CLAWPACK Version 4.0 User's Guide

Randall J. LeVeque
University of Washington

DRAFT VERSION
January 2, 2000

http://www.amath.washington.edu/~claw/

Authors.

Most of the one-dimensional and two-dimensional CLAWPACK routines were written by

R. J. LeVeque
University of Washington
Department of Applied Mathematics
Box 352420
Seattle, WA 98195-2420

The three-dimensional routines were written by

Jan Olav Langseth
Norwegian Defence Research Establishment
PO Box 25
N-2007 Kjeller
Norway

The adaptive mesh refinement part of AMRCLAW was written by

Marsha Berger
Courant Institute, NYU
251 Mercer St.
New York, NY 10012

Acknowledgements.

Numerous students and other users have contributed towards this software, by finding bugs, suggesting improvements, and exploring its use on new applications.

Development of this software has been supported in part by

NSF Grants DMS-8657319, DMS-9204329, DMS-9303404, and DMS-9505021, DMS-96226645,
DOE Grants DE-FG06-93ER25181, DE-FG03-96ER25292, DE-FG02-88ER25053,
DE-FG02-92ER25139,
AFOSR grant F49620-94-0132,
The Norwegian Research Council (NFR) through the program no. 101039/420.
The Scientific Computing Division at the National Center for Atmospheric Research (NCAR).

Copyright and Usage Restrictions:

Contents

1 The basic CLAWPACK software **5**
1.1 Version 4.0 5
1.2 References 6
1.3 Problem format 6
1.4 Obtaining CLAWPACK 7
1.4.1 Local users at `amath.washington.edu` 7
1.4.2 From the web 7
1.4.3 Anonymous ftp 8
1.5 Getting started 8
1.5.1 Creating the directories 8
1.5.2 Environment variables for the path 8
1.5.3 Compiling the code 9
1.5.4 Makefiles 9
1.5.5 MATLAB graphics 9
1.6 Using CLAWPACK — A guide through `example1` 9
1.6.1 The main program (`driver.f`) 10
1.6.2 The initial conditions (`qinit.f`) 11
1.6.3 The `claw1ez` routine 11
1.6.4 Boundary conditions 11
1.6.5 The Riemann solver 11
1.6.6 Other user-supplied routines and files 12
1.6.7 The input file `claw1ez.data` 13
1.7 Auxiliary arrays and `setaux.f` 15
1.8 An acoustics example 17
1.9 Two space dimensions (`claw2ez.f`) 17
1.9.1 Riemann solvers 18
1.10 Three space dimensions (`claw3ez.f`) 18
1.11 Plotting results with MATLAB 19
1.12 Other applications in `claw/applications` 20

2 Adaptive Mesh Refinement and AMRCLAW **21**
2.1 The input file `amr2ez.data` 21
2.2 Boundary conditions 23
2.3 Plotting results with MATLAB 23
2.4 The adaptive algorithm 24
2.5 Error estimation and regridding 25
2.6 Comments and warnings 25

Chapter 1

The basic CLAWPACK software

CLAWPACK (Conservation LAWs PACKage) is a package of Fortran subroutines for solving time-dependent hyperbolic systems of partial differential equations in 1, 2, and 3 space dimensions. This includes nonlinear systems of conservation laws, but one can also solve nonconservative hyperbolic systems and systems with variable coefficients, as well as systems including source terms.

1.1 Version 4.0

This version is still being developed. Check the webpage
`http://www.amath.washington.edu/~claw/changes.html`
for recent changes, particularly if you are using the version on the `amath` system and your code stops working.

The basic CLAWPACK algorithms in Version 4.0 are the same as in previous versions. The primary differences are:

- A new set of routines `claw1ez`, `claw2ez`, `claw3ez` are included which should make it easier to apply CLAWPACK to standard problems. These driver routines read in parameters from a data file that is assumed to be in a standard form and make the appropriate call to the CLAWPACK routines. It is still possible to call the lower-level CLAWPACK routine directly as before, although the calling sequences have changed slightly.
- When using the `ez` routines, it is assumed that the user-supplied subroutines have standard names, *e.g.*, the one-dimensional Riemann solver should be called `rp1`, since these names are no longer passed in by the user.
- Subroutines `setprob` and `setaux` are assumed to exist, which set up problem-specific parameters and the `aux` array respectively. Default versions in the CLAWPACK library which do nothing can be used if these routines are not needed.
- A new routine `b4step1` is assumed to exist by the routine `claw1` (even if `claw1ez` is not being used). This routine is called every time step before the routine `step1` is called. Similarly in two dimensions.
- Output is produced in a standard format which can be used by a set of MATLAB routines for graphical display, called `plotclaw1` and `plotclaw2`.
- The adaptive mesh refinement code AMRCLAW is now included and has been modified to expect essentially the same input and user subroutines as CLAWPACK. Output produced by AMRCLAW is in the same format and can be viewed with the same set of MATLAB routines. This makes it very easy to convert a code running in CLAWPACK to use the adaptive mesh refinement version.

1.2 References

The one- and two-dimensional wave-propagation algorithms are described in the paper [9]. Familiarity with this paper is assumed. The three-dimensional algorithms are developed and analyzed in [7]. The ideas are presented for the relatively simple case of the advection equation in two and three dimensions in [8]. Other papers where these algorithms are described or further developed include [5], [6], [10].

The adaptive mesh refinement routines used in `amrclaw` were developed with Marsha Berger, based on her work on adaptive refinement for conservation laws, particularly the Euler equations, [4], [1], [2]. Some of the issues involved in coupling these codes together with CLAWPACK, and generalizing them to allow nonconservative systems, can be found in the paper [3].

Please cite the relevant paper(s) in publications where CLAWPACK has been used.

1.3 Problem format

In one space dimension, the routine `claw1` can be used to solve a system of equations of the form

$$\kappa(x)q_t + f(q)_x = \psi(q, x, \kappa) \tag{1.1}$$

where $q = q(x,t) \in \mathbb{R}^m$. The standard case of a homogeneous conservation law has $\kappa \equiv 1$ and $\psi \equiv 0$,

$$q_t + f(q)_x = 0. \tag{1.2}$$

The flux function $f(q)$ can also depend explicitly on x and t as well as on q. Hyperbolic systems that are not in conservation form, *e.g.*,

$$q_t + A(q, x, t)q_x = 0, \tag{1.3}$$

can also be solved.

The basic requirement on the homogeneous system is that it be hyperbolic in the sense that a Riemann solver can be specified that, for any two states Q_{i-1} and Q_i, returns a set of M_w waves $\mathcal{W}_i^p$ and speeds λ_i^p satsifying

$$\sum_{p=1}^{M_w} \mathcal{W}_i^p = Q_i - Q_{i-1} \equiv \Delta Q_i.$$

The Riemann solver must also return a left-going fluctuation $\mathcal{A}^-\Delta Q_i$ and a right-going fluctuation $\mathcal{A}^+\Delta Q_i$. In the standard conservative case (1.2),

$$\mathcal{A}^-\Delta Q_i + \mathcal{A}^+\Delta Q_i = f(Q_i) - f(Q_{i-1}) \tag{1.4}$$

must be satisfied, and the fluctuations define a "flux-difference splitting". Typically

$$\mathcal{A}^-\Delta Q = \sum_p (\lambda_i^p)^- \mathcal{W}_i^p, \qquad \mathcal{A}^+\Delta Q = \sum_p (\lambda_i^p)^+ \mathcal{W}_i^p, \tag{1.5}$$

where $\lambda^- = \min(\lambda, 0)$ and $\lambda^+ = \max(\lambda, 0)$. In the nonconservative case (1.3), there is no "flux function" $f(q)$, and the constraint (1.4) need not be satisfied.

Only the fluctuations are used for the first-order Godunov method, which is implemented in the form

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\left[\mathcal{A}^+\Delta Q_i + \mathcal{A}^-\Delta Q_{i+1}\right], \tag{1.6}$$

assuming $\kappa \equiv 1$. The waves and speeds are used for high-resolution methods, which take the form

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\left[\mathcal{A}^+\Delta Q_i + \mathcal{A}^-\Delta Q_{i+1}\right] - \frac{\Delta t}{\Delta x}(\tilde{F}_{i+1} - \tilde{F}_i) \tag{1.7}$$

where

$$\tilde{F}_i = \frac{1}{2}\sum_{p=1}^{m} |\lambda^p| \left(1 - \frac{\Delta t}{\Delta x}|\lambda^p|\right) \tilde{\mathcal{W}}_i^p. \tag{1.8}$$

Here $\tilde{\mathcal{W}}_i^p$ represents a limited version of the wave $\mathcal{W}_i^p$, obtained by comparing $\mathcal{W}_i^p$ to $\mathcal{W}_{i-1}^p$ if $\lambda^p > 0$ or to $\mathcal{W}_{i+1}^p$ if $\lambda^p < 0$.

When a capacity function $\kappa(x)$ is present, the Godunov method becomes

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\kappa_i \Delta x}\left[\mathcal{A}^+\Delta Q_i + \mathcal{A}^-\Delta Q_{i+1}\right], \tag{1.9}$$

See [9] for discussion of this algorithm and its extension to the high-resolution method.

The Riemann solver must be supplied by the user in the form of a subroutine `rp1`. If applicable, a routine `src1` must also be supplied that solves the source term equation $q_t = \psi(q, \kappa)$ over a time step. A fractional step method is used to couple this with the homogeneous solution. Boundary conditions are imposed by setting values in ghost cells each time step. A few standard boundary conditions are implemented in a library routine, but this can be modified to impose other conditions.

In two space dimensions, the general form is

$$\kappa(x,y)q_t + A(q,x,y,t)q_x + B(q,x,y,t)q_y = \psi(q,\kappa), \tag{1.10}$$

where $q(x,y,t) \in \mathbb{R}^m$. The comments above for the one-dimensional case apply in general in two dimensions as well. The problem can be solved using dimensional splitting by setting `method(3) = -1` or `-2` (see Section 1.9). For nonnegative values of `method(3)` a multidimensional method is used and in general it is necessary to provide two Riemann solvers, `rpn2` and `rpt2`, one in the direction normal to the interface and the other in the transverse direction, see [9].

The equations (1.10) are solved on a uniform Cartesian grid, but more general curvilinear grids can be handled if there is a smooth coordinate transformation to a uniform grid. The capacity function $\kappa(x,y)$ is then used for the Jacobian of the grid mapping.

1.4 Obtaining CLAWPACK

The discussion here assumes you are using the Unix operating system. The Unix prompt is denoted by `unix>`.

1.4.1 Local users at `amath.washington.edu`

If you are running on the Applied Math computer system at the University of Washington, you can find everything mentioned below in the directory `~claw`. You will have to copy an application directory (*e.g.*, `claw/clawpack/1d/example1`) over to your account in order to run the code and produce output, but you can refer to the library routines from `claw/clawpack/1d/lib` without having to copy over everything.

1.4.2 From the web

The easiest way to download CLAWPACK is probably from the web, at
`http://www.amath.washington.edu/~claw/`
Go to "download software" and select the portion you wish to obtain. At a minimum, to get started with the one-dimensional routines you will need

```
claw/clawpack/1d
```

You might want to also download the 2d and 3d versions at this time, in which case you can select all of

```
        claw/clawpack
```

If you plan to use MATLAB to plot results, some useful scripts are in

```
        claw/matlab
```

Other plotting packages can also be used, but you will have to figure out how to properly read in the solution produced by CLAWPACK.

The basic CLAWPACK directories `1d`, `2d`, and `3d` each contain one or two examples in directories such as

```
        claw/1d/example1
```

which illustrate the basic use of CLAWPACK.

Various applications of CLAWPACK can be found in

```
        claw/applications.
```

See Section 1.12 for a partial list, or consult the web for the current list.

1.4.3 Anonymous ftp

If you have trouble obtaining these files using a web browser, they are also available by anonymous ftp from `amath.washington.edu`:

```
   unix> ftp amath.washington.edu
   [ log in as anonymous ]
   ftp> cd pub/claw
   ftp> get claw1d.tar.gz
   ftp> quit
```

1.5 Getting started

1.5.1 Creating the directories

The files you download will be gzipped tar files. If you download the 1d files, for example, you will obtain the file `claw1d.tar.gz` and you need to execute the commands

```
    unix> gunzip claw1d.tar
    unix> tar -xvf claw1d.tar
```

This will create a directory `claw` and the appropriate subdirectories, or add to `claw` if it already exists. Make sure you are in the directory where you want `claw` to appear. You should now have a directory named `<path>/claw` where `<path>` is the pathname to this directory which depends on where you put it and the local naming conventions on your file system.

1.5.2 Environment variables for the path

You should now set the environment variable `CLAW` in Unix so that the proper files can be found:

```
        unix> setenv CLAW <path>/claw
```

For example, if you are working on the AMath system, you can type

```
        unix> setenv CLAW /home/claw
```

(You might want to put this line in your `.cshrc` file so it will automatically be executed when you login or create a new window.) Now you can refer to `$CLAW/clawpack/1d`, for example, and reach the correct directory.

1.5.3 Compiling the code

Go to the directory `claw/clawpack/1d/example1`. There is a file in this directory named `compile`, which should be executable so that you can type

```
unix> compile
```

This should invoke `f77` to compile all the necessary files and create an executable called `xclaw`. To run the program type

```
unix> xclaw
```

and the program should run, producing output files that start with `fort.` In particular, `fort.q0000` contains the initial data and `fort.q0001` the solution at the first output time. The file `fort.info` has some information about the performance of CLAWPACK.

1.5.4 Makefiles

The `compile` file simply compiles all of the routines needed to run CLAWPACK on this example. This is simple but if you make one small change in one routine then everything has to be recompiled. Instead it is generally easier to use a `Makefile` which specifies what set of object files (ending with `.o`) are needed to make the executable, and which fortran files (ending with `.f`) are needed to make the object files. If a fortran file is changed then it is only necessary to recompile this one rather than everything. This is done simply by typing

```
unix> make
```

A complication arises since the `example1` directory only contains a few of the necessary fortran files, the ones specific to this particular problem. All the standard CLAWPACK files are in the directory `claw/clawpack/1d/lib`. You should **first** go into that directory and type `make` to create the object files for these library routines. This only needs to be done once if these files are never changed. Now go to the `example1` directory and also type `make`. Again an executable named `xclaw` should be created. See the comments at the start of the `Makefile` for some other options.

1.5.5 MATLAB graphics

If you wish to use MATLAB to view the results, you should download the directory `claw/matlab` and then set the environment variable

```
unix> setenv MATLABPATH ".:$CLAW/matlab"
```

before starting MATLAB, in order to add this directory to your MATLAB search path. This directory contains the plotting routines `plotclaw1.m` and `plotclaw2.m` for plotting results in 1 and 2 dimensions respectively.

With MATLAB running in the `example1` directory, type

```
Matlab> plotclaw1
```

to see the results of this computation. You should see a pulse advecting to the right with velocity 1, and wrapping around due to the periodic boundary conditions applied in this example.

1.6 Using CLAWPACK — A guide through `example1`

The program in `claw/clawpack/1d/example1` solves the advection equation

$$q_t + uq_x = 0$$

with constant velocity $u = 1$ and initial data consisting of a Gaussian hump

$$q(x, 0) = \exp(-\beta(x - 0.3)^2). \tag{1.11}$$

The parameters $u = 1$ and $\beta = 200$ are specified in the file `setprob.data`. These values are read in by the routine `setprob.f` described in Section 1.6.6.

1.6.1 The main program (`driver.f`)

The main program is located in the file `driver.f`. It simply allocates storage for the arrays needed in CLAWPACK and then calls `claw1ez`, described below. Several parameters are set and used to declare these arrays. The proper values for these parameters depend on the particular problem. They are:

`maxmx`: The maximum number of grid cells to be used. (The actual number `mx` is later read in from the input file `claw1ez.data` and must satisfy `mx` $\leq$ `maxmx`.)

`meqn`: The number of equations in the hyperbolic system, *e.g.*, `meqn = 1` for a scalar equation, `meqn = 2` for the acoustics equations in one dimension, etc.

`mwaves`: The number of waves produced in each Riemann solution. Often `mwaves = meqn` but not always.

`mbc`: The number of "ghost cells" used for implementing boundary conditions, as described in Section ??. Setting `mbc = 2` is sufficient unless changes have been made to the CLAWPACK software resulting in a larger stencil.

`mwork`: A work array of dimension `mwork` is used internally by CLAWPACK for various purposes. How much space is required depends on the other parameters:

```
mwork  ≥  (maxmx + 2*mbc) * (2 + 3*meqn + mwaves + meqn*mwaves)
```

If the value of `mwork` is set too small, CLAWPACK will halt with an error message telling how much space is required.

`maux`: The number of "auxiliary" variables needed for information specifying the problem, which is used in declaring the dimensions of the array `aux` (see below).

Three arrays are declared in `driver.f`:

`q(1-mbc:maxmx+mbc, meqn)`: This array holds the approximate solution Q_i^n (a vector with `meqn` components) at a given time t_n. The value of i ranges from 1 to `mx` where `mx <= maxmx` is set at run time from the input file. The additional "ghost cells" numbered `(1-mbc):0` and `(mx+1):(mx+mbc)` are used in setting boundary conditions.

`work(mwork)`: Used as work space.

`aux(1-mbc:maxmx+mbc, maux)`: `maux > 0`: This array holds auxiliary variables used to define a particular problem. For example, in a variable-coefficent advection problem the velocity at $x_{i-1/2}$ might be stored in `aux(i,1)`. See Section 1.7 for an example and more discussion.

`maux = 0`: There are no auxiliary variables and `aux` can simply be declared as a scalar or not declared at all since this array will not be referenced.

1.6.2 The initial conditions (`qinit.f`)

The subroutine `qinit.f` sets the initial data in the array `q`. For a system with `meqn` components, `q(i,m)` should be initialized to a cell average of the `m`'th component in the `i`'th grid cell. If the data is given by a smooth function then it may be simplest to simply evaluate this function at the center of the cell, which agrees with the cell average to $O((\Delta x)^2)$. The left edge of the cell is at `xlower + (i-1)*dx` and the right edge is at `xlower + i*dx`. It is only necessary to set values in cells `i = 1:mx`, not in the ghost cells. The values of `xlower`, `dx`, and `mx` are passed into `qinit.f`, having been set in `claw1ez`.

1.6.3 The `claw1ez` routine

The main program `driver.f` sets up array storage and then calls the subroutine `claw1ez`, which is located in `claw/clawpack/1d/lib`, along with other standard CLAWPACK subroutines described below. The `claw1ez` routine provides an easy way to use CLAWPACK which should suffice for many applications. It reads input data from a file `claw1ez.data`, which is assumed to be in a standard form described below. It also makes other assumptions about what the user is providing and what type of output is desired. After checking the inputs for consistency, `claw1ez` calls the CLAWPACK routine `claw1` repeatedly to produce the solution at each desired output time.

The `claw1` routine (located in `claw/clawpack/1d/lib/claw1.f`) is much more general and can be called directly by the user if more flexibility is needed. See the documentation in the source code for this routine.

1.6.4 Boundary conditions

Boundary conditions must be set in every time step and `claw1` calls a subroutine `bc1` in every step to accomplish this. The manner in which this is done is described in Section ??. For many problems the choice of boundary conditions provided in the default routine `claw/clawpack/1d/lib/bc1.f` will be sufficient. For other boundary conditions the user must provide an appropriate routine. This can be done by copying the `bc1.f` routine to the application directory and modifying it to insert the appropriate boundary conditions at the points indicated.

When using `claw1ez`, the `claw1ez.data` file contains parameters specifying what boundary condition is to be used at each boundary (see Section 1.6.7 where the `mthbc` array is described).

1.6.5 The Riemann solver

The file `claw/clawpack/1d/example1/rp1ad.f` contains the Riemann solver, a subroutine which should be named `rp1` if `claw1ez` is used. (More generally the name of the subroutine can be passed as an input to `claw1`). The Riemann solver is the crucial user-supplied routine which specifies the hyperbolic equation being solved. The input data consists of two arrays `ql` and `qr`. The value `ql(i,:)` is the value Q_i^L at the left edge of the i'th cell, while `qr(i,:)` is the value Q_i^R at the right edge of the i'th cell, as indicated in Figure 1.1. Normally `ql = qr` and both values agree with Q_i^n, the cell average. More flexibility is allowed because in some applications, or in adapting CLAWPACK to implement different algorithms, it is useful to allow different values at each edge. For example, we might want to define a piecewise linear function within the grid cell as illustrated in Figure 1.1 before solving the Riemann problems. Also, in the adaptive mesh refinement algorithms of AMRCLAW this feature is needed in applying corrections at the edges of refined patches (see [3]).

Note that the Riemann problem at the interface between cells $i-1$ and i has data

$$\begin{aligned}\text{left state: } Q_{i-1}^R &= \texttt{qr}(\texttt{i}-1,:)\\ \text{right state: } Q_i^L &= \texttt{ql}(\texttt{i},:).\end{aligned} \tag{1.12}$$

This notation is rather confusing since normally we use q_l to denote the left state and q_r to denote the right state in specifying Riemann data. The routine `rp1` must solve the Riemann problem for each value of `i`, and return the following:

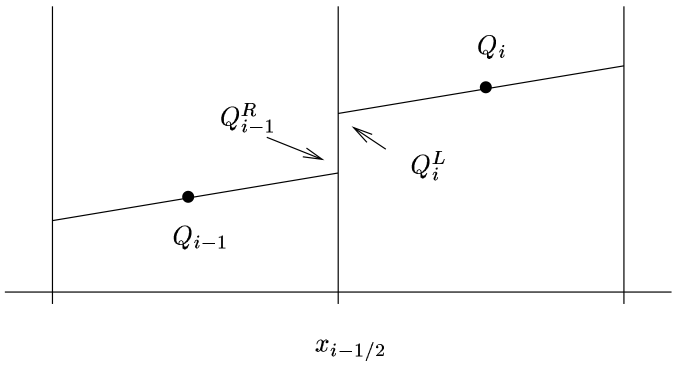

Figure 1.1: The states used in solving the Riemann problem at the interface $x_{i-1/2}$.

`amdq(i,1:meqn)` The vector $\mathcal{A}^-\Delta Q_i$ containing the left-going fluctuation.

`apdq(i,1:meqn)` The vector $\mathcal{A}^+\Delta Q_i$ containing the right-going fluctuation.

`wave(i,1:meqn,p)` The vector $\mathcal{W}_{i-1/2}^p$ representing the jump in q across the p'th wave in the Riemann solution at $x_{i-1/2}$, for `p = 1, 2, ..., mwaves`. (In the code `mw` is used in place of p.)

`s(i,p)` The wave speed $\lambda_{i-1/2}^p$ for the p'th wave.

For Godunov's method, only the fluctuations `amdq` and `apdq` are actually used, and the update formula (1.6) is employed. The waves and speeds are only used for high-resolution correction terms (1.8).

For the advection equation, the Riemann solver in `example1` returns

$$\begin{aligned}
&\texttt{wave(i,1,1)} = \texttt{ql(i,1)} - \texttt{qr(i-1,1)}\\
&\texttt{s(i,1)} = u\\
&\texttt{amdq(i,1)} = \min(u,0)*\texttt{wave(i,1,1)}\\
&\texttt{apdq(i,1)} = \max(u,0)*\texttt{wave(i,1,1)}
\end{aligned}$$

Sample Riemann solvers for a variety of other applications can be found in `claw/book` and `claw/applications`. Often these can be used directly rather than writing a new Riemann solver.

1.6.6 Other user-supplied routines and files

`setprob.f` The `claw1ez` routine always calls `setprob` at the beginning of execution. The user can provide a subroutine which sets any problem-specific parameters or does other initialization. For the advection problem solved in `example1`, this is used to set the advection velocity u. This value is stored in a common block so that it can be passed into the Riemann solver, where it is required. The parameter `beta` is also set and passed into the routine `qinit.f` for using in setting the initial data according to (1.11).

When `claw1ez` is used, a `setprob` subroutine must always be provided. If there is nothing to be done, the default subroutine `claw/clawpack/1d/lib/setprob.f` can be used, which does nothing but `return`.

`setaux.f` The `claw1ez` routine calls a subroutine `setaux` before the first call to `claw1`. This routine should set the array `aux` to contain any necessary data used in specifying the problem. For the example in `example1` no `aux` array is used (`maux = 0` in `driver.f`) and the default subroutine `claw/clawpack/1d/lib/setaux.f` is specified in the `Makefile`. This routine does nothing but `return`.

`b4step1.f` The routine `claw1` calls `b4step1` just before every call to `step1`. This routine can be supplied by the user to perform any actions that are desired at the start of every time step. For example, if the `aux` array contains variable coefficients which are time dependent, they will have to be reset every timestep and this can be done in the `b4step1` routine. In some applications it may be desirable to print out the value of the solution at some point every time step (*e.g.*, at one of the boundaries) and this routine could also be used for this purpose. **Note:** If a time step must be retaken with a smaller Δt when using variable time steps (`method(1)=1`) then the routine `b4step1` will be called again at the same time before this new step. The default subroutine `claw/clawpack/1d/lib/b4step1.f` does nothing but `return`.

1.6.7 The input file `claw1ez.data`

The `claw1ez` routine reads data from a file named `claw1ez.data`. Figure 1.2 shows the file from `example1`. Typically one value is read from each line of this file. Any text following this value on each line is not read and is there simply as documentation. The values read are:

`mx`: The number of grid cells for this computation. (Must have `mx < maxmx`, where `maxmx` is set in `driver.f`.)

`nout`: Number of output times at which the solution should be written out.

`outstyle`: There are three possible ways to specify the output times. This parameter selects the desired manner to specify the times, and affects what is required next.

`outstyle = 1`: The next line contains a single value `tfinal`. The computation should proceed to this time and the `nout` outputs will be at times `(tfinal - t0)/nout`, where the initial time `t0` is set below.

`outstyle = 2`: The next line(s) contain a list of `nout` times at which the outputs are desired. The computation will end when the last of these times is reached.

`outstyle = 3`: The next line contains two values

```
nstepout, nsteps
```

A total of `nsteps` time steps will be taken, with output after every `nstepout` time steps. This is most useful if you want to insure that time steps of maximum length are always taken with a desired Courant number. With the other output options the time steps are adjusted to hit the desired times exactly. With this option, the value of `nout` is ignored.

`dtv(1)`: The initial value of Δt used in the first time step. If `method(1) = 0`, then fixed size time steps are used and this is the value of Δt in all steps. In this case Δt must divide the time increment between all requested outputs an integer number of times.

`dtv(2)`: The maximum time step Δt to be allowed in any step (in the case where `method(1) = 1` and variable Δt is used). Variable time steps are normally chosen based on the Courant number, and this value can simply be set to some very large value so that it has no effect. For some problems, however, it may be necessary to restrict the time step to a smaller value based on other considerations, *e.g.*, the behavior of source terms in the equations.

`cflv(1)`: The maximum Courant number to be allowed. The Courant number is calculated after all the Riemann problems have been solved by determining the maximum wave speed seen. If the Courant number is no larger than `cflv(1)` then this step is accepted. If the Courant number is larger, then:

`method(1)=0`: (fixed time steps), the calculation aborts.

`method(1)=1`: (variable time steps), the step is rejected and a smaller time step is taken.

Usually `cflv(1) = 1` can be used.

`cflv(2)`: The desired Courant number for this computation. Used only if `method(1)=1` (variable time steps). In each time step, the next time increment Δt is based on the maximum wave speed found in solving all Riemann problems in the *previous* time step. If the wave speeds do not change very much then this will lead to roughly the desired Courant number. It's typically best to take `cflv(2)` to be slightly smaller than `cflv(1)`, say `cflv(2) = 0.9`.

`nv(1)`: The maximum number of time steps allowed in any single call to `claw1`. This is provided as a precaution to avoid too-lengthy runs.

`method(1)`: Tells whether fixed or variable size time steps are to be used.

`method(1) = 0`: A fixed time step of size `dtv(1)` will be used in all steps.

`method(1) = 1`: CLAWPACK will automatically select the time step as described above based on the desired Courant number.

`method(2)`: The order of the method.

`method(2) = 1`: The first-order Godunov's method is used.

`method(2) = 2`: High-resolution correction terms are also used.

`method(3)`: This parameter is not used in one space dimension. In two and three dimensions it is used to further specify what correction terms are applied, and for dimensional splitting.

`method(4)`: This controls the amount of output printed on the screen as CLAWPACK progresses.

`method(4) = 0`: No output.

`method(4) = 1`: Every time step the value Δt and Courant number are reported.

`method(5)`: Tells whether there is a source term in the equation. If so, then a fractional step method is used. Time steps on the homogeneous hyperbolic equation are alternated with time steps on the source term. The solution operator for the source terms must be provided by the user in the routine `src1.f`.

`method(5) = 0`: There is no source term. In this case the default routine `claw/clawpack/1d/lib/src1.f` can be used which does nothing, and in fact this routine will never be called.

`method(5) = 1`: A source term is specified in `src1.f` and the "first order (Godunov)" fractional step method should be used.

`method(5) = 2`: A source term is specified in `src1.f` and a Strang splitting is used.

The Godunov splitting is generally recommended rather than the Strang splitting.

`method(6)`: Tells whether there is a "capacity function" in the equation.

`method(6) = 0`: No capacity function, $\kappa \equiv 1$ in (1.1).

`method(6) = mcapa > 0`: There is a capacity function and the value of κ in the i'th cell is given by `aux(i,mcapa)`, *i.e.*, the `mcapa` component of the `aux` array is used to store this function. In this case "capacity-form differencing" is used, as described in [9].

`method(7)`: Tells whether there are any auxiliary variables stored in an `aux` array.

`method(7) = 0`: No auxiliary variables. In this case the array `aux` is not referenced and can be a dummy variable.

`method(7) = maux > 0`: There is an `aux` array with `maux` components. In this case the array must be properly declared in `driver.f`.

Note that we must always have `maux` $\geq$ `mcapa`. The value of `method(7)` specified here must agree with the value of `maux` set in `driver.f`.

`meqn`: The number of equations in the hyperbolic system. This is also set in `driver.f` and the two should agree.

`mwaves`: The number of waves in each Riemann solution. This is typically equal to `meqn` but need not be. This is also set in `driver.f` and the two should agree.

`mthlim(1:mwaves)`: The limiter to be applied in each wave family. Several different limiters are provided in CLAWPACK

`mthlim(mw) = 0`: No limiter ("Lax-Wendroff")

`mthlim(mw) = 1`: Minmod

`mthlim(mw) = 2`: Superbee

`mthlim(mw) = 3`: van Leer

`mthlim(mw) = 4`: MC (monotonized centered)

Other limiters are easily added by modifying the routine `lib/philim.f`.

`t0`: The initial time.

`xlower`: The left edge of the computational domain.

`xupper`: The right edge of the computational domain.

`mbc`: The number of ghost cells used for setting boundary conditions. Usually `mbc = 2` is used.

`mthbc(1)`: The type of boundary condition to be imposed at the left boundary. The following values are recognized:

`mthbc(1) = 0`: The user will specify a boundary condition. In this case you must copy the file `claw/clawpack/1d/lib/bc1.f` to your application directory and modify it to insert the proper boundary conditions in the location indicated.

`mthbc(1) = 1`: Zero-order extrapolation.

`mthbc(1) = 2`: Periodic boundary conditions. In this case you must also set `mthbc(2) = 2`.

`mthbc(1) = 3`: Solid wall boundary conditions. This set of boundary conditions only makes sense for certain systems of equations, such as acoustics, Euler equations, shallow water.

`mthbc(2)`: The type of boundary condition to be imposed at the right boundary. The same values are recognized as described above.

1.7 Auxiliary arrays and `setaux.f`

The array `q(i,1:meqn)` contains the finite-volume solution in the i'th grid cell. Often other arrays defined over the grid are required to specify the problem in the first place. For example, in a variable-coefficient advection problem

$$q_t + u(x)q_x = 0$$

the Riemann solution at any cell interface $x_{i-1/2}$ depends on the velocities u_{i-1} and u_i. The `aux` array can be used to store these values and pass them into the Riemann solver. In the advection example we need only one auxiliary variable so `maux = 1` and we store the velocity u_i in `aux(i,1)`.

```
50                  mx          = cells in x direction

11                  nout        = number of output times to print results
1                   outstyle    = style of specifying output times
2.2d0               tfinal      = final time

0.1d0               dtv(1)      = initial dt (used in all steps if method(1)=0)
1.0d99              dtv(2)      = max allowable dt
1.0d0               cflv(1)     = max allowable Courant number
0.9d0               cflv(2)     = desired Courant number
500                 nv(1)       = max number of time steps per call to claw1

1                   method(1)   = 1 for variable dt
2                   method(2)   = order
0                   method(3)   = not used in one dimension
1                   method(4)   = verbosity of output
0                   method(5)   = source term splitting
0                   method(6)   = mcapa
0                   method(7)   = maux (should agree with parameter in driver)

1                   meqn        = number of equations in hyperbolic system
1                   mwaves      = number of waves in each Riemann solution
3                   mthlim(mw)  = limiter for each wave  (mw=1,mwaves)

0.d0                t0          = initial time
0.0d0               xlower      = left edge of computational domain
1.0d0               xupper      = right edge of computational domain

2                   mbc         = number of ghost cells at each boundary
2                   mthbc(1)    = type of boundary conditions at left
2                   mthbc(2)    = type of boundary conditions at right
```

Figure 1.2: A typical `claw1ez.data` file, from `claw/clawpack/1d/example1` for advection.

Of course one could hard-wire the specific function $u(x)$ into the Riemann solver or pass it in using a common block, but the use of the auxiliary arrays gives a uniform treatment of such data arrays. This is useful in particular when adaptive mesh refinement is applied, in which case there are many different `q` grids covering different portions of the computational domain and it is very convenient to have an associated `aux` array corresponding to each.

The `claw1ez` routine always calls a subroutine `setaux` before beginning the computation. This routine, normally stored in `setaux.f`, should set the values of all auxiliary arrays. If `maux = 0` then the default routine `claw/clawpack/1d/lib/setaux.f` can be used which does nothing.

1.8 An acoustics example

The directory `claw/clawpack/1d/example2` contains a sample code for the constant-coefficient acoustics equations

$$\begin{aligned} p_t + K u_x &= 0 \\ \rho u_t + p_x &= 0. \end{aligned} \tag{1.13}$$

The parameters ρ and K are the density and bulk modulus of the material, respectively. The variables p and u are the pressure perturbation and velocity in an acoustic wave. The value of the density and bulk modulus are set in `setprob.f` (where they are read in from a data file `setprob.data`). In this routine the sound speed c and impedance $Z = \rho c$ are also computed and passed to the Riemann solver in a common block, since these are used in the eigenvalues and eigenvectors:

$$\begin{aligned} &\lambda^1 = -c, &&\lambda^2 = c \\ &r^1 = \begin{bmatrix} -Z \\ 1 \end{bmatrix}, &&r^2 = \begin{bmatrix} Z \\ 1 \end{bmatrix}. \end{aligned} \tag{1.14}$$

To plot the results in MATLAB, again use `plotclaw1`. Note that when prompted "Which component of q?", you can respond with either 1 or 2 to view the pressure or velocity respectively, or with the entire range 1:2 to view both together.

1.9 Two space dimensions (`claw2ez.f`)

In two space dimensions the programs in `claw/clawpack/2d` are organized in roughly the same way as in `claw/clawpack/1d`. An example can be found in `claw/clawpack/2d/example1`, which uses the routine `claw/clawpack/2d/lib/claw2ez.f` to solve a nonlinear scalar equation in two dimensions. The data file `claw2ez.data` is very similar to the one-dimensional data files with a few additional parameters:

`my`: The number of grid cells in the y-direction for this computation. (Must have `my < maxmy`, where `maxmy` is set in `driver.f`.)

`method(3)`: If this parameter is negative then dimensional splitting is used.

`method(3) = -1`: Dimensional splitting with the Godunov splitting. In each step an x-sweep is applied and then a y-sweep.

`method(3) = -2`: Dimensional splitting with the Strang splitting. In each step an x-sweep is applied with time step $\Delta t/2$, then a y-sweep with time step Δt, and finally another x-sweep is applied with time step $\Delta t/2$. In most cases this is not recommended.

If `method(3)` is nonnegative, then the unsplit algorithm described in [9] is used. In this case this parameter indicates what type of transverse propagation is applied:

`method(3) = 0`: No transverse propagation. In this case `rpt2` is not called. This method is generally stable only for Courant numbers less than $1/2$.

`method(3) = 1`: Transverse propagation of increment waves only.

`method(3) = 2`: Transverse propagation of correction waves also.

`ylower`: The bottom edge of the computational domain.

`yupper`: The top edge of the computational domain.

`mthbc(3)`: The type of boundary condition to be imposed at the bottom boundary. The same values are recognized as described above for `mthbc(1)`.

`mthbc(4)`: The type of boundary condition to be imposed at the top boundary. The same values are recognized as described above for `mthbc(1)`.

1.9.1 Riemann solvers

Two Riemann solvers must now be provided, as described in [9]. The transverse solver `rpt2` is called only if `method(3) > 0` and a dummy routine could be provided otherwise.

`rpn2`: Solves the Riemann problem normal to a cell interface, analogous to `rp1`. The CLAWPACK routine `flux2` calls this routine repeatedly with a single slice of data along a row or column of grid cells in the two-dimensional domain. The parameter `ixy` indicates whether the slice is in the x-direction or in the y-direction:

`ixy = 1`: Slice is in the x-direction and this routine should return the solution to the Riemann problem $q_t + A(q, x, y, t)q_x = 0$ from (1.10).

`ixy = 2`: Slice is in the y-direction and this routine should return the solution to the Riemann problem $q_t + B(q, x, y, t)q_y = 0$ from (1.10).

`rpt2`: Solves the Riemann problem in the transverse direction. The input is an array `asdq` along a slice of the grid, where `asdq` represents one of the fluctuations `amdq` or `apdq` which came out of the normal Riemann solver `rpn2`. This fluctuation must be split into an up-going portion `bpasdq` and a down-going portion `bmasdq`. (Here "up" refers to larger values of i or j on the grid, "down" to smaller values.)

Again the parameter `ixy` indicates whether the slice is in x or y. In addition, a parameter `imp` indicates whether `asdq` represents `amdq` or `apdq`:

`imp = 1`: `asdq` represents `amdq`, the fluctuation which is propagating into the cell to the left of the interface.

`imp = 2`: `asdq` represents `apdq`, the fluctuation which is propagating into the cell to the right of the interface.

This information may be needed for problems with variable coefficients depending on x and y. Such coefficients might be stored in the `aux` arrays. The slice of the `aux` array corresponding to the slice of the grid on which we are currently working is passed into `rpt2` in the array `aux2`. In addition the slice from the row "below" is passed in `aux1`, and the slice from the row "above" is passed in `aux3`. Values from the adjacent rows may be needed to compute the portion of `asdq` which propagates up or down into the neighboring cells.

1.10 Three space dimensions (`claw3ez.f`)

The three-dimensional CLAWPACK routines are found in `claw/clawpack/3d/lib`. This is a fairly direct extension of the two-dimensional routines with obvious extensions to the third dimension and corresponding additional parameters required in the input file. The main changes from two dimensions are the following:

`method(3)`: Again setting this to -1 or -2 invokes dimensional splitting (Godunov or Strang splitting respectively). Setting `method(3) = 0` gives the unsplit method but with no transverse splitting. If `method(3) > 0` then the routine `rpt3` is used to do transverse splitting. In three dimensions one might want to apply "double-transverse" corrections as well as transverse corrections, as described in [7]. For example, waves arising from solving the Riemann problem in x may be split into transverse waves in y to update the cells above and below, but these waves may be further split into waves in the z-direction to give the proper corner coupling. If `method(3) > 0` then it should take one of the following values:

- `method(3) = 10`: Transverse propagation of the increment wave as in 2D. This method is unconditionally unstable!
- `method(3) = 11`: Corner transport upwind of the increment wave. This method is unconditionally unstable!
- `method(3) = 20`: Both the increment wave and the correction wave propagate as in the 2D case. Only to be used with method(2) = 2.
- `method(3) = 11`: Corner transport upwind of the increment wave, and the correction wave propagates as in 2D. Only to be used with method(2) = 2.
- `method(3) = 11`: 3D propagation of both the increment wave and the correction wave. Only to be used with method(2) = 2.

`rpn3`: Solves the Riemann problem normal to a cell interface, analogous to `rpn2`. The parameter `ixyz = 1,2,3` indicates whether the slice is in the x- y- or z-direction.

`rpt3`: Solves the Riemann problem in the transverse directions, but now for each coordinate direction there are two orthogonal axes. The parameter `ixyz` indicates what direction the slice of data lies in, as in `rpn3`. The parameter `icoor` indicates which of the transverse directions is to be used for the transverse splitting:

- `ixyz=1, icoor=2`: Data is in x, split in y.
- `ixyz=1, icoor=3`: Data is in x, split in z.
- `ixyz=2, icoor=2`: Data is in y, split in z.
- `ixyz=2, icoor=3`: Data is in y, split in x.
- `ixyz=3, icoor=2`: Data is in z, split in x.
- `ixyz=3, icoor=3`: Data is in z, split in y.

1.11 Plotting results with MATLAB

In one dimension use `plotclaw1`. In two dimensions use `plotclaw2`. These can be found in `claw/matlab`. Good three-dimensional graphics are not yet available.

Various parameters used in these routines are initialized in the file `setplot1.m` (or `setplot2.m` in two dimensions). If no such file exists in the current directory, then the default file `claw/matlab/setplot1.m` is used. Values can be changed by modifying this file, or by typing `k` at the `plotclaw` prompt and then entering a new value.

Several other characters can be typed at the `plotclaw` prompt:

k Keyboard input. Type any commands and then "return".

r Redraw current frame.

j Jump to a particular frame.

i Print info about parameters and solution.

q Quit.

See Section 2.3 for some additional information on `plotclaw2`.

1.12 Other applications in `claw/applications`

The applications directory is organized by particular application. See `claw/applications/index` for a list of applications currently available. These subdirectories are typically further subdivided into `1d`, `2d`, and sometimes `3d`, and then within each of these is a subdirectory `rp` which contains the Riemann solver(s) and other subdirectories for solving particular problems or classes of problems.

Chapter 2

Adaptive Mesh Refinement and AMRCLAW

CLAWPACK 4.0 contains the adaptive mesh refinement routines of AMRCLAW developed with Marsha Berger, based on her codes for the Euler equations. These have been extended to handle general systems of equations based on Riemann solvers in exactly the same form as required by the basic CLAWPACK routines. Other user-supplied routines such as `qinit.f`, `setprob.f`, `setaux.f` and source term routines also have exactly the same form as in CLAWPACK.

The boundary condition routine which sets ghost cell values is slightly more complicated for AMRCLAW, as described in Section 2.2, but the standard boundary conditions handled automatically in CLAWPACK (extrapolation, periodic, solid walls) are also implemented in AMRCLAW in such a way that the user need only specify appropriate values for the `mthbc` array in the data file, just as in CLAWPACK (see Section 2.1).

As a result, it should be quite easy to convert a running CLAWPACK code to AMRCLAW and take advantage of adaptive mesh refinement. In simple cases only the input data file and the `Makefile` need to be changed. Currently AMRCLAW is available only in two space dimensions. A three-dimensional version is being developed.

The basic two-dimensional routines can be found in `claw/amrclaw/2d/lib`. You must first do a `make` in this directory in order to create `.o` files for each library routine. For an example of the use of AMRCLAW see the directory `claw/clawpack/2d/example1/amr`. This solves the same problem as in `claw/clawpack/2d/example1` but with adaptive refinement. To use this code you will need to do a `make` first in `claw/amrclaw/2d/lib` and in `claw/clawpack/2d/example1` and then in this directory. This produces an executable `xamr`. Results can again be viewed in MATLAB using the `plotclaw2` script (see Section 2.3). Note that there are no user-supplied fortran routines in these directories; the `Makefiles` refer to the routines in the directory above.

2.1 The input file `amr2ez.data`

The input data file is different and is now called `amr2ez.data`. This has essentially the same form as the `claw2ez.data` file used by `claw2ez`, but for the adaptive routines some additional parameters must be specified. An example file is shown in Figure 2.1 and continued in Figure 2.2. The parameters `mxnest` and `inrat` and all those from `iousr` onwards are required only in the `amr2ez.data` file. Deleting these lines would produce the corresponding `claw2ez.data` file.

The new parameters needed by AMRCLAW are as follows:

`mxnest`: Maximum number of levels of grid refinement. `mxnest = 1` means a single uniform grid will be used. This should give identical answers as the non-adaptive CLAWPACK routines on the same grid and checking that this works is a good first step in converting a code to AMRCLAW.

`inrat(1:max(mxnest-1,1))`: Refinement ratios for each level. Grids at Level 2 will be finer than the Level 1 grid by a factor of `inrat(1)` in both x and y. In general, grids at Level `L` will be finer than grids at level `L-1` by a factor `inrat(L-1)`. Only `mxnest-1` components of `inrat` are actually needed, but at least one value is always read so that this line can remain in the input file even if `mxnest` is set to 1. Each refinement ratio must be an even integer. Values 2 or 4 are typically used.

`auxtype(1:maux)`: If `maux > 0` then for each component of the auxiliary array, a type must be specified from the following list, depending on what the corresponding component of `aux` represents:

`"leftedge"`	a value associated with the left edge of the cell.
`"bottomedge"`	a value associated with the bottom edge of the cell.
`"center"`	a cell-centered value.
`"capacity"`	a cell-centered capacity function.

At most one component may have the type `"capacity"` and the value of `mcapa` should be set in a consistent manner. This component is used as a capacity function in capacity-form differencing.

The `auxtype` array is required for adaptive refinement because auxiliary arrays must be handled slightly differently at refinement boundaries depending on how these values are used.

`iousr`: A checkpoint file is dumped every `iousr` time steps on the coarse grid. These are binary files with names of the form `fort.chkSNUM` where `SNUM` is the step number. The solution and grid structure is output in a form that can be used to later restart the calculation from this point. This is useful when doing long runs in case the computer goes down or the algorithm fails at some point in the calculation. It is also useful if you want to go to some large time and then start doing frequent outputs in order to examine the time-evolution of the solution more carefully.

In addition to creating a checkpoint file every `iousr` time steps, a final checkpoint file is created at the end of the computation. This can be used to restart the calculation from the final time if you wish to evolve it further.

`restart`: If `restart = T` then a restart is performed. Information read in from the file `restart.data` is used to resume a previous calculation. This file should be copied from a checkpoint file created in the previous calculation.

When a restart is performed, other parameters in this `amr2ez.data` file should be consistent with values used in the previous calculation, with some exceptions:

- The number of time steps requested `nstop` or the output times requested `tout(1:nout)` now refer to the new calculation.
- The maximum number of refinement levels `mxnest` can be larger than the number of levels previous used, but not smaller.
- Parameters specifying the method to use, `method(1:7)`, `mthlim`, `mthbc`, can be changed.

`tol`: Tolerance used in flagging grid cells which need to be refined. An estimate of the truncation error is computed and compared with this value, see Section 2.5. Smaller values will lead to more refinement.

`tolsp`: Another tolerance used in flagging grid cells which need to be refined. This is used in checking the magnitude of the spatial gradient, as computed by central differences.

`kcheck`: Error estimation and regridding is performed every `kcheck` time steps.

`ibuff`: Size of the buffer zone around flagged cells. Certain cells are flagged for refinement and then clustered (see Section 2.5) into finer grids. In addition to the cells flagged by the error estimation, all cells within `ibuff` cells of these are also flagged. This insures that structures in the solution which require refinement will remain in the refined region for at least `ibuff` time steps, since the Courant number must be no greater than 1. The value of `ibuff` should generally be consistent with the value of `kcheck`, with `ibuff` $\geq$ `kcheck` if the Courant number is close to 1.

`cutoff`: Parameter used in the clustering algorithm (see Section 2.5). Typically 0.7 is a good value.

`PRINT option`: Logical variable. If `T`, the solution values on all grids are output in the file `fort.amr` along with other information about the time stepping. Usually not used except on very coarse grids for debugging purposes.

`NCAR graphics`: Logical variable. If `T`, the solution is output in `fort.ncar` in a form suitable for NCAR graphics.

`Matlab graphics`: Logical variable. If `T`, the solution is output in the form suitable for viewing with `plotclaw2` in MATLAB.

`Xprint`: A number of other values can be set to `T` if you desire more output to be sent to `fort.amr` describing each grid, indicating which points were flagged for refinement, etc. Used primarily for debugging. An exception is `tprint` which is useful in general to keep track of how far along the code has progressed.

2.2 Boundary conditions

The boundary condition routine is somewhat more complicated for the adaptive code, since the edge of a grid may not be at a physical boundary. A grid might be adjacent to other grids at the same refinement level or to coarser grids. In either case the AMR routines automatically provide appropriate ghost cell values. The routine `claw/amrclaw/2d/lib/bc2amr.f` sets boundary conditions at the physical boundaries. It recognizes the same set of `mthbc` values as used in `claw2ez`, so that if these standard boundary conditions (extrapolation, periodic, or solid walls) are desired the user need not worry about this routine. To implement other boundary conditions, the `bc2amr.f` file can be copied to the user directory and modified as described in the documentation at the beginning of this routine and following the examples of these standard boundary conditions.

2.3 Plotting results with MATLAB

The MATLAB routine `plotclaw2` can be used for viewing the AMRCLAW output. Note that information about Frame 1, for example, is stored in `fort.t0001` and includes the value `ngrids` which tells how many grids exist at this time. The file `fort.q0001` contains the solution on each of these grids. The data for each grid is preceeded by information about this grid: what level it is at and where the lower left corner is located.

For plotting results from AMRCLAW the following parameters are useful. These are initialized in the `setplot2.m` file in the current directory, or if there isn't one then by the default file `claw/matlab/setplot2.m`.

`PlotData(L)`: Data at Level `L` is plotted only if `PlotData(L) > 0`. If you want to quickly step through many frames looking only at a course representation of the solution, this can be used to suppress time-consuming plotting of the finer grids.

`PlotGrid(L)`: For `pcolor` plots the grid lines will be plotted on grids at Levels `L` for which `PlotGrid(L) > 0`. You may want to plot the grid lines on coarser grids but suppress them on finer grids where they would obscure the data.

`PlotGridEdge(L)`: Even if grid lines are not plotted, a box showing the location of each grid will be plotted for grids at Levels `L` for which `PlotGridEdge(L) > 0`.

2.4 The adaptive algorithm

This is a very brief description of the basic steps in adaptive time stepping. First suppose there are only two grid levels. The algorithm proceeds as follows:

1. All grids at Level 1 (the coarsest level) are advanced by the coarse time step. Often there is only one grid at this level but since there is a limit on the maximum size of each grid, the domain may be automatically split into more than one coarse grid. Before advancing at this level the `bc2amr` routine is called to set ghost cells where needed.

2. All grids at Level 2 are advanced by `inrat(1)` time steps which are each smaller than the coarse time step by a factor `inrat(1)`. Time steps are refined in the same way as the grid spacing so that the Courant number is roughly the same at all levels.

   Before each time step, ghost cell values must be set. In general there are three types of ghost cells:

   (a) Those which lie within adjacent grids at Level 2, and the values are copied directly from the adjacent grid,

   (b) Those which lie within the physical domain but at a point where there is only a Level 1 grid. At these points interpolation is used to set the values based on the coarser grid. Since we have already advanced the coarse grid in time, we can use space-time interpolation to set an appropriate value. This is needed since the ghost cell may be at an intermediate time between coarse time steps as well as at an intermediate spatial point relative to the coarse grid.

   (c) Those which lie outside the physical domain. For these cells `bc2amr` must set the appropriate value based on the physical boundary conditions.

   An exception to Type (c) is when periodic boundary conditions are used. In this case the ghost cell is of Type (a) or (b) depending on the grid structure at the opposite edge of the domain. Ghost cells of Type (a) and (b) are handled automatically by AMRCLAW and the `bc2amr` routine must only check whether a ghost cell lies outside the physical domain and handle this case properly.

3. Once all grids at Level 2 have been advanced to the same time as the Level 1 grids, the values on the two sets of grids must be made consistent. For coarse grid cells which are covered by a fine grid, the fine grid presumably contains more accurate information and so the coarse grid value is replaced by the average of the fine grid values over all fine cells covering this coarse cell.

4. When a conservation law is being solved, we must insure that conservation is maintained. This requires some modifications at the edges of the fine grids since different fluxes were used on the fine grid than on the adjacent coarse grid. This is described in [3].

5. Every `kcheck` time steps on each level, error estimation and regridding is performed. This is done as described in Section 2.5.

If there are more than 2 levels, then this same algorithm is applied recursively at each of the finer levels. For every time step on Level 2, we take `inrat(2)` time steps on all Level 3 grids. Every `kcheck` time steps on Level 2, new Level 3 grids are changed, etc.

2.5 Error estimation and regridding

Every `kcheck` time steps on each level, the error is estimated at all cells on grids at this level. Cells where the error is above some cutoff are flagged for refinement. The norm used to measure the error can be adjusted, see below. The cells which have been flagged are then clustered into rectangular regions to form grids at the next level. The clustering is done in light of the tradeoffs between a few large grids (which usually means refinement of many additional cells which were not flagged) or many small grids (which typically results in fewer fine grid cells but more grids and hence more overhead and less efficient looping over shorter rows of cells). The parameter `cutoff` in `amr2ez.data` is used to control this tradeoff. At least this fraction of the fine grid cells should result from coarse cells that were flagged as needing refinement. `cutoff = 0.7` is usually reasonable.

Cells are flagged for refinement in `errf1` by one of two possible mechanisms.

1. The spatial gradient of the solution in cell (i, j) is estimated by simply computing the values `q(i+1,j,m)-q(i-1,j,m)` and `q(i,j+1,m)-q(i,j-1,m)` and maximixing over all components `m`. A cell is flagged if this is greater than the parameter `tolsp` in `amr2ez.data`. This is done in subroutine `errsp` where this norm could be adjusted (for example to only look at one particular component of `q` or use a different norm).

2. An estimate of the error which would be incurred on the present grid is obtained by doing two computations and comparing the errors:

   - The equations are advanced by two time steps on the current grid with the current Δt.
   - The equations are advanced by one time step on a grid that is twice as coarse (half as many points in each direction) with time step $2\Delta t$.

   Richardson extrapolation is then performed on these two solutions to obtain an estimate of the error on the present grid at time $2\Delta t$. Any cell where this estimate is above the value of `tol` specified in `amr2ez.data` is flagged for refinement. This procedure is performed in `errest` and the Richardson extrapolation is done in `errf1`. Coarsening by a factor of 2 is done rather than refining since this is cheaper to perform. (For this reason the initial grid must have an even number of cells in each direction.)

In general the Richardson extrapolation procedure should give a better indication of which cells need refinement, but can fail in some cases and so the simple spatial gradient estimate is also used.

In the procedure `errf1` it is indicated how to allow refinement at each level only in some regions of the domain and not elsewhere. This is useful if you wish to zoom in on some structure in a known location but don't want the same level of refinement elsewhere. Points are flagged only if one of the errors is greater than the corresponding tolerance and also `allowed(x,y,level)` has the value `.true.`.

2.6 Comments and warnings

For many problems the adaptive code should work immediately when a CLAWPACK code converted. However, there are several subtleties of the adaptive refinement that can lead to problems. Here are some comments on things to watch out for.

- **Auxiliary arrays.** Each time new grids are generated the routine `setaux` is called to set up the corresponding auxiliary array (if `maux > 0`). In CLAWPACK, `setaux` is called only once at the initial time, but in AMRCLAW it is called for every new grid at each regridding time, so it should be written in a manner that works in this more general context.

  If any components of the auxiliary arrays are reset at each time in the subroutine `b4step2`, this will happen automatically on the new grids before the first call to `step2`. In this case the `setaux` routine need only set the time-independent values in `aux` which are not set by `b4step2`.

- **Error estimation time steps.** In the error estimation process time steps are taken to estimate the error which are not part of the main calculation. The routine `b4step2` and the source term routine `src2` are called in this process as in any other time step. If the `b4step2` routine is used to adjust values of the solution or do other operations which should only be done once at each distinct time, this could be a problem.

  In the error estimation time steps a fixed time step is used, `method(1) = 0`, and it is not adjusted based on the Courant number.

```
 50                mx          = cells in x direction
 50                my          = cells in y direction
  3                mxnest      = max. number of allowable grid levels
 2 2 2             inrat       = refinement ratios (1..mxnest)

0                  nout        = number of output times to print results
3                  outstyle    = style of specifying output times
15,150             iout,nstop  = output interval, total number of steps

0.01d0             dtv(1)      = initial dt (used in all steps if method(1)=0)
1.0d99             dtv(2)      = max allowable dt
1.0d0              cflv(1)     = max allowable Courant number
0.9d0              cflv(2)     = desired Courant number
500                nv(1)       = max number of time steps per call to claw2

1                  method(1)   = 1 for variable dt,   = 0 for fixed dt
2                  method(2)   = order
2                  method(3)   = transverse order
1                  method(4)   = verbosity of output
0                  method(5)   = source term splitting
4                  method(6)   = mcapa
5                  method(7)   = maux (should agree with parameter in driver)
 "center"          auxtype(1)
 "leftedge"        auxtype(2)
 "center"          auxtype(3)
 "capacity"        auxtype(4)
 "bottomedge"      auxtype(5)


1                  meqn        = number of equations in hyperbolic system
1                  mwaves      = number of waves in each Riemann solution
3                  mthlim(mw)  = limiter for each wave  (mw=1,mwaves)

0.d0               t0          = initial time
0.0d0              xlower      = left edge of computational domain
1.0d0              xupper      = right edge of computational domain
0.0d0              ylower      = bottom edge of computational domain
1.0d0              yupper      = top edge of computational domain

2                  mbc         = number of ghost cells at each boundary
2                  mthbc(1)    = type of boundary conditions at left
2                  mthbc(2)    = type of boundary conditions at right
2                  mthbc(3)    = type of boundary conditions at bottom
2                  mthbc(4)    = type of boundary conditions at top
```

Figure 2.1: A sample `amr2ez.data` file, Part 1.

```
F               do a restart (input from restart.data)
1000            iousr        (how often to checkpoint in fort.chk****)

0.5             tol      (tolerance for Richardson estimator)
0.5             tolsp    (spatial tolerance for refinement)
2               kcheck   (how often to est. error - related to buffer size)
3               ibuff    (buffer zone size - add # of flagged pts to add)
.70             cutoff   (efficiency cutoff for grid generator)


F               PRINT option    (if T prints soln. to fort.amr)
F               NCAR graphics
T               Matlab graphics


F dprint -  verbose domain flags
F eprint -  error estimation - output the flags
F edebug -  even more error est. output
F gprint - grid generation output (bisection, clustering,...)
F nprint - proper nesting output
F pprint - projectiong of tagged pts. output
F rprint - verbose regridding - output new grid summary
F sprint - space (memory) output
T tprint - time step reporting - for each level
F uprint - updating/upbnding reporting
```

Figure 2.2: A sample amr2ez.data file, Part 2.

Bibliography

[1] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53:484–512, 1984.

[2] M. J. Berger. On conservation at grid interfaces. *SIAM J. Num. Anal.*, 24:967–984, 1987.

[3] M. J. Berger and R. J. LeVeque. Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems. *SIAM J. Numer. Anal.*, 35:2298–2316, 1998.

[4] M. J. Berger and I. Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Trans. Sys. Man & Cyber.*, 21:1278–1286, 1991.

[5] D. Calhoun and R. J. LeVeque. Solving the advection-diffusion equation in irregular geometries. submitted to J. Comput. Phys., 1998. (`ftp://amath.washington.edu/pub/rjl/papers/dc-rjl:advdiff.ps.gz`).

[6] T. Fogarty and R. J. LeVeque. High-resolution finite volume methods for acoustics in periodic or random media. *J. Acoust. Soc. Am.*, 1999. (`ftp://amath.washington.edu/pub/rjl/papers/trf-rjl:acou.ps.gz`).

[7] J. O. Langseth and R. J. LeVeque. A wave-propagation method for three-dimensional hyperbolic conservation laws. Submitted to J. Comput. Phys., `ftp://amath.washington.edu/pub/rjl/papers/jol-rjl:3d.ps.gz`, 1999.

[8] R. J. LeVeque. High-resolution conservative algorithms for advection in incompressible flow. *SIAM J. Numer. Anal.*, 33:627–665, 1996.

[9] R. J. LeVeque. Wave propagation algorithms for multi-dimensional hyperbolic systems. *J. Comput. Phys.*, 131:327–353, 1997.

[10] R. J. LeVeque. Balancing source terms and flux gradients in high-resolution Godunov methods. *J. Comput. Phys.*, 146:346–365, 1998.