Solving Hyperbolic PDEs in MATLAB

L.F. Shampine
Mathematics Department
Southern Methodist University, Dallas, TX 75275
lshampin@mail.smu.edu

May 31, 2005

1 Introduction

We develop here software in MATLAB to solve initial–boundary value problems for first order systems of hyperbolic partial differential equations (PDEs) in one space variable x and time t. We allow PDEs of three general forms, viz.

$$\begin{aligned} u_t &= f(x,t,u,u_x) && (1)\\ u_t &= f(x,t,u)_x + s(x,t,u) && (2)\\ u_t &= f(u)_x && (3) \end{aligned}$$

and we allow general boundary conditions. Solving PDEs of this generality is not routine and the success of our software is not assured. On the other hand, it is very easy to use and has performed well on a wide variety of problems.

Explicit central finite difference methods are quite attractive for hyperbolic PDEs of this generality. We have implemented four: A two-step variant of the Lax-Friedrichs (LxF) method [8], Richtmyer's two-step variant of the Lax-Wendroff (LxW) method [6], and the LxW method with a nonlinear filter [1] are available for all three forms. A variant of the Nessyahu-Tadmor (NT) method [4] is available for systems of form 3. The basic methods are generally recognized as effective, but the variants implemented have some advantages. For instance, the variant of LxF is dissipative and damps middle frequencies less than the usual scheme. The nonlinear filter can be quite helpful in dealing with the oscillations characteristic of LxW. The variant of NT is *much* better suited to the MATLAB problem solving environment (PSE).

An important part of this investigation was to devise a convenient way to deal with general boundary conditions. In the case of the NT method, little attention has been given to conditions other than periodicity, so in §4.2 we develop fully a treatment of general boundary conditions for systems of equations. To be sure, this is only one aspect of a user interface that we have crafted to make as easy as possible to use whilst still capable of solving a large class of problems.

With methods that are at most of order two, even the graphical accuracy appropriate to a PSE may require hundreds of mesh points. To accomplish this with run times appropriate to a PSE, it is necessary in MATLAB to vectorize the computation. This has influenced our choice of algorithms and in particular, dictated the variant of NT that we implemented. Along with a suitable user interface and careful coding, we have managed to reduce run times dramatically, a fact illustrated by examples in §5.

Our solver consists of two functions, `setup` and `hpde`, and five auxiliary functions. The functions and a good many examples of their use are available from the author. They run in MATLAB version 6.5 and later. Several of the examples are described briefly in this article and numerical results for three are presented in §6. These examples include systems of one to three PDEs, all three forms, and all the kinds of boundary conditions that we allow.

2 User Interface

We split the numerical solution of a system of PDEs into two parts. Every computation begins with a call of the form

```
sol = setup(form,pdefun,t,x,u,method,periodic,bcfun,Neumann)
```

that defines the problem and how it is to be solved. It also initializes a solution structure that can have any name, but is called here `sol`.

The PDEs are defined by `form` and `pdefun`. As stated in §1, the PDEs can have three forms. When `form` is 1, `pdefun` is a function that evaluates $f(t, x, u, u_x)$ and similarly when `form` is 3, it evaluates $f(u)$. When `form` is 2, `pdefun` is a cell array {`flux,source`} with `flux` and `source` being functions that evaluate $f(t, x, u)$ and $s(t, x, u)$, respectively. In the common event of $s(t, x, u)$ being a constant vector, it can be supplied as `source`. `pdefun` must be coded in a way that we illustrate for `form` 1: When called with a scalar `T` and arrays `X,U,U_x`, `pdefun` must return an array `V` with column `m` (which is `V(:,m)` in MATLAB) equal to $f($`T,X(m),U(:,m),U_x(:,m)`$)$ for each `m`. The number of columns is the same for all the arrays of this call, but this number varies from one call to the next.

The integration starts at time `t`. The solution is computed on a fixed mesh `x` in the interval $[a, b]$. The mesh points $a = x_1 < x_2 < \ldots < x_M = b$ must be equally spaced. (This is conveniently coded as `linspace(a,b,M)`.) The approximate solution at time `t` is provided in the array `u`, which is to say that the vector `u(:,m)` $\approx u(x_m,$ `t`$)$ for $m = 1, \ldots, M$. The string `method` indicates which of the four numerical methods discussed in §3 is to be used. The remaining arguments specify boundary conditions, so they are discussed in §4.

The solution structure `sol` is initialized in `setup` and thereafter used as both input and output for the solver `hpde`. Fields of particular interest are

- `sol.t`—current time
- `sol.x`—(fixed) mesh

- `sol.u`—approximate solution at current time
- `sol.nstep`—number of steps taken to reach current time

A call to `hpde` has the form

```
sol = hpde(sol,howfar,timestep)
```

Each call advances the integration from the current time t to $t + $`howfar` using step sizes specified by `timestep`. If `timestep` is a positive scalar, the solver takes steps of this size. If it is a function of the form `dt = timestep(dx,t,x,u)`, the solver calls this function at each step with the current approximation `u` to the solution at time `t`, as well as the mesh `x` and its constant spacing `dx`. The solver then takes a step of the size `dt` returned by the function. (In either case the last step is shortened as necessary to produce an approximate solution at $t + $`howfar`.) The two possibilities are illustrated by the example programs discussed in §6. `howfar` and a scalar `timestep` can be changed at each call.

It is interesting to contrast our approach to a more general one coded in FORTRAN. In §6 we present some numerical results for a model of heat transfer formulated by W.E. Schiesser [7, §3.1]. In the system

$$u_{1,t} + v\, u_{1,x} = c_1(u_2 - u_1) \tag{4}$$

$$u_{2,t} = c_2(u_1 - u_2) \tag{5}$$

the quantities v, c_1, c_2 are constant. The problem is set on $[0, 1]$ and because $v > 0$, appropriate initial–boundary conditions have the form

$$u_1(x, 0) = f(x), \quad u_2(x, 0) = h(x), \quad u_1(0, t) = g(t)$$

Schiesser provides a collection of tools for solving PDEs by the method of lines and in particular, subroutines for applying standard spatial difference operators. The user has to write a program to formulate initial and boundary conditions, apply suitable difference operators, and call upon standard software to integrate the resulting ODEs. A great virtue of the approach is that it applies to all types of PDEs. For hyperbolic PDEs like (4),(5), Schiesser uses the Runge–Kutta code RKF45 [9] for time integration. The `Schiesser` example program shows that it is much easier to formulate and solve (4),(5) with a solver for the task rather than a set of tools. Schiesser computes results for $t = 0 : 1 : 10$ and compares them to analytical values for $u_1(1, t)$. The analytical expressions involve special functions and integrals that must be evaluated numerically. He specifies homogeneous initial values and four boundary functions $g(t)$ that make this task somewhat easier. An incompatibility of initial and boundary values for some of the data sets results in a discontinuity of u_1 that propagates to the right. With $v = 2.031$ this discontinuity reaches $x = 1$ before he begins measuring the error there. As illustrated in §6, the visualization tools of the MATLAB PSE quickly reveal where $u_1(x, t)$ has its most interesting behavior.

It is also interesting to contrast our approach with one that is more specialized and coded in MATLAB. The `onedimwave` program of Stanoyevitch [12] solves equations of the form

$$u_{tt} = c^2(t, x, u, u_x)\, u_{xx} \tag{6}$$

with Dirichlet boundary conditions. To solve such a problem with `hpde`, we must write the equation as a first order system. It is straightforward to do this with variables $v_1 = u, v_2 = u_x, v_3 = u_t$. However, Stanoyevitch allows boundary conditions of the form $u(0,t) = A(t)$. Obviously $v_1(0,t) = A(t)$, but it also follows that $v_3(0,t) = A'(t)$. The `onedimwave` program has the user specify the number of equally spaced mesh points in an interval $[0, L]$ and the number of equally spaced points in a time interval $[0, T]$. A finite difference method specifically for the wave equation is used to compute an approximation at all points in space and time and return it as an array. For the numerical example of §5, this array is 352×802. This approach to output is less satisfactory for systems and finer meshes. Indeed, we solve this problem in `twostrings` as a system of 3 equations and use 1000 mesh points instead of 352. Our approach is to return answers only at specific times so as to reduce the storage required. Also, because we allow the user to adapt the step size to the solution by means of a `timestep` function, we do not know how many time steps will be required.

3 The Methods

The argument `method` of the `setup` function indicates which numerical method is to be used. It is a string that can have four values, namely 'LxF', 'LxW', 'SLxW', 'NT'. (The values are not case-sensitive.) In this section we discuss briefly certain aspects of the variants of the Lax–Friedrichs (LxF) and Lax–Wendroff (LxW) methods that we implement and review the Nessyahu–Tadmor (NT) method. Some of the examples that accompany the solver allow all four methods to be applied and many allow three. For instance, Pearson [5] develops a perturbation solution to the PDEs

$$u_t + uu_x + \eta_x = 0 \tag{7}$$
$$\eta_t + [u(1+\eta)]_x = 0 \tag{8}$$

for two sets of initial data, a periodic disturbance and an isolated disturbance, both of size ϵ. He presents plots for the surface elevation $\eta(x, 15)$ computed with perturbation and numerical methods. Our example program `Pearson` solves both problems with the PDEs written in form 3. The periodic problem is solved easily by all the methods and 100 mesh points provides acceptable resolution. The isolated disturbance is confined to $|x| < 1/2$. The discontinuous disturbance splits and moves in both directions, reaching the neighborhood of $x = \pm 15$ by $t = 15$. In `Pearson` this problem is solved on $[-20, 20]$ with the undisturbed values of zero imposed on the numerical solution at the boundaries. Because the interval is much larger, the program uses 1000 mesh points for this problem. LxW provides an acceptable solution, but $\eta(x, 15)$ has an oscillation that is not physical. The filtering of SLxW improves this. The first order LxF is qualitatively correct, but the solution is rather damped. The second order NT provides the best solution for this conservation law.

`hpde` is actually a driver that calls one of three functions depending on the form of the PDEs. `hpde1` and `hpde2` solve PDEs of forms 1 and 2, respectively. They implement both LxF and LxW and have smoothing of LxW as an option. The formulas differ in obvious ways because the forms of the equations are different and correspondingly the coding differs in obvious ways. `hpde3` solves PDEs of form 3 with NT. PDEs of this form are solved with the other methods by treating the form as a special case of form 2 and using `hpde2`.

3.1 LxF and LxW

Richtmyer's two-step Lax–Wendroff method [6] is indicated by the string 'LxW'. It is of order two and dissipative of order four. This popular method is dispersive, so we have provided an option to follow each step with a nonlinear filter to reduce the total variation of the numerical solution. The smoothed LxW method is specified by the string 'SLxW'. Engquist et alia [1] propose several nonlinear filters that can be used for this purpose. Their algorithms are developed for scalar problems and there is some discussion in the paper as to what should be done for systems. Like the authors of [10], we found that applying the filter by components was satisfactory. As discussed more fully in §5, we chose Algorithm 2.1 of [1] for `hpde`. When the solution is smooth, filtering has little effect, but it can be counterproductive because it flattens features. It is by no means a panacea, but we have found it to be surprisingly effective.

It is convenient to implement a two-step Lax–Friedrichs scheme along with the two-step Lax–Wendroff scheme of Richtmyer. It is specified with the string 'LxF'. In this scheme a half step is taken with LxF on a staggered mesh. If a second half step is taken with LxF, a solution is obtained on the original mesh. Richtmyer's LxW scheme is just a full step taken with this data and the leapfrog method. The usual LxF scheme is not dissipative, but it is shown in [8] that this variant is dissipative of order two. Moreover, this variant does not damp middle frequencies as much as the usual scheme. As a monotone method, the LxF scheme has attractive properties that make it useful when the solution is not smooth.

3.2 NT

Nessyahu and Tadmor [4] have developed a simple and effective method for solving a system of conservation laws (3) that is known as the NT method. They consider problems with initial values on the whole real line, but later Levy and Tadmor [3] considered how to treat inflow and outflow boundary conditions for scalar problems posed on a finite interval. In §4.2 we develop fully a treatment of general boundary conditions for systems of equations when a different form of the method is used.

Nessyahu and Tadmor consider two variants of their scheme. Their numerical experiments in [4] and those of later papers are all done with one variant, but we find the other to be much better suited to our purposes. To understand why and later to appreciate the boundary treatment, it will be necessary to review the derivation of the NT method for a scalar equation. The method approximates $u(x,t)$ on a uniform mesh $x_m = m\,\Delta x$. The cell I_p is the interval $|x - x_p| \leq \Delta x/2$ and its characteristic function, $\chi_p(x)$, has value 1 if $x \in I_p$ and 0 otherwise. The solution is approximated at time $t^n = n\,\Delta t$ and all x by the piecewise constant function

$$w(x,t^n) = \sum w_p^n\,\chi_p(x) \tag{9}$$

The quantity w_p^n approximates the cell average of the solution

$$w_p^n = \frac{1}{\Delta x}\int_{I_p} w(x,t^n)\,dx \approx \frac{1}{\Delta x}\int_{I_p} u(x,t^n)\,dx$$

It can also be regarded as an approximation to the point value $u(x_p,t^n)$. The approximate solution at t^{n+1} is computed on a staggered mesh which we define for all p as $x_{p+1/2} = x_p + \Delta x/2$ and similarly, $I_{p+1/2}$. Also, we write $\lambda = \Delta t/\Delta x$.

The step to time t^{n+1} begins by improving the piecewise constant approximation (9), a process called reconstruction. Information about the solution from adjacent cells is used to form a piecewise linear approximation

$$w(x,t^n) = \sum \left[w_p^n + \frac{w_p'}{\Delta x}\,(x - x_p) \right] \chi_p(x) \tag{10}$$

This form preserves w_p^n as the cell average and its interpretation as an approximation to $u(x_p,t^n)$, so the question is what to use for the slope of the line. Although there are other possibilities considered in [4], the simplest is

$$\frac{w_p'}{\Delta x} = MM\left(\frac{w_{p+1}^n - w_p^n}{\Delta x}, \frac{w_p^n - w_{p-1}^n}{\Delta x} \right) \tag{11}$$

Here MM is the MinMod limiter which can be defined for two scalar arguments as

$$MM(a,b) = \frac{1}{2}\,(sign(a) + sign(b))\min(|a|,|b|)$$

Although the expression (11) shows clearly that we consider two natural approximations to the slope and choose one to reduce oscillation in the numerical solution, we actually compute the scaled quantity w_p'.

Let $W(x,t)$ be the solution of equation (3) with $W(x,t^n)$ equal to the $w(x,t^n)$ of (10). Integrating the equation over the rectangle $I_{p+1/2} \times [t^n, t^{n+1}]$ and a little manipulation provides an analytical expression for the cell average of $W(x,t^{n+1})$ in terms of the cell average of $W(x,t^n)$ and integrals of the flux f along the sides of the rectangle:

$$\frac{1}{\Delta x}\int_{x_p}^{x_{p+1}} W(x,t^{n+1})\,dx = \frac{1}{\Delta x}\int_{x_p}^{x_{p+1}} W(x,t^n)\,dx$$
$$+\lambda\left[\frac{1}{\Delta t}\int_{t^n}^{t^{n+1}} f(W(x_{p+1},\tau))\,d\tau - \frac{1}{\Delta t}\int_{t^n}^{t^{n+1}} f(W(x_p,\tau))\,d\tau \right]$$

Using the initial data (10) we can compute analytically the cell average at time t^n. On the cell $I_{p+1/2}$ the initial data is smooth (linear) except for a discontinuity at $x_{p+1/2}$. Nessyahu and Tadmor point out that if the step size satisfies a CFL condition, the effects of the discontinuity do not reach the sides of the rectangle by time t^{n+1}. This is a crucial observation because it tells us that with an appropriate restriction on the step size, the integrands in the integrals over the sides of the rectangle are smooth enough that we can obtain a second order approximation to the integral with the midpoint rule. For the right side of the rectangle, this rule requires $f(W(x_{p+1}, t^{n+1/2}))$. We can approximate the argument by

$$\begin{aligned} W(x_{p+1}, t^{n+1/2}) &= W(x_{p+1}, t^n) + \frac{\Delta t}{2} \frac{\partial W}{\partial t}(x_{p+1}, t^n) + \dots \\ &= W(x_{p+1}, t^n) + \frac{\Delta t}{2} \frac{\partial f}{\partial x}(W(x_{p+1}, t^n)) + \dots \qquad (12) \\ &= w^n_{p+1} + \frac{\Delta t}{2} \frac{\partial f}{\partial x}(w^n_{p+1}) + \dots \end{aligned}$$

One possibility is to proceed much as we did with the derivative of the solution, namely use values $F_p = f(w^n_p)$ to approximate the partial derivative scaled by Δx with

$$F'_{p+1} = MM(F_{p+2} - F_{p+1}, F_{p+1} - F_p)$$

With these values we have

$$w^{n+1/2}_{p+1} = w^n_{p+1} + \frac{\lambda}{2} F'_{p+1} \qquad (13)$$

Nessyahu and Tadmor favor expressing the partial derivative of equation (12) in terms of the Jacobian of f,

$$W(x_{p+1}, t^{n+1/2}) = w^n_{p+1} + \frac{\Delta t}{2} \frac{\partial f}{\partial u}(W(x_{p+1}, t^n)) \frac{\partial W}{\partial x}(x_{p+1}, t^n) + \dots$$

This leads to the approximation

$$w^{n+1/2}_{p+1} = w^n_{p+1} + \frac{\lambda}{2} \frac{\partial f}{\partial u}(w^n_{p+1})\, w'_{p+1} \qquad (14)$$

Nessyahu and Tadmor prove similar theoretical results for both variants, but for reasons we take up in §5, we find (13) to be much more satisfactory for our purposes.

Assembling these ingredients, the version of the NT algorithm that we implement goes as follows: At time t^n we begin with a piecewise constant approximation (9). For all p, we calculate

$$w'_p = MM(w^n_{p+1} - w^n_p, w^n_p - w^n_{p-1}) \qquad (15)$$

evaluate $F_p = f(w^n_p)$, and then calculate

$$F'_p = MM(F_{p+1} - F_p, F_p - F_{p-1})$$

We form

$$w_p^{n+1/2} = w_p^n + \frac{\lambda}{2} F_p'$$

evaluate $f(w_p^{n+1/2})$, and then calculate

$$\begin{aligned} w_{p+1/2}^{n+1} &= \frac{1}{2}\left(w_p^n + w_{p+1}^n\right) + \frac{1}{8}\left(w_p' - w_{p+1}'\right) \\ &\quad + \lambda\left[f(w_{p+1}^{n+1/2}) - f(w_p^{n+1/2})\right] \end{aligned} \tag{16}$$

This provides a piecewise constant approximation at time t^{n+1} on cells $I_{p+1/2}$. When we solve problems defined for all x, the next time step is not unduly complicated by this staggered mesh, but there are obvious difficulties at the boundaries when the interval in x is finite.

Nessyahu and Tadmor precede their derivation of the second order NT method in [4] with the simpler derivation of a first order method. With the interpretation that $w_p^n \approx u(x_p, t^n)$, this method is equivalent to the variant of the Lax-Friedrichs method that we implement. The nice properties they prove for the method augment those cited in §3.

4 Boundary Conditions

Our solver allows quite general boundary conditions. In this section we begin with the design of the software and then discuss some of the details of implementation. In the case of the NT method, we must work out a suitable treatment.

General boundary conditions are specified by means of a function of the form `[uL,uR] = bcfun(t,uLex,uRex)`. At each step the solver computes approximations `uLex` and `uRex` to $u(a,t)$ and $u(b,t)$, respectively. As the names suggest, they are computed by extrapolation from the interior. They are passed to `bcfun` along with the current time `t`. `bcfun` defines the components of `uL` and `uR` so as to impose the desired boundary conditions at x_1 and x_M, respectively. In a simple case like an incoming flow for a particular component, `bcfun` would return to the solver a specific value for this component. Similarly, if the flow is out of the region for a particular component, `bcfun` would return the value input for this component because it was computed using values interior to the region and a one-sided formula. Swope and Ames [13] formulate the oscillation of a string as it is traversed and wound on a bobbin as a pair of first order PDEs. There are Dirichlet conditions at both ends of the interval because one end of the string is fixed and the other moves in a prescribed way. Our example program `threadline` solves the problem with parameter values corresponding to Fig. 10 of [13]. Initializing the boundary values to values extrapolated from the interior is accomplished easily in `bcfun` by choosing the same names for input and output variables and then the Dirichlet conditions on the first component are coded as

```
function [uL,uR] = bcfun(t,uL,uR)
    uL(1) = 0; uR(1) = 0.1*sin(t*pi/2);
```

Using the current `t` and tentative approximations at the boundaries, a wide variety of boundary conditions can be implemented by means of `bcfun`. This way of handling boundary conditions gives the user great flexibility without exposure to the details of the method and how the conditions are implemented.

Periodic boundary conditions are included in the general case, but extrapolation is not the best way to treat this kind of boundary condition. Accordingly, there is an optional argument `periodic` in `setup` that is set `true` to impose periodic boundary conditions and otherwise, `false` (or `[]`). Implementation of this special case is straightforward for all the methods because periodicity is used to compute `u(:,1)` $\approx u(a,t)$ just as at an interior mesh point and its value is assigned to `u(:,M)` by periodicity.

There is another kind of boundary condition that is common, but not included in the general case, namely homogeneous Neumann boundary conditions. If there are any conditions of this kind, the user specifies them with the optional argument `Neumann` in `setup`. It is a cell array with two entries that specify the conditions at the two ends of the interval. If $u_i(a,t)_x = 0$ for some components i, the indices i are provided as a vector `NeumannL`. There may be no such components, in which case this vector is the empty array `[]`. Homogeneous Neumann boundary conditions at the right end are similarly indicated with a vector `NeumannR`. The argument `Neumann` is then {`NeumannL,NeumannR`}. There may be no other boundary conditions, as with the `shocktube` example discussed in §6, but if there are, they are imposed first and then the homogeneous Neumann conditions. For instance, the `traffic` example discussed in §6 computes default values at both ends by extrapolation and then imposes a homogeneous Neumann condition at the right end.

4.1 LxF and LxW

The variant of the Lax-Friedrichs method that we implement involves a staggered mesh, so it is convenient to code it as two half steps from values on the mesh `sol.x` to values on the same mesh. The first half step is also the first half step of the variant of the Lax-Wendroff method that we implement. Because the mesh is staggered, we do not need to construct approximate solutions on the boundaries at the first half step for either method. In the next half step these explicit methods allow us to compute approximations at all interior points of `sol.x` without consideration of values at the boundary. A common and effective technique for obtaining boundary values then is to extrapolate from interior points or equivalently, use one-sided formulas. For both methods we compute tentative approximations at the boundaries by linear extrapolation. The order of convergence is not reduced for either method by extrapolation of this order. The methods are dissipative, so the effects of the boundary treatment are localized. This approach fits well with our design which passes tentative values to the user for imposition of boundary conditions and it has proved quite satisfactory in our experiments.

4.2 NT

An effective approach to imposing boundary conditions on other explicit, second-order methods is to compute a tentative solution on the boundary using data obtained by linear extrapolation of values at points interior to the region. Here we propose something along these lines for the NT method. The integration begins with input values w_p^0 that can be interpreted as $u(x_p, t^0)$ for $p = 1, \ldots, M$. Because the grid is staggered, we proceed by double steps, first taking a step from an approximation in Case I and then a step from an approximation in Case II.

Case I Suppose that we have a piecewise constant approximate solution $w(x, t^n)$ on the cells I_p for $p = 1, \ldots, M$. This case is distinguished from the pure initial value problem discussed in §3.2 only by the boundary cells $I_1 = [x_1, x_{3/2}]$ and $I_M = [x_{M-1/2}, x_M]$ being half the size of the interior cells. In the reconstruction we can compute a (scaled) slope w'_p in the usual way for the interior cells, but we must proceed differently for w'_1 because we do not have the value w_0^n that appears in the recipe (15). Because the reconstruction uses a constant slope on cells, linear extrapolation from the interior amounts to taking $w'_1 = w'_2$. A good reason for proceeding in this way is that the value w'_2 has benefited from the application of the MinMod limiter to control oscillation in the solution. Similarly, we take $w'_M = w'_{M-1}$. We can now apply the usual formula to compute $w_{p+1/2}^{n+1}$ for $p = 1, \ldots, M-1$ and so arrive at a piecewise constant approximation $w(x, t^{n+1})$ of a form that we consider as a second case.

Case II Suppose that we have a piecewise constant approximate solution $w(x, t^{n+1})$ on the cells $I_{p+1/2}$ for $p = 1, \ldots, M-1$. That is, $w(x, t^{n+1})$ is a constant $w_{p+1/2}^{n+1}$ on $[x_p, x_{p+1}]$. In the reconstruction we can form slopes $w'_{p+1/2}$ in the usual way for $p = 2, \ldots, M-2$. At the boundaries we extrapolate from the interior to define $w'_{3/2} = w'_{5/2}$ and $w'_{M-1/2} = w'_{M-3/2}$. We can now apply the usual formula to compute w_p^{n+2} for the interior cells $p = 2, \ldots, M-1$. To obtain an approximation $w(x, t^{n+2})$ of the form considered in Case I, we must develop a special formula to approximate the average of the solution on the half cell $[x_1, x_{3/2}]$, a value that we denote as w_1^{n+2} so as to correspond to Case I. We proceed just as at an interior point, but now integrating the conservation law over $[x_1, x_{3/2}] \times [t^{n+1}, t^{n+2}]$. After a little manipulation we get

$$\frac{2}{\Delta x} \int_{x_1}^{x_{3/2}} W(x, t^{n+2})\, dx = \frac{2}{\Delta x} \int_{x_1}^{x_{3/2}} W(x, t^{n+1})\, dx$$
$$+2\lambda \left[\frac{1}{\Delta t} \int_{t^{n+1}}^{t^{n+2}} f(W(x_{3/2}, \tau))\, d\tau - \frac{1}{\Delta t} \int_{t^{n+1}}^{t^{n+2}} f(W(x_1, \tau))\, d\tau \right] \quad (17)$$

Along the base of this rectangle the reconstruction is

$$W(x, t^{n+1}) = w_{3/2}^{n+1} + \frac{w'_{3/2}}{\Delta x} (x - x_{3/2}) \quad (18)$$

so the first integral on the right hand side of (17) is

$$\frac{2}{\Delta x}\int_{x_1}^{x_{3/2}} W(x,t^{n+1})\,dx = w_{3/2}^{n+1} - \frac{1}{4}\,w'_{3/2}$$

In computing w_2^{n+2} we formed an approximation to the integral over the right side of this rectangle that we can reuse here, namely

$$\frac{1}{\Delta t}\int_{t^{n+1}}^{t^{n+2}} f(W(x_{3/2},\tau))\,d\tau \approx f(w_{3/2}^{n+3/2})$$

Here the intermediate value is given by the usual formula

$$w_{3/2}^{n+3/2} = w_{3/2}^{n+1} + \frac{\lambda}{2}\,F'_{3/2}$$

We proceed similarly on the left side, but there are some complications. The approximate solution at x_1 is given by (18) as

$$w_1^{n+1} = w_{3/2}^{n+1} - \frac{1}{2}\,w'_{3/2} \tag{19}$$

The usual formula for the intermediate value

$$w_1^{n+3/2} = w_1^{n+1} + \frac{\lambda}{2}\,F'_1 \tag{20}$$

involves F'_1. Extrapolating from the interior, we take $F'_1 = F'_{3/2}$. With this intermediate value we can approximate the integral on the left side of the rectangle in the usual way,

$$\frac{1}{\Delta t}\int_{t^n}^{t^{n+1}} f(W(x_1,\tau))\,d\tau \approx f(w_1^{n+3/2})$$

With approximations to all the integrals on the right hand side of (17), we can now approximate the cell average on the left hand side by

$$w_1^{n+2} = w_{3/2}^{n+1} - \frac{1}{4}\,w'_{3/2} + 2\lambda\left[f(w_{3/2}^{n+3/2}) - f(w_1^{n+3/2})\right] \tag{21}$$

In summary, we deal with the left boundary by defining two values by extrapolation from the interior, $w'_{3/2} = w'_{5/2}$ and $F'_1 = F'_{3/2}$. With them we can evaluate w_1^{n+1} using (19), then $w_1^{n+3/2}$ using (20), and finally w_1^{n+2} using (21). The equivalent at the right end is to define by extrapolation $w'_{M-1/2} = w'_{M-3/2}$ and $F'_M = F'_{M-1/2}$. With them we can evaluate

$$w_M^{n+1} = w_{M-1/2}^{n+1} + \frac{1}{2}\,w'_{M-1/2}$$

then

$$w_M^{n+3/2} = w_M^{n+1} + \frac{\lambda}{2}\,F'_M$$

and finally

$$w_M^{n+2} = w_{M-1/2}^{n+1} + \frac{1}{4}\, w'_{M-1/2} + 2\lambda \left[f(w_M^{n+3/2}) - f(w_{M-1/2}^{n+3/2}) \right]$$

With these values we have a piecewise constant approximation of the form considered in Case I. The vector w_1^{n+2} is a tentative approximation to $u(x_1, t^{n+2})$, so we supply it and the corresponding approximation at the other end of the interval to `bcfun`. We replace the tentative values with those returned from `bcfun` to complete Case II and advance the integration to t^{n+2}.

5 Vectorization

We began this project by drafting solvers that advanced the solution one mesh point at a time. With hundreds of mesh points, LxF and LxW were slow in Matlab and NT was impractical. The quadrature codes of Matlab require that the integrand be evaluated at all points of a mesh in a single call. Something similar is very advantageous when solving boundary value problems for ODEs. With this in mind, we suspected that requiring users to evaluate the PDEs at all mesh points in a single call could be used to speed up the solvers greatly, and so it turned out. It is generally not hard to vectorize the evaluation of a system of PDEs and even if it is, we found that the big advantage lies in vectorizing the solver and evaluating the PDEs at all mesh points in one call rather than vectorizing the evaluation of the PDEs *per se*. As we began rewriting the solvers, vectorization caused us to reconsider some of the algorithms.

To illustrate the effects of vectorization, we prepared a version of the solver for PDEs of form 1 that uses LxF and does the array computations one column at a time. In particular, it evaluates the PDEs one mesh point at a time. All the run times reported here were obtained with Matlab v. 7.0.0 and a moderately fast PC. Run times depend strongly on the circumstances, so the values we give are most useful as a rough guide to relative costs. To measure run time we place `tic` and `toc` around a loop that calls `setup` and `hpde` 10 times and divide the elapsed time reported at the end of the run by 10. It is better practice to solve the system (7), (8) in the conservative form 3, but to study the effects of vectorization, we solved it with the PDEs written in form 1. In terms of the vector $V = (u, \eta)^T$, the function can be coded as

```
function F = pdes(t,x,V,V_x)
    F = zeros(size(V));
    F(1,:) = - ( V(1,:).*V_x(1,:) + V_x(2,:) );
    F(2,:) = - ( V_x(1,:).*(1 + V(2,:)) + V(1,:).*V_x(2,:) );
```

A conventional coding of this function would evaluate the PDEs one mesh point at a time. In our design they must be evaluated for all the mesh points in a single call, which can be coded as

```
function F = spdes(t,x,V,V_x)
    F = zeros(size(V));
    for m = 1:size(V,2)
        F(1,m) = - ( V(1,m)*V_x(1,m) + V_x(2,m) );
        F(2,m) = - ( V_x(1,m)*(1 + V(2,m)) + V(1,m)*V_x(2,m) );
    end
```

We see that only a few changes are needed to vectorize the computation. All the input arrays have the same number of columns, but this number can vary from one call to the next. Notice how the preallocation of storage for `F` accounts for this. Solving the isolated disturbance problem of the `Pearson` example with the vectorized solver and `pdes` took 4.27s. Replacing `pdes` with `spdes` increased the average run time to 4.41s. The PDEs are simple and the JIT accelerator is doing a good job of evaluating the function efficiently, so the penalty is not at all large. On the other hand, the version of the solver that does array operations a column at a time and evaluates the PDEs one mesh point at a time took 82.18s! In §6 we solve a traffic model of one PDE that arises in form 1. With vectorization this problem is solved with LxF in 0.44s and without, in 13.08s. The kind of difference in run time seen in these two examples is of the utmost importance in a problem solving environment.

In §2 we discussed a MATLAB program that Stanoyevitch wrote to solve (6) with Dirichlet boundary conditions. He coded `onedimwave` to take advantage of array operations and requires the user to vectorize evaluation of the boundary functions. Example 12.6 of [12] studies the propagation of a wave through two strings of different density. Stanoyevitch notes that a "very large" number of mesh points and time steps are needed to deal with discontinuities in the data and even then, there are small oscillations in the solution at the end of the run. We found that the average time spent in `onedimwave` was 20.86s. We solve an equivalent set of 3 PDEs of form 1 in the `twostrings` program. Using the same mesh points and constant time step, the computation took an average of 0.44s using LxF, 0.46s using LxW, and 0.66s using SLxW. Evidently our approach to the task was very advantageous. Indeed, it is so much faster that we specify 1000 mesh points in `twostrings` and just reduced this to 352 for the comparison with `onedimwave`.

We tried two of the nonlinear filters proposed in [1], namely Algorithms 2.1 and 2.2. An important distinction is that the first makes only one pass over the data and the second may make repeated passes. One difficulty with implementing these filters efficiently is that we apply them by components and they treat components differently. On the other hand, all our examples involve at most 3 equations. It is not clear to us how to vectorize the application of these filters. Though Algorithm 2.2 sometimes provided a better solution, we found Algorithm 2.1 to be satisfactory and so did the authors of [10]. Because Algorithm 2.1 proved to be remarkably faster in MATLAB, we chose it for `hpde`.

In §3.2 we sketched the derivation of two variants of the NT method. The user must supply a function for $f(u)$ in either case, but if the variant (14) is adopted, a function for the Jacobian must also be supplied. This inconvenience is already a strong argument in favor of the variant (13) for a problem solving environment. Vectorizing functions is critical to the efficient solution of PDEs in MATLAB and this is much more complicated when a Jacobian matrix has to be evaluated at all the mesh points and the result returned as a multidimensional array. One reason Nessyahu and Tadmor favor (14) is that it requires fewer applications of the MinMod limiter. However, this limiter can be coded efficiently in MATLAB as a single command, even for systems: If a is an array with each column corresponding to a vector solution at a mesh point and b is a similar array, the MinMod operation can be applied to corresponding columns of a and b with

```
    v = 0.5*(sign(a) + sign(b)) .* min(abs(a),abs(b));
```

Array operations and other built-in commands execute much faster in MATLAB than equivalent scalar operations, so evaluation of the MinMod limiter is inexpensive in this computing environment. Considering all these issues, the variant (13) was clearly the better choice for `hpde`.

6 Numerical Examples

Many examples are available from the author along with the solver. One purpose they serve is to provide examples of vectorizing the evaluation of PDEs. Typically an example has a menu of methods and often a menu of forms for the PDEs. As illustrated here, the examples range from 1 to 3 PDEs and include all forms of the PDEs. All the kinds of boundary conditions that we allow are also represented.

Many texts study models of traffic flow. After discussing such models Gustafson [2, p. 266 ff.] sets the computational Problem B.9 that we solve in the `traffic` program. The PDE arises in form 1, namely $\rho_t + c(\rho)\rho_x = 0$ with $c(\rho) = 76.184 - 17.2\log(\rho)$. The initial density $\rho(x,0)$ has a sharp peak. Gustafson reports that the leapfrog method and two codes based on the Lax-Wendroff method "...failed to produce adequate backward movement of the traffic bulge." `traffic` reproduces the figure on p. 393 that was computed using the method of characteristics. The figure shows $\rho(x,0)$ on [0,1] and the shock that has developed at $t = 0.009$. The interval is big enough that the peak does not reach the left boundary by this time, so we specify a homogeneous Neumann boundary condition at $x = 1$ and no boundary condition at $x = 0$. It is convenient to write a function that evaluates $c(\rho)$ for an array `rho` and then use it to evaluate the PDEs and a time step that satisfies a CFL condition of 0.9:

```
function v = c(rho)
    v = 76.184 - 17.2*log(rho);

function F = pdefun(t,x,rho,rho_x)
    F = - c(rho).*rho_x;
```

```
function dt = timestep(dx,t,x,rho)
    dt = 0.9*dx/max(abs(c(rho)));
```

Evidently vectorization of these functions is quite easy. Partly because the PDE is written in non-conservative form and partly because of the shock, we use a rather fine mesh, namely 1000 equally spaced points. We measured run times as explained in §5 and found that the average run time for LxF was 0.44s, for LxW was 0.46s, and for SLxW was 0.49s.

In §2 we discussed a pair of equations (4),(5) of form 2 that W.E. Schiesser [7, §3.1] solved with the method of lines. To facilitate evaluation of a semi-analytical solution, he considers homogeneous initial values and four boundary functions. The `Schiesser` program has a menu for the four data sets and another for the three applicable methods. Schiesser uses only 21 equally spaced points in x and reports the error in $u_1(1,t)$ for $t = 0 : 1 : 10$. A mesh of 21 points is rather crude for the first and second order methods of our solver. Even so, when using LxF, the maximum error ranged from 4×10^{-4} to 3×10^{-3} for the four data sets and when using either LwW or SLxW, it ranged from 4×10^{-4} to 8×10^{-4} . To show the movement of a discontinuity more clearly, the `Schiesser` program uses a mesh of 100 points and plots solution profiles for $t = 0.1 : 0.1 : 0.5$. It then displays the overall behavior of u_1 with a surface plot for $t = 1 : 1 : 10$. Figure 1 shows profiles computed when $u_1(0,t) = \exp(-0.1t)$. There is a typical oscillation and overshoot in the solution computed with LxW that is still present in this solution computed with SLxW, but the filter has suppressed the oscillation and reduced the overshoot. A monotone numerical solution is computed with LxF.

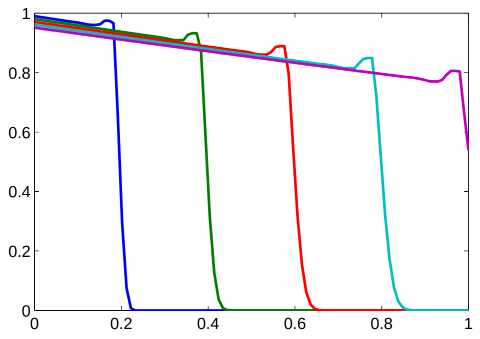

Figure 1: $u_1(x,t)$ for $t = 0.1 : 0.1 : 0.5$ using SLxW in `Schiesser`.

Shock tube problems with Riemann data are widely used for testing PDE solvers. The `RIM` program solves the three Euler equations of gas dynamics with the two sets of initial data due to Sod and Lax that constitute the RIM1 and RIM2 examples of [4]. Nessyahu and Tadmor use a constant time step in [4], but `RIM` computes an appropriate time step as in Sod [11]. Thomas [14] uses a third set of initial data for the HW 0.0.3 example studied throughout his text. All the shock tube problems have solutions with a shock, contact discontinuity, and expansion fan. We solve Thomas' example in the `shocktube` program. He discusses boundary conditions at length and then solves the PDEs with homogeneous Neumann boundary conditions. He uses 200 equally spaced mesh points and a constant time step. Corresponding to plots in [14], `shocktube` sets `howfar` to 0.2 and plots solution profiles for times `0:0.2:1`. A menu allows any of the four methods to be specified. It is best to write the PDEs in the conservative form 3, but for illustrative purposes, all three forms are available. Thomas writes the PDEs in terms of density, momentum, and energy, but plots density, velocity, and pressure. Velocity is computed from momentum and density, and pressure is computed from the equation of state. Figures 2 and 3 show that the nonlinear filter of SLxW has done a remarkably good job of dealing with the oscillations that are characteristic of LxW. As exemplified by Fig. 4, NT provides a quality solution for a system of conservation laws. As with the `traffic` program, we measured the average time spent in `setup` and the five calls to `hpde` in each run. In our experience the values displayed in Table 1 for `shocktube` are representative, though often LxW is a little slower than LxF. Filtering is not expensive despite the calculation not being vectorized. SLxW is more expensive relative to LxW for `shocktube` than `traffic` because there are more solution components. NT provides a better solution than SLxW, often a much better solution, but it is substantially more expensive and, of course, is limited to PDEs of form 3. LxF has the same qualitative behavior as NT and can be used for all three forms. It is only first order, but it is so much faster than NT that an equally good solution might well be computed in the same run time by using a much finer mesh.

Method	Form 1	Form 2	Form 3
LxF	0.20	0.22	0.22
LxW	0.20	0.22	0.22
SLxW	0.27	0.28	0.28
NT			0.74

Table 1: Run times (in seconds) for `shocktube`.

References

[1] B. Engquist, P. Lötstedt, and B. Sjögreen, Nonlinear filters for efficient shock computation, Math. Comp., 52 (1989) 509–537.

[2] K.E. Gustafson, Partial Differential Equations and Hilbert Space Methods, 2nd ed., Wiley, New York, 1987.

[3] D. Levy and E. Tadmor, Non-oscillatory boundary treatment for staggered central schemes, `http://www.cscamm.umd.edu/~tadmor/`

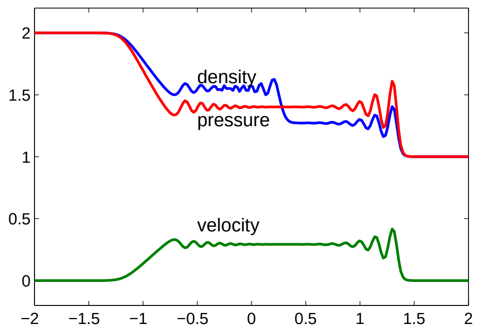

Figure 2: Result of `shocktube` at $t = 1$ using LxW and form 3.

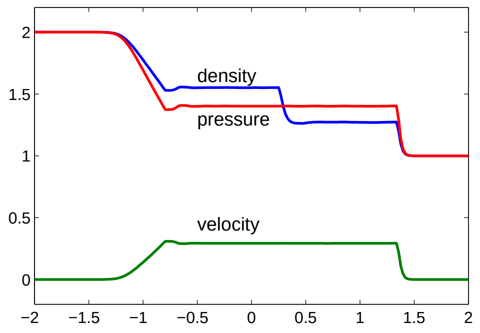

Figure 3: Result of `shocktube` at $t = 1$ using SLxW and form 3.

[4] H. Nessyahu and E. Tadmor, Non-oscillatory central differencing for hyperbolic conservation laws, J. Comp. Phys., 87 (1990) 408–463.

[5] C.E. Pearson, Dual time scales in a wave problem governed by coupled nonlinear equations, SIAM Review, 23 (1981) 425–433.

[6] R.D. Richtmyer and K.W. Morton, Difference Methods for Initial–Value Problems, 2nd ed., Wiley Interscience, New York, 1967.

[7] W.E. Schiesser, Computational Mathematics in Engineering and Applied Science, CRC Press, Boca Raton, FL, 1994.

[8] L.F. Shampine, Two-step Lax–Friedrichs method, Appl. Math. Letters, to appear.

[9] L.F. Shampine and H.A. Watts, The art of writing a Runge-Kutta code, Part I, pp. 257–275 in J.R. Rice, ed., Mathematical Software III, Academic,

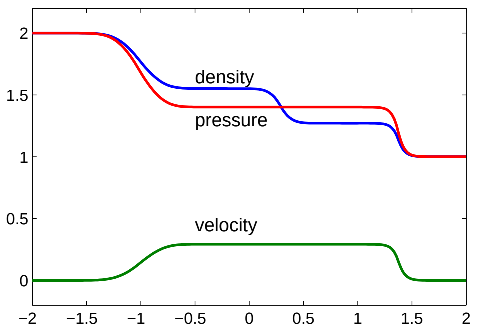

Figure 4: Result of `shocktube` at $t = 1$ using NT and form 3.

New York, 1977, and The art of writing a Runge-Kutta code, II, Appl. Math. Comp. 5 (1979) 93–121.

[10] W. Shyy, M.-H. Chen, R. Mital, and H.S. Udaykumar, On the suppression of numerical oscillations using a non-linear filter, J. Comp. Phys., 102 (1992) 49–62.

[11] G.A. Sod, A survey of several finite difference methods for systems of non-linear hyperbolic conservation laws, J. Comp. Phys., 27 (1978) 1–31.

[12] A. Stanoyevitch, Introduction to Numerical Ordinary and Partial Differential Equations Using MATLAB, Wiley, New York, 2005.

[13] R.D. Swope and W.F. Ames, Vibrations of a moving threadlne, J. Franklin Inst., 275 (1963) 36–55.

[14] J.W. Thomas, Numerical Partial Differential Equations *Finite Difference Methods*, Springer, New York, 1995, and Numerical Partial Differential Equations *Conservation Laws and Elliptic Equations*, Springer, New York, 1999.