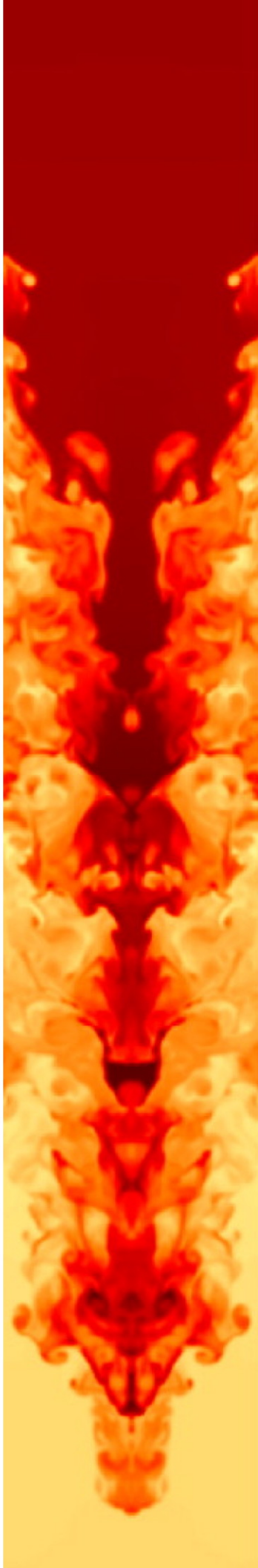
EDWARD BROWN

NUMERICAL TECHNIQUES IN ASTROPHYSICS



About the cover: The image is from a simulation of a single-mode Rayleigh-Taylor instability. Made with the FLASH code.
Credit: Calder et al. (2002) *The Astrophysical Journal Supplement Series*, 143, 201. Image copyright American Astronomical Society.

git version d2406547 ...

Preface

These notes are from a graduate-level course on numerical techniques in astrophysics at Michigan State University. The only preparation assumed is that the students have completed an undergraduate degree in physics or astronomy. The course was taught as a half-semester module, and therefore the scope is limited to being an introduction to assorted topics, rather than amore focused exploration of any one area.

The text layout uses the `tufte-book`[1] LaTeX class: the main feature is a large right margin in which the students can take notes; this margin also holds small figures and sidenotes. Exercises are embedded throughout the text. These range from "reading exercises" to longer, more challenging problems.

[1] https://tufte-latex.github.io/tufte-latex/

These notes are under active development; to refer to a specific version, please use the eight-character stamp labeled "git version" on the copyright page.

Contents

List of Figures

List of Exercises

1
Arithmetic at finite precision

1.1 Representation of integer numbers

An integer i can be represented by the model

$$i = s \times \sum_{k=1}^{q} w_k r^{k-1} \tag{1.1}$$

Here s is the **sign bit**, $r > 1$ is the **base** (usually 2), $q = N - 1$ where N is the total number of bits, and $0 \leq w_k < r$. The largest integer representable with this model is clearly $r^q - 1$. In the remainder of this discussion, we'll assume that $r = 2$ (binary arithmetic).

Addition is performed in the usual fashion. To perform subtraction—addition of a negative number—there is no need to test for the value of s and have a separate algorithm; instead a negative number $-j$ ($j > 0$) is stored as its **complement**, which is $2^q - j$, with the sign bit set to $s = 1$. Hence $i - j = i + (-j) = i + 2^q - j$. We also stipulate that addition acts like an odometer that "rolls over." This scheme automatically sets the sign bit correctly.

For example, suppose we have 4 bits available, so $q = 3$ in eq. (1.1). Then we can represent the positive integers from 0 to $2^q - 1 = 7$ as

```
 0   0000
 1   0001
 2   0010
 3   0011
...   ...
 7   0111
```

Suppose we wish to compute $7 - 1 = 7 + (-1)$. We represent the number -1 as $8 - 1 = 7$ and set the sign bit, giving `1111`. Doing the addition,

```
 7   0111
-1   1111
-----------
 6   0110
```

which is the correct result. Try this scheme for, e.g., $3 - 5$ and verify

that you get the correct result.

EXERCISE 1.1 — What is the output of the following loop?

```
integer :: a, i

a = 1
do i = 0,32
  print '(i3,tr1,i12)',i,a
  a = a+a
end do
```

1.2 *Representation of real numbers*

Let $x \neq 0$ be a real number. Then x is represented by the model (according to the `fortran` standard)

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}. \tag{1.2}$$

Here b is termed the **radix** and p the **digits**: $b, p > 1$. The exponent e lies in the range $e_{\min} \leq e \leq e_{\max}$, where $e_{\min}, e_{\max}$ are the **minimum and maximum exponents**. Finally, $0 \leq f_k < b$ with $f_1 \neq 0$.

EXERCISE 1.2 — In the integer representation of eq. (1.1) for a 32-bit number, what are the largest and smallest representable integers?

EXERCISE 1.3 — Write a program to find the following characteristics of your machine: radix b, digits p, and minimum and maximum exponents $e_{\min}$, $e_{\max}$. Note that `fortran` has built-in functions to return these values. If you wish to use `C++`, you can find analogous functions in the class `numeric_limits<float>`, described in the header file `<limits>`.

EXERCISE 1.4 — In terms of the model parameters b and p, what is the smallest number ε such that $1.0 - \varepsilon \neq 1.0$? What is its decimal value on your machine?

EXERCISE 1.5 — Using the model in equation 1.2, find an expression in terms of b and p for the spacing S between model numbers about $x = 3\pi/4$. Write a code to confirm your answer.

1.3 *Binary storage: which end is up?*

In the previous examples, we have written the numbers with the most-significant bit (the biggest term in the sum, eq. [1.1] and [1.2]) coming first. This is a **big-endian scheme**. Another possible scheme is to store the bits starting with the least significant, a **little-endian** scheme. Both schemes are in use, and this fact, among others, makes files of raw binary data non-portable.

EXERCISE 1.6 — Suppose you could evaluate a function $f(x)$ at points a, $a+h$, and $a+2h$. Construct an approximation to $\mathrm{d}f(x)/\mathrm{d}x|_{x=a}$ that is accurate to $\mathcal{O}(h^2)$.

EXERCISE 1.7 — Compute a numerical approximation to the derivative of $\cos(x)$,

$$\frac{\mathrm{d}\cos(x)}{\mathrm{d}x} \approx \frac{\cos(x+h)-\cos(x)}{h} \tag{1.3}$$

for $h = S \times 2^q$ with S from problem 1.5 and $q = 20, 19, 18, \ldots, 0$. For each q, compute the relative error between the numerical approximation of $\mathrm{d}\cos(x)/\mathrm{d}x$ and $-\sin(x)$. Plot your results, and explain any trends you see.

EXERCISE 1.8 — Now repeat problem 1.7, but this time don't make h a binary multiple of S (for example, make $h = 0.9 \times S \times 2^q$). How do your results differ from those of problem 1.7?

2
Finding Roots

2.1 Introduction

A common numerical task is finding the root(s) x_r of a function $f(x)$ such that $f(x)|_{x=x_r} = 0$. In practice, this means finding x_r to within some tolerance Δ. Recall that it may not be even possible to represent x_r exactly. It is desirable that the algorithm be

- Efficient: it should converge to the desired tolerance in as few a number of function evaluations as possible.
- Robust: if x_r is known to lie in the interval $[x_1, x_2]$, then the algorithm should converge.

As a worked example, suppose we wanted to find the root of $f(x) = \cos(x)$ for $x \in [0, \pi]$ (see Fig. 2.1).

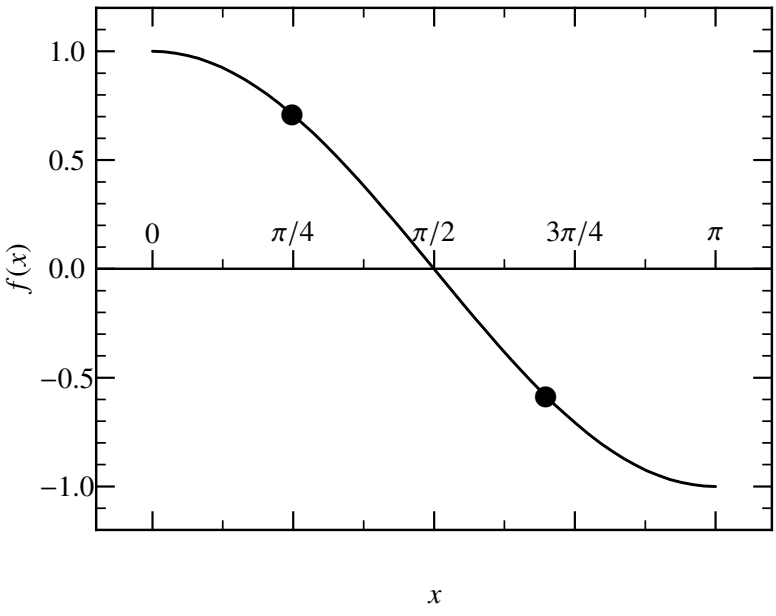


Figure 2.1: Function $f(x) = \cos(x)$ on the interval $x \in [0, \pi]$.

2.2 *Bisection*

One fail-proof method is bisection. If we have the root bracketed between $[x_1, x_2]$, meaning that $f(x_1)$ and $f(x_2)$ have different signs, then the following algorithm will always find a root. Find the midpoint of the interval $x_m = (x_1 + x_2)/2$ and compute $f(x_m)$. Determine the subinterval, either $[x_1, x_m]$ or $[x_m, x_2]$, in which the root lies. Now repeat by finding the midpoint in which the root lies, and so on until the width of the interval is less than the desired tolerance Δ. For example, in Figure 2.1, our guesses are at $x_1 = 0.25\pi$ and $x_2 = 0.7\pi$. The midpoint of this interval is then 0.475π, and our next interval will be $[0.475\pi, 0.7\pi]$.

EXERCISE 2.1 — In the bisection scheme, for a given starting interval size Δ_{initial}, how many iterations are required to reach a given tolerance Δ?

EXERCISE 2.2 — Write a bisection scheme. Find the root $\cos(x) = 0$ on the interval $x \in [0, \pi]$. Plot the error in the bracket as a function of iteration number.

2.3 *Newton's method*

A second, classic, method is due to Newton. On each iteration n, with trial guess x_n, evaluate the function $f(x_n)$ and its first derivative $f'(x_n)$ and compute a new guess x_{n+1} by expanding the function to first order and solving for the correction $\delta x_n = x_{n+1} - x_n$,

$$0 = f(x_{n+1}) \approx f(x_n) + f'(x_n)\delta x_n, \tag{2.1}$$

whence $x_{n+1} = x_n - f/f'|_{x_n}$. For example, in Fig. 2.1, if our initial guess is $x_0 = \pi/4$, then $f(x_0) = \cos(\pi/4) = 1/\sqrt{2}$ and $f'(x_0) = -1/\sqrt{2}$. Hence our next guess for the root would be $x_1 = \pi/4 + \sqrt{2}/\sqrt{2} = (\pi + 4)/4$, which differs from the root, $\pi/2$, by $1 - \pi/4 = 0.21$. This is an improvement from our previous guess. If the method is converging, δx_n will decrease rapidly with n, and one stops when $\delta x_n < \Delta$.

EXERCISE 2.3 — Explain geometrically how Newton's method works.

EXERCISE 2.4 — Derive an expansion series for δx_n, as defined in equation (2.1), in terms of the previous correction δx_{n-1}. Show that the leading order term is $\mathcal{O}(\delta x_{n-1}^2)$, and use this to estimate the number of iterations required to converge a given tolerance Δ.

EXERCISE 2.5 — For the function in Fig. 2.1, what is the smallest value of the initial guess x_0 such that Newton's method converges to $x_r = \pi/2$? What happens if the initial guess is smaller than this?

EXERCISE 2.6 — Write a Newton's rootfind scheme. Test it by finding the root of $\cos(x) = 0$ for $x \in [0, \pi]$. Plot the error in the root as a function of the iteration number.

3
Ordinary Differential Equations

3.1 Background

A common computational task is the solution of ordinary differential equations. To motivate this, consider some number N particles interacting in a potential. The particles obey the equations of motion

$$\frac{\mathrm{d}\boldsymbol{r}_i}{\mathrm{d}t} = \frac{\boldsymbol{p}_i}{m_i}, \tag{3.1}$$

$$\frac{\mathrm{d}\boldsymbol{p}_i}{\mathrm{d}t} = -\boldsymbol{\nabla}\Phi\left(\{\boldsymbol{r}\}\right). \tag{3.2}$$

for $i = 1, \ldots, N$. This is a system of $6N$ ODE's. Some familiar examples for the potential Φ are

$$\text{gravitational,} \qquad \Phi\left(\{\boldsymbol{r}\}\right) = -\frac{1}{2}\sum_{i\neq j}\frac{Gm_i m_j}{|\boldsymbol{r}_i - \boldsymbol{r}_j|};$$

$$\text{molecular,} \qquad \Phi\left(\{\boldsymbol{r}\}\right) = -\frac{\epsilon}{2}\sum_{i\neq j}\left[2\left(\frac{r_0}{|\boldsymbol{r}_i - \boldsymbol{r}_j|}\right)^6 - \left(\frac{r_0}{|\boldsymbol{r}_i - \boldsymbol{r}_j|}\right)^{12}\right];$$

$$\text{or screened coulomb,} \qquad \Phi\left(\{\boldsymbol{r}\}\right) = -\frac{1}{2}\sum_{i\neq j}\frac{Z_i Z_j e^2}{|\boldsymbol{r}_i - \boldsymbol{r}_j|}\exp\left(-|\boldsymbol{r}_i - \boldsymbol{r}_j|/r_0\right).$$

The gravitational interaction gives us Kepler's problem and for large N models of globular clusters. The Lenard-Jones (molecular) and Yukawa (screened coulomb) potentials are common in molecular dynamics problems.

The Lane-Emden Equation

A classic problem in stellar evolution is the construction of *polytropic* stellar models. One makes the *ansatz* that the pressure P is related to the density ρ via

$$P(r) = K\rho^{1+1/n}(r) \tag{3.3}$$

where n and K are constants. Further define the dimensionless variable θ via

$$\rho(r) = \rho_c \theta^n(r), \tag{3.4}$$

where the subscript c denotes the central value at $r = 0$. Substituting these definitions into Poisson's equation,

$$\nabla^2 \Phi = 4\pi G\rho, \tag{3.5}$$

and the equation for hydrostatic equilibrium,

$$\boldsymbol{\nabla} P = -\rho \boldsymbol{\nabla} \Phi, \tag{3.6}$$

we obtain the *Lane-Emden* equation for index n,

$$\xi^{-2} \frac{\mathrm{d}}{\mathrm{d}\xi} \left(\xi^2 \frac{\mathrm{d}\theta}{\mathrm{d}\xi} \right) = -\theta^n. \tag{3.7}$$

Here $\xi = r/r_n$ is the dimensionless coordinate, and

$$r_n = \left[\frac{(n+1)P_c}{4\pi G \rho_c^2} \right]^{1/2} \tag{3.8}$$

is the radial length scale.

For a stellar model, we have the following boundary conditions,

$$\theta(\xi)\big|_{\xi=0} = 1, \tag{3.9}$$

$$\theta'(\xi)\big|_{\xi=0} = 0. \tag{3.10}$$

From the form of equation (3.7), it follows that $\theta(-\xi) = \theta(\xi)$, that is, the solution is *even*. A power-series solution to θ out to order ξ^6 is

$$\theta(\xi) = 1 - \frac{1}{6}\xi^2 + \frac{n}{120}\xi^4 - \frac{n(8n-5)}{15120}\xi^6 + \mathcal{O}(\xi^8) \tag{3.11}$$

Finally, there are analytical solutions for $n = 0$, 1, and 5. In particular,

$$\theta_0(\xi) = 1 - \frac{\xi^2}{6} \tag{3.12}$$

$$\theta_1(\xi) = \frac{\sin \xi}{\xi}. \tag{3.13}$$

We will use these analytical solutions to verify the ordinary differential equation solver in the project. The location of the first zero, ξ_1, is taken as the "radius" for the stellar model. For example, if $n = 0$ (eq. [3.12]), $\xi_1 = \sqrt{6}$.

TO SOLVE THE LANE-EMDEN EQUATION NUMERICALLY, first decompose the Lane-Emden equation (eqs. [3.7], [3.9], and [3.10]) into two first-order differential equations. Define

$$t = \xi \tag{3.14}$$

$$y = \theta \tag{3.15}$$

$$z = \frac{\mathrm{d}\theta}{\mathrm{d}\xi}. \tag{3.16}$$

With this decomposition, we now have two coupled first-order ODEs,

$$y' = z \tag{3.17}$$
$$z' = -\frac{2}{t}z - y^n \tag{3.18}$$

with boundary conditions

$$y(0) = 1 \tag{3.19}$$
$$z(0) = 0. \tag{3.20}$$

NB: This system is indeterminate at $t = 0$. To get around this, you can start the integration at $t = h$, using the expansion (eq. [3.11]).

The two-body gravitational problem

Another classic problem is Kepler's: two point masses interacting via a gravitational potential. For this problem, we can solve it analytically, which will give us something to compare our numerical solutions against. Let's first recall some analytical properties of our 2-body problem. We'll set $G = 1$ to make our equation simpler. In doing these calculations, it is good practice to scale quantities so that they are of order unity. The final numerical results can then be scaled to CGS values when we wish to extract physical values.

By boosting to a center-of-mass frame, we can reduce our problem to that of a single particle of mass $\mu \equiv m_1 m_2/(m_1 + m_2)$ obeying the equations of motion

$$\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t} = \frac{\boldsymbol{p}}{\mu}, \qquad \frac{1}{\mu}\frac{\mathrm{d}\boldsymbol{p}}{\mathrm{d}t} = -\frac{M}{r^3}\boldsymbol{r}, \tag{3.21}$$

where $M = m_1 + m_2$ and $\boldsymbol{r} = \boldsymbol{r}_2 - \boldsymbol{r}_1$. If the total energy $E = p^2/(2\mu) - M/r < 0$, then the motion is in an ellipse, with semi-major axes,

$$a = -\frac{M\mu}{2E}, \qquad b = \frac{L}{(2\mu E)^{1/2}} \tag{3.22}$$

Here $L = \mu \boldsymbol{r} \times \boldsymbol{p} = \text{const}$ is the angular momentum. The period of the orbit is

$$T = \frac{\pi}{\sqrt{2}} M \mu^{3/2} |E|^{-3/2} \tag{3.23}$$

From this solution, we can get back the positions of our two masses via the equations

$$\boldsymbol{r}_1 = -\frac{m_2}{M}\boldsymbol{r}, \qquad \boldsymbol{r}_2 = \frac{m_1}{M}\boldsymbol{r},$$

where we take the origin to be at the center-of-mass.

3.2 *A worked example: Development of a two-body code*

Let's build a lightweight code to integrate the equations of motion. Here we'll break the code into two pieces: a *module* that will hold all of the "tools" needed to do the calculation, and a driver routine that sets up the problem (perhaps taking input from the user) and the calls upon the tools in the module to do the work.

How should we integrate our equations (3.21)? To start, suppose we have a solution $z = (\boldsymbol{r}, \boldsymbol{p})$ at time t. Here z denotes a point in our $6N$-dimensional phase space. We can approximate the solution at a slightly later time $t + h$ as

$$z(t+h) = z(t) + h \left.\frac{\mathrm{d}z}{\mathrm{d}t}\right|_t + \frac{1}{2!}h^2 \left.\frac{\mathrm{d}^2 z}{\mathrm{d}t^2}\right|_t + \ldots$$

But, we have expressions for $\mathrm{d}z/\mathrm{d}t$: these are just the right-hand sides of equations (3.21). So the simplest expression, accurate to $\mathcal{O}(h^2)$ is to approximate our integral by a sequence of steps

$$z(t+h) \approx z(t) + h \left.\frac{\mathrm{d}z}{\mathrm{d}t}\right|_t .$$

To discretize this equation, let $t_n = t_0 + nh$, and let $z_n = z(t_n)$. Denoting $\mathrm{d}z/\mathrm{d}t|_{t_n} = f(z_n)$, we arrive at our equation,

$$z_n = z_{n-1} + hf(z_{n-1}), \tag{3.24}$$

where $f(z)$ is the right-hand side of equations (3.21). This method is called *forward Euler*. It is not all that accurate, but it makes a good starting point for developing our methods.

EXERCISE 3.1 — Suppose we have an ODE

$$\frac{\mathrm{d}u}{\mathrm{d}t} = f(t, u). \tag{3.25}$$

Given a set of solutions $\varphi_k = u(t = k \times h),\ k = 0, \ldots, n$, construct an approximate solution φ_{n+1} at time $t_{n+1} = (n+1)h$, where h is the stepsize, by the relation

$$\varphi_{n+1} = \varphi_n + h\left[af(t_n, \varphi_n) + bf(t_{n-1}, \varphi_{n-1})\right], \tag{3.26}$$

where a and b are constants. Find a and b such that φ_{n+1} agrees to $u[t = (n+1)h]$ to order h^2 if $\varphi_n = u(t = nh)$ and $\varphi_{n-1} = u[t = (n-1)h]$. If you did this correctly, you will have derived the second-order *Adams-Bashforth* method.

Truncation Error

The forward Euler method is a **first-order** method. The truncation error on each step is $\mathcal{O}(h^2)$. To integrate over a fixed interval T takes T/h steps, making the global error $\mathcal{O}(h)$.

Suppose we take a *predictor step* to the midpoint of the interval,

$$z^{\rm p}_{n+1/2} = z_n + \frac{h}{2} f(t_n, z_n), \tag{3.27}$$

and then use $z^{\rm p}_{n+1/2}$ in a midpoint *corrector step*,

$$z_{n+1} = z_n + h f(t_{n+1/2}, z^{\rm p}_{n+1/2}). \tag{3.28}$$

One can show (see Exercise 3.2) that z_{n+1} agrees with the solution to $\mathcal{O}(h^2)$; this procedure is a second-order **Runge-Kutta** method.

EXERCISE 3.2 — Show that the scheme in equations (3.27)–(3.28) gives an approximate solution φ_{n+1} that agrees with the exact solution $u[t = (n+1)h]$ to order h^2.

Note that there is more than one way to construct a scheme of a given order. For example, we might split the steps as follows. For $z_n = (r_n, p_n)$, compute z_{n+1} via

$$\begin{aligned} r_{n+1/2} &= r_n + \frac{h}{2}\frac{p_n}{m}, \\ p_{n+1} &= p_n - \frac{h}{m}\left(\frac{\partial \Phi}{\partial r}\right)_{r_{n+1/2}}, \\ r_{n+1} &= r_{n+1/2} + \frac{h}{2}\frac{p_{n+1}}{m}. \end{aligned} \tag{3.29}$$

This is known as the **Verlet** or **leapfrog** method.

Project Organization

In general, it is usually good to organize the effort *before* coding; imposing order is more difficult when there is already a lot of files lying about. What kind of files will we have? First, we will have files containing the fortran source code. It might be desirable to keep those in a separate directory—let's call it `source`. Second, the fortran compiler will generate many extra files as it builds the code, and if there are a lot of source files, we'd like to have a script that controls the build. Most of these files are clutter, so let's put them into a directory `build`. Finally, of course, we'll want to run our code and produce graphs, so let's make a third directory, `exec`, which will hold the executable as well as any analysis. All three of these directories will reside in a top-level project directory, `2body`.

Note that the way I am laying things out is not the only way to do things, and I make no assertion that it is the best way. But it is important to have some organization, or else you will waste too much time and energy.

Code Organization

Having organized our directory, we now need to think about the structure of our code. What features should it have? Again, there is more than one way to do this. Let's first look at our module. We'll put this into a file `gravbody.f`. Because we'll have a lot of related routines and data, we'll enclose them into a single within the statements

```
7   module gravbody

108 end module gravbody
```

In the driver program, which we'll put in a file `orbits.f`, the statement

```
1   program orbits
2           use gravbody
```

will make all of the routines defined in `gravbody` accessible by the calling program.

Now to fill in structure of our `gravbody` module. First, it might be nice to group our data for a particle (mass, position, and velocity) into one structure. Fortran allows us to define a data type.

```
14          type body
15                  real :: M
16                  real, dimension(Ndim) :: r, v
17          end type body
```

This allows us to hold all of the information about our pseudo particle in one place. Note that `Ndim` is a parameter that will need to be specified elsewhere. (We could make this a calculation in 3 dimensions, but we know from the equations that the orbit is in a plane.) We can pass a variable of type `body` to functions to compute the acceleration, kinetic energy, or potential energy. In this type, we will use the member `M` to denote the total mass. The member variables are accessed in the following way: if `pt` is a variable of type `body`, then `pt%M` accesses the component `M` of `pt`.

Now we can add to our module functions that operate on `body`. First, let's introduce a helper function to compute the norm (length) of a vector; we'll need this to compute both the acceleration and the potential energy.

```
19          contains
20          function norm(x) result(nx)
21                  real, dimension(:), intent(in) :: x
22                  real :: nx
23                  nx = sqrt(dot_product(x,x))
24          end function norm
```

The `contains` command must precede all of the subroutines in the module. We can then compute the acceleration,

```
        function acceleration(pt) result(a)
                type(body), intent(in) :: pt
                real, dimension(Ndim) :: a
                real :: r3

                r3 = norm(pt%r)**3
                a = -pt%M*pt%r/r3
        end function acceleration
```

the kinetic energy per unit reduced mass,

```
        function kinetic_energy(pt) result(K)
        ! kinetic energy per unit of reduced mass
                type(body), intent(in) :: pt
                real :: K

                K = 0.5*dot_product(pt%v,pt%v)
        end function kinetic_energy
```

and the potential energy per reduced mass,

```
        function potential_energy(pt) result(Phi)
        ! potential energy per unit of reduced mass
                type(body), intent(in) :: pt
                real :: Phi

                Phi = -pt%M/norm(pt%r)
        end function potential_energy
```

We've chosen to work in the center of mass frame, so we don't actually need to track the masses of the individual objects; we just need the total mass to get the acceleration. That is why we compute the kinetic energies *per reduced mass*; for comparison with a physical system we can easily just scale our numerical results.

GIVEN THE ACCELERATION $\boldsymbol{a}(\boldsymbol{r})$, HOW DO WE INTEGRATE OUR EQUATIONS OF MOTION? Let's start with the forward Euler algorithm (eq. [3.24]). Given our initial conditions $\boldsymbol{r}_0$ and $\boldsymbol{v}_0$, we compute

$$\begin{aligned} \boldsymbol{r}_1 &= \boldsymbol{r}_0 + h\boldsymbol{v}_0, \\ \boldsymbol{v}_1 &= \boldsymbol{v}_0 + h\boldsymbol{a}(\boldsymbol{r}_0). \end{aligned}$$

This is our approximation to the solution at $t = h$. We then repeat this step and march our solution along. Our function will take two arguments: a variable of type `body`, which will be modified by the routine, and a stepsize `dt`.

```
68          subroutine forward_Euler(pt, dt)
69                  type(body), intent(inout) :: pt
70                  real, intent(in) :: dt
71                  real, dimension(Ndim) :: a
72
73                  a = acceleration(pt)
74                  pt%r = pt%r + pt%v*dt
75                  pt%v = pt%v + a*dt
76          end subroutine forward_Euler
```

One further piece of organization. I might want to run this code with a different stepsize algorithms. I therefore provide some tags,

```
10          integer, parameter :: fEuler = 1
11          integer, parameter :: leapfrog = 2
12          integer, parameter :: rk2 = 3
```

and a wrapper to switch routines,

```
51          subroutine step(pt, dt, method)
52                  type(body), intent(inout) :: pt
53                  real, intent(in) :: dt
54                  integer, intent(in) :: method
55
56                  select case (method)
57                  case (fEuler)
58                          call forward_Euler(pt,dt)
59                  case (leapfrog)
60                          call verlet(pt,dt)
61                  case (rk2)
62                          call runge_kutta_2(pt,dt)
63                  case default
64                          stop 'need to code your method first!'
65                  end select
66          end subroutine step
```

Here `leapfrog` and `rk2` are some other integration routines that we'll cover later. The routine `step` is a "generic" wrapper that selects the method to use.

Now that we have our module, let's look at our driver program. What should it do? There are basically 4 things:

1. read in parameters controlling the run;
2. set the initial conditions;
3. integrate over the desired time; and
4. print diagnostics

Let's take these in order. We will want to run the code with different initial conditions, different stepsizes, different integration times, and so on. We don't want to recompile our code every time; instead, it would be good if we could read in the parameters of our code from a file. Fortran provides a lightweight means to do this using a *namelist*,

```
1  program orbits
2          use gravbody
3
4          type(body) :: pt
5          real :: x0, v0, m
6          real :: K, Phi, E, Einit
7          real :: dt, dtdiag, dtprint, tend, tdiag, tprint, t
8          integer :: method
9          character(len=*),parameter :: inputfile = 'twobody.in'
10
11         ! set up the run: we'll use a namelist to input parameters
12         namelist /twobody/ dt,dtdiag,dtprint,tend,method
13         namelist /init/ m,x0,v0
14         open (unit=10,file=inputfile,status='old')
15         read(10,nml=twobody)
16         read(10,nml=init)
17         close(10)
```

Lines 12 and 13 define two namelists, `twobody` and `init`, which contain our run-time parameters. We then, in line 14, open the file "twobody.in"; the flag `status='old'` will cause the code to return an error if the file doesn't exist. We then read in the namelists and close the file. Here is the sample file `twobody.in`.

```
1  ! methods:        1 forward euler
2  !                       2        leapfrog
3  !                       3        runge-kutta 2nd order
4  &twobody
5          dt = 0.01,
6          dtdiag = 1.0,
7          dtprint = 0.1,
8          method = 2,
9          tend = 10.0 /
10 &init
11         m = 4.0,
12         x0 = 1.0,
13         v0 = 1.75 /
```

Having read in our run-time parameters, we can initialize our variables,

```
19         t = 0.0
```

```
        tprint = 0.0
        tdiag = 0.0

        ! initialize the body
        pt%M = m
        pt%r = (/ x0, 0.0 /)
        pt%v = (/ 0.0, v0 /)

        ! compute initial energy
        K = kinetic_energy(pt)
        Phi = potential_energy(pt)
        Einit = K + Phi
        E = Einit
```

Now we are ready to set up a loop that will step our solution forward. Along the way, we'll want to print out some diagnostics. We don't necessarily want to do this after every step, so let's define two time intervals, `dtprint`, which is the time interval between printing out the coordinates, and `dtdiag`, which is the time interval between printing out the energy. We'll keep track of the time at which we last printed out these diagnostics. A simple loop that runs until the time exceeds `tend` is

```
        print '(A,es11.4,A,f8.5)', 'evoloving with dt = ', dt,'; tend = ',tend
        call pretty_print_energy
        call print_coordinates
        do
                if (tend-t < dt) dt = tend-t
                call step(pt,dt,method)
                t = t + dt
                if (t >= tend) exit
                if (t-tprint >= dtprint) then
                        call print_coordinates
                end if
                if (t-tdiag >= dtdiag) then
                        call pretty_print_energy
                end if
        end do
        call print_coordinates
        call pretty_print_energy
```

Notice the form of the do-loop: it runs until the exit condition

```
                if (t >= tend) exit
```

is reached. The routines `print_coordinates` and `pretty_print_energy` are defined within the program.

```
        contains
        subroutine print_coordinates
                print '(f8.5,2(tr4,2(f8.5,tr1)))',t,pt%r,pt%v
                tprint = t
        end subroutine print_coordinates

        subroutine pretty_print_energy
                K = kinetic_energy(pt)
                Phi = potential_energy(pt)
                E = K + Phi
                print '(64("="),/,A,3(f8.5,tr1),es11.4,/,64("="))', &
                        & 'K,Phi,E,err = ',K,Phi,E,abs((E-Einit)/Einit)
                tdiag = t
        end subroutine pretty_print_energy

end program orbits
```

Because these routines are "contained" in the program, they can access all of the variables used in the program.

Building the code

With the code complete, we are now ready to compile it. Of course, we can just do this by typing the appropriate command, such as `gfortran`. But for complex codes with tens to hundreds of files, this becomes unwieldy. It is also inefficient to recompile everything if just one file changed. A way to organize this is to use the `make` utility or an integrated development environment, such as CodeWarrior™ or Xcode™. Here I'll illustrate the use of `make`, which is a scripting tool for compilation.

In the build directory, we create a file, `makefile`. The syntax is a bit cumbersome, but is fairly self-explanatory. First, I define macros that tell where the various directories are,

```
# LAYOUT
PROJECT_DIR = ..
EXEC_DIR = $(PROJECT_DIR)/exec
SRC_DIR = $(PROJECT_DIR)/source
VPATH = $(SRC_DIR)
```

The `VPATH` macro is internal to `make` and is a list of directories to search when looking for source code. I then give a list of macros that control options for compiling the code.

```
# COMPILER
FC = gfortran

# COMPILER FLAGS
```

```
FCimpno = -fimplicit-none
FCreal = -fdefault-real-8
FCchecks = -fbounds-check
FCwarn = -Wunused-value -Werror -W
FCfixed = -ffixed-form -ffixed-line-length-132
FCfree = -ffree-form
FCopt = -O2
FCdebug = -g

INCLUDES = -I$(SRC_DIR)

COMPILE_BASE = $(FC) $(INCLUDES) $(FCimpno) $(FCreal) $(FCfree) -c
COMPILE_TEST = $(COMPILE_BASE) $(FCwarn) $(FCchecks) $(FCopt)

COMPILE = $(COMPILE_TEST)
```

Last set of macros; the desired name of the executable and the *object* files. These are intermediate files that contain the machine language instructions, but aren't yet linked into an executable file.

```
EXEC = twobody
OBJS =  gravbody.o orbits.o
```

Now we're ready for the instructions to build the code. First, we tell `make` how to create an object file (`.o` suffix) from a fortran file (`.f` suffix) of the same name.

```
%.o:%.f
        $(COMPILE) $<
```

Rules for compilation all have this form

```
target:<dependencies>
        <instructions to make target from dependencies>
```

In the build directory, issue the command `make gravbody.o`. What happens? Make sees the rule `%.o:%.f` and therefore looks for a file the the name `gravbody.f`, first in the current directory, and then in the directories specified in the `VPATH` macro. It then expands the macro `$(COMPILE)` and operates that command on the dependencies (the `$<` macro). What happens if you type `make gravbody.o` a second time? This time nothing happens, because the source file, `gravbody.f`, has not changed since `gravbody.o` was last made. Only when the dependencies are newer than the target (or if the target doesn't exist) are the instructions for building the target executed.

The next rule gives the instructions for making the full executable, which we named `twobody`.

```
33 $(EXEC): $(OBJS)
34         $(FC) -o $@ $(OBJS)
```

In order to build `twobody`, we require that make first build `gravbody.o` and `orbits.o`; it will do this as needed using the rule we defined. In the next line, the `-o $@` directive tells the compiler to put the executable in a file with the target name (`$@` is a make macro for the current target).

The last two targets,

```
36 install: $(EXEC)
37         cp $(EXEC) $(EXEC_DIR)
38
39 clean:
40         -@rm -f *.o *.mod $(EXEC)
```

don't refer to files at all, so they are always executed. The first target, `install`, will force a build of the executable if necessary, and then copies into our specified location. The second removes all of the build files. Note that in addition to the `*.o` files, the compilation makes a module file `gravbody.mod`. This contains information about the routines in that module, and is used when compiling `orbits.o`. As a result, we have to have `gravbody.o` compiled before `orbits.o`.

3.3 Symplectic Integration

When you run the code with the forward Euler method, you will find that the scheme does not preserve energy well. in fact, if your timestep is too large, you will find your system becoming unbound! In general, we must go to higher-order methods, i.e., methods for which on a given step the approximate solution matches the Taylor series out to $\mathcal{O}(h^n)$, where $n > 2$. Before doing this, however, let's take a nice detour to show some algorithms, called *sympletic integrators*, that have much better energy conserving properties.

To develop a set of sympletic integrators, let's recall some facts from quantum mechanics. Recall that the time evolution of a state is given as

$$|\Psi(t)\rangle = e^{-iHt/\hbar} |\Psi(0)\rangle$$

if the Hamiltonian operator H does not depend on time explicitly. Since H is Hermitian, we can from the corresponding row vector,

$$\langle\Psi(t)| = \langle\Psi(0)| e^{iHt/\hbar}.$$

Given an operator A that doesn't explicity depend on time, the expectation value of A at time t is

$$\langle A\rangle = \langle\Psi(t)|A|\Psi(t)\rangle = \left\langle \Psi(0) \left| e^{iHt/\hbar} A e^{-iHt/\hbar} \right| \Psi(0) \right\rangle;$$

this defines the time-dependent operator $A(t) = e^{iHt/\hbar}Ae^{-iHt/\hbar}$. Now take the time derivative of $A(t)$ to obtain

$$\begin{aligned} i\hbar\frac{\partial}{\partial t}A(t) &= -H\left(e^{iHt/\hbar}Ae^{-iHt/\hbar}\right) + \left(e^{iHt/\hbar}Ae^{-iHt/\hbar}\right)H \\ &= A(t)H - HA(t) = [A(t), H]\,. \end{aligned}$$

Recall that the commutator divided by $i\hbar$ corresponds to a Poisson bracket in classical mechanics, and we have a formalism for evolving the phase space vector $z = (q, p)$ in time:

$$\dot{z} = \{z, H\}\,. \tag{3.30}$$

Here

$$\{A, B\} = \sum_i \frac{\partial A}{\partial q_i}\frac{\partial B}{\partial p_i} - \frac{\partial A}{\partial p_i}\frac{\partial B}{\partial q_i}$$

is the Poisson bracket.

Formally, we may define an operator $\mathcal{D}_H(a) = \{a, H\}$ so that equation (3.30) becomes

$$\dot{z} = \mathcal{D}_H(z).$$

which has the formal solution $z(t) = \exp(t\mathcal{D}_H)z(0)$. Right now this is just formalism, but this expression is a *canonical transformation*. Why is that important? It can be shown that such a transformation preserves volumes of phase space along the trajectory followed by the system. This leads to a way to preserve better energy conservation properties into our system of equations.

Now let $t \to h$, where h is some very small interval of time. Furthermore, suppose that $H(q, p) = V(q) + K(p)$, i.e., the Hamiltonian has a kinetic term that depends only on the momenta and a potential term that depends only on the coordinates. Then $\mathcal{D}_H$ separates into $\mathcal{D}_H = \mathcal{D}_V + \mathcal{D}_K$. We can therefore write our solution at time h as

$$z(h) = \exp(h\mathcal{D}_V)\exp(h\mathcal{D}_K)z \approx (1 + h\mathcal{D}_V)\,(1 + h\mathcal{D}_K)\,z(0).$$

Now, $\mathcal{D}_K z = \{z, \mathcal{D}_K\} = (\partial z/\partial q)(\partial K/\partial p) = (p/\mu)$. Hence, the action of the operator $1 + h\mathcal{D}_K$ is to perform the mapping,

$$(q, p) \to \left(q(h) = q(0) + h\frac{p(0)}{\mu}, p(0)\right).$$

Since $p/\mu = v$, this just looks like the first-order Euler step for the position q. Now we can feed this result into the second term,

$$(1 + h\mathcal{D}_V)\,(q(h), p(0)) = \left(q(h), p(h) = p(0) - h\left.\frac{\partial V}{\partial q}\right|_{q(h)}\right).$$

Here $-\partial V/\partial q$ is just the force, so this second stage is an Euler step for the momentum. Notice, however, that we evaluated the force not at $q(0)$,

the position at the beginning of the step, but rather at $q(h)$, the updated position. It may be hard to believe, but this small change makes a huge difference in how well the scheme conserves energy!

To construct a higher order method, construct an operator

$$\prod_{i=1}^{k} \exp(c_i h\mathcal{D}_V) \exp(d_i h\mathcal{D}_K),$$

where the c_i and d_i are coefficients chosen to that this operator agrees with the exact, formal operator $\exp(h(\mathcal{D}_V + \mathcal{D}_K)$ to the desired order in h. Each of these operators is a canonical transformation, so the product For example, let $k = 2$ and expand our operators to second-order in h,

$$\begin{aligned}
&\prod_{i=1}^{k} \exp(c_i h\mathcal{D}_V) \exp(d_i h\mathcal{D}_K) \\
&\quad\approx\quad (1 + c_1 h\mathcal{D}_V)(1 + d_1 h\mathcal{D}_K)(1 + c_2 h\mathcal{D}_V)(1 + d_2 h\mathcal{D}_K) \\
&\quad\approx\quad 1 + (c_1 + c_2)h\mathcal{D}_V + (d_1 + d_2)h\mathcal{D}_K \\
&\qquad\quad +(c_1 d_1 + c_2 d_2)h^2\mathcal{D}_V\mathcal{D}_K + d_1 c_2 h^2\mathcal{D}_K\mathcal{D}_V. \qquad (3.31)
\end{aligned}$$

Keep in mind that $\mathcal{D}_V$ and $\mathcal{D}_V$ do not commute; further, you can verify that $\mathcal{D}_V^2 = \mathcal{D}_K^2 = 0$. Comparing equation (3.31) with the formal expansion

$$\exp[h(\mathcal{D}_V + \mathcal{D}_K)] \approx 1 + h(\mathcal{D}_V + \mathcal{D}_K) + \frac{h^2}{2}\left(\mathcal{D}_V\mathcal{D}_K + \mathcal{D}_K\mathcal{D}_V\right),$$

we see that by choosing $c_1 = c_2 = 1/2$, $d_1 = 1$, and $d_2 = 0$, we will obtain the desired method.

Let's translate this second-order method to an algorithm. We'll define the velocity $v = p/\mu$ and the acceleration $a = -(1/\mu)\partial V/\partial q$. Then our algorithm becomes

$$\begin{aligned}
q_1 &= q_0 + \frac{h}{2}v_0 \\
p_2 = p_1 &= p_0 + ha_1 \\
q_2 &= q_1 + \frac{h}{2}v_1. \qquad (3.32)
\end{aligned}$$

This is known as the *Verlet* or *leapfrog* method. There are systematic methods for creating higher-order methods.

3.4 *The Fourth Order Runge-Kutta Method*

A widely used method in the fourth-order Runge-Kutta algorithm. A step is taken to a midpoint, and this is used to extend the integration across the step. Given a set of first order equations $\mathrm{d}y/\mathrm{d}t = f(t, y)$ the solution $y_{n+1} = y(t + h)$ is determined from $y_n = y(t)$ as follows. Define

$$k_1 = f(t, y_n) \qquad (3.33)$$

$$k_2 = f\left(t + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \tag{3.34}$$

$$k_3 = f\left(t + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \tag{3.35}$$

$$k_4 = f(t + h, y_n + hk_3)\,. \tag{3.36}$$

in terms of which

$$y_{n+1} = y_n + \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right). \tag{3.37}$$

3.5 *Convergence and Error*

One method of quantifying the error in our calculation is based on *Richardson* extrapolation [Richardson and Gaunt, 1927]. To motivate this, suppose we wanted to compute the derivative of a function $f(x)$ by finite difference. We could use our central difference approximation

$$f'_{\rm cd} = \frac{f(x+h) - f(x-h)}{2h} = f'(x) + g(x)h^2 + \mathcal{O}(h^4). \tag{3.38}$$

Here $g(x) = f'''(x)/3!$ and is by assumption unknown to us. Suppose we use equation (3.38) twice, once with a "coarse" $h_{\rm c}$ and once with a "fine" $h_{\rm f}$, i.e., $h_{\rm f} < h_{\rm c}$. We would then have

$$f'_{\rm c} = f'(x) + g(x)h_{\rm c}^2 + \mathcal{O}(h_{\rm c}^4), \tag{3.39}$$

$$f'_{\rm f} = f'(x) + g(x)h_{\rm f}^2 + \mathcal{O}(h_{\rm f}^4). \tag{3.40}$$

By subtracting $f'_{\rm f}$ from $f'_{\rm c}$, we can eliminate the (unknown) exact solution $f'(x)$ and obtain

$$g(x) = \frac{f'_{\rm c} - f'_{\rm f}}{h_{\rm c}^2 - h_{\rm f}^2} + \mathcal{O}(h_{\rm c}^4). \tag{3.41}$$

Substituting this approximation for $g(x)$ back into equation (3.40), we then have an approximation for the exact solution $f'(x)$,

$$f'(x) \approx \frac{(h_{\rm c}/h_{\rm f})^2 f'_{\rm f} - f'_{\rm c}}{(h_{\rm c}/h_{\rm f})^2 - 1} + \mathcal{O}(h^4), \tag{3.42}$$

that is good to fourth order!

Just for amusement, we can use this technique, as Richardson did, to approximate the value of π. Imagine a circle of unit diameter, so that its circumference is π. Now imagine we inscribe a square in this circle. Pythagorus would compute the circumference of this square as $C_4 = 2\sqrt{2}$, if he could believe in irrational numbers. Now inscribe a hexagon in this circle. Since a hexagon is made of six equilateral triangles, we can see that it must have a circumference of $C_6 = 3$. For our order parameter, we can use $h_{\rm c} = 1/4$ and $h_{\rm f} = 1/6$. Then applying equation (3.42) gives us an approximation,

$$\pi \approx \frac{(3/2)^2 3 - 2\sqrt{2}}{(3/2)^2 - 1} = \frac{27 - 8\sqrt{2}}{5} \approx 3.137,$$

which is good to 0.1%. If you want to practice your mental arithmetic skills, you can try computing the above, including extracting $\sqrt{2}$, in your head.

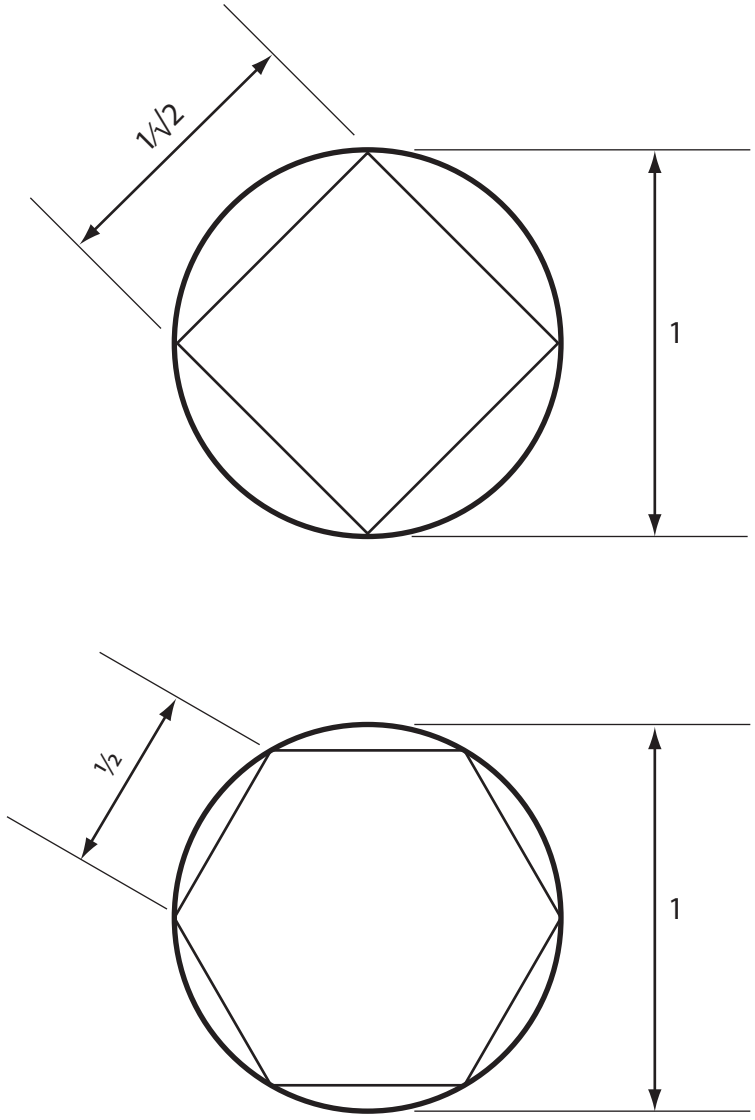


Figure 3.1: Constructing an estimate of π. The circle shown has a circumference of π. The inscribed circle, with vertex angle $2\pi/4$, has circumference $2\sqrt{2}$. The inscribed hexagon, with vertex angle $2\pi/6$, has circumference 3.

In practice, it is better to use the difference between a coarse and fine solution as an estimate of the error, rather than attempt to improve the solution. We don't have any precise way of determining the error in the Richardson extrapolation itself; moreover, *higher order doesn't necessarily mean better accuracy*.

As an example, suppose we are integrating the Lane-Emden equation

using our fourth-order Runge-Kutta scheme. We could make the following argument. Suppose we try a step with size $2h$; then

$$y_c = y(t + 2h) + (2h)^5\varphi + \mathcal{O}(h^6). \tag{3.43}$$

Here $y(t + 2h)$ is the exact solution, y_c is our approximate numerical solution for the coarse stepsize, and φ is some number. If we now repeat the step with a fine stepsize h, we get a different numerical solution,

$$y_f = y(t + 2h) + 2(h^5)\varphi + \mathcal{O}(h^6). \tag{3.44}$$

Subtracting eq. (3.43) from eq. (3.44) gives us an estimate of the error,

$$\Delta = y_f - y_c = -30h^5\varphi + \mathcal{O}(h^6).$$

Thus the quantity

$$y_f + \frac{\Delta}{15} = y(t + 2h) + 2(h^5)\varphi - 2h^5\varphi + \mathcal{O}(h^6)$$

should be of order h^6.

This *local extrapolation* looks good, but there is no way to guarantee that it actually works! The quantity φ may not be the same for the coarse and the fine solutions, for example; there is simply no way to know. Moreover, we lose our quantification of the error in doing this. A better approach is to use Δ as an indication of the error, and then adjust the stepsize h to reduce Δ to a desired amount. This is expensive, since we are now taking 8 function evaluations for one step. More sophisticated approaches are detailed in Press et al. [2007].

EXERCISE 3.3 — Write a 4th-order Runge-Kutta integrator using a fixed stepsize h, to solve the Lane-Emden equation (§ 3.1) for an arbitrary index n. Compute the error $\mathcal{E} = \|t_1 - \xi_1\|$ in the radii as a function of decreasing stepsize h for the analytical solutions, eq. (3.12) and (3.13). One way to compute t is to find where $y(t)$ changes sign, and then use these two steps as brackets to find the root $y(t) = 0$. Since the Runge-Kutta scheme gives you the function and derivative at the two bracketing points (call them t_+ and t_-, with $t_- = t_+ + h$ and $y(t_+) > 0$ and $y(t_-) < 0$), you can use bisection to find the root where $y(t_+ + xh) = 0$, with $0 \le x \le 1$. To evaluate $y(t_+ + xh)$, use the interpolation formula [Press et al., 2007]

$$\begin{aligned} y(t_+ + xh) &= (1 - x)y(t_+) + xy(t_-) + x(x - 1) \\ &\quad \times [(1 - 2x)(y(t_-) - y(t_+)) + (x - 1)hz(t_+) + xhz(t_-)]\,. \end{aligned} \tag{3.45}$$

The error in this interpolation formula is $\mathcal{O}(h^4)$. Plot $\mathcal{E}(h)$, and verify that your code converges as expected.

4
Ordinary Differential Equations and Stiffness

The following discussion is based on the review paper by Cash [2003].

4.1 A single ODE

Consider the equation

$$\frac{\mathrm{d}u}{\mathrm{d}t} = -\lambda u \tag{4.1}$$

with $\lambda > 0$ a constant. Suppose you wish to solve this by forward Euler differencing,

$$u_n = u_{n-1} + h \left.\frac{\mathrm{d}u}{\mathrm{d}t}\right|_{n-1} = u_{n-1} - h\lambda u_{n-1}. \tag{4.2}$$

where u_n is the solution at $t = n \times h$. Then given the initial condition $u_0 = u(t = 0)$, we have

$$\begin{aligned} u_1 &= (1 - h\lambda)u_0, && (4.3)\\ u_2 &= (1 - h\lambda)u_1 = (1 - h\lambda)^2 u_0 && (4.4)\\ &\dots && (4.5)\\ u_n &= (1 - h\lambda)^n u_0. && (4.6) \end{aligned}$$

In order to have $u_n \to 0$ as $n \to \infty$, we must have $|1 - h\lambda| < 1$, or

$$0 < h < 2/\lambda. \tag{4.7}$$

Note that this holds no matter the initial condition u_0. Even if we take many tiny steps until our solution has become quite small, taking steps with too large a size leads to numerical instability. This condition holds even in the general case $z' = \lambda z$ with $\lambda \in \mathbb{C}, \Re\lambda < 0$. In this case, $h\lambda$ must lie in a unit circle (in the complex plane) centered on $z = -1$. Schemes such as the forward Euler are known as *explicit* schemes, because the solution at step n is given as a function of previous steps.

In contrast, suppose we used *backward Euler* differencing,

$$u_n = u_{n-1} - h\lambda u_n, \tag{4.8}$$

so that $u_n = (1 + h\lambda)^{-n} u_0$. In this case stability requires that

$$\left| \frac{1}{1 + h\lambda} \right| < 1,$$

which is satisfied (so long as $\lambda < 0$) for all $h > 0$. In other words, we may take as large a stepsize as we wish, and our solution will converge. This does mean it will be accurate! This is still a first-order method. But we are guaranteed that our solution will have the right asymptotic behavior as we take more and more steps. Schemes such as backward Euler are implicit schemes: the solution at step n must be found by solving a system of equations. For just one equation, they don't add much; if we wanted to solve the system accurately we would be taking $h \ll 2/\lambda$ in any case, so stability is not much of a consideration. The picture becomes considerably different, however, when there is more than one equation.

4.2 *A system of two ODEs*

Now, having warmed up with a single ODE, consider the system of equations,

$$\frac{\mathrm{d}u}{\mathrm{d}t} = z, \tag{4.9}$$

$$\frac{\mathrm{d}z}{\mathrm{d}t} = -\lambda u - (1 + \lambda) z. \tag{4.10}$$

You can verify that this system of equations has the solution

$$u = e^{-t} + e^{-\lambda t}, \tag{4.11}$$

$$z = -e^{-t} - \lambda e^{-\lambda t}. \tag{4.12}$$

In this case the extension of eq. (4.6) is

$$\begin{pmatrix} u_n \\ z_n \end{pmatrix} = \mathbf{C}^n \cdot \begin{pmatrix} u_0 \\ z_0 \end{pmatrix} \tag{4.13}$$

where $\mathbf{C}$ is a 2×2 matrix,

$$\mathbf{C} = \begin{pmatrix} 1 & h \\ -h\lambda & 1 - h(1 + \lambda) \end{pmatrix}. \tag{4.14}$$

In order for $(u_n \;\; z_n)^T \to 0$ as $n \to \infty$, the analog to the analysis following equation (4.7) is to require that the two eigenvalues of $\mathbf{C}$ are less than one in absolute value. First, we must find the two eigenvalues of $\mathbf{C}$. Although there is a formal method for doing this, we can find the by recalling that the eigenvalues ξ_i satisfy the relations

$$\prod_{i=1}^{2} \xi_i = \det \mathbf{C}, \qquad \sum_{i=1}^{2} \xi_i = \mathrm{tr}\, \mathbf{C}. \tag{4.15}$$

In this case,

$$\det \mathbf{C} = 1 - h(1+\lambda) + h^2\lambda$$

and

$$\operatorname{tr} \mathbf{C} = 2 - h(1+\lambda),$$

so the two eigenvalues must be

$$\xi_1 = 1 - h, \qquad \xi_2 = 1 - \lambda h.$$

Hence for stability we must have

$$0 < h < \min\left(2, \frac{2}{\lambda}\right). \tag{4.16}$$

Here we begin to see the problem with explicit schemes. We are forced to follow the shortest timescale in the problem, even if that variable is negligible. This is a critical common problem in reaction networks, where the timescales can differ by orders of magnitude. This would make many problems simply intractable, and an implicit solver is imperative.

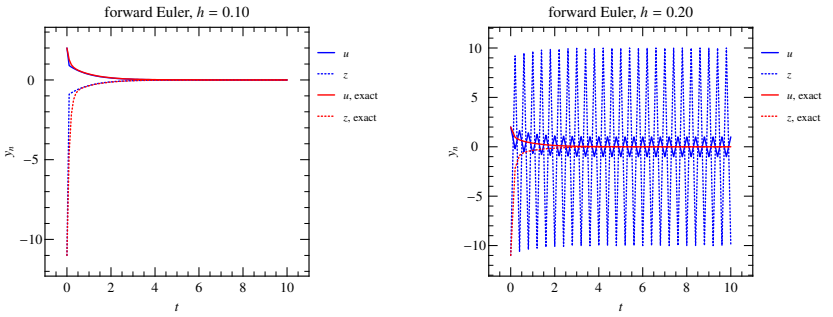


Figure 4.1: Integration of equations (4.9)–(4.10) with $\lambda = 10$ using a forward Euler method. (*Left*) With $h\lambda = 1.0$ the solution is stable, although not particularly accurate; but with $h\lambda = 2.0$ the numerical scheme becomes unstable (*right*).

How would a backward scheme look? In this case our vector at step n is given explicitly in terms of the step at $n+1$,

$$\begin{pmatrix} u_n \\ z_n \end{pmatrix} = \begin{pmatrix} 1 & -h \\ \lambda h & 1 + h(1+\lambda) \end{pmatrix} \cdot \begin{pmatrix} u_{n+1} \\ z_{n+1} \end{pmatrix}. \tag{4.17}$$

We can invert this equation using the formula $a_{ij}^{-1} = C_{ji}/|\det A|$, where C_{ji} is the ji-th cofactor of a_{ij},

$$\begin{pmatrix} u_{n+1} \\ z_{n+1} \end{pmatrix} = \frac{1}{|(1+h)(1+\lambda h)|} \begin{pmatrix} 1 + h(1+\lambda) & h \\ -\lambda h & 1 \end{pmatrix} \cdot \begin{pmatrix} u_n \\ z_n \end{pmatrix}. \tag{4.18}$$

It is easy to show that the eigenvalues for the matrix in equation (4.18) are

$$\xi_1 = \frac{1}{1+h}, \qquad \xi_2 = \frac{1}{1+\lambda h},$$

so this scheme is again stable for all $h > 0$ if $\lambda > 0$.

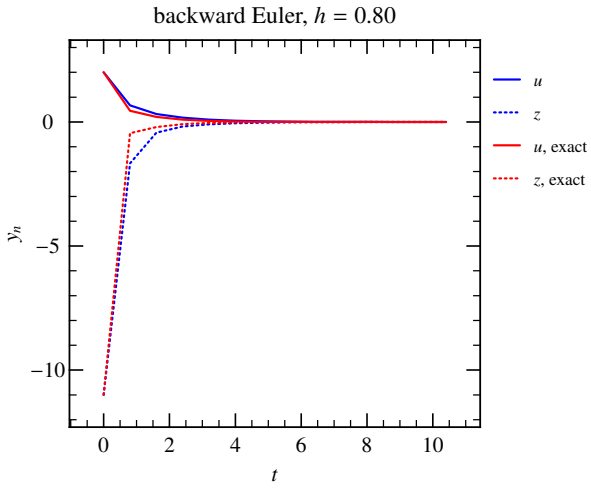


Figure 4.2: Integration of equations (4.9)–(4.10) with $\lambda = 10$ using a backward Euler step. In contrast to the forward Euler method, this scheme converges even with $h\lambda = 8.0$ (although accuracy is rather poor).

A third method for solving this problem is the *trapezoidal method*, which is second order and mixes an explicit and implicit step,

$$y_j^{n+1} = y_j^n + \frac{h}{2}\left[f_j(y_i^n) + f_j(y_i^{n+1})\right) .$$

For the system of equations (4.9)–(4.10), this method has eigenvalues

$$\xi_1 = \frac{1 - h/2}{1 + h/2}, \qquad \xi_2 = \frac{1 - h\lambda/2}{1 + h\lambda/2},$$

which implies stability for $h > 0, \lambda > 0$. But note that for very stiff problems (large λ), $|\xi|_2 \to 1$; as a result the transient $e^{-\lambda t}$ is only weakly damped and convergence is rather poor.

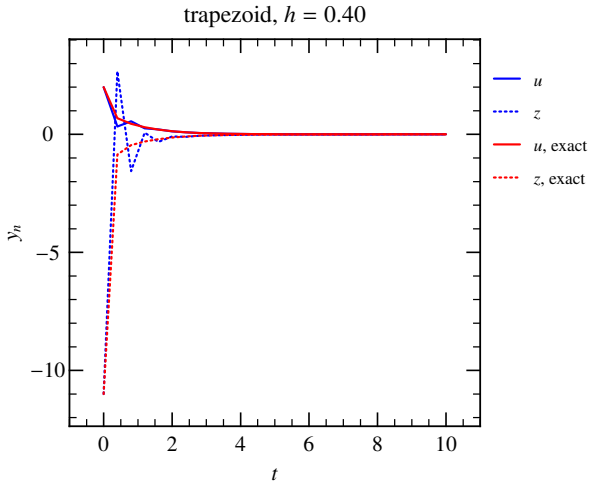


Figure 4.3: Integration of equations (4.9)–(4.10) with $\lambda = 10$ using a trapezoid method. Here $h\lambda = 4.0$; although the solution does converge, the numerical solution oscillates on account of the weak damping of the transient part of the solution.

EXERCISE 4.1 — Show that the system of equations that must be solved at each step in the trapezoidal scheme is

$$\begin{pmatrix} 1 & -\frac{h}{2} \\ \frac{\lambda h}{2} & 1+\frac{h}{2}(1+\lambda) \end{pmatrix} \begin{pmatrix} u_{n+1} \\ z_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & \frac{h}{2} \\ -\frac{\lambda h}{2} & 1-\frac{h}{2}(1+\lambda) \end{pmatrix} \begin{pmatrix} u_n \\ z_n \end{pmatrix} \tag{4.19}$$

EXERCISE 4.2 — Write a forward Euler loop to solve equations (4.9)–(4.10) with $\lambda = 10$. Integrate over a sufficiently large range in t, e.g., out to $t = 10$ to catch any bad behavior. Verify that the solution is unstable for h larger than the limit you derived. Then repeat with a backward Euler scheme and verify that the integration is stable. What happens if you use a trapezoidal scheme?

4.3 *Real problems, not linear*

Implicit schemes must solve at each step equations of the form $y_{n+1} = f(t_n, y_{n+1})$. This is fine if we have a linear equation, but what happens if this equation is nonlinear? We can do a rootfind (using a multidimensional Newton method perhaps); another option is to linearize our equations. Here we'll give an example of taking a semi-implicit Euler step. Given a set of M ODEs

$$\dot{y}_i = f_i(y_j), \quad i,j = 1,\ldots,M,$$

we expand f_i in terms of $(y_j^{n+1} - y_j^n)$,

$$f_i(y_j^{n+1}) = f_i(y_j^n) + \frac{\partial f_i}{\partial y_j}\left(y_j^{n+1} - y_j^n\right),$$

to obtain

$$\left(y_i^{n+1} - y_i^n\right)\left(\delta_{ij} - h\frac{\partial f_i}{\partial y_j}\right) = h f_i(y_j^n). \tag{4.20}$$

Here $\partial f_i/\partial y_j$ is a $M \times M$ matrix called the *Jacobian* and δ_{ij} is the identity matrix. By inverting the matrix

$$A_{ij} \equiv \left(\delta_{ij} - h\frac{\partial f_i}{\partial y_j}\right),$$

we obtain the desired solution

$$y_i^{n+1} = y_i^n + h(A^{-1})_{ij} f_j. \tag{4.21}$$

This is a *semi-implicit* scheme, so it is not guaranteed to be stable. Strict monitoring of errors and adaptive stepsizes are a must.

5
Traffic

Since you now have a tested RK4 solver, let's use it to do a toy simulation of cars on a street. This is a warm-up to treating fluids; we'll first look at a system of a small number of interacting particles and then move to the limit of treating the fluid as a continuum.

Suppose we have N cars, and each car's location is described by $X_i(t), i = 1, \ldots, N$. The velocity of each car is $V_i = X_i'$ and each car drives according to a very simple rule:

$$V_i = V_{\max}\left(1 - \frac{\rho_i}{\rho_{\max}}\right), \tag{5.1}$$

where

$$\rho_i = \frac{1}{x_{i+1} - x_i}. \tag{5.2}$$

That is, as the distance to the car in front gets smaller, the velocity gets smaller. A car with an open road in front drives at the speed limit $V_{\max}$.

5.1 Characteristics: A worked example

To connect our cars with fluid mechanics, lets put $N = 100$ cars on a grid. We will set $\rho_{\max} = 1.0$, $V_{\max} = 1.0$. We will space them according to the following prescription: for $x < 40$, $\rho = 0.25$; for $80 < x < 120$, $\rho = 0.9$; and for $x > 160$, $\rho = 0.25$. We interpolate smoothly between the high and low regions as follows: for $40 < x < 80$,

$$\rho = 0.25 + 0.65 \sin^2\left(\frac{\pi}{2}\frac{x - 40}{80 - 40}\right).$$

The use of the $\sin^2$ ensures the derivatives are continuous as well, i.e., the interpolation is smooth. We shall use periodic boundary conditions: as cars move off the grid on the right, they are added back on the left. We plot the density distribution in Figure 5.1 at time $t = 0$ (*left panel*) and at $t = 79$ (*right panel*). What do we observe? First note that although the cars are moving from left to right, the bump in density propagates

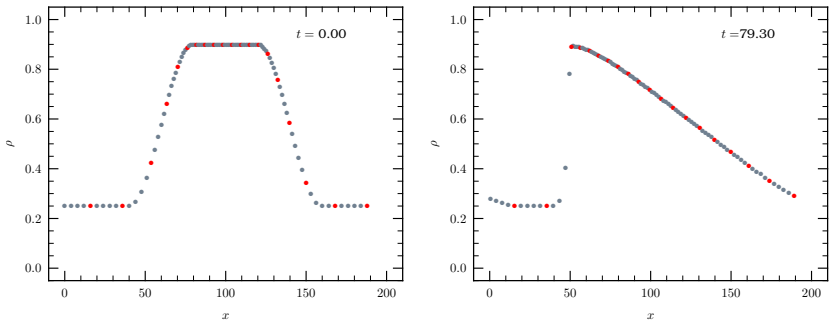


Figure 5.1: Characteristics of traffic flow.

backward. Furthermore, notice that the left edge of the bump steepens while the right becomes more shallow.

To understand this behavior in more detail, let's introduce an equation expressing the conservation of cars (no fusion or fission allowed!):

$$\partial_t \rho + \partial_x \left[\rho V_{\max} \left(1 - \frac{\rho}{\rho_{\max}} \right) \right] = 0. \tag{5.3}$$

The quantity in [ ] is the flux of cars and is only a function of ρ. We can therefore write eq. (5.3) as

$$\partial_t \rho + f'(\rho) \partial_x \rho = 0,$$

where $f = \rho V_{\max}(1 - \rho/\rho_{\max})$ is the flux.

Now, let us look for *characteristic* curves: paths $X(t)$ such that along path, ρ is constant. Such a path satisfies

$$\frac{\mathrm{d}}{\mathrm{d}t} \rho(x = X(t), t) = (\partial_x \rho) \frac{\mathrm{d}X}{\mathrm{d}t} + \partial_t \rho = 0. \tag{5.4}$$

Expanding eq. (5.3) and eliminating $\partial_t \rho$ from eq. (5.4) gives (here we assume that whatever mathematical requirements are needed to ensure a solution do in fact exist) an equation for the characteristics:

$$\frac{\mathrm{d}X}{\mathrm{d}t} = f'(\rho) = V_{\max} \left(1 - 2 \frac{\rho}{\rho_{\max}} \right). \tag{5.5}$$

In our example (Fig. 5.1), the characteristics in the high-density bump, $\rho/\rho_{\max} = 0.9$, move *backward*: $\dot{X} = -0.8 V_{\max}$. For the low-density regions, $\rho/\rho_{\max} = 0.25$, the characteristics move forward: $\dot{X} = 0.5 V_{\max}$. Note that these are not the velocities of individual cars!

The bump moves backwards because information in a high density ($\rho > 0.5\rho_{\max}$) region moves backward along characteristics. The left edge of the bump steepens because the characteristics are converging there. Since the characteristics in a region where $f(\rho)$ is smooth do not cross, the convergence along the left-edge of the bump implies the formation of a discontinuity in the solution: a shock front.

On the right-hand side of the bump, the characteristics are diverging. This produces a *rarefaction*, the properties of which will be explored in the exercise.

EXERCISE 5.1 —

1. **A light turning green—** Put $N = 60$ cars on the road. The cars' positions at $t = 0$ are

$$X_i(t=0) = \begin{cases} 5(i-19)/\rho_{\max} & i = 1, \ldots, 19 \\ (i-19)/\rho_{\max} & i = 20, \ldots, N \end{cases} \tag{5.6}$$

   The car in front ($i = N$) sees a density $\rho_N = 0$. Integrate the positions of all the cars forward in time at least to $t = 150.0(\rho_{\max} V_{\max})^{-1}$ (you may need to integrate longer for the second problem). Plot the trajectories of all the cars on a single plot of time versus position.

2. **A slow driver—** The cars' positions at $t = 0$ are

$$X_i(t=0) = 5(i-N)/\rho_{\max} \quad i = 1, \ldots, N \tag{5.7}$$

   The car in front, i.e., car N, is driven by "Speedy" who as a matter of principle never drives faster than $0.6V_{\max}$, even with an open road ($\rho_N = 0$) in front (any perceived relation between Speedy and anyone you know is purely coincidental). Integrate the positions of all the cars forward in time. Plot the trajectories of all the cars on a single plot of time versus position.

3. **A rarefaction—** Let's construct an initial density profile with two piecewise constant values separated by a discontinuity at $x = 0$. (A problem of this nature is called the *Riemann problem*.) That is,

$$\rho(x, t=0) = \begin{cases} \rho_{\mathrm{L}}, & x < 0 \\ \rho_{\mathrm{R}}, & x \geq 0 \end{cases} \tag{5.8}$$

   with $\rho_{\mathrm{L}} = 0.9$ and $\rho_{\mathrm{R}} = 0.25$.

   Arrange cars on the grid following this prescription and start their engines. Plot the density as a function of position at some later time (cf. Fig. 5.1). You should see a rarefaction front. Make sure you have enough cars (say $N = 100$) to see this front develop. Now, let's understand what's going on in a bit more detail. You may notice that the front begins to appear almost as a straight line connecting a region with $\rho = \rho_{\mathrm{L}}$ to a region with $\rho = \rho_{\mathrm{R}}$. Let's run with that: from the discontinuity at $x = 0$ there will be two diverging characteristics that set the rightmost point of ρ_{L} and the leftmost point of ρ_{R}. The equations of these characteristics are $X_{\mathrm{L}}(t) = -0.8V_{\max}t$ and $X_{\mathrm{R}}(t) = 0.5V_{\max}t$. (Recall that the characteristics satisfy the equation $\dot{X} = V_{\max}(1 - 2\rho/\rho_{\max})$.)

   Suppose that for $X_{\mathrm{L}}(t) < x < X_{\mathrm{R}}(t)$ the density does indeed follow a straight line with $\rho(x = X_{\mathrm{L}}(t)) = \rho_{\mathrm{L}}$ and $\rho(x = X_{\mathrm{R}}(t)) = \rho_{\mathrm{R}}$. Derive an equation for ρ in this region in terms of x and t. Then derive an equation for the characteristics in this region. Does this match your numerical solution?

6
The Equations of Fluid Mechanics

6.1 Fluids as continua

Over scales that are large compared to the collisional mean free paths between particles, we can treat the fluid as a continuous medium. That is, we suppose that we can find a scale that is infinitesimal compared to the macroscopic scales, but still much larger than the scales for microscopic interactions. Thus, we can define thermodynamic quantities at a location.

Consider such a macroscopically small volume V. Its mass is $M = \int_V \rho \, \mathrm{d}V$, where ρ is the mass density. If $\boldsymbol{u}(\boldsymbol{x}, t)$ is the velocity, then the flux of mass into the element is

$$-\int_{\partial V} \rho \boldsymbol{u} \cdot \mathrm{d}\boldsymbol{S} = \frac{\partial}{\partial t} \int_V \rho \, \mathrm{d}V$$

where the right-hand side follows from mass conservation. Using Gauss's law to transform the left-hand side into an integral over V and combining terms, we have

$$\int_V \left\{ \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) \right\} \mathrm{d}V = 0.$$

Since this equation holds for any V, the integrand must vanish, and we have our first equation,

$$\partial_t \rho + \nabla \cdot (\rho \boldsymbol{u}) = 0. \tag{6.1}$$

Our next equation is to get the analog of $\boldsymbol{F} = m\boldsymbol{a}$. Ignoring viscous effects, the net force on our fluid element (with volume V) is due to the pressure over its surface P and the gradient of the gravitational potential Φ:

$$\int_V \rho \frac{\mathrm{d}^2 \boldsymbol{r}}{\mathrm{d}t^2} \, \mathrm{d}V = \int_V \boldsymbol{F} \, \mathrm{d}V = -\int_V \rho \boldsymbol{\nabla} \Phi \, \mathrm{d}V - \int_{\partial V} P \, \mathrm{d}\boldsymbol{S}.$$

Transforming the second integral on the right-hand side to a volume integral, and assuming that $\boldsymbol{\nabla}\Phi$ and $\boldsymbol{\nabla}P$ vary on macroscopic lengthscales, we arrive at an equation for the acceleration,

$$\frac{\mathrm{d}^2 \boldsymbol{r}}{\mathrm{d}t^2} = -\boldsymbol{\nabla}\Phi - \frac{1}{\rho}\boldsymbol{\nabla}P. \tag{6.2}$$

where $\boldsymbol{r}(t)$ is the position of the particle so that the left-hand side is the acceleration. Here we must be careful: $\boldsymbol{u}(\boldsymbol{x}, t)$ refers to velocity of the fluid at a given point in space and a given instance of time, *not* to the velocity of a given particle. A fluid element can still accelerate even if $\partial_t \boldsymbol{u} = \boldsymbol{0}$ by virtue of moving a different location. At time t this particle has the velocity

$$\left.\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t}\right|_t = \boldsymbol{u}(\boldsymbol{x} = \boldsymbol{r}|_t, t) \tag{6.3}$$

where we use the fact that the particle is moving along a streamline of the fluid. At a slightly later time h, the particle has moved to a location $\boldsymbol{r}(t+h) \approx \boldsymbol{r}(t) + h\boldsymbol{u}$, and the velocity is now

$$\left.\frac{\mathrm{d}\boldsymbol{r}}{\mathrm{d}t}\right|_{t+h} = \boldsymbol{u}(\boldsymbol{x} = \boldsymbol{r}|_{t+h}, t+h) \approx \boldsymbol{u} + h(\boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u} + \partial_t \boldsymbol{u}), \tag{6.4}$$

where we evaluate the derivatives at time t. Subtracting equation (6.3) from equation (6.4) and dividing by h gives us the acceleration; inserting this into Newton's law and dividing by volume gives us *Euler's* equation of motion,

$$\partial_t \boldsymbol{u} + \boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u} = -\boldsymbol{\nabla}\Phi - \frac{1}{\rho}\boldsymbol{\nabla}P. \tag{6.5}$$

6.2 *The conservation laws*

We will start from the equations expressing conservation of mass[1], momentum, and energy. We already derived the continuity (conservation of mass) equation,

$$\partial_t \rho + \nabla\cdot(\rho\boldsymbol{u}) = 0, \tag{6.6}$$

and the Euler equation,

$$\partial_t \boldsymbol{u} + \boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u} = -\boldsymbol{\nabla}\Phi - \frac{1}{\rho}\boldsymbol{\nabla}P. \tag{6.7}$$

Note that if we multiply eq. (6.7) by ρ, we can rewrite it, using eq. (6.6), as

$$\partial_t(\rho\boldsymbol{u}) + \nabla\cdot[\boldsymbol{u}(\rho\boldsymbol{u})] = -\rho\boldsymbol{\nabla}\Phi - \boldsymbol{\nabla}P. \tag{6.8}$$

The left-hand side is interpreted as expressing the conservation of momentum ($\rho\boldsymbol{u}$) in the absence of forces, analogous to eq. (6.6) for the conservation of mass (ρ).

Note the general form of a conservation equation:

$$\partial_t(\text{conserved quantity}) + \nabla\cdot(\text{flux of conserved quantity}) = (\text{sources}) - (\text{sinks}).$$

Because the momentum density $\rho\boldsymbol{u}$ is a vector, its flux is a tensor: $[\boldsymbol{u}(\rho\boldsymbol{u})]_{ij} \equiv \rho u_i u_j$.

[1] In a relativistic system, we would instead start from conservation of baryon number.

The next equation is that of energy conservation. Here we must consider both the internal energy per unit volume $E/V = \rho\epsilon$ and the kinetic energy per unit volume $\rho u^2/2$. In this section ϵ represents the internal energy per unit mass of the fluid. In a fixed volume of the fluid the total energy is then

$$\int_V (\rho \frac{1}{2} u^2 + \rho\epsilon)\, \mathrm{d}V.$$

The flux of energy into this volume will clearly include

$$-\int_{\partial V} \left(\frac{1}{2}\rho u^2 + \rho\epsilon\right) \boldsymbol{u} \cdot \mathrm{d}\boldsymbol{S}.$$

But wait, there's more! In addition, we have a conductive heat flux,

$$-\int_{\partial V} \boldsymbol{F} \cdot \mathrm{d}\boldsymbol{S}.$$

Moreover, the pressure acting on fluid flowing into our volume does work on the gas at a rate

$$-\int_{\partial V} P\boldsymbol{u} \cdot \mathrm{d}\boldsymbol{S}.$$

As a result, the net change of energy in our volume is

$$\partial_t \int_V \left(\frac{1}{2}\rho u^2 + \rho\epsilon\right) \mathrm{d}V = \\ -\int_{\partial V} \mathrm{d}\boldsymbol{S} \cdot \left[\boldsymbol{u}\left(\frac{1}{2}\rho u^2 + \rho\epsilon + P\right) + \boldsymbol{F}\right] + \int_V (\rho\boldsymbol{u}\cdot\boldsymbol{g} + \rho q)\,. \quad (6.9)$$

On the right-hand side we've added in the work done by gravity and the heating evolved by nuclear reactions (this could also involve sinks, such as neutrinos, which have a long mean free path). Expressed in differential form, this is

$$\partial_t \left(\frac{1}{2}\rho u^2 + \rho\epsilon\right) + \nabla\cdot\left[\rho\boldsymbol{u}\left(\frac{1}{2}u^2 + \epsilon + \frac{P}{\rho}\right)\right] + \nabla\cdot\boldsymbol{F} = \rho q + \rho\boldsymbol{u}\cdot\boldsymbol{g}. \quad (6.10)$$

You are possibly wondering why I didn't put gravity, which can be expressed as a potential, on the left hand side of this equation. The reason is that the gravitational stresses cannot be expressed in a *locally* conservative form; it is only when integrating over all space that the conservation law appears.

Equations (6.6), (6.8), and (6.10) are supplemented by an equation of state, which allows one to get from the pressure P, the temperature T, and the mass fractions X_i of the species present, the remaining thermodynamical quantities, such as mass density ρ and specific energy ϵ. In addition, Poisson's equation

$$\nabla^2\Phi = 4\pi G\rho, \quad (6.11)$$

specifies the gravitational acceleration $\boldsymbol{g} = -\boldsymbol{\nabla}\Phi$. We then need one more equation to specify the heat flux F. If the typical lengths over which

particles or photons travel before scattering is very small compared to the lengthscale over which the macroscopic properties of the fluid vary, we expect the flux to obey a conduction equation of the form

$$\boldsymbol{F} = -K\boldsymbol{\nabla}T. \tag{6.12}$$

This assumption is clearly questionable in many astrophysical plasmas.

6.3 *Thermodynamical quantities*

In most textbooks on thermodynamics and statistical mechanics, the thermodynamics are formulated in terms of some sample of fixed size. For example, in the first law,

$$\mathrm{d}E = T\mathrm{d}S - P\mathrm{d}V, \tag{6.13}$$

the energy E and entropy S are extensive quantities, and scale with the number of particles N in our sample. In a fluid, however, these quantities are all functions of position. By $S(r)$, we mean that we can define a small portion of the star about the coordinate r that is large enough particles to ensure that quantities such as pressure and temperature are well-defined, but small enough that we can treat $S(r)$ as a continuous function of position when integrating over the whole star.

Using extensive quantities in fluid mechanics is cumbersome, so we instead use quantities like the energy per unit mass $\epsilon = E/(\rho V)$ or the entropy per unit mass $s = S/(\rho V)$. Since a fixed mass of fluid M occupies a volume $V = M/\rho$, we can divide the first law, eq. (6.13), by M to obtain

$$\mathrm{d}\epsilon = T\mathrm{d}s - P\mathrm{d}\left(\frac{1}{\rho}\right) = T\mathrm{d}s + \frac{P}{\rho^2}\mathrm{d}\rho. \tag{6.14}$$

The other extensive variables can be re-defined into mass-specific forms in a similar fashion.

7
A simple PDE: the linear advection equation

7.1 Background

As a simple example of a partial differential equation, let's consider the *advection* equation in 1-d. Suppose we have a fluid in a pipe flowing with velocity u. Now suppose we inject a bit of dye into our pipe. If we ignore diffusion, then the concentration of dye $\varphi(x,t)$ will be described by

$$\partial_t \varphi = -u \partial_x \varphi, \tag{7.1}$$

which is just the one-dimensional form of the conservation equation

$$\partial_t \varphi + \nabla \cdot (\boldsymbol{u}\varphi) = 0. \tag{7.2}$$

Systems of equations where a quantity obeys a conservation law like eq. (7.2) are quite common in physics.

7.2 Discretization in space

To solve equation (7.1), let's discretize our solution on a uniform grid in the x-coordinate, with a grid spacing h. One way of representing $\partial_x \varphi$ is to use a finite difference scheme. There are several ways to do this. Let $x_j = j \times h, j = 1, \ldots, M$ be the points on our grid. Let Q_j^n be our approximate solution for $\varphi(x = x_j, t = t_n)$. Then we can approximate $\partial_x \varphi$ at $x = x_j$ by, for example,

$$\partial_x \varphi \approx \mathcal{D}_u Q_j^n \equiv \frac{Q_j^n - Q_{j-1}^n}{h} \qquad \text{upwind differencing} \tag{7.3}$$

$$\partial_x \varphi \approx \mathcal{D}_c Q_j^n \equiv \frac{Q_{j+1}^n - Q_{j-1}^n}{2h} \qquad \text{central differencing.} \tag{7.4}$$

Schemes with even more grid points included are possible as well.

Exercises

1. Show that the upwind differencing scheme is first-order in grid spacing h, i.e., $\mathcal{D}_u Q_j^n = \partial_x \varphi|_{x=x_j, t=t_n} + \mathcal{O}(h)$ for $Q_j^n = \varphi(x = x_j, t = t_n)$.

2. Show that the central differencing scheme is second-order in grid spacing h.

7.3 *Discretization in time*

The simplest scheme for advancing in time from $t = t_n$ to $t_{n+1} = t_n + \tau$ is the forward Euler step,

$$Q_j^{n+1} = Q_j^n - \tau u \mathcal{D} Q_j^n. \tag{7.5}$$

Inserting the central differencing scheme, for example, would produce

$$Q_j^{n+1} = Q_j^n - \left(\frac{\tau u}{h}\right) \frac{Q_{j+1}^n - Q_{j-1}^n}{2}. \tag{7.6}$$

Notice that our choice of time step appears in combination with $\tau u/h$. Clearly for our scheme to be faithful to the underlying differential equations, we must have this combination be small, say $\tau u/h = 0.1$ to be concrete. Physically, this just means we don't want to fluid to cross more than a fraction of a grid during any given time step.

7.4 *A worked example*

Now let's try out these schemes. Let's set $u = 1.0$, $h = 0.1$ and make the grid $x_j = j \times h$, for $j = 1, 2, \ldots, 500$. We can then set $\tau = 0.01$, so that $\tau u/h = 0.1$. Finally, we need to establish an initial profile. Rather than having a sharp step, let's use the following function, which smooths out the front,

$$Q_j^0 = \frac{1}{2}\left[1 - \tanh\left(\frac{x_j - 5}{5h}\right)\right]. \tag{7.7}$$

This equation produces a front centered at $x = 5$, with a width of roughly $10 \times h$, so that it is well-resolved (see Fig. 7.1).

Exercises

1. Verify that the solution to equations (7.1) and (7.7) at time $t = t_n$ is

$$Q_j^n = \frac{1}{2}\left[1 - \tanh\left(\frac{x_j - ut_n - 5}{5h}\right)\right]. \tag{7.8}$$

2. Solve equations (7.1) and (7.7) using upwind differencing (eq. [7.3]). Let it run until the front reaches the midpoint of your domain (i.e., $t = 20$, $n = 2000$). How does the numerical solution compare with the exact solution, eq. (7.8)?

3. Now repeat exercise 2, but use a central differencing scheme (eq. [7.4]). How do things change? Are they what you expected?

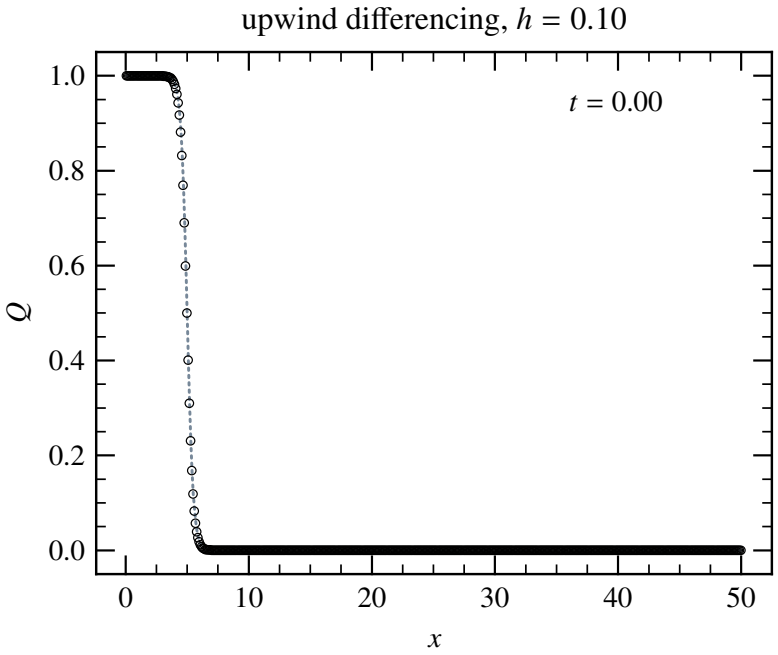


Figure 7.1: Initial profile for the advection equation.

7.5 *Stability*

As the problem 3 showed, not all difference schemes are stable! Let's understand this in a bit more detail. Rather than do a standard analysis, which writes our solution as the sum of the solution to the underlying PDE and a error term, we'll take our numerical solution to be the *exact* solution for some PDE, and try to see what PDE our solution actually solves. This example is taken from LeVeque [2002].

For upwind differencing, we take $g(x,t)|_{x=x_i,t=t_n} = Q_i^n$. From equation (7.3) and (7.5) the function g must satisfy

$$g(x,t+\tau) - g(x,t) = -\mathrm{Co}\left[g(x,t) - g(x-h,t)\right].$$

Here $\mathrm{Co} = u\tau/h$ is the *Courant number*. Expanding both sides to second order in h and τ,

$$\partial_t g\tau + \frac{1}{2}\partial_{tt} g\tau^2 = -\mathrm{Co}\left[\partial_x gh - \frac{1}{2}\partial_{xx} gh^2\right] \tag{7.9}$$

and rearranging terms gives

$$\partial_t g + u\partial_x g = -\frac{\tau}{2}\partial_{tt} g + \frac{uh}{2}\partial_{xx} g. \tag{7.10}$$

The left hard side is our original advection equation, but what is the right hand side? Differentiate equation (7.9) by t to obtain

$$\tau\partial_{tt} g = -u\tau\partial_{xt} g - \frac{\tau^2}{2}\partial_{ttt} g + \frac{u\tau h}{2}\partial_{xxt} g. \tag{7.11}$$

This may not look like much help, but if we then differentiate equation (7.9) with respect to x,

$$\tau\partial_{tx}g = -u\tau\partial_{xx}g - \frac{\tau^2}{2}\partial_{ttx}g + \frac{u\tau h}{2}\partial_{xxx}g$$

we can use the equality $\partial_{xt} = \partial_{tx}$ to eliminate the first right-hand term in equation (7.11) and obtain to lowest order that $\partial_{tt}g = u^2\partial_{xx}g$. Here we use the fact that τ/h is a fixed number, so that all the other terms are indeed second order. Substituting this into equation (7.10) then gives

$$\partial_t g + u\partial_x g = \frac{uh}{2}\left(1 - \mathrm{Co}\right)\partial_{xx}g. \tag{7.12}$$

We recognize the right-hand side as being a *diffusive term*. For $\mathrm{Co} < 1$, the diffusion coefficient is positive, and hence has stable solutions. We thus have a stable differencing scheme, but it comes at a cost: our solution has an effective kinematic viscosity of order $u \times h$. The effect is that our front is smeared out, as can be seen in Fig. 7.2.

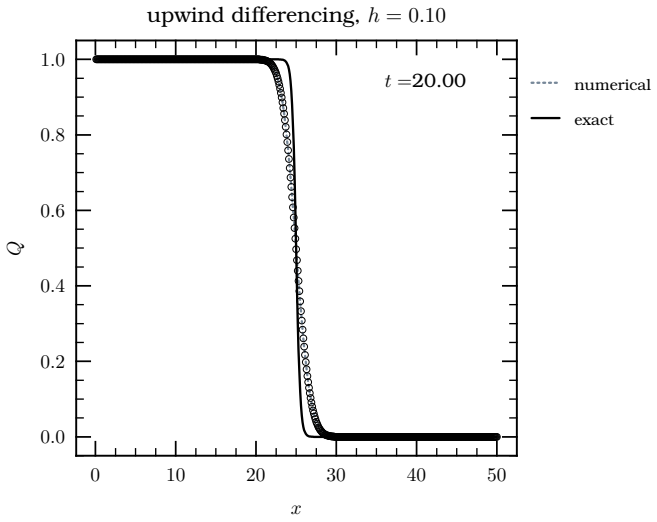


Figure 7.2: Exact and numerical solutions of advection equation at $t = 20$.

Recall that for the Reynolds number is defined by $\mathrm{Re} \equiv uL/\nu$, where L is a characteristic lengthscale and ν is the kinematic viscosity. In a plasma, $\nu \sim c_s\lambda$, where c_s is the sound speed (the molecular speed) and λ is the mean free path, the average distance a particle goes before having a collision. Hence the ratio of the effective Reynolds number of a simulation to that of the physical system, for fluid flows of order the sound speed, is

$$\frac{\mathrm{Re}_{\mathrm{num}}}{\mathrm{Re}} \approx \frac{\lambda}{h}.$$

Note that $\mathrm{Re}_{\mathrm{num}} \approx N$, where $N = L/h$ is the number of grid points. In an accretion disk, $\mathrm{Re} \sim 10^{12}$. When examining results of numerical

simulations of astrophysical plasmas, bear in mind that the simulations are likely to have an effective Reynolds closer to that of snake oil[1] ($\nu_{\text{snake}} \approx 0.3\,\text{cm}^2/\text{s}$) than that of the physical system.

[1] Assuming it is similar to whale oil.

8
Flux-conservative algorithms

The following notes make extensive use of the excellent book by LeVeque [2002].

8.1 Introduction: Burgers Equation

For a simple example that illustrates the difference between a conservative differencing scheme and a non-conservative one, consider *Burgers equation*,

$$\partial_t u + u\partial_x u = 0. \tag{8.1}$$

This is written as a nonlinear advection equation. We can rewrite it, however, in the conservative form

$$\partial_t u + \partial_x \left(\frac{1}{2}u^2\right) = 0. \tag{8.2}$$

This is called a conservative form because it has the form

$$\partial_t(\text{quantity}) + \nabla \cdot (\text{flux of quantity}) = 0,$$

if we identify the flux as $u^2/2$.

To see why the distinction matters, let's make a simple case. Define u on the grid $x \in [-1, 4]$, and define

$$u(x, t = 0) = \begin{cases} 2 & x \leq 0 \\ 1 & x > 0 \end{cases}, \tag{8.3}$$

with boundary conditions $u(x = -1, t) = 2, u(x = 4, t) = 1$. Use an upwind differencing scheme (eq. [7.3]), and numerically integrate the equations (8.1) and (8.2) from $t = 0$ to $t = 2$. Determine the location of the "step" at this time. Make sure you have sufficient grid resolution to determine the step's position to within 1%.

The result of this exercise is plotted in Figure 8.1. Both methods have converged, but to different solutions. As a first clue to what is happening,

let's compare the upwind differenced forms for both methods:

$$u_i^{n+1} = u_i^n - \frac{\tau}{h} u_i^n \left(u_i^n - u_{i-1}^n\right), \qquad \text{quasilinear;} \quad (8.4)$$

$$u_i^{n+1} = u_i^n - \frac{\tau}{h}\left[\frac{1}{2}\left(u_i^n\right)^2 - \frac{1}{2}\left(u_{i-1}^n\right)^2\right], \qquad \text{conservative.} \quad (8.5)$$

Both methods are trivially the same at $t = 0$ (and in fact anywhere there is a smooth solution) *except at the discontinuity!* For example, for $u_i^0 = 1$ and $u_{i-1}^0 = 2$, the quasilinear finite difference gives $u_i^1 = 1 + \tau/h$, whereas the conservative method gives $u_i^1 = 1 + 3\tau/(2h)$.

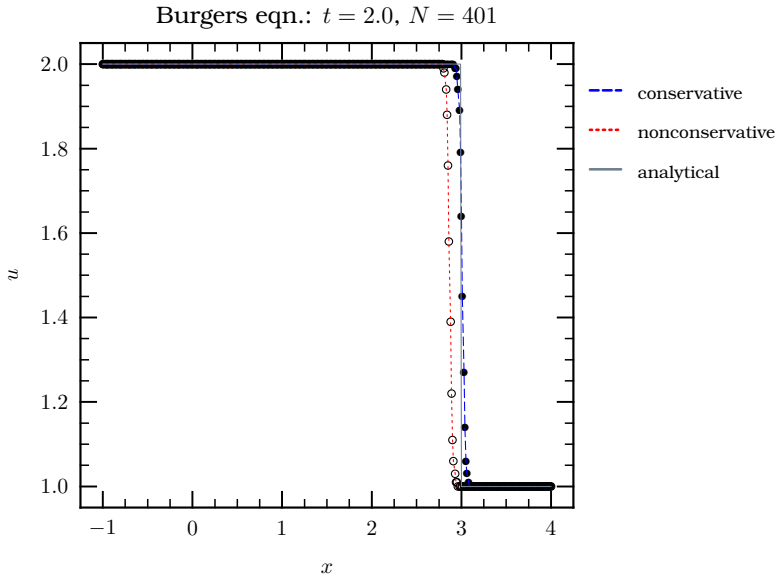


Figure 8.1: Solution of Burgers equation, for both a quasilinear and conservative form.

The problem here is not with our differencing scheme, but rather with the underlying partial differential equation. Although both equations (8.1) and (8.2) are equivalent where u is smooth, continuous, and differentiable, neither is defined at a discontinuity. Such discontinuities do arise in fluid mechanics. Properties like pressure and density are really defined on macroscopic scales, that is, on scales much larger than a particle mean free path. On very small scales the fluid equations simply do not apply. In a shock front, the fluid properties change on a scale of a few particle mean free paths and thus appear as discontinuities. Conservation laws—of mass, of momentum, and of energy—do still apply, however. For example, suppose we have a long tube along which a shock is propagating. Consider a region of the tube that contains the shock, as depicted in Fig. 8.2.

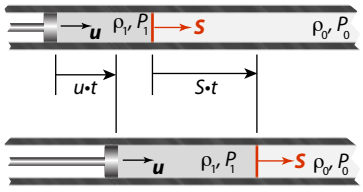


Figure 8.2: Schematic of piston-driven shock.

In a time t, the shock as moved a distance $S \cdot t$ to the right, and the mass in a region containing the shock has therefore increased by $St(\rho_1 - \rho_0)$. This must be equal to the net mass flux into the region, $t(u_1\rho_1 - u_0\rho_0) = tu_1\rho_1$ since for this example the upstream velocity is $u_0 = 0$. Thus the speed of the shock is $S = u_1\rho_1/(\rho_1 - \rho_0)$. In general, if we have

a small interval containing a discontinuity in a conserved quantity φ, with upstream and downstream values $\varphi_{\rm up}$ and $\varphi_{\rm down}$, and upstream and downstream fluxes $f_{\rm up}$ and $f_{\rm down}$, then the discontinuity propagates with speed

$$S = \frac{f_{\rm down} - f_{\rm up}}{\varphi_{\rm down} - \varphi_{\rm up}}. \tag{8.6}$$

We can apply this to Burgers equation if it is written in conservative form (eq. [8.2]; this is the only form for which a discontinuous solution exists). The upstream quantity and flux are $u_{\rm up} = 1.0$, $f_{\rm up} = u_{\rm up}^2/2 = 1/2$; their downstream counterparts are $u_{\rm down} = 2.0$, $f_{\rm down} = u_{\rm down}^2/2 = 2.0$, As a result, the discontinuity travels with speed $S = 1.5$ and at time $t = 2.0$ will have reached $x = 3.0$ (cf. Fig. 8.1). The flux-conservative algorithm converges to the correct result even when given discontinuous input data.

It might be tempting to ascribe this case as being an artifact of a problem in which the initial data is discontinuous. This is not the case, however. Such discontinuities do arise in nature, and in fact can arise even if the initial conditions are smooth. Suppose we replace the initial data (eq. [8.3]) with a smooth function, for example

$$u(x, t = 0) = \frac{1}{2}\left[3 - \tanh\left(\frac{x}{50h}\right)\right]. \tag{8.7}$$

Our profile is now initially smooth, but how does it propagate? The characteristics satisfy the equations

$$X = ut + X_0$$

and along a characteristic we have

$$\begin{aligned} \frac{{\rm d}u(x = X(t), t)}{{\rm d}t} &= \partial_t u + \partial_x u \frac{{\rm d}X}{{\rm d}t} \\ &= \partial_t u + u\partial_x u = 0. \end{aligned}$$

Thus the solution is propagated along characteristics: $u(x, t) = u(x - ut, 0)$. Because the "back" of the front has $u = 2$ while the "front" has $u = 1$, the back of the wave "catches up" to the front of the wave, so that the disturbance will steepen as it propagates, as shown in Figure 8.3.

This steepening is a common occurrence in gaseous flows. In general, the formation of shocks and other discontinuities is a generic feature of hyperbolic equations. It is better to use the integral, that is, conservative, forms of the equations, which are able to handle such discontinuities.

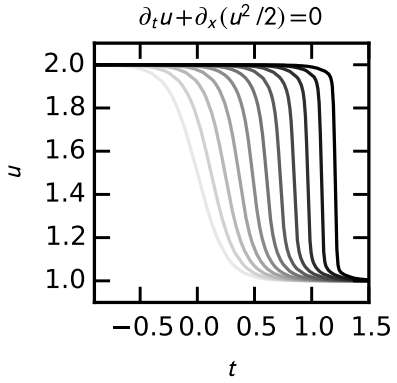


Figure 8.3: The plots show the solution of Burgers's equation for a disturbance propagating along the x-direction. Because the sound speed is greater in the compressed region, the "back" of the disturbance moves faster than the front: as a result, the disturbance steepens. The disturbance steepens into a shock in a time $t \approx \Delta/(u_L - u_R)$, where Δ is the initial width of the transition and u_L and u_R are the values of u on the left and right sides of the disturbance.

8.2 *A Worked Example: Linear Acoustics*

To see how hyperbolic problems can be decomposed in terms of characteristics, let us consider the equations of mass and momentum continu-

ity,

$$\partial_t \rho + \partial_x(\rho u) = 0 \tag{8.8}$$

$$\partial_t(\rho u) + \partial_x(\rho u^2 + p) = 0. \tag{8.9}$$

Let's look for perturbations about a state with constant density and velocity, $\rho = \rho_0 + \rho_1(x,t)$, $u = u_0 + u_1(x,t)$. We assume that fluids with a subscript "1" are small, so we will expand equations (8.8) and (8.9) keeping only terms with at most 1 subscripted power per term. The two equations thus become

$$\partial_t \rho_1 + \rho_0 \partial_x u_1 + u_0 \partial_x \rho_1 = 0 \tag{8.10}$$

$$u_0 \left[\partial_t \rho_1 + \partial_x(\rho u)_1\right] + \rho_0 \partial_t u_1 + \rho_0 u_0 \partial_x u_1 + \frac{\mathrm{d}p}{\mathrm{d}\rho}\partial_x \rho_1 = 0. \tag{8.11}$$

The term in [] vanishes due to mass continuity, and so we can write these equations as a linear system,

$$\begin{pmatrix} \partial_t \rho_1 \\ \partial_t u_1 \end{pmatrix} + \begin{pmatrix} u_0 & \rho_0 \\ \rho_0^{-1} c_s^2 & u_0 \end{pmatrix} \begin{pmatrix} \partial_x \rho_1 \\ \partial_x u_1 \end{pmatrix} = 0 \tag{8.12}$$

This just looks like a matrix form of the linear advection equation,

$$\partial_t \mathbf{q} + \mathbf{A}\partial_x \mathbf{q} = 0,$$

with

$$\mathbf{A} = \begin{pmatrix} u_0 & \rho_0 \\ \rho_0^{-1} c_s^2 & u_0 \end{pmatrix}, \qquad \mathbf{q} = \begin{pmatrix} \rho_1 \\ u_1 \end{pmatrix}.$$

I've also used $\mathrm{d}p/\mathrm{d}\rho = c_s^2$ to simplify things. If we insert a trial characteristic solution, $\rho_1 = \rho(x - st)$, $u_1 = u(x - st)$, then the equation becomes the eigenvalue problem

$$\begin{pmatrix} u_0 & \rho_0 \\ \rho_0^{-1} c_s^2 & u_0 \end{pmatrix} \begin{pmatrix} \rho' \\ u' \end{pmatrix} = s \begin{pmatrix} \rho' \\ u' \end{pmatrix} \tag{8.13}$$

with eigenvalues $s = u_0 \pm c_s$.

The eigenvectors for this problem are

$$\mathbf{e}_1 = \begin{pmatrix} -\rho_0/c_s \\ 1 \end{pmatrix}, \qquad s_1 = u_0 - c_s \tag{8.14}$$

$$\mathbf{e}_2 = \begin{pmatrix} \rho_0/c_s \\ 1 \end{pmatrix}, \qquad s_2 = u_0 + c_s. \tag{8.15}$$

Hence the general solution to our problem is

$$\begin{pmatrix} \rho(x,t) \\ u(x,t) \end{pmatrix} = w_1(x - s_1 t) \begin{pmatrix} -\rho_0/c_s \\ 1 \end{pmatrix} + w_2(x - s_2 t) \begin{pmatrix} \rho_0/c_s \\ 1 \end{pmatrix}. \tag{8.16}$$

If the solution at $t = 0$ is $[\tilde{\rho}(x), \tilde{u}(x)]$, then we can solve for $w_1(x), w_2(x)$:

$$\begin{aligned} w_1(x) &= -\frac{c}{2\rho_0}\tilde{\rho} + \frac{1}{2}\tilde{u} \\ w_2(x) &= \frac{c}{2\rho_0}\tilde{\rho} + \frac{1}{2}\tilde{u}, \end{aligned}$$

and the general solution for the perturbations is

$$\begin{aligned} \rho(x,t) = &\frac{1}{2}\left[\tilde{\rho}(x - s_1 t) + \tilde{\rho}(x - s_2 t)\right] \\ &-\frac{\rho_0}{2c_s}\left[\tilde{u}(x - s_1 t) - \tilde{u}(x - s_2 t)\right] \qquad (8.17) \\ u(x,t) = &-\frac{c_s}{2\rho_0}\left[\tilde{\rho}(x - s_1 t) - \tilde{\rho}(x - s_2 t)\right] \\ &+\frac{1}{2}\left[\tilde{u}(x - s_1 t) + \tilde{u}(x - s_2 t)\right]. \qquad (8.18) \end{aligned}$$

Now apply these solutions to the *Riemann problem*, which is to find how our state evolves with

$$\tilde{\rho}, \tilde{u} = \begin{cases} \rho_L, u_L & x < 0 \\ \rho_R, u_R & x > 0 \end{cases}$$

Here ρ_L and ρ_R are constants, as are u_L and u_R. For simplicity, take $u_0 = 0$, so that $s_1 = -c$ and $s_2 = c$. Then $\tilde{\rho}(x - s_1 t) = \tilde{\rho}(x + ct)$; along the interface $x = 0$ for $t > 0$, this is just ρ_R. This hold for the other eigenvalues as well: those with $s = -c$ come from the right, and those with $s = c$ come from the left. Along the interface, the solution is then

$$\begin{aligned} \rho(0,t) &= \frac{1}{2}\left[\rho_L + \rho_R\right] + \frac{\rho_0}{2c_s}\left[u_L - u_R\right] \qquad (8.19) \\ u(0,t) &= \frac{c_s}{2\rho_0}\left[\rho_L - \rho_R\right] + \frac{1}{2}\left[u_L + u_R\right]. \qquad (8.20) \end{aligned}$$

If $x = 0$ represents the interface between two cells in our fluid, we can then use these solutions to reconstruct the fluxes ρu and $\rho u^2 + p$ along the cell interface, and then update the solutions for ρ, u in the ith cell using these fluxes. For example

$$\int_i \rho^{n+1} dx = -h\left[(\rho u)^n_{i+1/2} - (\rho u)^n_{i-1/2}\right]. \qquad (8.21)$$

Here $(\rho u)_{i+1/2}$ is formed from equations (8.19) and (8.20) with the "left" quantities referring to cell i and the "right" quantities referring to cell $i+1$, and similarly for $(\rho u)_{i-1/2}$. This method of reconstructing the fluxes using the solution to the Riemann problem along the interface forms the essence of *Godunov's scheme*.

9
Solving a Parabolic PDE

9.1 The Diffusion Equation

The diffusion equation,

$$\partial_t \theta = \nabla \cdot (\mathcal{D} \boldsymbol{\nabla} \theta) \,, \tag{9.1}$$

where $\mathcal{D}$ is the diffusivity, is an example of a parabolic partial differential equation. Equation (9.1) is the divergence of a flux $F = -\mathcal{D}\boldsymbol{\nabla}\theta$, and typically has boundary conditions $\theta|_B = \theta_B(t)$ (*Dirichlet*) or $F|_B = F_B(t)$ (*Neumann*).

9.2 The Crank-Nicholson Algorithm

A classic algorithm for integrating equation (9.1) *stably* is the *Crank-Nicholson* formula. Define a uniform mesh with spacing h by $x_j = x_0 + j \cdot h,\ j = 1, \ldots, M$, and further let τ be the step in time, so that $t_n = t_0 + n \cdot \tau$. If φ_j^n is the approximate solution at $x = x_j$, $t = t_n$, then we can define the flux at the half-mesh points $j + 1/2$, $j - 1/2$ as

$$F^n_{j-1/2} = -\mathcal{D}_{j-1/2} \frac{\varphi^n_j - \varphi^n_{j-1}}{h}, \qquad F^n_{j+1/2} = -\mathcal{D}_{j+1/2} \frac{\varphi^n_{j+1} - \varphi^n_j}{h}. \tag{9.2}$$

In the remainder of this project, we shall take $\mathcal{D}$ to be constant, so we can drop its mesh index. Taking the difference of the fluxes in equation (9.2) gives us an approximation for the RHS in equation (9.1),

$$-\mathcal{D}\partial_x \left(-\partial_x \theta\right) \approx \mathcal{D} \frac{\varphi^n_{j-1} - 2\varphi^n_j + \varphi^n_{j+1}}{h^2}. \tag{9.3}$$

A first-order representation of the time derivative is

$$\frac{\varphi^{n+1}_j - \varphi^n_j}{\tau}; \tag{9.4}$$

the characteristic timescale in the problem is $\sim \mathcal{O}(h^2\mathcal{D})$. To avoid getting clobbered taking very small steps in time, it is useful to define an implicit

step, in which the RHS is evaluated at t_{n+1} instead of t_n. The Crank-Nicholson scheme uses the *average* of the implicit and explicit evaluation of the spatial derivative,

$$\frac{\varphi_j^{n+1} - \varphi_j^n}{\tau} = \frac{\mathcal{D}}{2}\left[\frac{\varphi_{j-1}^{n+1} - 2\varphi_j^{n+1} + \varphi_{j+1}^{n+1}}{h^2} + \frac{\varphi_{j-1}^{n} - 2\varphi_j^{n} + \varphi_{j+1}^{n}}{h^2}\right]. \quad (9.5)$$

A stability analysis shows that this scheme is stable, and has the additional benefit of being second-order in both time and space.

Exercises

Verify that equation (9.5) is second-order in h and τ. To verify that it is second-order in τ, you should expand around the half-step point $t = t_{n+1/2}$.

9.3 *Numerical solution*

Define $\chi \equiv \mathcal{D}\tau/(h^2)$ and rearrange equation (9.5) to obtain

$$-\frac{\chi}{2}\varphi_{j-1}^{n+1} + (1+\chi)\varphi_j^{n+1} - \frac{\chi}{2}\varphi_{j+1}^{n+1} = \frac{\chi}{2}\varphi_{j-1}^{n} + (1-\chi)\varphi_j^{n} + \frac{\chi}{2}\varphi_{j+1}^{n}. \quad (9.6)$$

This system of equations is *tridiagonal*: that is, it has the form

$$\mathbf{A}\cdot\boldsymbol{x} = \boldsymbol{b} \quad (9.7)$$

where $\boldsymbol{x} = (\varphi_1^{n+1}, \varphi_2^{n+1}, \ldots, \varphi_M^{n+1})$, $b_j = (\chi/2)\varphi_{j-1}^n + (1-\chi)\varphi_j^n + (\chi/2)\varphi_{j+1}^n$, and

$$\mathbf{A} = \begin{pmatrix} \cdot & \cdot & 0 & 0 & 0 \\ \cdot & \cdot & \cdot & 0 & 0 \\ 0 & -\frac{\chi}{2} & 1+\chi & -\frac{\chi}{2} & 0 \\ 0 & 0 & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \cdot & \cdot \end{pmatrix}. \quad (9.8)$$

The only elements non-zero elements of $\mathbf{A}$ are in a band spanning elements at most once removed from the diagonal. Tridiagonal matrices are very easily inverted, which makes this algorithm very efficient. A standard `lapack` routine for solving equation (9.8) is `xGTSV`, where `x` is either `s` (single precision) or `d` (double precision). Note that in these routines, the entire matrix $\mathbf{A}$ is not stored, but only the diagonal, the sub-diagonal, and the super-diagonal components.

For example, the call to `dgtsv` is

```
call dgtsv(M,nrhs,subdiag,diag,supdiag,rhs,M,info).
```

Here `M` is the rank of the array, and the vectors `subdiag(1:M-1)`, `diag(1:M)`, `superdiag(1:M-1)` hold the sub-diagonal, diagonal, and super-diagonal elements.

Boundary conditions

For Dirichlet boundary conditions, one may simply set $\varphi_1 = \varphi_L$, where φ_L is the fixed value of the solution at the left-hand boundary, for example. For Neuman boundary conditions, the problem is more subtle. Here it is instructive to construct *ghost* zones, at $j = 0$ and $M + 1$. These zones are not included in the matrix solve (eq. 9.7), but the values of φ_0 and φ_{M+1} are adjusted to enforce the boundary conditions. For example, suppose we want an *insulating boundary*, i.e., one with zero flux. In this case we have

$$\begin{aligned} \varphi_2 - \varphi_0 &= 0, \\ \varphi_{M+1} - \varphi_{M-1} &= 0. \end{aligned}$$

Substituting these relations into the finite difference equation about $j = 0$ and $j = M$ gives

$$\begin{aligned} (1+\chi)\varphi_1^{n+1} - \chi\varphi_2^{n+1} &= (1-\chi)\varphi_1^n + \chi\varphi_2^n \\ -\chi\varphi_{M-1}^{n+1} + (1+\chi)\varphi_M^{n+1} &= \chi\varphi_{M-1}^n + (1-\chi)\varphi_M^n. \end{aligned}$$

The use of ghost zones is a common technique for enforcing boundary conditions. Note that we could have enforced the boundary conditions without ghost zones, but this would have required either a lower-order representation of the boundary, or more than two points (and hence a loss of the efficient tridiagonal structure of the matrix).

EXERCISE 9.1 — Solve a diffusion problem in a rod of unit length with insulating boundary conditions. Let the initial thermal profile be

$$\varphi(x, t = 0) = \frac{1}{2}\left[1 - \cos(2x\pi)\right],$$

so that the boundaries have $\varphi(0) = \varphi(1) = 0$ at $t = 0$ and $\varphi'(0) = \varphi'(1) = 0$ as well. Predict the asymptotic solution before you do this numerically.

9.4 An reaction-diffusion problem

Now let's apply our diffusion solver to a more complicated equation, a *reaction-diffusion* equation,

$$\partial_t\theta = \mathcal{D}\partial_x^2\theta + \mathcal{R}\theta(1-\theta). \tag{9.9}$$

The non-linear term on the RHS is called a *KPP* source term. The solution to equation (9.9) is a front with width $\delta \equiv \theta/|\partial_x\theta|_{\max} \propto \sqrt{D/R}$ that travels with speed $u \propto \sqrt{DR}$.

The tricky part of eq. (9.9) is that it is non-linear, so our Crank-Nicholson scheme is not directly applicable. To get around this, we can

use the concept of *Strang splitting*. In general, suppose we can write an equation

$$\partial_t \varphi = \mathcal{L}_1 \varphi + \mathcal{L}_2 \varphi, \tag{9.10}$$

where $\mathcal{L}_1$ and $\mathcal{L}_2$ are operators for which we have a technique to integrate them if only the other operator weren't there. For example, in this case,

$$\mathcal{L}_1 \varphi = D\partial_x^2 \varphi, \qquad \mathcal{L}_2 \varphi = \mathcal{R}\varphi(1-\varphi).$$

One can advance the solution in time by working on each part separately,

$$\varphi_j^* = (1 + \frac{\tau}{2}\mathcal{L}_1)\varphi_j^n \tag{9.11}$$

$$\varphi_j^{**} = (1 + \tau\mathcal{L}_2)\varphi_j^*, \tag{9.12}$$

$$\varphi_j^{n+1} = (1 + \frac{\tau}{2}\mathcal{L}_1)\varphi_j^{**}. \tag{9.13}$$

In words, we would use the Crank-Nicholson scheme to advance the solution a half-step, use that "halfway" solution to compute the reaction term, and then finish the step with the Crank-Nicholson piece.

One can show that this splitting is second-order accurate in time if the individual operators $\mathcal{L}_1$, $\mathcal{L}_2$ are second-order accurate in time.

EXERCISE 9.2 — Numerically solve equation (9.9). You will need to decide on the mesh and time-step. What size must your grid be, and how fine must the mesh be, in order for the flame to be captured? Show that after an initial transient, the flame reaches a steady velocity that scales as $\sqrt{DR}$, and that the width is $\propto \sqrt{D/R}$.

A
Performance

We aren't going to touch on issues of performance or efficiency much in this course, but it's a topic of which one should at least be aware. This exercise is meant as a brief introduction.

1. FORTRAN stores matrices as a sequential stream of numbers, for which the row index advances fastest. For example, the 3×3 matrix $A(i,j)$ is stored sequentially in memory as

   $$A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), A(3,2), A(1,3), A(2,3), A(3,3).$$

   To see how this affects performance, cook up two large matrices (e.g., 1024×1024 so that they don't fit into cache memory). One way to construct the matrices is to make each element some function of the indices, e.g., $A(i,j) = i + j$ or $B(i,j) = i - j + 1$. Now add them in three different ways,

   (a) with the inner loop over the column (second) index;

   (b) with the inner loop over the row (first) index;

   (c) using the built-in addition, $A = B + C$

   Time these methods. *Note:* Some compliers will automatically optimize your code, which will make this example not as interesting! Make sure you compile without the `-O` flag.

   To time a loop, you can use the `cpu_time` routine, as demonstrated in the following simple code.

```
1  ! sample_time.f
2  ! A simple example of timing a section of code.
3  !
4  ! AST 911, Spring 2008
5  ! Edward Brown, Michigan State University
6  !
7  program time_a_loop
```

```
8     implicit none
9     integer, parameter :: N  = 1024
10    real, dimension(N,N) :: a
11    integer :: i,j
12    real :: t0, t1
13
14    call cpu_time(t0)
15    do i = 1,N
16      do j = 1,N
17        a(i,j) = i+j
18      end do
19    end do
20    call cpu_time(t1)
21    print *,'time = ',t1-t0
22  end
```

2. Consider the multiplication of two matrices of size $N \times N$. First, verify that the number of floating-point operations required to multiply the two matrices together is proportional to N^3.

3. Generate a set of non-trivial square matrices with $N = 2^s$, $s = 4, \ldots, 10$. Now, for each size N, multiply the two matrices by each of the following methods.

   (a) a loop nest

   (b) the intrinsic `matmul` proceedure

   (c) if available, a tuned BLAS routine[1].

   Tabulate the run time for each method as a function of N.

[1] BLAS (Basic Linear Algebra Subprograms) routines are designed for high-performance matrix operations, and are the building blocks of LAPACK.

Suppose you had to perform the operation $C = A + B^T$, that is, $C(i,j) = A(i,j) + B(j,i)$. Here you can't avoid non-sequential memory access. To see how BLAS routines are tuned for performance, time the following two algorithms and see if there is a difference. Again, let the matrices be square and of rank $N = 1024$.

1. A straightforward loop nest

```
22  do i = 1,N
23    do j = 1,N
24      a(j,i) = a(j,i) + b(i,j)
25    end do
26  end do
```

   Note that elements in array b are not accessed sequentially.

2. Now work on 2×2 blocks, and cut the inner loop into two loops each of size $N/2$.

```
do i = 1,N,2
  do j = 1,N/2,2
    d(j,i)     = d(j,i) + b(i,j)
    d(j+1,i)   = d(j+1,i) + b(i,j+1)
    d(j,i+1)   = d(j,i+1) + b(i+1,j)
    d(j+1,i+1) = d(j+1,i+1) + b(i+1,j+1)
  end do
end do
do i = 1,N,2
  do j = N/2 + 1,N,2
    d(j,i)     = d(j,i) + b(i,j)
    d(j+1,i)   = d(j+1,i) + b(i,j+1)
    d(j,i+1)   = d(j,i+1) + b(i+1,j)
    d(j+1,i+1) = d(j+1,i+1) + b(i+1,j+1)
  end do
end do
```

What is the difference in memory access between method 1 and method 2? Is there a significant gain in performance?

Bibliography

J. R. Cash. Review Paper: Efficient numerical methods for the solution of stiff initial-value problems and differential algebraic equations. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 459(2032):797–815, 2003.

Randall J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes*. Cambridge University Press, third edition, 2007.

Lewis F. Richardson and J. Arthur Gaunt. The Deferred Approach to the Limit. Part I. Single Lattice. Part II. Interpenetrating Lattices. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 226(636-646):299–361, 1927.