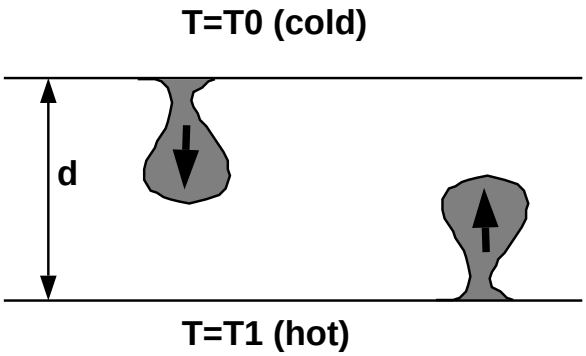


Figure 2.1: The geometry and physics of Rayleigh-Benard thermal convection. I.e. Hot goes up... cold goes down.

is the *vorticity*. Vorticity can be thought of as the local rate of rotation of a fluid particle. Comparison of Eqs. (2.2.2) and (2.2.7) shows that one is an advection-diffusion equation for Temperature, and the other is for vorticity, however the vorticity equation also has a source term $g\alpha\partial T/\partial x$ i.e. lateral variations in temperature will drive rotational flow (i.e. convection). To finish the derivation we note that because the fluid is incompressible (Eq. 2.2.1) we can rewrite the velocity as

$$\mathbf{V} = \boldsymbol{\nabla} \times \psi \mathbf{k} \tag{2.2.9}$$

where ψ is the *streamfunction*. This relationship is true because in general, $\boldsymbol{\nabla} \cdot (\boldsymbol{\nabla} \times \mathbf{F}) = 0$ for all vectors F.

Substituting into Eqs. (2.2.1)–(2.2.3) yields the dimensional 2-D equations in stream-function vorticity form

$$\frac{\partial T}{\partial t} + (\boldsymbol{\nabla} \times \psi \mathbf{k}) \cdot \boldsymbol{\nabla} T = \kappa \nabla^2 T \tag{2.2.10}$$

$$\frac{\partial \omega}{\partial t} + (\boldsymbol{\nabla} \times \psi \mathbf{k}) \cdot \boldsymbol{\nabla} \omega = \nu \nabla^2 \omega - g\alpha \frac{\partial T}{\partial x} \tag{2.2.11}$$

$$\nabla^2 \psi = -\omega \tag{2.2.12}$$

where Eq. (2.2.12) arises from the definitions of ψ and ω

2.2.2 Scaling

To determine the various magnitudes of each of the terms for the problem of a uniform layer of depth d with top temperature T_0 and lower temperature T_1 (see Fig. 2.1), it is now useful to scale everything to the thermal diffusive time scale. which is given by the time it takes for heat to diffuse across the layer. The typical scaling for this problem is

$$\begin{aligned} (x, z) &= d(x, z)' \\ t &= \frac{d^2}{\kappa} t' \end{aligned}$$

$$\begin{aligned}
\boldsymbol{\nabla} &= \frac{1}{d}\boldsymbol{\nabla}' \\
\mathbf{V} &= \frac{\kappa}{d}\mathbf{V}' \\
\omega &= \frac{\kappa}{d^2}\omega' \\
\psi &= \kappa\psi' \\
T &= T_0 + (T_1 - T_0)T'
\end{aligned} \tag{2.2.13}$$

Substituting and dropping primes yields the dimensionless equations

$$\frac{\partial T}{\partial t} + (\boldsymbol{\nabla} \times \psi\mathbf{k}) \cdot \boldsymbol{\nabla} T = \nabla^2 T \tag{2.2.14}$$

$$\frac{1}{\mathrm{Pr}}\left(\frac{\partial \omega}{\partial t} + (\boldsymbol{\nabla} \times \psi\mathbf{k}) \cdot \boldsymbol{\nabla}\omega\right) = \nabla^2\omega - \mathrm{Ra}\frac{\partial T}{\partial x} \tag{2.2.15}$$

$$\nabla^2\psi = -\omega \tag{2.2.16}$$

where $\mathrm{Pr} = \nu/\kappa$ is the *Prandtl Number* which is the ratio of momentum diffusivity to thermal diffusivity (i.e. Pe/Re) and

$$\mathrm{Ra} = \frac{\rho\alpha g \Delta T d^3}{\eta\kappa} \tag{2.2.17}$$

is the *Raleigh Number* which measures the relative strength of buoyant time scale to the diffusive time scale.

For the Earth's mantle, $\mathrm{Pr} > 10^{24}$ so the inertial terms are completely negligible while $\mathrm{Ra} > 10^6$ so buoyancy forces are enormous. Thus when we are solving for mantle convection we usually assume the the Prandtl number is infinite and Eq. (2.2.15) reduces to

$$\nabla^2\omega = \mathrm{Ra}\frac{\partial T}{\partial x} \tag{2.2.18}$$

Note that the Peclet number seems to have disappeared from Eq. (2.2.14). It is actually there, however it has been assumed to equal 1 because all the interesting convection problems happen when the solid flow rate is comparable to the diffusion rate.

Ta da! (now don't you feel much better). By all this jiggery pokery, we've turned 4 equations with 3 free parameters into 3 equations with one adjustable parameter. Note also that equations (2.2.18) and (2.2.16) have no time derivatives and must be satisfied everywhere instantaneously in space. These equations are known as Poisson equations and crop up in mathematical physics all the time. We will deal with their solution in gory detail later on.

2.2.3 Some solutions and a bit of physics

Figure 2.2 shows some numerical solutions for infinite Prandtl number Rayleigh-Benard convection in a 2 by 1 rectangular box. The beauty of RB convection is that, although there are three coupled equations for T, ω and ψ, there is only one

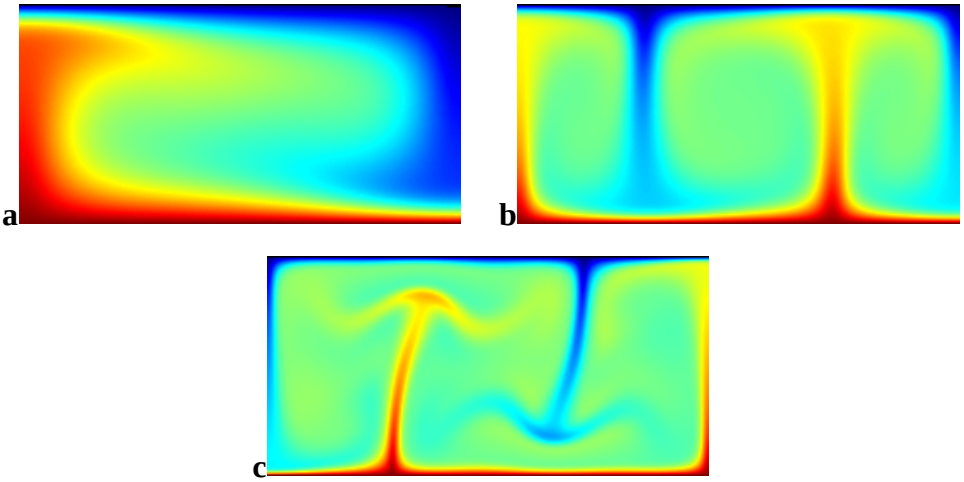


Figure 2.2: Some solutions for infinite Prandtl number Rayleigh Benard convection for different Rayleigh numbers. All of these solutions have free-stress boundary conditions with reflection sides for temperature. For numerical solution they use a combination of semi-Lagrangian and multi-grid techniques. *(a)* $\mathrm{Ra} = 10^4$: convection goes to steady state with a broad symmetric upwelling and matching downwelling. *(b)* $\mathrm{Ra} = 10^5$: upwellings and downwellings are narrower, more plentiful and become weakly time-dependent. *(c)* $\mathrm{Ra} = 10^6$: convection becomes vigorous and time-dependent.

adjustable parameter, the Rayleigh number Ra. Thus if we can do a suite of numerical runs spanning Ra space we can map out the behaviour of these equations.[1] However, even with only one parameter the behaviour of these equations can be quite complex. Fortunately, there is an immense literature on this problem which lays out all the approximate and analytic (and numerical solutions) that can help you immensely in understanding new problems.

The most important feature of these equations is that there is a critical value of the Rayleigh number, below which no convection occurs. The actual value depends on the geometry of the box and the boundary conditions on temperature and flow. Below the critical Ra there is no motion and temperature is a vertical gradient. Right above the critical Ra, the problem usually forms a steady state set of convection rolls with equally spaced hot upwellings and cold downwellings (Fig. 2.2a). In 3-D these rolls can assume many interesting patterns (squares, hexagons, zig-zag rolls etc.). As Ra is increased, more energy is added to the system and the rolls begin to go time dependent. At very high Ra the system can go chaotic (Fig. 2.2c).

[1] this is a bit of a lie, actually. To solve these equations for a specific instance, you also have to impose boundary and initial conditions for the problem and this can actually increase the total number of problem specific parameters dramatically.

2.2.4 Another approach to convection: the Lorenz Equations and chaos

Another version of the Rayleigh-Benard convection problem also features prominently in the story of chaos (see Gleick [1] for a good time) as it is the foundation for the most famous of chaotic problems *The Lorenz Equations*. As discussed, quite elegantly, by Edward Lorenz [2], the Lorenz equations are a simplified toy model of convection in the atmosphere that were developed to demonstrate the unpredictability of chaotic systems. They also form a good example of another important class of problems that need numerical solutions and that is systems of non-linear ordinary differential equations (or non-linear dynamical systems).

What Lorenz did was rather than solving the full PDE's Eqs. (2.2.14)–(2.2.16) he assumed he knew the spatial structure of the velocity and temperature field and only solved for the time-dependent part. More specifically he assumed he could write ψ and T as a truncated 2-D Fourier series as

$$\psi = W(t)\sin(\pi a x)\sin(\pi z) \tag{2.2.19}$$

$$T = (1-z) + T_1(t)\cos(\pi a x)\sin(\pi z) + T_2(t)\sin(2\pi z) \tag{2.2.20}$$

which assumes that the velocity field can be described by a pair of counter-rotating rolls with a wavelength of $2/a$ and that the temperature can be described as the sum of three modes. The first is a steady state ramp that is hot on the bottom and cold on the top, the second mode controls horizontal temperature gradients and the third mode controls vertical temperature gradients. Since the spatial variation is assumed known, the only unknowns are the time-dependent coefficients W, T_1, T_2. Substituting Eqs. (2.2.19) and (2.2.20) into Eqs. (2.2.14)–(2.2.16) and collecting terms with common modes yields a system of non-linear Ordinary differential equations for the time dependent coefficients.

$$\begin{aligned} \frac{dW}{dt} &= \mathrm{Pr}(T_1 - W) \\ \frac{dT_1}{dt} &= -WT_2 + rW - T_1 \\ \frac{dT_2}{dt} &= WT_1 - bT_2 \end{aligned} \tag{2.2.21}$$

where r is the value of the Raleigh number normalized by the critical Rayleigh number $(\mathrm{Ra}/\mathrm{Ra}_c)$ and $b = 4/(1+a^2)$. Equation (2.2.21) is a good example of a *spectral method* where the solution is expanded in terms of Fourier modes; however, in this case the expansion is severely truncated to just the first few modes.

For r close to 1, the solution of the Lorenz equations is a good approximation to that of the full equations. At high values of r, however, it is not because it does not have enough degrees of freedom to generate new convection cells. Nevertheless, this system of equations does show remarkable aperiodic behaviour in that the direction the convection roll turns flips in a chaotic fashion. Figure 2.3 shows the time-series for the classic solution of the equations with $r = 28$, $\mathrm{Pr} = 10$ and $b = 8/3$.

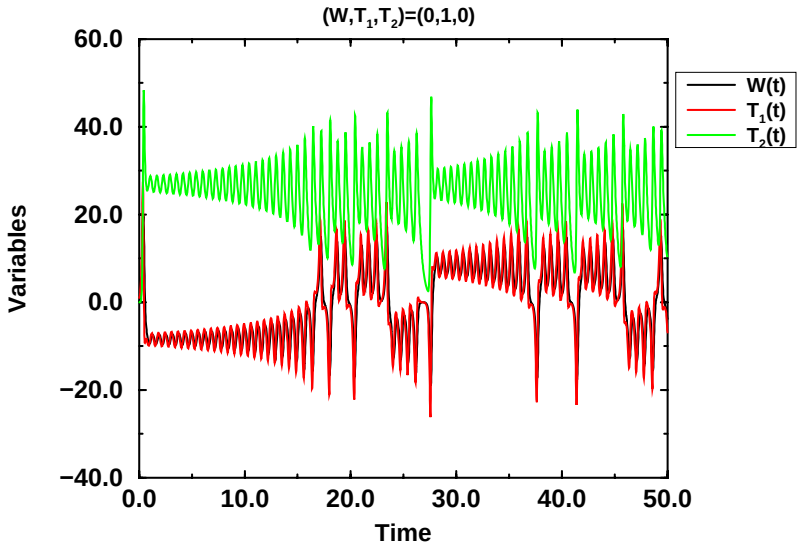


Figure 2.3: Time series of W, T_1 and T_2 for the Lorenz equations with $r = 28$, $\mathrm{Pr} = 10$ and $b = 8/3$. Positive W means clockwise rotation, Negative W means counter-clockwise rotation. Note the erratic flipping with growing oscillations that is characteristic of these chaotic equations.

2.3 Shallow water equations

Convection governs vertical motions of fluids in the earth; however, for many problems such as large-scale ocean and atmosphere dynamics, the scale of horizontal motions is much larger than the scale of vertical motions. Thus it is often convenient to consider an approximate version of the equations of motions in a rotating frame Eq. (1.3.8) where we assume that the fluid is confined to layer which is much thinner than it is wide. We also assume that vertical motions only change the layer depth and that the pressure gradient is near hydrostatic. A full derivation of the shallow water equations and a discussion of where they are valid can be found in any good ocean-atmosphere text such as Gill [3].

The shallow-water equations for flow of fluid on a rotating sphere (aka the Earth) in spherical polar coordinates was worked out by Laplace to be

$$\frac{Du}{Dt} - \left(2\Omega + \frac{u}{r\cos\phi}\right) v \sin\phi = -\frac{g}{r\cos\phi}\frac{\partial \eta}{\partial \lambda} \tag{2.3.1}$$

$$\frac{Dv}{Dt} + \left(2\Omega + \frac{u}{r\cos\phi}\right) u \sin\phi = -\frac{g}{r}\frac{\partial \eta}{\partial \phi} \tag{2.3.2}$$

$$\frac{\partial \eta}{\partial t} + \frac{1}{r\cos\phi}\left\{\frac{\partial}{\partial\lambda}[(H+\eta)u] + \frac{\partial}{\partial\phi}[(H+\eta)v]\right\} = 0 \tag{2.3.3}$$

where λ is the longitude and ϕ is the latitude. u is the horizontal velocity in the longitudinal direction or zonal flow ($u > 0$ is eastward flow), v is meridional flow ($v > 0$ is northward flow). Note, the material derivative D/Dt only applies to horizontal motion. Vertical motion is neglected in these equations except for how

it affects η, the perturbed thickness of the water layer which has an equilibrium depth of H in the absence of any motion. Ω is the Coriolis rotational velocity and r is the radius of the sphere. In general these equations look quite awful but most of that is due to the interactions of the spherical geometry and the Coriolis terms (and this is what makes ocean-atmosphere dynamics interesting). The basic physics is straightforward however. Variations in layer thickness η drive horizontal fluid motions which are modified by the Coriolis forces which make the velocities turn. The divergence or convergence of the horizontal velocities, however, change the layer thickness and the three variables feed-back on each other and propagate as waves. While it is not obvious from the form of Equations (2.3.1)–(2.3.3), these equations are effectively a complicated wave-equation for how a thin inviscid layer tries to adjust to equilibrium.

2.3.1 Linearized Shallow water equations for the equatorial β plane

Unless you're interested in planetary scale flow, Eqs. (2.3.1)–(2.3.3) are a bit of overkill[2] and additional simplifications can be made for specific regions of the planet. An important region for the evolution of climate is the tropics and a useful approximation of the full spherical equations near the equator can be made using the *Equatorial Beta plane* approximation. Near the equator, ϕ is small so $\sin\phi \approx \phi$ and $\cos\phi \approx 1$. Moreover we can project our spatial positions onto a plane tangent to the equator such that our new east-west coordinate is $x = r\lambda$ and our north-south coordinate is $y = r\phi$. The Coriolis parameter f is defined as $f = 2\Omega\sin\phi$ and we can also define the beta parameter as

$$\beta = \frac{1}{r}\frac{df}{d\phi} = \frac{2\Omega\cos\phi}{r} \tag{2.3.4}$$

and thus near the equator, the Coriolis parameter becomes $f = \beta y$ and thus vanishes at the equator and the absolute value of f increases away from the equator. . Substituting these relationships into Eqs. (2.3.1)–(2.3.3) and assuming that the overall velocities are sufficiently small that products of velocities are much smaller than the velocity themselves (i.e. $u^2, v^2, uv \ll u, v$) then the linearized shallow-water equations on the equatorial beta plane can be written

$$\frac{\partial u}{\partial t} - \beta y v = -g\frac{\partial \eta}{\partial x} \tag{2.3.5}$$

$$\frac{\partial v}{\partial t} + \beta y u = -g\frac{\partial \eta}{\partial y} \tag{2.3.6}$$

$$\frac{\partial \eta}{\partial t} + \frac{\partial (Hu)}{\partial x} + \frac{\partial (Hv)}{\partial y} = 0 \tag{2.3.7}$$

These equations admit a large number of analytic wave solutions and are discussed in detail in Gill [3]. The most important feature of these equations is that the Coriolis force vanishes on the equator $y = 0$ yet increases in both directions away

[2] an interesting application of them however is for the development of atmospheric patterns in the atmosphere of Jupiter by our own Lorenzo Polvani [4]

from it. These features of the beta plane cause wave energy to be trapped near the equator, i.e. the equator acts as a wave guide. In particular there are two important kinds of waves that arise in the tropics that are of interest in climate studies. The fastest moving waves are the eastward propagating *Kelvin Waves*, that have no north-south component of velocity and move at a constant velocity, independent of the east-west wavenumber. Typical Kelvin waves for the pacific might move about 3 ms^{-1} and take about 2 months to cross the Pacific. The other important waves are the equatorial *Rossby waves* or *planetary waves.* These are more slowly moving (the fastest Rossby wave is about a third the speed of the Kelvin wave) and have phase velocities that propagate westward. It is generation of Kelvin and Rossby waves by the atmosphere that gives rise to the Pacific climate oscillation known as El Niño.

2.3.2 El Niño prediction the Cane/Zebiak way

Equations (2.3.5)–(2.3.7) play a fundamental role in the Cane-Zebiak model of El Niño forecasting (e.g. [5–7]). This model is a simplified, coupled ocean-atmosphere model where the ocean is described by the forced equatorial shallow water equations (plus a little bit). Qualitatively, the tropical dynamics in this model is that variations in sea-surface temperature drive winds in the atmosphere. These winds then drive the ocean which generates Kelvin and Rossby waves which transport the sea-surface temperature field and so on. the principal additions to Eqs. (2.3.5)–(2.3.7) are the forcing by the wind stress vector $\tau = (\tau_x, \tau_y)$ and an appropriate damping term to dissipate the forced energy. The model also assumes a constant undisturbed layer depth H_0. With these assumptions, the dimensional ocean model for Cane-Zebiak can be written.

$$\frac{\partial u}{\partial t} - \beta y v = -gH_0 \frac{\partial \eta}{\partial x} + \tau_x/\rho - ru \tag{2.3.8}$$

$$\frac{\partial v}{\partial t} + \beta y u = -gH_0 \frac{\partial \eta}{\partial y} + \tau_y/\rho - rv \tag{2.3.9}$$

$$\frac{\partial \eta}{\partial t} + \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} - r\eta = 0 \tag{2.3.10}$$

The dissipation terms, $-r\mathbf{v}, -r\eta$ are simplistic *Rayleigh Friction* terms that act exactly the same way radioactive decay works. Without this term, the energy in the ocean model would just keep increasing with time which is unrealistic. In addition to the bulk ocean transport, the model also includes a surface *frictional layer* that mimics local ocean upwelling due to surface wind divergence. Variations in the oceanic upwelling also affects the sea surface temperature by bringing up (or pushing down) colder water at depth. The remarkable thing about this model is its simplicity. It is not a fully non-linear coupled ocean-atmosphere model with thermodynamics and the kitchen sink. It is actually a very graceful simplified notion of the interaction of the ocean and atmosphere in the tropical pacific that captures the essential physics with the minimal effort and is actually useful for predicting climate changes (sometimes). That's the hallmark of an excellent model and a style you should strive for.

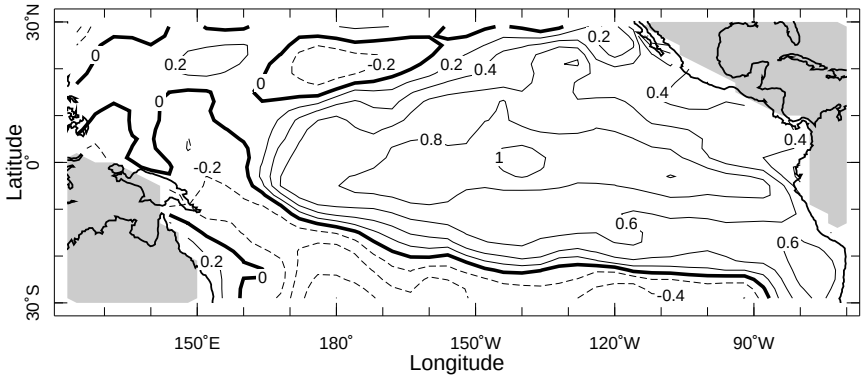


Figure 2.4: Picture of the predicted Sea Surface Temperature anomaly (SSTA) from the current LDEO2 model of El Niño. Predictions are for September 1998. Model domain does not include the grey areas. The basic physics of this model is that variations in the sea-surface temperature drive winds in the atmosphere (which is also governed by a set of shallow "water" equations). These winds, induce wave motions in the oceans and the currents associated with the waves transport the sea-surface temperatures. For more information see `http://rainbow.ldeo.columbia.edu/`

2.4 Seismic Wave propagation

The ocean and atmosphere are an endless source of wave propagation problems. The other classic source, of course, is in seismology where 90% of what we know about the structure and properties of the solid earth deeper than a few kilometers comes from understanding the behaviour of seismic waves. Much like a full solution of the Navier stokes equation for the ocean and atmosphere is completely unfeasible, so is a full solution of the wave equation for the interior of the earth. Nevertheless, there have been a large number of clever people who have worked out extremely useful approximate schemes that actually allow us to do sophisticated problems without actually solving the full wave equation numerically. Nevertheless, there are times when the approximate theory is not enough or the geometry of the seismic velocity fields are too complicated that it is necessary to actually brute force it. Times when numerical solutions are useful include doing time migration in exploration seismics, analyzing the interaction between seismic waves and geologic structures or simply desiring to make pretty pictures. The techniques themselves are not particularly difficult, the challenge is to model efficiently. The following discussion is heavily cribbed from a very useful set of notes by Gustavo Correra, LDEO.

2.4.1 Basic derivation: linear elastic media

The general equations for conservation of momentum for a deformable continuum is given by Eq. (1.2.7) as

$$\rho \frac{D\mathbf{V}}{Dt} = \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \rho \mathbf{g} + \mathbf{f} \tag{2.4.1}$$

where **f** are any other transient forces (explosions, earthquake sources) in addition to gravity. For the case of wave propagation in an elastic material we can assume that the overall displacements will be small such that

$$\frac{D\mathbf{V}}{Dt} \approx \frac{\partial \mathbf{V}}{\partial t} \tag{2.4.2}$$

and that the stress tensor for an isotropic, linearly elastic solid can be written in component form as

$$\sigma_{ij} = 2\mu\epsilon_{ij} + \lambda\epsilon_{kk}\delta_{ij} \tag{2.4.3}$$

where μ is the *shear modulus*, λ is the*bulk modulus* and the strain tensor is

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{2.4.4}$$

and **u** is the local displacement vector. Furthermore, we will assume that both the stress tensor and the total displacement can be decomposed into a static component that balances the loading by body forces (the *pre-stressed displacements*) and a transient component that is important during elastic wave propagation. i.e.

$$\mathbf{u}_{tot} = \mathbf{u}_0 + \mathbf{u} \tag{2.4.5}$$

$$\boldsymbol{\sigma}_{tot} = \boldsymbol{\sigma}_0 + \boldsymbol{\sigma} \tag{2.4.6}$$

where $\mathbf{u}_0$ are the solutions of the static problem

$$\boldsymbol{\nabla} \cdot \boldsymbol{\sigma}_0 + \rho \mathbf{g} = 0 \tag{2.4.7}$$

Taking the time derivatives of (2.4.5) and (2.4.6) and substituting in Eqs. (2.4.1), (2.4.3) and (2.4.4) yields the equations for wave propagation

$$\rho \frac{\partial \mathbf{V}}{\partial t} = \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \mathbf{f} \tag{2.4.8}$$

$$\frac{\partial \sigma_{ij}}{\partial t} = \mu\left(\frac{\partial V_i}{\partial x_j} + \frac{\partial V_j}{\partial x_i}\right) + \lambda \boldsymbol{\nabla} \cdot \mathbf{V}\delta_{ij} \tag{2.4.9}$$

which in 3-D is 9 coupled first order PDE's for the three displacements and six independent components of the stress tensor (don't forget $\boldsymbol{\sigma}$ is symmetric such that $\sigma_{ij} = \sigma_{ji}$). These equations form the basis for the most common forms of solution of the elastic wave equations.

It is also possible to use this approach to solve for *acoustic waves* in a material that cannot support shear (i.e. fluids with $\mu = 0$). In this case stress is just given by a scalar pressure

$$\sigma_{i,j} = -P\delta_{ij} \tag{2.4.10}$$

If we also define the *dilation rate* $\mathcal{D} = \boldsymbol{\nabla}\cdot\mathbf{V}$ as the rate of expansion (or contraction) then by substituting (2.4.10) into (2.4.8) and taking the divergence of this equation we get.

$$\frac{\partial \mathcal{D}}{\partial t} = -\boldsymbol{\nabla}\cdot\frac{1}{\rho}\boldsymbol{\nabla}P + \boldsymbol{\nabla}\cdot\frac{\mathbf{f}}{\rho} \tag{2.4.11}$$

and Eq. (2.4.9) becomes

$$\frac{\partial P}{\partial t} = -\lambda\mathcal{D} \tag{2.4.12}$$

Alternatively, we can combine Eqs. (2.4.11) and (2.4.12) into a a single second order equation for the pressure (or dilation rate)

$$\frac{1}{\rho c^2}\frac{\partial^2 P}{\partial t^2} = \boldsymbol{\nabla}\cdot\frac{1}{\rho}\boldsymbol{\nabla}P - \boldsymbol{\nabla}\cdot\frac{\mathbf{f}}{\rho} \tag{2.4.13}$$

where $c = \sqrt{\lambda/\rho}$ is the acoustic wave speed.

Figure 2.5 shows the behaviour of the pressure field for a calculation that uses a *pseudo-spectral* technique to model the behaviour of a seismic pulse in a layered sedimentary basin with salt in it (Correa, pers. comm). We will visit the numerical tricks a and traps of this problem later but suffice it to say that the real difficult part of this problem is implementing useful boundary conditions. However, given the model, synthetic seismograms and record sections can be constructed that can be compared with data to understand what features are diagnostic (Fig. 2.6)

2.5 Flow in porous media

We've done fluids... We've done solids... now it's time to talk about that murky region where fluids and solids interact. In the Earth there are a large number of problems that can be described by the interaction of a low viscosity fluid (water, oil, gas, magma) in a permeable (and possibly deformable) matrix. First we will discuss the classic equations for flow in rigid porous media. Then section 2.5.2 will develop the equations for flow in deformable porous media, in the context of the the most drastic of these problems, magma migration from the convecting mantle. Finally we show that all the standard problems of flow in rigid or elastic media (hydrology, fluid flow in sedimentary basins) can be derived from this more general framework.

2.5.1 Rigid porous media

Darcy's Law is the classic, empirically derived equation for the flux $\mathbf{q}$ of a low viscosity fluid in a permeable matrix and can be written

$$\mathbf{q} = \frac{k_\phi}{\mu}\left[\boldsymbol{\nabla}P - \rho_f\mathbf{g}\right] \tag{2.5.1}$$

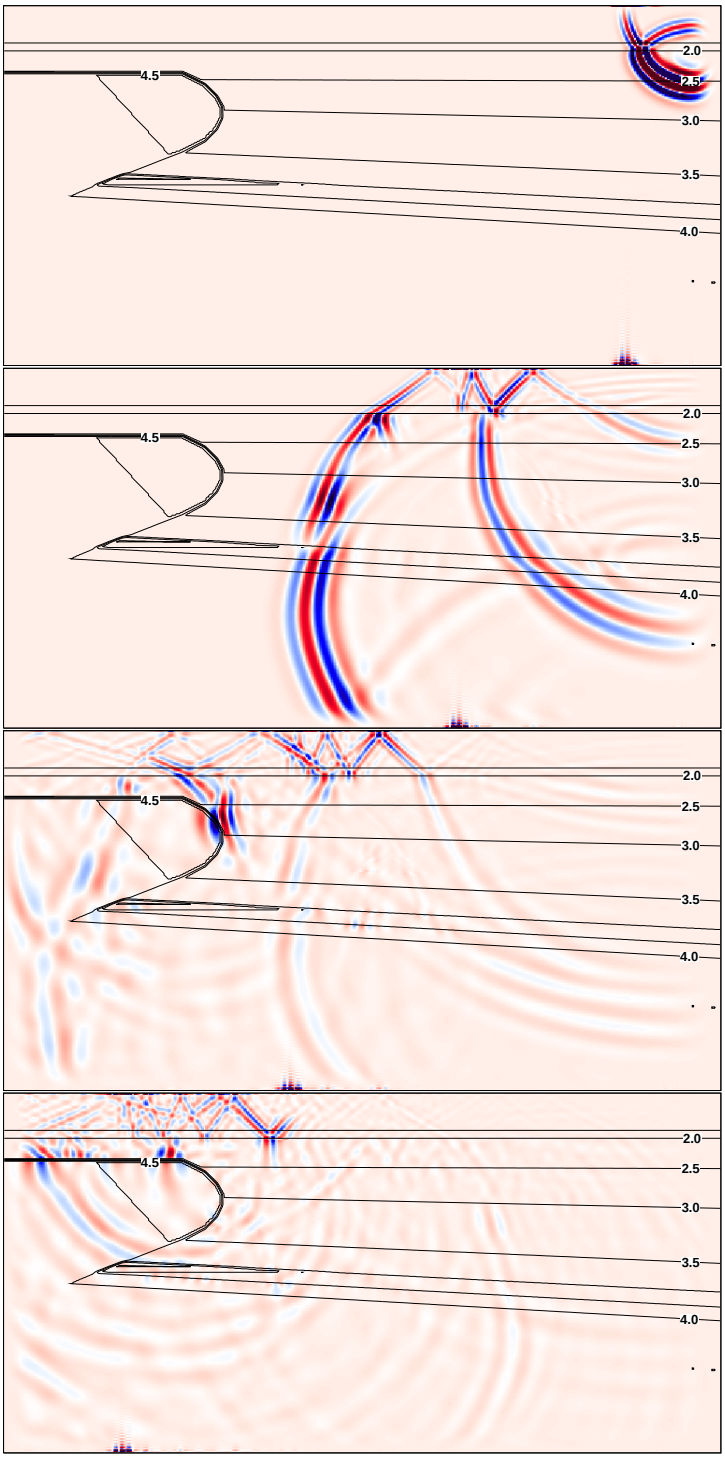


Figure 2.5: Evolution of pressure from a single shot in a layered sedimentary basin overlain by water.. Time increases from top to bottom. High pressure regions are red (like you can see this), low pressure regions are blue and lines show P-wave velocity structure. The high velocity blob to the left is a region of salt the top layer is water.. This calculation was done by Gustavo Correa using a pseudo-spectral technique which is highly accurate for wave problems.

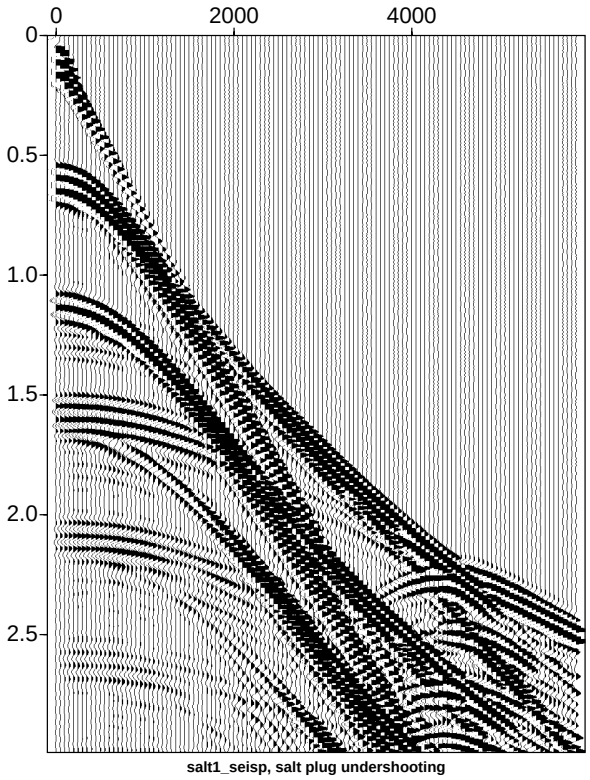


Figure 2.6: Synthetic shot gather from calculation show in Figure 2.5. Again this calculation was done by Gustavo Correa at LDEO.

where k_ϕ is the macroscopic permeability of the medium (and can be a tensor), μ is fluid viscosity, P is the fluid pressure and ρ_f is the fluid density. This equation assumes that flow in the pores or cracks of the medium is essentially laminar and provides the *average* flux through a representative area that is larger than the pore scale and smaller than the scale of significant permeability variation (if such a scale exists). Various approaches have been used to justify this rule from first principles (e.g. see Dagan [8]) but it generally seems to work. At any rate the principal unkown in all of these problems is the proper functional form for the permeability which can vary by orders of magnitude over relatively small distances. Most of hydrology and reservoir modeling is concerned with coming up with reasonable permeability structures, not actually solving the equations.

One way to solve these equations however is to note that, in a rigid medium, if the fluid is incompressible and there is no significant mass transfer between solid and liquid, then

$$\boldsymbol{\nabla} \cdot \mathbf{q} = 0 \tag{2.5.2}$$

because whatever fluid enters a volume must come out. Substituting in Eq. (2.5.1) and rearranging yields a modified Poisson problem for the fluid pressure

$$\boldsymbol{\nabla} \cdot \frac{k_\phi}{\mu} \boldsymbol{\nabla} P = \boldsymbol{\nabla} \cdot \frac{k_\phi \rho \mathbf{g}}{\mu} \tag{2.5.3}$$

which says that the fluid pressure gradient must adjust to balance the buoyancy forces driven by gravity. This equation is another boundary value problem, much like the standard Poisson problems we saw in Section 2.2. However, because the permeabilities are not, in general, constant, many of the rapid methods that can be used for solving $\nabla^2\Phi = f$ cannot be used. However, iterative *multigrid* methods (Chapter 9) can often be used with equal efficiency on either kind of problem. Figure 2.7 shows the 2-D flow field through a rigid and slightly elastic porous media with heterogeneous permeability.

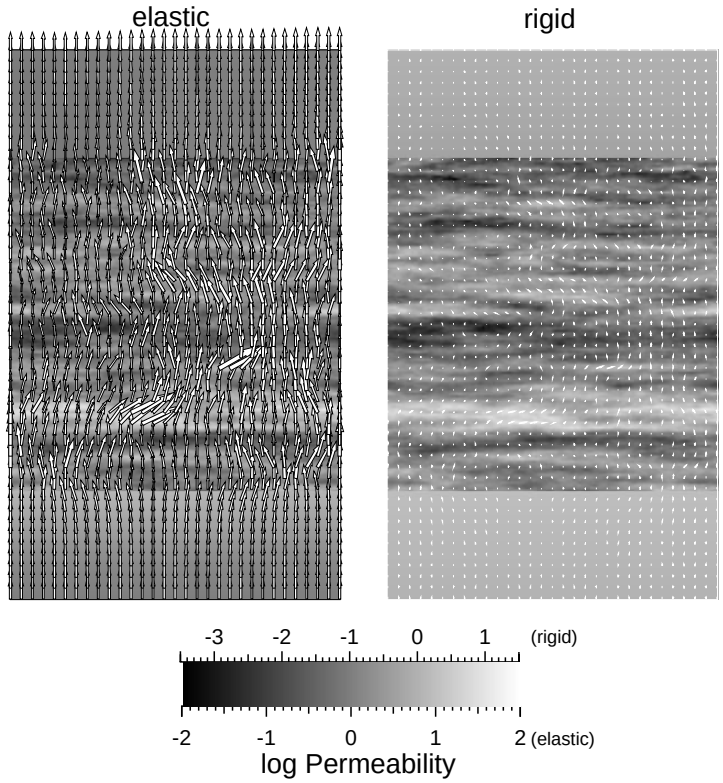


Figure 2.7: Flow field through a heterogeneous porous medium.

2.5.2 Deformable porous media: magma migration

Flow in rigid and elastic media is useful for problems in hydrology and crustal fluid flow. However, The melting and motion of partially molten rocks is a fundamental feature of plate tectonics and controls the geochemical evolution of the planet. To understand the behaviour of partially molten rock in the mantle (e.g. for regions beneath mid-ocean ridges, subduction zones and mantle plumes) requires a theory that has, at the very minimum, four important properties. The system needs at least two phases (solid and liquid), there must be significant mass-transfer between the solid and liquid (i.e. melting and freezing), the solid must be permeable at some scale, and the solid in the mantle must be viscously deformable so that, in

the absence of melting the theory is consistent with mantle convection (Section 2.2). Given these basic assumptions and a will to succeed, McKenzie [9] and others [10–12] derived a system of conservation equations for a two-phase mixture of a low viscosity liquid in a viscously deformable porous medium. McKenzie [9] provides a particularly detailed derivation that uses many of the concepts in Chapter 1. The general, dimensional equations for conservation of mass and momentum look like

$$\frac{\partial(\rho_f \phi)}{\partial t} + \boldsymbol{\nabla}\cdot(\rho_f \phi \mathbf{v}) = \Gamma \tag{2.5.4}$$

$$\frac{\partial[\rho_s(1-\phi)]}{\partial t} + \boldsymbol{\nabla}\cdot[\rho_s(1-\phi)\mathbf{V}] = -\Gamma \tag{2.5.5}$$

$$\phi(\mathbf{v}-\mathbf{V}) = \frac{-k_\phi}{\mu}\left[\boldsymbol{\nabla}P - \rho_f \mathbf{g}\right] \tag{2.5.6}$$

$$\boldsymbol{\nabla}P = -\boldsymbol{\nabla}\times[\eta\boldsymbol{\nabla}\times\mathbf{V}] + \boldsymbol{\nabla}\left[(\zeta+4\eta/3)\boldsymbol{\nabla}\cdot\mathbf{V}\right] + \mathbf{G} - \bar{\rho}\mathbf{g} \tag{2.5.7}$$

$$k_\phi \sim \frac{a^2\phi^n}{b} \tag{2.5.8}$$

Where ρ_f, ρ_s are the melt and solid densities, ϕ is the volume fraction of melt (porosity), **v** and **V** are the melt and solid velocities and Γ is the total rate of mass transfer from solid to liquid. k_ϕ is the permeability which is a non-linear function of porosity (Eq. 2.5.8), μ is the melt viscosity, P is the fluid pressure and **g** is the acceleration due to gravity. Finally, η is the solid shear viscosity, $(\zeta + 4\eta/3)$ is the combination of solid bulk and shear viscosity that controls volumes changes of the matrix, $\mathbf{G}(\eta, \mathbf{V})$ are the cross-terms that arise for non-constant shear viscosity (and vanish if η is constant) and $\bar{\rho} = \rho_f\phi + \rho_s(1-\phi)$ is the mean density of the two phase system. Equations (2.5.4) and (2.5.5) conserve mass for the melt and solid respectively and allow mass-transfer between the phases. Equation (2.5.6) governs the separation between melt and solid and Eq. (2.5.7) governs stress-balance and deformation of the solid phase.

So far, this is the ugliest set of equations we've had to deal with yet but it's surprising what a bit of analysis (and ten years of banging your head against a wall) can do to bring out the behaviour in what was essentially a new problem in Earth science. The first trick was to realize that Eqs. (2.5.4)–(2.5.8) can be rewritten in a more tractable from that highlights the essential physics. All the goodies here are described in gory detail by Spiegelman [13, 14].

Equations in potential form

The key feature of these equations is that the solid phase can deform in two fundamentally different ways. It can shear and convect like the standard incompressible convection equations; however, it can also compact or expand to expel (or inflate with) melt. One way to explicitly separate these two kinds of behaviour is is to use *Helmholtz' Theorem* that says that any vector field can be decomposed into a incompressible and a compressible part, i.e.

$$\mathbf{V} = \boldsymbol{\nabla}\times\boldsymbol{\Psi}^{\boldsymbol{s}} + \boldsymbol{\nabla}\mathbf{u} \tag{2.5.9}$$

where $\mathbf{\Psi^s}$ is the stream-function as before and $\mathbf{u}$ is a scalar potential field. Remembering the basic vector calculus identities that $\boldsymbol{\nabla}\cdot\boldsymbol{\nabla}\times\mathbf{F} \equiv 0$ and $\boldsymbol{\nabla}\times\boldsymbol{\nabla}\mathbf{F} \equiv 0$ shows that the first term is incompressible and the second term is irrotational. Using these definitions and also defining the *compaction rate*

$$\mathcal{C} = \boldsymbol{\nabla}\cdot\mathbf{V} \tag{2.5.10}$$

and substituting into the 2-D equations with constant viscosities and densities yields the governing equations in *potential form*

$$\frac{\partial\phi}{\partial t} + \mathbf{V}\cdot\boldsymbol{\nabla}\phi = (1-\phi)\mathcal{C} + \frac{\Gamma}{\rho_s} \tag{2.5.11}$$

$$-\boldsymbol{\nabla}\cdot\frac{k_\phi}{\mu}(\zeta+4\eta/3)\boldsymbol{\nabla}\mathcal{C} + \mathcal{C} = \boldsymbol{\nabla}\cdot\frac{k_\phi}{\mu}\left[\eta\boldsymbol{\nabla}\times\nabla^2\psi^s\mathbf{j} - (1-\phi)\Delta\rho g\mathbf{k}\right] + \Gamma/\rho_s \tag{2.5.12}$$

$$\nabla^2\mathcal{U}^s = \mathcal{C} \tag{2.5.13}$$

$$\nabla^4\psi^s = \frac{g}{\eta}\frac{\partial\rho}{\partial x} \tag{2.5.14}$$

Now doesn't that make you feel much better.

Scaling

Some solutions

Some more solutions

2.6 Geochemical Transport/Reactive flows

$$\frac{\partial\rho_f\phi c_i^f}{\partial t} + \boldsymbol{\nabla}\cdot\left[\rho_f\phi c_i^f\mathbf{v}\right] = \boldsymbol{\nabla}\cdot\phi\mathcal{D}_i^f\boldsymbol{\nabla}c_i^f + \sum_{j=1}^{J} c_{ij}^{f*}\Gamma_j \tag{2.6.1}$$

$$\frac{\partial\rho_s(1-\phi)c_i^s}{\partial t} + \boldsymbol{\nabla}\cdot\left[\rho_s(1-\phi)c_i^s\mathbf{V}\right] = -\sum_{j=1}^{J} c_{ij}^{f*}\Gamma_j \tag{2.6.2}$$

where c_i^f, c_i^s are the concentration of component i in the melt and solid respectively. $\mathcal{D}_i^f$ is the diffusivity (or dispersivity) of component i in the melt (we assume negligible solid diffusion) and c_{ij}^{f*} is the concentration of component i in the fluid that is produced by reaction j. Γ_j is just the rate of mass transfer for reaction j. Specific forms for different types of reactions will be introduced below. To make Eqs. (2.6.1)–(2.6.2) consistent with Eqs. (2.5.4)–(2.5.5) and the property that the sum of all concentrations in any phase must add to 100% (e.g. $\sum_{i=1}^{N} c_i^f = \sum_{i=1}^{N} c_i^s = \sum_{i=1}^{N} c_{ij}^{f*} = 1$) requires that the total mass transfer rate be

$$\Gamma = \sum_{j=1}^{J}\Gamma_j \tag{2.6.3}$$

and that $\sum_{i=1}^{N}\boldsymbol{\nabla}\cdot\phi\mathcal{D}_i^f\boldsymbol{\nabla}c_i^f = 0$ because only $N-1$ concentrations can freely diffuse. The final component must be anti-diffusive to conserve mass. For extensions to more complicated multi-phase systems see [15].

Bibliography

[1] J. Gleick. Chaos, Penguin, 1987, Q172.5.C45G54.

[2] E. N. Lorenz. Deterministic non-periodic flow, J. Atmos. Sci. 20, 130, 1963.

[3] A. E. Gill. Atmosphere-ocean dynamics, International geophysics series, Academic Press, New York, 1982.

[4] J. Y. K. Cho and L. M. Polvani. The morphogenesis of bands and zonal winds in the atmospheres on the giant outer planets, Science 273, 335–337, Jul 1996.

[5] S. E. Zebiak. Tropical atmosphere-ocean interaction and the El Niño/Southern Oscillation phenomenon, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1985.

[6] M. A. Cane, S. E. Zebiak and S. C. Dolan. Experimental forecasts of el nino, Nature 321, 827–832, 1986.

[7] S. E. Zebiak and M. A. Cane. A model el niño-southern oscillation, Mon. Wea. Rev. 115, 2262–2278., 1987.

[8] G. Dagan. Flow and Transport in Porous Formations, Springer-Verlag, Berlin, 1989.

[9] D. McKenzie. The generation and compaction of partially molten rock, J. Petrol. 25, 713–765, 1984.

[10] D. R. Scott and D. Stevenson. Magma solitons, Geophys. Res. Letts. 11, 1161–1164, 1984.

[11] D. R. Scott and D. Stevenson. Magma ascent by porous flow, J. Geophys. Res. 91, 9283–9296, 1986.

[12] A. C. Fowler. A mathematical model of magma transport in the asthenosphere, Geophys. Astrophys. Fluid Dyn. 33, 63–96, 1985.

[13] M. Spiegelman. Flow in deformable porous media. part 1. simple analysis, J. Fluid Mech. 247, 17–38, 1993.

[14] M. Spiegelman. Flow in deformable porous media. part 2. numerical analysis — the relationship between shock waves and solitary waves, J. Fluid Mech. 247, 39–63, 1993.

[15] C. I. Steefel and A. C. Lasaga. A coupled model for transport of multiple chemical species and kinetic precipitation/dissolution reactions, Am. J. Sci. 294, 529–592, 1994.

Chapter 3

. . . and how to solve them: A survey of techniques

This section will describe a general approach to solving quantitative problems and provide a quick survey of possible methods, resources and directions of attack.

Getting Started: Formulating the problem

1. Start with a physical problem that you are interested in.
2. Generate appropriate conservation equations.
3. Scale and approximate the equations until they are tractable.
4. Consider appropriate boundary and initial conditions
5. At this point you're problem will probably fall into one or more of several categories
   (a) Sets of Ordinary Differential Equations (ODE's) with a single independent variable
   (b) Initial Value problems (PDE's)
   (c) Boundary Value problems (PDE's)
6. Now figure out how to solve it

Fundamental Tools 1: Paper and Pencil

1. Find an exact analytical solution: If you're very lucky someone extremely clever (or yourself) will have worked out a solution to your problem (or one close enough for jazz). This is rare but always the best approach. If it's a classical problem (and most were worked out last century) it should exist in a book, use Columbia's LibraryWeb).
2. Find an approximate analytical Solution: Often by setting coefficients to constants, considering small perturbations, or just throwing out terms *a priori*, your equations can be muscled into something that is straightforward (or someone has already solved). If the problem is essentially linear these solutions will be quite good. If the problem is non-linear,

BEWARE, stopping too soon can lead to dull solutions. Nevertheless, approximate solutions are very important for checking numerical solutions.

3. Produce a simple scaling argument: Very powerful technique that is always correct to a factor of $2, \pi, 5$ or 10. You will have to do this at some point in the process, best to do it before you write a single line of code.

Fundamental Numerical Tools: algorithms and discretization If at this point you still need a numerical solution you will need to choose a method. All numerical solutions are discrete approximations to continuous solutions. Most of the differences in techniques arise from the choice of how to discretize the problem. Some common methods and their pros and cons are.

1. Finite Differences: Approximate a function at a point by a truncated Taylor Series and combine the series of adjacent points to approximate the governing equations.

   pros Simple, efficient easy to code in 1, 2 and 3-D. Good for simple geometries, static meshes and problems with smoothly varying properties. Fastest way to get a feel for the problem.

   cons Not as good for complex geometries with sharp changes in material properties. Not particularly good for tracking internal boundaries

2. Finite Elements: Approximate a function as a set of piecewise continuous "elements" where the solution is approximated locally by some interpolating function. Link all the elements together and require that the error be a minimum in a global sense.

   pros Very good for problems with complex geometries, strongly varying internal properties or a need to track internal boundaries and materials. Also good for local refinement in resolution and Lagrangian moving-mesh problems.

   cons Significantly greater computational complexity. Standard techniques can be very expensive (computationally) to scale up to larger problems. Finite elements have just as many (or more) numerical artifacts as finite difference. In general, except for simple problems finite elements are not exactly a code-it-yourself problem and there are publicly and commercially available packages.

3. Finite Volumes/Finite Volume Elements: a hybrid between finite element and finite difference techniques. Discretizes space as a set of discrete volumes that trade fluxes at their boundaries and considers the volume averages of properties within the volume. Finite Volume Elements also uses some of the finite element formalism to define interpolation schemes between nodes. Finite Volumes is the natural extension of the conservation approach for coming up with discrete equations.

pros Combines the ease of finite difference with the formalism of finite elements. Is a useful approach for deriving discrete equations in stranger geometries and more general problems. Can be naturally conservative.

cons This technique has all of the numerical artifacts of any discretization. Also usually requires that material properties be smoothly varying over the mesh spacing.

4. Spectral and Pseudo-Spectral techniques: Rather than discretizing the functions in space, the functions are replaced by truncated Fourier series (discretized in frequency). Usually only the spatial terms are transformed to frequency, the time dependent terms are handled as ODE's or with finite difference techniques.

   pros Good for wave problems. Can be very fast if they make use of FFT's and orthogonal functions. If the problems are band width limited, these solutions are effectively continuous up to the highest frequencies maintained. Can also often reduce a PDE to a set of ODE's that can be solved efficiently (e.g. the Lorenz Equations).

   cons Usually only works easily for constant coefficients, regular geometries and smooth functions.

5. Particle-based/characteristic methods/Semi-lagrangian methods: Uses much of the physics of hyperbolic systems to track particles around on a regular grid. Combines the best of random particle methods and Eulerian grid based methods. Examples include Semi-Lagrangian methods and "Contour Surgery" methods.

   pros Extremely good for for advection dominated problems. Low dispersion, proper upwind structure, can move relatively steep fronts through regular grids, has **NO** Courant stability condition so it can decouple time steps from spatial resolution. Excellent for open *outflow* boundaries One of my favourite schemes.

   cons The simplest form is not conservative or positive definite. Can be computationally expensive... requires significant per point computation due to heavy interpolation. probelmatic around complex boundaries. Slightly more difficult to parallelize.

6. Microscopic techniques: In addition to standard finite something techniques which are used to solve macroscopic continuum equations. Some problems can also be solved by approximating microscopic principles. Several common approaches are "Lattice Gas cellular Automata", molecular dynamics, and granular media which all approximate the motion of large sets of particles that are constrained either by "interaction rules" on a fixed grid or by "potentials".

   pros Good for wave problems. Very good for large parallel machines. Can handle very complicated internal structures with simple rules.

   cons Need large number of particles. Not a very good technique for static or quasi-static problems. In cellular automata it is some-

times quite difficult to relate the interaction rules to the macroscopic constitutive relations.

Scrounge for code When you have decided on a technique, it can be very worthwhile to look for already written pieces of source code to implement it (particularly for complex algorithms). These pieces should not be treated as black-boxes but understood and tested as if it were your own code. Nevertheless, the typos you save may be your own. There are several useful sources of code for many problems and many of them are free and available on the network.

Numerical Recipes Modestly priced, well documented codes for just about everything but PDE's. Comes in most flavours of languages (FORTRAN, C, Pascal and even Basic). Probably one of the most accessible books on the subject but beware some of the codes are flaky (but improved in the second edition).

Netlib `http://www.netlib.org/` Netlib is an on-line library of public domain numerical codes stored at the University of Tennessee and Oak Ridge National laboratories. Netlib interface also provides access to High-performance computing software (HPC-Netlib), sources for parallel processing software, a high-performance computing data-base and other useful things.

PetSc the Portable, Extensible Toolkit for Scientific Computation. An enormous package (over 15Mb) of scientific codes for both serial and parallel computers written in an Object Oriented style modular format. Version 2.0.24 is currently available at `http://www.mcs.anl.gov/petsc/`. Most finite difference algorithms can be found here plus many parallel iterative methods. Written in MPI (even the serial bits), code can seem somewhat unwieldy but could well be the shape of the future.

Commercial software There are several commercially available software libraries, most Notable are IMSL, NAG and IBM's ESSL. These are very high quality, extremely efficient, tested routines but you can't access the source code. Useful for production work but not portable. Xnetlib has information on how to contact NAG. I believe Lamont owns the IMSL libraries and we have both ESSL and ParallelESSL on the SP2.

Other sources: the library Never hurts to see if someone has already done your problem. The engineering library at Columbia has quite a large numerical methods collection. When you have a better idea of what you are looking for use Columbia's LibraryWeb to go find it anywhere.

A note on Languages You can program in pretty much any language you please, however much of the classic source code in earth sciences is written in FORTRAN and to a lesser extent in C. These are the two principle languages of scientific computing. If you know Basic learn fortran. If you know pascal, learn C. In general, for simple array manipulation, FORTRAN is simpler, has better compilers and more flexible

array handling in subroutines. C is better for complex data structures and string handling. However, the compilers are converging and new languages such as Fortran90, High-performance Fortran or C++ are rapidly taking hold.

Put it together

1. Pick a platform: There is almost a continuous array of machines in terms of price, performance and ease of use (the latter is often inversely proportional to the first two). There are only three things to keep in mind when picking the right machine for the job
   (a) You will often spend more time in development and testing than in running so finding a good programming environment is often more important than finding a fast box.
   (b) When you are in production, any job that takes more than about 12 hours is probably too big for the machine you're working on (unless it's the biggest thing you can find). Don't spend 1000 hours computing on a macplus 512 when you can do the same problem in about 20 minutes on an UltraSparc (or 600Mhz PentiumIII, or SGI or IBM...). On the other hand, the principal time you don't want to waste is your own (so see the previous point).
   (c) Therefore, keep your code flexible, conservative and portable!

   Useful and available machines at Lamont are Macs, an increasing number of Wintel/Linux boxes, Sun Workstations in all flavours and other Unix machines (SGI's) including our 30-node IBM SP2 distributed memory parallel-but-rapidly-becoming-obsolete "SuperComputer". For bigger machines (CRAY's and really big parallel super-computers) you need to go out of house (but if you can't get going with what we have here, you have a seriously ugly problem).
2. The development cycle: Compile, Debug, Run (repeat ad nauseum). There are several tools that are useful for shortening the development time of codes.

   Development Environment While everything can be done with simple text-editors. `emacs` or `Xemacs` provides a powerful (and free) development environment for writing code. It understands many keywords, keeps your indentation clean and allows you to interact with the compilers and debuggers in a much more natural way.

   Numerical debugger The most important tool, though, is a good interactive debugger which allows you to step through the code and see how it works and where it blows up. On the Suns we will use the `Workshop debugger` which is incorporated into Xemacs.. On the SP2 use `xldb`.

   Visual debuggers A picture is often worth a million numbers. So a quick look at the output visually can often lead to spotting bugs much faster. Workshop actually allows visualization of 1-D and

2-D arrays directly in the debugger. For visualization of output however try.

1-D problems Matlab (available on all platforms?) Suns/Linux: xmgrace; Macs: Excel

2-D problems Suns: Matlab, Spyglass transform, GMT system, Envi, various imagetools (pbmtools, xv, xanim for animations,gimp). Macs: Matlab, Spyglass transform, ??

3-D problems Suns : AVS. SP2 DX, SGI DataExplorer. Macs: Spyglass Dicer (if still available)

In general we will do most of our work in Matlab which is great but not generally available without moolah

3. Production: When it works and is bug free (or close enough for jazz) you still need to treat your original problem as a Numerical experiment and explore *parameter space* which is spanned by the values of your dimensionless parameters. The more you understand the less time you will waste in production!

4. Understanding: Once you have a full numerical description of the behaviour of your numerical system you still need to accomplish two important tasks.

   (a) Prove to yourself (and others) that the codes are doing what they should be doing.

   (b) Reduce the numerical results to a simple parameterization that can be used by anyone without typing a single character.

   SCALING ARGUMENTS are your best bet for both of these tasks so it's back to paper and pencil.

5. Completion and writing up: When you understand your problem so well that you don't have to model it anymore it's time to write it up. Only at this point should you discuss the relevance of the model to reality. A good solution stands on it's own and should be discussed on it's own (you never can be wrong this way). Whether it is a good model of reality depends on the available data. Nevertheless, where this approach is most useful is when there is a minimum of data and you need some quantitative conception of what to look for. This is the most useful contribution of modeling.

Chapter 4

Solution of ordinary Differential Equations

Highly suggested Reading

Press *et al.*, 1992. Numerical Recipes, 2nd Edition. Chapter 16. (excellent description, can't be beat). See also chapter 17.

4.1 What they are and where they come from

Ordinary differential equations (ODE's)[1] are equations where all variables are functions of only a single independent variable such as time or position. The simplest ODE (which we will use as a model problem) is the equation for radioactive decay of some element.

$$\frac{dc}{dt} = -\lambda c \tag{4.1.1}$$

where c is the mean concentration of the element in some volume and λ is the decay constant. Equation (4.1.1) states that the rate of the decay of the element is proportional to its current concentration and has the analytic solution $c = c_0 \exp(-\lambda t)$ where c_0 is the initial concentration at time $t = 0$. Figure 4.1 demonstrates the basic numerical problem inherent in solving Eq. (4.1.1) (with $\lambda = 1$). The right hand side of (4.1.1) forms a *direction field* that shows how any given concentration (and time) will change. For a given initial condition, the direction field should form the tangents to a trajectory that is the true solution. I.e. the trick is to start at some specific concentration c_0 at $t = 0$ and pick your way carefully through the direction field to some final concentration at time t.

Before discussing how we actually solve this sort of problem, it is worth discussing other sources of ODE's. Single equations for higher order ODE's can also be derived such as

$$\frac{d^2 f}{dx^2} + p(x)\frac{df}{dx} + q(x) = 0 \tag{4.1.2}$$

[1] What's ordinary about them is beyond me, except that maybe they are readily solved by lots of methods and are therefore dull to mathematicians.

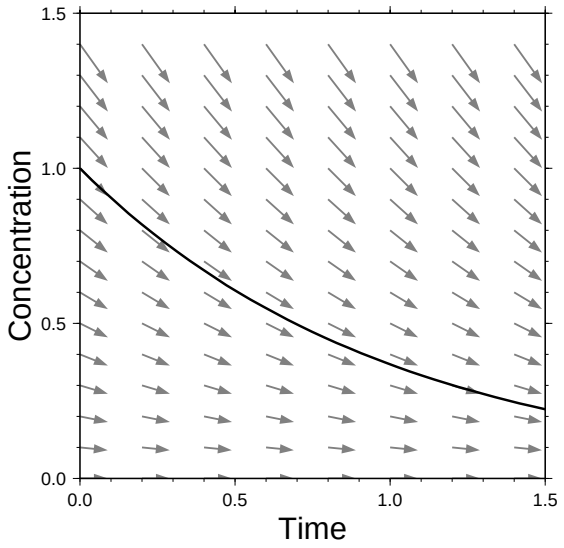


Figure 4.1: The direction field and solution trajectory for the simplest ODE $dc/dt = -c$. The basic numerical approach is to note that the derivative of the function is known for all values of concentration and time (and forms the field of arrows). The problem is to shoot your way *accurately* and efficiently from point to point. This is the entire raison d'etre of ODE solvers.

for example from problems of coupled blocks and springs. Fortunately, one of the lovely features of higher order ODE's is that they can always be rewritten as systems of first-order ODE's by defining *auxiliary variables*, e.g. Eq. (4.1.2) can also be written as

$$\frac{df}{dx} = g \tag{4.1.3}$$

$$\frac{dg}{dx} = -p(x)g - q(x) \tag{4.1.4}$$

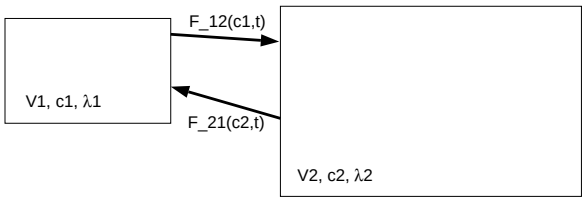
so if Eq. (4.1.1) can be solved for one variable, the systems can be solved using the same techniques for multiple variables (if all the variables are known at $t = 0$).

A more physical source for systems of ODE's are *Box Models* (see Chapter 1) where each box or reservoir contains the *average* value of some species (chemical, biological or fluid mechanical) and the boxes are connected by fluxes and may have sources or sinks. For example if we treat the upper and lower mantle as two reservoirs of volumes V_1 and V_2 that are connected by fluxes F_{12} and F_{21}, e.g.

Then we can write a conservation equation for the average concentration of a radioactive species in both reservoirs as

$$\frac{dc_1}{dt} = \frac{1}{V_1}\left[F_{21}(c_2,t) - F_{12}(c_1,t)\right] - \lambda c_1 \tag{4.1.5}$$

$$\frac{dc_2}{dt} = -\frac{1}{V_2}\left[F_{21}(c_2,t) - F_{12}(c_1,t)\right] - \lambda c_2 \tag{4.1.6}$$



As long as the right hand side of these equations can be evaluated for any given concentrations and time, then the concentrations can be updated using a classic ODE solver. It is straightforward to extend this two box model to a multi-box or to predator-prey models etc.

Less obvious sources of ODE's come from *particle tracking* problems where we have some set of particles in an imposed flow field $\mathbf{V}$ and want to follow them around and record any changes of property that they might experience. In this case each particle can be described by a system of ODE's like

$$\frac{dc}{dt} = f(\mathbf{x}, t) \tag{4.1.7}$$

$$\frac{d\mathbf{x}}{dt} = \mathbf{V} \tag{4.1.8}$$

where $\mathbf{x}$ is the vector position of the particle and $f(\mathbf{x}, t)$ is a function that says how locally c changes with time. What is not obvious is that Eq. (4.1.7) is actually equivalent to the Lagrangian formulation of the simplest partial differential equation

$$\frac{\partial c}{\partial t} + \mathbf{V} \cdot \boldsymbol{\nabla} c = f \tag{4.1.9}$$

(we will discuss the relationship between ODE's and PDE's in a later section). In addition, ODE's can also be generated by *modal expansions* or *spectral methods*. If we can write a trial solution of our problem as a sum of known spatial functions (such as sines and cosines) with time-dependent amplitudes, i.e.

$$c(\mathbf{x}, t) = \sum_{n=1}^{N} a_n(t) f_n(\mathbf{x}) \tag{4.1.10}$$

then substitution into more general partial differential equations will yield a (generally non-linear) mess of ODE's for the N amplitudes i.e.

$$\frac{d\mathbf{a}}{dt} = \mathbf{g}(\mathbf{a}) \tag{4.1.11}$$

This is the approach used to derive the Lorenz equations which are a simplified version of Rayleigh-Benard convection (i.e. convection in a sheet). This approach is also often used in finite-element techniques for time-dependent problems.

No matter where they come from, however, the general form of ODE's that we will be dealing with is

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}, t), \qquad \mathbf{y} = \mathbf{y}_0, t = t_0 \tag{4.1.12}$$

where **y** is a vector of state variables(i.e. a point in state space) and **f** is a function that describes how the position will change as a function of time and state. As long as the right hand side of these equations can be evaluated at a point, then the numerical problem reduces to trying to step from point to point as accurately and efficiently as possible (see Fig. 4.1).

4.2 Basic Stepping concepts and algorithms

More mathematically, the *initial value problem* reduces to solving the integral equation

$$\mathbf{y} = \mathbf{y}_0 + \int_{t_0}^{t} \mathbf{f}(\tau, \mathbf{y}(\tau))d\tau \tag{4.2.1}$$

Actually, the numerical problem only requires evaluating the integral in (4.2.1) for a single stepsize h (i.e. for $t = t_0 + h$), because in initial value problems, if we can solve the problem for a single step, we can just restart the problem from the end-point and repeat ourselves till we get to where we want to go. Thus the problem reduces to finding a cheap but accurate approximation for the integral

$$k = \int_0^h f(\tau, y(\tau))d\tau \tag{4.2.2}$$

Now the cheapest (but least accurate approximation is an *Euler Step* where we approximate the integral just using the value of f at time 0. i.e. $k \sim hf(t_0, y_0)$ so that our approximation becomes

$$y(t_0 + h) = y_0 + hf(t_0, y_0) \tag{4.2.3}$$

Unfortunately, if we compare that to the Taylor series expansion of y

$$y(t_0 + h) = y(t_0) + hf(t_0, y_0) + \frac{h^2}{2}\frac{d^2y}{dt^2} + \ldots \tag{4.2.4}$$

We see that if the function has any curvature, even a moderate step-size h will cause an error that is only of order h smaller than our approximation (which is why the $O(h^2)$ term is known as a *first order error*). More physically, inspection of Fig. 4.2a shows that an Euler step is simply a linear extrapolation of our derivative at t_0. Clearly, if our true solution is anything but a straight line, we need to take a very small step to stay on course.

So what to do? Qualitatively, it would seem that the more information we have about our function the better we can approximate the integral Eq. (4.2.2). For example, we ought to be able to get some curvature information with two function evaluations and the *mid-point method* does just that. Fig. 4.2b shows that the mid-point first takes Euler step with step-size $h/2$ then uses the derivative information at the midpoint. This algorithm can be written

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + h/2, y_n + k_1/2) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned} \tag{4.2.5}$$

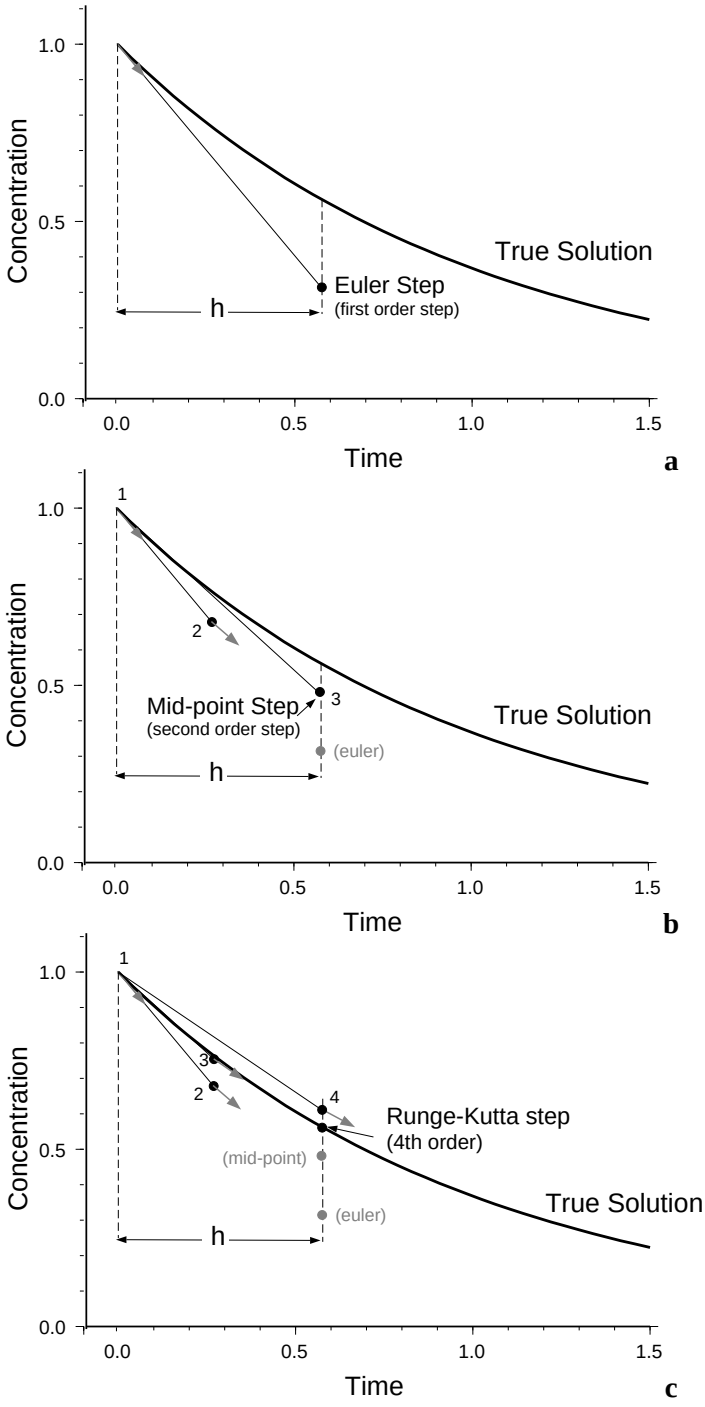


Figure 4.2: Simple stepping schemes of increasing order. (*a*) Euler Step: just uses derivative information at $t = 0$ and extrapolates linearly. Highly inaccurate for functions with curvature. (*b*) Mid-point scheme: 2nd order accurate stepper, uses two derivative evaluations to gain information on curvature. (*c*) Standard 4th order Runge-Kutta scheme: uses 4 function evaluations for high order accuracy.

which is a second order method.

It follows that if we gain more information about our function we can get even higher order schemes[2]. A classic is the *4th-order Runge Kutta* scheme which gets us 4th order accuracy with only 4 function evaluations (which turns out to be the turning point for RK schemes). This stepping scheme is illustrated in Fig. 4.2c and can be written

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + h/2, y_n + k_1/2) \\ k_3 &= hf(t_n + h/2, y_n + k_2/2) \\ k_4 &= hf(t_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5) \end{aligned} \tag{4.2.6}$$

(If you're really wondering, this scheme is related to Simpson's rule for integration of quadratic functions).

4.3 Getting clever: adaptive stepping

Equation (4.2.6) is a workable scheme that will get us from point A to B in a single step. The question is still "how big a step can we take?". Any good ODE integrator worth it's salt will sort this out for you by using *adaptive stepping*. The basic idea is that if we know the order of error as a function of step-size for any stepping scheme (and can monitor that error), we can adjust the step-size to keep the error within some tolerance. One simple approach is to use *step-doubling* which first takes a big step of size h and then starts from the same point but takes two half-steps of size $h/2$. In a perfect world, the two trial solutions should be the same but the actual truncation error on the big step is of order h^5 while the error on the two half-steps is of order $2(h/2)^5$. Comparing the difference between the two results gives a measure of the *relative truncation error* Δ which should still be

$$\Delta \propto h^5 \tag{4.3.1}$$

Thus, if our measured truncation error for a trial step h_0 is Δ_0, and we would really like our error to be some other Δ_1 then we should adjust our step-size to a h_1 such that

$$h_1 = h_0 \left| \frac{\Delta_1}{\Delta_0} \right|^{1/5} \tag{4.3.2}$$

Actually as Numerical Recipes discusses in detail, there are some slightly more practical ways to choose the new step-size but the idea is the same. All that's left now is to somehow specify the desired accuracy Δ_1. In the Numerical recipes routines, this is done by specifying a tolerance `eps` and some scale `yscale(i)` for each variable `y(i)` such that

$$\Delta_1 = \texttt{eps} \times \texttt{yscale(i)} \tag{4.3.3}$$

[2]Beware: higher order does not always mean higher accuracy unless your function is well approximated by high order polynomials

Moreover, they suggest that a handy trick for getting constant fractional error is to set `yscale(i)=abs(y(i))+abs(h*dydt(i))`, which protects you from zero-crossings. The result of this approach, is to make the scheme track the fastest changing variable (which is as it should be).

Beyond Step-doubling: embedded Runge-Kutta Step-doubling is rough and ready but costs eleven function evaluations for every good step h[3]. A more efficient but inscrutable technique uses *embedded Runge-Kutta* schemes such as the *5th-order Runge-Kutta Cash-Karp* scheme implemented in Numerical recipes. This scheme is 5th-order and uses six function evaluations per step. The peculiar part though is that there exists another combination of these 6 evaluations that is 4th order. Thus with 6 evaluations you get your cake and eat it too, i.e. you get your 5th order accuracy along with a measure of the truncation error. For rough and ready jobs, these schemes are quite robust and quick.

A note on the NumRec routines Numerical Recipes (Press *et al.*, 1992) provides a very nice set of ODE integrators which are easily adapted to most problems. In particular, they have taken a very modular approach to coding and have separated the routines into 3 levels. At the lowest level is the user supplied routine `derivs(t,y,dydt)` such as

```
c**********************************************************************
c decay1:  subroutine that returns the rate of decay of a radioactive
c   substance as a function of  concentration (and time)
c   c is concentration
c   dcdt is the change in c with time
c   t is time
c   here, dcdt= -c
c**********************************************************************

      subroutine decay1(t,c,dcdt)
      implicit none
      real t, c, dcdt

      dcdt= -c

      return
      end
```

which returns an array of values `dydt(i)` given a time `t` and a state-vector `y(i)`. Unfortunately, if your function requires additional parameters to evaluate the derivatives, these need to passed into the routine using *common blocks* (in Fortran or global variables in C). Ugly but necessary. Some examples are given in the problem set. Still, for the extra coding headache, these routines provide quite good flexibility.

Above the `derivs` level is the *algorithm* which takes one step without regard to accuracy. In Numerical Recipes, the standard RK algorithm Eq. (4.2.6) is implemented in `rk4` while the Runge-Kutta Cash-Karp scheme is in `rkck`. Above the algorithmic level comes quality control which adjusts the step size until the desired accuracy is reached. This *stepper* routine (e.g. `rkqs`) job is to try to take

[3] that's 3 steps with 4 evaluations per step minus 1 because they start in the same place

the largest step possible while remaining within the described tolerance. This is where the adaptive stepping is implemented and the routine will keep adjusting the step size to get to its goal. Finally, there are the *driver* routines (e.g. `odeint` or my version of it `odeintnc`). which keep track of the progress, say when to stop and possibly store intermediate information. In general, most user modifications go in the driver routines and they should be tailored to your problem.

4.4 Beyond Runge-Kutta: Bulirsch-Stoer methods

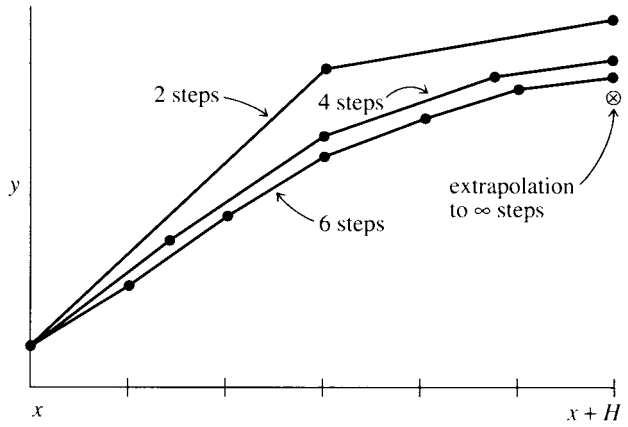


Figure 4.3: Richardson extrapolation as used in the Bulirsch-Stoer method. A large interval H is spanned by different sequences of finer and finer substeps. Their results are extrapolated to an answer that is supposed to correspond to infinitely fine substeps. In the Bulirsch-Stoer method, the integrations are done by the modified midpoint method, and the extrapolation technique is rational function or polynomial extrapolation. (figure and caption from Numerical Recipes, 2nd ed., p. 719)

Embedded Runge-Kutta with adaptive step-size control is a pretty solid ODE integrator that will get you through most problems without any trouble. In particular, if you have coarsely gridded data or the function is cheap to evaluate or you have internal singularities that need to be stepped carefully around then these schemes are just peachy. But of course you can always do better. One useful approach for problems that require high accuracy is the *Bulirsch-Stoer* method which actually approximates an infinite order scheme yet allows you to take very large steps. Sound too good to be true? Then check out Fig. 4.3.

The basic idea is that if we have a scheme that allows us to take a large step H with a variable number of smaller substeps, then as we increase the number of steps we form increasingly better approximations to the end solutions. Moreover, if the progression of approximate solutions is smooth and well behaved, we can actually *extrapolate* our series to the limit of taking an infinite number of sub-steps (i.e. if $h = 0$). The trick in the Bulirsch-Stoer method is to use a *modified mid-point* scheme to take the sub-steps. This scheme is an example of a *leapfrog or centered*

difference scheme where we take an Euler step for the first and last points but use a mid-point method to take us from point y_{n-1} to y_{n+1} using the derivative information at y_n (easier to show than to say). As discussed in Numerical Recipes, the important feature of the modified mid-point method is that the relative truncation error between successive trials with increasing numbers of substeps have only even powers of step-size so you tend to get two-orders of magnitude error reduction for only a few extra function calls. With the modular form of the Numerical recipes routines, it is just as easy to use BS methods as RK methods, you simply replace the algorithm routine with `mmid` and the stepper routine with `bsstep`. These routine implement a more complicated form of adaptive step-size control that is still based on the relationships between step size and truncation error inherent in the method. You don't really have to worry about how it works in detail. What you have to worry about is when not to use it. Basically, these schemes assume that function behaves very smoothly and has no internal singularities. All of the grace of this method will be lost if your function is rough or has a spike somewhere in the middle of it. For these problems go back to the Runge Kutta schemes.

4.5 Even more ODE's: stiff equations

The routines so far have never failed me but that may have more to do with the type of problems I solve where either the variables all change at similar rates or I have lots of time and computer resources. Sometimes, however, you may run into the problem of *stiff equations* where one or more of your variables change at a rate that is much faster than the thing you are interested in. An adaptive stepper routine could catch this but it forces your problem to evolve on the time scale of your most annoying variable rather than your most interesting one. Numerical recipes supplies a specific example (and we will see this phenomena in other situations as well). One standard approach is to use *implicit differencing schemes* which are more stable but more difficult to solve. Numerical Recipes discusses them in detail and provides routines for stiff equations.

Chapter 5

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

Selected Reading

Numerical Recipes, 2nd edition: Chapter 19

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

5.1 Introduction

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

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

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

5.2 Non-diffusive initial value problems and the material derivative

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.3 Grid based methods and simple finite differences

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

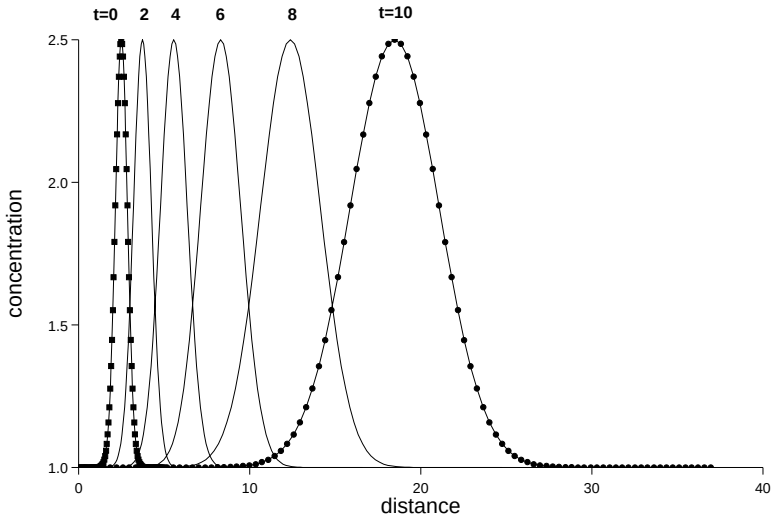


Figure 5.1: Evolution of gaussian initial condition in a velocity field $V = 0.2x$ This velocity field leads to stretching. Analytic solution is by method of characteristics

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

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

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

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

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

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

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

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

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

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

easily taken a step backwards to get

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

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

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

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

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

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

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

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

5.3.1 Another approach to differencing: polynomial interpolation

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

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

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

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

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

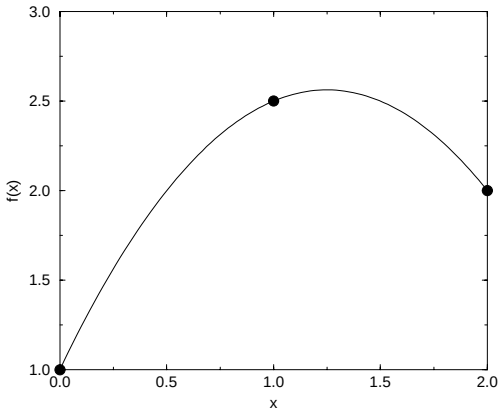


Figure 5.2: The second order interpolating polynomial that passes through the three points $(0\Delta x, 1)$, $(1\Delta x, 2.5)$, $(2\Delta x, 2)$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

for all points in the interval because the 2nd derivative of a parabola is constant. Generalizing to higher order schemes just requires using more points to define the stencil. Likewise it is straightforward to work out weighting schemes for unevenly spaced grid points (although these are often not as accurate).

5.3.2 Putting it all together

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

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

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

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

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

where

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

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

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

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

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

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

5.4 Understanding differencing schemes: stability analysis

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

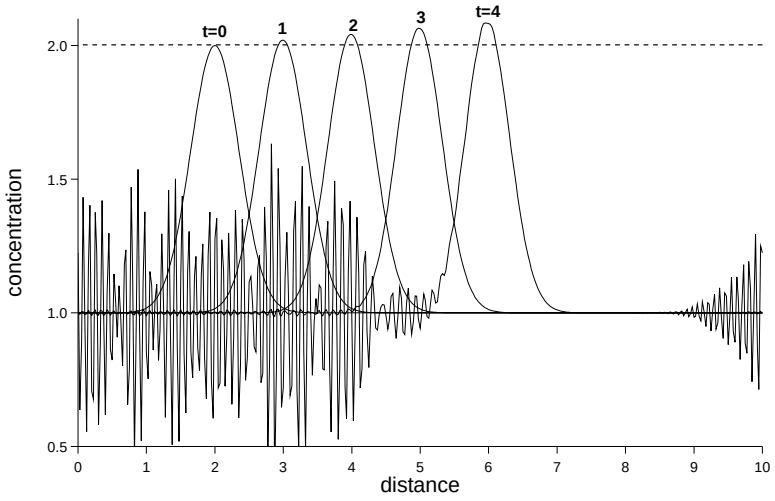


Figure 5.3: Evolution of a gaussian initial condition in a constant velocity field $V = 1$ using a FTCS scheme. A perfect scheme would have the initial gaussian just propagate at constant speed without changing its shape. The FTCS scheme however causes it to grow and eventually for high frequency noise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

5.4.1 Hirt's method

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

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

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

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

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

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

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

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

equation (with constant V) we get

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

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

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

Thus Eq. (5.4.4) becomes

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

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

5.4.2 Von Neumann's method

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

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

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

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

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

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

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

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

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

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

Thus

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

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

5.5 Usable Eulerian schemes for non-diffusive IVP's

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

5.5.1 Staggered Leapfrog

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

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

or as an updating scheme

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

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

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

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

or multiplying through by ζ we get the quadratic equation

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

which has the solution that

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

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

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

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

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

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

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

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

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

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

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

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

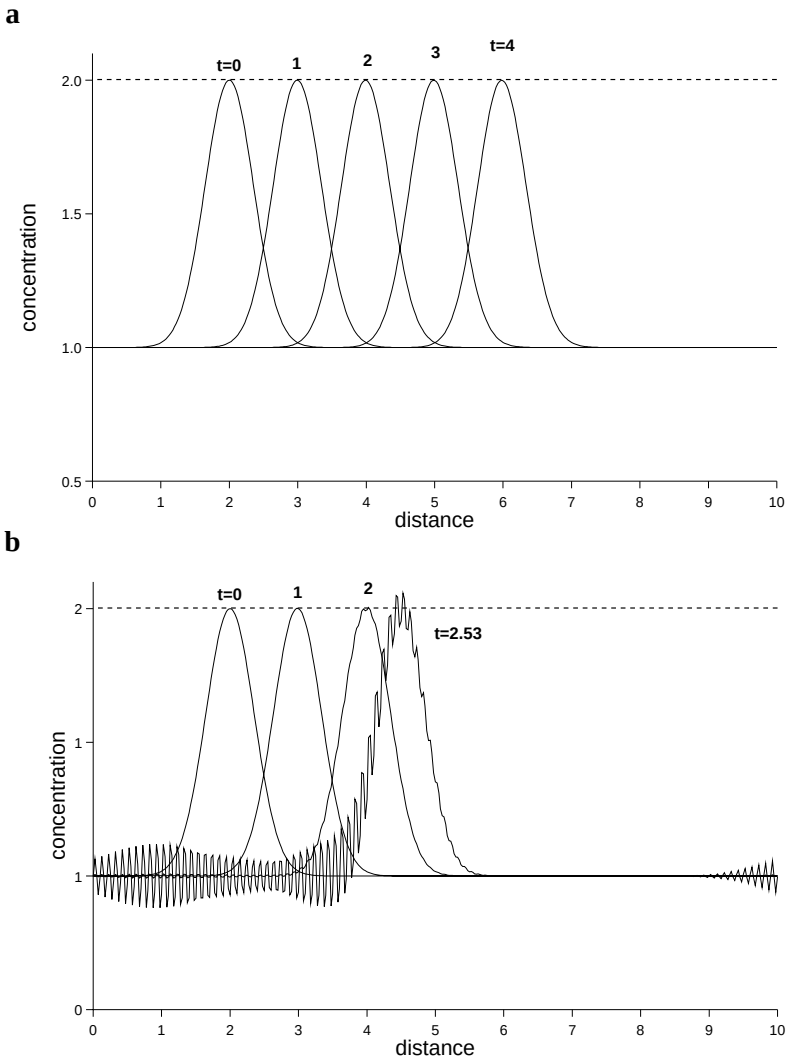


Figure 5.4: (a) Evolution of gaussian initial condition in using a staggered leapfrog scheme with $\alpha = 0.9$. (b) $\alpha = 1.01$ is unstable.

5.5.2 A digression: differencing by the finite volume approach

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

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

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

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

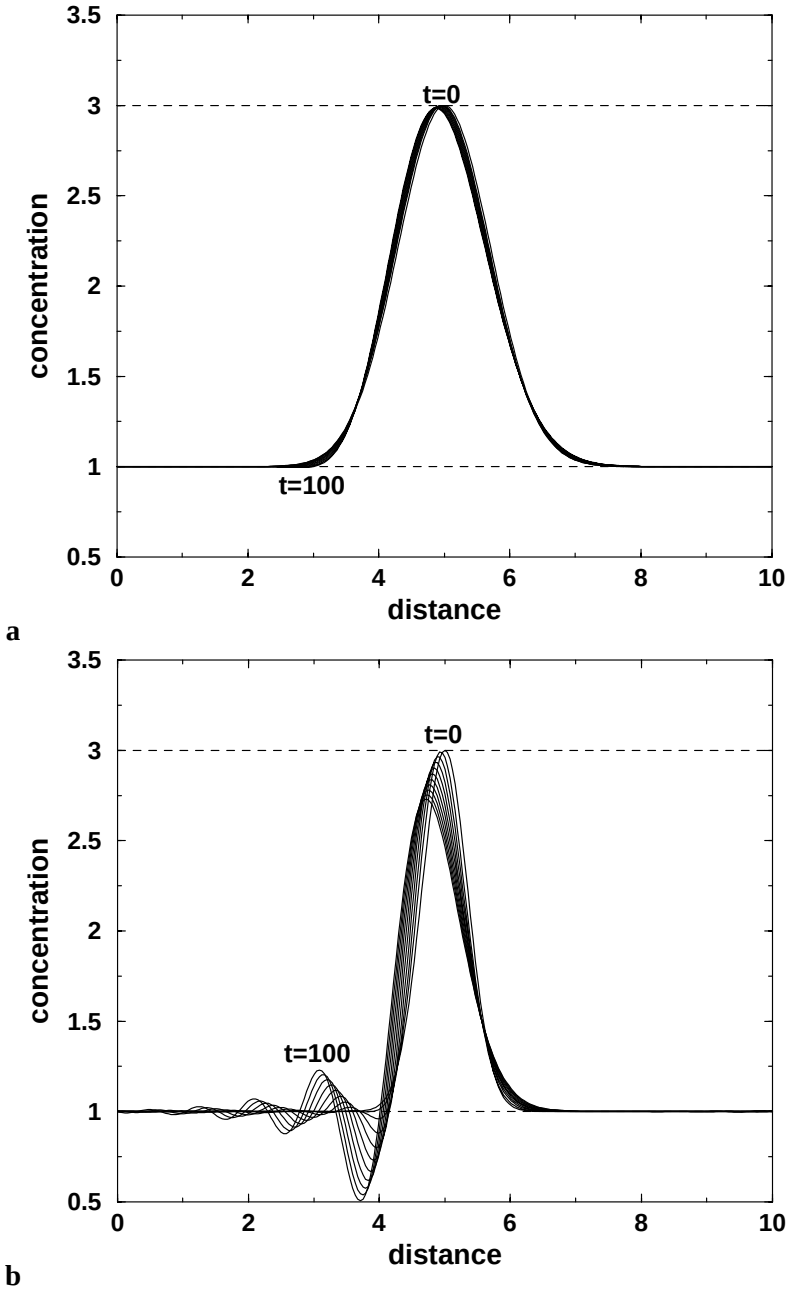


Figure 5.5: (a) Evolution of gaussian initial condition (amplitude 3, width 1., 257 grid points) using a staggered leapfrog scheme with $\alpha = 0.5$. (b) Gaussian is narrower (width=.5) but $\alpha = .5$ For well resolved problems with α close to one however, this is an easy and accurate scheme.

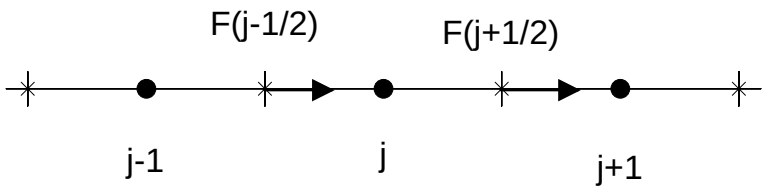


Figure 5.6: A simple staggered grid used to define the control volume approach. Dots denote nodes where average values of the control volume are stored. X's mark control volume boundaries at half grid points.

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

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

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

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

is

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

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

5.5.3 Upwind Differencing (Donor Cell)

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

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

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

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

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

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

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

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

or in terms of the Courant number

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

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

5.5.4 Improved Upwind schemes: mpdata

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

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

where

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

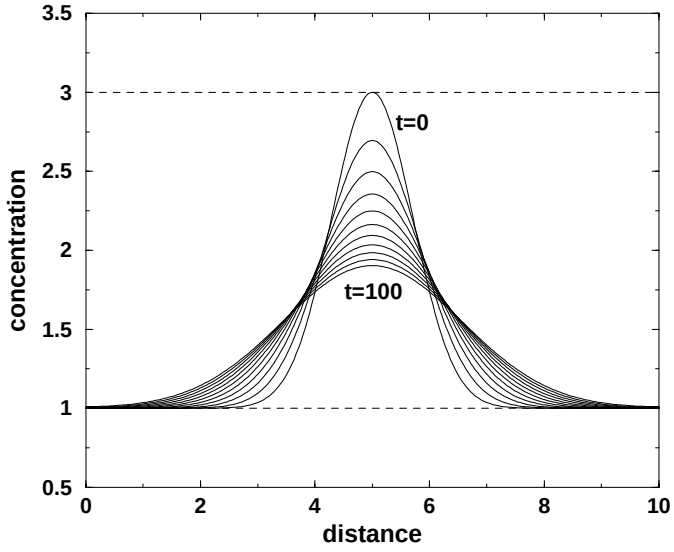


Figure 5.7: Evolution of gaussian initial condition (amplitude 3, width 1., 257 grid points) using an upwind differencing donor-cell algorithm $\alpha = 0.5$. After 10 passes around the grid ($t = 100$) the numerical diffusion has reduced the initial condition to less than half of its amplitude and broadened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

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

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

in the next corrective step.

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

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

where

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

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

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

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

If we define the donor cell algorithm as

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

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

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

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

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

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

5.6 Semi-Lagrangian schemes

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

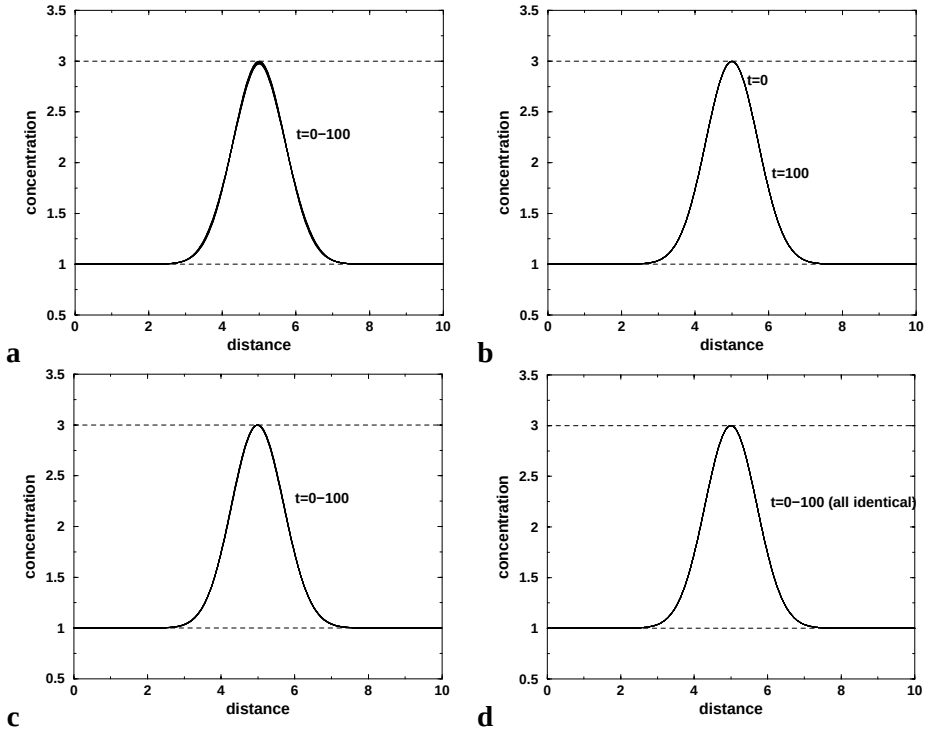


Figure 5.8: Some schemes that actually work. Evolution of gaussian initial condition (amplitude 3, width 1., 257 total grid points) with a variety of updating schemes. All times are for a SunUltra 140e compiled `f77 -fast` *(a)* mpdata with a single upwind correction. $\alpha = 0.5$. (0.49 cpu sec)*(b)*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

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

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

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

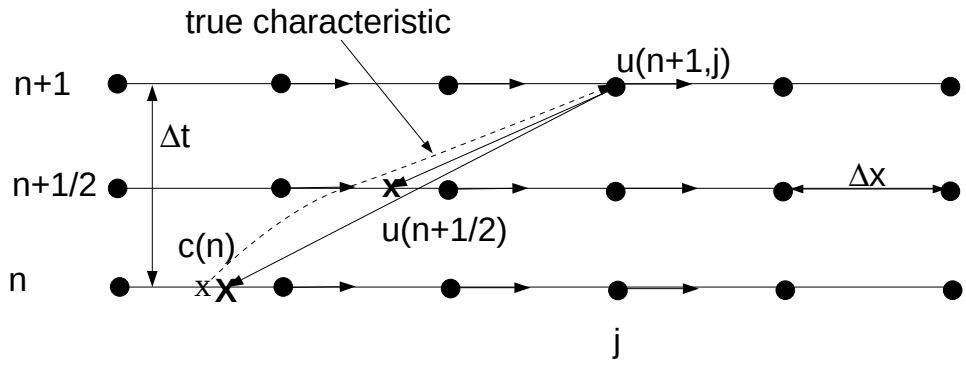


Figure 5.9: Schematic description of the semi-lagrangian advection scheme. x marks the point at time n that lies on the true characteristic that will pass through our grid-point j in time Δt. X is the best approximation for this point found by taking a backwards mid-point step from the future time; To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

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

For each point j

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

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

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

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

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

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

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

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

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

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

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

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

```
      subroutine upsemlag1(arn,aro,wta,ina,ni,is,ie)
      implicit none
      integer ni,i,is,ie
      real arn(ni),aro(ni),wta(4,ni)
      integer ina(4,ni)

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

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

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

      hdt =.5*dtdz

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

      include 'semilagsubs1d.f'
```

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

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

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

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

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

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


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

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

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

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

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

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

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

      hdt=.5*dt
      hdtdx=hdt/dx

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

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

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

```
return
end

include 'semilagsubs1d.f'
```

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

5.6.1 Adding source terms

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

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

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

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

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

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

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

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

or re-arranging

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

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

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

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

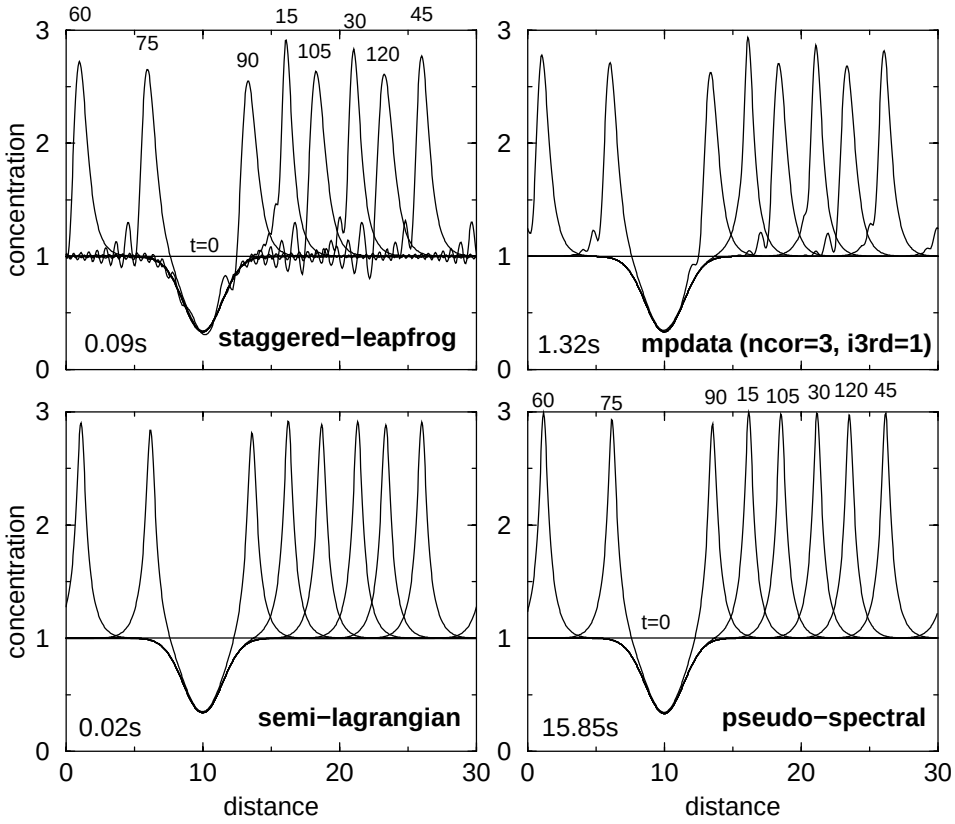


Figure 5.10: Comparison of solutions for advection of a tracer in a non-constant velocity field all runs have 513 (or 512) grid points. Times in figures are for user time (cpu time) on a SunUltra140e compiled with `f77 -fast` (*a*) Staggered-Leapfrog showing significant numerical dispersion with courant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

5.7 Pseudo-spectral schemes

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

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

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

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

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

where Δ is the grid spacing and

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

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

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

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

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

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

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

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

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

In Matlab, such a routine looks like

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

[3] Fastest Fourier Transform in the West

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

In f77, a subroutine might look something like

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

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

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

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

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

5.7.1 Time Stepping

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

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

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

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

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

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

A snippet of code that implements this is

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

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

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

      integer n
      real ap

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

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

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

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

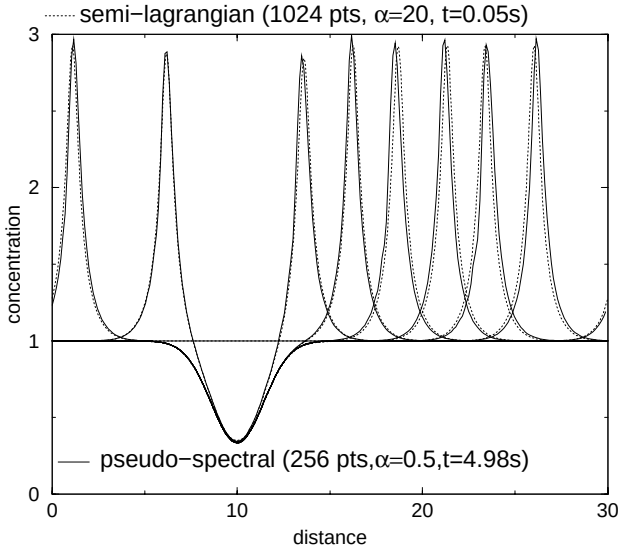


Figure 5.11: Comparison of non-constant velocity advection schemes for Semi-Lagrangian scheme with 1025 points and a Pseudo-Spectral scheme at 256 points. The two schemes are about comparable in accuracy but the semi-lagrangian scheme is nearly 100 times faster.

5.8 Summary

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

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

Bibliography

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

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

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

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

Chapter 6

Diffusion: Diffusive initial value problems and how to solve them

Selected Reading

Numerical Recipes, 2nd edition: Chapter 19

This section will consider the physics and solution of the simplest partial differential equations for diffusive initial value problems in the absence of advection. The archetypal equation of this sort is the "Heat flow equation" which for constant density material can be written

$$\frac{\partial T}{\partial t} = \boldsymbol{\nabla} \cdot \kappa \boldsymbol{\nabla} T \tag{6.0.1}$$

where T is the temperature and $\kappa = k/(\rho c_P)$ is the thermal diffusivity (which has units of length^2/time). I hope most people have some physical feeling for diffusion and heat flow as a smoothing and smearing process, however, we will start by emphasizing some of the more quantitative relationships between diffusive timescales and length scales (as well as developing a simple rule for how not to scorch a turkey).

6.1 Basic physics of diffusion

As a representative problem we will consider one-dimensional diffusion with constant diffusivity.

$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2} \tag{6.1.1}$$

If we consider the evolution of a single sin wave of wavelength λ in an infinite medium and seek solutions of the form

$$T = T_0 e^{\sigma t} \sin\left(\frac{2\pi x}{\lambda}\right) \tag{6.1.2}$$

Then substitution of (6.1.2) into (6.1.1) shows that it is a valid solution if

$$\sigma = -\frac{4\pi^2\kappa}{\lambda^2} \tag{6.1.3}$$

i.e. the full solution is

$$T = T_0 \exp\left[-\frac{4\pi^2\kappa t}{\lambda^2}\right] \sin\left(\frac{2\pi x}{\lambda}\right) \tag{6.1.4}$$

and the time it takes for the amplitude of this sin wave to decay by a factor of $e^{-1} = .37$ is the thermal relaxation time

$$t = \frac{\lambda^2}{4\pi^2\kappa} \tag{6.1.5}$$

The important points to notice are that the amplitude decays *exponentially* with a "decay rate" that only depends on wavelength, not on the initial temperature, and that short wavelength variations decay much more quickly than long wavelength variations. Physically, this makes sense as heat flows down temperature *gradients* and for the same amplitude, short wavelength variations have much sharper gradients than longer wavelength ones. Because we can construct any arbitrary initial condition in an infinite (or periodic) medium out of sines and cosines[1], the overall behaviour of these equations is that the high-frequency information (corners, wiggles and edges) will rapidly smooth out but the larger scale features will remain for a while. The point is that for any scale feature and diffusivity there will be a characteristic time scale that we need to resolve if we want to solve for these things accurately. This time scale will motivate much of the discussion of stability and efficiency of numerical schemes for solving diffusive equations.

As for the turkey, business. The important point of heat conduction is that the time it takes for heat to move a fixed distance is independent of the temperature. Thus if a 20 lb turkey takes 5 hours to cook at 375, you can't cook it for 2.5 hours at 750. All you'll get is a Turkey tootsie pop - hard shell outside, chewy-gooey inside.

6.2 The numerics of diffusion

We'll begin considering how to solve diffusion numerically by deriving some finite difference approximations to the diffusion term. A control volume approach is particularly useful for this purpose. Given a 1-D grid with n control volumes we can consider the divergence of the heat flux out of cell j as

$$\boldsymbol{\nabla} \cdot \mathbf{F} = \frac{1}{\Delta x}\left(F_{j+1/2} - F_{j-1/2}\right) \tag{6.2.1}$$

but for diffusion the heat flux at point $j + 1/2$ is

$$F_{j+1/2} = \kappa_{j+1/2} \frac{\partial T}{\partial x}_{j+1/2} \tag{6.2.2}$$

[1] Well actually we can construct any piecewise continuous function.

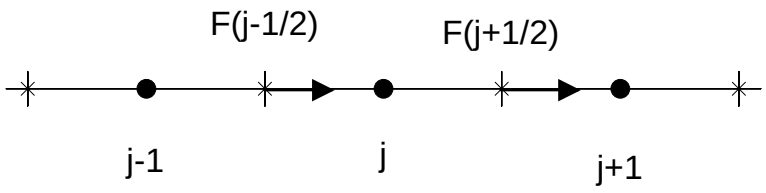


Figure 6.1: our handy-dandy control volume grid again.

If we do a centered difference for the temperature gradient at the half-point we get

$$\frac{\partial T}{\partial x}_{j+1/2} \approx \frac{1}{\Delta x}\left(T_{j+1} - T_j\right) \tag{6.2.3}$$

and therefore we can approximate the entire diffusion term as

$$\frac{\partial}{\partial x}\left(\kappa\frac{\partial T}{\partial x}\right) \approx \frac{1}{\Delta x^2}\left[\kappa_{j-1/2}T_{j-1} - (\kappa_{j-1/2} + \kappa_{j+1/2})T_j + \kappa_{j+1/2}T_{j+1}\right] \tag{6.2.4}$$

For convenience and future notation, it is useful to write (6.2.4) as a *stencil*

$$\frac{\partial}{\partial x}\left(\kappa\frac{\partial T}{\partial x}\right) \approx \frac{1}{\Delta x^2}\Big[\ \kappa_{j-1/2} \quad -(\kappa_{j-1/2} + \kappa_{j+1/2}) \quad \kappa_{j+1/2}\ \Big]T_j \tag{6.2.5}$$

which is an operator that acts on a point T_j and its two nearest neighbors. Unless it is specifically noted, the central point in the stencil is the coefficient of point T_j. In this notation, the approximate diffusion term for a constant diffusivity looks like

$$\frac{\kappa}{\Delta x^2}\Big[\ 1 \quad -2 \quad 1\ \Big]T_j \tag{6.2.6}$$

Using a Taylor's series approach it is straightfoward to show that the truncation error on this discretization is order $\Delta x^2 T_{xxxx}$. Thus while the error only changes as Δx^2 this approximation is actually accurate up to third order as any function that can be fit exactly by a cubic polynomial will have no truncation error (i.e. $T_{xxxx} = 0$). Thus diffusion tends to be quite accurate. Higher order schemes, are readily derived but cause the stencils to be larger.

Adding time Given a discretization of the diffusion term we still need to add the time derivative for the left hand side. While the FTCS scheme was always unstable for advection problems, it turns out that it can be stable for diffusion problems (why? As an interesting point the centered time, centered space staggered leapfrog approach is always *unstable* for diffusion problems...c'est la guerre.) Using a forward time step for the LHS and rearranging, the FTCS approximation to the diffusion equation with constant diffusivity can be written

$$\begin{aligned} T_j^{n+1} &= T_j^n + \beta\Big[\ 1 \quad -2 \quad 1\ \Big]T_j^n \\ &= \Big[\ \beta \quad (1-2\beta) \quad \beta\ \Big]T_j^n \end{aligned} \tag{6.2.7}$$

where

$$\beta = \frac{\kappa \Delta t}{\Delta x^2} \tag{6.2.8}$$

is similar to the courant number and physically corresponds to the number of grid points that heat can diffuse in a time step (or it is the inverse of the number of time steps required for heat to diffuse a grid space). Another way to look at β is to note that it is the number of decay times for the highest frequency components on the grid (compare to Eq. (6.1.5).

Inspection of (6.2.7) shows that as long as $\beta < 1/2$, the FTCS scheme acts as a simple 3 point smoother where every point is replaced by some fraction $(1 - 2\beta)$ of its previous value and some mixture of the values of it's nearest neighbor. For example, if $\beta = 1/4$ the stencil looks like $\begin{bmatrix} 1/4 & 1/2 & 1/4 \end{bmatrix}$ and therefore, after one pass of the algorithm, a unit spike on the grid gets reduced to half its height and smeared out over its two nearest neighbors (i.e $\begin{bmatrix} 0 & 1 & 0 \end{bmatrix} T_j \rightarrow \begin{bmatrix} 1/4 & 1/2 & 1/4 \end{bmatrix}$). If $\beta > 1/2$, however, the scheme is clearly unstable as a single step will make the temperature go negative (which is not physical and will eventually explode), thus the stability criteria for the FTCS step is $\beta < 1/2$. Expanding (6.2.8) shows that this stability criterion requires that

$$\Delta t < \frac{\Delta x^2}{2\kappa} \tag{6.2.9}$$

which is the time constant for the highest frequency components to decay. This is the curse of the FTCS explicit method. You need to track the decay of the fastest decaying wavelength to remain stable even when you are not interested in features that are spanned by 1 or 2 grid points (if you are then your grid is too coarse and those features will be badly underresolved). Unfortunately, as the grid is refined so that the features of interest are spanned by at least 3 to 4 grid points (or more), the time step needs to get smaller as Δx^2. This means that to reduce the grid spacing by a factor of 2 requires 8 times longer to calculate to the same t_{max} (it's 8, not 4 because there are twice as many points and 4 times as many steps). Even in 1-D, this sort of scaling can make it costly to have well resolved grids. Fortunately, FTCS is not the only scheme available and the following section will develop *implicit* schemes which are unconditionally stable and you can (but shouldn't) take as large a step as you want. Before we develop these schemes, however, we need to consider how to include boundary conditions.

6.2.1 Boundary conditions

Any PDE with 2nd derivatives (and above) must have boundary conditions specified at the edges of the domain. Boundary conditions are extremely important and often control the behaviour of the solution. Unfortunately, they are also often the most difficult part of a problem to get to behave (many times the boundary conditions can be a source of instability even if the general scheme is stable). In discrete systems, boundary conditions are also required to make sure there are enough equations for the unknowns, i.e. within the domain, all the points satisfy

the stencil equation (6.2.7), however, at the edges T_1 and T_N we have to specify additional information to make up for the lack of a T_0 or T_{N+1} point. For second order differential equations, there are basically 4 types of boundary conditions we will be concerned with

Dirichlet Boundary conditions The simplest boundary conditions are known as Dirichlet conditions and simply state that the value at the boundary (T_1 or T_N) is a known function of time. In a 1-D problem, if both boundaries are of dirichlet type, then a unique solution exists (I think) and the equations are easy to solve as the only points that are unknown are the interior points $T_j, j = 2, \ldots, N-1$.

Neumann Boundary conditions In addition to specifying the temperature at the boundary, one could also specify the heat flux (or temperature gradient) at the boundary. These conditions are known as Neumann conditions and produce a great deal more freedom in the behaviour of the governing equations (although the equations may become ill-posed). If a boundary has Neumann BC's then the temperature on that boundary is variable with time and requires an additional equation. To derive the boundary conditions for a specified flux boundary it is convenient to use the control volume formalism. If we specify that the heat flux at point 1 is F_1, then the diffusion term for the half-sized control volume between x_1 and $x_{1+1/2}$ is

$$\frac{\partial F}{\partial x} \approx \frac{1}{\Delta x/2}\left(\kappa_{j+1/2}\frac{T_{j+1}-T_j}{\Delta x} - F_1\right) \tag{6.2.10}$$

or in the case of an insulating boundary $F_1 = 0$

$$\approx \beta\Big[\begin{array}{ccc} 0 & -2 & 2 \end{array}\Big]T_1 \tag{6.2.11}$$

Note that (6.2.11) would have the identical effect to having T_1 be an internal point but $T_0 = T_2$ i.e. a zero flux boundary must also be a reflection boundary. The extension to the right hand boundary at T_N is straightforward.

Mixed Boundary conditions It is also possible to have mixed Dirichlet/Neumann boundary conditions of the form

$$T_1 + a\frac{\partial T}{\partial x}_1 = f(t) \tag{6.2.12}$$

This is just a more general version of the Neumann condition and will produce an additional auxilary equation (stencil) for the boundary points.

Periodic or wraparound Boundary conditions These boundary conditions are extremely convenient numerically because they approximate an infinite periodic system by simply specifying that $T_N = T_1$ and therefore $T_{N+1} = T_2$ and $T_0 = T_{N-1}$. With just a simple patch at the end of each iteration, every point behaves as an interior point. These boundary conditions can be very useful ones when developing a code as only the stability of the internal solution is important.

6.3 Implicit Schemes and stability

The stability of the FTCS scheme is contingent on tracking the fastest decaying wavelengths (which is the wavelength of the grid noise) even when those wavelengths are not of interest. The problem arises because the rate of diffusion calculated at the present time is always greater than at later times and thus too large a time step will overshoot the solution. The fix for this is to use an *implicit method* which uses information about diffusion rates at *future times*. To see how it works, it is useful to develop implicit approaches for the analogous problem of radioactive decay.

6.3.1 An analogy with radioactive decay

Radioactive decay is actually an almost identical problem to diffusion if we note that individual frequencies decay exponentially with a "decay rate" given by the thermal time-constant. Now the simplest ODE for the decay of a radioactive species is

$$\frac{dc}{dt} = -\lambda c \tag{6.3.1}$$

which we could solve by taking an explicit Euler step

$$c^{n+1} = c^n - \lambda \Delta t c^n = (1 - \lambda \Delta t) c^n \tag{6.3.2}$$

which says to use the decay rate at time n ($-\lambda c^n$) and use this rate to extrapolate to a future time by taking a step of Δt. Unfortunately, the rate at time n is always an overestimate and we have to take very small steps in order to not undershoot the solution (remember how inaccurate the euler method is). Moreover, if we take too big a step ($\lambda \Delta t > 1$) we will drive the concentration to negative values which are unphysical and unstable.

The fix to this problem is to take an implicit step that uses the decay rate at a future time to predict the rate of change, i.e. the implicit version of the euler step (also called a backwards euler step) is

$$c^{n+1} = c^n - \lambda \Delta t c^{n+1} \tag{6.3.3}$$

While we don't know what the concentration c^{n+1} is *a priori*, we can rearrange (6.3.3) to solve for it by

$$c^{n+1} = \frac{c^n}{1 + \lambda \Delta t} \tag{6.3.4}$$

This scheme is stable for any sized time step Δt as in the limit of an enormous time step $c^{n+1} \to 0$ which is the steady-state solution. Just because this scheme is stable, however, does not mean that it is accurate! If you want to track the decay of an element accurately with time you still need to respect the intrinsic time scale of decay and take a time step that can resolve the changes (or use a higher order scheme that incorporates more information about the decay rate). Nevertheless, the implicit schemes allow you to choose a time scale of interest rather than being forced to follow the stability criterion. The same is true for implicit methods for solving diffusion equations.

6.3.2 Fully implicit schemes

The generalization of an implicit scheme to diffusion is straightforward. All we do is to rewrite Eq. (6.2.7) and replace the diffusion rate at timestep n with that at timestep $n+1$. For a constant diffusivity we can write

$$T_j^{n+1} = T_j^n + \beta \Big[\begin{array}{ccc} 1 & -2 & 1 \end{array} \Big] T_j^{n+1} \tag{6.3.5}$$

with the obvious extension to non-constant κ. This is an implicit method because we don't know in advance what the RHS will evaluate to, however, we can again rearrange (6.3.5) to solve for T^{n+1} as

$$\Big[\begin{array}{ccc} -\beta & (1+2\beta) & -\beta \end{array} \Big] T_j^{n+1} = T_j^n \tag{6.3.6}$$

Inspection of (6.3.6) shows that it is actually a system of linear equations of the form

$$\mathsf{A}\mathbf{x} = \mathbf{b} \tag{6.3.7}$$

where A is a *tridiagonal matrix* that is primarily zero except for the diagonal (which has value $(1+2\beta)$) and one super and sub diagonal of value $-\beta$. For (6.3.7) the vector $\mathbf{x} = \mathbf{T}^{n+1}$ corresponds to the array of temperature values at timestep $n+1$ and the vector $\mathbf{b}$ is the known temperatures at time n. Thus if we can invert A we can solve for $\mathbf{T}^{n+1}$ as

$$\mathbf{T}^{n+1} = \mathsf{A}^{-1}\mathbf{T}^n \tag{6.3.8}$$

Fortunately, A is symmetric and positive definite and is readily inverted. Better yet, tridiagonal matrices can be inverted in order N operations where N is the total number of grid points. To understand what a blessing this is, normal dense matrix inversion requires order N^3 operations and rapidly becomes unfeasible for even moderately large grids. While Numerical recipes (and netlib) provide efficient implementations of tridiagonal solvers, some care should be taken when using Matlab to solve tridiagonal systems of equations. In general there are (at least) three ways to solve the linear system in Eq. (6.3.7) using Matlab. Unfortunately, the standard approach turns out to be approximately an N^2 method, rather than an order N method. Figure 6.2 compares solution times using three Matlab schemes. The first `x=inv(A)*b` is horrendously inefficient and should never be used. The second `x=A\b` implements Gaussian elimination (without calculating the inverse of A). While this is about an order of magnitude faster, it is still an N^2 solver. The correct way to use Matlab (without writing a special case tridiagonal solver) is to use LU decomposition which is order N (see [1]). The correct syntax is `[L,U]=lu(A); x=U\(L\b);`. This becomes a feasible scheme.

Given an efficient solver, , this scheme is unconditionally stable so there is no critical time-step at which the scheme explodes. If you want the scheme to be accurate, however, requires taking time steps

$$\Delta t < \frac{\lambda^2}{4\pi^2\kappa} \tag{6.3.9}$$

where d is the wavelength of interest which for a well resolved feature should be at least 3–4 Δx. This scheme, however is still only first order in Δt and therefore

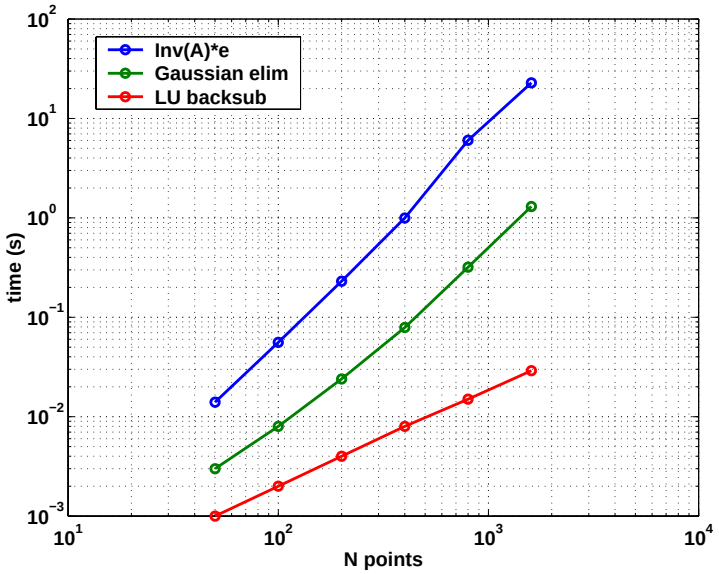


Figure 6.2: Comparison of solution times for three different solutions of $\mathsf{A}\mathbf{x} = \mathbf{b}$ where A is tridiagonal. The blue curve calculates full inverse of A, the green curve uses Gaussian elimination, and the red curve uses LU decomposition and back substitution. The first two schemes scale as N^2 while the LU scheme is order N (and much faster).

halving Δt will only improve the accuracy by a factor of two. A better second order scheme is the *Crank-Nicholson scheme.*

6.3.3 Crank-Nicholson schemes

The Crank-Nicholson scheme is a second order scheme because it uses the diffusion at a time step that is centered in time between t^n and t^{n+1} (remember that centered differences are second order while forward and backward differences are first order). Symbolically we can write the CN scheme as

$$T_j^{n+1} = T_j^n + \beta \begin{bmatrix} 1 & -2 & 1 \end{bmatrix} T_j^{n+1/2} \tag{6.3.10}$$

Unfortunately, we don't have a solution for the half time step, however, we can estimate it as the average of the values at time T^n and T^{n+1} i.e.

$$T_j^{n+1/2} = \frac{1}{2}\left(T_j^n + T_j^{n+1}\right) \tag{6.3.11}$$

Substituting into (6.3.10) and rearranging gives us a new matrix equation

$$\begin{bmatrix} -\frac{\beta}{2} & (1+\beta) & -\frac{\beta}{2} \end{bmatrix} T_j^{n+1} = \begin{bmatrix} \frac{\beta}{2} & (1-\beta) & \frac{\beta}{2} \end{bmatrix} T_j^n \tag{6.3.12}$$

or multiplying by 2 as

$$\begin{bmatrix} -\beta & 2(1+\beta) & -\beta \end{bmatrix} T_j^{n+1} = \begin{bmatrix} \beta & 2(1-\beta) & \beta \end{bmatrix} T_j^n \tag{6.3.13}$$

which can be solved by tridiagonal inversion. Note: although the RHS of (6.3.12) is more complicated it is still known. This scheme is also unconditionally stable (with all the previous caveats in place). For simple diffusional problems, this is probably your best bet.

6.3.4 Boundary conditions for implicit schemes

Implicit schemes also require boundary conditions at possibly several time steps. Fortunately, they are easy to implement by modifying the form of the matrix equation. In the simplest form, Dirichlet conditions modify the RHS side vector (i.e. **b**) while Neumann and periodic boundary conditions modify the matrix A. For dirichlet conditions, consider the fully implicit stencil for the first unknown T_2. Expanding (6.3.6) we for $j = 2$ gives

$$-\beta T_1^{n+1} + (1 + 2\beta)T_2^{n+1} - \beta T_3^{n+1} = T_2^n \tag{6.3.14}$$

Since we already know the value of T_1^{n+1}, then we can move it over to the RHS and rewrite the equation as

$$(1 + 2\beta)T_2^{n+1} - \beta T_3^{n+1} = T_2^n + \beta T_1^{n+1} \tag{6.3.15}$$

Thus in matrix form we simply add βT_1^{n+1} to b_1 and βT_N^{n+1} to b_N. Crank nicholson schemes produce similar modifications. An alternative approach for dirichlet conditions that is often easier to implement is to change both the stencil and the RHS for the *Dirichlet points*, i.e. if T_1 is dirichlet and equals T_b then an appropriate stencil for this point is simply

$$\begin{bmatrix} 0 & 1 & 0 \end{bmatrix} T_1 = T_b \tag{6.3.16}$$

and then Eq. (6.3.14) can just be solved as an interior point.

For Neumann conditions, the first unknown is the boundary temperature T_1 and we need to use the auxilary equation (6.2.10). Using this equation for the RHS of the diffusion equation in a fully implicit scheme (with constant κ) gives

$$T_1^{n+1} = T_1^n + \beta\left(-2T_1^{n+1} + 2T_2^{n+1} - 2\Delta x F_1\right) \tag{6.3.17}$$

or in stencil notation as

$$\begin{bmatrix} 0 & 1 + 2\beta & -2\beta \end{bmatrix} T_1^{n+1} = T_1^n - 2\beta\Delta x F_1 \tag{6.3.18}$$

where F_1 is the heat flux at the $j = 1$ boundary. Thus a full Neumann boundary modifies the coefficient of T_2 in the matrix A and changes the RHS vector **b** by an amount proportional to the flux. If the boundary is a no-flux boundary, only the matrix is modified. At the $j = N$ boundary a similar thing occurs, however it is the coefficient of the first interior point T_{N-1}^{n+1} that is modified. Mixed boundary conditions and Crank Nicholson schemes are readily generalized (although in CN schemes, boundary conditions are needed at both times n and $n + 1$).

Wraparound boundaries, are a bit more difficult, because they add off-diagonal terms to the matrix A and make it ever-so-slightly non-tridiagonal (by 2 points). This may seem to be trivial but it is enough to destroy the utility of the tridiagonal inversion. Fortunately, this feature occurs so frequently that people more clever than myself have a fix for it. Methods for solving these *cyclic* tridiagonal problems using a Sherman-Morrison correction are discussed in detail in Numerical recipes (2nd edition) where a subroutine `cyclic` is given. Due to the nature of the correction, cyclic tridiagonal systems take approximately twice as much work to invert as simple tridiagonal systems. Nevertheless, for the gain in stability in time stepping, a cyclic crank-nicholson scheme is still more efficient and accurate than an explicit scheme.

6.4 Non-constant diffusivity

Much of the discussion here has considered constant diffusivities although the generalizations to non-constant diffusivities are straightforward using the control volume approach. This scheme is always flux conservative, however sometimes, if $\kappa(x,t)$ is known analytically[2], then it can be more accurate to expand the diffusion term by the chain-rule to get

$$\frac{\partial \kappa}{\partial x}\frac{\partial T}{\partial x} + \kappa \frac{\partial^2 T}{\partial x^2} \tag{6.4.1}$$

which introduces an effective advection term. Whatever you do, never make the mistake of simply writing the diffusion term as

$$\kappa(x)\frac{\partial^2 T}{\partial x^2} \tag{6.4.2}$$

as this is simply incorrect. When in doubt, start deriving the governing equations from the integral form of the conservation equations and you'll never go wrong. Non-constant diffusivities can cause apparent source terms and strange behaviour (as well as some artifacts). Some care should be taken to understand the effects. Another difficulty may arise if the diffusivities are temperature dependent as this will lead to non-linear equations. The non-linearites are not difficult to deal with using explicit schemes but cause difficulties for implicit schemes. Several options are to linearize the equations and iterate, or a more powerful option is to use a non-linear relaxation scheme such as FAS multi-grid (which we will deal with in future lectures).

6.5 Summary

For simple 1-D diffusion use a Crank-Nicholson scheme.

[2] Actually, if the diffusivity is only a function of space $\kappa(x)$ and it is sufficiently simple analytically, then there may be an analytic solution in terms of the eigenfunctions of the separated equations. When in doubt check the bible [2].

Bibliography

[1] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling. Numerical Recipes, Cambridge University Press, Cambridge, U.K., 2nd edn., 1992.

[2] M. Abramowitz and I. Stegun. Handbook of Mathematical Functions, vol. 55 of Applied Maths series, National Bureau of Standards, Washington, 1964.

Chapter 7

Combos: Advection-Diffusion and operator splitting

Up to this point we have treated advection and diffusion as two separate processes and considered appropriate numerical techniques for solving each process. However, in many problems of interest both transport and diffusion occur simultaneously and we would like to be able to use the previously discussed methods to solve these more general problems. Unfortunately, as we have seen, a good technique for advection may not work for diffusion and vice versa (the simplest example is that diffusion equations are unstable if differenced using a staggered-leapfrog scheme while advection is unstable if solved with a FTCS scheme). This section will consider some additional techniques for solving coupled advection-diffusion problems and compare their accuracy.

In this section we will consider solutions to the 1-D scaled equations for the advection and diffusion of heat

$$\frac{\partial T'}{\partial t'} + \mathrm{Pe}\frac{\partial T'}{\partial z'} = \frac{\partial^2 T'}{\partial z'^2} \tag{7.0.1}$$

where T' is the dimensionless temperature and $\mathrm{Pe} = w_0 d/\kappa$ is the Peclet number. Because we will deal with strongly diffusive processes in this section, I have scaled the equations by the diffusion time $t = (d^2/\kappa)t'$. For strongly advective problems where $\mathrm{Pe} >> 10$, it makes more sense to scale the equations by the advection time $t = (d/w_0)t''$ (i.e. $t'' = \mathrm{Pe}t'$), in which case the equation becomes

$$\frac{\partial T'}{\partial t''} + \frac{\partial T'}{\partial z'} = \frac{1}{\mathrm{Pe}}\frac{\partial^2 T'}{\partial z'^2} \tag{7.0.2}$$

Either approach is valid and must give the same dimensional solutions.

For the initial condition of a gaussian distribution of temperature

$$T'(z', 0) = A \exp\left[-\frac{(z - z_0)^2}{\sigma^2}\right] \tag{7.0.3}$$

in an infinite medium, equation (7.0.1) has the convenient analytical solution

$$T'(z', t') = \frac{A}{\sqrt{1 + 4t'/\sigma^2}} \exp\left[-\frac{[(z - \mathrm{Pe}t') - z_0]^2}{\sigma^2 + 4t'}\right] \tag{7.0.4}$$

where A is the initial amplitude, z_0 is the initial position of the peak and σ is the original "half-width" (one standard-deviation). Thus in a perfect solution, the original bump will move off at a constant speed Pe and widen and decrease in amplitude. The question is how well do the various numerical schemes reproduce this result.

7.1 A smorgasbord of techniques

This section will compare several numerical schemes to the analytic solution (7.0.4). All of the problems will use $A = 2$ (over a background $T = 1$), $\mathrm{Pe} = 5$, $z_0 = 10$, $\sigma = 1$. The problem will be solved over a region $0 < z' < 50$ with 501 grid points $\Delta z = .1$, for a maximum time of $t' = 4$.

7.1.1 Explicit FTCS

The simplest scheme of all is a forward-time centered-space (FTCS) discretization of the full equation, i.e.

$$\frac{T_j^{n+1} - T_j^n}{\Delta t} = -Pe\frac{T_{j+1}^n - T_{j-1}^n}{2\Delta z} + \frac{T_{j+1}^n - 2T_j^n + T_{j-1}^n}{\Delta z^2} \tag{7.1.1}$$

or rearranging for T_j^{n+1} we get the simple updating scheme in stencil form

$$T^{n+1} = \left[\begin{array}{ccc} (\beta + \alpha) & (1 - 2\beta) & (\beta - \alpha) \end{array} \right] T^n \tag{7.1.2}$$

where $\alpha = \mathrm{Pe}\Delta t/(2\Delta z)$ is the local courant number and $\beta = \Delta t/\Delta z^2$ is the grid-space diffusion time. This scheme can be coded using something akin to

```
cm=beta+alpha
cc=1-2.*beta
cp=beta-alpha
do i=2,npnts-1
   ar1(i)= cm*ar2(i-1)+cc*ar2(i)+cp*ar2(i+1)
enddo
```

While FTCS is completely unstable for pure advection (because of negative numerical diffusion), it is stabilized by the addition of some real diffusion as long as the time step is smaller than the diffusive stability limit, i.e $\beta < 1/2$. Figure 7.1 compares the analytic solution and the calculated solution and their fractional error $\epsilon = T_{calc}/T_{true} - 1$ for the FTCS scheme. Because of the inherent negative diffusion, the calculated solutions do not decay as fast as they should and the overall error near the peak is positive and of order 5×10^{-3}. Which is not bad but it is noticeable (and we can do much better). In addition, the step size for a FTCS scheme must be very small and this solution requires 1600 time steps.

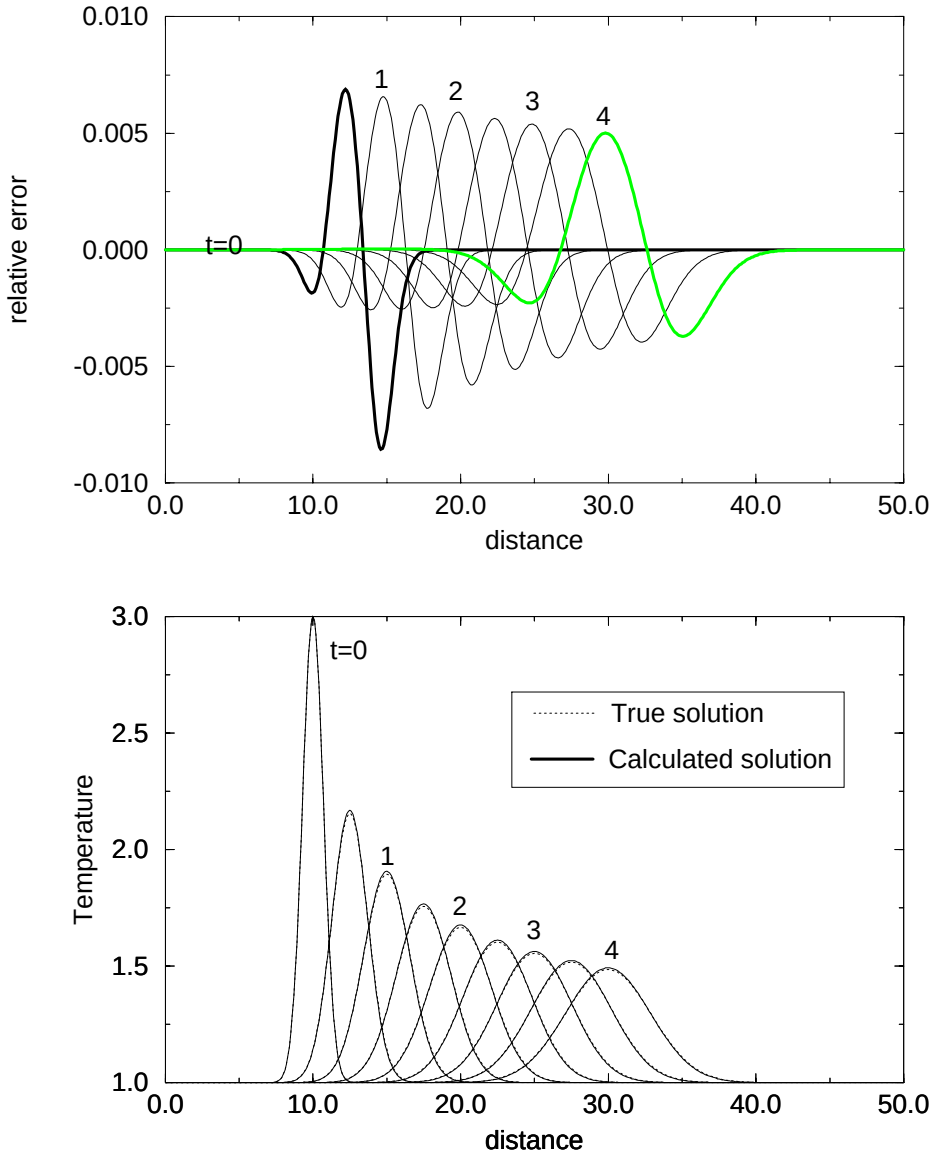


Figure 7.1: Solution and error of the model problem using an explicit FTCS differencing scheme.

7.1.2 Generalized Crank-Nicholson

The second scheme we will try is a more generalized version of the Crank-Nicholson scheme for diffusion which is implicit, stable for relatively large time steps and about a factor of 5 more accurate than the FTCS scheme. If we consider the FTCS scheme as a general operator equation of the form

$$T_j^{n+1} = T_j^n + \mathcal{L}T_j^n \tag{7.1.3}$$

where $\mathcal{L}$ is an operator that conveniently soaks up all the details, then a generalized crank nicholson scheme looks like

$$T_j^{n+1} = T_j^n + \frac{1}{2}\left[\mathcal{L}T_j^{n+1} + \mathcal{L}T_j^n\right] \tag{7.1.4}$$

as it is just the average of the rates at the present and future time. Using centered differences for the advection and diffusion parts of the operator, the implicit crank-nicholson formulation for advection-diffusion can be written in stencil form as

$$\left[\begin{array}{ccc} -(\beta/2+\alpha) & (1+\beta) & (\alpha-\beta/2) \end{array}\right]T^{n+1} = \left[\begin{array}{ccc} (\beta/2+\alpha) & (1-\beta) & (\beta/2-\alpha) \end{array}\right]T^n \tag{7.1.5}$$

Here $\alpha = \mathrm{Pe}\Delta t/(4\Delta z)$ and β is the same as before. This system of equations is tridiagonal and can be inverted using a subroutine such as Numerical recipes `tridag`. The relevant fortran snippets would look like

```
c---------set the matrix coefficients
      cm = -(alpha + .5*beta)           !sub-diagonal   T_(i-
1)
      cc = (1.+beta)                    ! diagonal T_i
      cp = alpha - .5*beta              ! super diagonal T_(i+1)
      do i=1,npnts
        a(i)= cm
        b(i)= cc
        c(i)= cp
      enddo

c---------set the right hand side

      cm =  (alpha + .5*beta)           !sub-diagonal T_(i-1)
      cc = (1.-beta)                    ! diagonal T_i
      cp = .5*beta  - alpha             ! super diagonal T_(i+1)
      do i=2,npnts-1                    ! interior points
        d(i)=cm*tar(i-1)+cc*tar(i) + cp*tar(i+1)
      enddo

c-----magically patch up the boundary conditions

      YOUR CODE HERE

c--------invert the matrix
      call tridag(a,b,c,d,ar,npnts)
```

Figure 7.2 shows the results of this scheme which requires only 400 steps and does not exhibit negative numerical diffusion. Because of the asymmetry in the operator matrix, this scheme will probably become unstable for large values of Pe (I need to work this out or it can be extra credit for those inclined).

7.1.3 Operator Splitting, MPDATA/Semi-Lagrangian scehmes+ Crank Nicholson

For simple differencing schemes, it is straightforward to extend standard techniques to include both advection and diffusion. However, we would also like to

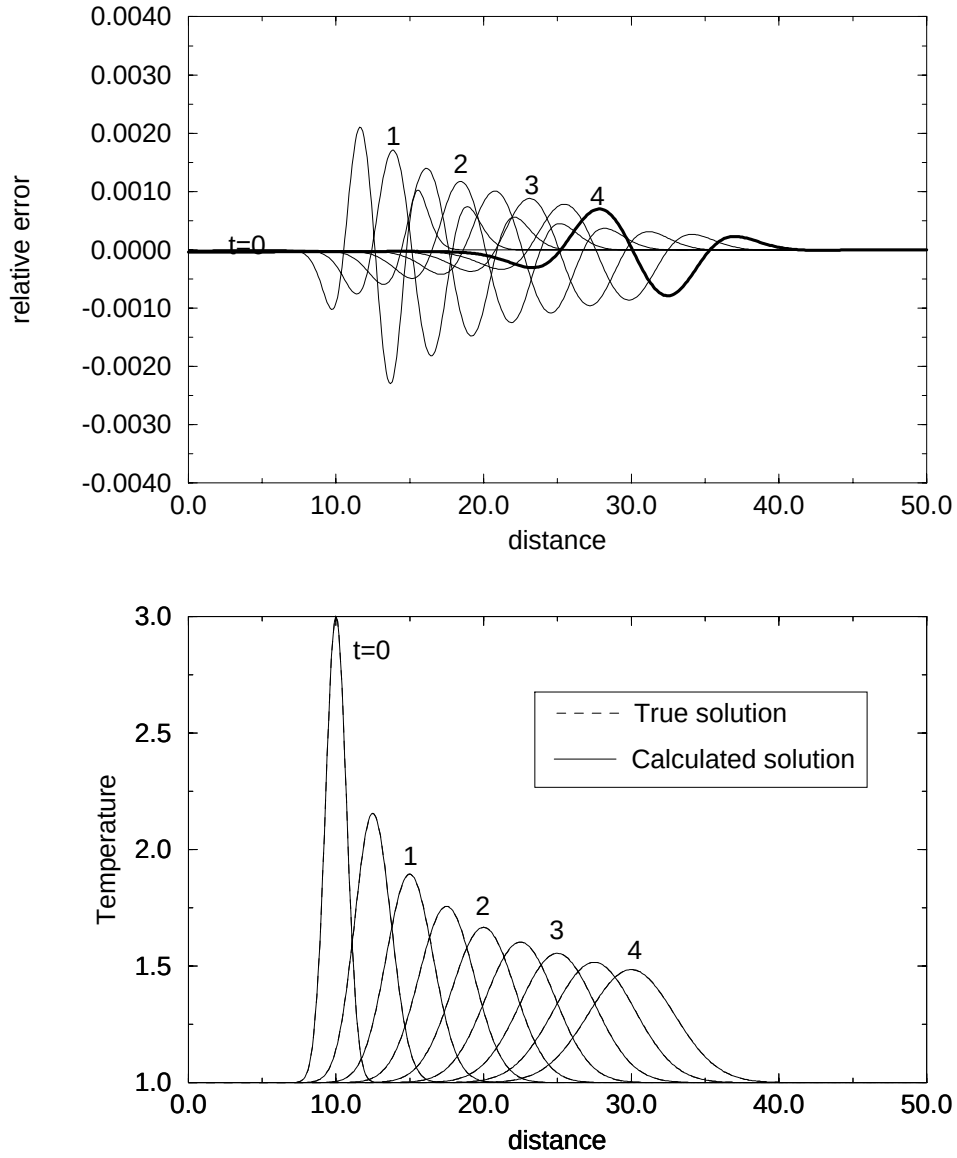


Figure 7.2: Solution and error of the model problem using an implicit crank-nicholson scheme.

be able to use something fancy like MPDATA or a semi-lagrangian scheme to handle the advection without having to rewrite the scheme to include diffusion[1]. In addition we would like to use a crank-nicholson scheme for just the diffusion part because it is highly accurate and unconditionally stable. The answer to our dilemma is a useful kludge known as *operator splitting* which simply suggests that we use each method for each process separately i.e. first advect and then diffuse (or vice versa).

[1]Although look at Section 7.2 as it is not actually much harder to do this with semi-lagrangian schemes

More formally, we can think of our initial equation as being of the form

$$\frac{\partial T}{\partial t} = \mathcal{L}_{adv}T + \mathcal{L}_{diff}T \tag{7.1.6}$$

where we have two (or more operators) that affect T in an additive way. Following Numerical recipes, we will now suppose that for each operator we have a good updating scheme that would take T^n to T^{n+1} if it were the only operative process. i.e. we could write the updatings as

$$\begin{aligned} T^{n+1} &= \mathcal{U}_{adv}(T^n, \Delta t) \\ T^{n+1} &= \mathcal{U}_{diff}(T^n, \Delta t) \end{aligned} \tag{7.1.7}$$

(we could also continue to add more processes and more updating schemes). Now all that operator splitting says is to proceed from time n to time $n+1$ by the following series of updates

$$\begin{aligned} T^{n+(1/2)} &= \mathcal{U}_{adv}(T^n, \Delta t) \\ T^{n+1} &= \mathcal{U}_{diff}(T^{n+(1/2)}, \Delta t) \end{aligned} \tag{7.1.8}$$

that is, advect the solution for a time-step Δt without any diffusion, then take this new solution and diffuse it for the same amount of time without advection (and repeat ad nauseum). Note that we have not advanced the time to $n+2$ however both processes have occurred for a time of Δt (as they should).

While it may seem that this approach is not hugely physical, it turns out in practice, that if the time-steps are relatively small and the separate updating schemes are sufficiently superior to any hybrid scheme, then any inaccuracies introduced in the sequential application of the operators are still much smaller than the inaccuracies of a scheme that is a compromise between methods. To illustrate this, Figures 7.3–7.5 show the results of this operator splitting approach using MPDATA for the advection scheme and Crank-Nicholson for the diffusion scheme. The relevant fortran would look something like

```
c-------set  tridiagonal matrix coefficients for CN step (alpha=0)
      call setcofcn1(a,b,c,npnts,0.,beta)

c------loop in time, doing an mpdata step then a CN step
      do n=1,nsteps
        t=t+dt*(n-1)
        call mpdata1(wp,T,npnts,ncor,i3rd,wk(1,1),wk(1,2)) !advect
        call setrhscn1(T,d,npnts,0.,ad)  !set the rhs with the cur-
rent temp.
        call tridag(a,b,c,d,T,npnts)     ! diffuse ala CN step
      enddo
```

The only differences in Figures 7.3–7.5 are the number of corrections `ncor` used in the mpdata step. For `ncor=1`, (7.3) the advection is a standard donor-cell upwind differencing and is terribly over-diffusive. One mpdata correction (`ncor=2`, Fig. 7.4) however reduces the maximum error to less than 3×10^{-4} which is an *order of magnitude better* than the combined Crank-Nicholson step.

Taking another correction (`ncor=3`, Fig. 7.5) reduces the error by another factor of 2. All of these improvements are at the expense of running time (and they can be severe) however, if high accuracy or very long runs are required it's nice to know that it works.

Figure 7.6 shows an operator splitting solution where the advective step is taken using a semi-lagrangian scheme. As expected, this scheme works at least as well as the most expensive mpdata scheme but is much faster. Trial and error showed that the minimum error for the problem was attained for a time-step that was 2.5 times the size of the critical stability step for the mpdata schemes (For comparison, Fig. 7.5 has a time step that is only 0.5 of the critical step-size). Thus for shear number of time steps, this scheme is a factor of five faster. The actual speed difference is larger than this because of the relative complexities of the mpdata scheme.

7.1.4 A caveat on operator splitting (and other numerical schemes)

There may be another deeper reason why operator splitting works so well for this model problem and that is because the analytic solution is inherently separable, i.e. advection and diffusion are completely uncoupled in this problem. To see this, note that this problem could also be solved by moving into a frame that is moving with the peak. In this frame the problem is only a diffusion problem that could be solved very well using a crank-nicholson scheme. Thus in the actual analytic solution, it is possible to calculate an arbitrary amount of diffusion and then advect the peak a long way and still get the right solution. Whether this method works with problems where the transport and diffusion cannot be separated is less clear.

The general caveat for comparison to analytic solutions is that quite often, the reason that an analytic solution exists is because the problems are linear, or separable and in some sense, special case solutions. Whether a numerical scheme that works for a linear solution (or constant coefficient equation) will carry over to more complex problems is never guaranteed. When in doubt, do a convergence test by changing the grid space and time step and see how much the solutions vary. Finally, do enough analysis to convince yourself (and others) that the solutions are doing what the equations say they should. Remember *You're not finished, until you can prove that your solution is reasonable.*

7.2 Another approach: Semi-Lagrangian Crank-Nicholson schemes

To be honest, there is always something vaguely fishy about operator-splitting because it is often not symmetric with respect to the processes being considered. I.e. should you advect then diffuse or diffuse then advect? It's not always obvious (although in this case it makes no difference). In a perfect world you would like a scheme that handles all the possible processes in one consistent step yet doesn't add any new artifacts. As it turns out, for the advection-diffusion problem, there is a combined method that has all these properties and can solve for any advection-diffusion problem from $\mathrm{Pe} = 0 \rightarrow \infty$ with no artifacts and is a simple modification

to the schemes we already know. The place to begin is to note that we could write Eq. (7.0.2) in terms of the material derivative as

$$\frac{DT}{Dt} = \frac{1}{\mathrm{Pe}}\frac{\partial^2 T}{\partial x^2} \tag{7.2.1}$$

Now normally we couldn't solve (7.2.1) using the method of characteristics because the right-hand side depends on temperature gradients and therefore requires information about temperature on other characteristics. However, we do know how to solve something like (7.2.1) when things aren't moving. In Chapter 6 we just used a Crank-Nicholson scheme. The only difference now between Eq. (7.2.1) and Eq. (6.1.1) is that the time derivative is in a frame following a piece of material. But this is the frame in which we solve semi-Lagrangian schemes so it should be possible to combine the two approaches. If we just define $T^n_{j'}$ as the temperature at the point j' which will move to point j in time Δt then we can difference (7.2.1) with a Crank-Nicholson scheme like

$$\frac{T^{n+1}_j - T^n_{j'}}{\Delta t} = \frac{1}{2\mathrm{Pe}\Delta x^2}\left(\begin{bmatrix} 1 & -2 & 1 \end{bmatrix} T^n_{j'} + \begin{bmatrix} 1 & -2 & 1 \end{bmatrix} T^{n+1}_j\right) \tag{7.2.2}$$

which is identical to the standard Crank-Nicholson scheme except that now the point where the old temperature and old diffusion are calculated is at point j' not at point j. Put another way, we are just assuming that diffusion is a source term for the semi-Lagrangian solver and we are going to assume that the change in temperature along the flow path is just the integral of the amount of diffusion experienced during travel. Either way, we can rearrange (7.2.2) into a standard Crank-Nicholson Scheme

$$\begin{bmatrix} -1 & (\gamma+2) & -1 \end{bmatrix} T^{n+1}_j = \begin{bmatrix} 1 & (\gamma-2) & 1 \end{bmatrix} T^n_{j'} \tag{7.2.3}$$

where

$$\gamma = \frac{2\mathrm{Pe}\Delta x^2}{\Delta t} = \frac{2\mathrm{Pe}}{\beta} \tag{7.2.4}$$

The only difference is that the RHS is now evaluated at the interpolation point j' not at the grid point j. The important feature of this scheme is that the advection is hidden and doesn't change the symmetry of the operator. Moreover it is rather convenient that both simple diffusion and interpolation are linear operators and therefore, you can exchange the order of operations and calculate the RHS on the grid and then interpolate the value of the RHS. I.e. first calculate

$$d_j = \begin{bmatrix} 1 & (\gamma-2) & 1 \end{bmatrix} T^n_j \tag{7.2.5}$$

then the right hand side is just $d_{j'}$. Inflow boundary conditions can be tricky though with this technique.

Inspection of Eq. (7.2.4) shows that if $\mathrm{Pe} \to \infty$ then problem becomes completely dominated by the diagonal terms and $\gamma T^{n+1}_j = \gamma T^n_{j'}$ or $T^{n+1}_j = T^n_{j'}$ which is just the semi-Lagrangian scheme. For small Pe the scheme looks like a Crank-Nicholson scheme (although it probably should be rescaled). In the case that locally $w = 0$ then it just behaves like a CN scheme. Figure 7.7 shows the behaviour

of this scheme and it is equally accurate to the operator-splitting approach. This scheme however, is slightly more prone to round-off error and I find it best to solve this one in double precision.

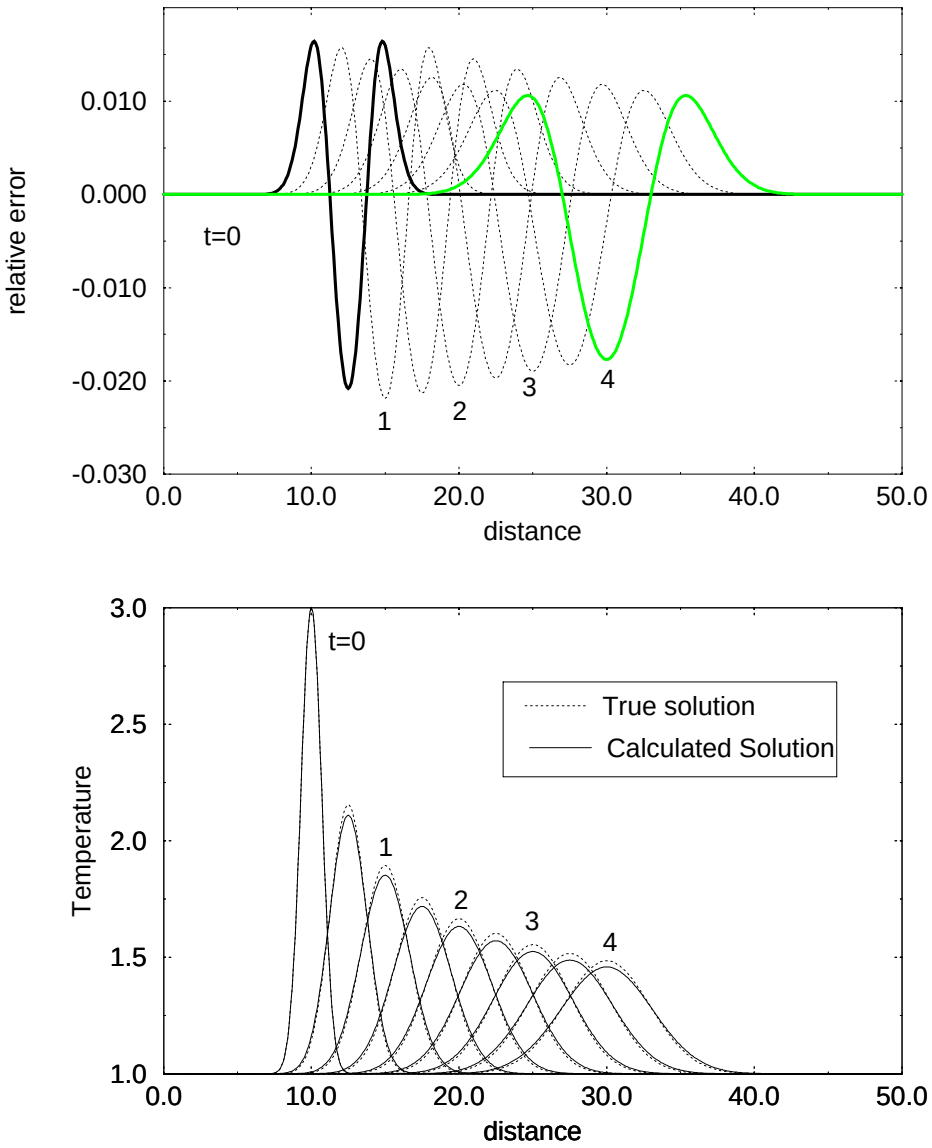


Figure 7.3: Solution and error of the model problem using operator splitting between a first order MPDATA (upwind-differencing, `ncor=1`) scheme for advection and a crank-nicholson scheme for diffusion. This problem has 400 steps. Note the extreme errors due to upwind differencing

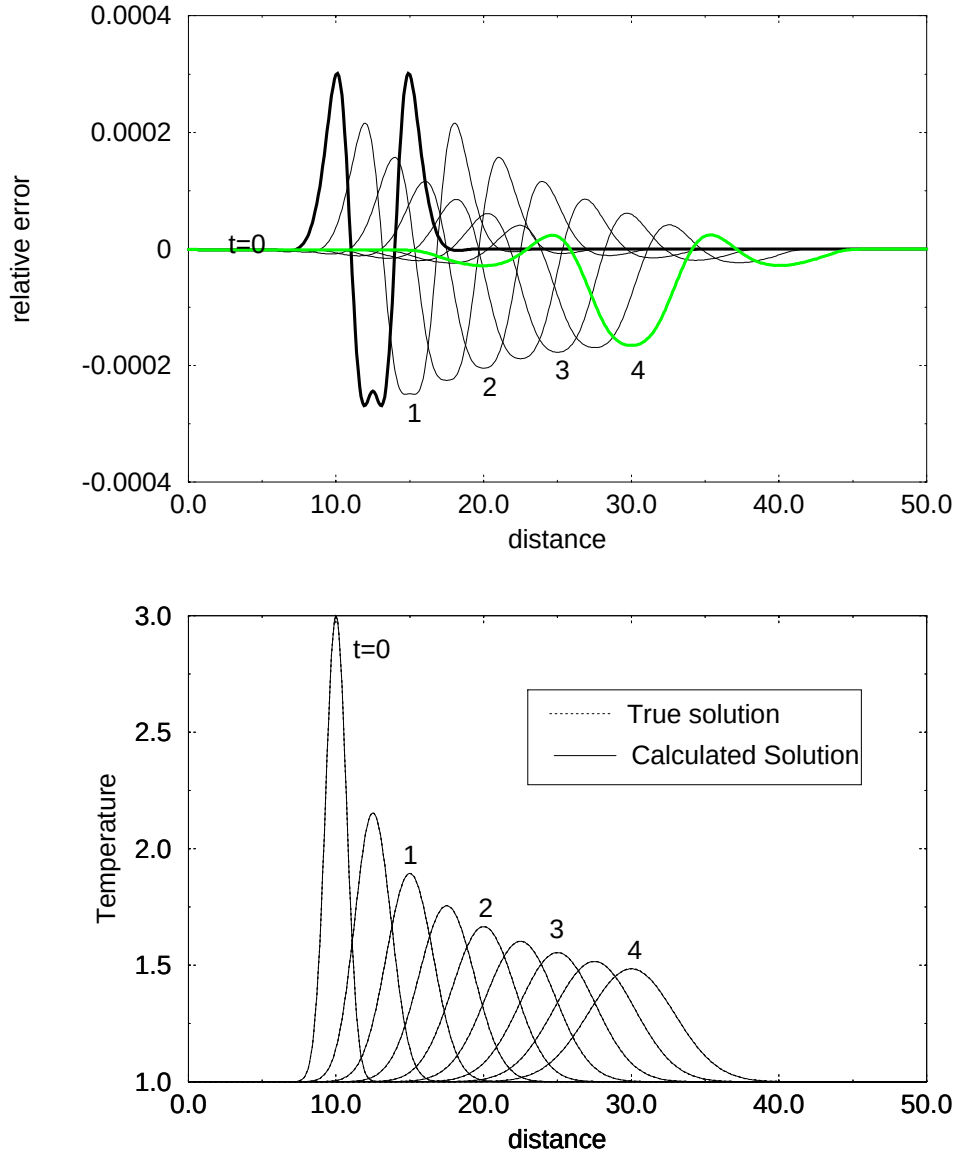


Figure 7.4: Same as Fig. 7.3 but with a second correction in MPDATA. Pretty amazing huh?. But see Fig. 7.6.

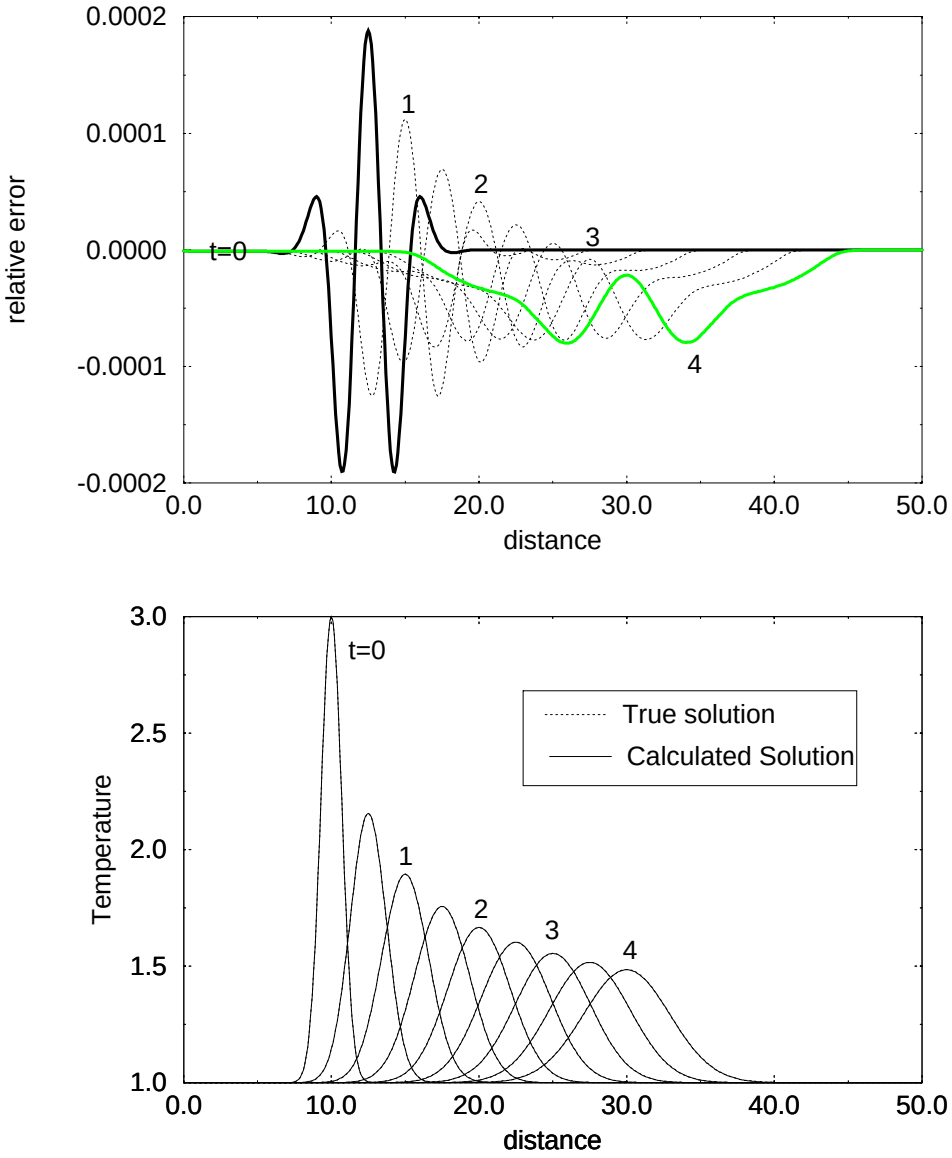


Figure 7.5: Same as Fig. 7.3 but with a third correction in MPDATA. Adding the anti-dispersive correction is almost indistinguishable as the diffusion removes the steep gradients that feed dispersion.

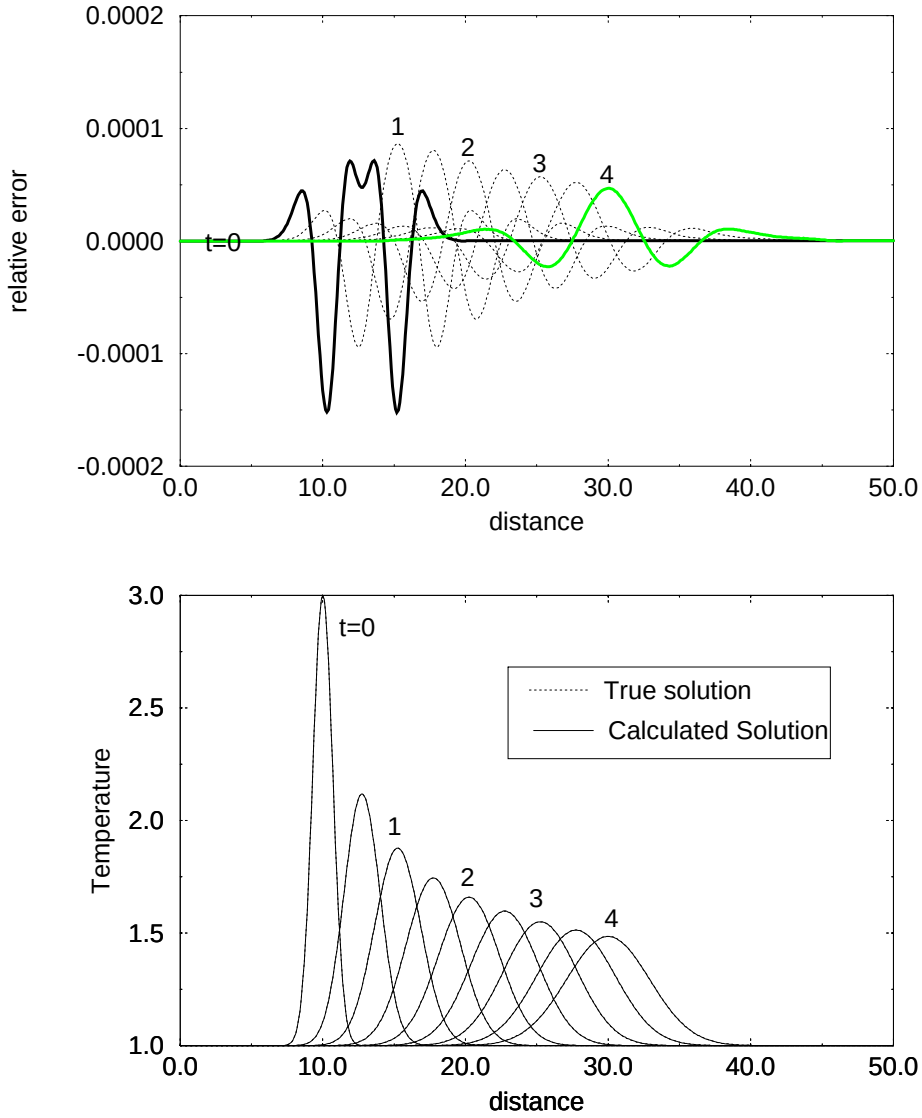


Figure 7.6: Same as Fig. 7.5 but now the advection step is done using a semi-lagrangian scheme with fractional stability condition $\alpha = 2.5$ (which seems to produce the smallest overall errors. Both larger and smaller step cause the error to increase). This scheme has comparable or smaller error than the best mpdata scheme uses 1/5th of the steps and is about an order of magnitude faster.

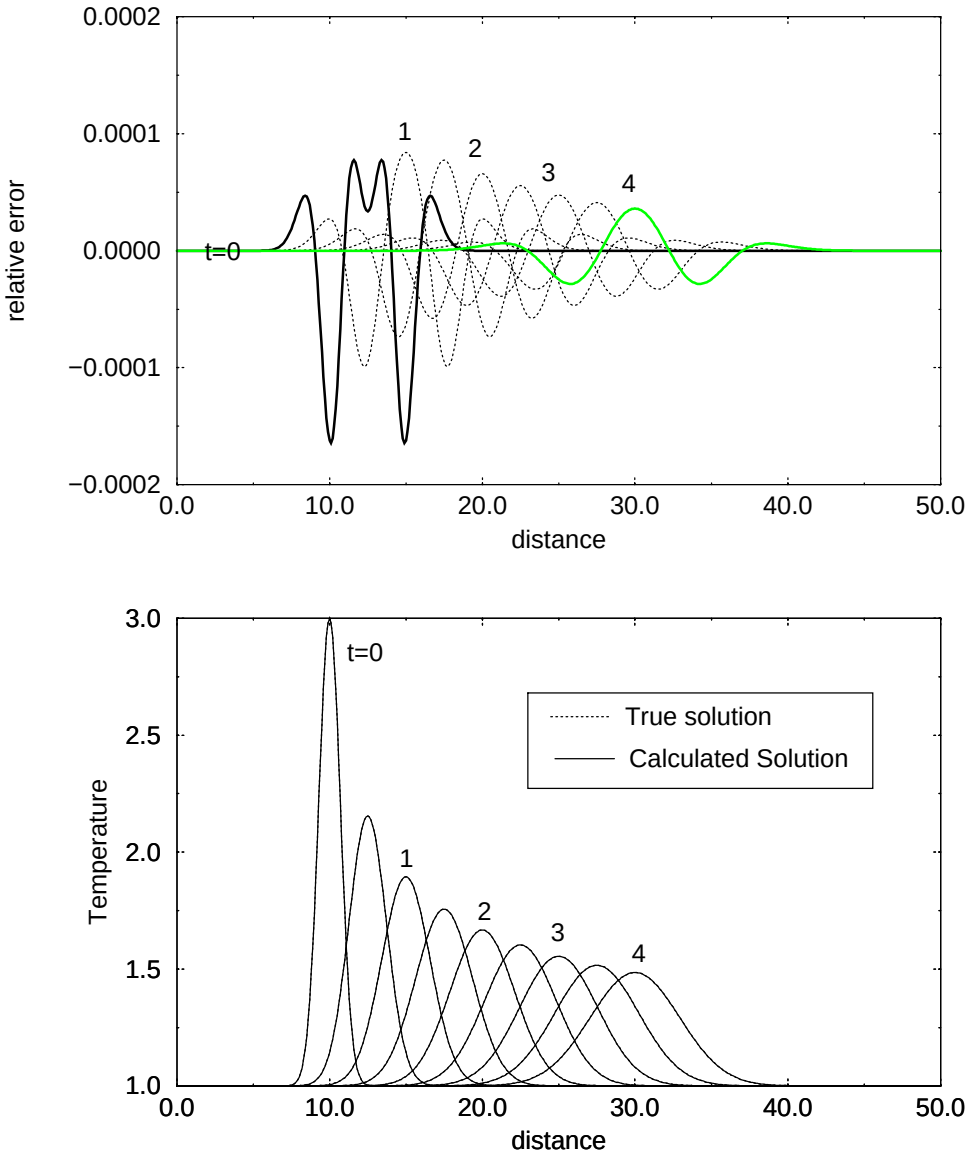


Figure 7.7: Results for the Semi-Lagrangian-Crank-Nicholson scheme which are identical to those for the SL-CN operator splitting problem (Fig. 7.6). This scheme, however, requires double precision to avoid some floating point drift engendered from the tridiagonal solvers.

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.

settohdf `Usage: settohdf filename` converts a 2-D set to a HDF file (Hierarchical Data Format) (filename.hdf) for easy reading into Transform or NCSA image.

settogrd `Usage: settogrd filename` converts a 2-D set to a netCDF .grd file (filename.grd) for use in GMT software.

settofld `Usage: settofld filename` converts either 2 or 3-D sets to the native AVS field file (filename.fld). Particularly useful for 3-D visualization.

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`, `c2dtoxyz`, `c2dtopgm`, `c2dtomat`, `c2dtohdf`, `c2dtogrd`, `c2dtofld`. To get the basic usage and any options just type the name of the script. In addition, there are two more scripts to make pseudo-colored gif files `c2dtogif` and `c2dtogifg`. The former makes colored file mapped between some prescribed min and max value (or just uses the min and max of each file) while the latter maps the colors to the min and max values of the entire run. To set the width of any of the image files set the environment variable `PGMWIDTH`, i.e. to get images that are 200 pixels wide first use `setenv PGMWIDTH 200` then call `c2dtogif`. Finally I have a few routines for making animated 2-D movies. The most general is `mergegif` which will make a multi-part rainbow colored gif movie that can be played back with `xanim` or `netscape`. This script also uses an environment variable `MOVIEENV` which controls the width of the movie in pixels, the delay in milli-seconds between frames and the number of loops that the movie runs. To make a movie that is 200 pixels wide, can be controlled in speed and loops forever, use `setenv MOVIEENV "200 1 0"` (note the quotes are important). All 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 **h** is the displacement vector from point **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 (**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

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

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

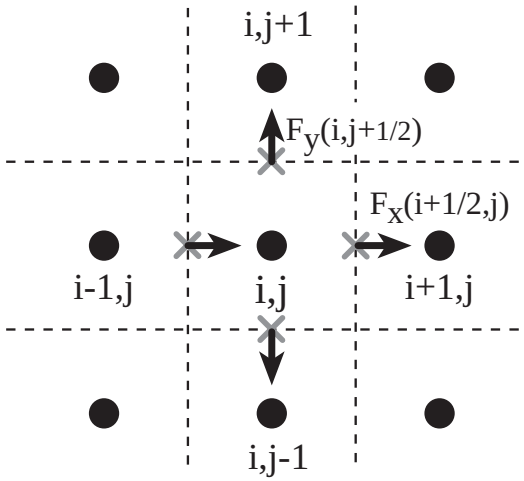


Figure 8.1: A sample of a staggered-mesh used for developing finite difference schemes using the control-volume approach. The principal nodes are denoted by dots, the half-nodes on the staggered mesh are shown by crosses. A cell-centered control volume is the small rectangular region of area $\Delta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

any flux through this volume can be approximated by

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

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

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

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

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

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

$$T_{i,j}^{n+1} = \left[\begin{array}{ccc} & a_5 & \\ a_2 & a_3 & a_4 \\ & a_1 & \end{array} \right]_{i,j} T_{i,j}^{n} \tag{8.3.12}$$

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

$$\mathbf{a}_{1,j} = \left[\begin{array}{ccc} & 0 & \\ 0 & 1 & 0 \\ & 0 & \end{array} \right] \tag{8.3.13}$$

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

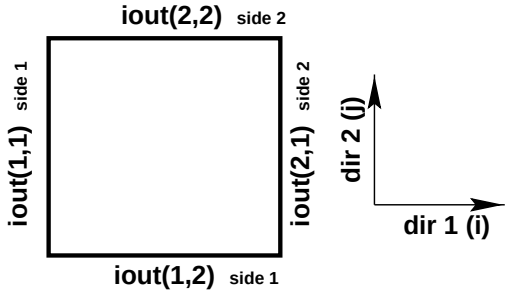


Figure 8.2: A figure trying to explain the ordering of the `iout` array.

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 $\mathbf{V}$ 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

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

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.

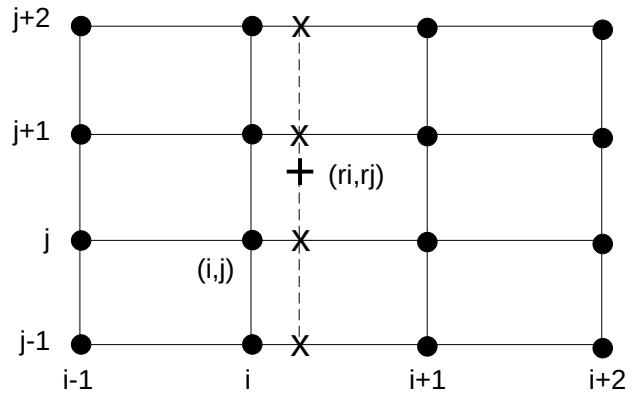


Figure 8.3: Diagram showing bicubic interpolation. Given a regular 2-D grid of values (note Δx not necessarily $= \Delta y$) we wish to find the value of the interpolated function at some intermediate point (ri, rj). As shown here, first do a horizontal interpolation along each of the j lines to find 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 ($\boldsymbol{\nabla} \cdot \mathbf{V} = 0$) 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 $\boldsymbol{\nabla} \cdot \mathbf{V} = 0$. 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

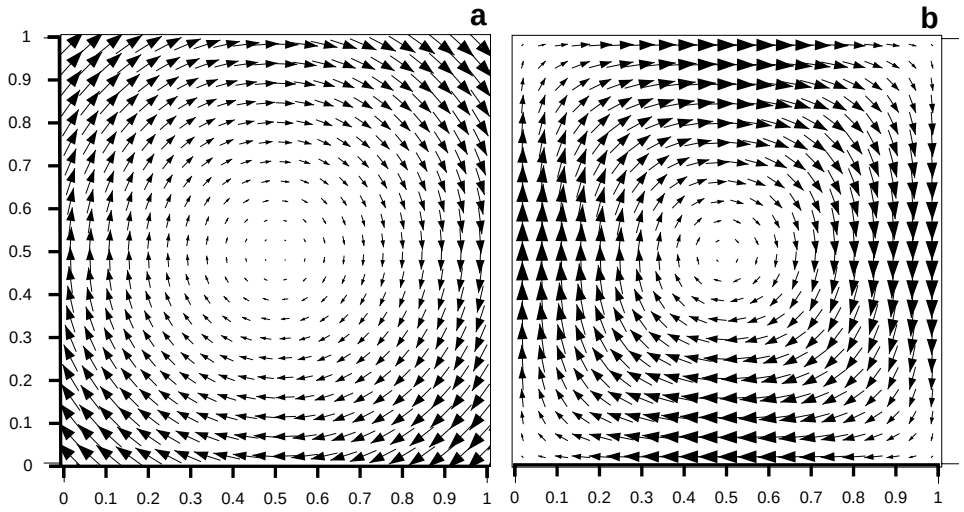


Figure 8.4: Vector plots showing advective velocity fields for (a) rigid-body rotation and (b) a single shear cell. The maximum velocity in 8.4a is 0.71, and is 1. in 8.4b.

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

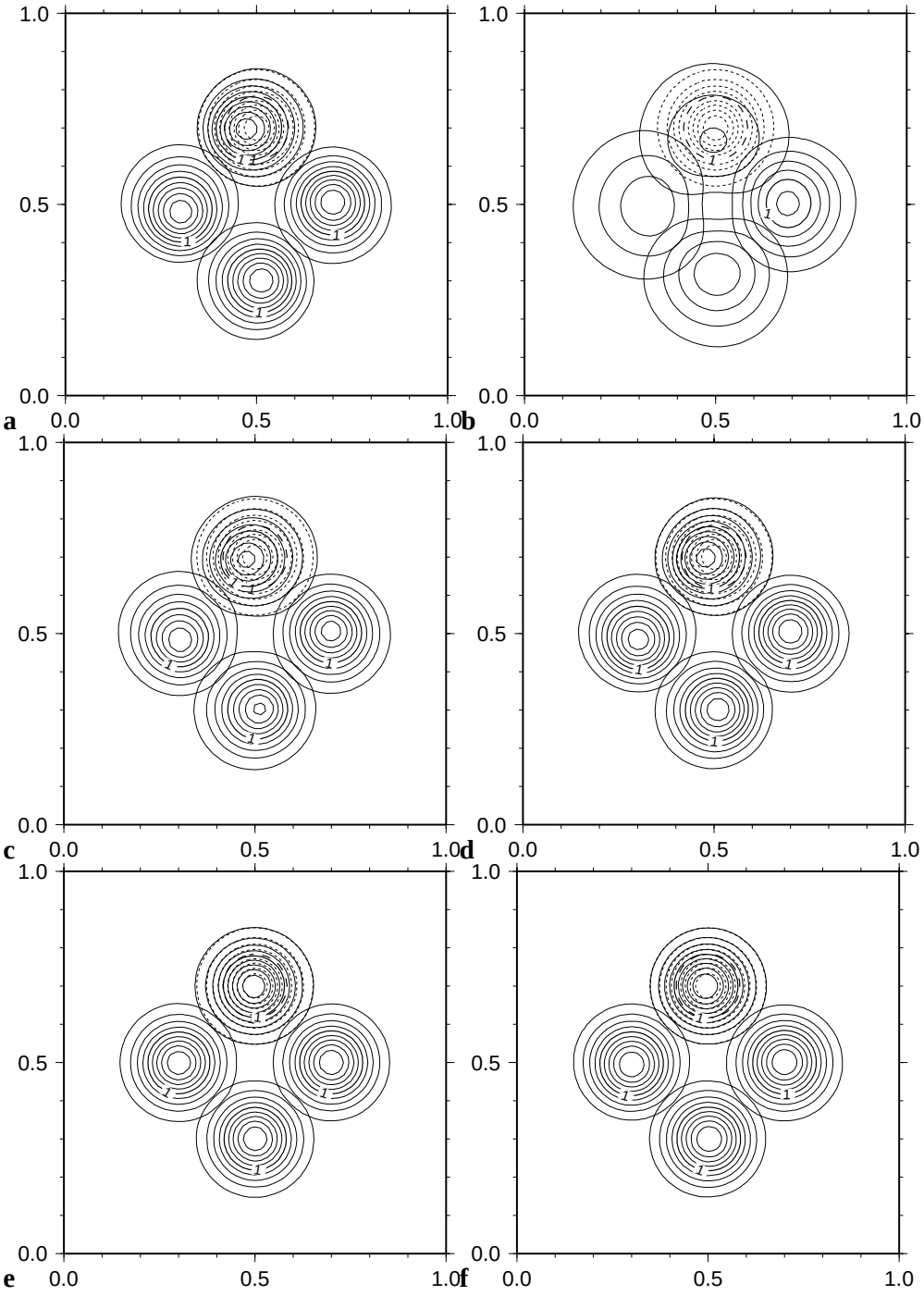


Figure 8.5: Results of the rigid body rotation test for a variety of methods. In each panel, the original gaussian initial condition is shown by dotted contours with the solution at subsequent quarter rotations shown in solid contours up to one full rotation. The direction of rotation is clockwise. The 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.

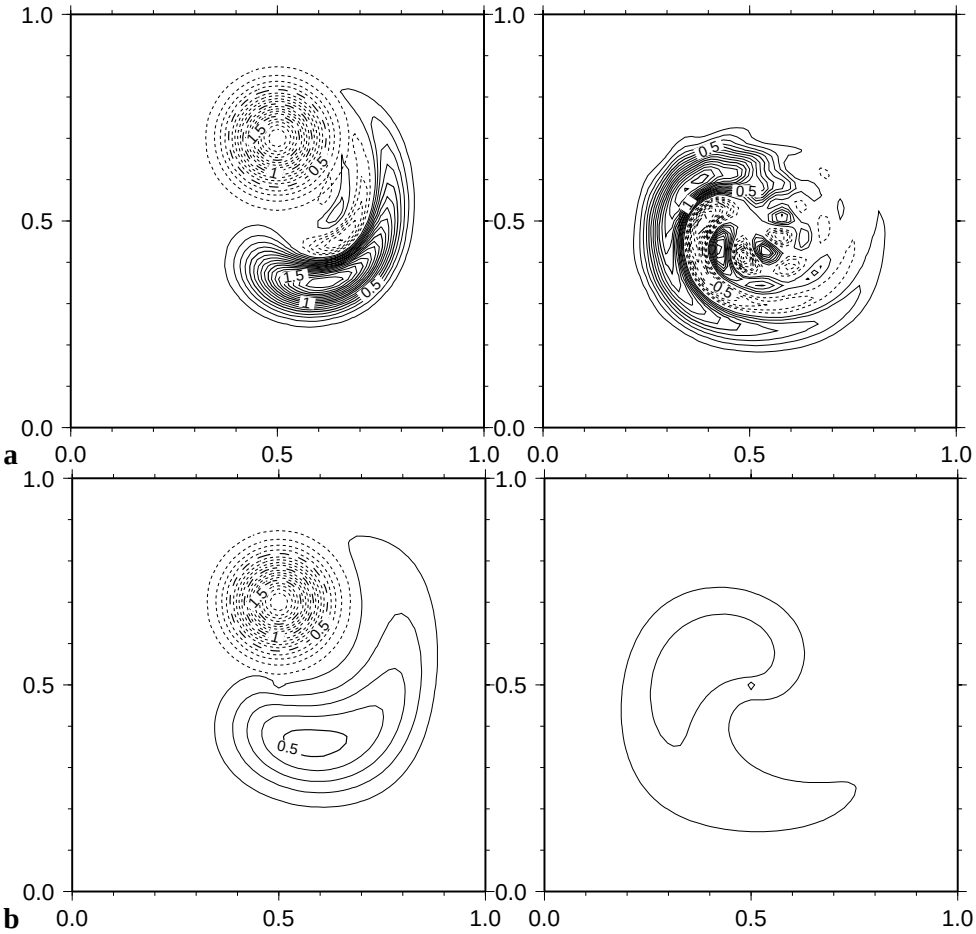


Figure 8.6: Results of the shear cell rotation test for a variety of methods. For each method, the first panel shows the original gaussian initial condition (dotted contours) and the solution at a half rotation (solid contours). The second panel shows the solution after a full rotation. The direction 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)

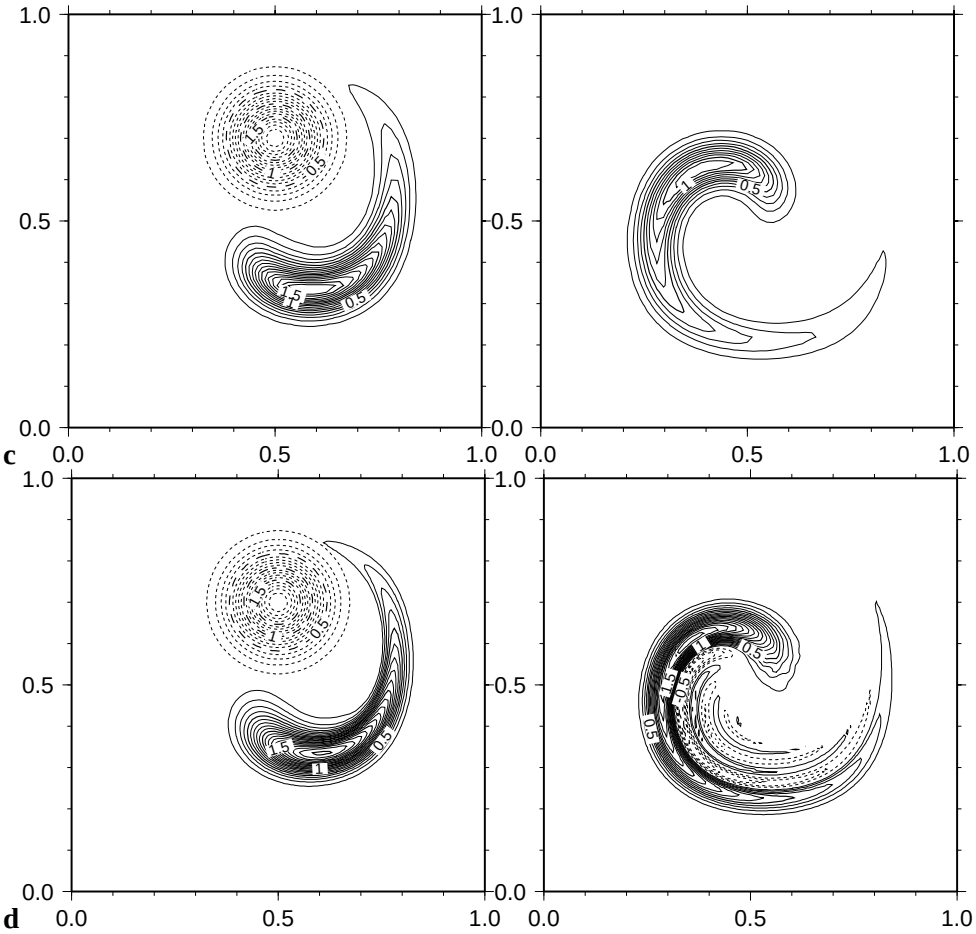


Figure 8.6 contd.: (c) even the best `mpdata` scheme with 2 corrections + 3rd order anti-dispersive correction shows significant diffusion (d) staggered-leapfrog double resolution has noticeable dispersion but might be reasonable for short runs and is cheap and easy.

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

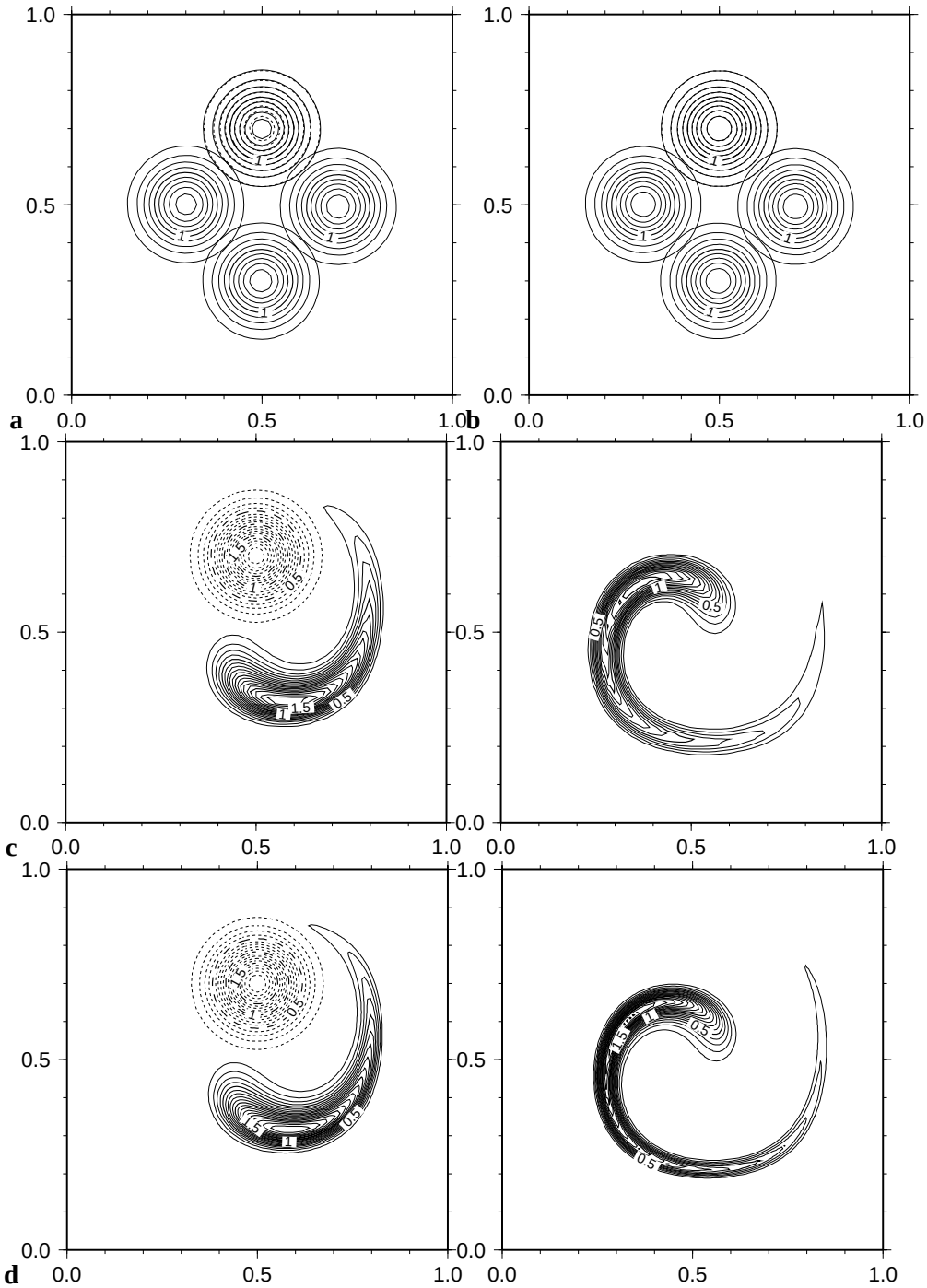


Figure 8.7: Showing off the semi-Lagrangian solvers for rigid body and shear cell tests. (*a*) Rigid body test on a 65×65 grid. (*b*) Rigid body test on 129×129 grid. (*c*) Shear test, 65×65 grid. (*d*) shear test, 129×129. In particular for the high-resolution shear test, the semi-Lagrangian schemes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

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

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

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

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

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

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 $\mathbf{x}_0$

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

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

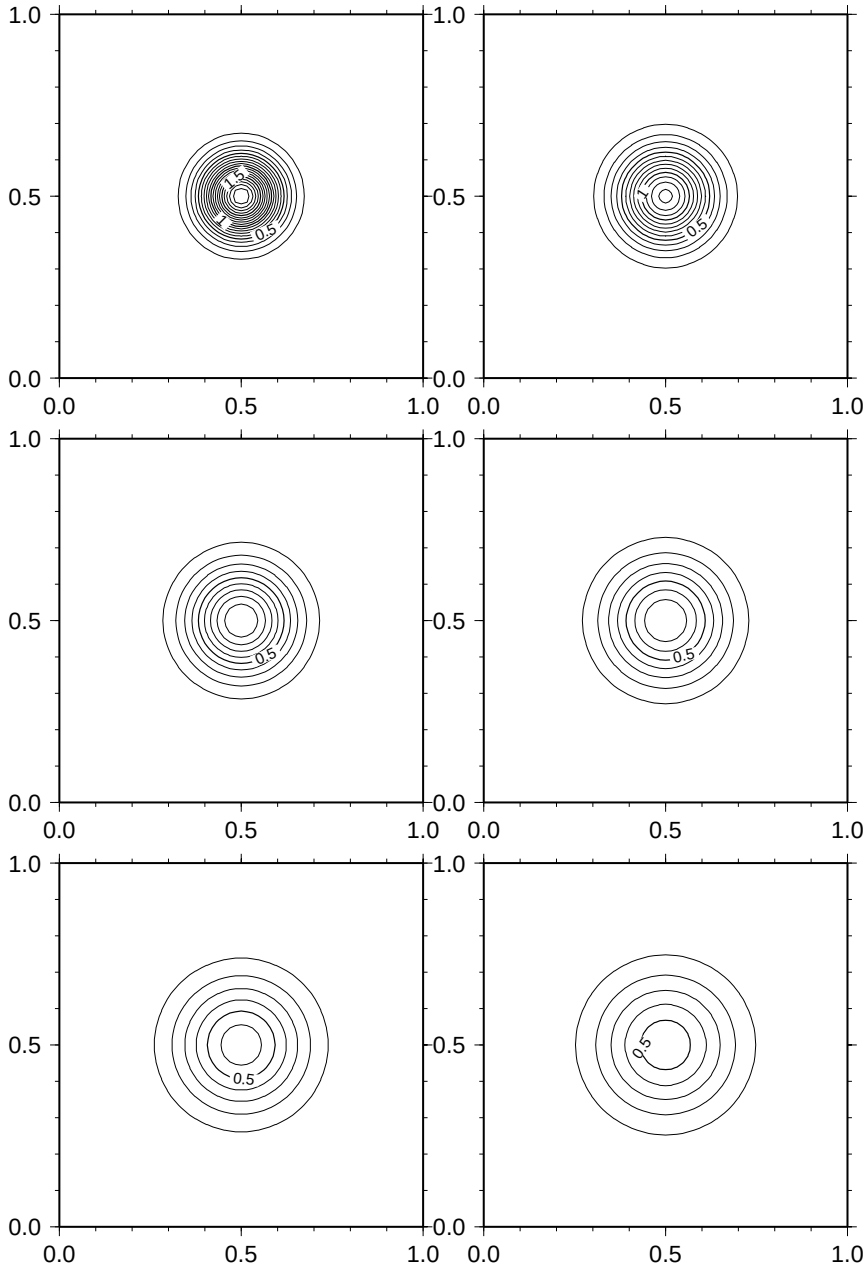


Figure 8.8: Contour plots showing the diffusion of a gaussian initial condition. This plot uses a FTCS explicit scheme on a 65×65 grid with no-flux boundary conditions on all 4 walls. Stability of this scheme requires a small time step and this run uses 400 steps (although the elapsed time is ~ 1 second).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

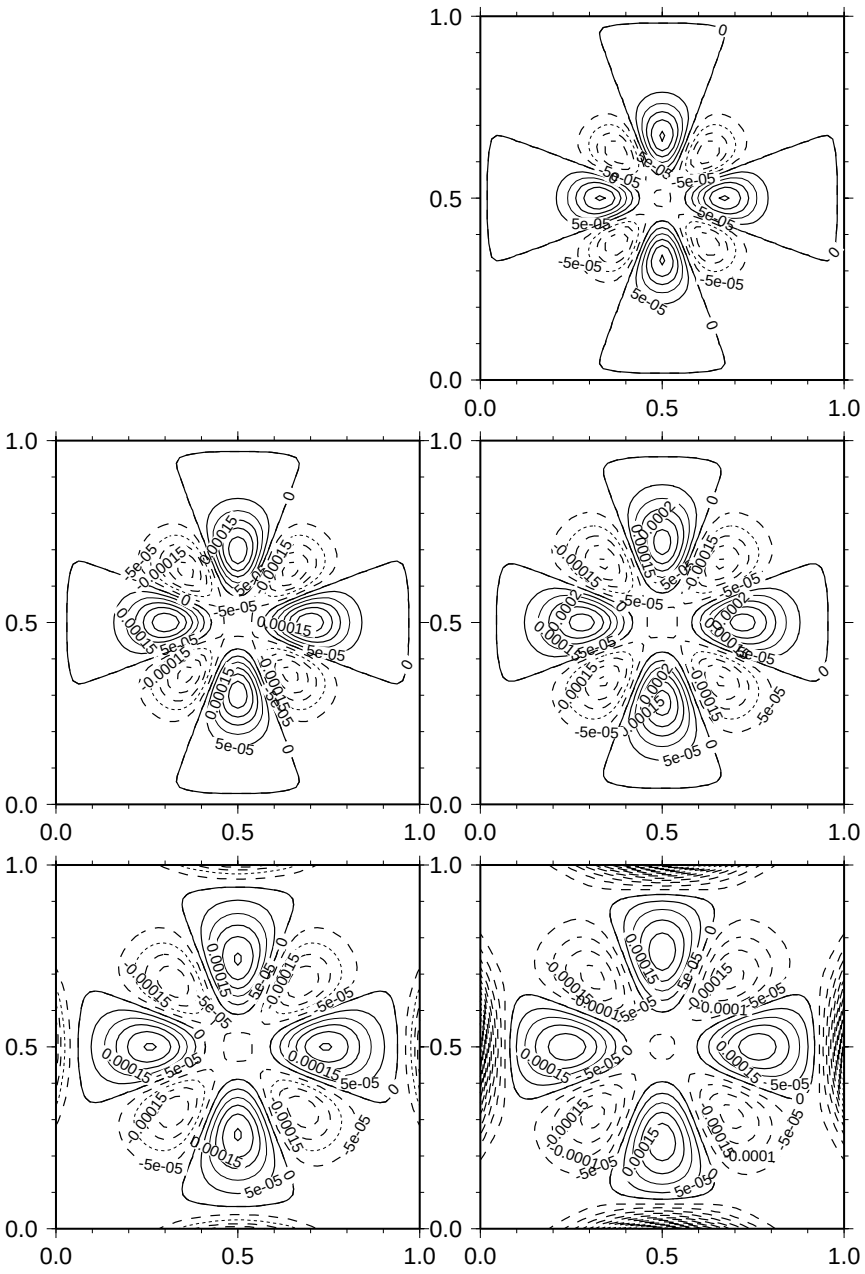


Figure 8.9: Contour plots showing the error from the true solution for figure 8.8 and the FTCS scheme. The maximum error in this scheme (not counting the boundaries) is about 2×10^{-4}. Note the symmetry of the grids that is apparent in the error plots.

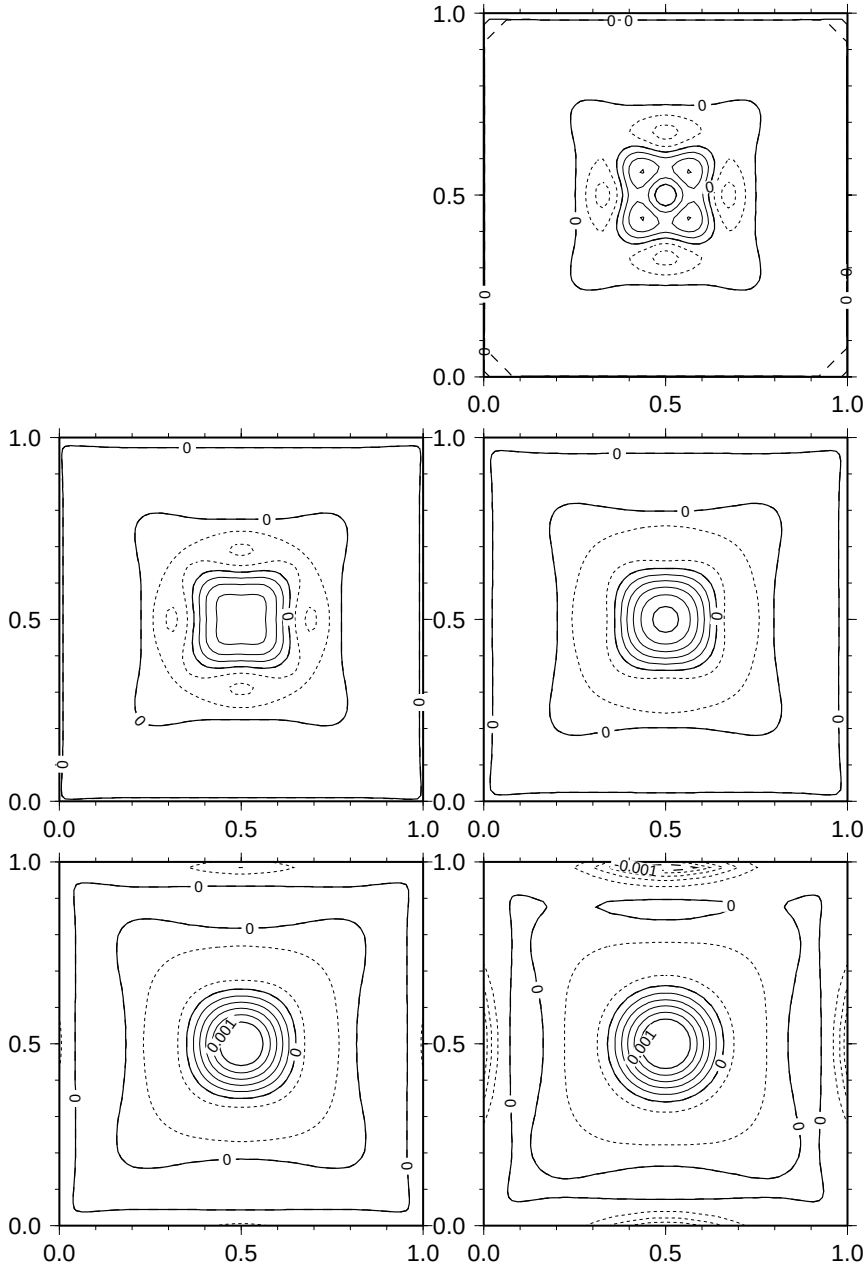


Figure 8.10: Contour plots showing the error from the true solution for figure 8.8 and the ADI scheme. The maximum error in this scheme (not counting the boundaries) is about 1×10^{-3}. The error in this scheme tends to be larger, but more evenly distributed (and more than accurate enough for jazz).

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.

Chapter 9

Boundary Value problems

Selected Reading

Numerical Recipes, 2nd edition: Chapter 19

Briggs, William L. 1987, A Multigrid Tutorial, SIAM, Philadelphia.

In previous sections we have been been concerned with time dependent *initial value problems* where we start with some assumed initial condition, calculate how this solution will change in time and then simply march through time updating as we go. We have considered both explicit and implicit schemes, but the basic point is that given a starting place it's relatively straightforward to get to the next step. The principal difficulty with time dependent schemes is stability and accuracy (they're not the same thing), i.e. how to march through time without blowing up and arriving at a future time with most of your intended physics intact.

Boundary value problems, are a somewhat different animal. In a boundary value problem we are trying to satisfy a steady state solution everywhere in space that agrees with our prescribed boundary conditions. For a flux conservative problem, the problem becomes finding the set of fluxes at all the nodes such that for every node, what comes in goes out. In general, boundary value problems will reduce, when discretized, to a large and sparse set of linear equations. While stability is no longer a problem, efficiency in solving these equations becomes tantamount. The following sections will first develop some physical intuition into the types and sources of boundary value problems, then show how to discretize them and finally present a potpourri of solution techniques.

9.1 Where BVP's come from: basic physics

Most Boundary value problems arise from steady state solutions of transient problems (or from implicit schemes in transient problems). For example if we were considering the general transient heat flow problem for diffusion plus a source term we could solve the dimensionless equation

$$\frac{\partial T}{\partial t} = \nabla^2 T - \rho(\mathbf{x}) \tag{9.1.1}$$

starting with some initial condition (plus appropriate boundary conditions) and march through time to steady state where $\partial T/\partial t = 0$. If your initial guess is far from steady state, though, this may take some time. Alternatively if you were only interested in the final steady state temperature distribution then you would like to solve immediately for the temperature by finding the temperature distribution that satisfies

$$\nabla^2 T = \rho(\mathbf{x}) \tag{9.1.2}$$

This is simply the temperature distribution where the divergence of the heat flux ($\boldsymbol{\nabla} \cdot k\boldsymbol{\nabla}T$) out of any volume exactly balances the source terms in that volume. Of course, for this solution to make any sense, both the boundary conditions and source terms must be independent of time. An analogous problem shows up in solving for fluid flow in a porous medium where the flux is governed by Darcy's law. Here we want the net flux out of any region to balance any source or sink terms so we have

$$\boldsymbol{\nabla} \cdot k\boldsymbol{\nabla}\mathcal{P} = S(\mathbf{x}) \tag{9.1.3}$$

where k is the hydraulic conductivity (or permeability) and $\mathcal{P}$ is the fluid pressure gradient (not exactly but close enough for jazz).

Similarly if we were interested in the flow of a very viscous fluid, we could solve the full Navier Stokes equation

$$\frac{\partial \mathbf{V}}{\partial t} + \mathbf{V} \cdot \boldsymbol{\nabla}\mathbf{V} = \nu\nabla^2 V - \frac{1}{\rho}\boldsymbol{\nabla}P + \mathbf{g} \tag{9.1.4}$$

by considering the accelerations (RHS terms) of each particle of fluid and updating. However, for problems where the dynamic viscosity ν is very large, these accelerations are negligible and the stress propagates almost instantaneously across the medium. Thus rather than being constrained to solve on the short viscous relaxation time we would like to assume that the system comes into balance instantaneously such that

$$0 = \nu\nabla^2 V - \frac{1}{\rho}\boldsymbol{\nabla}P + \mathbf{g} \tag{9.1.5}$$

which is again an elliptic (2nd order) boundary value problem for the vector velocity $\mathbf{V}$. A completely analogous problem exists for elastic materials, i.e. we can either track the transient stress changes which propagate at the speed of sound as seismic waves through the medium until they settle down, or we can just assume they will eventually settle out and solve for the static case

$$\boldsymbol{\nabla} \cdot \boldsymbol{\tau} = \mathbf{f} \tag{9.1.6}$$

for the deviatoric stress tensor $\boldsymbol{\tau}$. If we can get to the steady state case faster than it would take to track the seismic waves then we have made a huge improvement in efficiency. The big problem for boundary value problems is how to jump to the finish line efficiently. By the way, it is important to note that some problems may not have steady state solutions or if they do, these solutions are unstable to perturbations. In general, it pays to think and be careful, half the fun of any problem may be in getting there....

. . . and how to solve them In the following sections we will describe several approaches for discretizing and solving elliptic (2nd order) boundary value problems. For illustration we will consider the simplest elliptic problem which is a *Poisson problem* of the form

$$\nabla^2 U = f(\mathbf{x}) \tag{9.1.7}$$

and its first generalization to a medium with spatially varying properties

$$\boldsymbol{\nabla} \cdot k \boldsymbol{\nabla} U = f(\mathbf{x}) \tag{9.1.8}$$

where k may be something like a conductivity or permeability and is a non-constant function of space (in worse cases, k may be a function of U and then the problem is non-linear. We won't deal with that complication here.)

In a perfect world, Eq. (9.1.8) may have analytic solutions or integral solutions by way of Green functions or Fourier integrals. Any book on Complex analysis or mathematical physics will have details. It should be noted, however, that often, the "analytic" solution requires at least as much computational power to solve as its discrete numerical solution. Still if you can find it, an analytic solution often is often more informative.

9.2 Discretization

But the world is never perfect and this is a course on modeling afterall, so if you want to solve your problem numerically we need to begin by making it discrete. This section will discuss discretization by finite-difference techniques. A later section will discuss discretization into Fourier components.

Actually, we have already discussed finite difference approximations to n-dimensional spatial derivatives in the previous section on n-D Initial value problems. For boundary value problems, nothing has changed except that we don't have any time derivatives to kick around any more. For example, the standard 5-point discretization of Eq. (9.1.7) on a regular 2-D cartesian mesh with grid spacing Δx and Δy is

$$\frac{1}{\Delta x^2}[U_{i-1,j} - 2U_{i,j} + U_{i+1,j}] + \frac{1}{\Delta y^2}[U_{i,j-1} - 2U_{i,j} + U_{i,j+1}] = f_{i,j} \tag{9.2.1}$$

or multiplying through by Δy^2 and writing in stencil form

$$\begin{bmatrix} & 1 & \\ \alpha^2 & -2(1+\alpha^2) & \alpha^2 \\ & 1 & \end{bmatrix} U_{i,j} = \Delta y^2 f_{i,j} \tag{9.2.2}$$

where $\alpha = \Delta x / \Delta y$ is the aspect ratio of a grid cell. For a uniform grid spacing $\Delta x = \Delta y = \Delta$, this is the familiar 5-point stencil

$$\begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix} U_{i,j} = \Delta^2 f_{i,j} \tag{9.2.3}$$

Which is readily generalized to Eq. (9.1.8) on a staggered mesh to be

$$\begin{bmatrix} & k_{i,j+1/2} & \\ \alpha^2 k_{i-1/2,j} & -\Sigma & \alpha^2 k_{i+1/2,j} \\ & k_{i,j-1/2} & \end{bmatrix} U_{i,j} = \Delta y^2 f_{i,j} \tag{9.2.4}$$

where

$$\Sigma = \alpha^2(k_{i-1/2,j} + k_{i+1/2,j}) + k_{i,j-1/2} + k_{i,j+1/2} \tag{9.2.5}$$

is simply the sum of the off-center components of the stencil and $k_{i+1/2,j}$ is the conductivity mid-way between points i and $i+1$ (etc.). It is straightforward to show that for constant k, Eqs. (9.2.4) and (9.2.2) are identical up to the scale of k.

Equations (9.2.1) through (9.2.5) are all systems of linear equations that can be written in the general matrix form

$$\mathsf{A}\mathbf{x} = \mathbf{b} \tag{9.2.6}$$

where A is a penta-diagonal matrix that looks like Figure 9.1. Where each stencil forms one line of the matrix. The important feature of these matrices is that they are incredibly sparse and the problem is to solve for $\mathbf{x}$ efficiently in terms of time and storage. Here we will discuss three general approaches to solving Eq. (9.2.6): *Direct matrix methods* that attempt to find the exact inverse matrix; *rapid methods* that make use of the specific properties of certain classes of problems to provide quick solutions; and *iterative methods* that try to approximate the inverse matrix and apply this approximate matrix repetitively to reduce the error. Each of these methods has its strengths and weaknesses and (as usual) we will highlight the basic do's and dont's.

Finally, a note about boundary conditions. In addition to specifying the interior stencil for determining A it is necessary to specify the boundary conditions (after all it's a boundary value problem). In general boundary conditions add auxiliary information that modifies the matrix or the right hand-side or both. However, there are many ways to implement the boundary conditions and these depend somewhat on the method of solution and will be dealt with in each of the sections below.

9.3 Direct Methods

Given that the basic problem is to somehow solve $\mathbf{x} = \mathsf{A}^{-1}\mathbf{b}$, one might think that the best way to proceed is to use Matlab or some other *direct-solver* to calculate A^{-1} directly and do a matrix multiplication on $\mathbf{b}$. Actually , this is not necessarily a bad idea for small problems or problems with complicated matrices, or for problems where you really only have to calculate A^{-1} once. However, beware general matrix inversion is not a trivial operation. It can be shown that for a general non-sparse matrix with N diagonals (i.e. N unknowns in $\mathbf{x}$), that calculating A^{-1} is an order N^3 operation (ouch!) and therefore simply doubling your grid in each direction in 2-D causes solution time to increase by 64 times. This is not good. The point here is not to use a general dense-matrix solver like the LU decomposition schemes in Numerical Recipes for a matrix as sparse as those shown in Fig.

9.1 because you mainly spend your time operating on zero entries. Much better routines are found in Matlab or public domain software such as `Y12M` (which is available from netlib) that implement sparse LU decomposition that tries to minimize the number of zero operations and the number of zero *fill-ins* in the inverse matrix. Nevertheless while these solvers can be much better than N^3 solvers, the solution times still can scale like $N^{1.5}$ to N^2 thus the larger the problem the much longer the time. It is also important to realize that the memory requirements of these schemes also scale badly because it is not usually the case that the inverse matrix has the same sparsity as the matrix itself. Figure 9.2 shows a *contour* plot of the largest entries in A^{-1} and shows a much larger *bandwidth*. In general, for

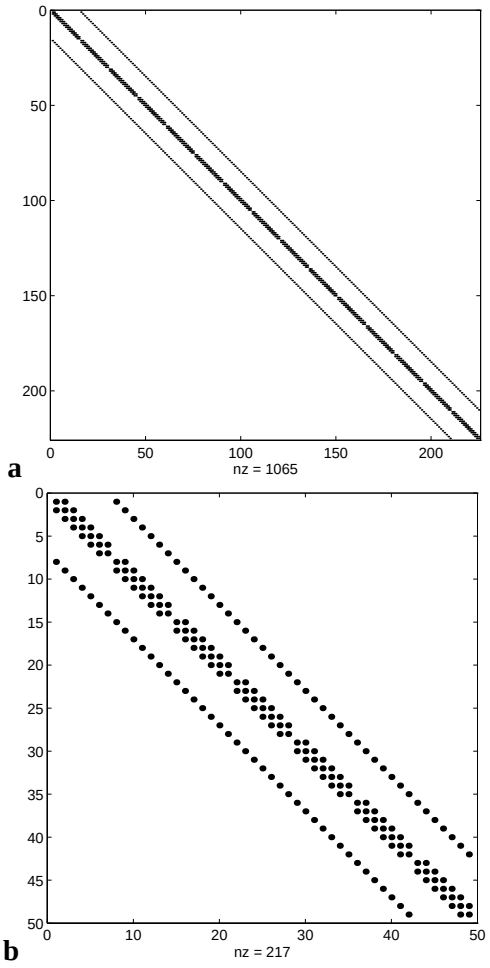


Figure 9.1: (**a**) Structure of a 5-point Laplacian operator on a 17×17 grid with dirichlet boundary conditions. Each dot marks the position of a single non-zero entry. The actual size of matrix A is $(15\times15)^2 = 225^2 = 50625$ entries for the 225 unknown interior points. Note the incredible sparseness of this matrix. Of the 50625 entries, only about 1125 are non-zero. (**b**) Same array but for a 9×9 grid (49 unknowns) to show the structure of the array better

Poisson problems, the required storage for the inverse matrix also scales as $N^{1.5}$ and therefore it is common to run out of memory for even small problems (and the cost of doing the multiplication $\mathsf{A}^{-1}\mathbf{b}$ also increases as $N^{1.5}$.

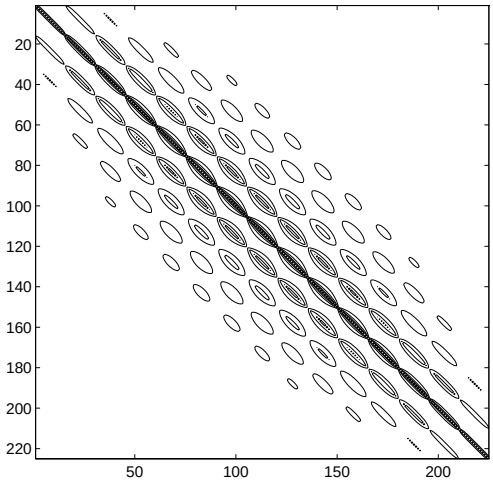


Figure 9.2: Contour map of the inverse of A showing just the largest non-zero entries. Note that the sparsity of A is not necessarily retained by A^{-1}.

9.3.1 Boundary conditions

As promised, we need to talk about implementing boundary conditions in these direct schemes. Because there is no "grid" in these problems (as least as far as the matrix solver is concerned), any modifications due to boundary conditions must be included either in the matrix A or in the right hand side vector $\mathbf{b}$. For *dirichlet* boundary conditions either approach can be used depending on whether you want to include the boundary points in the matrix inversion or not. In general, because matrix methods can be so expensive, the fewer points the better so one approach is just to solve for the unknown interior points[1]. With this method, discretize the unknown points next to the boundary and take all the known values over to the right hand side. For example, if the left edge $i = 1$ is dirichlet, the stencil equation for any non-corner point at $i = 2$

$$\begin{bmatrix} & d & \\ a & b & c \\ & e & \end{bmatrix} x_{2,j} = b_{2,j} \tag{9.3.1}$$

would be replaced by

$$\begin{bmatrix} & d & \\ 0 & b & c \\ & e & \end{bmatrix} x_{2,j} = b_{2,j} - a x_{1,j} \tag{9.3.2}$$

[1]This is why the matrix shown in Fig. 9.1b has $7 \times 7 = 49$ unknowns for a 9×9 grid with dirichlet conditions.

because $x_{1,j}$ is known. Again in sparse-schemes, the zero element in the stencil would not be stored (and actually doesn't lie within the matrix). The other approach for dirichlet conditions is to solve for the boundaries explicitly by writing a stencil equation for the boundary points like

$$\begin{bmatrix} & 0 & \\ 0 & 1 & 0 \\ & 0 & \end{bmatrix} x_{1,j} = f_{1,j} \tag{9.3.3}$$

where $f_{1,j}$ is a known function. This technique allows all the points in the grid to be solved for once without having to deal with subarrays. The actual cost of a dirichlet point is small because Eq. (9.3.3) can be inverted immediately.

For *Neumann or mixed* conditions we modify the matrix as explained in previous sections. I.e. if the left edge ($i = 1$) of the problem is a reflection boundary then we replace the centered stencil equation

$$\begin{bmatrix} & d & \\ a & b & c \\ & e & \end{bmatrix} x_{1,j} = b_{1,j} \tag{9.3.4}$$

with

$$\begin{bmatrix} & d & \\ 0 & b & a+c \\ & e & \end{bmatrix} x_{1,j} = b_{1,j} \tag{9.3.5}$$

For *periodic wrap-around* boundaries, additional matrix entries must be added to make sure the stencil includes the points near the far boundary.

A warning about boundary conditions Note, for elliptic problems and direct solvers, not all boundary conditions will produce unique answers in which case the problem is *ill posed*. For example, the solution of a Poisson problem like Eq. 9.1.7 with either doubly periodic or all Neumann boundaries is ill posed because given any solution u_0 then any solution such $u* = u_0 + c$ where c is a constant is also a solution that matches the boundary conditions. This might seem to be a trivial problem, however, it actually forces the matrix A to be *singular* and therefore to have no unique inverse. This will cause a direct solver to fail although other methods can still be used for this problem.

9.4 Rapid Methods

Direct methods can be useful for small problems with very strange matrix structure (e.g. for some of the matrices produced by finite element methods on unstructured grids). However for simpler problems you can usually do much better. In fact for certain classes of problems (of which the Poisson problem is one), there are *rapid* methods that can take advantage of some of the special properties of these the underlying matrix. Here we will deal with two rapid methods *Fourier Transform* spectral methods and combined *Fourier-cyclic reduction techniques*.

9.4.1 Fourier Transform Solutions

Problems with regular boundaries and constant coefficient stencils can often be solved using Fourier transform or *spectral techniques*. What makes them practical is that the underlying FFT (fast-Fourier transform) is an order $N \log_2 N$ transformation that takes N points in the space-domain exactly to N points in the frequency domain (and back again).

These methods start by noting that any continuous function on a periodic domain of $N_i \times N_j$ points can be expressed exactly by its discrete inverse Fourier transform

$$u_{ij} = \mathcal{F}^{-1}[\hat{u}] = \frac{1}{N_i N_j} \sum_{m=1}^{m=N_i} \sum_{n=1}^{n=N_j} \hat{u}_{mn} e^{-ik_m x} e^{-ik_n y} \tag{9.4.1}$$

where the i's in the exponentials are actually the imaginary number $\sqrt{-1}$ and

$$x = (i-1)\Delta x \qquad y = (j-1)\Delta y \tag{9.4.2}$$

$$k_m = \frac{2\pi(m-1)}{(N_i-1)\Delta x} \qquad k_n = \frac{2\pi(n-1)}{(N_j-1)\Delta y} \tag{9.4.3}$$

where k_m, k_n are the horizontal and vertical discrete wave numbers. The important features of the discrete Fourier transform is that it is linear, invertible and by the magic of the *Fast Fourier Transform* (FFT) can be evaluated in order $N \log_2 N$ operations in each direction[2] (not to mention there are lots of highly optimized versions around).

To use the FFT to solve Eq. (9.1.7), notice that that because ∇^2 is a linear operator, we can find $\nabla^2 u$ by taking the Laplacian of each term in the summation, i.e.

$$\nabla^2 u_{ij} = \frac{1}{N_i N_j} \sum_{m=1}^{m=N_i} \sum_{n=1}^{n=N_j} -(k_m^2 + k_n^2) \hat{u}_{mn} e^{-ik_m x} e^{-ik_n y} \tag{9.4.4}$$

but we also know that we can write the right hand side as

$$f_{ij} = \mathcal{F}^{-1}\left[\hat{f}\right] = \frac{1}{N_i N_j} \sum_{m=1}^{m=N_i} \sum_{n=1}^{n=N_j} \hat{f}_{mn} e^{-ik_m x} e^{-ik_n y} \tag{9.4.5}$$

thus term by term it must be true that

$$-(k_m^2 + k_n^2)\hat{u}_{mn} = \hat{f}_{mn} \tag{9.4.6}$$

Now $\hat{f}_{mn} = \mathcal{F}[f]$ is simply the Fourier transform of the right hand side and is readily evaluated. Thus to solve Eq.(9.1.7) we first find the FFT of the right-hand side, then divide each component by $(k_m^2 + k_n^2)$ (being particularly careful around

[2] so for simple power of 2 grids where $N_i = 2^p$ and $N_j = 2^q$ then the 2-D FFT takes $O(N_i N_j pq)$ operations e.g. a 64×64 grid takes $O(36 \times 65^2)$ operations

the D-C component which should be zero[3]) and then finding u_{ij} by inverse transformation, i.e.

$$u_{ij} = \mathcal{F}^{-1}\left[\frac{-\hat{f}_{mn}}{(k_m^2 + k_n^2)}\right] \tag{9.4.7}$$

This approach will automatically work for doubly periodic boundaries because the discrete FFT's implicitly assume the functions are periodic. For dirichlet boundaries where $u = 0$, you can expand the function in terms of discrete $\sin$ series, likewise for reflection boundaries you can use $\cos$ series. Numerical Recipes discusses these and more general boundary conditions in some detail as well as a slightly different approach for solving the discreet equation $\mathsf{A}\mathbf{u} = \mathbf{f}$ directly.

As an aside, spectral methods also can be used in initial value problems although they most often occur as *pseudo-spectral* techniques where the equations are not actually solved in the wave-number domain, however FFT's are used to evaluate spatial derivatives. For example, so far we have been evaluating advection terms like $\mathbf{V} \cdot \boldsymbol{\nabla} T$ using a local stencil for the gradient operator. However, using the same trick we used to evaluate $\nabla^2 U$ in Eq. (9.4.4) we could also evaluate $\boldsymbol{\nabla} T$ as

$$\boldsymbol{\nabla} T = \mathcal{F}^{-1}\left[-i\mathbf{k}\mathcal{F}\left[T\right]\right] \tag{9.4.8}$$

where $\mathbf{k}$ is the *wave-vector* $k_m\mathbf{i} + k_n\mathbf{j}$. Thus we first forward transform our temperature file, multiply by $i\mathbf{k}$ and then transform back. The net effect is to calculate the gradients of all the Fourier components simultaneously resulting in an extremely high-order method. Once we have $\boldsymbol{\nabla} T$ we simply form the dot-product with the velocity field and away we go. For problems with smoothly varying fields and regular boundaries, these techniques can be highly accurate with much fewer grid points.

9.4.2 Cyclic Reduction Solvers

Spectral methods work for problems with constant coefficients and regular boundaries. A slightly more general set of rapid methods, however exists for problems that are *separable* in the sense of separation of variables. These methods include *cyclic reduction* and *FACR* (Fourier-analysis and cyclic reduction). Numerical Recipes gives a brief explanation of how these work which I will not repeat. What I will do is tell you about a lovely collection of FACR codes by Swarzrauber and Sweet called FISHPAK[4]. These are a collection of fortran routines for solving more general Helmholtz problems such as

$$\nabla^2 U + \lambda U = f(\mathbf{x}) \tag{9.4.9}$$

in cartesian, cylindrical and polar coordinates on regular and staggered meshes. They also can solve for 3-D cartesian coordinates and the general 2-D separable

[3]it is the possibility of a non-zero mean of the right-hand side that causes this problem to be singular with doubly periodic boundary conditions. This is the same problem that causes the direct methods to fail.

[4]For all you francophones Poisson is fish in french

elliptic problem

$$a(x)\frac{\partial^2 u}{\partial x^2} + b(x)\frac{\partial u}{\partial x} + c(x)u + d(y)\frac{\partial^2 u}{\partial y^2} + e(y)\frac{\partial u}{\partial y} + f(y)u = g(x,y) \quad (9.4.10)$$

for any combination of periodic or mixed boundary conditions. These codes are available from netlib (`http:/www.netlib.org`) and are extremely fast with solution time scaling like $N_i N_j \log_2 N_i$. The principal (minor) draw back however is that they were written (extremely well) back in the late 70's and early 80's and the fortran is inscrutable and therefore hard to modify for more general boundary conditions. In addition they still will only work for separable problems and could not, for example solve the more general problem $\boldsymbol{\nabla} \cdot k\boldsymbol{\nabla} T = \rho$, for a generally spatially varying conductivity. However, the only solvers that can compete in time with these routines and handle spatially varying coefficients are the iterative multi-grid solvers (see below). So if you have a relatively straightforward Poisson problem (particularly in non-cartesian geometries) these codes are highly recommended.

Because these codes are both highly useful and extremely inscrutable, the main thrust of this section is not to explain how they work but how you might use it. As an example, the documentation for the driver for `HWSCRT` (Helmholtz, regular grid, cartesian) is in the program and looks like.

```
      SUBROUTINE HWSCRT (A,B,M,MBDCND,BDA,BDB,C,D,N,NBDCND,BDC,BDD,
     1                   ELMBDA,F,IDIMF,PERTRB,IERROR,W)
C
C
C     * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C     *                                                               *
C     *                        F I S H P A K                          *
C     *                                                               *
C     *                                                               *
C     *     A PACKAGE OF FORTRAN SUBPROGRAMS FOR THE SOLUTION OF      *
C     *                                                               *
C     *      SEPARABLE ELLIPTIC PARTIAL DIFFERENTIAL EQUATIONS        *
C     *                                                               *
C     *                  (VERSION 3.1 , OCTOBER 1980)                  *
C     *                                                               *
C     *                             BY                                *
C     *                                                               *
C     *        JOHN ADAMS, PAUL SWARZTRAUBER AND ROLAND SWEET         *
C     *                                                               *
C     *                             OF                                *
C     *                                                               *
C     *         THE NATIONAL CENTER FOR ATMOSPHERIC RESEARCH          *
C     *                                                               *
C     *                BOULDER, COLORADO  (80307)  U.S.A.             *
C     *                                                               *
C     *                   WHICH IS SPONSORED BY                       *
C     *                                                               *
C     *              THE NATIONAL SCIENCE FOUNDATION                  *
C     *                                                               *
C     * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
C
C     * * * * * * * * *  PURPOSE    * * * * * * * * * * * * * * * * * *
C
C          SUBROUTINE HWSCRT SOLVES THE STANDARD FIVE-POINT FINITE
C     DIFFERENCE APPROXIMATION TO THE HELMHOLTZ EQUATION IN CARTESIAN
C     COORDINATES:
C
C          (D/DX)(DU/DX) + (D/DY)(DU/DY) + LAMBDA*U = F(X,Y).
```

```
C
C
C
C     * * * * * * * *    PARAMETER DESCRIPTION     * * * * * * * * * *
C
C           * * * * * *   ON INPUT    * * * * * *
C
C     A,B
C       THE RANGE OF X, I.E., A .LE. X .LE. B.  A MUST BE LESS THAN B.
C
C     M
C       THE NUMBER OF PANELS INTO WHICH THE INTERVAL (A,B) IS
C       SUBDIVIDED.  HENCE, THERE WILL BE M+1 GRID POINTS IN THE
C       X-DIRECTION GIVEN BY X(I) = A+(I-1)DX FOR I = 1,2,...,M+1,
C       WHERE DX = (B-A)/M IS THE PANEL WIDTH. M MUST BE GREATER THAN 3.
C
C     MBDCND
C       INDICATES THE TYPE OF BOUNDARY CONDITIONS AT X = A AND X = B.
C
C       = 0  IF THE SOLUTION IS PERIODIC IN X, I.E., U(I,J) = U(M+I,J).
C       = 1  IF THE SOLUTION IS SPECIFIED AT X = A AND X = B.
C       = 2  IF THE SOLUTION IS SPECIFIED AT X = A AND THE DERIVATIVE OF
C            THE SOLUTION WITH RESPECT TO X IS SPECIFIED AT X = B.
C       = 3  IF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO X IS
C            SPECIFIED AT X = A AND X = B.
C       = 4  IF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO X IS
C            SPECIFIED AT X = A AND THE SOLUTION IS SPECIFIED AT X = B.
C
C     BDA
C       A ONE-DIMENSIONAL ARRAY OF LENGTH N+1 THAT SPECIFIES THE VALUES
C       OF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO X AT X = A.
C       WHEN MBDCND = 3 OR 4,
C
C            BDA(J) = (D/DX)U(A,Y(J)), J = 1,2,...,N+1  .
C
C       WHEN MBDCND HAS ANY OTHER VALUE, BDA IS A DUMMY VARIABLE.
C
C     BDB
C       A ONE-DIMENSIONAL ARRAY OF LENGTH N+1 THAT SPECIFIES THE VALUES
C       OF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO X AT X = B.
C       WHEN MBDCND = 2 OR 3,
C
C            BDB(J) = (D/DX)U(B,Y(J)), J = 1,2,...,N+1  .
C
C       WHEN MBDCND HAS ANY OTHER VALUE BDB IS A DUMMY VARIABLE.
C
C     C,D
C       THE RANGE OF Y, I.E., C .LE. Y .LE. D.  C MUST BE LESS THAN D.
C
C     N
C       THE NUMBER OF PANELS INTO WHICH THE INTERVAL (C,D) IS
C       SUBDIVIDED.  HENCE, THERE WILL BE N+1 GRID POINTS IN THE
C       Y-DIRECTION GIVEN BY Y(J) = C+(J-1)DY FOR J = 1,2,...,N+1, WHERE
C       DY = (D-C)/N IS THE PANEL WIDTH.  N MUST BE GREATER THAN 3.
C
C     NBDCND
C       INDICATES THE TYPE OF BOUNDARY CONDITIONS AT Y = C AND Y = D.
C
C       = 0  IF THE SOLUTION IS PERIODIC IN Y, I.E., U(I,J) = U(I,N+J).
C       = 1  IF THE SOLUTION IS SPECIFIED AT Y = C AND Y = D.
C       = 2  IF THE SOLUTION IS SPECIFIED AT Y = C AND THE DERIVATIVE OF
C            THE SOLUTION WITH RESPECT TO Y IS SPECIFIED AT Y = D.
C       = 3  IF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO Y IS
C            SPECIFIED AT Y = C AND Y = D.
C       = 4  IF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO Y IS
C            SPECIFIED AT Y = C AND THE SOLUTION IS SPECIFIED AT Y = D.
C
C     BDC
C       A ONE-DIMENSIONAL ARRAY OF LENGTH M+1 THAT SPECIFIES THE VALUES
C       OF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO Y AT Y = C.
C       WHEN NBDCND = 3 OR 4,
C
C            BDC(I) = (D/DY)U(X(I),C), I = 1,2,...,M+1  .
```

```
C
C       WHEN NBDCND HAS ANY OTHER VALUE, BDC IS A DUMMY VARIABLE.
C
C     BDD
C       A ONE-DIMENSIONAL ARRAY OF LENGTH M+1 THAT SPECIFIES THE VALUES
C       OF THE DERIVATIVE OF THE SOLUTION WITH RESPECT TO Y AT Y = D.
C       WHEN NBDCND = 2 OR 3,
C
C            BDD(I) = (D/DY)U(X(I),D), I = 1,2,...,M+1  .
C
C       WHEN NBDCND HAS ANY OTHER VALUE, BDD IS A DUMMY VARIABLE.
C
C     ELMBDA
C       THE CONSTANT LAMBDA IN THE HELMHOLTZ EQUATION.  IF
C       LAMBDA .GT. 0, A SOLUTION MAY NOT EXIST.  HOWEVER, HWSCRT WILL
C       ATTEMPT TO FIND A SOLUTION.
C
C     F
C       A TWO-DIMENSIONAL ARRAY WHICH SPECIFIES THE VALUES OF THE RIGHT
C       SIDE OF THE HELMHOLTZ EQUATION AND BOUNDARY VALUES (IF ANY).
C       FOR I = 2,3,...,M AND J = 2,3,...,N
C
C            F(I,J) = F(X(I),Y(J)).
C
C       ON THE BOUNDARIES F IS DEFINED BY
C
C            MBDCND     F(1,J)        F(M+1,J)
C            ------     ---------     --------
C
C              0        F(A,Y(J))     F(A,Y(J))
C              1        U(A,Y(J))     U(B,Y(J))
C              2        U(A,Y(J))     F(B,Y(J))     J = 1,2,...,N+1
C              3        F(A,Y(J))     F(B,Y(J))
C              4        F(A,Y(J))     U(B,Y(J))
C
C
C            NBDCND     F(I,1)        F(I,N+1)
C            ------     ---------     --------
C
C              0        F(X(I),C)     F(X(I),C)
C              1        U(X(I),C)     U(X(I),D)
C              2        U(X(I),C)     F(X(I),D)     I = 1,2,...,M+1
C              3        F(X(I),C)     F(X(I),D)
C              4        F(X(I),C)     U(X(I),D)
C
C       F MUST BE DIMENSIONED AT LEAST (M+1)*(N+1).
C
C       NOTE
C
C       IF THE TABLE CALLS FOR BOTH THE SOLUTION U AND THE RIGHT SIDE F
C       AT  A CORNER THEN THE SOLUTION MUST BE SPECIFIED.
C
C     IDIMF
C       THE ROW (OR FIRST) DIMENSION OF THE ARRAY F AS IT APPEARS IN THE
C       PROGRAM CALLING HWSCRT.  THIS PARAMETER IS USED TO SPECIFY THE
C       VARIABLE DIMENSION OF F.  IDIMF MUST BE AT LEAST M+1  .
C
C     W
C       A ONE-DIMENSIONAL ARRAY THAT MUST BE PROVIDED BY THE USER FOR
C       WORK SPACE.  W MAY REQUIRE UP TO 4*(N+1) +
C       (13 + INT(LOG2(N+1)))*(M+1) LOCATIONS.  THE ACTUAL NUMBER OF
C       LOCATIONS USED IS COMPUTED BY HWSCRT AND IS RETURNED IN LOCATION
C       W(1).
C
C
C            * * * * * *   ON OUTPUT     * * * * * *
C
C     F
C       CONTAINS THE SOLUTION U(I,J) OF THE FINITE DIFFERENCE
C       APPROXIMATION FOR THE GRID POINT (X(I),Y(J)), I = 1,2,...,M+1,
C       J = 1,2,...,N+1  .
C
C     PERTRB
```

```
C       IF A COMBINATION OF PERIODIC OR DERIVATIVE BOUNDARY CONDITIONS
C       IS SPECIFIED FOR A POISSON EQUATION (LAMBDA = 0), A SOLUTION MAY
C       NOT EXIST.  PERTRB IS A CONSTANT, CALCULATED AND SUBTRACTED FROM
C       F, WHICH ENSURES THAT A SOLUTION EXISTS.  HWSCRT THEN COMPUTES
C       THIS SOLUTION, WHICH IS A LEAST SQUARES SOLUTION TO THE ORIGINAL
C       APPROXIMATION.  THIS SOLUTION PLUS ANY CONSTANT IS ALSO A
C       SOLUTION.  HENCE, THE SOLUTION IS NOT UNIQUE.  THE VALUE OF
C       PERTRB SHOULD BE SMALL COMPARED TO THE RIGHT SIDE F.  OTHERWISE,
C       A SOLUTION IS OBTAINED TO AN ESSENTIALLY DIFFERENT PROBLEM.
C       THIS COMPARISON SHOULD ALWAYS BE MADE TO INSURE THAT A
C       MEANINGFUL SOLUTION HAS BEEN OBTAINED.
C
C    IERROR
C       AN ERROR FLAG THAT INDICATES INVALID INPUT PARAMETERS.  EXCEPT
C       FOR NUMBERS 0 AND 6, A SOLUTION IS NOT ATTEMPTED.
C
C       = 0  NO ERROR.
C       = 1  A .GE. B.
C       = 2  MBDCND .LT. 0 OR MBDCND .GT. 4  .
C       = 3  C .GE. D.
C       = 4  N .LE. 3
C       = 5  NBDCND .LT. 0 OR NBDCND .GT. 4  .
C       = 6  LAMBDA .GT. 0  .
C       = 7  IDIMF .LT. M+1  .
C       = 8  M .LE. 3
C
C       SINCE THIS IS THE ONLY MEANS OF INDICATING A POSSIBLY INCORRECT
C       CALL TO HWSCRT, THE USER SHOULD TEST IERROR AFTER THE CALL.
C
C    W
C       W(1) CONTAINS THE REQUIRED LENGTH OF W.
C
C
C    * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Well if you understood all that, an example call for a rectangular domain $x_{min} \leq x \leq x_{max}$, $y_{min} \leq y \leq y_{max}$ with `NIxNJ` points with dirichlet boundary conditions would look like

```
 call hwscrt(xmin,xmax,(ni-1),1,tmp,tmp,
&                ymin,ymax,(nj-1),1,tmp,tmp,
&                0.,rhs,ni,pertrb,ierror,wk)
```

where `wk` and `tmp` are temporary work arrays. In general, these routines (like many useful pieces of freeware) are very hard to understand on a line-by-line level. However they are easy to test because we can always work out a simple test solution by forward substitution. As an example, if we assume our true solution is

$$u(x,y) = \sin\left(\frac{n\pi x}{x_{max}}\right)\sin\left(\frac{m\pi y}{y_{max}}\right) \tag{9.4.11}$$

(which is a $m \times n$ set of shear cells on a rectangular domain that stretches from $0 \leq x \leq x_{max}$, $0 \leq y \leq y_{max}$ see Fig. 9.3) then u is a solution of Eq. (9.1.7) if

$$f(x,y) = -\pi^2[(\frac{n}{x_{max}})^2 + (\frac{m}{y_{max}})^2]\sin\left(\frac{n\pi x}{x_{max}}\right)\sin\left(\frac{m\pi y}{y_{max}}\right) \tag{9.4.12}$$

and we use dirichlet boundaries $u = 0$ on all boundaries. Similar tests for Neumann boundaries can be made by swapping $\cos$ for $\sin$. The problem set will provide a host of methods that apply this test .

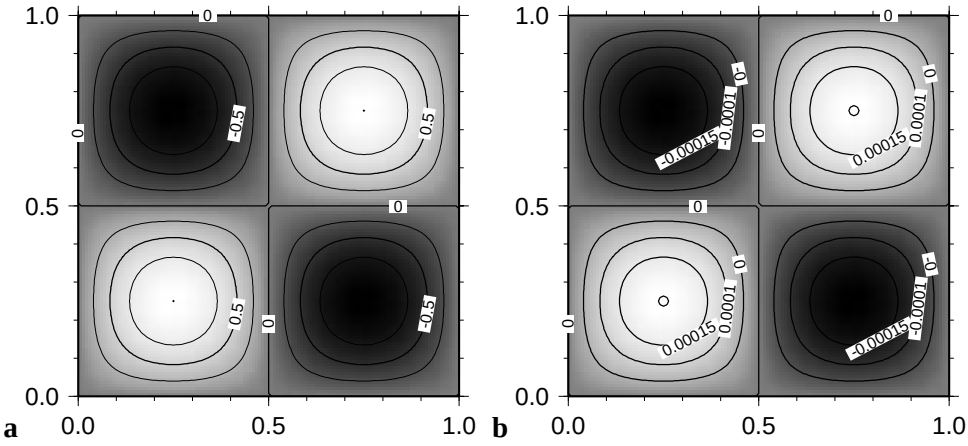


Figure 9.3: example of a 2×2 sin-cell test problem on the unit square 128^2 grid (using a multi-grid method). (*a*) typical solution. *b*) typical errors

9.5 Iterative Methods

In both direct and rapid solutions, we supply a right hand side (and possibly a matrix) and the technique returns the solution. Iterative techniques take a slightly different tack and are actually more related to solving the time dependent problem

$$\frac{\partial u}{\partial t} = \mathcal{L}u - f(\mathbf{x}) \tag{9.5.1}$$

to steady state ($\mathcal{L}$ is an arbitrary elliptic or higher order differential operator). I.e. we start out with some initial guess for our solution u and then iterate in "time" until the solutions stops changing. Because the iterations are effectively like time steps, all the standard techniques for implementing boundary conditions in time dependent problems can be use directly in iterative schemes. The only real trick to iterative methods is to do the problem in much less time than it would take to solve the time-dependent problem.

Another way to look at iterative methods is from the point of view of solving the matrix problem

$$\mathsf{A}\mathbf{u} = \mathbf{f} \tag{9.5.2}$$

Suppose we could split the matrix A into two pieces one that was easy to invert and one that was hard to invert, i.e. $\mathsf{A} = \mathsf{E} + \mathsf{H}$. Then we could rewrite Eq. (9.5.2) as

$$\mathsf{E}\mathbf{u} + \mathsf{H}\mathbf{u} = \mathbf{f} \tag{9.5.3}$$

and form the iterative scheme

$$\mathbf{u}^{n+1} = \mathsf{E}^{-1}\left[\mathbf{f} - \mathsf{H}\mathbf{u}^{n}\right] \tag{9.5.4}$$

where $\mathbf{u}^n$ is our solution at iteration n and $\mathbf{u}^{n+1}$ is our improved guess. The matrix $\mathsf{E}^{-1}\mathsf{H}$ is known as the *iteration matrix* and it must have the property that all of its

eigenvalues must be less than one. The point is that the iteration matrix is trying to reduce the errors in our guess $\mathbf{u}^n$ and those errors can always be decomposed into orthogonal eigenvectors with the property that

$$\mathsf{E}^{-1}\mathsf{H}\mathbf{x} = \lambda\mathbf{x} \tag{9.5.5}$$

where $\mathbf{x}$ is the eigenvector and λ is the eigenvalue. If λ is not less than one, repeated iteration of Eq. (9.5.4) will cause the error to grow and blow up. Even for non-exploding iteration schemes, however, the rate of convergence will be controlled by the largest eigenvalue. Unfortunately for most iteration matrices, the largest eigenvalue approaches 1 as the number of points increase and thus simple schemes tend to converge quite slowly. The amount of decay caused by this largest eigenvalue is called the *spectral radius* ρ_s which goes asymptotically to one as the grid-size is increased (we will see why shortly).

Before we go on to illustrating some of these issues with the simplest (and not very good) classical iteration schemes, it is worth rewriting Eq. (9.5.4) in a slightly different form. If we simply add and subtract $\mathsf{E}\mathbf{u}^n$ within the brackets of the right hand side. i.e.

$$\mathbf{u}^{n+1} = \mathsf{E}^{-1}\left[\mathbf{f} - (\mathsf{E}+\mathsf{H})\mathbf{u}^n + \mathsf{E}\mathbf{u}^n\right] \tag{9.5.6}$$

then we can show that Eq. (9.5.4) is equivalent to

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \mathsf{E}^{-1}\mathbf{r} \tag{9.5.7}$$

where

$$\mathbf{r} = \mathbf{f} - \mathsf{A}\mathbf{u}^n \tag{9.5.8}$$

is the *residual*, i.e. the difference between the right-hand side and the right-hand side that would produce our initial guess $\mathbf{u}^n$. When (and if) we have converged $\mathbf{r} \to 0$ and $\mathbf{u}^{n+1} \to \mathbf{u}^n$. Thus another way to look at iterative methods is that we start with a guess, calculate the residual and try to feed it back into our guess in a way that makes the residual smaller.

9.5.1 Classical Methods: Jacobi, Gauss-Seidel and SOR

Clear as mud? Well let's illustrate these concepts with the simplest iterative scheme called a *Jacobi* scheme. This is a very old and very bad scheme but it illustrates all the points and in conjunction with multi-grid methods can actually be a very powerful technique.

In the Jacobi scheme we split our matrix A as

$$\mathsf{A} = [\mathsf{D} + \mathsf{L} + \mathsf{U}] \tag{9.5.9}$$

where D is the diagonal of the matrix and L and U are the lower and upper triangular sections of the matrix respectively (see Fig. 9.1). Now the diagonal has the trivial inverse $\mathsf{D}^{-1} = 1/D_{ij}$, thus the Jacobi scheme can be written generally as

$$\mathbf{u}^{n+1} = \mathsf{D}^{-1}\left[\mathbf{f} - (\mathsf{L}+\mathsf{U})\mathbf{u}^n\right] \tag{9.5.10}$$

or for the simple regular Poisson stencil

$$A_{ij} = \frac{1}{\Delta^2}\begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix} \tag{9.5.11}$$

we can write the Jacobi scheme in stencil notation as

$$u_{ij}^{n+1} = -\frac{1}{4}\left[\Delta^2 f_{ij} - \begin{bmatrix} & 1 & \\ 1 & 0 & 1 \\ & 1 & \end{bmatrix} u_{ij}^n\right] \tag{9.5.12}$$

If that looks vaguely familiar it should because we have actually seen it before in our time dependent problems. A more physical way to get at (and understand) the Jacobi scheme is to start from the time dependent form of the diffusion equation

$$\frac{\partial u}{\partial t} = \nabla^2 u - f \tag{9.5.13}$$

and simply perform a FTCS differencing on it to produce

$$u_{ij}^{n+1} = -\beta\left[\Delta^2 f_{ij} - \begin{bmatrix} & 1 & \\ 1 & (1/\beta - 4) & 1 \\ & 1 & \end{bmatrix} u_{ij}^n\right] \tag{9.5.14}$$

where $\beta = \Delta t/\Delta x^2$ is the diffusion parameter (see Section 8, Eq. (8.5.6)). Thus the Jacobi scheme is identical to taking the maximum FTCS step allowable with $\beta = 1/4$, and that's the big problem. If you remember from the discussion on explicit diffusion schemes, then you will recall that β is effectively the time scale for the highest frequency components to decay. However, in general the amount of decay an arbitrary wavelength d will experience in one time step (for a scaled diffusivity $\kappa = 1$) is approximately

$$\rho_s \simeq \exp\left[-\frac{4\pi^2\Delta t}{d^2}\right] \tag{9.5.15}$$

so if our grid has $N \times N$ points, our longest wavelength is approximately $d = N\Delta x$ and taking the maximum time step $\Delta t = \Delta x^2/4$ shows that within one time step the amount of decay we can expect for our longest wavelength error is about

$$\rho_s \simeq \exp\left[-\frac{\pi^2}{N^2}\right] \tag{9.5.16}$$

which for large N is

$$\rho_s \sim 1 - \frac{\pi^2}{N^2} \tag{9.5.17}$$

(the actual spectral radius for the Jacobi step is really $\rho_s = 1 - \pi^2/(2N^2)$) thus the physical meaning of the spectral radius is the amount of decay that our longest

wavelength error component will decay in one iteration. The amount of decay we can expect in J iterations is therefore

$$\rho_j \simeq \exp\left[-\frac{\pi^2 J}{N^2}\right] \tag{9.5.18}$$

and thus to reduce the error by a factor of e^{-1} with this scheme will require on order N^2 iterations. This is not good.

Can we do better? Well there are some additional simple schemes which are worth knowing although they are not particularly good either. The first variation on this theme are called *Gauss-Seidel* schemes which are easier to see in Fortran than in matrix notation. The Jacobi scheme in Fortran looks like

```
loop over i and j
    unew(i,j)=-.25*(f(i,j)-(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)))
end loop
```

where we do all the smoothing into a new array. The simplest Gauss-Seidel scheme could be written

```
loop over i and j
     u(i,j)=-.25*(f(i,j)-(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1)))
end loop
```

i.e. we just do the smoothing in place using the new values of u as they come up. What do we gain from this? Not a whole lot but the spectral radius of a Gauss-Seidel scheme is the square of that for the Jacobi scheme so it converges in about a factor of 2 less time (still no good). The other advantage is that the Gauss-Seidel scheme can remove certain frequencies (like a checkerboard pattern of errors) that are invisible to the Jacobi scheme. Speaking of checkerboards, there is another variant of Gauss-Seidel called *red-black Gauss-Seidel* where instead of looping over all the points, we notice from the above algorithm that if we colored our grid points red and black like a checkerboard, the red points only depend on the black ones and vice-versa. So we could update all the red ones and then all the black ones which gains us about another factor of 2 and produces less biased errors. We will use the red-black schemes quite a bit.

Okay, even with the Gauss-Seidel schemes, these simplest iterative schemes are still impractical because they require N^2 iterations which makes these schemes order N_{tot}^2 schemes where $N_{tot} = N^2$ is the total number of points on the grid. Clearly for even moderately sized grids this is a disaster. The only way you can do better than this on a simple iterative scheme is to use *Successive Over Relaxation* or SOR. SOR is based on the notion that we can write either the Red-Black Gauss-Seidel or the Jacobi scheme in residual form as

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \mathsf{D}^{-1}\mathbf{r} \tag{9.5.19}$$

or in component notation as

$$u_{ij} = u_{ij} + r_{i,j}/D_{ij} \tag{9.5.20}$$

now we could generalize this by including a relaxation parameter ω such that

$$u_{ij} = u_{ij} + \omega r_{i,j}/D_{ij} \tag{9.5.21}$$

where $0 < \omega < 2$; i.e. we add in a variable amount of the residual (and hope that it improves the rate of convergence). If $\omega < 1$ its called *underelaxation*; for $1 < \omega < 2$ it's *overrelaxation.* For $\omega > 2$ it blows up. It can be shown that only overrelaxation can produce better convergence than Gauss-Seidel and then only for an optimal value of ω such that

$$\omega = \frac{2}{1 + \sqrt{1 - \rho^2_{jacobi}}} \tag{9.5.22}$$

with this value, the spectral radius of SOR is

$$\rho_{SOR} \sim 1 - \frac{2\pi}{N} \tag{9.5.23}$$

for large N. This is a big improvement, but not really big enough because the solution time now scales as $N^{1.5}_{tot}$. A perfect scheme would have the solution time increase linearly with N_{tot} but for this we need multi-grid. Figure 9.4 shows that SOR does converge in about N iterations, however, it does so in a rather peculiar way where the residual increases dramatically before decreasing. By the way, SOR only works this well for a narrow region around the optimal value of ω. However, for more general problems than Poisson equations, finding the optimal value of ω can be problematic, particularly if the operator changes with time.

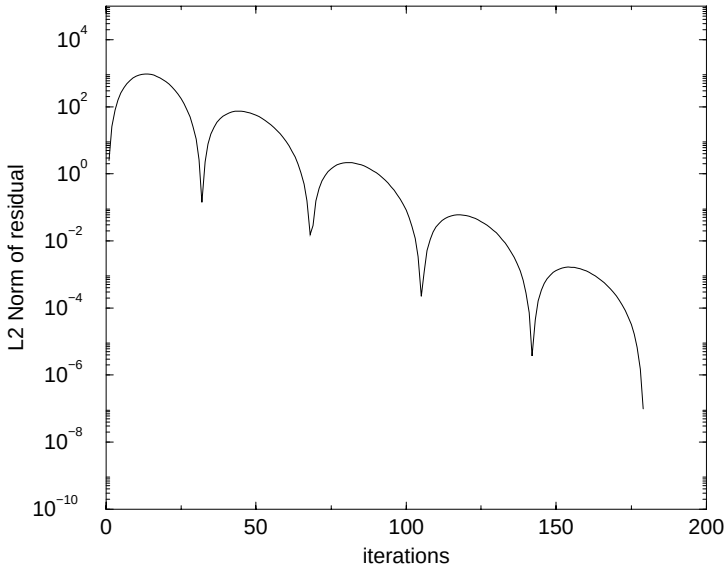


Figure 9.4: Convergence behaviour of optimal SOR (with chebyshev acceleration and red-black ordering) for a 128×128 square grid. Note that the residual increases wildly before decaying and still requires of order N iterations to converge.

9.5.2 Multigrid Methods

Multigrid techniques start from the realization that the simple iterative schemes of the previous section are not actually very good relaxers; however, they are very good smoothers, i.e. while they cannot reduce errors at all frequencies equally, they are very efficient at smoothing out the highest frequency errors. Moreover we note that the "highest frequencies" in our problem are defined only by the grid spacing. Thus it ought to be possible to use these simple smoothing schemes on a hierarchy of grids with different spacings to remove errors at all frequencies. This is the essence of multi-grid and the rest of this chapter will show how we can actually implement this idea to produce iterative schemes that are as efficient as the best FACR scheme for both constant and variable coefficient matrices and even non-linear problems.

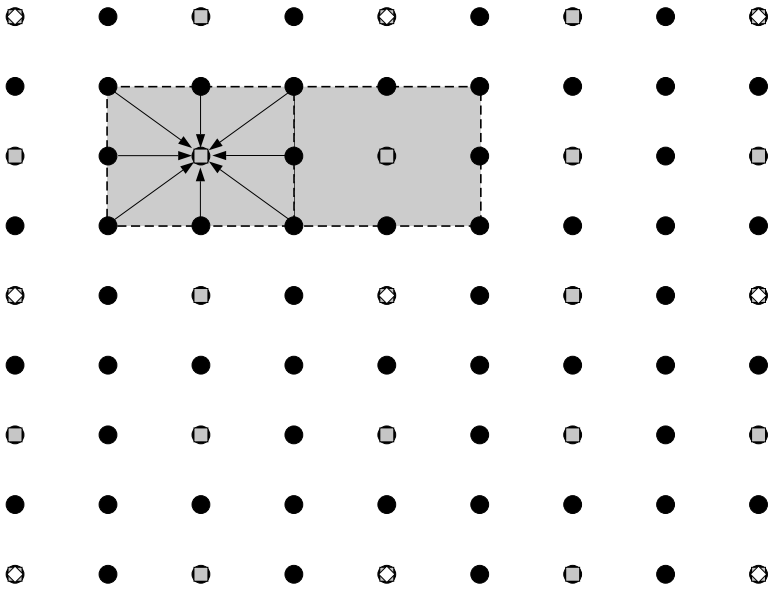


Figure 9.5: An example of a 3 level nested multi-level grid (or just a multi-grid). The *fine grid* or *level 1* is shown by the dots and has 9×9 grid points. Level 2 is shown by the grey squares and has been coarsened by a factor of 2 to have only 5×5 points. The coarsest grid is level 3 (diamonds) with only 3×3 points. Note that in 2-D each coarser grid has only 1/4 of the points of the next finer grid. The grey boxes show a restriction operation from level 1 to level 2 where we set the value of the coarse grid point equal to the average value of the control volume defined on the fine grid. Interpolation takes information from coarse grids to fine grids

Before we discuss algorithms, we first need to define what we mean by a *multi-level* grid. Figure 9.5 shows a 3-level grid where each coarser grid has a grid spacing that is twice that of the next finer grid (and therefore as ~ 4 times less points on each level). Note that all the grids share the same boundaries and the number of grid points in both directions must be repeatedly divisible by two. While we can't have completely arbitrary numbers of grid points in each direction, the restrictions are less severe than simple spectral methods which usually require power

of 2 grids. In general to guarantee the properties of the multi-grid we calculate the number of grid points in each direction by first specifying the *aspect ratio* of the coarsest grid in *grid cells* (i.e. $n_{ci} \times n_{cj}$) then the number of total levels in the grid n_g. Given these three numbers, the number of grid points in the fine grid is given by

$$N_i = n_{ci} 2^{(n_g-1)} + 1 \tag{9.5.24}$$

$$N_j = n_{cj} 2^{(n_g-1)} + 1 \tag{9.5.25}$$

For the example shown in Fig. 9.5 $n_{ci} = n_{cj} = 2$, $n_g = 3$ and therefore $N_i = 9$ and $N_j = 9$. Once the number of grid points on any level is known, the number of grid-points in the next coarser level is simply $N_i^{l+1} = N_i^l/2 + 1$ etc. where integer math is assumed (we will also need to know how to relate coordinates between grids but we will get to that shortly).

Given the basic definition of the multi-level grid we'll start by just understanding the algorithm for a two-level problem and then show that you can create all other schemes by recursive calls to the two level scheme. Consider that we want to solve some linear elliptic problem

$$\mathcal{L}\mathbf{u} = \mathbf{f} \tag{9.5.26}$$

which is approximated by the discrete matrix problem

$$\mathsf{A}^h \mathbf{u}^h = \mathbf{f}^h \tag{9.5.27}$$

on a grid with grid spacing h (this is the fine grid, the next coarser grid on level 2 has grid spacing $2h$). Now let's apply a simple relaxation scheme like red-black gauss Seidel to our initial guess for just a few (e.g. 2–3) relaxation sweeps to give us an improved solution $\tilde{\mathbf{u}}^h$. $\tilde{\mathbf{u}}^h$ will still have a large amount of long wavelength error, but the high-frequency components will have been smoothed out[5]. We can now write our true discrete solution solution as

$$\mathbf{u}^h = \tilde{\mathbf{u}}^h + \mathbf{e}^h \tag{9.5.28}$$

where $\mathbf{e}^h$ is the *correction* that we would need to add to our current guess to get the correct solution[6]. Substituting Eq. (9.5.28) into (9.5.27) and using the fact that A^h is a linear operator, we can rewrite (9.5.27) as

$$\mathsf{A}^h \mathbf{e}^h = \mathbf{r}^h \tag{9.5.29}$$

where $\mathbf{r}^h = \mathbf{f}^h - \mathsf{A}^h \tilde{\mathbf{u}}^h$ is the residual. Equation (9.5.29) shows that for linear problems, one can solve for either the solution or the correction interchangeably (in multi-grid we will usually be solving for the correction). The point is that if could solve (9.5.29) for the correction, then we could add it back to our improved guess

[5] While you have a lot of latitude in the relaxation scheme you use in multi-grid, don't use SOR because the over-relaxation destroys the smoothing behaviour of the relaxation and will cause things to blow up.

[6] the correction **e** is also often called the *error* in the solution.

and we would be done. Of course it is not any easier to solve for the correction than for the solution on the fine grid. However, we have already pre-smoothed our guess on the fine grid so the residual (and therefore the correction) should not contain any fine-grid scale errors. Thus if we could solve for the correction on a coarser grid we would not be losing any information but we would be gaining an enormous amount of time given the factor of 4 reduction in points at the next grid level. More specifically, we want to solve

$$\mathsf{A}^{2h}\mathbf{e}^{2h} = \mathbf{r}^{2h} \tag{9.5.30}$$

where A^{2h} is our operator defined on the next coarser grid and

$$\mathbf{r}^{2h} = \mathcal{R}_h^{2h}\mathbf{r}^h \tag{9.5.31}$$

is the *restriction* of the fine grid residual onto the coarse grid. $\mathcal{R}_h^{2h}$ is called the restriction operator and is some mechanism for transferring information from a fine grid to one coarser grid. There are many ways to do restriction but a standard approach is to simply use the average value of the control volume defined on the coarse grid (see Fig. 9.5). Thus for every point on the coarse grid (ic, jc) we can define the corresponding fine grid coordinates

$$if = 2ic + 1 \tag{9.5.32}$$
$$jf = 2jc + 1 \tag{9.5.33}$$

and the *full weighting restriction* for an interior point can be written in stencil form as

$$r_{ic,jc}^{2h} = \begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{bmatrix} r_{if,jf}^{h} \tag{9.5.34}$$

where the stencil operates on the fine grid but is only evaluated for the number of points on the coarse grid. As for defining the coarse grid operator, this is probably the most difficult problem in multi-grid methods, however, the standard heuristic is to just define the operator as if you were simply differencing the problem on the coarse grid. Only start getting clever when that doesn't work. In the case of a Poisson problem the operator is just

$$A_{ic,jc}^{2h} = \frac{1}{(2h)^2}\begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix} \tag{9.5.35}$$

Given the operator and the right hand side, we could now solve the matrix problem Eq. (9.5.30) using any of the solvers we have already discussed but for much less computational cost on the coarse grid.

Just for argument sake, let's pretend that we can solve Eq. (9.5.30) exactly for the correction $\mathbf{e}^{2h}$ on the coarse grid. Again, since $\mathbf{e}^{2h}$ shouldn't have any high frequency components it can be related to the correction on the fine grid by *interpolation*, i.e.

$$\mathbf{e}^h = \mathcal{I}_{2h}^h\mathbf{e}^{2h} \tag{9.5.36}$$

where $\mathcal{I}_{2h}^{h}$ is the *interpolation* or *projection operator* that moves information from the coarse grid to the fine grid. The simplest interpolation operator is just bilinear interpolation where fine-grid points that are coincident with coarse grid points are set to the coarse value, fine-grid points that lie on coarse grid line are just half their nearest coarse neighbors and fine points that are in the center of coarse cells are just $1/4$ of the corner points.

Finally, given our new solution for the correction on the fine grid we update our initial guess using Eq. (9.5.28). If the restriction, and interpolation processes added no high frequency error to our solution then we would be done; however, in general some error is always introduced moving between grids and the final step is to relax a few times starting with our improved guess to remove any high-frequency errors that have been introduced. To recap, the two-level correction scheme is

- Given an initial guess, **relax** on the fine grid `NPRE` times for $\tilde{\mathbf{u}}^h$
- Form the fine grid **residual** $\mathbf{r}^h = \mathbf{f}^h - \mathsf{A}^h\tilde{\mathbf{u}}^h$
- **restrict** the residual to the coarse grid and form the new right hand side $\mathbf{r}^{2h} = \mathcal{R}_h^{2h}\mathbf{r}^h$
- **solve** $\mathsf{A}^{2h}\mathbf{e}^{2h} = \mathbf{r}^{2h}$ for $\mathbf{e}^{2h}$
- **interpolate** the correction to the fine grid via $\mathbf{e}^h = \mathcal{I}_{2h}^h\mathbf{e}^{2h}$
- **correct** $\tilde{\mathbf{u}}^h$ to form $\mathbf{u}^h = \tilde{\mathbf{u}}^h + \mathbf{e}^h$
- **relax** the new solution `NPOST` times

This two-level scheme can be repeated until convergence however it can also be applied recursively. Right now we have assumed that we have some technique (e.g. a direct solver) to solve exactly for the coarse grid correction. However, if the coarse grid is still fairly large it will also contain long-wavelength errors that will be costly to remove in a single pass. An alternative is to just continue to repeat the process iteratively. i.e. rather than solve Eq. (9.5.30) exactly for $\mathbf{e}^{2h}$ we can apply our simple relaxation scheme a few times on the coarse grid to remove order $2h$ wavelength errors. Our new guess will still have longer wavelength errors so we need to do a coarse grid correction for the coarse grid correction (crystal clear eh? wait a bit and all will be revealed). We repeat the process on increasingly coarser grids, each time just removing the errors that are comparable to the grid spacing until we reach a sufficiently coarse level so that it really is cheap to solve exactly for the correction (to the correction to the correction to the $\ldots$). We then move back down again correcting each level. This particular iterative scheme is called a *V-cycle* and is illustrated schematically in Fig. 9.6. In pseudo-code we can define a V-cycle algorithm like

```
do levels from 1 to Ngrids-1  ! go up the V
   relax(npre times)
   find_residual(level)
   restrict(residual(level) to rhs(level+1))
enddo

solve correction on coarsest level
```

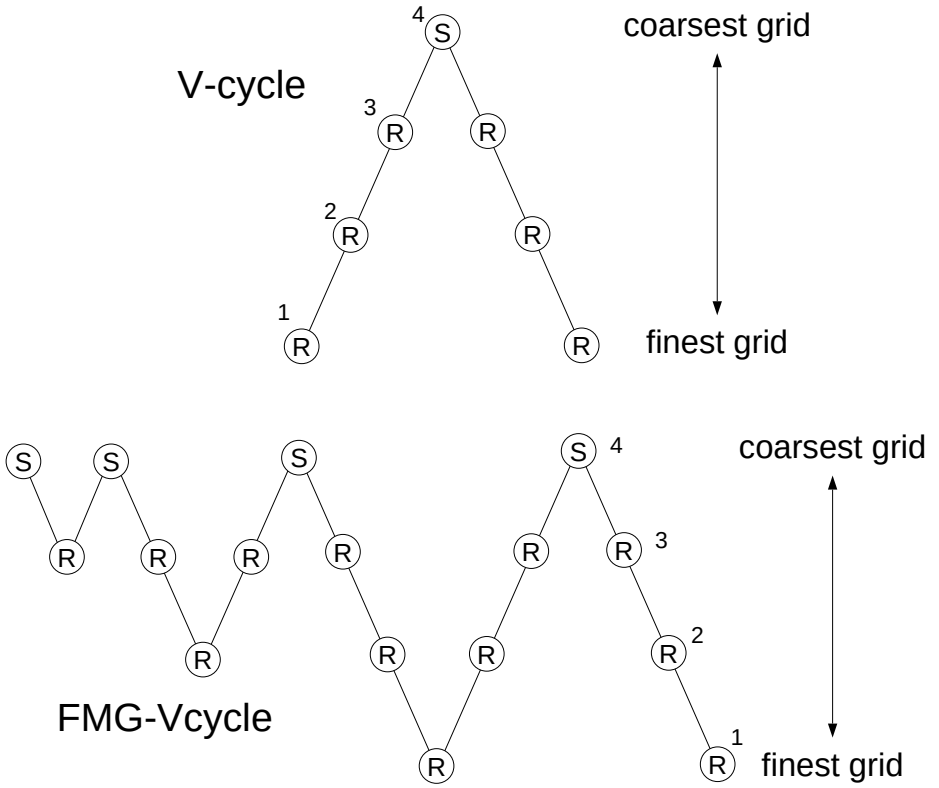


Figure 9.6: schematic illustrations of V-cycle and Full Multi-grid FMG-cycle schemes for a four level grid. Level 1 is the fine grid, level 4 is the coarse grid. Circles with R's denote relaxation (smoothing) steps, Circles with S's denotes exact solution on the coarse grid. Ascending lines denote restriction (fine-to-coarse) operations. Descending lines denote interpolation (coarse-to-fine) operations. In general a V-cycle is used when you have a good initial guess for your solution. Each V-cycle should reduce the fine-grid residual by about an order of magnitude. FMG-cycles (which are actually nested V-Cycles) are used when an initial guess is unknown. Properly coded, a FMG-cycle can behave like an order N direct solver.

```
do levels from Ngrids-1,1,-1 ! go down the V
    interpolate(correction(level+1) to correction(level))
    add(correction(level)+solution(level))
    relax(npost times)
enddo
```

Because you can implement different number of relaxations in the V-Cycle we will often specify them in the name of the V-cycle. For example, a useful scheme for Poisson problems is to use a (2,1) V-cycle where `NPRE=2` and `NPOST=1`. The V-cycle can then be called successively monitoring the norm of the fine-grid residual after every V-cycle. When the residual has been reduced to some predetermined tolerance, we can stop. Properly implemented, a standard V-cycle can reduce the norm of the residual by an order of magnitude per V-cycle, with only about as much

work as it takes to smooth the finest grid (NPRE+NPOST+3) times. Moreover, this convergence rate is independent of the size of the problem (because all frequencies are reduced efficiently) and the solution time scales only as the number of fine-grid points. Truly amazing.

In the next section we will show how to specifically implement this algorithm. Before we do that, however it is also useful to mention another cycling scheme called *Full Multigrid* or *nested iteration*. In the V-cycle, we begin on the finest grid with what we hope is a reasonable first guess for our solution. If we are doing a time dependent problem, we could use the solution at the last time step which is often very close. However, if we really don't have a good guess for the fine grid solution, we can use FMG. The FMG scheme is actually a set of nested V-cycles of increasingly more levels. First we need to restrict the right-hand side f to all the levels and then, instead of starting on the fine grid we start on the coarse grid and *solve* for $\mathbf{u}^H = \mathsf{A}^{-1^H}\mathbf{f}^H$. Given this coarse grid solution, we interpolate it to one finer grid and use it as the first guess of a 2 level V-cycle. Given this improved guess we interpolate it down again to use as the initial guess of a 3-level V-cycle etc. until we reach the finest grid. The FMG-cycle is shown schematically in Fig. 9.6 and can be represented in pseudo-code as

```
do levels from 1 to Ngrids-1 ! first restrict f
   restrict(rhs(level) to rhs(level+1))
enddo

solve for u(ngrid)

do levels=Ngrids-1 to 1 ! nested vcycles
   interpolate(u(level+1) to u(level))
   vcycle(u(level),from level to Ngrids)
enddo
```

9.5.3 Practical computation issues: how to actually do it

The previous discussion is lovely and abstract however it doesn't really tell you how to write your own multi-grid solver. Actually, I will be giving you a working 2-D multi-grid solver with a fair numbers of bells and whistles but this section tells you how it works. Given a few initial pointers (particularly on storage), multi-grid schemes are not particularly hard to write because the individual pieces (relax, resid, restrict, solve, interpolate), are quite modular, represented by simple stencil operations and at most only have to deal with two-grids at a time. The most difficult part of these schemes is getting the storage right. Numerical Recipes (which is usually phenomenal) really screws this one up. The scheme I'll use is discussed in Briggs [1] and makes rather elegant use of all the features of Fortran array handling that I have been emphasizing.

The first issue is how to store all of these grids of different sizes in an efficient and orderly manner. As usual we will store the entire multi-level grid in one composite 1-D array; however, because the different grid levels have different numbers of grid points, the best way to pack them into memory is to use *index pointers*. Remember that in Fortran, all arrays are passed by address and it is possible to pass 1-D arrays to subroutines that treat them like $n-D$ arrays. Thus in Fortran it is legal to have a call like

```
real array(100)
integer ni,nj
ni=5
nj=5
call do_something(array(11),ni,nj)
...
subroutine do_something(arr,ni,nj)
integer ni,nj
real arr(ni,nj)
...
```

which will operate on the 25 consecutive points *starting at* `array(11)` (i.e. it will work on the memory in the subarray `array(11:35)` and treat it like a 2-D array within the subroutine). How convenient. Thus to store an entire 4-level grid in a single 1-D array, we need to calculate the offset indices (or index pointers) so that `ip(1)` is the index of the start of the fine grid, `ip(2)` points to the beginning of the level 2 grid etc up to `ip(ngrids)`. Here's a little subroutine that does just that

```
c**********************************************************************
c     subroutine to calculate index pointers for composite arrays
c     ip: is array of pointers
c     ng: total number of grid levels
c     ni,nj: number of grid points in i and j direction on finest grid
c     len: total length of composite array
c **********************************************************************

      subroutine initpointer(ip,ng,ni,nj,len)
      implicit none
      integer ng,ni,nj,len
      integer ip(ng)

      integer n,nni,nnj

      ip(1)=1
      nni=ni
      nnj=nj
      do n=2,ng                  ! increasing coarseness
         ip(n)=ip(n-1)+nni*nnj
         nnj=nnj/2+1
         nni=nni/2+1
      enddo
      len=ip(ng)-1+nni*nnj
      return
      end
```

Given `ip(1:ng)` and `len`, the storage scheme of Briggs needs 3 composite arrays of length `len`: `u(len)` to hold the solution (and corrections), `rhs(len)` to hold the the right-hand sides, and `res(len)` and the residual and interpolations. In addition we may also need to pass in a variable coefficient operator which in the most general case could be a full 5-point stencil operator `aop(5,len)`. These four arrays and their storage layout are shown in Figure 9.7 for a 4-level grid

With this storage scheme it is straightforward to efficiently implement a vcycle in Fortran. Here is an example for a simple version with hard-wired boundary conditions

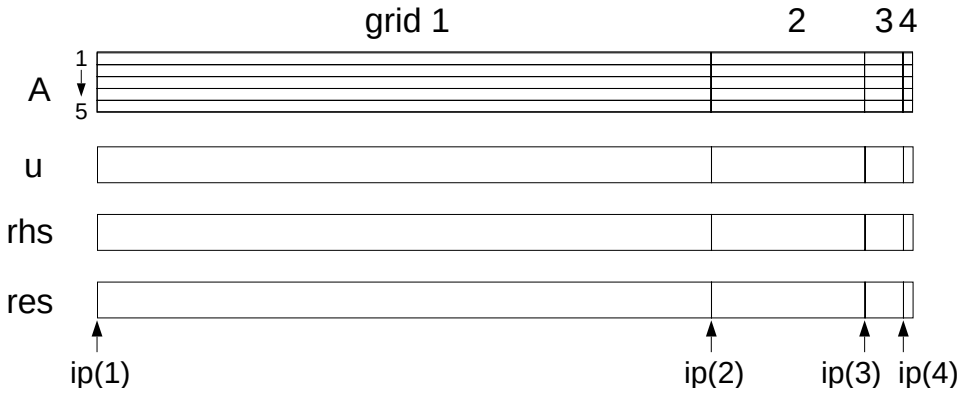


Figure 9.7: Multi-grid storage scheme ala Briggs [1] illustrated for a 4 level grid. All the composite grids for the operator, solutions, right-hand side, and residual are stored as 1-D arrays with index pointers `ip(1:ng)`. `u(ip(1))` is the beginning of the fine grid, `u(ip(2))` is the beginning of the next coarser grid (which has $\sim 1/4$ the points in 2-D), up to `u(ip(ng))` which is the coarsest grid. In general he total length of the composite grid will always be less than $2^D/(2^D-1)$ times the size of the fine grid (For series fans, $\sum_{n=0}^{\infty} 1/2^{nD} = 2^D/(2^D-1)$) Thus in 2-D the total length is less than $4/3N_{fine}$ and in 3-D it's less than $8/7N_{fine}$.

```
c**********************************************************************
c subroutine mg2d_vcyc
c one V cycle, of height ng
c NB: vcyc has no knowledge of the total number of grids, thus
c     ip must be passed such that cmp(ip(1)) is the beginning
c     of the solution array at height'th grid!!!!
c     Also, dz must be dz for the height'th grid!!!!
c   important variables
c  aa:  composite array for  5 point stencil operator
c  uu:  composite array for solution and corrections
c  rhs: rhs
c  res: composite array for residual and interpolants
c  ip:  1-D,array of pointers to sub grids in uu,rhs,res
c  len:   length of composite grids,uu,rhs etc.
c  ng: maximum number of grids in vcycle
c  ni:  number of horizontal points in finest grid
c  nj:  number of vertical points in finest grid
c  ncycle:  number of vcycles before convergence check
c  npre: number of coarsening relaxations
c  npos: number of fining relaxations
c  nsolve:  number of relaxations at coarsets grid
c  dz:  true grid spacing on finest grid
c**********************************************************************
      subroutine mg2d_vcyc(aa,uu,rhs,res,ip,len,ng,ni,nj,ncycle,npre,npos,nsolve)

      implicit none

      integer len,ng
      real aa(5,ng)
      real uu(len)
      real rhs(len)
      real res(len)
      integer ip(ng),step,ncycle,npre,npos,pre,pos
      integer nni,nnj,cycle,nsolve,solve
      integer ni,nj

      do cycle=1,ncycle         ! the number of times to trace the 'V'
         nni=ni
         nnj=nj
```

```
         call arrfill0(uu(ip(2)),len-nni*nnj) ! zero out the correction
c                                              ! on the coarse grids
c-----Fine to coarse leg of the V
c
         do step=1,ng-1
            call mg2d_relax(npre,aa(1,step),uu(ip(step)),rhs(ip(step)),nni,nnj)
            call
            mg2d_resid(aa(1,step),res(ip(step)),uu(ip(step)),rhs(ip(step)),nni,nnj)
            nni=nni/2+1
            nnj=nnj/2+1
            call mg2d_rstrct(rhs(ip(step+1)),res(ip(step)),nni,nnj))
            call arrmult(rhs(ip(step+1)),nni*nnj,4.) ! why is this here?
         enddo
c
c-----Solve on coarsest grid by just relaxing nsolve times
c
         call mg2d_relax(nsolve,aa(1,ng),uu(ip(ng)),rhs(ip(ng)),nni,nnj)
c
c-----Coarse to fine leg the V
c
         do step=ng-1,1,-1
            nni=2*nni-1
            nnj=2*nnj-1
            call mg2d_addint(uu(ip(step)),uu(ip(step+1)),res(ip(step)),nni,nnj)
            call mg2d_relax(npos,aa(1,step),uu(ip(step)),rhs(ip(step)),nni,nnj)
         enddo                  !step (reached ng)
      enddo                     !cycle (number of Vs)
      return
      end
```

Note that while all the storage is done in 1-D arrays, the actual stencil operations are implemented in subroutines that use 2-D arrays. As an example, here is the code for a red-black gauss-Seidel relaxation scheme on any level using dirichlet boundary conditions.

```
c*********************************************************************
c     SUBROUTINE mg2d_relax(nits,a,u,rhs,ni,nj)
c non-vectorised red black Gauss-Seidel relaxation
c              for a general constant 5 point stencil. Dirichlet
c              Boundary conditions
c  Variables:
c               nits: number of red-black iterations
c               a: 5 point operator stencil
c  u, rhs:  solution and right hand side (assumes premult by h^2)
c  ni,nj: horizontal and vertical dimension of u, rhs
c*********************************************************************
      subroutine mg2d_relax(nits,a,u,rhs,ni,nj)
      implicit none
      integer ni,nj,nits
      real a(5),rhs(ni,nj),u(ni,nj)
      integer i,ipass,j
      integer im,ip,jm,jp,rb,m,n

      rb(m)=m-2*((m/2))         ! red-black toggle


c     do red-black gauss-seidel relaxation for nits
c
      do n=1,nits
         do ipass=0,1           !do red then black
            do j=2,nj-1
               jm=j-1
               jp=j+1
               do  i=2+rb(ipass+1+j),ni-12
                  im=i-1
                  ip=i+1
                  u(i,j)=a(1)*(rhs(i,j)-(a(2)*u(im,j)+a(3)*u(ip,j)+a(4)*u(i,jm)+a(5)*u(i,jp)))
               enddo
```

```
          enddo
        enddo
      enddo
      return
      end
```

Figures 9.8 and 9.9 illustrate schematically how the storage scheme and local grid operators work in implementing a single vcycle on a 4-level grid. This scheme is efficient (and rather elegant) in use of storage etc. Additional composite arrays can be added easily.

9.5.4 A note on Boundary conditions in multi-grid

Boundary conditions are the curse of numerics but they are also often the most important part. For multi-level schemes implementation of boundary conditions may seem particularly confusing (and is often neglected in discussions of techniques) because boundary conditions must be enforced on all of the different levels as well as for the restriction operator.[7] Personally, I've been through about 4 different ways of implementing boundary conditions but I think I have a fairly stable, easy to implement approach that will handle at least the most standard boundary conditions. Because iterative schemes are identical in operations to time-dependent problems, these boundary schemes can also be used in time-dependent update schemes. The basic idea is that when we are on an edge we have to specify what to do with the point that lies outside the grid. For example, if we want a reflection boundary on the left edge `i=1` then if our stencil operation would want `im=i-1=0` we would replace it with `im=2`. Likewise, for periodic boundaries we would set `im=ni-1` and for dirichlet boundaries we would skip over the `i=1` edge altogether and start with `i=2`. More generally, a standard 5-point star operation looks like

```
u(i,j)=a(1)*(rhs(i,j)-(a(2)*u(im,j)+a(3)*u(ip,j)+a(4)*u(i,jm)+a(5)*u(i,jp)))
```

where for an interior point `im=i-1`, `ip=i+1`, `jm=j-1`, `jp=j+1`. So all we really need to do is redefine `im,ip,jm,jp` appropriately when we are on an edge or corner. To do this in a general way, I pass in a small array `iout(2,2,ngrids)` which gives the value of the outside index for each of the sides and directions. The storage is
`iout(plus_minus,direction,grid)` where `direction=1` is the `i` direction of the grid `direction=2` is the `j` direction (and so on for `k` in 3-D). and `plus_minus=1` is the edge in the `direction-1` direction (oy!) and `plus_minus=2` is the edge in the `direction+1` direction. To make this simpler the left edge on grid `g` is `iout(1,1,g)`, the right edge is `iout(2,1,g)`; bottom is `iout(1,2,g)` and top is `iout(2,2,g)`. If `iout` is negative we assume the edge is dirichlet and we skip the edge. Here is a general relaxation scheme that implements this type of boundary condition information. Study it and compare it to the hard-wired dirichlet relaxer.

```
c**********************************************************************
c     SUBROUTINE mg2d_relaxc(nits,a,u,rhs,ni,nj,iout)
c non-vectorised red black Gauss-Sidel
c  Variables:
c               nits: number of red-black iterations
```

[7] Boundary conditions are not required for interpolation because interpolation on the boundary only requires information on the boundary.

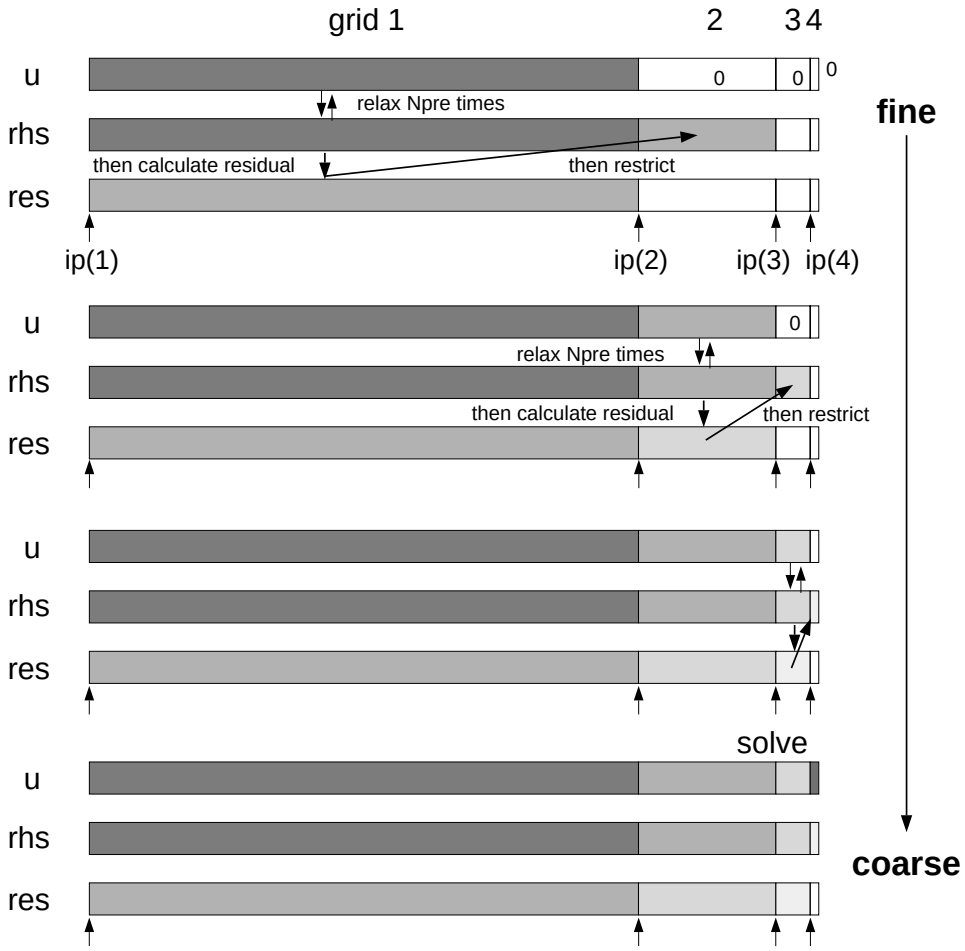


Figure 9.8: Schematic of how vcycle works in the Briggs storage scheme. This figure just shows the coarsening leg of the Vcycle and the coarse-grid solve. At the initial time, `u(ip(1))` holds our initial guess and all the rest of the array is set to zero. At this time only `rhs(ip(1))` contains the right-hand side for the finest grid. After a few relaxation steps, the fine grid residual is calculated in `res(ip(1))` and restricted to `rhs(ip(2))`. The coarse grid correction $\mathbf{e}^{2h}$ is approximated by relaxation and stored in `u(ip(2))` and the process is repeated up to the coarsest grid where the correction to the grid 3 correction is solved exactly (or by just lots of relaxations).

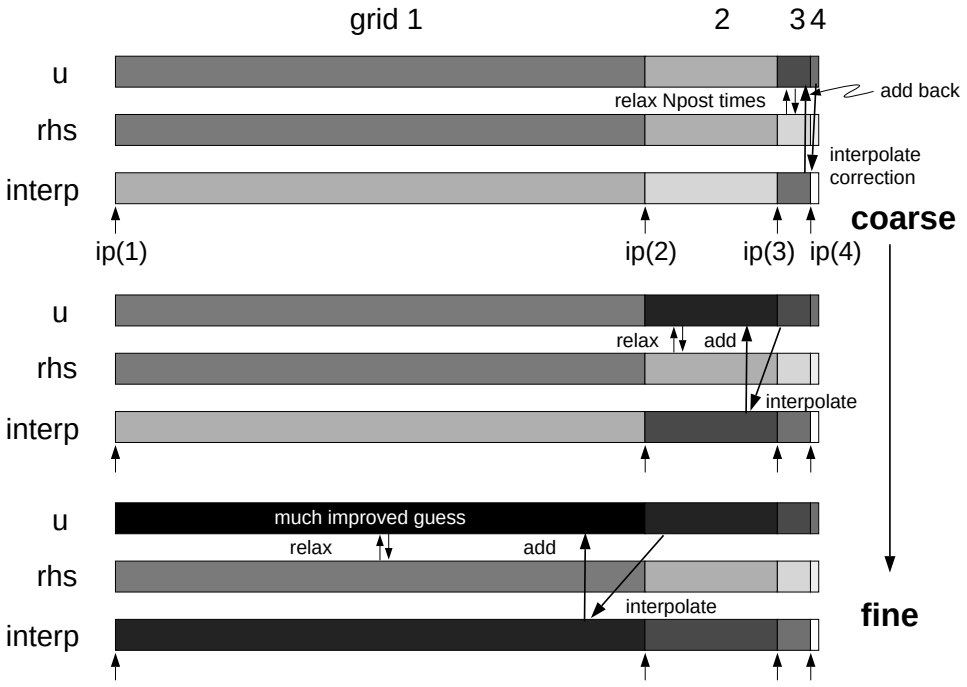


Figure 9.9: Coming back down. This figure shows the fine to coarse leg of the V-cycle. Given the solved coarsest correction in `u(ip(4))`, it is interpolated onto the storage of `res(ip(3))` then added to `u(ip(3))` which is then relaxed against `rhs(ip(3))` which has not been changed from the up-stroke. The processes is repeated until we return to the finest grid where we can repeat the procedure by just zeroing out the corrections on `u(ip(2))` to `u(ip(ng))`. Note that `rhs` is not changed on the down-stroke and that the interpolated correction just overwrites the residual array.

```
c               a: 5 point operator stencil
c  u, rhs:  solution and right hand side (assumes premult by h^2)
c  ni,nj: horizontal and vertical dimension of u, rhs
c  iout:  flags denoting boundary offsets
c*****************************************************************
      subroutine mg2d_relax(nits,a,u,rhs,ni,nj,iout)
      implicit none
      integer ni,nj,nits
      real a(5),rhs(ni,nj),u(ni,nj)
      integer iout(2,2),ioff(2,2)
      integer i,ipass,j
      integer is,ie,js,je
      integer im,ip,jm,jp,rb,m,n

      rb(m)=m-2*((m/2))          ! red-black toggle

      do j=1,2                  !set up dirichlet boundaries if iout<0
         do i=1,2
            if (iout(i,j).lt.0) then
               ioff(i,j)=1
            else
               ioff(i,j)=0
            endif
         enddo
      enddo
      is=1+ioff(1,1)
      ie=ni-ioff(2,1)
      js=1+ioff(1,2)
```

```
      je=nj-ioff(2,2)
c
c----is,ie,js,je are the starting and ending bounds for the grid, is=1
c----if non-dirichlet, otherwise is=2 etc.
c
c
c     do red-black gauss-seidel relaxation for nits
c
      do n=1,nits
         do ipass=0,1           !do red then black
            do j=js,je
               jm=j-1
               jp=j+1
               if (j.eq.1) then
                  jm=iout(1,2)
               elseif (j.eq.nj) then
                  jp=iout(2,2)
               endif
               do  i=is+rb(ipass+1+j),ie,2
                  im=i-1
                  ip=i+1
                  if (i.eq.1) then
                     im=iout(1,1)
                  elseif (i.eq.ni) then
                     ip=iout(2,1)
                  endif
                  u(i,j)=a(1)*(rhs(i,j)-(a(2)*u(im,j)+a(3)*u(ip,j)+a(4)*u(i,jm)+a(5)*u(i,jp)))
               enddo
            enddo
         enddo
      enddo
      return
      end
```

Note, with a good compiler, the `if` statements in the `do` loops should be peeled although this may cause minor problems when `is,ie,js,je` are not specified before hand. Another approach is to use the same scheme but use pre-compilers to conditionally compile in specific boundary conditions for specific jobs. Remember the time you save in run time may not be worth the hassle.

9.6 Summary and timing results

Okay so we've now looked at most of the commonly used schemes for boundary value problems (well not conjugate gradient techniques but they're order $N^{1.5}$ schemes anyway), so it's time to put our money where our CPU's are and compare them against each other in terms of solution time and accuracy. For this problem we will compare a direct solver (Y12M), a FACR scheme from FISHPAK (hwscrt), SOR and a constant stencil multigrid scheme (from your's truly) in both V-cycle mode and FMG mode for the 2-2 $\sin(k_x x)\sin(k_y y)$ cell test shown in Fig. 9.3. To compare behaviour for large problems, these tests have been run on a single node of an IBM SP2. Problems were done for square grids with dirichlet boundary conditions and $N = 65^2, 129^2, 257^2, 513^2, 1025^2$ points. Results are shown in Fig. 9.10 and show that for Poisson problems FACR and multi-grid are comparable in timing, scaling and errors. The generally larger times for plain V-cycle Multigrid are also somewhat misleading because they are trying to solve the problem from an initial guess $\mathbf{u} = 0$ and thus require about 5 V-cycles to converge. If this solution were part of a time dependent problem and we had a much better initial

guess, we could probably solve it in 1–2 V-cycles and the solution time would be proportionally smaller by a factor of 2 to 5.

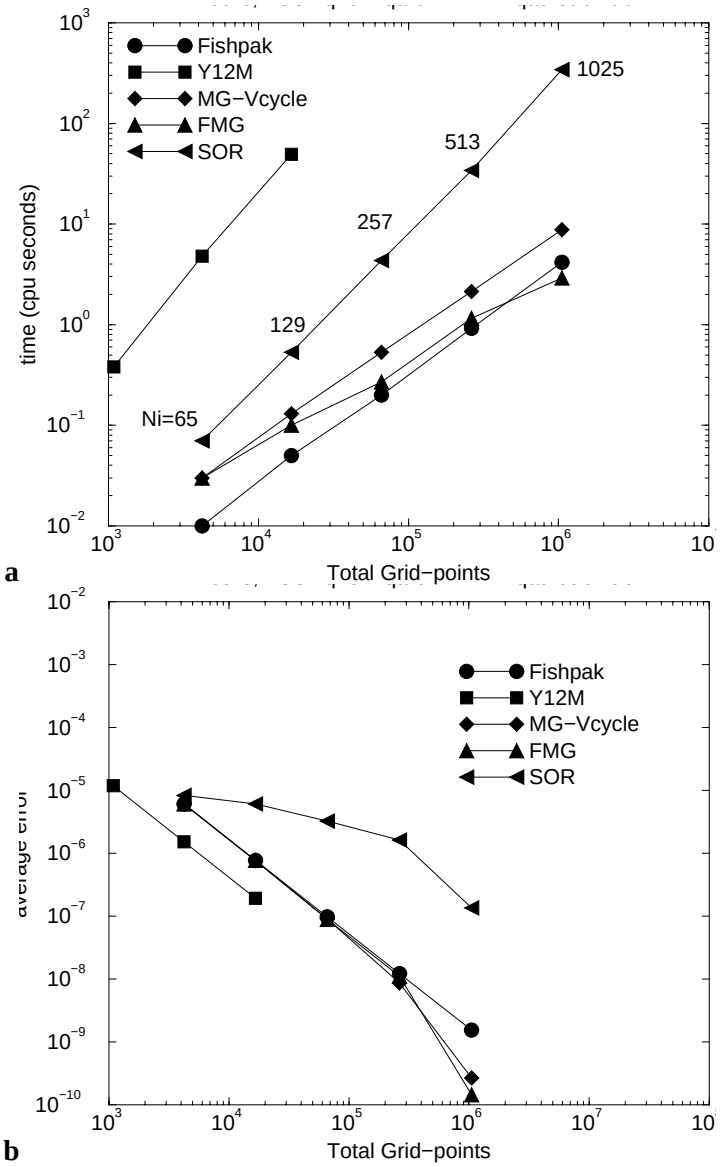


Figure 9.10: Comparison of solution time and true relative error for a bunch of elliptic solvers (direct, FACR, SOR, MG-Vcycle and FMG-Vcycle), Multi-grid techniques use a standard (2,1) V-cycle (i.e. 2 relaxations on the up-stroke and one on the down).(*a*) CPU times as measured on a RS6000 390H with 256Mb RAM (compiler flags are -O3 -qarch=pwr2 -qhot -qautodbl=dbl4) (*b*) average L_2 norm of the error from the true solution (this is actually the truncation error).

Recommendations If you have a Poisson or Helmholtz problem that can be addressed using the boundary conditions available in FISHPAK, by all means use

these routines (also if you have funny coordinate systems like polar coordinates). They are exceptionally fast and stable and are often a quick way to get going on a problem. For more general operators however Multi-Grid still maintains its remarkable convergence behaviour and it is not that difficult to modify the routines to handle more general problems and boundary conditions. In addition, Multi-grid is extremely modular so it is straightforward to replace any of the grid operators (for example changing the relaxation scheme to line relaxation) by just changing subroutines. In this way, Multi-grid is less of an algorithm and more of a strategy. Done correctly it can work wonderfully, done poorly... well what's new. Whatever you do, unless you really need the power of a direct solver, avoid them and for any problem where you would use SOR you should use multi-grid.

Bibliography

[1] W. Briggs. A Multigrid Tutorial, SIAM, Philadelphia, PA, 1987.

Chapter 10

High Performanc computing and parallel programming

Selected Reading

`http://mpi stuff`

10.1 Hardcore computing and the big Iron

10.2 Styles of Parallelism

10.3 Parallel programming on the IBM SP2: using MPI

Appendix A

Vector Calculus: a quick review

Selected Reading

H.M. Schey,. Div, Grad, Curl and all that: An informal Text on Vector Calculus, W.W. Norton and Co., (1973). (Good physical introduction to the subject)

Mase, George. Theory and problems of Continuum Mechanics: Schaum's outline Series. (Heavy on tensors but lots of worked problems)

Marsden, J.E. and Tromba, A.J. . Vector Calculus. W.H. Freeman (or any standard text on Vector calculus)

In modeling we are generally concerned with how physical properties change in space and time. Therefore we need a general mathematical description of both the variables of interest and their spatial and temporal variations. Vector calculus provides just that framework.

A.1 Basic concepts

Fields A field is a continuous function that returns a number (or sets of numbers) for every point in space and time $(\mathbf{x}, t)$. There are three basic flavours of fields we will deal with

scalar fields A scalar field $f(x, y, z, t)$ returns a single number for every point in space and time. Examples include temperature, salinity, porosity, density....

vector fields A vector field $\mathbf{F}(x, y, z, t)$ returns a vector for every point in space and is readily visualized as a field of arrows. Examples include velocity, elastic displacements, electric or magnetic fields.

tensor fields A second rank tensor field $\mathsf{D}(x, y, z, t)$ can be visualized as a field of ellipsoids (3 orthogonal vectors for every point). Examples include stress, strain, strain-rate.

Notation There are many different notations for scalars, vectors and tensors (and they're often mixed and matched); however, a few of the common ones are

scalars scalars are usually shown in math italics e.g. $f, g, t \ldots$

vectors come in more flavours. when typeset they're usually bold-roman characters e.g. $\mathbf{V}$, or the unit vectors $\mathbf{i}, \mathbf{j}, \mathbf{k}$. When hand written they usually have a line underneath them. Vectors can also be written in component form as $\mathbf{v} = v_x\mathbf{i} + v_y\mathbf{j} + v_z\mathbf{k}$ or in index notation $\mathbf{v} = v_i\hat{\mathbf{e}}_i$ where $\hat{\mathbf{e}}_i$ is another representation of the unit vectors.

tensors (actually second rank tensors) Typeset in sans-serif font D (or often just a bold $\boldsymbol{\sigma}$), handwritten with two underbars, or by component D_{ij}. 2nd rank tensors are also conveniently represented by matrices.

Definitions of basic operations

vector dot product

$$\mathbf{a} \cdot \mathbf{b} = a_i b_i = |\mathbf{a}||\mathbf{b}| \cos\theta \tag{A.1.1}$$

is a scalar that records the amount of vector **a** that lies in the direction of vector **b** (and vice versa). θ is the smallest angle between the two vectors.

vector cross product $\mathbf{c} = \mathbf{a} \times \mathbf{b}$ is a vector that is perpendicular to the plane spanned by vectors **a** and **b**. The direction that **c** points in is determined by the right hand rule. Note $\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$. The cross product is most easily calculated as the determinant of the matrix

$$\mathbf{c} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} \tag{A.1.2}$$

or

$$\mathbf{c} = (a_y b_z - a_z b_y)\mathbf{i} - (a_x b_z - a_z b_x)\mathbf{j} + (a_x b_y - a_y b_x)\mathbf{k} \tag{A.1.3}$$

or in index notation as $c_i = \epsilon_{ijk} a_j b_k$ where ϵ_{ijk} is the horrid permutation symbol.

tensor vector dot product is a vector formed by matrix multiplication of a tensor and a vector $\mathbf{c} = \mathsf{D} \cdot \mathbf{a}$. In the case of stress, the force acting on a plane with normal vector **n** is simply $\mathbf{f} = \boldsymbol{\sigma} \cdot \mathbf{n}$. Each component of the vector is most easily calculated in index notation with $c_i = D_{ij} a_j$ with summation implied over repeated indices (i.e. $c_1 = D_{11}a_1 + D_{12}a_2 + D_{13}a_3$ and so on for $i = 2, 3$.

A.2 Partial derivatives and vector operators

Definitions Given a scalar function of one variable $f(x)$, its derivative is defined as

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{A.2.1}$$

(and is locally the slope of the function). Given a function of more than one variable, $f(x, y, t)$, the partial derivative *with respect to* x is defined as

$$\frac{\partial f}{\partial x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x, y, t) - f(x, y, t)}{\Delta x} \tag{A.2.2}$$

i.e. if we sliced the function with a plane lying along x, the partial derivative would be the slope of the function in the direction of x (See Figure A.1a) likewise y or t.

In space in fact it is convenient to consider all of the spatial partial derivatives together in one handy package, the 'del' operator ($\boldsymbol{\nabla}$) a.k.a. the upside down triangle.. In Cartesian coordinates this operator is defined as

$$\boldsymbol{\nabla} = \mathbf{i}\frac{\partial}{\partial x} + \mathbf{j}\frac{\partial}{\partial y} + \mathbf{k}\frac{\partial}{\partial z} \tag{A.2.3}$$

In combination with vector and scalar fields, the del operator gives us important information on how these fields vary in space. In particular, there are 3 important combinations

the Gradient the gradient of a *scalar* function $f(\mathbf{x})$

$$\boldsymbol{\nabla} f = \mathbf{i}\frac{\partial f}{\partial x} + \mathbf{j}\frac{\partial f}{\partial y} + \mathbf{k}\frac{\partial f}{\partial z} \tag{A.2.4}$$

is a *vector* field where each vector points 'uphill' in the direction of fastest increase of the function (See Figure A.2).

the Divergence the divergence of a *vector* field

$$\boldsymbol{\nabla} \cdot \mathbf{F} = \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} + \frac{\partial F_z}{\partial z} \tag{A.2.5}$$

is a scalar field that describes the strength of local sources and sinks. If $\boldsymbol{\nabla} \cdot \mathbf{F} = 0$ the field has no sources or sinks and is said to be 'incompressible'.

the Laplacian the Laplacian of a *scalar* field

$$\nabla^2 f = \boldsymbol{\nabla} \cdot (\boldsymbol{\nabla} f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \tag{A.2.6}$$

is a scalar field that gives the local curvature (See Figure A.3).

the Curl the curl of a vector field

$$\boldsymbol{\nabla} \times \mathbf{F} = (\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z})\mathbf{i} - (\frac{\partial F_z}{\partial x} - \frac{\partial F_x}{\partial z})\mathbf{j} + (\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y})\mathbf{k} \tag{A.2.7}$$

is a vector field that that describes the local rate of rotation or shear.

Other useful relationships Given the basic definitions, there are several identities and relationships that will be important for the derivation of conservation equations.

Gauss' divergence theorem Gauss's theorem states that the flux out of a closed surface is equal to the sum of the divergence of that flux over the interior of that volume (it is actually closely related to the definition of the Divergence). Mathematically

$$\int_S \mathbf{F} \cdot d\mathbf{S} = \int_V \boldsymbol{\nabla} \cdot \mathbf{F} dV \tag{A.2.8}$$

useful identities the first homework will make you show that

1. $\boldsymbol{\nabla} \cdot (\boldsymbol{\nabla} \times \mathbf{F}) = 0$ (i.e. if a vector field can be written as $\mathbf{V} = \boldsymbol{\nabla} \times \mathbf{g}$ then it is automatically incompressible).
2. $\boldsymbol{\nabla} \times (\boldsymbol{\nabla} f) = 0$ (a gradient field is irrotational)
3. $\boldsymbol{\nabla} \times \boldsymbol{\nabla} \times \mathbf{V} = \boldsymbol{\nabla}(\boldsymbol{\nabla} \cdot \mathbf{V}) - \nabla^2 \mathbf{V}$
4. $\boldsymbol{\nabla} \times [(\mathbf{V} \cdot \boldsymbol{\nabla})\mathbf{V}] = (\mathbf{V} \cdot \boldsymbol{\nabla})[\boldsymbol{\nabla} \times \mathbf{V}]$

Figures A.1–A.5 show some examples of scalar and vector fields and their derivatives.

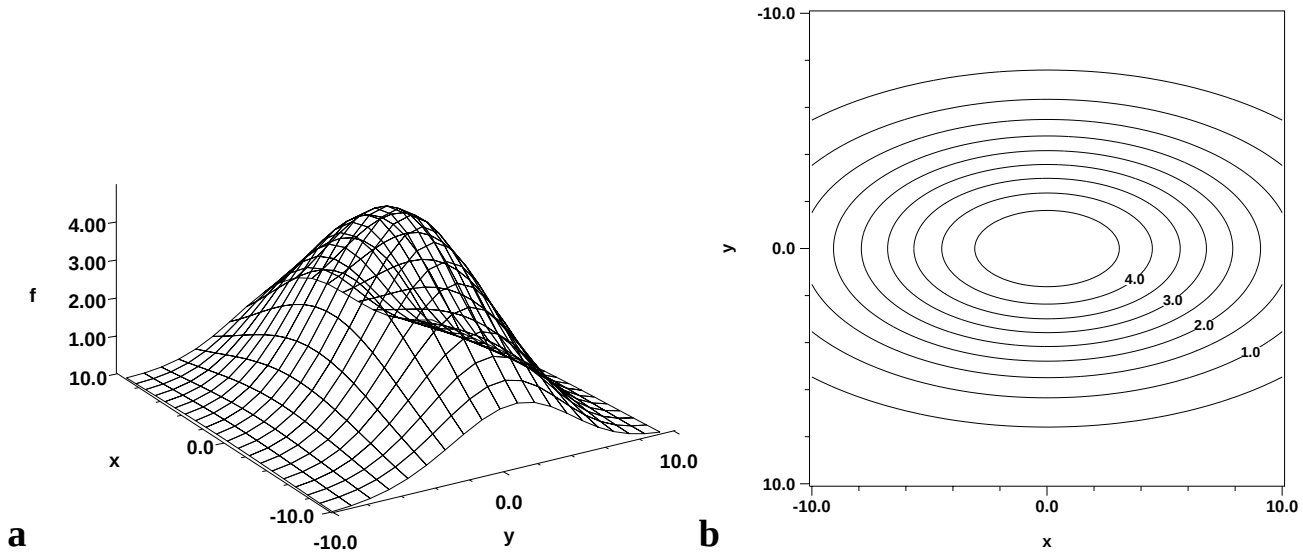


Figure A.1: (a) Surface plot of the 2-D scalar function $f(x, y) = 5 \exp\left[-(x^2/90 + y^2/25)\right]$ (b) contour plot of the same function.

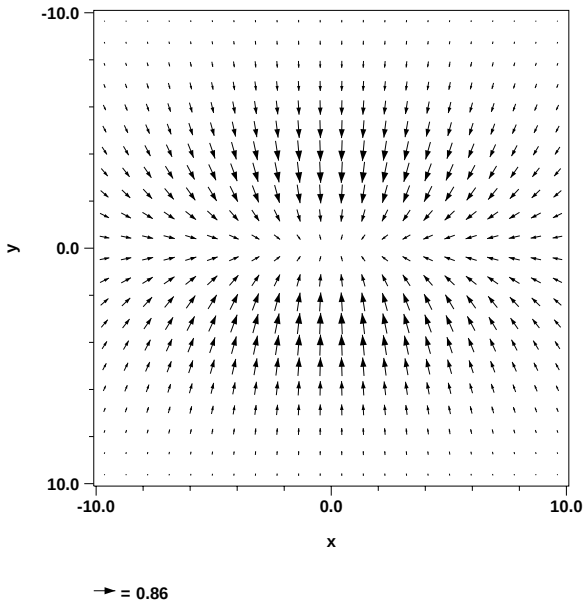


Figure A.2: Vector plot of $\boldsymbol{\nabla} f(x, y)$ for the function f in Figure A.1.

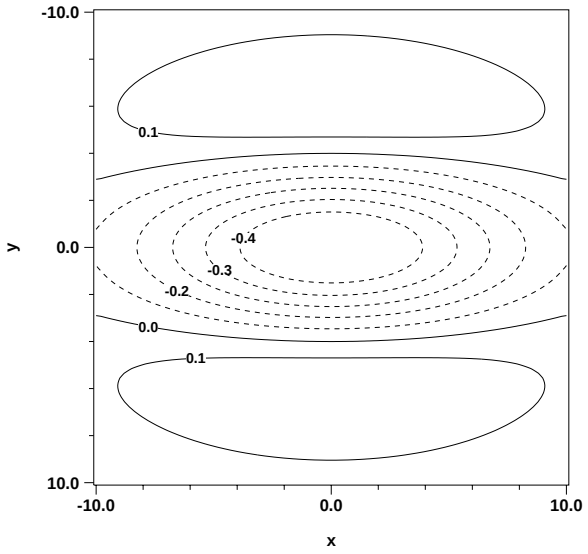


Figure A.3: contour plot of of the divergence of the vector field in Figure A.2. Because $\boldsymbol{\nabla} \cdot (\boldsymbol{\nabla} f) = \nabla^2 f$ this plot is also a measure of the curvature of the function f.

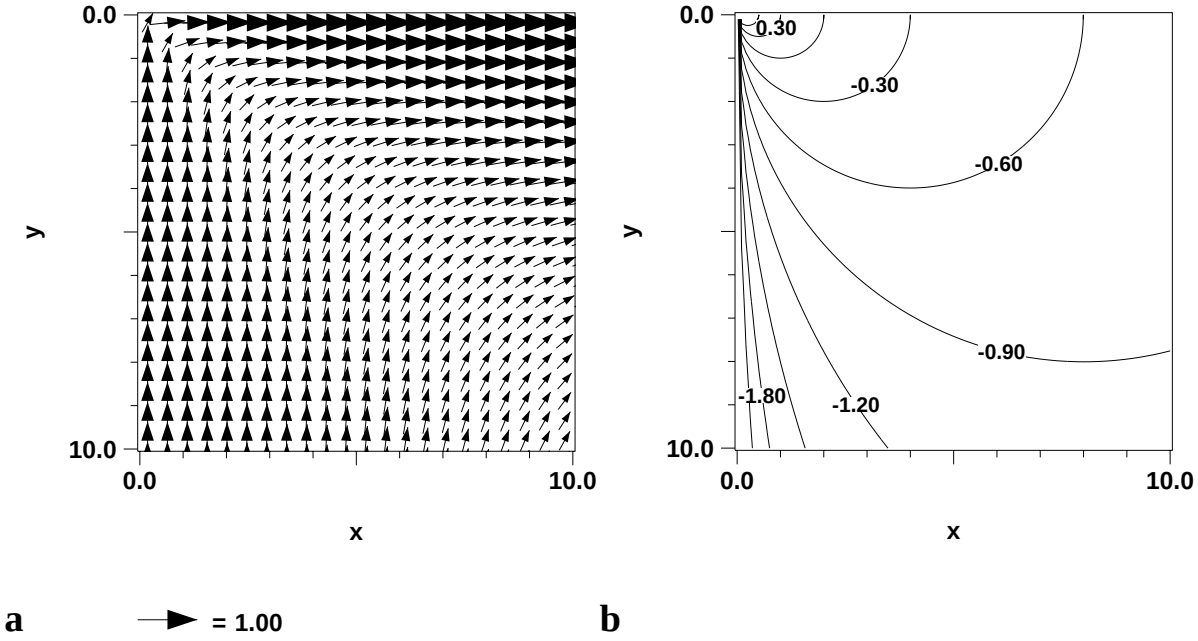


Figure A.4: (a) Vector plot of the 2-D corner flow velocity field $\mathbf{V}(x,y) = 2/\pi\left[\left(\tan^{-1}(x/y) - xy/(x^2+y^2)\right)\mathbf{i} - y^2/(x^2+y^2)\mathbf{j}\right]$ (b) contour plot of $\log_{10}(\boldsymbol{\nabla}\times\mathbf{V}\cdot\mathbf{k})$. The maximum rate of rotation is in the corner. There is no rotation directly on the x axis. This field is incompressible however and $\boldsymbol{\nabla}\cdot\mathbf{V} = 0$

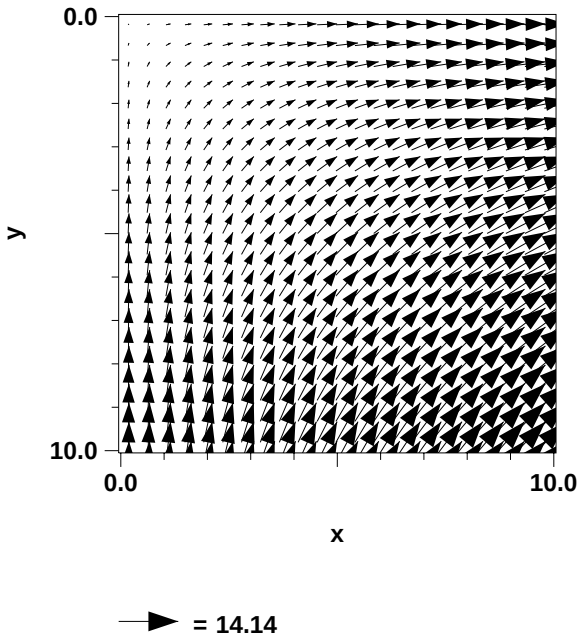


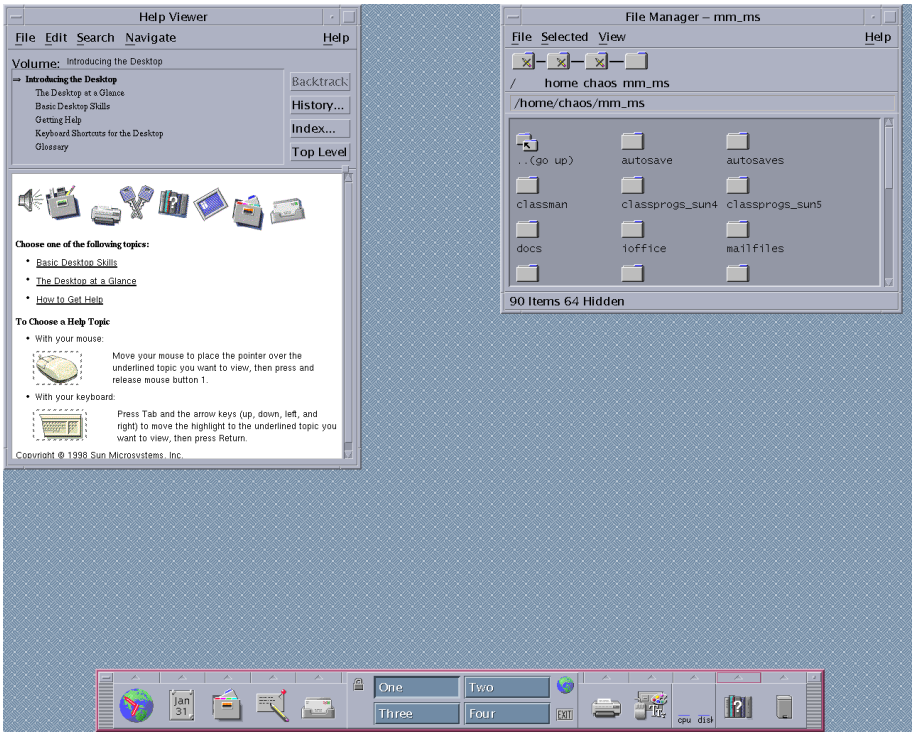
Figure A.5: Vector plot of pure-shear flow field $\mathbf{V}(x,y) = x\mathbf{i} - y\mathbf{j}$. Although the flow lines of this field are superficially similar to those of Figure A.4, this flow is locally irrotational i.e. $\boldsymbol{\nabla}\times\mathbf{V} = 0$. This flow is also incompressible.

Appendix B

Using the Suns at Lamont

B.1 Getting started: Logging in

Find any vacant (ha!) Sun Workstation that you are allowed to log into for free, and type your username at the login prompt. Your username for this course will be of the form `mm_??_00` the question marks relate to your name (my userid is `mm_ms`). The machine will then prompt for your password (but not reflect it). At the moment you can choose one of two windowing systems. I prefer CDE (Common Desktop Environment) which is becoming standard(ish) among Unix vendors. If you've chosen this and are logging in for the first time, your screen should look something like.



Changing your password If you are a first time user for this account you will have been issued a temporary password. You should change this pronto. First you will need to pop up a command window. To do this, move the cursor over the background and hold down the right button. This will produce the *root menu*. Go to programs and choose terminal (and while you're at it, pop up a Console). Now go to the terminal window type `passwd`. You will be prompted for your current password, then asked for your new passwd twice just to make sure. None of the typing will be echoed to the screen. For security passwords should be at least 6 letters long and contain some funky characters like !#$% or numbers. Do not use proper names, common words, birthdays, etc. If all goes well, your should see the following

```
2-mingus% passwd
Changing NIS password for mm_ms on chaos.
Old password:
New password:
Retype new password:
NIS entry changed on chaos
```

Getting around OpenWindows/CDE The CCE user-interface is a typical move-click-and-drag mouse driven system. It is similar to the macs with the principal distinction of having a 3 button mouse and no obvious menu bar. The initial help screen has 45 tedious pages of how to get around CDE most of which is pretty obvious. The most important thing to know is that the right button of the mouse will pop-up any appropriate menus for whatever application the mouse is pointing at. The most important menu is found on the root window (the background). This *Root Menu* allows you access to commonly used programs, let you set and save the properties of your workspace, calls up the help viewer and most importantly lets you exit the windowing system. In addition, CDE also has a bottom toolbar with many of the options of the rootmenu on it and it allows you to change your workspace. The left mouse button is for selecting things, double clicking and highlighting text. The best way to get a feel for all these widgets and buttons is to sit down and play and use the annoying helpviewer if you need to. **Remember:** The best way to learn this stuff is to poke and play at it. There is nothing you can break so go wild. For practice, a good thing to do at this point is to use the *Desktop Style* option on the toolbar to configure your workspace to your liking.

Getting around the filesystems Once you're comfortable mousing around, the next important task is to maneuver around the file system, change directories and deal with files. For this you have two options

1. Use the file manager. The file manager is a Mac-like interface with folders, files etc. It can be useful and is a good way to get a feel for the dimensions of the lamont filesystem but it is not a substitute for learning Unix.

2. Learn Unix. Go to the cmdtool and start hacking, or better yet, fire up XEmacs (type `xe`) and start the shell from there.

B.2 Basic Unix Concepts and Commands

Filenames and directories The basic working unit in Unix are files and directories (actually directories are just special files). Files and directories all have names which can be of any length but should have *no spaces* and should avoid the special characters !$*. Some common filename conventions and examples are

```
Articles        News            classprogs_sun4 intro.tex
Handouts        class.cshrc     dead.letter     picture_0.ps
```

Note: Unix file names are case sensitive so `news` and `News` are two different directories. The directory structure under Unix is a simple inverted tree starting with the root directory and continuing down different branch directories (e.g. my home directory is `/home/chaos/mm_ms`) with the individual files at the bottom. A few special directories are

. the current directory

.. the parent directory (one directory up closer to the root)

∼`userid` the home directory of a user called userid (e.g.`~mspieg`)

∼ your home directory

Commands and the Manual pages The following sections will outline most of the common commands for maneuvering and manipulating files and directories and doing just about everything. An example session of usage will be included at the end.

Commands are typed at the unix prompt usually followed by arguments and files to be operated on. The following list is organized by concepts. This list is not complete and does not supply all the possible thousand permutations of command line arguments. To get more information about any command, the most important command is `man` which will list the manuals page describing everything you can do with a command. Examples include

`man ls` Give us information about the command ls (list files)

`man -k gzip` Give a short description of any command that deals with file compression (actually, any command whose description contains the string compress)

man in Xemacs in XEmacs, a more convenient way to access the man pages is under the Help menu under manuals (or just type `M-x manual-entry`).

Another important source of online help is the AnswerBook, which contains the entire Sun documentation set online (with pictures and everything). Answerbook can be searched for any topic (not always successfully) and has gobs of information on Openwindows, basic UNIX, Fortran (f77 and f90), configuring your environment etc. You can start the answerbook from the toolbar library menu. When in doubt

1. RTFM
2. Ask somebody who might know (other students are particularly good resources).

Self Awareness Commands

whoami returns your userid (useful when your confused)

pwd returns the current directory you are in

hostname returns the name of the machine you are on

Moving around Directories

cd change directory, some examples

```
cd
cd ..
cd ~mm_ms
cd /chaos/chaos/magma/mspieg
```

ls list the contents of a directory, thousands of options

lf list the contents of a directory with file information (alias of ls -F)

llf long listing with file information (alias of ls -lF)

mkdir make a subdirectory in the current directory (`mkdir crud`)

rmdir remove an empty directory (`rmdir crud`)

Doing (unspeakable) things to any file

cp copy files to other files or files to directories, examples

```
cp file1 file2
cp file1 directory
cp file1 file2... filen directory
```

mv move or rename files. essentially a copy then delete. Same syntax as cp

rm REMOVE files. Careful with this one, rm is forever.

chmod change the read-write-execute permission of a file

Doing (unspeakable) things to text files

cat spit out a file to standard out, can also be used to concatenate several files into one file (see the section on pipes)

more see the contents of a file, one page at a time `more file.txt`.

textedit open up a simple mouse driven texteditor (aliased to te) `textedit file.txt`

vi native Unix texteditor, not mouse driven. Useful for dumb terminals but otherwise a steep learning curve

emacs another editor. A favourite among real Unix hacks but not immediately intuitive.

xemacs A much more beautiful version of Emacs for X-Windows. Lots of lovely buttons and menus for keystroke-impaired (highly recommended)

lpr send a file to the printer (usually a laserwriter)

enscript format a beautiful file and send to the printer

grep Very powerful tool for finding strings in textfiles and printing out matching lines (see examples)

Programming Tools

f77/f90 the fortran compiler

cc/gcc the C compiler

make a very useful tool for building multi-file programs

Workshop An interactive debugging and programming environment that can be run from within XEmacs.

X and remote machines in Unix and OpenWindows you can compute on another machine across the planet or network just as easily (but not as quickly) as the one your sitting in front of. To access a remote machine use

xhost on your local machine to allow remote X access (e.g. `xhost +`)

rlogin/telnet to remote login to another machine `rlogin ouzel`

xsetd set your X display to your local machine `xsetd host` (Not a real unix command just my alias for `setenv DISPLAY host:0`.

File transfer and the Internet There is a whole planet of machines and programs living out on the net. Some useful commands for connecting to outside machines and transferring files are

telnet connect to a remote machine on the internet (or locally) `telnet columbianet` or `telnet phx.cam.ac.uk`

ssh If available, better than telnet. starts the secure shell which will allow encrypted telnet and Xsessions. It will automagically set up the proper displays. ssh can be slow but it protects you against password sniffing etc.

ftp file transfer protocol, get and send files directly to other machines. Most useful as *anonymous ftp* (see xarchie) for searching for freely available software and information.

Netscape Okay, if you don't know what this is by now I have a brooklyn.bridge.com to sell to you. . . .

Panic Buttons and Bailing out Quite often things will go wrong, programs hang up or you just want to get out of something. There are a few Control sequences which you should know. These are often abbreviated with a caret or uparrow character for the control key. Thus Control-c means hold down the control key and type C. emacs shorthand for Control-c is C-c. Useful panic buttons and commands are

Control-c Exit the current program (within emacs shell use C-c C-c)

Control-z Stop the current program (C-c C-z)

Control-d Leave a cmdtool or shell (C-c C-d)

logout Leave a cmdtool or shell

kill kill a specified process (you need to find out the process id)

killproc kill a process by name `killproc text` will kill any process that contains the string text. **This is not a unix command but a local program I wrote** (do a `more ~mm_ms/classprogs/sun4/killproc` to see some real ugly shell programming).

If your whole workstation hangs up, don't panic just find another workstation, rlogin to the hung one and start killing processes selectively. If you're really panicked `killproc csh` Will blow everything back to the login prompt.

Spiegelman Specials many commands can be abbreviated, aliased, or included in *scripts* which are just files of commands. A few of these combined commands that I find particularly useful have been included for the class through the class.cshrc file. Just for reference a few of these are

lf alias of ls-F

pun find a process by name. E.g. `pun csh` will find all processed with the string csh. `pun mspieg` will find all process owned by mspieg.

te run texteditor in the background

xe run xemacs in the background

xcnet pop an X-term running columbianet

gv run the other postscript viewer, ghostview in the background

tidy removes the texteditor and emacs backup files (they end with a % or ~)

wpr find out what the current default printer is (alias printenv PRINTER)

spr set the default printer `spr cookie_lw`

xsetd set the X display

killproc kill processes by name

crexp do a global search and replace for a single pair of strings. E.g. `crexp string1 string2 file1 file2...filen` will replace every occurence of string1 with string2 in files 1 to n

getnr get numerical recipes (fortran) routines.

B.3 Additional commands and programs

This section outlines just some of the programs available at Lamont. They have been organized into three areas.

CDE Tools These are window based tools that come as part of CDE. Most of these are accessible from the root menu under Programs or the bottom toolbar.

Command Tool a scrollable command line interface

Text Editor a simple mouse driven text editor (not a word processor)

File Manager a mac-like file system browser and controller

Mail Tool an intuitive interface for e-mail (but also see VM in xemacs or the netscape mailers).

Calculator a reasonable scientific calculator

Perfmeter a graphical tool to show how much CPU you're burning

audiotool listen to sound files on cheesy speakers

imagetool a postscript and image previewer (I prefer gv and xv).

Games, Demos, and other things Lousy games for such an expensive machine but check out the root menu to see everything else. Spider is a good but hard solitaire game. (XEmacs has tetris though).

Freeware at Lamont These are general utilities and programs that have been installed at lamont. These programs are of variable quality and reliability but they can be very useful.

xemacs The kitchen sink of free-ware...Have I said how much I like this program? Reads your news, writes and debugs your code, washes the dishes and gives you a healthy coat and shiny teeth.

StarOffice A rather remarkable, *free* MicroSt office clone that runs under Unix/Linux/you-name-it. Has a particularly nice spreadsheet, much like Excel. To install type soffice at the command line and follow directions. Once installed it will show up in the toolbar.

acroread Free Adobe Acrobat reader for PDF's

rn, xvnews, vnews Various news readers (xvnews is X-window based, see also Gnus in xemacs and Netscape News)

latex, tex Donald Knuth's super duper technical typesetting program (also runs nicely under emacs)

Netscape Web-o-rama

xmgrace a free WYSIWYG xy plotting program

xfig a free drawing program, also allows you to annotate postscript. A little clunky but useful for quick figures and posters.

gmtsystem a script driven postscript and mapmaking toolkit

XGB Geobase – Bill Menke's graphical database browser (X-Version)

ahsystem Lamonts time-series filters (probably obsolete by now)

pbm toolkit A suite of simple *filters* for image processing

xv an interactive image viewer

xanim A useful animation viewer (gifs, mpgs, quicktime)

gimp Gnus Image Processing system, free photoshop...way cool.

emacs GNU (Gnu's Not Unix) editor environment

gcc/g77/g++ GNU's free c, fortran and C++ compilers

gzip, gunzip, gzcat GNU's free file compression/ uncompression tools

and much much more... look in `/usr/local/bin` for just a sampling.

Licensed Software at Lamont This software is commercial, slick and expensive. It also comes with the nasty baggage of floating licenses where anybody at lamont can run the programs from any SunWorkstation; however, if there are only two licenses, only two people can run it simultaneously. **SO DON'T LEAVE THEM OPEN IF YOU'RE NOT USING THEM**. Some of the more useful software is

matlab MathWorks interactive system, particularly good for signal processing and numerical linear algebra and practically everything. Extremely useful for quick prototyping and integrating calculations with graphics. Many licenses I believe.

mathematica Symbolic math and graphics package. Way cool if you can figure out the peculiar syntax.

avs Advanced Visual Systems program for way-cool 3-D visualization and computation. Very flexible and extendible. **4 licenses** needs at least a color monitor and is very cpu and memory intensive

dx IBM's version of avs. Available on the SP2

acrobat/distill Unix version of acrobat for Suns. **2 licenses**. `acroexch` allows manipulation of pdf files. `distill` is very useful for making excellent pdf files from postscript (and excellent postscript from pdf files).

xframe Frame Technology's Framemaker. WYSIWYG textprocessor and formatter.

ermapper Real slick, image processing environment. Only has a student license (talk to Jeff Weissel or Chris Small if you need more info).

envi More image processing goodies in the Visualization lab.

B.4 Shortcuts and the csh

Unix is more than a collection of commands it's also a philosophy (oooooh, deeeeeeeeep) that by combining lots of simple tools that do just one thing, you can build custom tools that do anything. There are therefore, some shortcuts and

useful structures you can use to do all kinds of things. The `csh` (pronounced C-Shell - or seashell), sits between you and the computer and interprets commands. Most of these commands and shortcuts use certain special characters. For a full understanding of the power of the csh, look at `man csh` or answerbook. On a day to day basis however there are a few csh shortcuts that will make your typing faster and your commands immensely cryptic. (Alternatively you can run the shell from within xemacs which has some handy filename completion and history functions that are great).

Filename completion and wildcards Any filename can be extended using the two wildcard characters `*` and `?`. The csh will expand `string*` to any filename that starts with string and has any number of characters that follow. `*string*` matches any file that contains string. `*` matches all files. `?` works the same way but only expands to single characters. Examples

```
lf *.tex
grep howdy *.tex
rm * (DON'T EVER DO THIS WITHOUT A GOOD REASON)
more file_??.ps
```

history substitution the csh keeps a record of your last 25 commands (this number can be changed). Typing `h` (alias to `history`) gives you a list of these commands and a number. Using the `!` character you can rerun these commands quickly. Important examples

!! Run the previous command

!21 run command number 21

!more run the most recent command that contained the string more

If you are running the shell under xemacs, it has a much nicer history function which you can access from either the *Complete menu* or by using Control and arrow keys.

pipes and redirecting input and output A standard unix command reads a file from its standard input and writes it to it's standard output (usually the terminal). This *data stream* can also be redirected into files or other programs so that the output of one program becomes the input to another. Thus very powerful processing filters can be built out of simple tools. The principal symbols for redirection are the pipe symbol `|` and the to/from file symbols $<, >$ for directing output from and to a file and $>>$ to append to a file. Some examples

`gzcat file.gz|more` uncompress the compressed file `file.gz` and pipe the output through more just to see the thing one page at a time

`gzcat file.ps.gz |lpr` send the postscript file to a printer

`gzcat problems.tar.gz |tar -xvf -` uncompress and unpack a tar (tape archive) file. We will use this a lot.

`myprogram < input > output` run a program and take its standard input from file `input` and write it's standard output to file `output`.

quotes single and double quotes in unix have different meanings and can be used to lump strings together and protect commands from file expansion by the shell. loads of fun

B.5 A typical unix session: sort of old now

```
1-mingus% whoami
mm_ms
2-mingus% cd
3-mingus% pwd
/tmp_mnt/data/sarah/mm_ms
4-mingus% lf
Articles  class.cshrc  mout
Handouts/  classprogs_sun4/ mylog.uu
News/   dead.letter  unix/
5-mingus% more class.cshrc
#####################################################################
#   Greetings:  This is the class .cshrc file for Myth's and
# Methods in Modeling (mandms)
# This file will help get you all set up to do the homeworks and
# use some of the available tools.  If this file is included in
# your personal .cshrc file,  any changes and additions for the class
# will automatically be added to your environment.  Have fun
#####################################################################

######################
#  set up TeX Environment for interactive shells
######################
if (  { tty -s } ) then
source /lamont/scratch/mspieg/tex/tex.cshrc2
endif

######################
#  set up GMT Environment
######################
gmtenv

######################
#  set up  path for shared course programs
######################
setenv CLASSBIN /data/sarah/mm_ms/classprogs_sun4
setenv MYBIN ${HOME}/${ARCH}
set path= ( $path $MYBIN $CLASSBIN )


######################
#  set up  default permissions (user+group=read,write,execute others read+x)
######################
umask 002


######################
#  Useful (?) aliases
######################
alias cp cp -i        #  check before overwriting a file
alias mv mv -i       #  check before overwriting a file
alias hg 'history | grep \!* | grep -v hg' #  check history by name
alias lf 'ls -F'      #  show filetype in short listing
alias llf 'ls -lF'      #  show filetype in long listing
alias pun  'ps -aux | grep \!* | grep -v grep'  # see processes by name
```

```
alias pv 'pageview \!*  &'     #  run pageview in the background
alias te 'textedit \!*  &'     #  run textedit in the background
alias tidy 'rm *%'         #  clean up texteditor backup files
alias spr 'setenv PRINTER \!*'     #  set a printer by name
alias wpr 'printenv PRINTER '     #  see what printer is set
alias xsetd 'setenv DISPLAY {\!$}:0'    #  set your X display by hostname

########################
#   configure opewindows default layout
########################

if ( ! -e $HOME/.openwin-init ) then
echo sorting out your windows
cp ~mm_ms/.setup/.??* $HOME
endif
6-mingus% lf
Articles  class.cshrc  mout
Handouts/  classprogs_sun4/ mylog.uu
News/   dead.letter  unix/
7-mingus% cd Handouts
8-mingus% lf
Conserv1.ps.Z Conserveq2.ps.Z Intro.ps.Z Vectorcalc.ps.Z problems1.ps.Z
9-mingus% zcat problems1.ps | head
%!PS-Adobe-2.0
%%Creator: dvips 5.47 Copyright 1986-91 Radical Eye Software
%%Title: problems.dvi
%%Pages: 2 1
%%BoundingBox: 0 0 612 792
%%EndComments
%%BeginProcSet: tex.pro
/TeXDict 200 dict def TeXDict begin /N /def load def /B{bind def}N /S /exch
load def /X{S N}B /TR /translate load N /isls false N /vsize 10 N /@rigin{
isls{[0 1 -1 0 0 0]concat}if 72 Resolution div 72 VResolution div neg scale
10-mingus% zcat problems1.ps | pv -
[1] 12660 12661
11-mingus% cd ..
12-mingus% lf
Articles  class.cshrc  mout
Handouts/  classprogs_sun4/ mylog.uu
News/   dead.letter  unix/
13-mingus% te dead.letter
[1] 12664
14-mingus% mkdir crud
15-mingus% lf
Articles  classprogs_sun4/ mylog.uu
Handouts/  crud/   unix/
News/   dead.letter
class.cshrc  mout
16-mingus% cp dead.letter crud
17-mingus% cd crud
18-mingus% lf
dead.letter
19-mingus% cp * junk.letter
20-mingus% lf
dead.letter junk.letter
21-mingus% rm junk*
22-mingus% h
     1 whoami
     2 cd
     3 pwd
     4 lf
     5 more class.cshrc
     6 lf
     7 cd Handouts
     8 lf
     9 zcat problems1.ps | head
    10 zcat problems1.ps | pv -
    11 cd ..
    12 lf
    13 te dead.letter
```

```
    14 mkdir crud
    15 lf
    16 cp dead.letter crud
    17 cd crud
    18 lf
    19 cp * junk.letter
    20 lf
    21 rm junk*
    22 h
23-mingus% hg junk
    19 cp * junk.letter
    21 rm junk*
24-mingus% cp ~mspieg/course/Unix.tex .
```

Appendix C

A Fortran Primer: (and cheat sheet)

This section will provide a basic intro to most of the commonly occuring features of Fortran that you will need for the course. This list is by no means exhaustive, but it should be enough to get you where you need to go. For more information, We have extensive fortran manuals scattered about the observatory. By the end of this section you should understand the basic data types, the structure of arrays, standard arithmetic operations, loops and conditional statements, subroutines and functions, and basic IO. This section will also briefly discuss how to build programs using `make` and how to debug them using `dbxtool`/`debugger`.

C.1 Basic Fortran Concepts and Commands

Data types for the most part there will be only three basic types of data you will have to deal with, integers, floating point numbers and characters. In fortran these data types are declared as

`integer` exact whole numbers (-3, 0, 5 234), usually stored in 4 bytes

`real` inexact representation of decimal numbers (3.1415, 27.2, 1.e23). Usually stored as 4 bytes, good for about 6-9 significant digits, ranges from about 10^{-38}–10^{38}

`double precision` same as real but much larger range and precision. Usually stored as 8 bytes, good for about 15-17 significant digits, ranges from about 10^{-308}–10^{308}. You can get more precision than this but you should have a good reason.

`character, character*n` either a single character or a string of characters of length `n`.

Constant and Variable names A constant or variable can have any name with up to 32 characters (To play it safe though, Standard Fortran allows only 6 characters). The name must start with a letter, but can have most of the

alphanumeric characters in it. Fortran also has the annoying feature of **implicit typing** so that if undeclared, any variable name that starts with the letters `i` through `n` are assumed to be integers, everything else is assumed real. Rigorous programming practice suggests that you start every program with `implicit none` and declare every variable. It's annoying but it keeps you honest. I will try to do this in my programs. A sample program declaration might look like

```
implicit none
integer npnts, n
real  c, dcdt
real  car(1000),tar(1000)
character*40 fileout
```

arrays Perhaps the most important feature for modeling is arrays. Arrays are simply ordered sets of numbers that can be addressed by index. Every array has a name (e.g. `car` or `tar` above) and a length (here both `car` and `tar` are arrays of real numbers of length 1000). Arrays can be of integers, reals, double precision or even characters (`fileout` is actually an array of 40 characters). Each member of an array can be addressed as by specifying its index in the array (`car(10)`, `tar(n)` etc.). Arrays can also have up to 7 dimensions. A two dimensional array `a(10,10)` can be thought of as 10 1-dimensional arrays of 10 numbers each (a total of 100 elements). Most importantly in fortran, the leading index increases fastest i.e. for `a(i,j)`, the `i=1` and `i=2` are next to each other in memory.

Simple operations integer and floating point numbers and arrays of them can all be operated on by standard mathematical options. The most commonly used arithmetic are

`=` assignment `x=y`

`**` exponential `a=x**y`

`/, *` divide, multiply `a=x/y, or a=x*y`

`+, -` add subtract `a=x+y, or a=x-y`

The order of evaluation is standard algebraic and parentheses can be used to group operands. In addition to the simple operations, Fortran also has some built in intrinsic functions. The most commonly occuring are

trigonometric functions `sin(x), cos(x), tan(x)`, `asin(x)`, `acos(x)`, `atan(x)`, `atan2(x)` (inverse trig functions) `sinh(x)`, `cosh(x)`, `tanh(x)` etc.

exponential functions `exp(x)`, `log(x)` (natural log), `log10(x)` (log base ten), `sqrt(x)` square-root.

conversion functions these functions will convert one data type to another, e.g. `int(x)` returns the integer value of `x`, `real(?)` converts anything

to a real, `dble(?)` converts anything to a double (also operations for complex numbers)

misc. functions see table 6.1

Program flow control Any program will just execute sequentially one line at a time unless you tell it to do something else. Usually there are only one of two things you want it to do, loop and branch. The control commands for these are

`do` **loops** Do loops simply loop over a piece of internal code for a fixed number of loops and increment a loop counter as they go. do loops come in the standard line number flavour

```
      do 10 i=1,n,2
        j=i+1
        other stuff to do
 10   continue
```

(note the spacing of the line number and starting commands at least 6 characters over is important (stupid holdover from punch cards). Or in the non-standard but more pleasant looking form

```
      do i=1,n,2
        j=i+1
        other stuff to do
      enddo
```

Do loops can be nested a fair number of times (but don't overdo it)

conditional statements Being able to make simple decisions is what separates the computers from the calculators. In fortran separate parts of the code can be executed depending on some condition being met. The statements that control this conditional execution are

`if` a single line of the form

```
      if (iam.eq.crazy) x=5.
```

will assign a 5. to the variable x if the expression in parentheses is true, otherwise it will simply skip the statement.

block if statements More useful blocks of execution can be delimited by block ifs of the form

```
      if ( moon.eq.full ) then
        call howl(ginsberg)
        x=exp(ginsberg)
      endif
```

the block if statements can also be extended to have a number of condition statements. If there are only two, it looks like

```
      if ( moon.eq.full ) then
        call howl(ginsberg)
        x=exp(ginsberg)
```

Table 6-1 Arithmetic Functions (continued)

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of*	
					Argument	*Function*
Absolute Value	|a| Read Note 6 $(ar^2 + ai^2)^{**}(1/2)$	1	ABS	IABS ABS DABS CABS CQABS♦ QABS ♦ ZABS ♦ CDABS♦	Integer Real Double Complex Complex*32 Real*16 Complex*16 Complex*16	Integer Real Double Real Real*16 Real*16 Double Double
Remainder	a1-int(a1/a2)*a2 Read Note 1	2	MOD	MOD AMOD DMOD QMOD ♦	Integer Real Double Real*16	Integer Real Double Real*16
Transfer of Sign	|a1| if a2 ≥ 0 -|a1| if a2 < 0	2	SIGN	ISIGN SIGN DSIGN QSIGN ♦	Integer Real Double Real*16	Integer Real Double Real*16
Positive Difference	a1-a2 if a1 > a2 0 if a1 ≤ a2	2	DIM	IDIM DIM DDIM QDIM ♦	Integer Real Double Real*16	Integer Real Double Real*16
Double and Quad Products	a1 * a2	2		DPROD QPROD ♦	Real Double	Double Real*16
Choosing Largest Value	max(a1, a2, ...)	≥ 2	MAX	MAX0 AMAX1 DMAX1 QMAX1 ♦	Integer Real Double Real*16	Integer Real Double Real*16
				AMAX0 MAX1	Integer Real	Real Integer
Choosing Smallest Value	min(a1, a2, ...)	≥ 2	MIN	MIN0 AMIN1 DMIN1 QMIN1 ♦	Integer Real Double Real*16	Integer Real Double Real*16
				AMIN0 MIN1	Integer Real	Real Integer

```
 else
    call wait(1,month)
 endif
```

or if there are several conditions

```
if ( expression ) then
   statements galore....
elseif (expression2)
   more statements
elseif (expression3)
   are you tired of this yet
else
  default statements
endif
```

Note: a block if of this type will execute the first true expression and the jump to the end if (even if several conditions are true).

relational operators The statement `moon.eq.full` is a conditional statement that evaluates to `true` if the variable `moon` is logically equivalent to the variable `full` (don't use = for `.eq.` very different animals). The operator `.eq.` is a relational operator that tests for equivalence. Other relational operators are `.lt.` (less-than),`.le.` (less-than-or-equal-to) ,`.ge.` (greater-than-or-equal-to),`.gt`] (greater-than). Individual conditional statements can also be linked together using the operators `.and.`, `.or.`, `.not.` (and a few less useful things like exclusive or `.xor.`. Examples include

```
 if ((moon.eq.full).and.(i.eq.werewolf))
&     call runlikehell()
  if ( (x.eq.0.5).or.(i.le.2) ) then
    x=x*i
  endif
```

Subroutines and Functions Subroutines and functions are separate pieces of code that are passed arguments, do something and return. Subroutines and functions are similar except that functions return a value and are used like the intrinsic functions i.e.

```
integer myfunction, k,j
external myfunction
real x
...
k=myfunction(x,j)
```

and are declared like

```
integer function myfunction(r,i)
real r
integer i
  myfunction=mod(int(r),i))
```

```
  return
  end
```

Note the variables `r` and `i` are "dummy variables" that hold the place of `x` and `j` in the main routine. Subroutines also have dummy variables but are "called" and don't return a value (although it will often change the values of what it is passed). Example subroutine call would be

```
integer  k,j
real x(100),a
....
j=10
a=2.5
call arrmult(x,j,a)
```

and the subroutine itself would look like

```
subroutine arrmult(ar,n,con)
integer n
real ar(n),con
integer i
do i=1,n
  ar(i)=ar(i)*con
enddo
return
end
```

Note that arrays are passed to subroutines and functions by name. Most importantly, the dimension of the array within the subroutine can be passed. In the above routine, only the first 10 elements of `x` are multiplied by `2.5`. If we wanted to multiply the elements from 6 to 16 we could also do

```
call arrmult(x(6),11,a)
```

Note that there are actually 11 elements between indexes 6 and 16. In addition, arrays that are declared as 1-D arrays in the main program can be operated on as variable length n-d arrays in a subroutine (and vice-versa). E.g.

```
integer  i,j
real x(1000),a
...
i=10
j=10
a=2.5
call arrmultij(x,i,j,a)
```

```
...
subroutine arrmultij(ar,m,n,con)
integer m,n
real ar(m,n),con
integer i,j
do j=1,n
  do i=1,m
    ar(i,j)=ar(i,j)*j*con
  enddo
enddo
return
end
```

We will make extensive use of this feature for efficient programming in n-dimensions. This is the one thing you cannot do well in C which is a real shame.

Input/Output The last important thing you might want to do is actually read information into a program and spit it out. This is perhaps the worst part of fortran, particularly when dealing with character strings. Nevertheless, with a few simple commands and unix, you can do most anything. A few important io concepts

logical units and open uggh, in fortran, files are referred to by "logical units" which are simply numbers. To open a file called `junk.txt` with logical unit 9 you would do something like

```
lu=9
open(lu,file='junk.txt')
```

The two most important files `stdin` and `stdout` already have logical units associated with them. `stdin` is 5 and `stdout` is 6.

reading a file to read data from a file you use the `read` statement. The principal version of this you will need to read things from the keyboard or from stdin looks like

```
read(5,*) tmax,npnts,(ar(i),i=1,3)
```

the `5` is the logical unit and the `*` says to just read in each datatype as whatever it was declared as so if `tmax` and `ar` are real and `npnts` is an integer it will assume that the numbers in stdout will be in those formats. Note the funny way of reading 3 items of array `ar` is known as an `implicit do loop` useful but clunky.

writing to a file to write data from a file you use the `write` statement which works just like read i.e.

```
write(6,*) 'here are some numbers ',tmax,npnts,(ar(i),i=1,3)
write(9,*) 'Howdy '
```

a synonym for `write(6,*)` is the `print` statement i.e.

```
print *, 'here are some numbers ',tmax,npts,(ar(i),i=1,3)
```

is equivalent to the first of the above two lines.

C.2 A Few pointers to good coding

Comment liberally Always write your code so that 6 months (or two weeks) from now you know what it does. comment lines start with a `c` in the first column or after a `!` in any column (the `!` comment is non-standard so be careful).

make 1 code to do 1 thing , Super-duper multi-purpose codes with hundreds of options become a nightmare to maintain and debug. Since you're only writing private code for yourself, I find it is most sensible to create separate programs for each difference in boundary or initial conditions etc. Using `make` and `makefiles` can simplify this process immensely. This way if you try something and it doesn't work, you can just go back to a previous version.

Write your loops right Always write your loops with the computer in mind. i.e. your inside loop should always be over the fastest increasing index. This will give the biggest increase in performance for the least amount of work.

NO GOTO's except in dire need avoid uncontrolled use of the `goto` statement as it will lead to immense confusion. See the first chapter of Numerical Recipes for the few necessary controlled uses of `goto`.

limit `ifs` and functional calls within array loops If a loop is designed to quickly handle an array, an embedded `if` statement or heavy function calls can destroy performance (although many optimizing compilers will do some strange things to try and prevent this).

keep it simple and conservative There are loads of fancy extensions in Sun fortran that might not work on other machines. The less fancy gee-gaws you use the less you have to replace when you change platforms

use `dbxtool`/`debugger` and `make` see below and man pages, these tools will make your life much easier for organizing and debugging code.

C.3 A sample program

```
      program euler1
c*****************************************************************
c euler1:  program to calculate the concentration of a
c   single radioactive element with time using an
c   euler method
c*****************************************************************

      implicit none
      integer nmax              !maximum array size
      parameter (NMAX=500)      ! set nmax to 500
      integer npnts, n          ! number of steps, counter
      real  c, dcdt             ! concentration, decay rate,
```

```
      real tmax, t,dt           ! max time,time, timestep
      real  car(NMAX),tar(NMAX) ! arrays for concentration and time
      character*40 fileout ! character array for output filename
      integer kf, lu  ! integer for character index, output file
      integer iprinttrue, iprinterr ! flag for calculating true solution,
                                ! or error (1 yes, 0 no)
      integer mylnblnk          ! even functions need to be typed
      external mylnblnk
c
c     -------read input
c
      read(5,*) fileout
      read(5,*) tmax, npnts, iprinterr, iprinttrue
c
c     -------set up initial parameters
c

      dt = tmax/ (npnts - 1)    ! set the time step (why is it n-1?)
      c = 1.                    ! initial concentration
      t = 0.                    ! initial time
      car(1)=c                  ! store in arrays car and tar
      tar(1)=t
c
c-------loop over time steps with an euler method
c
      do n=1,npnts-1            !start the loop
         call decay1(t,c,dcdt)  ! get the decay rate at the present step
         c=c + dcdt*dt          ! take an euler step
         t = dt*n               ! calculate the time for the next step
         car(n+1)=c             ! store in array car
         tar(n+1)=t             ! store the time in array tar
      enddo                     ! end the loop

c
c-------write the solution to fileout_c.xy
c
      lu=9                      ! set the fortran logical unit (ugh!) to any number
      kf=mylnblnk(fileout,len(fileout)) ! find the last blank space in string
      print *, 'Writing file ',fileout(1:kf)//'_c.xy'
      call writexy(lu,fileout(1:kf)//'_c.xy',tar,car,npnts) !writem-out
c
c-------if iprinterr=1, then calculate and write out fractional error between
c-------solution and true solution to fileout_cerr.xy
c
      if (iprinterr.eq.1) then
         do n=1,npnts
            car(n)=car(n)/exp( -1.*tar(n)) - 1.  ! calculate error
         enddo
         print *, 'Writing file ',fileout(1:kf)//'_cerr.xy'
         call writexy(lu,fileout(1:kf)//'_cerr.xy',tar,car,npnts)
      endif
c
c-------if iprinttrue=1, then calculate and write out true solution
c------- to fileout_ctrue.xy
c
      if (iprinttrue.eq.1) then
         do n=1,npnts
            car(n)=exp( -1.*tar(n)) ! calculate true concentration
         enddo
         print *, 'Writing file ',fileout(1:kf)//'_true.xy'
         call writexy(lu,fileout(1:kf)//'_ctrue.xy',tar,car,npnts)
      endif

c
c-------exit the program
c
      end


c**********************************************************************
c decay1:  subroutine that returns the rate of decay of a radioactive
c   substance as a function of  concentration (and time)
```

```
c   c is concentration
c   dcdt is the change in c with time
c   t is time
c   here, dcdt= -c
c*************************************************************************

      subroutine decay1(t,c,dcdt)
      implicit none
      real t, c, dcdt

      dcdt= -c

      return
      end

c*****************************************************************
c      mylnblnk integer function to return position of last non-character
c      in a string
c*****************************************************************

      integer function mylnblnk(str,lstr)
      character str(lstr),space
      integer l
      space=' '
      l=lstr
 10   if (str(l).eq.space) then
         l=l-1
         goto 10
      endif
      mylnblnk=l
      return
      end
C*************************************************************************
C     writexy:  writes 2 1-d arrays (x,y) to output file luout of name
C               fname.xy
C*************************************************************************

      subroutine writexy(luout,fname,xarr,yarr,npnts)
      integer luout
      character *(*) fname
      real xarr(npnts), yarr(npnts)

      open(luout,file=fname)
      do 10 j=1,npnts
         write(luout, *) xarr(j), yarr(j)
 10   continue
      close(luout)
      return
      end
```

C.4 A sample makefile

```
#########################################################################
#     makefile for the program euler1 which calculates
#    radioactive decay for a single element using an euler method
#               uses suns xtypemap to make double precision
#########################################################################

PROGRAM=euler1
OBJECTS=$(PROGRAM).o  decay1.o   writexy.o lnblnk.o
OPTLEVEL= -g
FFLAGS= $(OPTLEVEL) -xtypemap=real:64,double:64,integer:64
DESTDIR=$(HOME)/$(ARCH)


$(PROGRAM): $(OBJECTS)
$(FC)  $(FFLAGS) $(OBJECTS)  -o $(PROGRAM)

$(OBJECTS):
```

```
install: $(PROGRAM)
mv $(PROGRAM) $(DESTDIR)

clean:
rm -f *.o *.a core

cleanall:
rm -f *.o *.a core $(PROGRAM)
```